

8

Optimizing Your Design

After you have selected your compile strategy and constrained your design, based on the selected strategy, you can optimize your design (or module). If your design references any design more than once, however, you must resolve these multiple instances before optimizing.

This chapter has the following sections:

- Resolving Multiple Instances
- Preserving Subdesigns
- Understanding the Compile Cost Function
- Understanding the Optimization Process
- Performing Design Exploration
- Performing Design Implementation

Resolving Multiple Instances

If your design references any design more than once, you must resolve these multiple instances before running the compile command. You can resolve multiple instances by using any of the following methods:

- The `uniquify` method.

This method uses the `uniquify` command to create a copy of the design for each instance.

- The `compile-once-don't-touch` method.

This method uses the `set_dont_touch` command to preserve the subdesign during optimization.

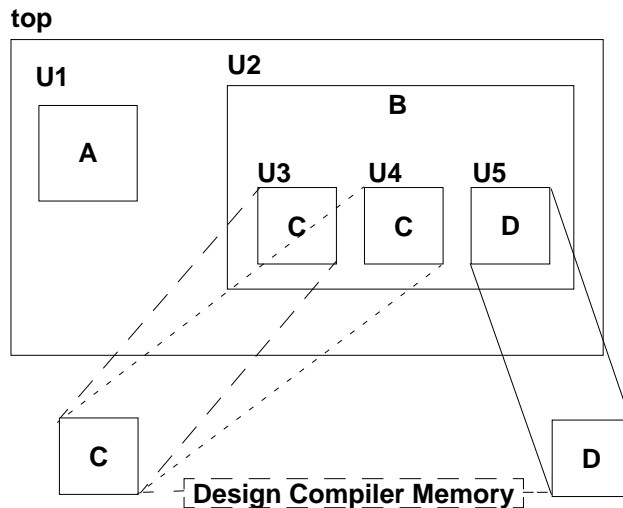
- The `ungroup` method.

This method uses the `ungroup` command to remove the hierarchy.

The following sections describe each method for resolving multiple instances of a design.

Figure 8-1 shows design top, which references design C twice (U2/U3 and U2/U4). This design is used to illustrate the various resolution methods.

Figure 8-1 Multiple Instances of a Design



Uniquify Method

If the environments around the design instances differ significantly, use the uniquify method to resolve multiple instances. This method involves using the `uniquify` command to copy and rename the design for each instance. Design Compiler optimizes each instance based on its unique environment.

By default, the `uniquify` command creates unique copies for all multiple instances in the current design (except those having a `dont_touch` attribute). You can create unique copies for specific references by using the `-reference` option, or you can specify specific cells to uniquify by using the `-cells` option. Design Compiler makes unique copies for cells specified with the `-reference` or `-cells` option, even if they have a `dont_touch` attribute.

Design Compiler uses the naming convention specified in the `uniquify_naming_style` variable to generate the name for each copy of the subdesign. The default naming convention is

`%s_%d`

`%s`

The original name of the subdesign (or the name specified in the `-base_name` option).

`%d`

The smallest integer value that forms a unique subdesign name.

To use the `uniquify` method to resolve multiple instances, follow these steps:

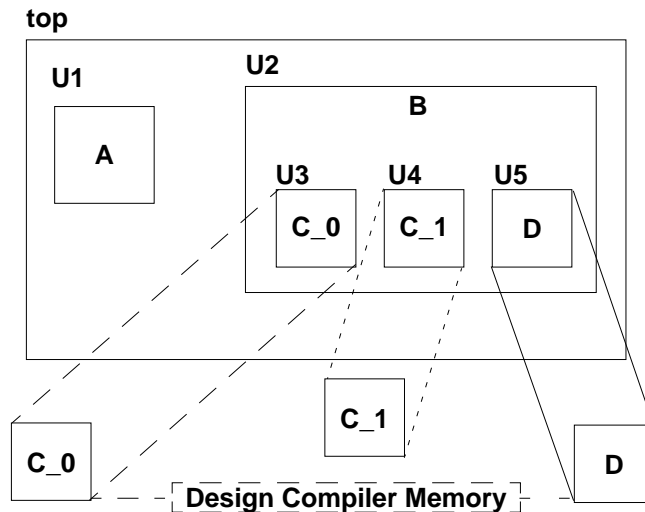
1. Uniquify the design.
2. Compile the design.

For example, the following command sequence resolves the multiple instances of design C in design top using the `uniquify` method:

```
dc_shell> current_design top  
dc_shell> uniquify  
dc_shell> compile
```

Figure 8-2 shows the result of running this command sequence.

Figure 8-2 Uniquify Results



Compared with the compile-once-don't-touch method, the uniquify method has the following characteristics:

- Requires more memory
- Takes longer to compile

Compile-Once-Don't-Touch Method

If similar environments surround the design instances, use the compile-once-don't-touch method. This method uses the `set_dont_touch` command to preserve the subdesign during optimization. See “Preserving Subdesigns” on page 8-9 for details about the `set_dont_touch` command.

To use the compile-once-don't-touch method to resolve multiple instances, follow these steps:

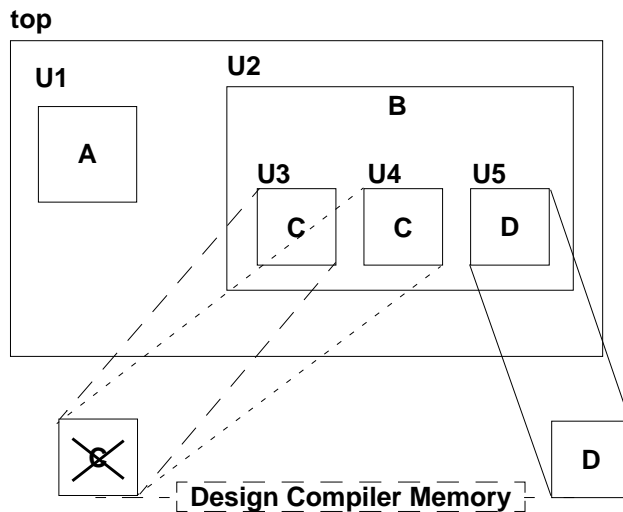
1. Characterize the design instance that has the worst-case environment.
2. Compile the reference design.
3. Use the `set_dont_touch` command to set the `dont_touch` attribute on the instances.
4. Compile the design.

For example, the following command sequence resolves the multiple instances of design C in design top, using the compile-once-don't-touch method (assuming U2/U3 has the worst-case environment):

```
dc_shell> current_design top  
dc_shell> characterize U2/U3  
dc_shell> current_design C  
dc_shell> compile  
dc_shell> current_design top  
dc_shell> set_dont_touch {U2/U3 U2/U4}  
dc_shell> compile
```

Figure 8-3 shows the result of running this command sequence.

Figure 8-3 Compile-Once-Don't-Touch Results



The compile-once-don't-touch method has the following advantages:

- Compiles the reference design once
- Requires less memory than the uniquify method
- Takes less time to compile than the uniquify method

The disadvantage of the compile-once-don't-touch method is that you cannot ungroup objects that have the `dont_touch` attribute.

Ungroup Method

The ungroup method has the same effect as the uniquify method (making unique copies of the design) but also removes levels of hierarchy. This method uses the `ungroup` command to produce a flattened netlist. See “Removing Levels of Hierarchy” in Chapter 6 for details about the `ungroup` command.

To use the ungroup method to resolve multiple instances, follow these steps:

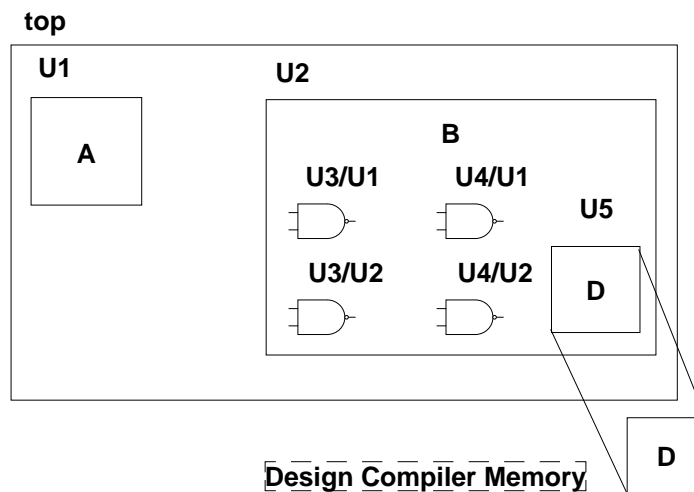
1. Ungroup the design.
2. Compile the design.

For example, the following command sequence resolves the multiple instances of design C in design top, using the ungroup method:

```
dc_shell> current_design B
dc_shell> ungroup {U3 U4}
dc_shell> current_design top
dc_shell> compile
```

Figure 8-4 shows the result of running this command sequence.

Figure 8-4 Ungroup Results



The ungroup method has the following characteristics:

- Requires more memory and takes longer to compile than the compile-once-don't-touch method
- Provides the best synthesis results

Preserving Subdesigns

The `set_dont_touch` command preserves a subdesign during optimization. It places the `dont_touch` attribute on cells, nets, references, and designs in the current design to prevent these objects from being modified or replaced during optimization.

Use the `set_dont_touch` command on subdesigns you do not want optimized with the rest of the design hierarchy. The `dont_touch` attribute does not prevent or disable timing through the design.

When you use `set_dont_touch`, remember the following:

- Setting `dont_touch` on a hierarchical cell sets an implicit `dont_touch` on all cells below it.
- Setting `dont_touch` on a library cell sets an implicit `dont_touch` on all instances of that cell.
- Setting `dont_touch` on a net sets an implicit `dont_touch` only on mapped combinational cells connected to that net. If the net is connected only to generic logic, optimization might remove the net.
- Setting `dont_touch` on a reference sets an implicit `dont_touch` on all cells using that reference during subsequent optimizations of the design.
- Setting `dont_touch` on a design has an effect only when the design is instantiated within another design as a level of hierarchy. In this case, the `dont_touch` on the design implies that all cells under that level of hierarchy are `dont_touch`. Setting `dont_touch` on the top-level design has no effect because the top-level design is not instantiated within any other design.

- You cannot ungroup objects marked as `dont_touch`.

Note:

The `dont_touch` attribute is ignored on synthetic part cells (for example, many of the cells read in from an HDL description) and on nets that have unmapped cells on them. During compilation, warnings appear for `dont_touch` nets connected to unmapped cells (generic logic).

Use the `report_design` command to determine whether a design has the `dont_touch` attribute set.

```
dc_shell> set_dont_touch SUB_A
Performing set_dont_touch on design 'SUB_A'.
dc_shell> report_design
```

```
*****
Report : design
Design : SUB_A
Version: 1999.05
Date   : Mon Jan 4 10:56:49 1999
*****
```

```
Design is dont_touched.
```

To remove the `dont_touch` attribute, use the `remove_attribute` or `set_dont_touch` command to false.

Understanding the Compile Cost Function

The compile cost function consists of design rule costs and optimization costs. By default, Design Compiler prioritizes costs in the following order:

1. Design rule costs

- a. Connection class
 - b. Multiple port nets
 - c. Maximum transition time
 - d. Maximum fanout
 - e. Maximum capacitance
 - f. Cell degradation
2. Optimization costs
- a. Maximum delay
 - b. Minimum delay
 - c. Maximum power
 - d. Maximum area
 - e. Minimum porosity

The compile cost function considers only those components that are active on your design. Design Compiler evaluates each cost function component independently, in order of importance.

When evaluating cost function components, Design Compiler considers only violators (positive difference between actual value and constraint) and works to reduce the cost function to 0.

Design Compiler tries to meet all constraints but, by default, gives emphasis to design rule constraints because design rule constraints are requirements for functional designs. Using the default priority, Design Compiler fixes design rule violations even at the cost of violating your delay or area constraints.

You can change the priority of the maximum design rule costs and the delay costs by using the `set_cost_priority` command to specify the ordering. You must run the `set_cost_priority` command before running the `compile` command.

You can disable evaluation of the design rule cost function by using the `-no_design_rule` option when running the `compile` command.

You can disable evaluation of the optimization cost function by using the `-only_design_rule` option when running the `compile` command.

Calculating Transition Time Cost

Design Compiler computes transition time for a driver by using the following equation:

$$D_T = \frac{R_{\text{driver}}(C_{\text{wire}} + C_{\text{pins}})}{\text{number_of_non_three_state_drivers}}$$

If the calculated transition time is greater than the `max_transition` value, Design Compiler reports a design rule violation and attempts to correct the violation.

Calculating Fanout Cost

Design Compiler computes fanout load for a driver by using the following equation:

$$\sum_{i=1}^m fanout_load_i$$

m is the number of inputs driven by the driver.

$fanout_load_i$ is the fanout load of the i th input.

If the calculated fanout load is greater than the `max_fanout` value, Design Compiler reports a design rule violation and attempts to correct the violation.

Calculating Capacitance Cost

Design Compiler computes the total capacitance for a driver by using the following equation:

$$\sum_{i=1}^m C_i$$

m is the number of inputs driven by the driver.

C_i is the capacitance of the i th input.

If the calculated capacitance is greater than the `max_capacitance` value, Design Compiler reports a design rule violation and attempts to correct the violation.

Calculating Cell Degradation Cost

The cell degradation tables in the technology library provide a secondary maximum capacitance constraint, based on the transition times at the cell inputs. Design Compiler evaluates this cost only if you set the `compile_fix_cell_degradation` variable to true.

If the `compile_fix_cell_degradation` variable is true and the calculated capacitance is greater than the `cell_degradation` value, Design Compiler reports a design rule violation and attempts to correct the violation. The maximum capacitance cost has a higher priority than the cell degradation cost. Therefore, Design Compiler fixes cell degradation violations only if it can do so without violating the maximum capacitance constraint.

Calculating Maximum Delay Cost

Design Compiler supports two methods for calculating the maximum delay cost:

- Worst negative slack (default behavior)
- Critical negative slack

The following sections describe these methods.

Worst Negative Slack Method

By default, Design Compiler uses the worst negative slack method to calculate the maximum delay cost. The worst negative slack method considers only the worst violator in each path group.

A path group is a collection of paths that Design Compiler considers as a group in maximum delay cost calculations. Each time you create a clock with the `create_clock` command, Design Compiler creates a path group that contains all the paths associated with the clock. You can also create path groups by using the `group_path` command (see “Creating Path Groups” on page 8-27 for information about the `group_path` command). Design Compiler places in the default group any paths that are not associated with any particular group or clock. To see the path groups defined for your design, run the `report_path_group` command.

Because the worst negative slack method does not optimize near-critical paths, this method requires fewer CPU resources than the critical negative slack method. Because of the shorter runtimes, the worst negative slack method is ideal for the exploration phase of the design. Always use the worst negative slack method during default compile runs.

Using the worst negative slack method, the equation for the maximum delay cost is

$$\sum_{i=1}^m v_i \times w_i$$

m is the number of path groups.

v_i is the worst violator in the i th path group.

w_i is the weight assigned to the i th path group.

Design Compiler calculates the maximum delay violation for each path group as

```
max (0, (actual_path_delay - max_delay))
```

Because only the worst violator in each path group contributes to the maximum delay violation, how you group paths affects the maximum delay cost calculation.

- If only one path group exists, the maximum delay cost is the amount of the worst violation multiplied by the group weight.
- When multiple path groups exist, the costs for all the groups are added to determine the maximum delay cost of the design.

During optimization, Design Compiler concentrates on reducing the delay of the most critical path. This path changes during optimization. If Design Compiler minimizes the initial path's delay so that it is no longer the worst violator, Design Compiler changes to the path that is now the most critical path in the group.

Critical Negative Slack Method

Design Compiler also supports the critical negative slack method to calculate the maximum delay cost. The critical negative slack method considers all violators in each path group that are within a specified delay margin (referred to as the critical range) of the worst violator.

For example, if the critical range is 2.0 ns and the worst violator has a delay of 10.0 ns, Design Compiler optimizes all paths that have a delay between 8.0 and 10.0 ns.

Note:

When the critical range includes all violators, the critical negative slack equals the total negative slack.

See “Creating Path Groups” on page 8-27 for information about specifying the critical range.

Using the critical negative slack method, the equation for the maximum delay cost is

$$\sum_{i=1}^m \left(\left\langle \sum_{j=1}^n v_{ij} \right\rangle \times w_i \right)$$

m is the number of path groups.

n is the number of paths in the critical range in the path group.

v_{ij} is a violator within the critical range of the i th path group.

w_i is the weight assigned to the i th path group.

Design Compiler calculates the maximum delay violation for each path within the critical range as

$$\max (0, (\text{actual_path_delay} - \text{max_delay}))$$

Calculating Minimum Delay Cost

The equation for the minimum delay cost is

$$\sum_{i=1}^m v_i$$

m is the number of paths affected by `set_min_delay` or `set_fix_hold`.

v_i is the i th minimum delay violation.

Design Compiler calculates the minimum delay violation for each path as

```
max (0, (min_delay - actual_path_delay))
```

The minimum delay cost for a design differs from the maximum delay cost. Path groups do not affect the minimum delay cost. In addition, all violators, not just the most critical path, contribute to the minimum delay cost.

Calculating Maximum Power Cost

Design Compiler computes the maximum power cost only if you have a Power-Optimization license and your technology library is characterized for power.

The maximum power cost has two components:

- Maximum dynamic power

Design Compiler calculates the maximum dynamic power cost as

```
max (0, actual_power - max_dynamic_power)
```

- Maximum leakage power

Design Compiler calculates the maximum leakage power cost as

```
max (0, actual_power - max_leakage_power)
```

For more information about the maximum power cost, see the *Power Products Reference Manual*.

Calculating Maximum Area Cost

Design Compiler computes the area of a design by summing the areas of each of its components (cells) on the design hierarchy's lowest level (and the area of the nets). Design Compiler ignores the following components when calculating circuit area:

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

The cell and net areas are technology-dependent. Design Compiler obtains this information from the technology library.

Design Compiler calculates the maximum area cost as

```
max (0, actual_area - max_area)
```

Calculating Minimum Porosity Cost

Design Compiler computes the porosity of a design by dividing the sum of the routing track area of each of its components on the design hierarchy's lowest level by the sum of all component areas. Design Compiler ignores the following components when calculating porosity:

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

The routing track area of a cell and the cell area are technology-dependent. Design Compiler obtains this information from the technology library.

Design Compiler calculates the minimum porosity cost as

```
max (0, min_porosity - actual_porosity)
```

Understanding the Optimization Process

During optimization, Design Compiler performs the following tasks:

- Architectural optimization
- Logic-level optimization
- Gate-level optimization

The following sections describe these tasks.

Architectural Optimization

Architectural optimization works on the HDL description. It includes high-level synthesis tasks, such as

- Sharing common subexpressions
- Sharing resources
- Selecting DesignWare implementations
- Reordering operators

With the exception of selecting DesignWare implementations, these high-level synthesis tasks do not occur when you reoptimize a gate-level netlist.

These tasks are based on your constraints and your coding style. For more information about how your coding style affects architectural optimization, see Chapter 3, “Working With Design Files,” in either the *HDL Compiler for Verilog Reference Manual* or in the *VHDL Compiler Reference Manual*.

Logic-Level Optimization

Logic-level optimization works on a generic technology-independent netlist. It includes the following processes:

Structuring

Adds intermediate variables and logic structure to a design. During structuring, Design Compiler searches for subfunctions that can be factored out and evaluates these factors, based on the size of the factor and the number of times the factor appears in the design. Design Compiler turns the subfunctions that most reduce the logic into intermediate variables and factors them out of the design equations.

By default, Design Compiler structures your design. Use the `set_structure` command and the `compile_new_boolean_structure` variable to control structuring of your design. The `set_structure` command and its options set the following attributes: `structure`, `structure_boolean`, and `structure_timing`.

Flattening

Attempts to convert the design to a two-level sum-of-products representation. During flattening, Design Compiler removes all intermediate variables, and therefore all 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, however, because it requires a large amount of CPU time and can increase area.

By default, Design Compiler does not flatten your design. Use the `set_flatten` command to control flattening of your design. The `set_flatten` command and its options set the following attributes: `flatten`, `flatten_effort`, `flatten_minimize`, and `flatten_phase`.

The structuring and flattening attributes enable fine-tuning of the optimization techniques used for each design in the hierarchy. Table 8-1 shows the default values for these attributes.

Table 8-1 Structuring and Flattening Attributes

Attribute	Default Setting
<code>structure</code>	<code>true</code>
<code>structure_boolean</code>	<code>false</code>
<code>structure_timing</code>	<code>true</code>
<code>flatten</code>	<code>false</code>

Set these attributes on a design-by-design basis; do not set them globally. If you specify both flattening and structuring, Design Compiler first performs flattening, then performs structuring.

Use the `report_compile_options` command to display these attributes for the current design.

Gate-Level Optimization

Gate-level optimization works on the technology-specific netlist. It includes the following processes:

Mapping

Uses gates (combinational and sequential) from the target technology libraries to generate a gate-level implementation that meets area and timing goals.

You control the mapping algorithms used by Design Compiler with the various options of the `compile` command.

Design rule fixing

Inserts buffers or resizes existing cells to correct any design rule violations present in the design after mapping.

By default, Design Compiler performs design rule fixing, even at the expense of violating optimization constraints. You can change the priority of design rule fixing by using the `set_cost_priority` command. You can disable design rule fixing by specifying the `-no_design_rule` option when you run the `compile` command.

Performing Design Exploration

Design exploration uses the default synthesis algorithm to gauge the design performance against your goals. To invoke the default synthesis algorithm, use the `compile` command with no options:

```
dc_shell> compile
```

The default compile uses the `-map_effort medium` option of the compile command and the default settings of the structuring and flattening attributes.

If the performance violates the timing goals by more than 10 percent, you must modify the HDL code or refine the design budget.

Performing Design Implementation

The default compile generates good results for about 60 percent of designs. If your design meets the optimization goals after design exploration, you are done. If not, try the techniques described in the following sections:

- Optimizing Random Logic
- Optimizing Structured Logic
- Optimizing for Maximum Performance
- Optimizing for Minimum Area

Optimizing Random Logic

If the default compile does not give the desired result for your random logic design, try the following techniques. If the first technique does not give the desired results, use the second technique, and so on, until you obtain the desired results.

- Flatten the design before structuring. Enter

```
dc_shell> set_flatten true
dc_shell> set_structure true
dc_shell> compile
```

When you run this command sequence, Design Compiler first flattens the logic, then goes back and restructures the design by sharing logic off the critical path.

- Increase the flattening effort. Enter

```
dc_shell> set_flatten true -effort medium
dc_shell> compile
```

- Fine-tune the results with minimization or phase inversion. Enter

```
dc_shell> set_flatten true \
           -minimize multiple_output -phase true
dc_shell> compile
```

The `set_flatten -minimize` command causes Design Compiler to share product terms between output logic cones (minimization). Minimization causes higher fanout but does not change the two-level sum-of-products representation.

If you select the `-minimize single_output` option, Design Compiler minimizes the equations for each output individually. The `-minimize multiple_output` option enables minimization of the entire design by allowing optimization to share terms among outputs. Minimization increases compile time; therefore, Design Compiler does not perform minimization during default flattening.

The `set_flatten -phase true` command inverts the polarity of the outputs, compares the original implementation with the complement, and keeps the best result. Setting the `-phase` option to true increases compile time; therefore, the default value for the `-phase` option is false.

Optimizing Structured Logic

If the default compile does not give the desired result for your structured logic design, try the following techniques. If the first technique does not give the desired results, try the second one.

- Map the design with no flattening or structuring. Enter

```
dc_shell> set_structure false
dc_shell> compile
```

- Flatten with structuring. Enter

```
dc_shell> set_flatten true
dc_shell> set_structure true
dc_shell> compile
```

When you run this command sequence, Design Compiler first flattens the logic, then goes back and restructures the design by sharing logic off the critical path.

Optimizing for Maximum Performance

This section provides techniques for improving the performance of your design. It covers the following topics:

- Creating path groups
- Fixing heavily loaded nets
- Flattening logic on the critical path
- Performing a high-effort incremental compile
- Performing a high-effort compile

Creating Path Groups

By default, Design Compiler groups paths based on the clock controlling the endpoint (all paths not associated with a clock are in the default path group). If your design has complex clocking, complex timing requirements, or complex constraints, you can create path groups to focus Design Compiler on specific critical paths in your design.

Use the `group_path` command to create path groups. The `group_path` command allows you to

- Control the optimization of your design
- Optimize near-critical paths
- Optimize all paths

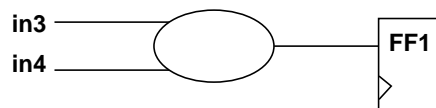
Controlling the Optimization of Your Design

You can control the optimization of your design by creating and prioritizing path groups, which affect only the maximum delay cost function. By default, Design Compiler works only on the worst violator in each group.

Set the path group priorities by assigning weights to each group (default weight = 1.0). The weight can be from 0.0 to 100.0.

For example, Figure 8-5 shows a design that has multiple paths to flip-flop FF1.

Figure 8-5 Path Group Example



To indicate that the path from input in3 to FF1 is the highest-priority path, use the following command to create a high-priority path group:

```
dc_shell> group_path -name group3 \  
           -from in3 -to FF1/D -weight 2.5
```

Optimizing Near-Critical Paths

When you add a critical range to a path group, you change the maximum delay cost function from worst negative slack to critical negative slack. Design Compiler optimizes all paths within the critical range.

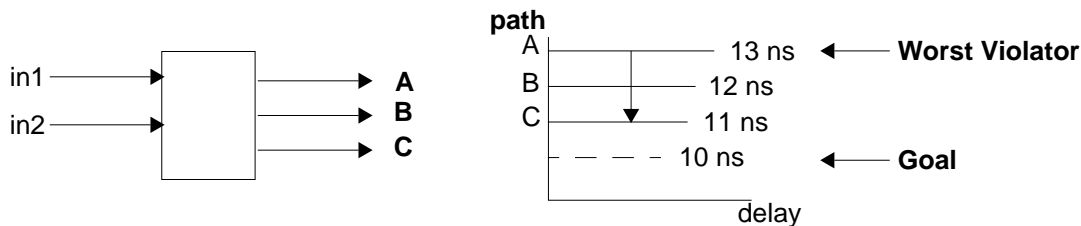
Specifying a critical range can increase runtime. To limit the runtime increase, use critical range only during the final implementation phase of the design and use a reasonable critical range value. A guideline for the maximum critical range value is 10 percent of the clock period.

Use one of the following methods to specify the critical range:

- Use the `-critical_range` option of the `group_path` command.
- Use the `set_critical_range` command.

For example, Figure 8-6 shows a design with three outputs, A, B, and C.

Figure 8-6 Critical Range Example



Assume that the clock period is 20 ns, the maximum delay on each of these outputs is 10 ns, and the path delays are as shown. By default, Design Compiler optimizes only the worst violator (the path to output A). To optimize all paths, set the critical delay to 3.0 ns.

```
dc_shell> create_clock -period 20 clk
dc_shell> set_critical_range 3.0 current_design
dc_shell> set_max_delay 10 {A B C}
dc_shell> group_path -name group1 -to {A B C}
```

Optimizing All Paths

You can optimize all paths by creating a path group for each endpoint in the design. Creating a path group for each endpoint enables total negative slack optimization but results in long compile runtimes.

Use the following script to create a path group for each endpoint:

```
endpoints = all_outputs() + all_registers(-data_pins)
foreach (endpt, endpoints) {
    group_path -name endpt -to endpt
}
```

Fixing Heavily Loaded Nets

Heavily loaded nets often result in critical paths. To reduce the load on a net, you can use either of two approaches:

1. If the large load resides in a single module and the module contains no hierarchy, fix the heavily loaded net by using the `balance_buffer` command. For example, enter

```
dc_shell> include constraints.con
dc_shell> compile
dc_shell> balance_buffer -from find(pin, buf1/Z)
```

Note:

The `balance_buffers` command provides the best results when your library uses linear delay models. If your library uses nonlinear delay models, the second approach provides better results.

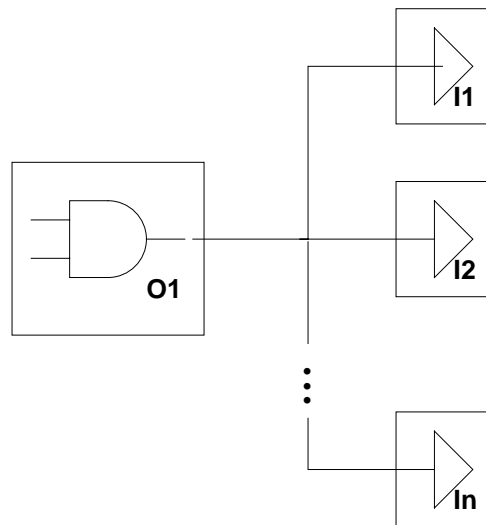
2. If the large loads reside across the hierarchy from several modules, apply design rules to fix the problem. For example, enter

```
dc_shell> include constraints.con
dc_shell> compile
dc_shell> set_max_capacitance 3.0
dc_shell> compile -only_design_rule
```

In rare cases, hierarchical structure might disable Design Compiler from fixing design rules.

In the sample design shown in Figure 8-7, net O1 is overloaded. To reduce the load, group as many of the loads (I1 through In) as possible in one level of hierarchy by using the `group` command or by changing the HDL. Then you can apply one of the strategies.

Figure 8-7 Heavily Loaded Net



Flattening Logic on the Critical Path

Flattening improves performance by representing the design as a two-level sum of products. However, flattening requires considerable CPU resources and it might not be possible to flatten the entire design. In this case, you can improve the performance by flattening just the logic on the critical path.

To flatten the logic on the critical path,

1. Identify the logic associated with the critical path, using the `all_fanin` command.

```
dc_shell> all_fanin -to all_critical_pins() -only_cells
dc_shell> cp_logic = dc_shell_status
```

2. Group the critical path logic.

```
dc_shell> group -design critical_block -cell_name cp1 \
              cp_logic
```

3. Characterize the critical path logic.

```
dc_shell> characterize cp1
```

4. Flatten the critical path logic.

```
dc_shell> current_design critical_block
dc_shell> set_flatten true
dc_shell> compile
dc_shell> set_flatten false
```

5. Ungroup the block of critical path logic.

```
dc_shell> current_design ..
dc_shell> ungroup -simple_names cp1
```

Performing a High-Effort Incremental Compile

If none of the previous strategies results in a design that meets your optimization goals, try a high-effort incremental compile.

A high-effort compile (`-map_effort high` compile option) pushes Design Compiler to the extreme to achieve the design goal. If you have a DC-Expert license, a high-effort compile invokes the critical-path resynthesis strategy to restructure and remap the logic on and around the critical path.

An incremental compile (`-incremental_mapping` compile option) allows you to incrementally improve your design by experimenting with different approaches. An incremental compile performs only gate-level optimization and does not perform logic-level optimization. The resulting design's performance is the same or better than the original design's.

This technique requires large amounts of CPU time but is the most successful at reducing the worst negative slack to 0. To reduce runtime, you can place a `dont_touch` attribute on all blocks that already meet timing constraints.

```
dc_shell> dont_touch noncritical_blocks
dc_shell> compile -map_effort high -incremental_mapping
```

This incremental approach works best for a technology library that has many variations of each logic cell.

Performing a High-Effort Compile

The optimization result depends on the starting point. Occasionally, the starting point generated by the default compile results in a local minimum solution, and Design Compiler quits before generating an optimal design. A high-effort compile might solve this problem.

The high-effort compile uses the `-map_effort high` option of the `compile` command on the initial compile (on the HDL description of the design).

```
dc_shell> elaborate my_design
dc_shell> compile -map_effort high
```

This compile strategy is CPU-intensive; use this approach only if you cannot meet your timing goals with an incremental approach.

Optimizing for Minimum Area

If your design has timing constraints, these constraints always take precedence over area requirements. For area-critical designs, do not apply timing constraints before you compile. If you want to view timing reports, you can apply timing constraints to the design after you compile.

If your design does not meet the area constraints, you can try the following methods to reduce the area:

- Disable total negative slack optimization
- Enable sequential area recovery
- Enable Boolean optimization
- Manage resource selection

- Use flattening
- Optimize across hierarchical boundaries

Disabling Total Negative Slack Optimization

By default, Design Compiler prioritizes total negative slack over meeting area constraints. This means Design Compiler performs area optimization only on those paths that have positive slack.

To change the default priorities (prioritize area over total negative slack), use the `-ignore_tns` option when setting the area constraints.

```
dc_shell> set_max_area -ignore_tns max_area
```

Enabling Sequential Area Recovery

By default, Design Compiler does not remap sequential elements during optimization. You might be able to reduce area by remapping the sequential elements that are not on the critical path. To enable this capability, set the `compile_sequential_area_recovery` variable to true. You must set this variable before you compile.

```
dc_shell> compile_sequential_area_recovery = true
```

Enabling Boolean Optimization

Boolean optimization uses algorithms based on the basic rules of Boolean algebra. Boolean optimization can use don't care conditions to minimize area. This algorithm performs area optimization only; do not use Boolean optimization for timing-critical designs.

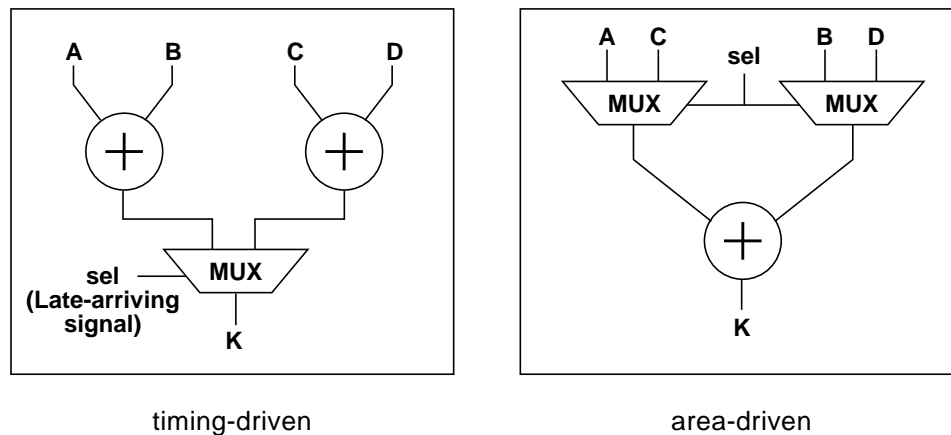
Use the `compile_new_boolean_structure` variable and the `-boolean true` option of the `set_structure` command to enable Boolean optimization. You must run these commands before you compile.

```
dc_shell> compile_new_boolean_structure = true
dc_shell> set_structure true -boolean true \
          -boolean_effort medium
```

Managing Resource Selection

The decisions made during resource sharing can also significantly affect area. Figure 8-8 shows that high-level optimization can allocate the arithmetic operators in the same HDL code in two very different ways.

Figure 8-8 Resource Sharing Possibilities



The operator implementation also affects area. For example, in Figure 8-8, the timing-driven version implements the adders as carry-lookahead adders, and the area-driven example uses a ripple adder implementation.

By default, high-level optimization performs resource allocation and implementation based on timing constraints. To change the default and force Design Compiler to base resource allocation and implementation on area constraints, set the following variables before compile:

```
dc_shell> set_resource_allocation area_only
dc_shell> set_resource_implementation area_only
```

To specify area-driven resource allocation and implementation for a specific design, set the following variables before you compile:

```
dc_shell> current_design subdesign
dc_shell> set_resource_allocation area_only
dc_shell> set_resource_implementation area_only
```

Using Flattening

In most cases, flattening increases the area. In highly random designs with unpredictable structures, flattening might reduce the area. However, flattening is CPU-intensive, and the process might not finish for some designs.

Use the `set_flatten` command on specific modules that might benefit from this technique; do not use the `set_flatten` command on the top-level design.

The `-minimize` and `-phase` options discussed in “Optimizing Random Logic” on page 8-24 can also reduce area.

Optimizing Across Hierarchical Boundaries

Design Compiler always respects levels of hierarchy and port functionality. Boundary optimizations, such as constant propagation through a subdesign, do not occur automatically.

To fine-tune the area, you can leave the hierarchy intact and enable boundary optimization. For greater area reduction, you might have to remove hierarchical boundaries.

Boundary Optimization

Direct Design Compiler to perform optimization across hierarchical boundaries (boundary optimization) by using one of the following commands:

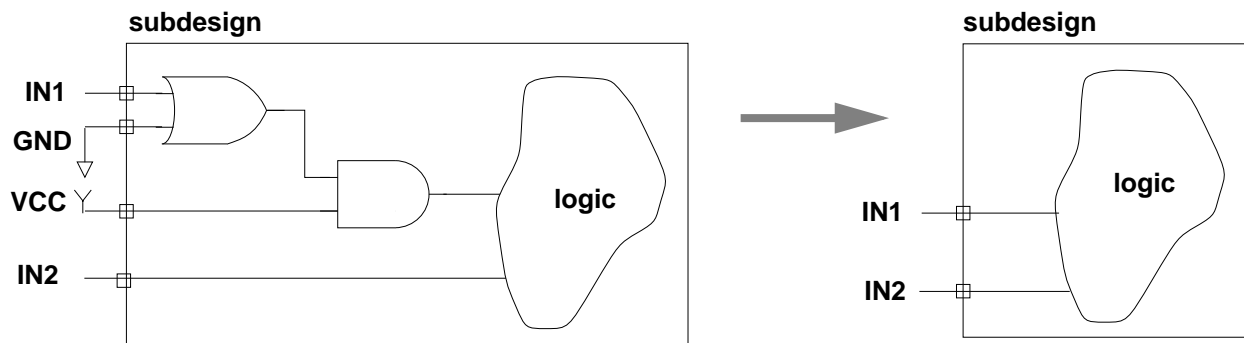
```
dc_shell> compile -boundary_optimization
```

or

```
dc_shell> set_boundary_optimization subdesign
```

If you enable boundary optimization, Design Compiler propagates constants, unconnected pins, and complement information. In designs that have many constants (VCC and GND) connected to the inputs of subdesigns, propagation can reduce area. Figure 8-9 shows this relationship.

Figure 8-9 Benefits of Boundary Optimization



Hierarchy Removal

Removing levels of hierarchy by ungrouping gives Design Compiler more freedom to share common terms across the entire design. Ungrouping DesignWare parts can also reduce area. See “Removing Levels of Hierarchy” in Chapter 6 for details about this task.