

7

Preparing for Optimization

This chapter contains the following sections:

- Defining the Design Environment
- Selecting a Compile Strategy
- Setting Design Rule Constraints
- Setting Optimization Constraints
- Analyzing the Precompiled Design

Perform the tasks described in these sections before invoking the optimization process.

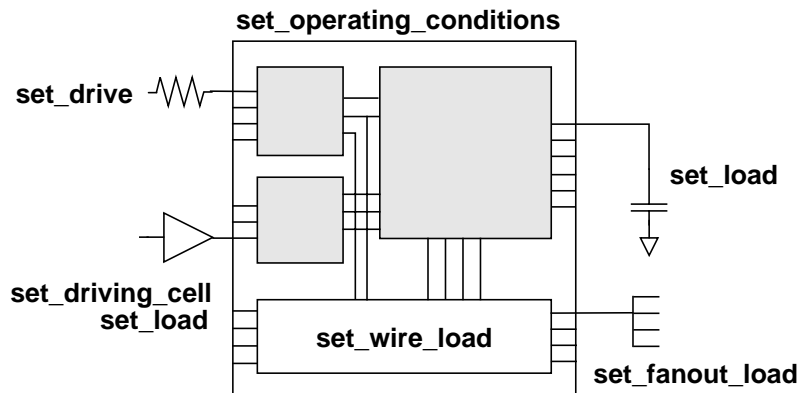
Defining the Design Environment

The design environment is a set of attributes and constraints that model the environment surrounding the design being synthesized. The design environment includes the following items:

- The operating conditions
- The wire load models
- The system interface

Figure 7-1 illustrates the commands used to define the design environment.

Figure 7-1 Commands Used to Define the Design Environment



Defining the Operating Conditions

In most technologies, variations in operating temperature, supply voltage, and manufacturing process can strongly affect circuit performance (speed). These factors are called operating conditions.

Operating temperature variation

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

Supply voltage variation

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

Process variation

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

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

Determining Available Operating Condition Options

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

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

```
dc_shell> read my_lib.db
dc_shell> report_lib my_lib
```

Example 7-1 shows the resulting operating conditions report.

Example 7-1 Operating Conditions Report

```
*****
Report : library
Library: my_lib
Version: 1999.05
Date   : Mon Jan 4 10:56:49 1999
*****
...
Operating Conditions:
```

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

```
...
```

Specifying Operating Conditions

There are no default operating conditions. You must explicitly specify the operating conditions for the current design by using the `set_operating_conditions` command.

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

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

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

Defining Wire Load Models

Wire load modeling estimates the effect of wire length and fanout on resistance, capacitance, and area. These factors are used to estimate wire delays and are important for the overall speed of the circuit.

The wire load models define a fanout-to-length relationship. Semiconductor vendors develop the wire load models, based on statistical information specific to the vendors' process. In the absence of back-annotated wire delays, Design Compiler uses these wire load models to estimate wire lengths in a design. See the Library Compiler documentation for more information about developing wire load models.

Design Compiler determines which wire load model to use for a design, based on the following factors, listed in order of precedence:

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

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

In hierarchical designs, Design Compiler must also determine which wire load model to use for nets that cross hierarchical boundaries. Design Compiler determines the wire load model for cross-hierarchy nets based on one of the following factors, listed in order of precedence:

1. Explicit user specification
2. Default specification in the technology library
3. Default mode in Design Compiler

The following sections discuss the selection of wire load models for nets and designs.

Understanding Hierarchical Wire Load Models

Design Compiler supports three modes for determining which wire load model to use for nets that cross hierarchical boundaries:

top

Design Compiler models nets as if the design has no hierarchy and uses the wire load model specified for the top level of the design hierarchy for all nets in a design and its subdesigns. Design Compiler ignores any wire load models set on subdesigns with the `set_wire_load` command.

Use top mode if you plan to flatten the design at a higher level of hierarchy before layout.

enclosed

Design Compiler uses the wire load model of the smallest design that fully encloses the net. If the design enclosing the net has no wire load model, Design Compiler traverses the design hierarchy upward until it finds a wire load model. Enclosed mode is more accurate than top mode when cells in the same design are placed in a contiguous region during layout.

Use enclosed mode if the design has similar logical and physical hierarchies.

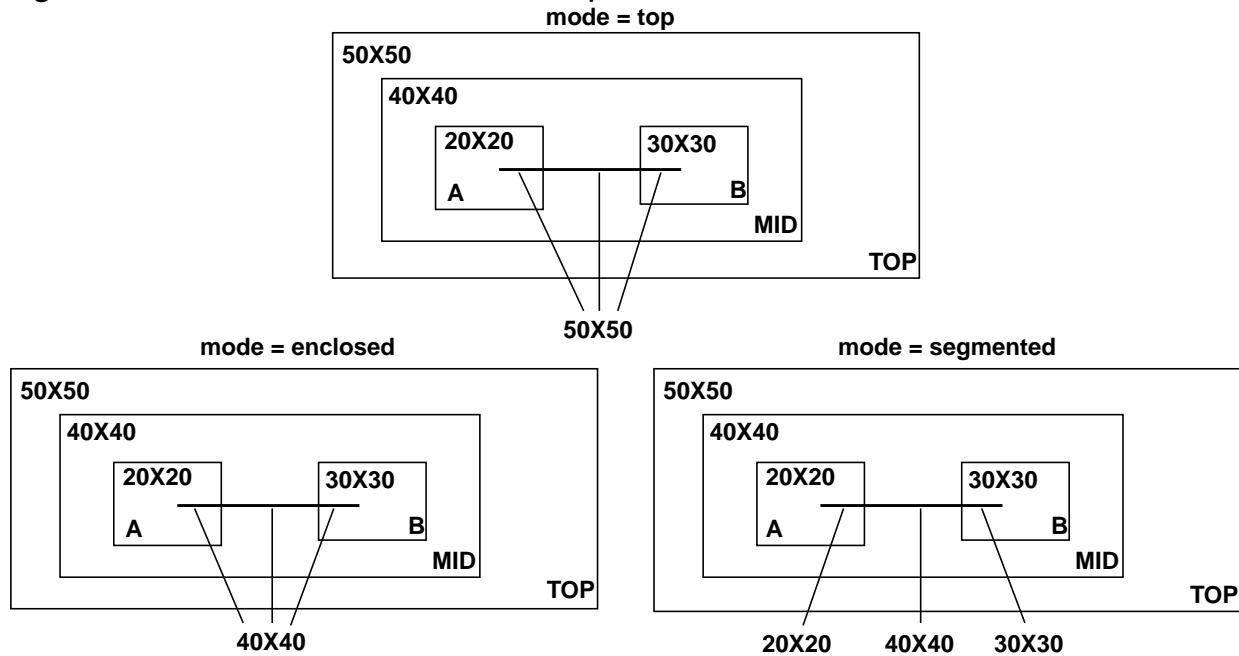
segmented

Design Compiler determines the wire load model of each segment of a net by the design encompassing the segment. Nets crossing hierarchical boundaries are divided into segments. For each net segment, Design Compiler uses the wire load model of the design containing the segment. If the design containing a segment has no wire load model, Design Compiler traverses the design hierarchy upward until it finds a wire load model.

Use segmented mode if the wire load models in your technology have been characterized with net segments.

Figure 7-2 shows a sample design with a cross-hierarchy net, `cross_net`. The top level of the hierarchy (design TOP) has a wire load model of 50x50. The next level of hierarchy (design MID) has a wire load model of 40x40. The leaf-level designs, A and B, have wire load models of 20x20 and 30x30, respectively.

Figure 7-2 Wire Load Mode Example



In top mode, Design Compiler estimates the wire length of net `cross_net`, using the 50x50 wire load model. Design Compiler ignores the wire load models on designs MID, A, and B.

In enclosed mode, Design Compiler estimates the wire length of net `cross_net`, using the 40x40 wire load model (the net `cross_net` is completely enclosed by design MID).

In segmented mode, Design Compiler uses the 20x20 wire load model for the net segment enclosed in design A, the 30x30 wire load model for the net segment enclosed in design B, and the 40x40 wire load model for the segment enclosed in design MID.

Determining Available Wire Load Models

Most technology libraries have predefined wire load models. Use the `report_lib` command to list the wire load models defined in a technology library. The library must be loaded in memory before you run the `report_lib` command. To see a list of libraries loaded in memory, use the `list_libs` command.

The wire load report contains the following sections:

- Wire Loading Model section

This section lists the available wire load models.

- Wire Loading Model Mode section

This section identifies the default wire load mode. If a library default does not exist, Design Compiler uses top mode.

- Wire Loading Model Selection section

The presence of this section indicates that the library supports automatic area-based wire load model selection.

To generate a wire load report for the `my_lib` library, enter

```
dc_shell> read my_lib.db
dc_shell> report_lib my_lib
```

Example 7-2 shows the resulting wire load models report. The library `my_lib` contains three wire load models: 05x05, 10x10, and 20x20. The library does not specify a default wire load mode (so Design Compiler uses top as the default wire load mode) and supports automatic area-based wire load model selection.

Example 7-2 Wire Load Models Report

```

*****
Report : library
Library: my_lib
Version: 1999.05
Date   : Mon Jan 4 10:56:49 1999
*****
...
Wire Loading Model:

Name       : 05x05
Location   : my_lib
Resistance  : 0
Capacitance : 1
Area       : 0
Slope      : 0.186
Fanout     Length  Points Average Cap Std Deviation
-----
1         0.39

Name       : 10x10
Location   : my_lib
Resistance  : 0
Capacitance : 1
Area       : 0
Slope      : 0.311
Fanout     Length  Points Average Cap Std Deviation
-----
1         0.53

Name       : 20x20
Location   : my_lib
Resistance  : 0
Capacitance : 1
Area       : 0
Slope      : 0.547
Fanout     Length  Points Average Cap Std Deviation
-----
1         0.86

Wire Loading Model Selection Group:

Name       : my_lib

      Selection      Wire load name
      min area  max area
-----

```

0.00	1000.00	05x05
1000.00	2000.00	10x10
2000.00	3000.00	20x20

...

Specifying Wire Load Models and Modes

The technology library can define a default wire load model that is used for all designs implemented in that technology. The `default_wire_load` library attribute identifies the default wire load model for a technology library.

Some libraries support automatic area-based wire load selection. Design Compiler uses the library function `wire_load_selection` to choose a wire load model based on the total cell area.

For large designs with many levels of hierarchy, automatic wire load selection can increase runtime. To manage runtime, set the wire load manually.

You can turn off automatic selection of the wire load model by setting the `auto_wire_load_selection` attribute to `false`. For example, enter

```
dc_shell> auto_wire_load_selection = false
```

The technology library can also define a default wire load mode. The `default_wire_load_mode` library attribute identifies the default mode. If the current library does not define a default mode, Design Compiler looks in the libraries specified in the `link_library` variable. (To see the link library, use the `list` command.) In the absence of a library default (and explicit specification), Design Compiler assumes that top mode is being used.

To change the wire load model or mode specified in a technology library, use the `set_wire_load` command. Select the wire load mode by using the `-mode` option. The wire load model and mode you define override all defaults. Explicitly selecting a wire load model also disables area-based wire load model selection for that design.

For example, to select the 10x10 wire load model, enter

```
dc_shell> set_wire_load "10x10"
```

To select the 10x10 wire load model and specify enclosed mode, enter

```
dc_shell> set_wire_load -mode enclosed "10x10"
```

The wire load model you choose for a design depends on how that design is implemented in the chip. Consult your semiconductor vendor to determine the best wire load model for your design.

Use the `report_timing` command to see the wire load model and mode defined for the current design.

To remove the wire load model, use the `set_wire_load` command with no model name.

Modeling the System Interface

Design Compiler supports the following ways to model the design's interaction with the external system:

- Defining drive characteristics for input ports
- Defining loads on input and output ports
- Defining fanout loads on output ports

The following sections discuss these tasks.

Defining Drive Characteristics for Input Ports

Design Compiler uses drive strength information to appropriately buffer nets in the case of a weak driver.

By default, Design Compiler assumes zero drive resistance on input ports, meaning infinite drive strength. Design Compiler provides two commands for overriding this unrealistic assumption:

- `set_driving_cell`
- `set_drive`

Note:

For heavily loaded driving ports, such as clock lines, keep the drive strength setting at 0 so that Design Compiler does not buffer the net. Each semiconductor vendor has a different way of distributing these signals within the silicon.

The two commands are similar, except for the command argument.

Both the `set_drive` and the `set_driving_cell` commands affect only the port transition delay. They do not place design rule requirements, such as `max_fanout` or `max_transition`, on the input port.

The most recently used command takes precedence. For example, setting a drive resistance on a port with the `set_drive` command overrides previously run `set_driving_cell` commands and vice versa.

set_driving_cell Command

Use the `set_driving_cell` command to specify drive characteristics on ports that are driven by cells in the technology library. This command is compatible with all the delay models, including the nonlinear delay model and piecewise linear delay model. The `set_driving_cell` command associates a library pin with an input port so that delay calculators can accurately model the drive capability of an external driver.

To return to the default of zero drive resistance, use the `set_driving_cell -none` command.

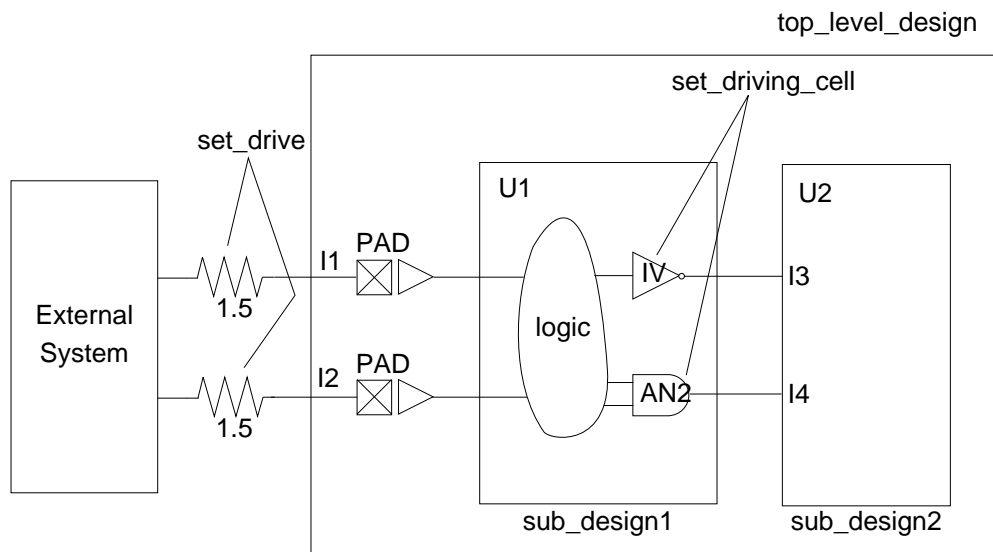
set_drive Command

Use the `set_drive` command to set the drive resistance on the top-level ports of the design when the input port drive capability cannot be characterized with a cell in the technology library.

You can use `set_drive` and the `drive_of` commands together to represent the drive resistance of a cell. However, these commands are not as accurate for nonlinear delay models as the `set_driving_cell` command is.

Figure 7-3 shows a hierarchical design. The top-level design has two subdesigns, U1 and U2. Ports I1 and I2 of the top-level design are driven by the external system and have a drive resistance of 1.5.

Figure 7-3 Drive Characteristics



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

1. Because ports I1 and I2 are not driven by library cells, use the `set_drive` command to define the drive resistance. Enter

```
dc_shell> current_design top_level_design
dc_shell> set_drive 1.5 {I1 I2}
```

2. To describe the drive capability for the ports on design `sub_design2`, change the current design to `sub_design2`. Enter

```
dc_shell> current_design sub_design2
```

3. An IV cell drives port I3. Use the `set_driving_cell` command to define the drive resistance. Because IV has only one output and one input, define the drive capability as follows. Enter

```
dc_shell> set_driving_cell -cell IV {I3}
```

4. An AN2 cell drives port I4. Because the different arcs of this cell have different transition times, select the worst-case arc to define the drive. For checking setup violations, the worst-case arc is the slowest arc. For checking hold violations, the worst-case arc is the fastest arc.

For this example, assume you want to check for setup violations. The slowest arc on the AN2 cell is the B-to-Z arc, so define the drive as follows. Enter

```
dc_shell> set_driving_cell -cell AN2 -pin Z \  
-from_pin B {I4}
```

Defining Loads on Input and Output Ports

By default, Design Compiler assumes zero capacitive load on input and output ports. Use the `set_load` command to set a capacitive load value on input and output ports of the design. This information helps Design Compiler select the appropriate cell drive strength of an output pad and helps model the transition delay on input pads.

For example, to set a load of 30 on output pin out1, enter

```
dc_shell> set_load 30 {out1}
```

Make the units for the load value consistent with the target technology library. For example, if the library represents the load value in picofarads, the value you set with the `set_load` command must be in picofarads. Use the `report_lib` command to list the library units.

Example 7-3 shows the library units for the library `my_lib`.

Example 7-3 Library Units Report

```
*****
Report : library
Library: my_lib
Version: 1999.05
Date   : Mon Jan 4 10:56:49 1999
*****

Library Type           : Technology
Tool Created           : 1999.05
Date Created           : February 7, 1992
Library Version        : 1.800000
Time Unit              : 1ns
Capacitive Load Unit  : 0.100000ff
Pulling Resistance Unit : 1kilo-ohm
Voltage Unit           : 1V
Current Unit           : 1uA
...

```

Defining Fanout Loads on Output Ports

You can model the external fanout effects by specifying the expected fanout load values on output ports, using the `set_fanout_load` command.

For example, enter

```
dc_shell> set_fanout_load 4 {out1}
```

Design Compiler tries to ensure that the sum of the fanout load on the output port plus the fanout load of cells connected to the output port driver is less than the maximum fanout requirement of the library, library cell, and design. (See “Setting Design Rule Constraints” on page 7-26 for more information about maximum fanout requirements.)

Fanout load is not the same as load. Fanout load is a unitless value that represents a numerical contribution to the total effective fanout. Load is a capacitance value. Design Compiler uses fanout load primarily to measure the fanout presented by each input pin. An input pin normally has a fanout load of 1, but you can assign higher values.

Selecting a Compile Strategy

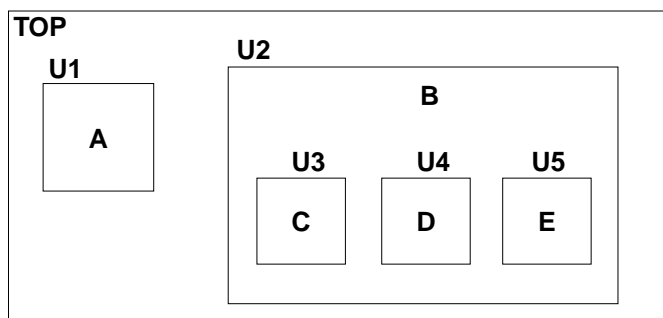
Select a compile strategy before you constrain or compile your design. You can use various strategies to compile your design. This chapter includes information about the following hierarchical compile strategies:

- Top-down compile
- Bottom-up compile

You can process your entire by design using one of these strategies or you can mix strategies, using the most appropriate strategy for each subdesign.

This chapter uses the design shown in Figure 7-4 to illustrate each of the compile strategies.

Figure 7-4 Design to Illustrate Compile Strategies



This design has the specifications shown in Table 7-1. The script shown in Example 7-4 defines these constraints. To prevent buffering of the clock network, the script sets the drive resistance of the clk input to zero (infinite drive strength).

Table 7-1 Design Specifications for Design TOP

Specification Type	Value
Operating condition	WCCOM
Wire load model	"20x20"
Clock frequency	40 MHz
Input delay time	3 ns
Output delay time	2 ns
Input drive strength	drive_of (IV)
Output load	1.5 pf

Example 7-4 Constraints File for Design TOP (defaults.con)

```
set_operating_conditions WCCOM
set_wire_load "20x20"
create_clock -period 25 clk
set_input_delay 3 -clock clk all_inputs()-find(port, clk)
set_output_delay 2 -clock clk all_outputs()
set_load 1.5 all_outputs()
set_driving_cell -cell IV all_inputs()
set_drive 0 clk
```

Top-Down Compile

Use the top-down compile strategy for designs that are not memory or CPU limited.

The top-down compile strategy has these advantages:

- Provides a push-button approach
- Takes care of interblock dependencies automatically

The top-down compile strategy has these disadvantages:

- Requires more memory than other strategies
- Might result in longer runtimes than other strategies for designs with over 100K gates

The top-down compile strategy requires these steps:

1. Read in the entire design.
2. Resolve multiple instances.

For details, see “Resolving Multiple Instances” in Chapter 8.

3. Apply attributes and constraints to the top level.

Attributes and constraints are based on the design specification. For information about attributes, see “Working With Attributes” in Chapter 6. For information about constraints, see “Setting Design Rule Constraints” on page 7-26 and “Setting Optimization Constraints” on page 7-31.

4. Compile the design.

Example 7-5 shows the script used to perform a top-down compile on design TOP. The script contains comments that identify each of the steps in the top-down compile strategy. The script applies constraints by including the constraint file (defaults.con) shown in Example 7-4 on page 7-19.

Example 7-5 Top-Down Compile Script

```
/* read in the entire design */
read -f verilog E.v
read -f verilog D.v
read -f verilog C.v
read -f verilog B.v
read -f verilog A.v
read -f verilog TOP.v
current_design TOP
link

/* resolve multiple references */
uniquify

/* apply constraints and attributes */
include defaults.con

/* compile the design */
compile
```

Bottom-Up Compile

Use the bottom-up compile strategy for medium and large designs.

The bottom-up compile strategy provides these advantages:

- Compiles large designs by using the divide-and-conquer approach
- Requires less memory than top-down compile
- Allows time budgeting

The bottom-up compile strategy has these disadvantages:

- Requires iterations until the interfaces are stable
- Requires manual revision control

The bottom-up compile strategy requires these steps:

1. Compile subblocks independently.

Develop both a default constraint file and subblock-specific constraint files. The default constraint file includes global constraints, such as the clock information and the drive and load estimates. The subblock-specific constraint file reflects the time budget allocated to each subblock.

2. Read in the entire compiled design.
3. Set the current design to the top-level design, link the design, and apply the top-level constraints. If the design meets its constraints, you are done. Otherwise, continue with the following steps.
4. Characterize the subblock with the worst violations.

The `characterize` command calculates the attributes and constraints that describe the environment of the subblock and replaces the existing attributes and constraints with the calculated values.

5. Use `write_script` to save the information from `characterize`.
6. Clear memory.
7. Read in the RTL description of the previously characterized subblock.

8. Set `current_design` to the subblock, and recompile the subblock, using the saved script of characterization data.
9. Read in the entire compiled design again without the old subblock.
10. Link to the recompiled subblock.
11. Choose another subblock, and repeat steps 3 through 9 until you have recompiled all subblocks with their actual environments.

When you use the bottom-up compile strategy, consider the following:

- The `read` command runs quickest with the `.db` format. If you will not be modifying your RTL code after the first time you read (or elaborate) it, save the unmapped design to a `.db` file. This saves time when you reread the design.
- The `compile` command affects all subblocks of the current design. If you want to optimize only the current design, you can remove or not include the subblock in your database, or you can place the `dont_touch` attribute on the subblock (using the `set_dont_touch` command).

Example 7-6 provides the script used to perform a bottom-up compile on design TOP. The script contains comments that identify each of the steps in the bottom-up compile strategy. This script assumes that block constraint files exist for each of the subblocks in design TOP. The compile script also uses the default constraint file (`defaults.con`) shown in Example 7-4 on page 7-19.

Example 7-6 Bottom-Up Compile Script

```
all_blocks = {E,D,C,B,A}

/* compile each subblock independently */
foreach (block, all_blocks) {
  /* read in block */
  block_source = block + ".v"
  read -format verilog block_source
  current_design block
  link
  uniquify
  /* apply global attributes and constraints */
  include defaults.con
  /* apply block attributes and constraints */
  block_script = block + ".con"
  include block_script
  /* compile the block */
  compile
}

/* read in entire compiled design */
read -format verilog TOP.v
current_design TOP
link
write -hierarchy -output first_pass.db

/* apply top-level constraints */
include defaults.con
include top_level.con

/* check for violations */
report_constraint

/* characterize all instances in the design */
all_instances = {U1,U2,U2/U3,U2/U4,U2/U5}
characterize -constraint all_instances

/* save characterize information */
foreach (block, all_blocks) {
  current_design block
  char_block_script = block + ".wscr"
```

```
    write_script > char_block_script
}

/* recompile each block */
foreach (block, all_blocks) {

    /* clear memory */
    remove_design -all

    /* read in previously characterized subblock */
    block_source = block + ".v"
    read -format verilog block_source

    /* recompile subblock */
    current_design block
    link
    uniquify
    /* apply global attributes and constraints */
    include defaults.con
    /* apply characterization constraints */
    char_block_script = block + ".wscr"
    include char_block_script
    /* apply block attributes and constraints */
    block_script = block + ".con"
    include block_script
    /* recompile the block */
    compile
}
}
```

Mixed Compile Strategy

You can take advantage of the benefits of both the top-down and the bottom-up compile strategies by using both.

- Use the top-down compile strategy for small hierarchies of blocks.
- Use the bottom-up compile strategy to tie small hierarchies together into larger blocks.

Setting Design Rule Constraints

This section discusses the most commonly specified design rule constraints:

- Transition time
- Fanout load
- Capacitance

Design Compiler also supports cell degradation and connection class constraints. See the *Design Compiler Reference Manual: Constraints and Timing* for information about these constraints.

Design Compiler uses attributes to represent design rule constraints. Table 7-2 provides the attribute name corresponding to each design rule constraint.

Table 7-2 Design Rule Attributes

Design Rule Constraint	Attribute Name
Transition time	max_transition
Fanout load	max_fanout
Capacitance	max_capacitance
Cell degradation	cell_degradation
Connection class	connection_class

Design rule constraints have two sources: attributes specified in the technology library and attributes you apply explicitly.

If a technology library defines these attributes, Design Compiler implicitly applies them to any design using that library when it compiles the design or creates a constraint report. You cannot remove the design rule attributes defined in the technology library, because they are requirements for the technology, but you can make them more restrictive to suit your design.

If both implicit and explicit design rule constraints apply to a design or a net, the more restrictive value applies.

Setting Transition Time Constraints

The transition time of a net is the time required for its driving pin to change logic values. Design Compiler calculates the transition time for each net by multiplying the drive resistance of the driving pin by the sum of the capacitive loads connected to the driving pin.

Design Compiler and Library Compiler model transition time restrictions by associating a `max_transition` attribute with each output pin on a cell. During optimization, Design Compiler attempts to make the transition time of each net less than the value of the `max_transition` attribute.

To change the maximum transition time restriction specified in a technology library, use the `set_max_transition` command. This command sets a maximum transition time for the nets attached to the identified ports or to all the nets in a design by setting the `max_transition` attribute on the named objects.

For example, to set a maximum transition time of 3.2 on all nets in the design `adder`, enter

```
dc_shell> set_max_transition 3.2 find(design,adder)
```

To undo a `set_max_transition` command, use the `remove_attribute` command. For example, enter

```
dc_shell> remove_attribute find(design,adder) \  
           max_transition
```

Setting Fanout Load Constraints

The maximum fanout load for a net is the maximum number of loads the net can drive.

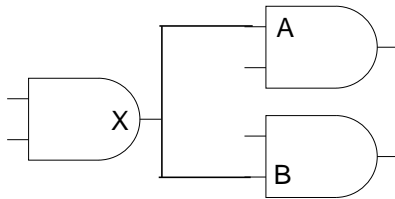
Design Compiler and Library Compiler model fanout restrictions by associating a `fanout_load` attribute with each input pin and a `max_fanout` attribute with each output (driving) pin on a cell.

The fanout load value does not represent capacitance; it represents the weighted numerical contribution to the total fanout load. The fanout load imposed by an input pin is not necessarily 1.0. Library developers can assign higher fanout load values to model internal cell fanout effects.

Design Compiler calculates the fanout of a driving pin by adding the `fanout_load` values of all inputs driven by that pin. To determine if the pin meets the maximum fanout load restriction, Design Compiler compares the calculated fanout load value with the pin's `max_fanout` value.

Figure 7-5 shows a small circuit in which pin X drives two loads, pin A and pin B. If pin A has a `fanout_load` value of 1.0 and pin B has a `fanout_load` value of 2.0, the total fanout load of pin X is 3.0. If pin X has a minimum fanout of 2.0 and a maximum fanout of 16.0, the pin meets the fanout constraints.

Figure 7-5 Fanout Constraint Example



During optimization, Design Compiler attempts to meet the fanout load restrictions for each driving pin. If a pin violates its fanout load restriction, Design Compiler tries to correct the problem (for example, by changing the drive strength of the component).

The technology library might specify default fanout constraints on the entire library or fanout constraints for specific pins in the library description of an individual cell.

To determine if your technology library is modeled for fanout calculations, you can search for the `fanout_load` attribute on the cell input pins:

```
dc_shell> get_attribute find(pin, my_lib/*/*) fanout_load
```

To set a more conservative fanout restriction than that specified in the technology library, use the `set_max_fanout` command on the design or on an input port. (Use the `set_fanout_load` command to set the expected fanout load value for output ports.)

The `set_max_fanout` command sets the maximum fanout load for the specified input ports or for all the nets in a design by setting the `max_fanout` attribute on the specified objects. For example, to set a `max_fanout` requirement of 16 on all nets in the design adder, enter

```
dc_shell> set_max_fanout 16 find(design, adder)
```

If you use the `set_max_fanout` command and a library `max_fanout` attribute exists, Design Compiler tries to meet the smaller (more restrictive) fanout limit.

To undo a `set_max_fanout` command, use the `remove_attribute` command. For example, enter

```
dc_shell> remove_attribute find(design,adder) max_fanout
```

Setting Capacitance Constraints

The transition time constraints do not provide a direct way to control the actual capacitance of nets. If you need to control capacitance directly, use the maximum capacitance constraint. The maximum capacitance constraint is fully independent, so you can use it in addition to the transition time constraints.

Design Compiler and Library Compiler model capacitance restrictions by associating a `max_capacitance` attribute with each output pin on a cell. Design Compiler calculates the capacitance on a net by adding the wire capacitance to the capacitance of the pins attached to the net. To determine if the net meets the capacitance restrictions, Design Compiler compares the calculated capacitance value with the pin's `max_capacitance` value.

To change or add to the maximum capacitance restriction specified in a technology library, use the `set_max_capacitance` command. The `set_max_capacitance` command sets a maximum capacitance for the nets attached to the named ports or to all the nets in a design by setting the `max_capacitance` attribute on the specified objects.

For example, to set a maximum capacitance of 3 for all nets in the design adder, enter

```
dc_shell> set_max_capacitance 3 find(design,adder)
```

To undo a `set_max_capacitance` command, use the `remove_attribute` command. For example, enter

```
dc_shell> remove_attribute find(design,adder) \  
max_capacitance
```

Setting Optimization Constraints

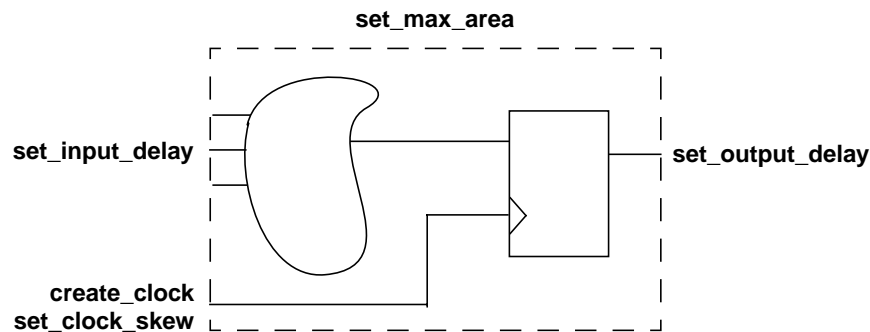
This section discusses the most commonly specified optimization constraints:

- Timing constraints
- Area constraints

Design Compiler also supports power and routability constraints. See the *Power Products Reference Manual* for information about power constraints. See the *Design Compiler Reference Manual: Constraints and Timing* for information about routability constraints.

Figure 7-6 illustrates some of the common commands used to define the optimization constraints.

Figure 7-6 Commands Used to Define the Optimization Constraints



Setting Timing Constraints

Timing constraints specify the required performance of the design. To set the timing constraints,

1. Define the clocks.
2. Specify the I/O timing requirements relative to the clocks.
3. Specify the combinational path delay requirements.
4. Specify the timing exceptions.

Table 7-3 lists the most commonly used commands for these steps.

Table 7-3 Commands to Set Timing Constraints

Command	Description
create_clock	Defines the period and waveform for the clock.
set_clock_skew	Defines the clock network delay.
set_input_delay	Defines the timing requirements for input ports relative to the clock period.
set_output_delay	Defines the timing requirements for output ports relative to the clock period.

Table 7-3 Commands to Set Timing Constraints (continued)

Command	Description
set_max_delay	Defines maximum delay for a combinational path.
set_min_delay	Defines minimum delay for a combinational path.
set_false_path	Specifies a false path.
set_max_delay	Defines maximum delay requirements.
set_min_delay	Defines minimum delay requirements.
set_multicycle_path	Specifies a multicycle path.

The following sections describe these steps in more detail.

Defining a Clock

For synchronous designs, the clock period is the most important constraint, because it constrains all register-to-register paths in the design.

Defining the Period and Waveform for the Clock

Use the `create_clock` command to define the period (`-period` option) and waveform (`-waveform` option) for the clock. If you do not specify the clock waveform, Design Compiler uses a 50 percent duty cycle.

Use the `create_clock` command on a pin or a port. For example, to specify a 25-megahertz clock on port `clk` with a 50 percent duty cycle, enter

```
dc_shell> create_clock clk -period 40
```

When your design contains multiple clocks, pay close attention to the common base period of the clocks. The common base period is the least common multiple of all the clock periods. For example, if you have clock periods of 10, 15, and 20, the common base period is 60.

Define your clocks so that the common base period is a small multiple of each of the clock periods. The common base period requirement is qualitative; no hard limit exists. If the base period is more than ten times larger than the smallest period, however, long runtimes and greater memory requirements can result.

If you have a register-to-register path where one register has a period of 10 and the other has a period of 10.1, the common base period is 1010.0. The timing analyzer calculates the setup requirement for this path by expanding both clocks to the common base period and determining the tightest single-cycle relationship for setup. Internally, the timing analyzer only approximates for extreme cases such as this because the paths are not really synchronous.

You can work around this problem by specifying a clock period without a decimal point and adjusting the clock period by inserting clock uncertainty.

```
dc_shell> create_clock -period 10 clk1
dc_shell> create_clock -period 10 clk2
dc_shell> set_clock_skew -plus_uncertainty 0.1 clk2
```

Use the `report_clock` command to show information about all clock sources in your design.

Use the `remove_clock` command to remove a clock definition.

Creating a Virtual Clock

In some cases, a system clock might not exist in a block. You can use the `create_clock -name` command to create a virtual clock for modeling clock signals present in the system but not in the block. By creating a virtual clock, you can represent delays that are relative to clocks outside the block.

```
dc_shell> create_clock -period 30 -waveform {10 25} \  
               -name sys_clk
```

Specifying Clock Network Delay

By default, Design Compiler assumes that clock networks have no delay (ideal clocks). Use the `set_clock_skew` command to specify timing information about the clock network delay. You can use this command to specify either estimated or actual delay information.

Use the `-propagated` option of the `set_clock_skew` command to specify that you want Design Compiler to calculate clock network delay by propagating times through the clock network. For example,

```
dc_shell> set_clock_skew -propagated clk
```

Use the `-plus_uncertainty` or the `-minus_uncertainty` options of the `set_clock_skew` command to add some margin of error into the system to account for variances in the clock network resulting from layout. For example, on the 20-megahertz clock mentioned previously, to add a 0.2 margin on each side of the clock edge, enter

```
dc_shell> set_clock_skew -plus_uncertainty 0.2 \  
               -minus_uncertainty 0.2 clk
```

Use the `-skew` option of the `report_clock` command to show clock network skew information.

Design Compiler uses the clock skew information when determining if a path meets setup and hold requirements. To see the effect of the skew definition, generate a verbose constraint report (using the `report_constraint -verbose` command).

Specifying I/O Timing Requirements

If you do not assign timing requirements to an input port, Design Compiler assumes that the signal arrives at the input port at time 0. In most cases, input signals arrive at staggered times. Use the `set_input_delay` command to define the arrival times for input ports. You define the input delay constraint relative to the system clock and to the other inputs. The earliest an input signal can arrive is time 0.

If you do not assign timing requirements to an output port, Design Compiler does not constrain any paths from a register output to the output port. Use the `set_output_delay` command to define the required output arrival time. You define the output delay constraint relative to the system clock.

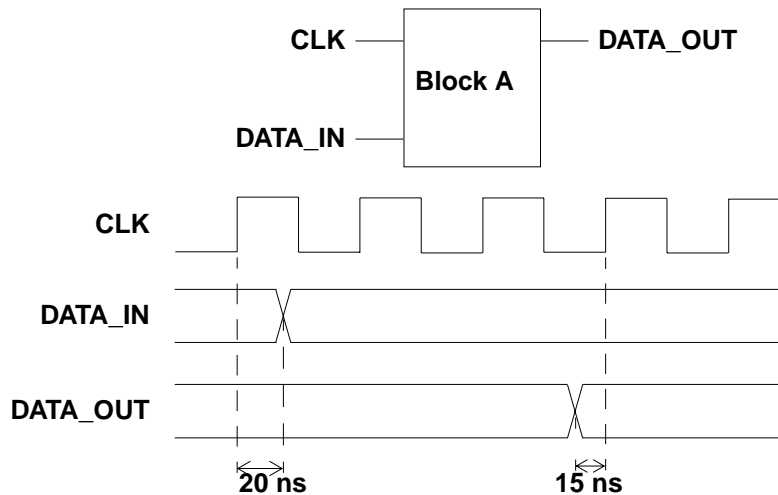
If an input or output port has multiple timing requirements (because of multiple paths), use the `-add_delay` option to specify the additional timing requirements.

Use the `report_port` command to list input or output delays associated with ports.

Use the `remove_input_delay` command to remove input delay constraints. Use the `remove_output_delay` command to remove output delay constraints.

Figure 7-7 shows the timing relationship between the delay and the active clock edge (the rising edge in this example).

Figure 7-7 Relationship Between Delay and Active Clock Edge



In the figure, block A has an input DATA_IN and an output DATA_OUT. From the waveform diagram, DATA_IN is stable 20 ns after the clock edge, and DATA_OUT needs to be available 15 ns before the clock edge.

After you set the clock constraint, using the `create_clock` command, use the `set_input_delay` and `set_output_delay` commands to specify these additional requirements. For example, enter

```
dc_shell> set_input_delay 20 -clock CLK DATA_IN
dc_shell> set_output_delay 15 -clock CLK DATA_OUT
```

Specifying Combinational Path Delay Requirements

For purely combinational delays that are not bounded by a clock period, use the `set_max_delay` and `set_min_delay` commands to define the maximum and minimum delays for the specified paths.

A common way to produce this type of asynchronous logic in HDL code is to use asynchronous sets or resets on latches and flip-flops. Because the reset signal crosses several blocks, constrain this signal at the top level.

For example, to specify a maximum delay of 5 on the RESET signal, enter

```
dc_shell> set_max_delay 5 -from RESET
```

To specify a minimum delay of 10 on the path from IN1 to OUT1, enter

```
dc_shell> set_min_delay 10 -from IN1 -to OUT1
```

Use the `report_timing_requirements` command to list the minimum delay and maximum delay requirements for your design.

Specifying Timing Exceptions

Timing exceptions define timing relationships that override the default single-cycle timing relationship for one or more timing paths. Use timing exceptions to constrain or disable asynchronous paths or paths that do not follow the default single-cycle behavior. Be aware that specifying numerous timing exceptions can increase the compile runtime.

Design Compiler recognizes only timing exceptions that have valid reference points.

- The valid startpoints in a design are the primary input ports or the clock pins of sequential cells.
- The valid endpoints are the primary output ports of a design or the data pins of sequential cells.

Design Compiler does not generate a warning message if you specify invalid reference points. You must use the `-ignored` option of the `report_timing_requirements` command to find timing exceptions ignored by Design Compiler.

You can specify the following conditions by using timing exception commands:

- False paths (`set_false_path`)
- Minimum delay requirements (`set_min_delay`)
- Maximum delay requirements (`set_max_delay`)
- Multicycle paths (`set_multicycle_path`)

Use the `report_timing_requirements` command to list the timing exceptions in your design.

Specifying False Paths

Design Compiler does not report false paths in the timing report or consider them during timing optimization. Use the `set_false_path` command to specify a false path. Use this command to ignore paths that are not timing-critical, that can mask other paths that must be considered during optimization, or that never occur in normal operation.

For example, Figure 7-8 shows a configuration register that can be written and read from a bidirectional bus (DATA) in a chip.

Creating a false path differs from disabling a timing arc. Disabling a timing arc represents a break in the path. The disabled timing arc permanently disables timing through all affected paths. Specifying a path as false does not break the path; it just prevents the path from being considered for timing or optimization.

Specifying Minimum and Maximum Delay Requirements

You can use the `set_min_delay` and `set_max_delay`, commands described earlier in this chapter, to specify path delay requirements that are more conservative than those derived by Design Compiler based on the clock timing.

To undo a `set_min_delay` or `set_max_delay` command, use the `reset_path` command with similar options.

Register-to-Register Paths

Design Compiler uses the following equations to derive constraints for minimum and maximum path delays on register-to-register paths:

$$\begin{aligned} \text{min_delay} &= (T_{\text{capture}} - T_{\text{launch}}) + \text{hold} \\ \text{max_delay} &= (T_{\text{capture}} - T_{\text{launch}}) - \text{setup} \end{aligned}$$

Override the derived path delay ($T_{\text{capture}} - T_{\text{launch}}$), using the `set_min_delay` and `set_max_delay` commands.

For example, assume you have a path launched from a register at time 20 that arrives at a register where the next active edge of the clock occurs at time 35.

```
dc_shell> create_clock -period 40 waveform {0 20} clk1
dc_shell> create_clock -period 40 -waveform {15 35} clk2
```

Design Compiler automatically derives a maximum path delay constraint of $(35 - 20) - (\text{library setup time of register at endpoint})$. To specify a maximum path delay of 10, enter

```
dc_shell> set_max_delay 10 -from reg1 -to reg2
```

Design Compiler calculates the maximum path delay constraint as $10 - (\text{library setup time of register at endpoint})$, which overrides the original derived maximum path delay constraint.

Register-to-Port Paths

Design Compiler uses the following equations to derive constraints for minimum and maximum path delays on register-to-port paths:

```
min_delay = period - output_delay  
max_delay = period - output_delay
```

If you use the `set_min_delay` or `set_max_delay` commands, the value specified in these commands replaces the period value in the constraint calculation. For example, assume you have a design with a clock period of 20. Output OUTPORTA has an output delay of 5.

```
dc_shell> create_clock -period 20 CLK  
dc_shell> set_output_delay 5 -clock CLK OUTPORTA
```

Design Compiler automatically derives a maximum path delay constraint of 15 ($20 - 5$). To specify that you want a maximum path delay of 10, enter

```
dc_shell> set_max_delay 10 -to OUTPORTA
```

Design Compiler calculates the maximum path delay constraint as $5 (10 - 5)$, which overrides the original derived maximum path delay constraint.

Asynchronous Paths

You can also use the `set_max_delay` and `set_min_delay` commands to constrain asynchronous paths across different frequency domains.

For example,

```
dc_shell> set_max_delay 17.1 \  
          -from find(clock, clk1) -to find(clock, clk2)  
dc_shell> set_max_delay 23.5 \  
          -from find(clock, clk2) -to find(clock, clk3)  
dc_shell> set_max_delay 31.6 \  
          -from find(clock, clk3) -to find(clock, clk1)
```

Setting Multicycle Paths

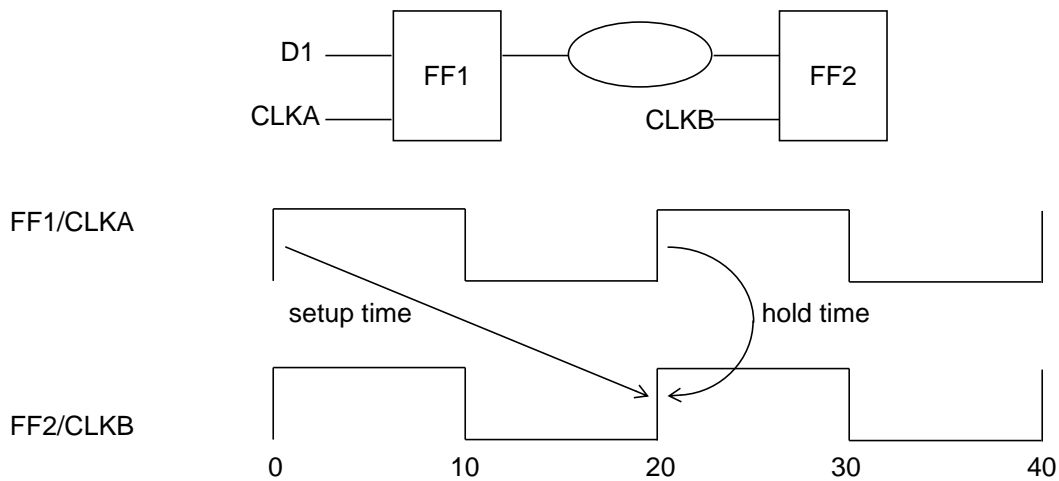
The multicycle path condition is appropriate when the path in question is longer than a single cycle or when data is not expected within a single cycle. Use the `set_multicycle_path` command to specify the number of clock cycles Design Compiler should use to determine when data is required at a particular endpoint.

You can specify this cycle multiplier for setup or hold checks. If you do not specify the `-setup` or `-hold` option with the `set_multicycle_path` command, Design Compiler applies the multiplier value only to setup checks.

By default, setup is checked at the next active edge of the clock at the endpoint after the data is launched from the startpoint (default multiplier of 1). Hold data is launched one clock cycle after the setup data but checked at the edge used for setup (default multiplier of 0).

Figure 7-9 shows the timing relationship of setup and hold times.

Figure 7-9 Setup and Hold Timing



The timing path starts at the clock pin of FF1 (rising edge of CLKA) and ends at the data pin of FF2. Assuming that the flip-flops are positive-edge-triggered, the setup data is launched at time 0 and checked 20 time units later at the next active edge of CLKB at FF2. Hold data is launched one (CLKA) clock cycle (time 20) and checked at the same edge used for setup checking (time 20).

The `-setup` option of the `set_multicycle_path` command moves the edge used for setup checking to before or after the default edge. For the example shown in Figure 7-9,

- A setup multiplier of 0 means that Design Compiler uses the edge at time 0 for checking
- A setup multiplier of 2 means that Design Compiler uses the edge at time 40 for checking

The `-hold` option of the `set_multicycle_path` command launches the hold data at the edge before or after the default edge, but Design Compiler still checks the hold data at the edge used for checking setup. As shown in Figure 7-9 (assuming a default setup multiplier),

- A hold multiplier of 1 means that the hold data is launched from CLKA at time 40 and checked at CLKB at time 20
- A hold multiplier of -1 means that the hold data is launched from CLKA at time 0 and checked at CLKB at time 20

To undo a `set_multicycle_path` command, use the `reset_path` command with similar options.

Using Multiple Timing Exception Commands

A specific timing exception command refers to a single timing path. A general timing exception command refers to more than one timing path. If you execute more than one instance of a given timing exception command, the more specific commands override the more general ones.

The following rules define the order of precedence for a given timing exception command:

- The highest precedence occurs when you define a timing exception from one pin to another pin.
- A command using only the `-from` option has a higher priority than a command using only the `-to` option.
- For clocks used in timing exception commands, if both `-from` and `-to` are defined, they override commands that share the same path defined by either the `-from` or the `-to` option.

This list details the order of precedence (highest at the top) defined by these precedence rules:

1. *command -from pin -to pin*
2. *command -from clock -to pin*

3. *command -from pin -to clock*
4. *command -from pin*
5. *command -to pin*
6. *command -from clock -to clock*
7. *command -from clock*
8. *command -to clock*

For example, in the following command sequence, paths from A to B are treated as two-cycle paths because specific commands override general commands:

```
dc_shell> set_multicycle_path 2 -from A -to B
dc_shell> set_multicycle_path 3 -from A
```

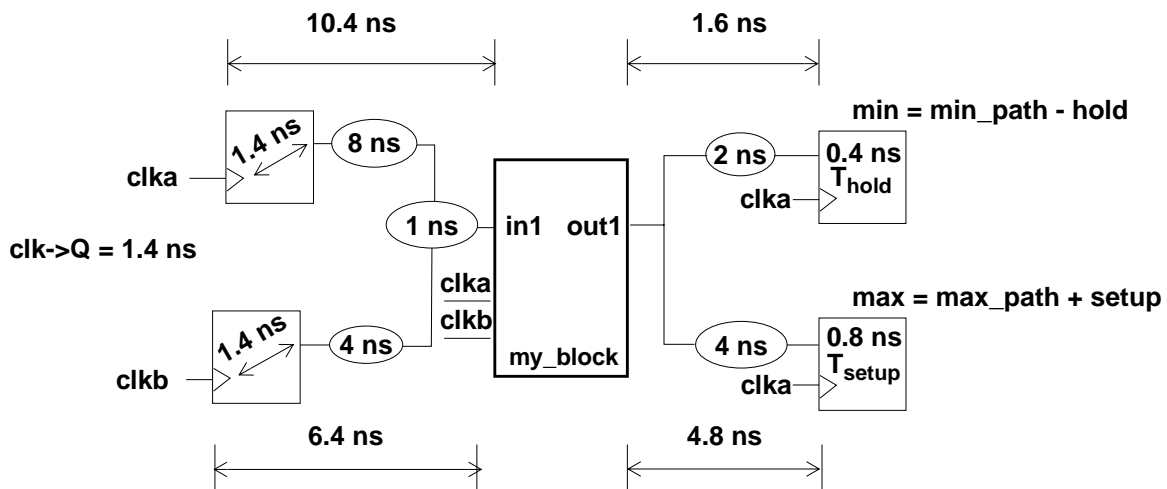
The following rules summarize the interaction of the timing exception commands:

- General `set_false_path` commands override specific `set_multicycle_path` commands.
- General `set_max_delay` commands override specific `set_multicycle_path` commands.
- Specific `set_false_path` commands override specific `set_max_delay` or `set_min_delay` commands.
- Specific `set_max_delay` commands override specific `set_multicycle_path` commands.

Constraining a Sample Block

Figure 7-10 illustrates the timing requirements for the block `my_block`. Example 7-7 shows the script used to specify these timing requirements.

Figure 7-10 Timing Requirements for `my_block`



Example 7-7 Timing Constraints for `my_block`

```
create_clock -period 20 -waveform {5 15} clka
create_clock -period 30 -waveform {10 25} clkb
set_input_delay 10.4 -clock clka in1
set_input_delay 6.4 -clock clkb -add_delay in1
set_output_delay 1.6 -clock clka -min out1
set_output_delay 4.8 -clock clka -max out1
```

Setting Area Constraints

The `set_max_area` command specifies the maximum area for the current design by placing a `max_area` attribute on the current design. Specify the area in the same units used for area in the technology library.

For example, to set the maximum area to 100, enter

```
dc_shell> set_max_area 100
```

Design area consists of the areas of each component and net. The following components are ignored when Design Compiler calculates design area:

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

Cell (component) area is technology-dependent; Design Compiler obtains this information from the technology library.

When you specify both timing and area constraints, Design Compiler attempts to meet timing goals before area goals. To prioritize area constraints over total negative slack (but not over worst negative slack), use the `-ignore_tns` option when you specify the area constraint.

```
dc_shell> set_max_area -ignore_tns 100
```

To optimize a small area, regardless of timing, remove all constraints except for maximum area. You can use the `remove_constraint` command to remove constraints from your design. Be aware that this command removes *all* optimization constraints from your design.

Analyzing the Precompiled Design

Before compiling your design, verify that

- The design is consistent

Use the `check_design` command to verify design consistency. See “Checking for Design Consistency” in Chapter 9 for information about the `check_design` command.

- The attributes and constraints are correct

Design Compiler provides many commands for reporting the attributes and constraints. See “Analyzing Design Problems” and “Analyzing Timing Problems” in Chapter 9 for information about these commands.

