

Introduction

So you're new to Design Compiler and have been assigned to do the synthesis for a given design. Where do you begin? From tutorials you may get scripts, but upon review the scripts seem to be written in a foreign language. All you want to know is how do I read in the files, what constraints do I need, and how do I generate reports.

The goal of this manual is to provide you that information. The manual starts with the synthesis flow and ends with writing files. Everything in between, including reading files, preparing constraints, compiling, and report generation, is included. Chapter1 gives an overall idea about DC. Chapter 2 discusses the flow through constraint generation. Chapter 3 picks up with compile techniques and ends with writing files.

Chapter1

Introduction

This chapter provides step-by-step procedure for compiling a design without giving any constraints. This is an introduction to the Synopsys design compiler. This tool takes a behavioral (program) version of a design and converts it to gate-level connected netlist using a Technology library.

In this lab, we will take sample Verilog code and convert it into a netlist. This is assumed to be functionally correct and does not need to be tested. Synopsys Design Compiler is used throughout this lab. Design Compiler is the core synthesis engine of Synopsys synthesis product family.

For the sake of this tutorial we will consider a counter program.

Design Compiler has 2 user interfaces: -

1. Design Vision- a GUI (Graphical User Interface)
2. dc_shell - a command line interface

Basic design flow

We will first setup the design, read the design and then compile. Though in real time, design is not compiled without applying constraints, this flow is only to make you comfortable with the tool. The actual flow starts from Chapter 2. But it is recommended for the students to first do this chapter and then continue with Chapter 2.

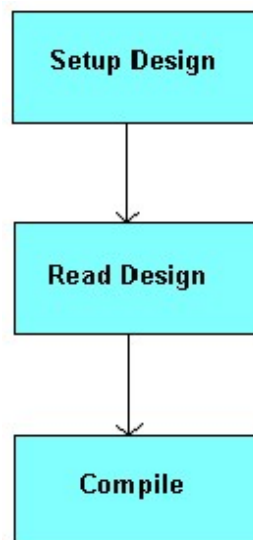


Figure 1

Setting up of your account

Before you begin the tutorial you need to copy the tutorial files from Linux home page to your working directory. Design Compiler options are setup in a file called `\synopsys_dc.setup`", which is placed in tutorial directory. Later we have discussed in detail about this file. The command used to copy the tutorial files to your working directory is

`% cp -r /home/tutorials .` (*Indicates the present directory*)

Defining Libraries

For most people libraries are not an interesting topic, but every tool needs them and Design compiler isn't any different. You need to tell Design Compiler what libraries to use when synthesizing your HDL code to gates. There are different types of libraries with different definitions. For example, the link and target libraries are technology libraries that define the semiconductor vendor's set of cells and related information, such as cell names, cell pin names, delay arcs, pin loading, design rules, and operating conditions.

Another type of library is called the synthetic or DesignWare library. This library is where your adders, multipliers, and so on come from. You do not need to specify the standard synthetic library, `standard.sldb`, which implements the built-in HDL operators; Design Compiler automatically uses this library. If you are using additional DesignWare libraries, you must specify these libraries by using the `synthetic_library` variable (for optimization purposes) and the `link_library` variable (for cell resolution purposes).

To specify the link, target, symbol, and synthetic libraries you use the `link_library`, `target_library`, `symbol_library`, and `synthetic_library` variables. These libraries are normally placed in a setup file named `.synopsys_dc.setup`.

Synopsys Setup File

Here is an example of a setup file using a `cb13fs120_tsmc_max` library. A discussion of the file follows.

```
# This is a Tcl-s script
```

```
define_design_lib work -path ./work
set link_library {* ./ cb13fs120_tsmc_max.db dw01.sldb dw02.sldb dw_foundation.sldb}
set target_library {./ cb13fs120_tsmc_max.db}
set symbol_library {./ cb13fs120_tsmc_max.sdb ./ cb13fs120_tsmc_max.slib}

set search_path [list ./ ../LIB /remote/release/2003.03-2/libraries/syn]
set synthetic_library [list dw01.sldb dw02.sldb dw_foundation.sldb]
```

Setup File Description

The file has been written in Tcl. The lines starting with a pound sign (#) are comments.

- The `define_design_lib` command tells Design Compiler where you want intermediate files written. This command is discussed in more detail later.
- The `set link_library` command specifies the list of design files and libraries used during linking. The asterisk (*) shown in the line tells Design Compiler that the link command is to search all the designs loaded in `dc_shell` while trying to resolve references. You should always have this entry.
- The `set target_library` command specifies the list of technology libraries of components to be used when compiling a design.
- The `set symbol_library` command specifies the symbol libraries to use during schematic generation.
- The `set search_path` command allows you to tell Design Compiler what directories it should search in to find files.
- The `set synthetic_library` command specifies a list of synthetic libraries to use when compiling.

Start Design compiler

Before you invoke the Design Compiler, you have to be in tutorial directory. So use the following commands.

% cd tutorials

Starting and exiting DC

Now you are ready to invoke the DC. The command used is

% design_vision

The Design Vision window, shown in [Figure2](#), is displayed. In addition, the `design_vision-t` prompt (in `dcctl` mode) appears in the shell where you started Design Vision and you are ready to begin the tutorial exercise.

You can exit Design Compiler from Design Vision at any time. Choose `File > Exit` from the menu bar and then click OK in the warning message box, or enter the exit command on the command line. When all the windows are closed logout from your account. Design Compiler does not automatically save the designs that you work on. To save your designs, use the `File > Save` or `File > Save As` menu commands, or enter the `write -format keyword` command on the command line. Here keyword represents any of the design file formats supported by Design Compiler.

NOTE: If the design compiler is not closed in the method explained above, the server will get hanged and will have to be restarted again. So, follow the steps given above.

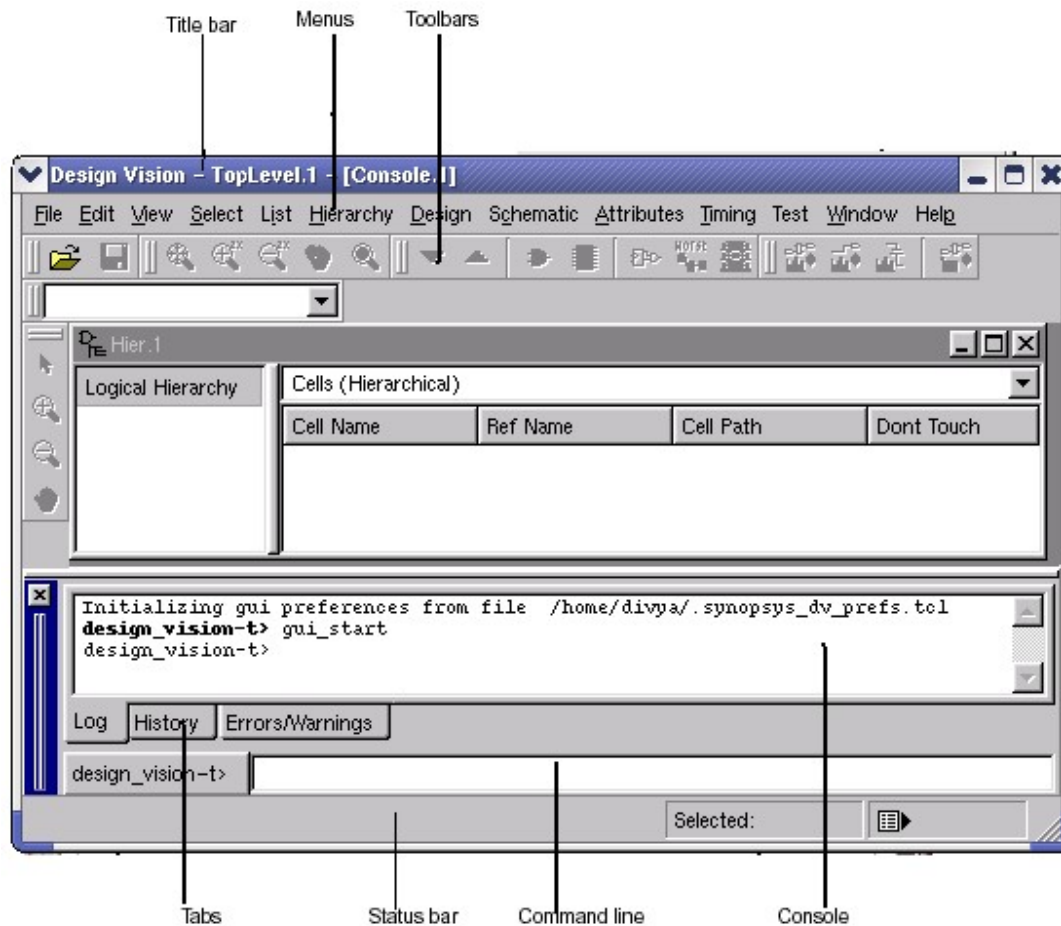


Figure 2

You can exit Design Compiler from Design Vision at any time. Choose File > Exit from the menu bar and then click OK in the warning message box, or enter the **exit** command on the command line.

Analyze / elaborate versus read_file

There are two methods to read files into Design Compiler. One is to use the analyze/elaborate **commands**; the other is to use the command `read_file`. This section describes how the commands work and includes a comparison between the two.

analyze

The analyze command does the following:

- Reads an HDL source file and performs HDL syntax checking and Synopsys rule checking.

- Checks file for errors without building generic logic for the design.
- Creates HDL library objects in an intermediate format.
- Stores the intermediate files in a location specified by you with the `define_design_lib` command. The command used for this is

```
%analyze -library work -format verilog {{/home/tutorials/counter.v}}
```

Elaborate

The elaborate command does the following:

- Translates the design into its GTECH representation.
- Allows changing of parameter values defined in the source code.
- Allows VHDL architecture selection.
- Replaces the HDL arithmetic operators in the code with Design Ware components.
- Performs link automatically.

The command is

```
% elaborate counter -architecture verilog -library work -update
```

After elaboration we can view the technology independent schematic as shown in figure 2.

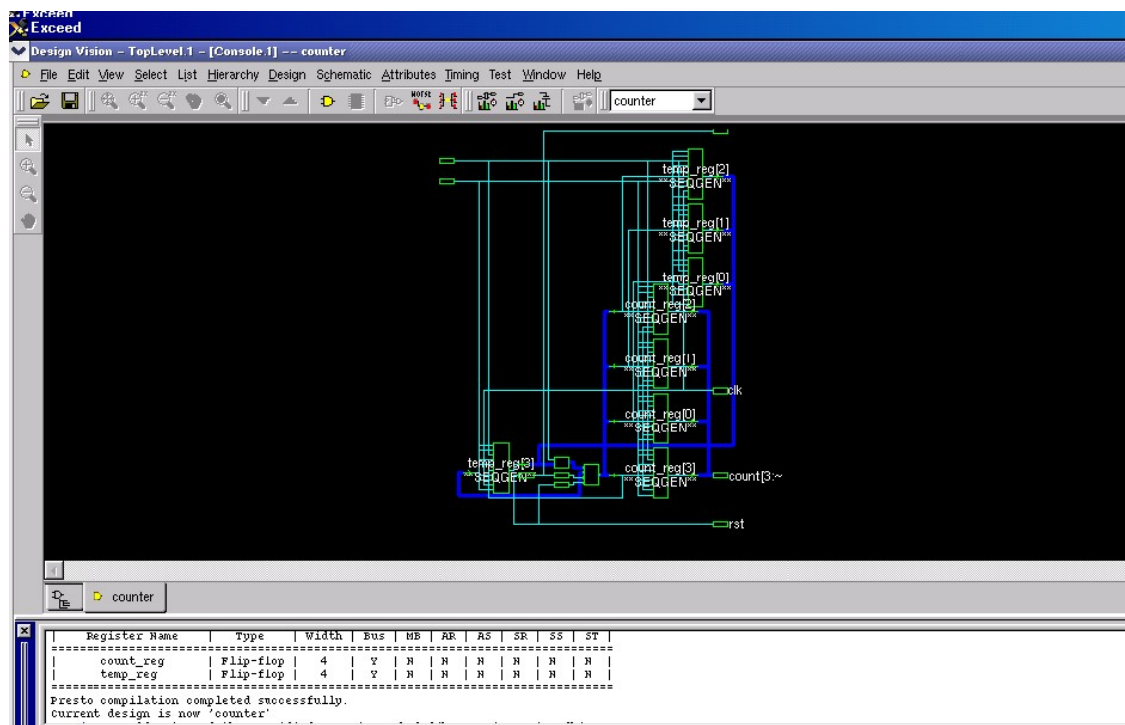


Figure 3

read_file

The read_file command does the following:

- Performs the same operations as analyze and elaborate in one step.
- It does not create any intermediate files for Verilog.
- Creates .mr and .st intermediate files for VHDL.
- It does not execute the link command automatically.

The command is

```
% read_file -format verilog counter.v
```

Creating the clock

To create the clock the command used is

```
% create_clock -period 3.33 -name myclk [get_ports clk]
```

Compile the design

Now we are ready to compile the design, the output of which is netlist. The command is

```
% compile -map_effort medium -area_effort medium
```

After compiling we can see the technology dependent schematic as shown in the figure 3.

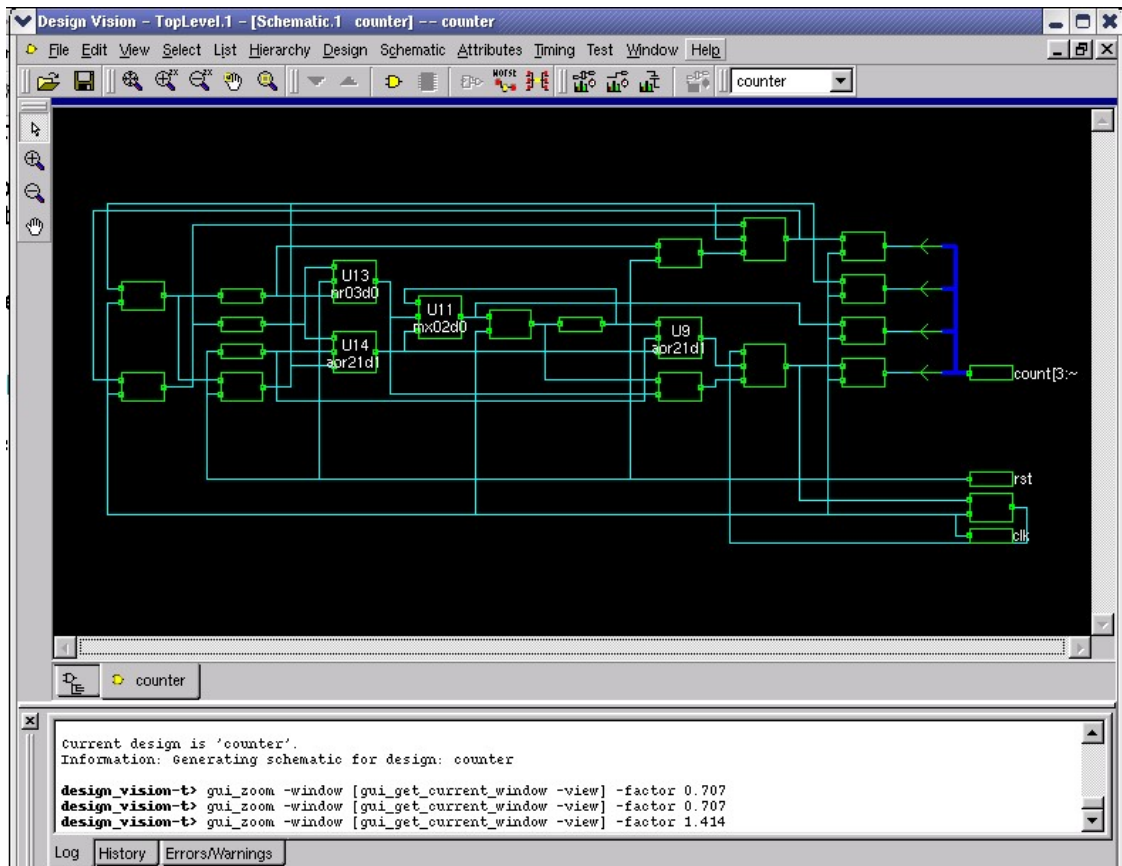


Figure 4

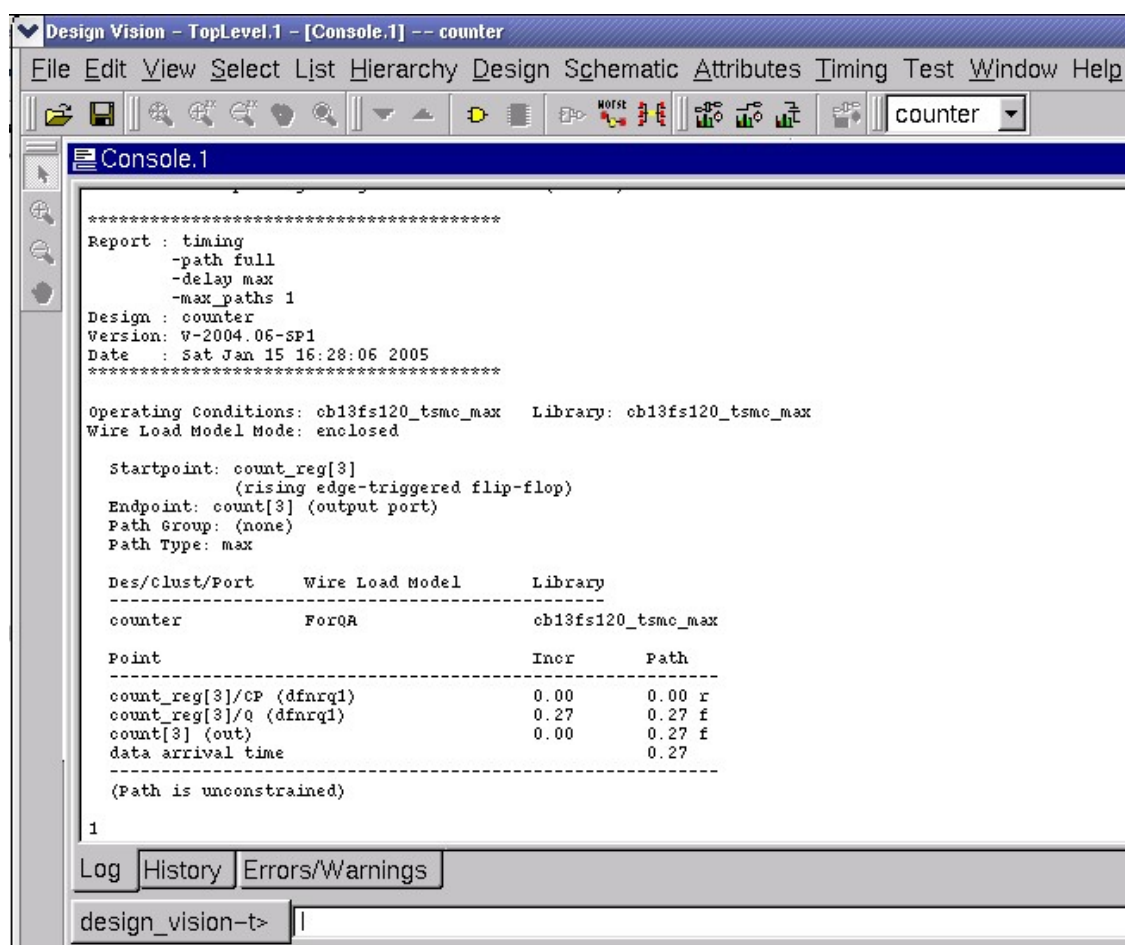
Checking the design

Various options are there to check the Design. Now we will use some basic option such as timing report and area report.

To check the timing report command used is

% report_timing

The timing report is displayed in console window as shown in figure 4.



```
Design Vision - TopLevel.1 - [Console.1] -- counter
File Edit View Select List Hierarchy Design Schematic Attributes Timing Test Window Help
Console.1
*****
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : counter
Version: V-2004.06-SP1
Date : Sat Jan 15 16:28:06 2005
*****
Operating Conditions: cb13fs120_tsmc_max Library: cb13fs120_tsmc_max
Wire Load Model Mode: enclosed

Startpoint: count_reg[3]
              (rising edge-triggered flip-flop)
Endpoint: count[3] (output port)
Path Group: (none)
Path Type: max

Des/Clust/Port  Wire Load Model  Library
-----
counter        ForQA        cb13fs120_tsmc_max

Point          Inor      Path
-----
count_reg[3]/CP (dfnrq1)      0.00      0.00 r
count_reg[3]/Q (dfnrq1)      0.27      0.27 f
count[3] (out)                0.00      0.27 f
data arrival time              0.27
-----
(Path is unconstrained)

1
Log History Errors/Warnings
design_vision-t> |
```

Figure 5

To check for the area command used is

% report_area

The area report is displayed in console window as shown in figure 5

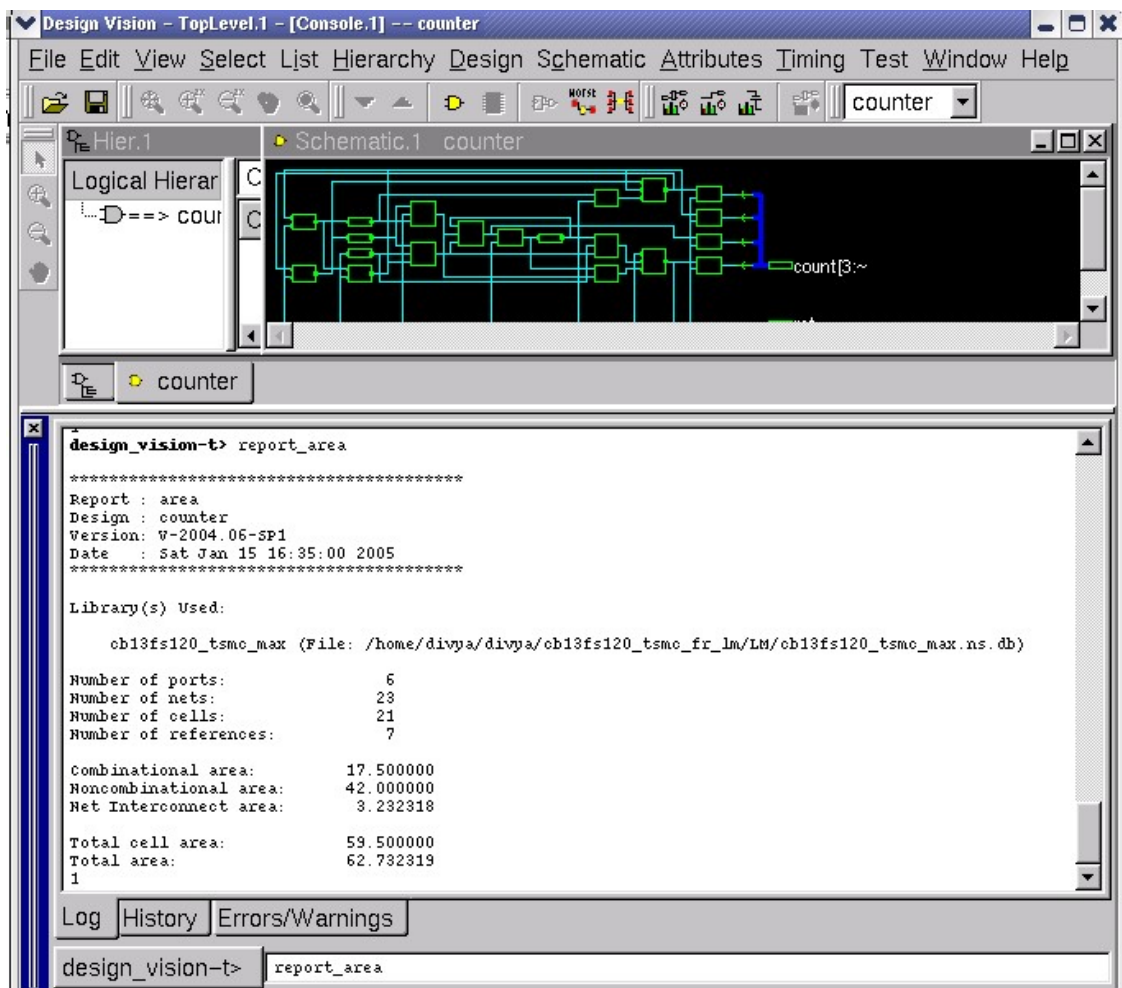


Figure 6

Saving the file

Finally the output file can be saved as .v or .vhd or .db. To save this file we use write command as shown.

- ❑ To save it as .v command is

```
% write -format verilog -output ab.v
```

- ❑ To save it as .vhd command is

```
% write -format vhdl -output ab.vhd
```

- ❑ To save it as .db command is

% write –format db
OR

% write –hierarchy –output ab.db.

Exiting Design Compiler

So here you are done with the first chapter of the tutorial. Now u can exit the design compiler. First remove all the designs. For this type the following command

% remove_design –designs

Then to come out of design compiler give

% exit

If it asks for exit design vision? Say ok.

Assignment

Write a verilog code for RS FF, which is to be saved as RSFF.v and carry out the steps explained in chapter 1.

Chapter 2

In this chapter we will see how to define constraints for a design. To do this chapter you should have done with chapter1. This chapter can be started after reading the design or after elaborate command of chapter1.

If you have finished with chapter 1 and closed the Design Compiler, then you have to invoke the design compiler and read the design as explained in chapter1. Then you are ready to go with chapter2.

Design flow

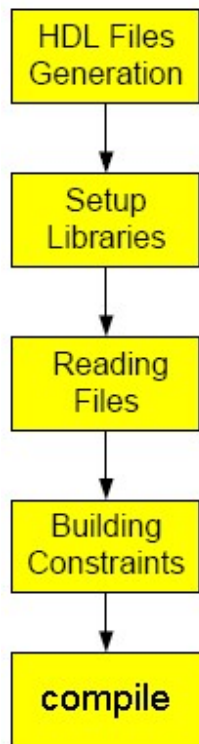


Figure 7

The design flow chart tells, what you need to know about getting the design into Design Compiler and preparing the design for synthesis.

Focused Constraints

Now that your design has been read into memory you need to define the design environment and constraints. For the design environment you'll want to tell Design Compiler about things it doesn't know anything about, such as what operating conditions the chip will see, which wire load models should be used, and system interface requirements. In addition Design Compiler needs to know about area and timing constraints. Now we will see this one by one.

Design Environment

The design environment constraints consist of operating conditions, wire load models, and system interface requirements. All areas are covered below.

Operating Conditions

The operating conditions consist of temperature, voltage, and process. The effects each of these can have on the chip need to be considered during synthesis and timing analysis. During timing analysis, Design Compiler must consider the worst-case and best-case scenarios for the expected variations in the temperature, voltage, and process factors.

Most libraries have default settings for operating conditions. To see what the default conditions are you can use `report_lib` command, or if you have the `.lib` file you can take a look at it in LINUX. Since we don't have `.lib` file, we have to use `report_lib` command.

Continuing with our "Read Verilog Example," let's see what libraries are loaded into memory using the **`list_libs`** command.

```
design_vision-t> list_libs
Logical Libraries:
-----
Library      File          Path
-----
standard.sldb standard.sldb  /usr/synopsys/V-2004.06-SP1/libraries/syn
gtech         gtech.db      /usr/synopsys/V-2004.06-SP1/libraries/syn
cb13fs120_tsmc_max cb13fs120_tsmc_max.ns.db /home/divya/divya/cb13fs120_tsmc_fr_lm/LM
1
```

Looking at this list we can find the name of our library 'cb13fs120_tsmc_max'. Now do a **`report_lib`** to see what the default operating conditions are.

```
design_vision-t> report_lib cb13fs120_tsmc_max
```

```
*****
```

```
Report : library
```

```
Library: cb13fs120_tsmc_max
```

```
Version: V-2004.06-SP1
```

```
Date   : Mon Jan 17 11:34:25 2005
```

```
*****
```

```
...
```

```
Operating Conditions:
```

```
Operating Condition Name : cb13fs120_tsmc_max
```

```
Library : cb13fs120_tsmc_max
```

```
Process : 1.20
```

```
Temperature : 125.00
```

```
Voltage : 1.08
```

```
Interconnect Model : worst_case_tree
```

```
Input Voltages:
```

```
No input_voltage groups specified.
```

```
Output Voltages:
```

```
No output_voltage groups specified.
```

```
default_wire_load_capacitance: 0.000096
```

```
default_wire_load_resistance: 0.000380
```

```
default_wire_load_area: 0.010000
```

There are 3 types of operating conditions. They are Worst Case Commercial (**WCCOM**), Worst Case Industrial (**WCIND**), and Worst Case Military (**WC MIL**). So your library is having some default operating conditions but not these.

If you want to specify explicit operating conditions, which supersede the default library conditions, you can use **set_operating_conditions** command as shown below.

```
design_vision-t> set_operating_conditions WCCOM -lib <library name>
```

Wire Load Models

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

Now that you know what a wire load model is, how can you tell what you have? Using the **report_lib** command again, you can find out this information.

design_vision-t> report_lib cb13fs120_tsmc_max

Report : library

Library: cb13fs120_tsmc_max

Version: V-2004.06-SP1

Date : Mon Jan 17 11:34:25 2005

...

Wire Loading Model:

Name : ForQA

Location : cb13fs120_tsmc_max

Resistance : 0.00038

Capacitance : 9.6e-05

Area : 0.01

Slope : 15.9634

Fanout Length Points Average Cap Std Deviation

1 5.88
2 13.03
3 20.49
4 28.29
5 36.45
6 44.98
7 53.91
8 63.26
9 73.05
10 83.30
11 94.03
12 105.26

Wire Loading Model Selection Group:

Name : predcaps

Selection		Wire load name
min area	max area	
0.00	200.00	ForQA
200.00	8000.00	8000
8000.00	16000.00	16000
16000.00	35000.00	35000
35000.00	70000.00	70000
70000.00	140000.00	140000
140000.00	280000.00	280000
280000.00	540000.00	540000
540000.00	1000000.00	1000000
1000000.00	2000000.00	2000000
2000000.00	4000000.00	4000000
4000000.00	8000000.00	8000000

Wire Loading Model Mode: enclosed.

Wire Loading Model Selection Group: predcaps.

This report provides a lot of information. Let's take a look at what it is telling you.

- The Wire Loading Model section tells you what wire load models are available.
- When the Wire Loading Model Selection Group section is available, this indicates that the library supports automatic area-based wire load model selection.
- The Wire Loading Model Mode section identifies the default wire load mode. If the library doesn't have a default, then Design Compiler will use the top mode.
- The Wire Loading Model Selection Group section tells which wire load models will be user for specific areas.

If you need to change the wire load model or mode you can use commands

set_wire_load_model and **set_wire_load_mode**, respectively. Here is an example:

```
design_vision-t> set_wire_load_model -name "ForQA"
Design counter: Using wire_load model 'ForQA' found in library 'cb13fs120_tsmc_max'.
1
design_vision-t> set_wire_load_mode enclosed
1
```

It is always a good idea to check your constraints by using the **report_design** command. Notice the highlighted areas.


```
design_vision-t> report_lib cb13fs120_tsmc_max
```

```
*****
Report : library
Library: cb13fs120_tsmc_max
Version: V-2004.06-SP1
Date   : Mon Jan 17 11:34:25 2005
*****
```

...

Wire Loading Model:

Selected manually by the user.

```
Name      : ForQA
Location   : cb13fs120_tsmc_max
Resistance : 0.00038
Capacitance : 9.6e-05
Area       : 0.01
Slope      : 15.9634
Fanout     Length   Points Average Cap Std Deviation
-----
1          5.88
2          13.03
3          20.49
4          28.29
5          36.45
6          44.98
7          53.91
8          63.26
9          73.05
10         83.30
11         94.03
12        105.26
13        117.01
14        129.30
15        142.15
```

Wire Loading Model Mode: enclosed.

Constraints

Let's review. You've specified libraries and read in the HDL code. You've told Design Compiler about the chip's operating conditions and wire load models. What's missing?

Well, we haven't indicated what clock speed the chip will be running at nor what size the chip should be. We'll address those now.

Timing

To accurately set up timing constraints, you need to tell Design Compiler about the following:

- Clocks
- I/O timing requirements
- Combinational path delay requirements
- Timing exceptions

create_clock example

```
design_vision-t> create_clock -period 10 [get_ports clk1]
```

The first command tells Design Compiler you have clock with a period of 10. Because the `-name` option was not used, the clock gets the same name as the clock source in this case, `clk1`. The clock will also have a 50 percent duty cycle because a waveform was not specified.

Example for clock constrains

Let's go through a quick example. The figure below is the circuit that we need to constrain. After reading in the file, we'll set the clock constraints then review the clocks using the `report_clock` command.

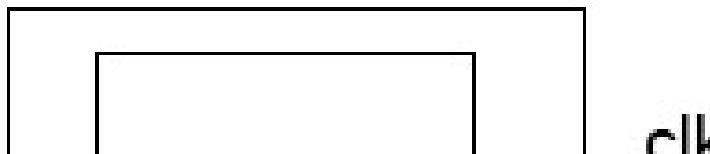


Figure 8

```
dc_shell-t> create_clock -p 10 clk1
set_clock_uncertainty 0.5 [get_clocks clk1]
set_clock_latency 1 [get_clocks clk1]
set_clock_latency -source 3 [get_clocks clk1]
create_generated_clock -name clk_2 -source clk1 -divide_by 2 [get_pins clk_2_reg/Q]
```

Clock Latency

Clock latency is the propagation time from the actual clock origin to the clock definition point in the design. There are two types of clock latency: network and source. Clock network latency is the time it takes a clock signal to propagate from the clock definition point to a register clock pin. Clock source latency is the time it takes for a clock signal to propagate from its actual waveform origin point to the clock definition point in the design. Clock source latency is normally used to model off-chip clock latency. The following figure shows the difference between the two.

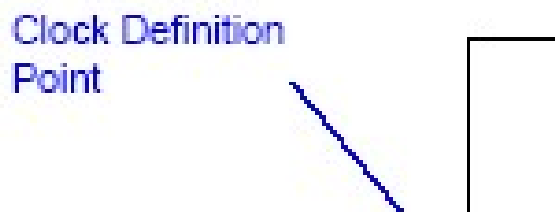


Figure 9

```
design_vision-t> set_clock_latency 1 [get_clocks clk]
design_vision-t> set_clock_latency -source 3 [get_clocks clk]
```

Clock uncertainty

Uncertainty accounts for varying delays between the clock network branches. There are two types of uncertainty: simple and interclock. Simple uncertainty means that setup uncertainty and hold uncertainty applies to all paths to the endpoint. Interclock uncertainty allows you to specify different skew between various clock domains. The recommendation is to set the uncertainty to the worst skew expected to the endpoint or between the clock domains. You can always increase the value to account for additional margin for setup and hold.

This example specifies that all paths leading to registers or ports clocked by CLK have setup uncertainty of 0.65 and hold uncertainty of 0.45.

```
set_clock_uncertainty -setup 0.65 [get_clocks CLK]
set_clock_uncertainty -hold 0.45 [get_clocks CLK]
```

The following example specifies interclock uncertainties between PHI1 and PHI2 clock domains.

```
set_clock_uncertainty 0.4 -from PHI1 -to PHI1
set_clock_uncertainty 0.4 -from PHI2 -to PHI2
set_clock_uncertainty 1.1 -from PHI1 -to PHI2
set_clock_uncertainty 1.1 -from PHI2 -to PHI1
```

Now for our design lets us give a uncertainty of 0.5

```
design_vision-t>set_clock_uncertainty 0.5 [get_clocks clk]
```

After this check for the clock constraints given by you by using **report_clock**.

```
design_vision-t> report_clock
```

```
Information: Updating design information... (UID-85)
```

```
Allocating blocks in 'counter'
```

```
.  
.   
.
```

```
*****
```

```
Report : clocks
```

```
Design : counter
```

```
Version: V-2004.06-SP1
```

```
Date   : Mon Jan 17 15:12:46 2005
```

```
*****
```

```
Attributes:
```

```
  d - dont_touch_network
```

```
  f - fix_hold
```

```
  p - propagated_clock
```

```
  G - generated_clock
```

Clock	Period	Waveform	Attrs	Sources
clk	10.00	{0 5}	p	{clk}

1

Now if you use **report_clocks -skew**, here is what you'll get.

```
design_vision-t> report_clock -skew
Information: Updating design information... (UID-85)
```

```
*****
Report : clock_skew
Design : counter
Version: V-2004.06-SP1
Date   : Mon Jan 17 15:26:03 2005
*****
```

Object	Rise	Fall	Min Rise	Min fall	Uncertainty	
	Delay	Delay	Delay	Delay	Plus	Minus
clk	1.00	1.00	1.00	1.00	0.50	0.50

Object	Max Source Latency				Min Source Latency			
	Early Rise	Early Fall	Late Rise	Late Fall	Early Rise	Early Fall	Late Rise	Late Fall
clk	3.00	3.00	3.00	3.00	3.00	3.00	3.00	3.00

By using this report you can check to see if the `set_clock_uncertainty` and `set_clock_latency` commands were used correctly. Looking at rise delay, fall delay and so on, we see a delay of 1ns. This delay came from the command **`set_clock_latency 1 [get_clocks clk]`**.

Clock uncertainty is 0.5 and source latency is 3.0 ns. These delay values match the commands that were used: **`set_clock_uncertainty 0.5 [get_clocks clk]`** and **`set_clock_latency -source 3 [get_clocks clk]`**.

I/O Timing Requirements

What about inputs and outputs? If you don't constrain inputs and outputs, Design Compiler assumes the signal arrives at the input ports at time 0 and does not constrain any paths that end at an output port. The **`set_input_delay`** and **`set_output_delay`** commands are used to constrain input and output ports.

Set_input_delay

The `set_input_delay` command is used to specify how much time is used by external logic. Design Compiler then calculates how much time is left for the internal logic. If you don't use this command the tool will assume an input port clock and pick a clock period. Let's look at an example.

External Logic

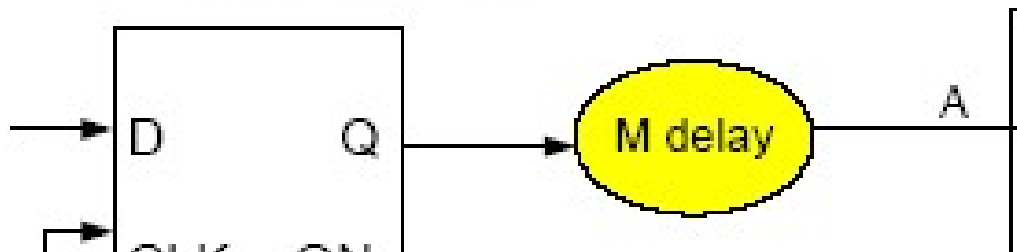


Figure 10

The delay for the external logic will be $T_{clk-q} + T_m$. The time remaining for internal logic is $T_n + T_{setup}$. So if $T_{clk-q} + T_m = 7.4$ ns, the clock period is 10ns, and the flop has a 1 ns setup requirement, what is the maximum delay for T_n ? The calculation will be as shown. $10 \text{ ns} - 7.4 \text{ ns} - 1 \text{ ns} = 1.6 \text{ ns}$. Here is what the constraints would look like.

- **create_clock -period 10 [get_ports clk]**
- **set_input_delay -max 7.4 -clock clk [get_ports A]**

set_output_delay

The `set_output_delay` command is used to constrain the output paths. You need to specify how much time the external log needs; then Design Compiler calculates how much time is left for the internal logic. Here's an example.

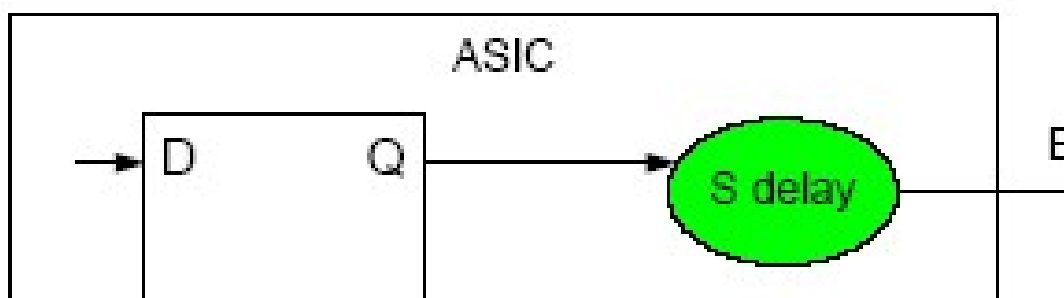


Figure 11

The external delay equals $T_{setup} + T_t$. The internal delay is $T_{clk-q} + T_s$. If the external delay is 7 ns, T_{clk-q} is 1.0 ns, and the clock period is 10 ns, what is the value of T_s ? T_s would be $10 \text{ ns} - 7 \text{ ns} - 1 \text{ ns} = 2 \text{ ns}$. Here is what the constraints would look like:

- **create_clock -period 10 [get_ports clk]**
- **set_output_delay -max 7.0 -clock clk [get_ports B]**

For simplicity let us give input and output delay as 3 ns

```
design_vision-t> set_input_delay 3 -clock clk [all_inputs]
1
design_vision-t> set_output_delay 3 -clock clk [all_outputs]
1
```

Area

Area is another topic that needs to be covered. Design Compiler will perform minimal area optimizations unless an area constraint is set. By using the **set_max_area** command, you can constrain area.

```
design_vision-t> set_max_area 100
1
```

Design Rules

Your library vendor may impose design rules that restrict how many cells are connected to one another based on capacitance, transition, and fanout. You 33 can override these rules using these commands: **set_max_capacitance**, **set_max_transition**, and **set_max_fanout**.

Set_max_capacitance

If you need to control capacitance, you can use the **set_max_capacitance** command. Using this command Design Compiler will try to make sure that the capacitance value for a net is less than the value specified.

```
design_vision-t> set_max_capacitance 2 [get_ports rst]
1
```

Set_max_transition

The transition time of a net is the time required for its driving pin to change logic values. This time is based on the library data. For the nonlinear delay model (NLDM), output transition time is a function of input transition and output load. If you need to change this value, use the **set_max_transition** command.

```
design_vision-t> set_max_transition 3 [get_ports rst]
1
```

Set_max_fanout

The maximum fanout load for a net is the maximum number of loads the net can drive. Design Compiler attempts to ensure that the sum of the **fanout_load** attributes for input pins on nets driven by the specified ports or all nets in the specified design is less than the given value set in the **set_max_fanout** command.

```
design_vision-t> set_max_fanout 6 [get_ports rst]
1
```

Now compile the design as given in chapter 1 and see the timing and area report.
Compare these reports with the reports that you got in chapter1.

```
design_vision-t> report_timing
Information: Updating design information... (UID-85)
```

Report : timing

-path full

-delay max

-max_paths 1

Design : counter

Version: V-2004.06-SP1

Date : Wed Jan 19 12:09:38 2005

Operating Conditions: cb13fs120_tsmc_max Library: cb13fs120_tsmc_max
Wire Load Model Mode: enclosed

Startpoint: rst (input port clocked by clk)

Endpoint: count_reg[3]

(rising edge-triggered flip-flop clocked by clk)

Path Group: clk

Path Type: max

Des/Clust/Port	Wire Load Model	Library
counter	ForQA	cb13fs120_tsmc_max

Point	Incr	Path
clock clk (rise edge)	0.00	0.00
clock network delay (ideal)	4.00	4.00
input external delay	3.00	7.00 r
rst (in)	0.00	7.00 r
U29/Z (or02d1)	0.14	7.14 r
U30/ZN (inv0d1)	0.08	7.23 f
U23/Z (aor21d1)	0.18	7.40 f
U33/ZN (inv0d1)	0.04	7.45 r
U15/ZN (oai321d1)	0.46	7.90 f
count_reg[3]/D (dfnrq1)	0.00	7.90 f
data arrival time		7.90
clock clk (rise edge)	10.00	10.00
clock network delay (ideal)	4.00	14.00

clock uncertainty	-0.50	13.50
count_reg[3]/CP (dfnrq1)	0.00	13.50 r
library setup time	-0.08	13.42
data required time		13.42

data required time		13.42
data arrival time		-7.90

slack (MET)		5.51

1

By observing the report we can see that slack is positive 5.51. So timing constraints have been met.

design_vision-t> report_area
Information: Updating design information... (UID-85)

Report : area
Design : counter
Version: V-2004.06-SP1
Date : Wed Jan 19 12:33:08 2005

Library(s) Used:

cb13fs120_tsmc_max (File:
/home/vasudev/vasudev/cb13fs120_tsmc_fr_lm/LM/cb13fs120_tsmc_max.ns.db)

Number of ports:	6
Number of nets:	25
Number of cells:	23
Number of references:	9

Combinational area:	19.500000
Noncombinational area:	42.000000
Net Interconnect area:	3.641935

Total cell area:	61.500000
Total area:	65.141937

1

From the report it is clear that given area constraints have been met.

Now save the design and quit from Design Compiler.

Exceptions

Almost every design has exceptions. Exceptions can be false paths or multicycle paths. The set_false_path command can be used to tell Design Compiler to ignore the timing

constraints on certain paths. This command is useful for constraining asynchronous paths and logically false paths. Let's look at an example for an asynchronous path.

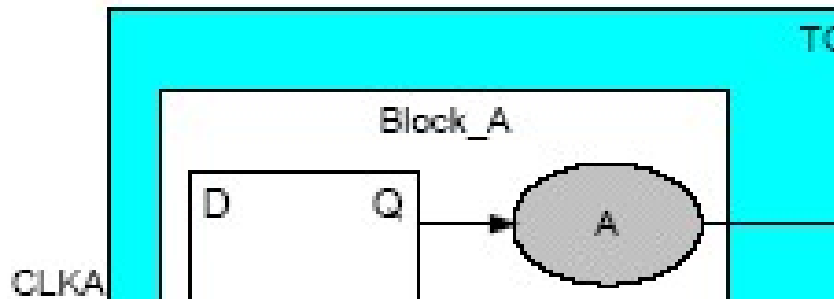


Figure 12

In this example CLKA and CLKB are asynchronous to each other, so how do we constrain it? First you want to define the clocks using `create_clock`, then tell Design Compiler not to optimize logic crossing the clock domains by using `set_false_path`.

- `set_false_path -from [get_clocks CLKA] -to [get_clocks CLKB]`
- `set_false_path -from [get_clocks CLKB] -to [get_clocks CLKA]`

Set_multicycle_path

The `set_multicycle_path` command is used to tell Design Compiler that you need longer than a single clock cycle for a path. Using this command you can specify how many clocks you will need. Let's go through an adder example.

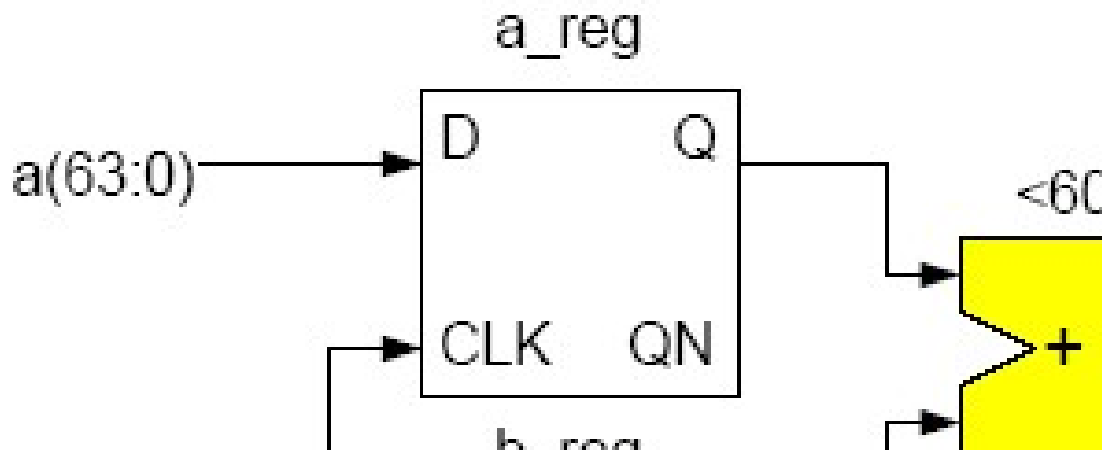


Figure 13

In this example the adder will take 6 clock cycles to finish. Here is how you would constrain this circuit.

- ```
create_clock -period [get_clocks clk]
set_multicycle_path 6 -setup -to [get_pins c_reg[*]/D]
```

**Important:** The default hold check is always performed on the clock edge before the setup check. In this example, the hold check would occur at 50ns.

### **Combinational path delay requirements**

Now that our clocks have been defined and inputs and outputs are constrained, what do we need to do for paths? To constrain a pure combinational path, you can use the **set\_max\_delay** and **set\_min\_delay** commands.

#### **Set\_max\_delay**

The **set\_max\_delay** command allows you to specify the maximum path length for any startpoint to any endpoint. Design Compiler will try to make the path less than the delay value you set.

#### **Set\_min\_delay**

The **set\_min\_delay** command allows you to specify the minimum path delay. If a path violates the requirement given in a **set\_min\_delay** command, Design Compiler adds delay to fix the violation if maximum delay cost is not increased.

You can use the **report\_timing\_requirements** command to see the maximum and minimum constraints.

### **Conclusion**

Congratulations, you have finished Chapter 2. A lot of ground was covered and there is more to come. Before continuing with chapter 3, review what you've learned and dig deeper into the commands. Using the figures and examples, you should be able to set up your libraries, read in you files, and write constraints.

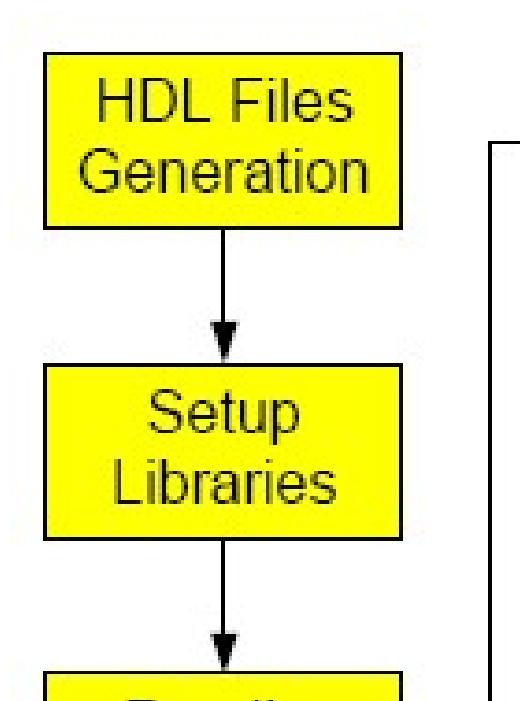
## Chapter 3

### Introduction

This chapter discusses about the different compile options. For a single design all the options for compile cannot be covered. So in general I have discussed all possible compile options, used for a design. Then which compile option you have to choose? That depends on your design and requirement.

### Basic Synthesis Flow

The flow chart describes the steps you need to cover for synthesis. The first step is to specify what libraries you want to synthesize with. Libraries are provided from library vendors. Typically, your ASIC vendor will let you know which library to use. Next you need to read in the files. The different methods for reading in files will be covered along with comparisons of each. Once the files have been read, you need to set constraints. When the constraints have been defined, you'll need to compile, analyze the results, and write out the synthesized netlist.



**Figure 14**

The yellow boxes cover what you need to know about getting the design into Design Compiler and preparing the design for synthesis. The purple boxes cover how to compile the design, perform static timing analysis, and write out necessary files.

## Quick Review

First step is to Libraries need to be defined before reading in your code. The `link_library`, `target_library`, `symbol_library`, and `synthetic_library` variables need to be set to define your libraries. Typically these variable definitions are located in your `.synopsys_dc.setup` file. You can read in your code by using the `analyze/elaborate` or `read_file`.

Once the code has been read in, you need to set constraints. Several constraints need to be written. For example,

- Design Environment Constraints
  - Operating Conditions
  - Wire Load Models
- Timing and Area Constraints
  - Timing
    - Clocks
    - I/O
    - Timing Exceptions
  - Area
  - Design Rules

If any of this quick review is not clear, review key section in chapter1 and chapter 2.

## Great Compiles

Once constraints have been defined, you're ready to optimize the design. The optimization process consists of three steps.

1. **Architectural** – The architectural optimization, sometimes called high-level synthesis, works on the HDL code. Optimization is based on the constraints you set and on HDL coding style. This phase includes tasks such as sharing common sub expressions, sharing resources, and selecting Design Ware components. These tasks, with the exception of Design Ware selection, occur on an unmapped design. After this optimization, the design is represented as GTECH library parts.
2. **Logic Level** – The logic-level optimization is broken into two processes: structuring and flattening. Both of these processes work on a GTECH design. I like to think of this process as Boolean equation manipulation.
  - a. **Structuring** – Optimization during this phase is influenced by the constraints you set. This process adds intermediate variables and logic structure to a design that can help reduce area. Design Compiler searches for subfunctions that can be factored out and evaluates these factors based on the size of the factor and number of times the factor appears in the design. Design Compiler turns the subfunctions that reduce the logic most into intermediate variables and factors them out of the design equations.

- b. Flattening** – Here the goal is to convert combinational logic paths into a sum-of-products representation. This conversion is independent of constraints. During this process, Design Compiler removes all intermediate variables from a design. This process increases CPU time and can increase area.
- 3. Gate Level** – It is during gate-level optimization that you have actual gates. This optimization has four processes: mapping, delay optimization, design rule fixing, and area optimization.
  - a. Mapping** – This optimization phase uses gates from the specified library to generate the gate-level implementation of the design.
  - b. Delay Optimization** – During this step, Design Compiler tries to fix delay violations that were introduced in the mapping phase. Design rule violations or area constraints are not addressed in this step.
  - c. Design Rule Fixing** – The goal is to correct any design rule violations that were introduced by inserting buffers or resizing existing cells. Design Compiler tries to fix these violations without affecting timing and area results.
  - d. Area Optimization** – The final step is to meet area constraints. By default, Design Compiler does not allow area recovery to introduce design rule or delay constraint violations to meet the area constraint.

Logic- and gate-level optimizations are performed by using the compiler command. Compile stops when all the constraints have been met, Design Compiler reaches a point of diminishing returns, or the user interrupts compile. When the compile stops, you have a gate-level presentation of your design. At this point, you should write out a Synopsys database (.db) file. How to do this is discussed in section called Important Output Files.

## Compile Strategies

The two most popular compile strategies are top-down or bottom-up. Other strategies include compile for timing, area, and runtime. All these things are discussed briefly here. For more detail information you can refer solvnet articles.

### Top-Down

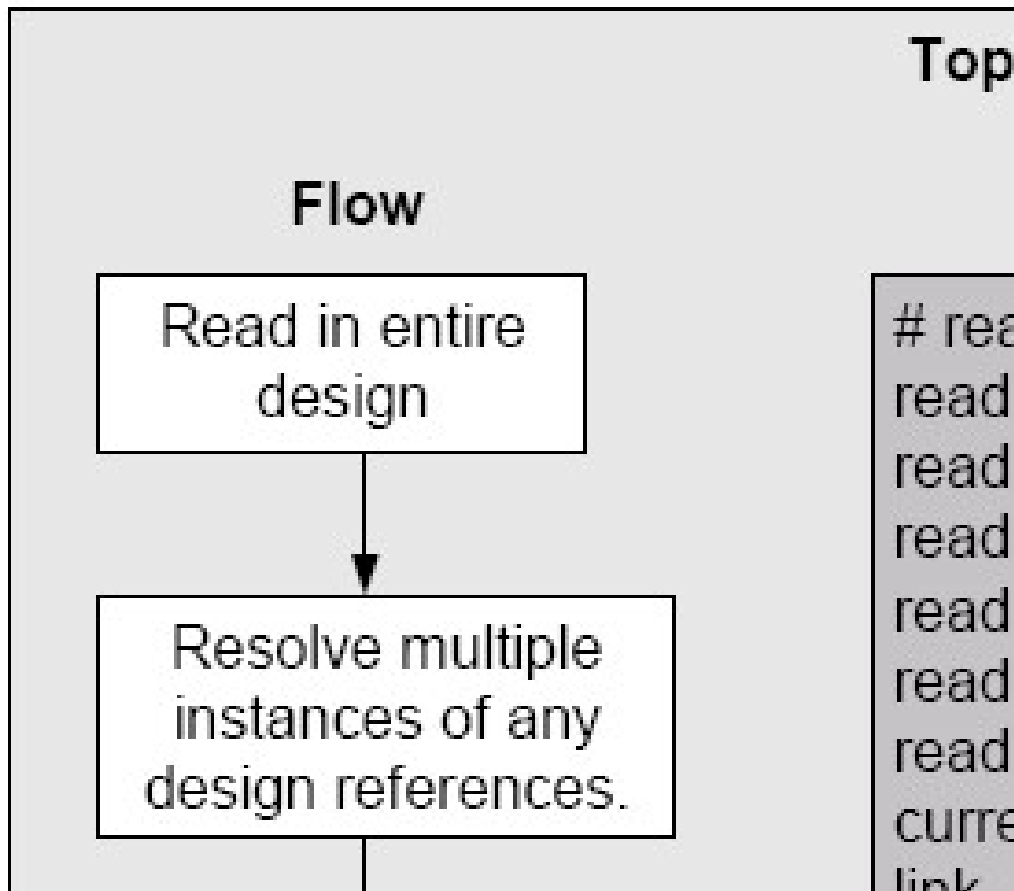
The limiting factors for top-down compiles are available memory and time. A general guideline is that if the compile takes longer than a day, you should consider breaking the design into smaller blocks. Here are the advantages of doing this:

- Push-button approach
- Automatically takes care of inter-block dependencies.
- Fewer scripts to write and maintain.
- The scripts are easier to understand and migrate to other designs.

- Design Compiler sees across the hierarchy and accurately computes the delays, loads, fanout, and so on.

## Top-Down Flow

Figure 15 shows an example of a top-down flow and script. The `uniquify` command used in the flow removes multiply-instantiated hierarchy in the current design by creating a unique design for each cell instance.



**Figure 15**

## Bottom-Up Compile

You should use a bottom-up approach for larger designs. Here are some advantages:

- Requires less memory.
- Compiles large designs by using the divide-and-conquer approach
- Quickly identifies the critical paths for possible recoding.

### **Bottom-Up Flow**

A bottom-up flow requires a lot more time than a top-down approach. Here are the required steps, including suggested steps to follow if timing is not met:

1. Generate a default constraint file and subdesign-specific constraint files. The default constraint file should include global constraints, such as the clock information and the drive and load estimates. The subdesign-specific constraint files reflect the time budget allocated to the subblocks.
2. Compile the subdesigns independently.
3. Read in the top-level design and any compiled subdesigns not already in memory.
4. Set the current design to the top-level design, link the design, and apply the top-level constraints. If the design meets its constraints, congratulations you can stop here. Otherwise, continue with the following steps.
5. Apply the characterize command to the cell instance with the worst violations. The characterize command captures information about the environment of specific cell instances, and assigns the information as attributes on the design to which the cells are linked.
6. Use write\_script to save the characterized information for the cell. You can then use this script to re-create the new attribute values when you are recompiling the cell's referenced subdesign. The write\_script command writes Design Compiler commands to save the current settings.
7. Use remove\_design -all to remove all designs from memory. The remove\_design command acts like its name. It removes the design.
8. Read in the RTL design of the previously characterized cell. Recompiling the RTL design instead of the cell's mapped design usually leads to better optimization.
9. Set current\_design to the characterized cell's subdesign and recompile, using the saved script of characterization data.
10. Read in all other compiled subdesigns.
11. Link the current subdesign.
12. Choose another subdesign, and repeat steps 3 through 9 until you have recompiled all subdesigns, using their actual environments.



Figure 16 shows an example script. The side in purple is the portion of the script you would need if you had timing violations.

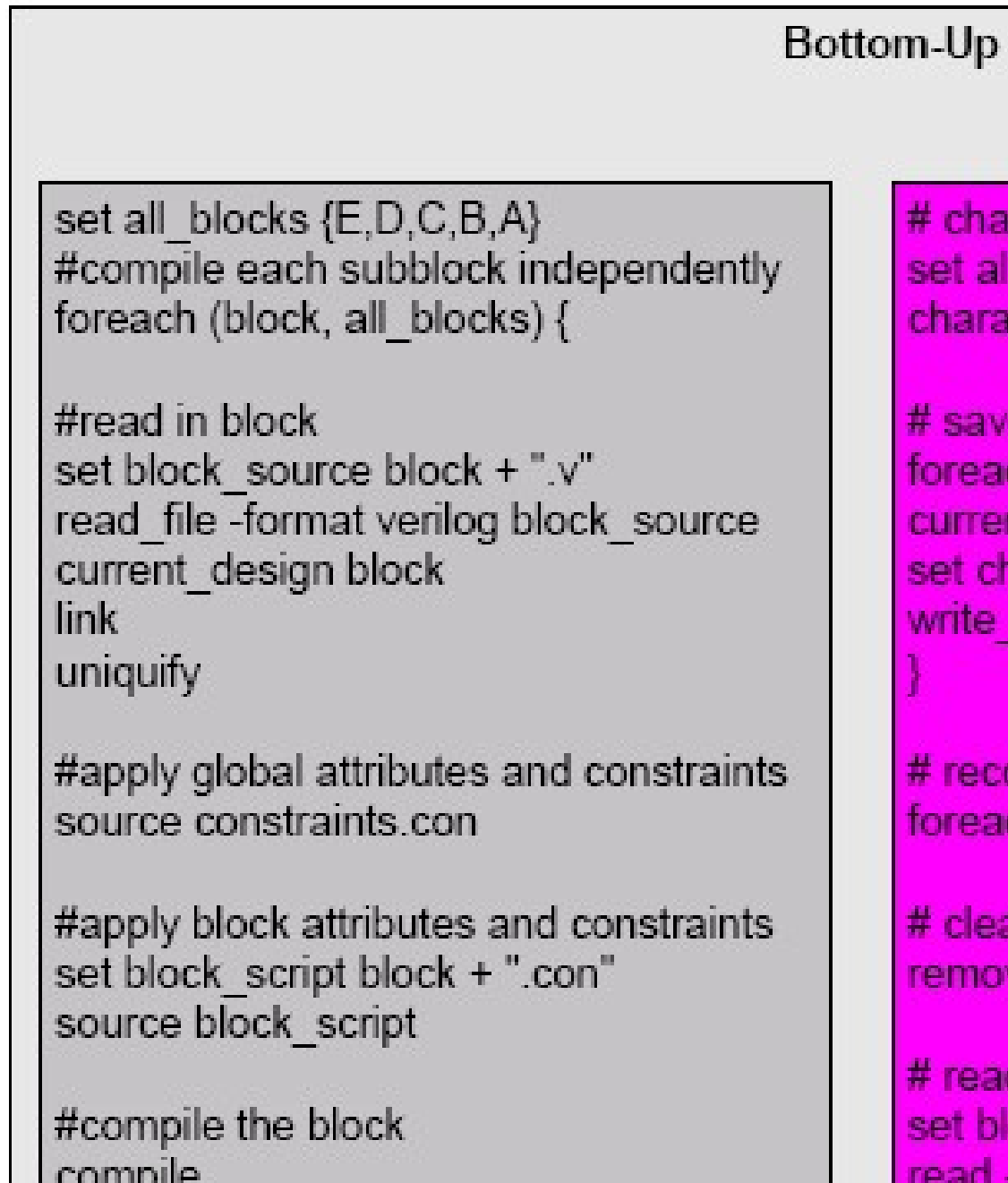


Figure 16

## Compile for Timing

If your timing is critical, here are some things to consider.

- Ungroup unnecessary hierarchies
- Use the `group_path` command and virtual clocks to isolate input/output paths
- Use DC-Ultra and DesignWare Foundation

## Compile for Area

If your design is area sensitive, check the following:

1. Understand your design's minimum area. You can find the design's minimum area by running simple compile mode with no clock or timing constraints. For example, set `simple_compile_mode true` compile
2. Ungroup smaller blocks to allow shared optimization across boundaries.
3. When area is critical, you should tell Design Compiler. You can do this by using the `set_max_area area_constraint [-ignore_tns]` command. The *-ignore\_tns* option specifies that area is to be prioritized above total negative slack
4. Using the compile `-area_effort high` or `-map_effort high` commands enables the gate composition algorithm to further reduce the number of cells.
5. You can use the compile `-auto_ungroup area` command to automatically ungroup small blocks during compile.
6. Try setting the `compile_sequential_area_recovery` variable to true to remap all sequential cells to try and recover area.
7. To reduce area, use Boolean structuring to take advantage of don't cares and redundancy by doing the following:

```
set compile_new_boolean_structure true
```

8. To delete registers that have constraints on the outputs, do the following:

```
set compile_seqmap_propagate_constraints true
```

## Compile for Runtime

Is runtime your enemy? Here are some suggestions to improve runtime.

- Design hierarchy can impact synthesis runtime and QOR, so if possible ungroup unnecessary hierarchical instances.
- Check your timing and design rule constraints. Setting timing exceptions by using wildcards, for example, can impact runtime. Critical range setting can also impact runtime.
- Use Presto Verilog for elaboration to reduce runtime. Presto Verilog is on by default.
- Try using simple compile mode if you know your blocks will meet timing.
- Set `simple_compile_mode` true.

## Helpful Report Generation

What happens when you don't meet timing and how do you know you have not met timing? We'll address those questions plus more in this section.

### Reporting Violations

A good starting point is to use the `report_constraint -all_violators` command. This command reports all the constraints that have been violated in the design, which includes design rules, setup, hold, and area. When you generate a report, you should notice whether you have big or small violations. A 10 percent or less violation is considered small. Here is an example report.

### Big Violations

Let's look at the following report.

```
dc_shell-t> report_constraint
Information: Updating design
```

Report : constraint  
-all\_violators  
Design : dff  
Version: 2003.06-2  
Date : Fri Aug 15 06:56:21 2003  
\*\*\*\*\*

In this report we see that the require path delay is 0 ns and the actual path delay is 0.39 ns. What can we do? Running another compile might not help unless you change something. Here are some things to check.

1. Modify the constraints. You need to make sure your constraints are realistic.
2. Check the design partition. If possible, try to ungroup unnecessary subblocks.
3. Change the set\_structure, compile\_new\_boolean\_structure, or set\_flatten options.
  - a. Set\_structure – By default, Design Compiler structures your design. You can use this command to set various structure attributes on a design to determine how the designs are structured during compile.
  - b. compile\_new\_boolean\_structure– When the compile\_new\_boolean\_structure variable is set to true (the default is false), it turns on the new Boolean (non-algebraic) structure optimization in the structuring phase of compile.
  - c. Set\_flatten – By default, Design Compiler does not flatten your design. You can use the set\_flatten command to control this behavior.
4. Recompile using a higher effort. The compile –map\_effort [low | medium | high] command controls how hard Design Compiler works on the critical path during gate-level optimization.
  - a. Low – shouldn't be used for production work or as starting point for other optimizations.
  - b. Medium – You should always start here (the default). Using medium effort typically produces good results.

- c. High – Using a high effort activates additional algorithms. Be warned—this is very CPU intensive.

5. Change the HDL source code. Sometimes it is necessary to change the HDL source code.

### **Small Violation**

What about small violations? In addition to the techniques described above, try these options:

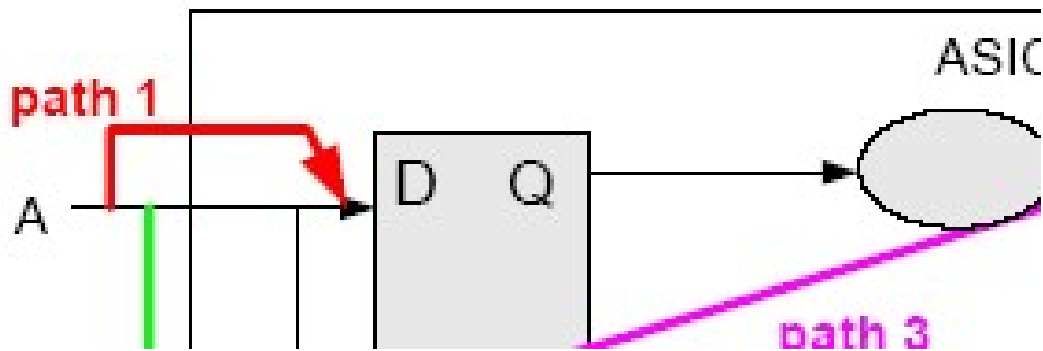
- Try an incremental compile. You can use the compile `–incremental_mapping` command to do only a gate-level optimization, making it faster than a normal compile. The design is not taken back to a GTECH level and no logic-level optimization is done, but Design Ware implementations might be changed.
- Use incremental compile with a high map effort (compile `–inc –map high`). Using this command activates additional algorithms that can be helpful.
- Use `set_critical_range` and compile `–inc`. If a critical range is used, Design Compiler works on more endpoints than just the most critical one. Therefore if the most critical endpoint can't be further improved, Design Compiler moves on to the next endpoint.

### **report\_timing**

The `report_timing` command is another useful tool for checking violations and timing of particular paths. If you look at all the options available for this command, they can be confusing. This section describes a few of the options and when you might use them. But before checking the options more closely, let's discuss timing paths and how delays are calculated.

### **Timing Paths**

Timing analysis involves breaking a design down into sets of timing paths then calculating the delay for each path. There are four types of timing paths as seen in the following figure.



**Figure 17**

The four paths are

- Path 1 – Primary input to D input of sequential cell.
- Path 2 – Primary input to primary output.
- Path 3 – Clock to D input of next sequential cell.
- Path 4 – Clock to primary output.

Notice that each path has a startpoint and endpoint. Valid startpoints are input ports and clock pins of flip-flops of designs. Valid endpoints are output ports and all input pins except clock pins of sequential devices.

### Delay Calculation

Timing reports are broken into two sections: one for calculating the data arrival time and the other for calculating the data required time. The end result, called slack, is the timing margin. A negative number indicates a violation.

The slack calculations for setup and hold checks are different. Here are the formulas:

- Setup slack = data required time – data arrival time
- Hold slack = data arrival time – data required time

What about calculations for data arrival and required time? There are four basic timing paths. The data arrival and required times are calculated differently, depending on the type of path being looked at. Let's take a look.

1. Path from a primary input port to a sequential device.

- a. Data arrival time = clock\_latency + input\_delay + cell/net\_delay
- b. Data setup required time = clock\_period + clock\_latency - clock\_uncertainty - cell\_setup\_time
- c. Data hold required time = clock\_latency + clock\_uncertainty + cell\_hold\_time

2. Path from a sequential device to a sequential device.

- c. Data arrival time = clock\_latency + cell/net\_delay

- d.  $\text{Data setup required time} = \text{clock\_period} + \text{clock\_latency} - \text{clock\_uncertainty} - \text{cell\_setup\_time}$
  - e.  $\text{Data hold requirement time} = \text{clock\_latency} + \text{clock\_uncertainty} + \text{cell\_hold\_time}$
3. Path from a sequential device to a primary output port.
- a.  $\text{Data arrival time} = \text{clock\_latency} + \text{cell/net\_delay}$
  - b.  $\text{Data setup required time} = \text{clock\_period} + \text{clock\_latency} - \text{clock\_uncertainty} - \text{output\_delay}$
  - c.  $\text{Data hold required time} = \text{clock\_latency} + \text{clock\_uncertainty} - \text{output\_delay}$
4. Path from a primary input port to a primary output port.
- a.  $\text{Data arrival time} = \text{clock\_latency} + \text{input\_delay} + \text{cell/net\_delay}$
  - b.  $\text{Data setup required time} = \text{clock\_period} + \text{clock\_latency} - \text{clock\_uncertainty} - \text{output\_delay}$
  - c.  $\text{Data hold required time} = \text{clock\_latency} + \text{clock\_uncertainty} - \text{output\_delay}$

### Default Options

Let's go through a report that is generated when no options are used. Notice that by default, a **full** path is reported, **max** delay is used, and **one** path is shown per path group. When you see delay max, then you know this report is for a setup check. If the delay was set to min, then you would be seeing a hold time report.

```
dc_shell-t> report_
```

```

```

```
Report : timing
```

```
-path full
```

```
delay max
```

```
Operating Conditions:
```

```
Wire Load Model Mode: top
```

```
Startpoint: out2 reg (rising edge-tri
```

| Des/Clust/Port | Wire Lo |
|----------------|---------|
|----------------|---------|

|     |       |
|-----|-------|
| dff | 05x05 |
|-----|-------|

| Point |
|-------|
|-------|

|                             |
|-----------------------------|
| clock clk1 (rise edge)      |
| clock network delay (ideal) |
| out2_reg/CP (FD1S)          |
| out2_reg/Q (FD1S)           |
| out2 (out)                  |

From this report you can find out information about what Operating Conditions are being used, which Library is used, and how the Wire Load Model Mode is set. You'll also want to observe the Startpoint, Endpoint, and Path Group. Remember valid startpoints are primary inputs and clock pins on sequential elements. Valid endpoints are all input pins except clock pins of sequential cells and primary output ports.

Here is how the slack was calculated. This path is from a sequential device to a primary output port, so the calculation is as follows.

1. Data arrival time = clock\_latency + cell/net\_delay



$$1.39 = 0.0 + 1.39$$

2. Data Setup required time = clock\_period + clock\_latency - clock\_uncertainty - output\_delay

$$7.00 = 10.00 + 0.00 - 1.00 - 2.00$$

3. Setup Slack = data required time – data arrival time

$$5.61 = 7.00 - 1.39$$

Where did the **clock uncertainty (-1.00)** and **output external delay (-2.00)** come from? These constraints are set by using the set\_clock\_uncertainty and set\_output\_delay commands.

### Options –to and -from

The –to and –from options are quite handy when you know you want timing to a particular start or end point. When these options are used, the report will contain only the paths to the named pins, ports, or clocks. The default behavior is to report the longest path to an output port if the design doesn't have constraints. If the design does have timing constraints, then the default is to report the path with the worst slack within each path group.

```
dc_shell-t> report_tin
```

```
-path full
-delay max
-max_paths 1
```

Design : dff

Version: 2003.06-2

Date : Wed Aug 20 13:15:28 2003

\*\*\*\*\*

Operating Conditions:

Wire Load Model Mode: top

Startpoint: in1 (input port clocked by

Endpoint: out1\_reg (rising edge-trig

Path Group: clk1

Path Type: max

Des/Clust/Port Wire Load Model

---

|     |       |       |
|-----|-------|-------|
| dff | 05x05 | class |
|-----|-------|-------|

|       |     |
|-------|-----|
| Point | Inc |
|-------|-----|

---

The endpoint in the above timing diagram is the endpoint specified in the report\_timing command.

### **-path full\_clock**

When you want to see the clock network, use the -path full\_clock option. When you use this option, you'll see the full clock paths for propagated clocks. To see the path, you need to propagate the clocks first before using this option. Here is an example.

```
dc_shell-t> set_propagated_clock [!
```

```
dc_shell-t> report_timing -path full_
```

```

```

Report : timing

-path full\_clock

-delay max

-max\_paths 1

Design : dff

Version: 2003.06-2

Date : Wed Aug 20 13:23:41 2003

```

```

Operating Conditions:

Wire Load Model Mode: top

Startpoint: out2\_reg (rising edge-tr

Endpoint: out2 (output port clocke

Path Group: clk1

Path Type: max

clock uncertainty

output external delay

data required time

-----

The `set_propagated_clock` command tells Design Compiler to propagate the delays through the clock network to determine latency at register clock pins. By looking at this report, you can see that the additional paths through buffers `U8/Z (IBUF2)` and `U2/Z (IBUF1)` have been included. You'll want to use this command after layout when your clock tree has been inserted.

You can tell this report is for a full clock by looking at the header `(-path full_clock)`. Notice that propagated is called out in parentheses beside clock network delay: `clock network delay (propagated)`. This tells you that the clock has been propagated instead of using ideal clock. If the `set_propagated_clock` command has not been used, ideal clocking is assumed. Ideal clocking means that clock networks have a specified latency (from the `set_clock_latency` command) or zero latency (by default).

### **-input\_pins**

This useful option allows you to look at cell and net delays separately. Although using the `-input_pins` option can result in long reports, it is a good way to ensure that you have the correct pins when you use `report_delay_calculation` to collect additional information. The `-input_pins` option tells Design Compiler to show input pins in the path report (the default is to show only output pins). In addition, this option shows the delays of the nets connected to these pins. For example,

```
dc_shell-t> report_timing -t
```

```

```

```
Report : timing
```

```
 -path full
```

```
Version: 2003.06-2
```

```
Date : Wed Aug 20 13:17:04 2003
```

```

```

```
Operating Conditions:
```

```
Wire Load Model Mode: ton
```

| Des/Clust/Port | Wire Lc |
|----------------|---------|
| dff            | 05x05   |

Point

---

```

clock clk1 (rise edge)
clock network delay (ideal)
input external delay
in1 (in)
U6/A (IBUF2)
U6/Z (IBUF2)
out1_reg/D (FD2)
data arrival time

```

If you look at the header of the report, you can see that **-input\_pins** has been included. It is a good idea to check the header before analyzing a report so you know what you're looking at.

In this report, you can see that the input signal (in1 (in)) passes through a buffer (U6/A (IBUF2)), and you can see the pins on the buffer (U6/Z (IBUF2)) to the D pin of a flip-flop. How can this information be used? Let's assume the delay through the buffer is quite large and you want to know why. One way to find out is to use the `report_delay_calculation` command.

#### **report\_delay\_calculation**

The `report_delay_calculation` command displays the actual timing arc calculation delay for a cell or net. For example,

```
dc_shell-t> report_delay_calcu
```

```

```

```
Report : delay calculation
```

From pin: U6  
To pin: U6/

Operating Conditions:  
Wire Load Model Mode: top

| Design | Wire Load Mod |
|--------|---------------|
| dff    | 05x05         |

arc sense: |  
arc type: |  
Input net transition times: |

$$0.0523 * (1 + 0.39) / 1$$

---

Total 0

## Transition calculations

rise resistance \* (pin cap +

The report\_delay\_calculation report contains a lot of information. This document will address only some of the information. For more information, see the SolvNet articles available on the web.

You should verify that this is the correct report by checking the From pin and To pin entries. This is similar to a report\_timing report, because you can see which Operating Conditions, Wire Load Model, and Library are used.

You should verify that this is the correct report by checking the From pin and To pin entries. This is similar to a report\_timing report, because you can see which Operating Conditions, Wire Load Model, and Library are used. The next section covers information about the arc sense, arc type, and the input net transition times.

You'll find the information about how the delay was calculated in the computation sections. In the timing report, look at the delay that was used for the buffer:

U6/Z (IBUF2) 0.84 1.84 f

The f stands for falling edge. With this knowledge, look back at the report\_delay\_calculation report and find the Fall Delay computation section. Notice that the calculated value is 0.842697. This matches the number used in report timing.

The last sections describe how rise and fall Transition calculations are calculated.

Where does all this data come from? The answer is, your library. If you have a .lib file, you can take a look at it and find the information shown when using report\_delay\_calculation.

### -delay max\_rise

You should use -delay max\_rise/max\_fall/min\_rise/min\_fall to see a specific path leading to a required transition. This option tells Design Compiler the path type at the endpoint. If this option is not used, the default is max.

```
dc_shell-t> report_timing -delay max
```

```

```

```
Report : timing
```

```
 -path full
```

```
 -delay max_rise
```

```
 -max_paths 1
```

```
Design : dff
```

```
Version: 2003.06-2
```

```
Date : Wed Aug 20 13:18:21 2003
```

```

```

```
Operating Conditions:
```

```
Wire Load Model Mode: top
```

```
Startpoint: out2_reg (rising edge-tri
```

```
Endpoint: out2 (output port clocked
```

```
Path Group: clk1
```

```
Path Type: max
```

```
Des/Clust/Port Wire Load Model
```

```

```

```

```

Notice again the header. This time a **max\_rise** path is used. This report tells you what the maximum path is for a rising edge to the end point. You can see the **r** in the path.

#### **-net/-cap/-tran**

Sometimes you need information on nets. You can get this information by using the **-net/-cap/-tran** options. **-nets**: This option tells Design Compiler to show nets in the path



report. If you want to see the delays for the nets, use the -input\_pins option as well. -  
transition\_time: This option shows the net transition time for each driving pin in the path.  
-capacitance: Use this option when you want to see the total (lump) capacitance in the  
report.

\*\*\*\*\*

Report : timing

-path full

-delay max

-nets

-max\_paths 1

-transition\_time

-capacitance

Design : dff

Version: 2003.06-2

Date : Wed Aug 20 13:20:30 2003

\*\*\*\*\*

|     |       |       |
|-----|-------|-------|
| dff | 05x05 | class |
|-----|-------|-------|

|       |        |     |
|-------|--------|-----|
| Point | Fanout | Cap |
|-------|--------|-----|

-----

|                        |  |  |
|------------------------|--|--|
| clock clk1 (rise edge) |  |  |
|------------------------|--|--|

|                             |  |  |
|-----------------------------|--|--|
| clock network delay (ideal) |  |  |
|-----------------------------|--|--|

|                    |  |  |
|--------------------|--|--|
| out2_reg/CP (FD1S) |  |  |
|--------------------|--|--|

|                   |  |  |
|-------------------|--|--|
| out2_reg/Q (FD1S) |  |  |
|-------------------|--|--|

|            |   |      |
|------------|---|------|
| out2 (net) | 1 | 0.39 |
|------------|---|------|

|            |  |  |
|------------|--|--|
| out2 (out) |  |  |
|------------|--|--|

|                   |  |  |
|-------------------|--|--|
| data arrival time |  |  |
|-------------------|--|--|

Did you see the header? How do you read this report? I've aligned the columns, but in most cases they won't be aligned. Here are a couple of tips. The transition time is seen for each driving pin. The capacitance driven by the driver is displayed in a column preceding both incremental path delay and transition time (specified by `-transition_time`). Unless the `-nets` option is used, capacitance is printed on the lines with nets rather than on the lines with driver pins.

Look for the word net to see the nets in the report.

### **`-max_paths, -nworst`**

Use the `-max_paths` and `-nworst` options to collect more information on the worst paths. `-max_paths`: This option allows you to specify the number of paths to report per path group. The default is 1. `-nworst`: This option allows you to specify the number of paths to report per endpoint.

Notice the difference between these two options. One is for path groups and the other for endpoints. An endpoint is all input pins, except clock pins of D sequential devices or primary output ports. If you use `-nworst 2` you will get multiple reports to the same endpoint. For example,

```
dc_shell-t> report_timing -nworst 2
```

```

```

Report : timing

-path full

-delay max

**-nworst 2**

-max\_paths 2

Design : dff

Version: 2003.06-2

Date : Wed Aug 20 13:21:27 2003

```

```

Operating Conditions:

Wire Load Model Mode: top

Startpoint: out2\_reg (rising edge-tri

**Endpoint: out2 (output port clocked**

clock clk1 (rise edge)

clock network delay (ideal)

clock uncertainty

output external delay

data required time

slack (MET)

5.61

Startpoint: out2\_reg (rising edge-tri

**Endpoint: out2 (output port clocked**

| Des/Clust/Port | Wire Lc |
|----------------|---------|
|----------------|---------|

|     |       |
|-----|-------|
| dff | 05x05 |
|-----|-------|

| Point |
|-------|
|-------|

|                           |
|---------------------------|
| clock clk1 (rise edge)    |
| clock network delay (idea |
| out2_reg/CP (FD1S)        |
| out2_reg/Q (FD1S)         |
| out2 (out)                |
| data arrival time         |

Notice the same **endpoint** for both reports.

### Path Groups

Path groups are automatically created when the `create_clock` or `group_path` command is used. The default path group contains all paths not captured by a clock. You can use the `report_path_group` command to see which groups you have. Figure 5 shows how path groups are formed.

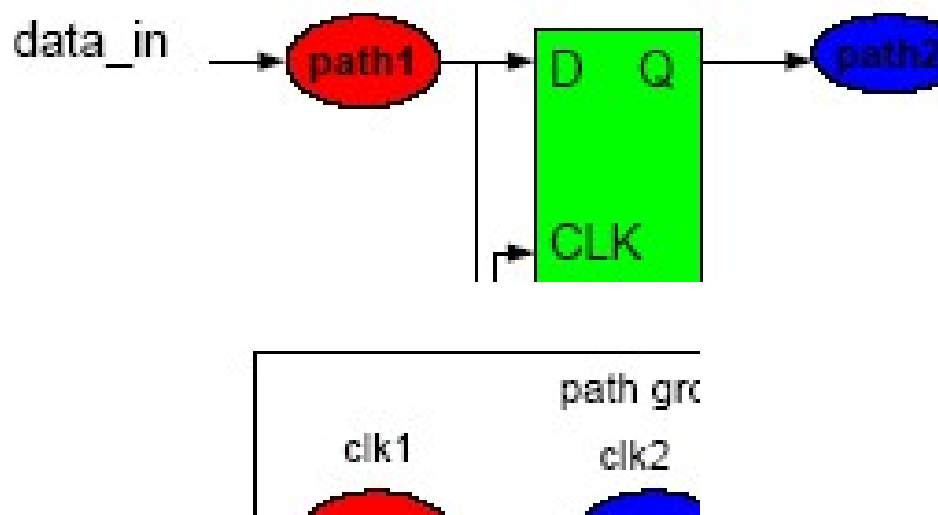


Figure 18

Here is an example of how to use command report\_path\_group:

```
dc_shell-t> report_p
```

```

```

```
Report : path_group
```

```
Design : dff
```

```
i. Versio
```

```
Date : Tue Aug 26
```

```

```

In this example, you can see two path groups, one for the default and one for a clock clk1. Now when you use `-max_paths 2`, you'll get paths for the path group. In this example, the path group is clk1.

```
dc_shell-t> report_timing -max_paths 2
```

\*\*\*\*\*

Report : timing

-path full

-delay max

-max\_paths 2

Design : dff

Version: 2003.06-2

Date : Tue Aug 26 08:10:24 2003

\*\*\*\*\*

Operating Conditions:

Wired Load Model Mode: top

Point

-----  
clock clk1 (rise edge)

clock network delay (idea

out2\_reg/CP (FD1S)

out2\_reg/Q (FD1S)

out2 (out)

data arrival time

clock clk1 (rise edge)

Startpoint: in2 (input port clocked b

Endpoint: out2\_reg (rising edge-trig

Path Type: max

| Des/Clust/Port | Wire Lc |
|----------------|---------|
| dff            | 05x05   |

Point

clock clk1 (rise edge)  
clock network delay (idea  
input external delay  
in2 (in)  
U7/Z (IBUF2)  
out2\_reg/D (FD1S)

## Important Output Files

Let's review. You've read in your HDL files and set your constraints, compile has completed, and you've started timing analysis. What's next? You need to write out a few files.

## Write Command

The write command has the following important options.

–format: Typically you'll want to write out a .db, Verilog, and/or VHDL format. The advantage of the Verilog or VHDL formats is that they allow you to look at the code and see what you have. The disadvantage is that none of the constraints are kept. If you use the .db format, you can't see the file but design constraints are saved. It is best to write out both.

–hierarchy: This option tells Design Compiler to write out all designs in the hierarchy. Don't forget this option.

–output: This option allows you to specify the file name for the output file.

After a compile, you should write a couple of files. You can then use these files in other tools such as PrimeTime. For example, write `-format verilog -hier -o design_names_gates.v` write `-format db -hier -o design_names_gates.db`

### **write\_test\_protocol**

If the plan is to next use TetraMAX, you need to use a variable named `test_stil_netlist_format` to specify what netlist format to use when you write out the stil file. Set this variable to match the netlist format of your design before issuing `write_test_protocol` so that the port names in the STIL protocol file match the port names in the netlist.

For example,

```
set test_stil_netlist_format verilog
write -format verilog -hier -out optimal_scan_stitched_netlist.v
write_test_protocol -format stil -out TMAX_protocol.spf
```

### **Conclusion**

Congratulations. You have just learned the ABCs of synthesis. A lot of information was covered. The next step is to review what you've learned and look more closely at all of the commands. By using the figures and examples presented in this tutorial, you should be able to compile your design, generate timing reports, and write out important files.