

Fakulti:	FAKULTI KEJURUTERAAN ELEKTRIK
Nama Matapelajaran: MAKMAL PBL TAHUN 3	Semakan : 1
Kod Matapelajaran : SKEL 3742	Tarikh Keluaran : 2020
	Pindaan Terakhir : 2020 5th May 2020
	No. Prosedur : PK-UTM-FKE-(0)-10



**SKEL
3742**

**SEKOLAH KEJURUTERAAN ELEKTRIK
UNIVERSITI TEKNOLOGI MALAYSIA
KAMPUS SKUDAI
JOHOR**

**VLSI SYSTEM DESIGN
LABORATORY (VLSI DESIGN)**

Pre – Lab : IC Design Flow

Prepared by : Mr. Izam bin Kamisian Dr. Muhammad Afiq Nurudin bin Hamzah Dr. Shahidatul Sadiyah binti Abdul Manan	Certified by : P.M. Dr. Ir. Rubita binti Sudirman (Head of ECE Department)
Signature :	Signature :
Stamp :	Stamp :
Date : 9 February 2020	Date : 9 February 2020

Task

In this lab, we will be using four Synopsys tools to design an integrated circuit (IC), which are:

1. Verilog Compiler and Simulator (VCS) for circuit design verification.
2. Design compiler (DC) for synthesis.
3. IC Compiler (ICC) for place and route.
4. Prime Time (PT) for static timing analysis (STA).

The complete design flow is shown in Figure 1 below:

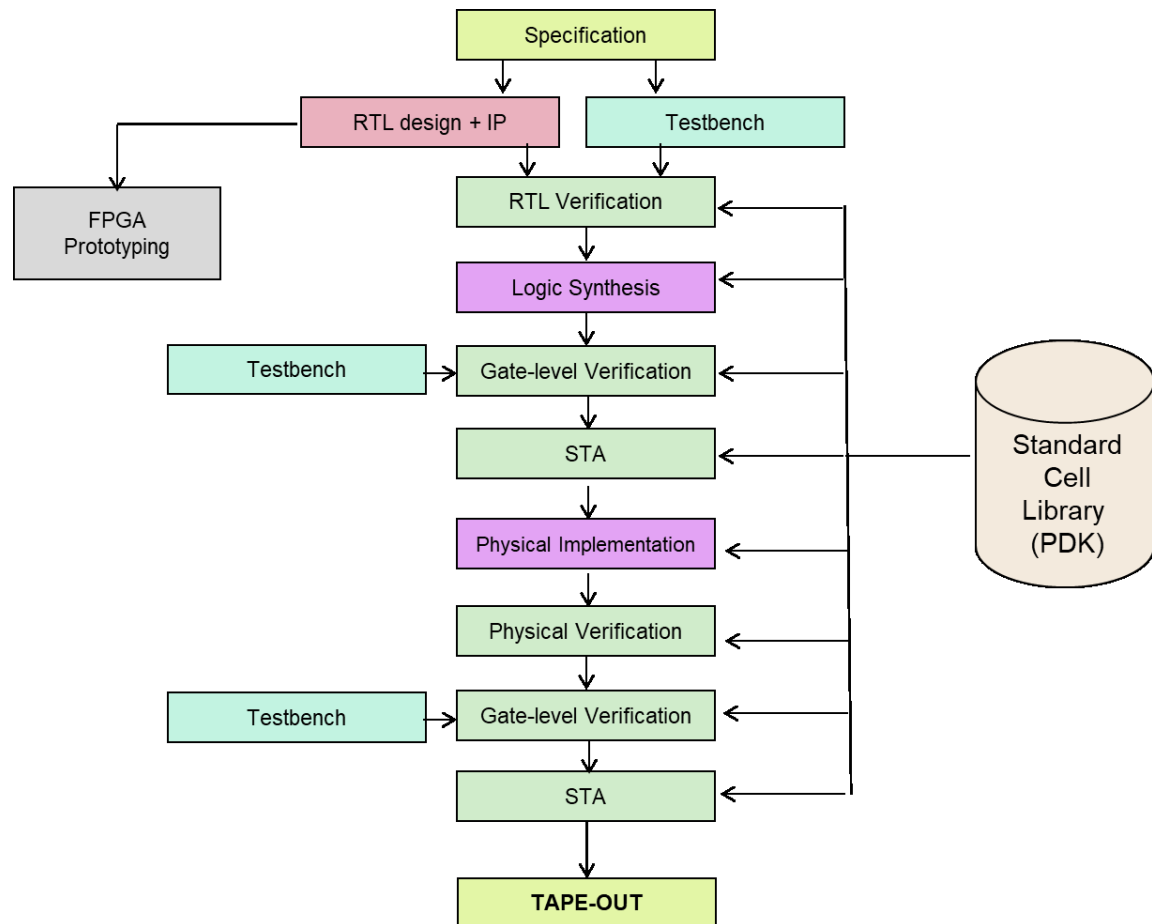


Figure 1: Complete IC Design Flow

For prelab, you are required to go through the tasks for all four tools before coming to the in-lab session and submit the answers for all the questions in the first lab session.

Verification Lab

Task 1

In this lab, you will run a basic verilog simulation using Synopsys VCS. This simulation exercise will use VCS's graphical user interface tool called DVE. The design used is a simple 4-bit adder.

Go to directory task1

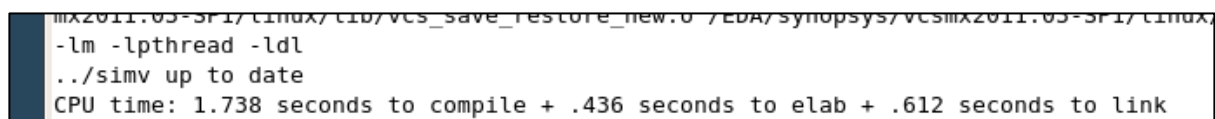
cd ~/lab_work/verification/task1

Use the "ls" command to view the files in this directory. You should find two files: adder.v (design file) and test_adder.v (testbench file)

Run VCS to check if your verilog files have any syntax errors:

vcs adder.v

On your screen, you will see the VCS copyright information, followed by various compile log information. Look for the message that says "../simv is up to date":

A terminal window showing the output of the 'vcs adder.v' command. The output includes the VCS version (mx2011.03-SP1), the path to the VCS installation, and the compilation options (-lm -lpthread -ldl). The key message is '../simv is up to date', indicating that the simulation files are current. The CPU time for compilation is 1.738 seconds, with 0.436 seconds for elaboration and 0.612 seconds for linking.

```
mx2011.03-SP1/linux/cib/vcs_save_restore_new.0 /EDA/Synopsys/VCS/mx2011.03-SP1/linux
-lm -lpthread -ldl
../simv up to date
CPU time: 1.738 seconds to compile + .436 seconds to elab + .612 seconds to link
```

This indicates your design does not have any syntax errors and is ready for simulation.

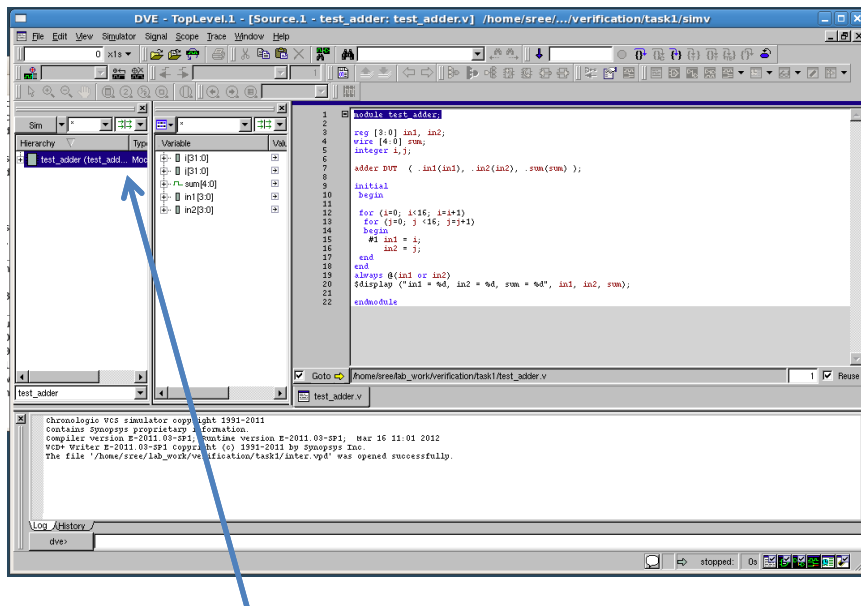
Now let's check the testbench. Since the testbench instantiates the adder, we need to compile both files:

vcs test_adder.v adder.v

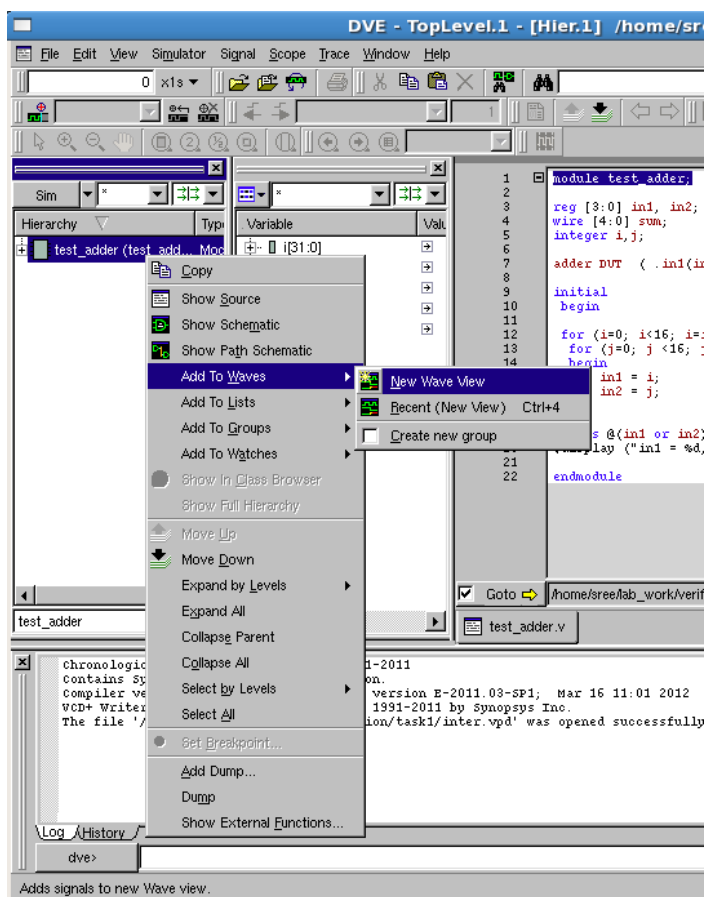
Did you get the message "../simv is up to date", if yes, then both files have no syntax errors. We will see how to fix syntax errors in the next lab. For now, let's run the simulation. To simulate the design:

vcs -R -gui -debug_all test_adder.v adder.v

The "-R" option tells VCS to run the simulation after compilation, while the "-gui" option invokes DVE, the graphical interface. The debug_all option allows for source tracing. Wait for DVE gui to appear. You will get the following window:



Right click on the Testbench module (test_adder) under the hierarchy pane, and select **Add to Waves - New Wave View** as in the figure below.



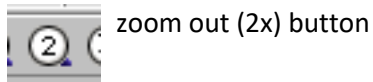
This will launch the waveform viewer, and load all the signals (ports, signals) into it. There is no waveform value yet, as the simulation has not been run.

Run the simulation by clicking on the **RUN** button

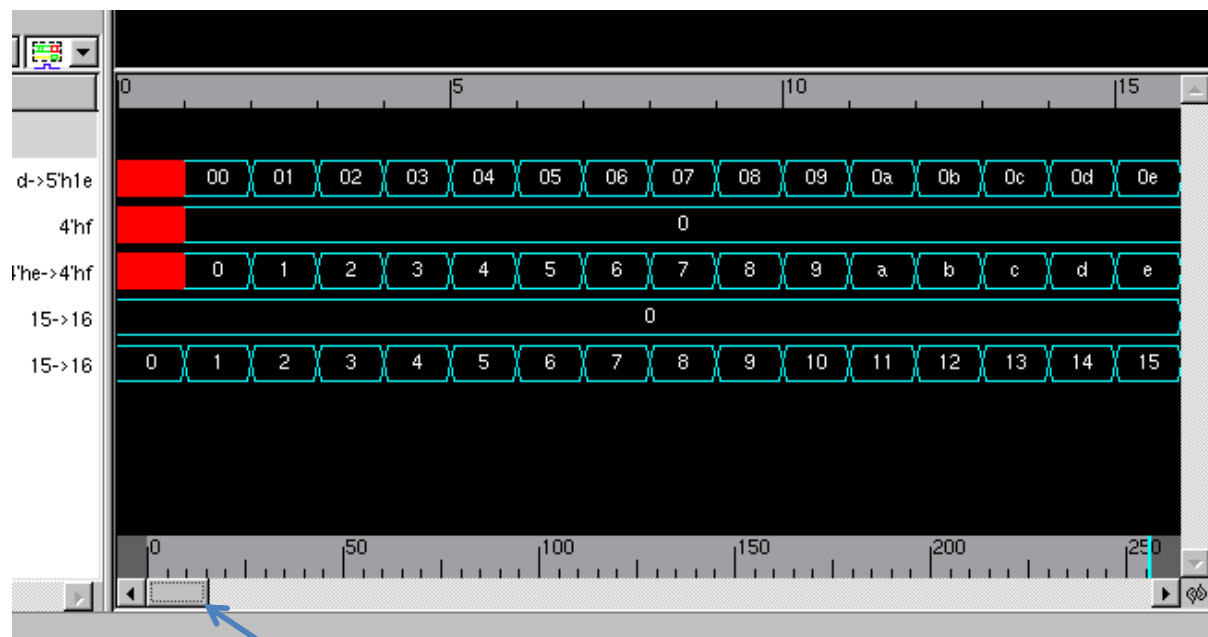


You can click on the run button on either the waveform window or the main DVE gui - both perform the same operation. This action will cause the simulation to run until completion. You will see the waveform displayed on the wave window.

Use the zoom in button on the toolbar to get a closer look. **Click** on the **zoom in button** a few times.



Then using the **scroll bar** at the bottom of the wave window, **scroll** to the beginning of the waveform.

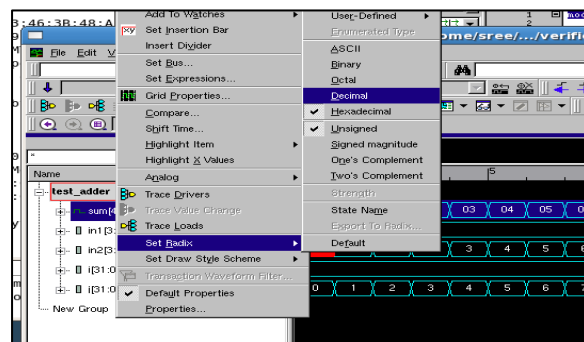


Scroll to the beginning. **Move** the **scroll bar** to this position.

Look at the waveform values. Do they match your expected value for an adder?

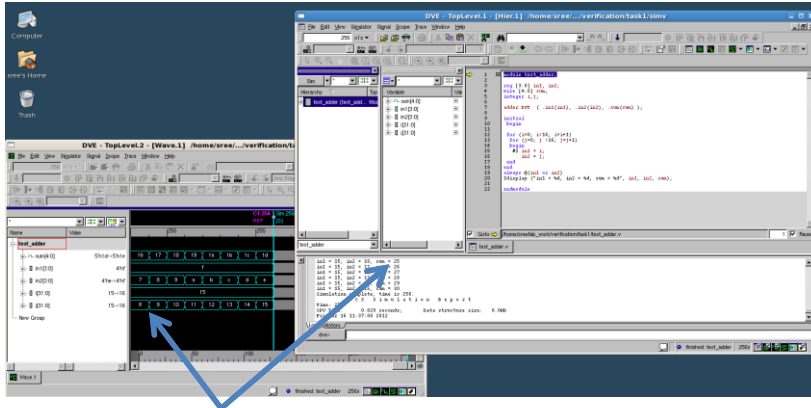
The values displayed for the bus signals (in1, in2, sum) is in hex. You can change the default radix to decimal (or binary or octal) by following this step:

Select the signal "**sum**". **Right click** on it, and select **Set Radix - Decimal**.



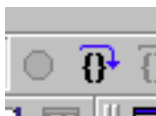
Repeat the above steps for the signal "in1" and "in2". Check the output value to see if its correct - it should be.

Next we will see how to use source code tracing. That is to see your code execution sequence. Arrange the windows so that you are able to see both of them, some overlap is fine.



Make sure you are able to see both windows.

On the main DVE gui, **click on Simulator - Start/Continue**. This brings the simulator back to time 0, the starting point. Notice that the waveform has cleared. Look at the source code window. Lets see what the execution sequence of your code is. Click on the **Next** button a few times:



The **yellow arrow** on the source code shows the current execution line. At the same notice that the waveform appears on the wave window. You may want to zoom in.

Continue clicking the Next button a few more times - do you see the sequence of your code execution? Based on your design knowhow, you should be able to deduce if the execution is correct - in this case it's correct. You can hit the Run button (the down arrow) to complete the simulation run.

This is how you verify a simple design. More complex designs will require advance verification techniques such as using assertions and native testbench constructs.

Now exit the simulator, **Left click on File**, then choose **Exit** and followed by **OK**.

This completes Task 1.

Task 2

In this lab, you will run a basic verilog simulation using Synopsys VCS. This simulation exercise will use VCS's shell (command line) interface. The same design from Task 1 is used here.

Go to directory task2

```
cd ~/lab_work/verification/task2
```

Use the "ls" command to view the files in this directory. You should find two files: adder.v (design file) and test_adder.v (testbench file)

Check the adder file for syntax errors. Type:

```
vcs adder.v
```

Did the design compile successfully? _____

VCS returns an error message like this:

```
Parsing design file 'adder.v'
Error-[SE] Syntax error
Following verilog source has syntax error :
"adder.v", 4: token is '['
  ouput [4:0] sum;
    ^
1 error
CPU time: .105 seconds to compile
```

This indicates a syntax error in your adder source file. The message "**adder.v**", **4**: indicates the file and line number where the error might have occurred. In general, the error might occur on the specified line or one line above. Can you tell what the error is? _____

The word "**output**" is spelt wrongly. It should be "**output**" and not "**ouput**".

Open the file, adder.v in a text editor (gedit), type:

```
gedit adder.v
```

Fix the error and **save** the file. **Exit** gedit when done.

Now recompile the design again:

```
vcs adder.v
```

The compilation should complete successfully with the message:

```
../simv up to date
```

Now check the testbench file:

```
vcs test_adder.v adder.v
```

Seems like more errors are detected. You should get the following message:

```

Error-[IBLHS-NRIRT] Illegal behavioral left hand side
test_adder.v, 15
  Non reg/integer/real/time/realtime cannot be used on the left hand side of
  this assignment
  The offending expression is : in1
  Source info: in1 = i;

Error-[IBLHS-NRIRT] Illegal behavioral left hand side
test_adder.v, 16
  Non reg/integer/real/time/realtime cannot be used on the left hand side of
  this assignment
  The offending expression is : in2
  Source info: in2 = j;

2 errors

```

The cause of this error is the testbench file, **test_adder.v**. Based on your knowledge of **verilog**, fix this error using the same method as above, then recompile the design.

Do not continue until you have fixed all errors, and compilation completes successfully.

The testbench contains self-checking constructs. Run the simulation using the command shell:

```
vcs test_adder.v adder.v -R -l run.log
```

The option "-l" that's small "l", captures all the output to a log (text file) called run.log. Once simulation completes, use gedit to open the run.log file:

```
gedit run.log
```

Browse through the file. Are the results correct? _____. It should be. Notice the message **"Simulation Completed Successfully"**. This is the message from the testbench indicating that the simulation run has completed with no **functional** errors i.e. NO BUGS!

Now we will create a functional error in the design adder. Functional errors are caught during simulation run, and the cause of the error is found during debug. Syntax errors are found during compilation.

Our design is an adder, but let's change the function to that of a subtractor. In the file, adder.v, change the "+" to a "-" sign:

from: assign sum = in1 + in2;

to: assign sum = in1 - in2;

Use **gedit** to make the above change and **save** the file. Now rerun the simulation:

```
vcs test_adder.v adder.v -R -l run.log
```

Once the simulation completes, open the log file, run.log with gedit, and look at the output. Notice all the error messages:

```

in1 = 15, in2 = 14, sum = 1
ERROR: SUM not equal to IN1 + IN2
in1 = 15, in2 = 15, sum = 0
ERROR: SUM not equal to IN1 + IN2
V C S   S i m u l a t i o n   R e p o r t

```


The above is a simple example of a self-checking testbench. Self-checking testbench is a better way to verify complex designs.

In this lab, you have seen how to fix syntax error and run the simulation using the shell (no GUI) by using self-checking testbench.

This completes VCS lab.

Design Compiler (DC) Synthesis Lab

Task 1

In this lab you will perform logic synthesis operation on a Verilog source code using Synopsys Design Compiler (DC). Logic synthesis optimizes and converts your RTL design into a Gate-level netlist.

Go to directory task1 under the lab_work/synthesis/task1 directory:

```
cd ~/lab_work/synthesis/task1
```

List down all the directories/files under the current directory:

```
ls
```

The directory **lib** contains the standard cell technology library required for synthesis. The **source** directory contains the Verilog source files that will be synthesized. Note that for synthesis, we don't require the testbench.

1. DC Setup

Before we can start using DC, we will need to create a setup file that specifies the standard cell library that we are going to use and its location. Using gedit, create a file called **".synopsys_dc.setup"** (note: there is a "." at the beginning of the file name).

```
gedit .synopsys_dc.setup
```

Add the following library settings to the file:

```
set search_path      "$search_path ./lib"  
set target_library   "saed90nm_max_pg.db"  
set link_library     " * $target_library"  
set symbol_library   "generic.sdb"
```

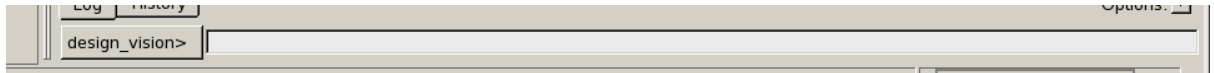
Save the ".synopsys_dc.setup" and **exit** gedit.

2. Read & Link Design

Launch the DC GUI, Design Vision (DV), at the terminal type:

design_vision

Locate the DV command prompt at the bottom of the DV window:



Here you can type in any DC commands. We will now verify that your library settings in the ".synopsys_dc.setup" file was applied correctly. Type:

printvar search_path

This will print to the screen all the search path that DC will look into to find files and libraries. Ignore the first few paths - they are the system wide default path which you should not modify. Look towards the end - do you see the library path: **./lib** ?

If yes, proceed to the next step, else quit DV (File - Exit - OK), and change the search_path setting in the ".synopsys_dc.setup" file as per the specification in "1. DC Setup".

Type:

printvar target_library

You should get:

target_library = "saed90nm_max_pg.db"

if you did not get the above setting, quit DV (File - Exit - OK), and change the target_library setting in the ".synopsys_dc.setup" file as per the specification in "1. DC Setup".

Type

printvar link_library

You should get:

link_library = " * saed90nm_max_pg.db"

if you did not get the above setting, quit DV (File - Exit - OK), and change the link_library setting in the ".synopsys_dc.setup" file as per the specification in "1. DC Setup".

Finally type:

printvar symbol_library

You should get:

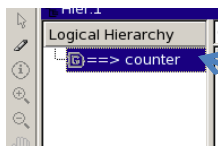
symbol_library = "generic.sdb"

if you did not get the above setting, quit DV (File - Exit - OK), and change the symbol_library setting in the ".synopsys_dc.setup" file as per the specification in "1. DC Setup".

If you got the results as above, your library setting is correct, we can then proceed to read in the design. Click on the **Read** icon:



Double Click on the **Source** directory, and select the file, **counter.v** and then click on **Open**. This will load the RTL design into DC's memory. In the **Logical Hierarchy** pane, **select** the design, **counter**:



Select this design (when selected, its highlighted!)

To view the block diagram of the design, click on the "Create Schematic of Selected Objects"



toolbar icon:

You can also **right-click** on the "counter" design in the **Logical Hierarchy** pane and select **Schematic View**. Press the "F" key to zoom fit the view or use the zoom icons on the toolbar. This will show the block diagram, with the input ports on the left, and output ports on the right. Can you identify the 4 input and one output port?

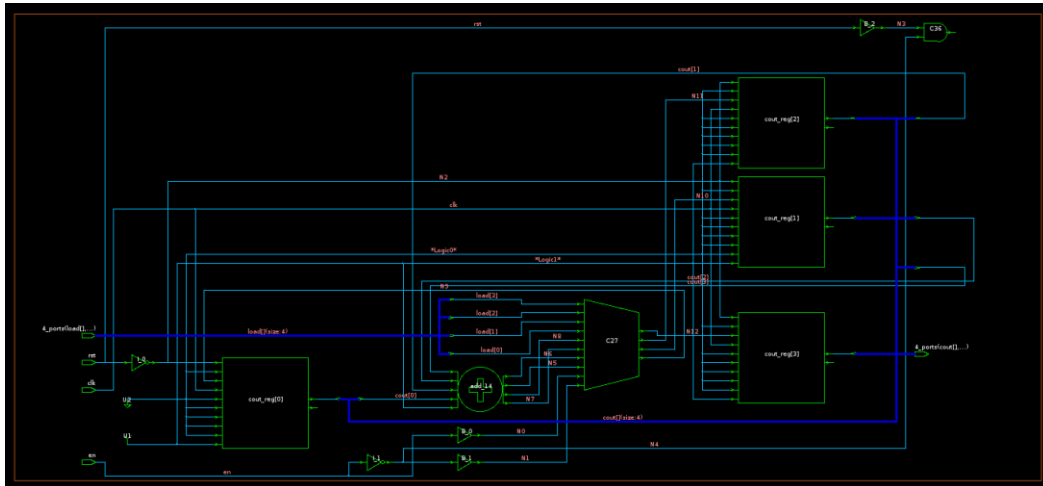
Block View:



What is the size (number of bits) of the output port? _____

You can also get the schematic view of the RTL code – double click on the Block Diagram view of the Counter design

This is view of the unmapped RTL netlist:



3. Design Constraints

Next step is to apply design constraints. For this design we will apply the following sets of constraints:

1. Get the smallest possible design
2. Clock period of 5ns
3. Clock transition of 0.5ns
4. Input delay of 1ns
5. Output delay of 1ns
6. Output load of 0.001pf
7. Input transition (for all inputs except clock): 0.8ns

The constraints are written using TCL syntax. Type the following in DV command prompt:

```
set_max_area 0  
  
create_clock -period 5 [get_port clk]  
  
set_clock_transition 0.5 [get_clock clk]  
  
set_input_delay 1 -clock clk [get_port "rst en load"]  
  
set_output_delay 1 -clock clk [get_port cout]  
  
set_load 0.001 [get_port cout]  
  
set_input_transition 0.8 [get_port "rst en load"]
```

When you type in any of the above command, DV will return a value of "1" like this:

```

design_vision> set_max_area    0
1
design_vision> create_clock    -period 5    [get_port    clk]
1
design_vision> set_clock_transition    0.5    [get_clock    clk]
1

```

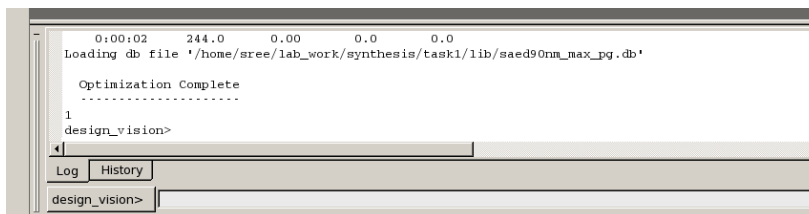
A return value of "1" indicates the command as has been successfully accepted by DC. If you make any mistakes, just retype the command again.

Next step is to optimize the design.

4. Optimize Design

We will now optimize and map the design to cells from the target library. To do that, **click on Design - Compile Design - Ok**. Do not change any setting in the dialog box.

Wait until optimization completes. You will get the following message in DV's log window:



Notice how when the optimization completes the design hierarchy view is cleared. **Click on Hierarchy – New Logical Hierarchy View** to load the compiled (mapped) design.

Take a look at the design schematic again. **Right Click** on **counter** design under the Logical Hierarchy pane and choose **Schematic View**. **Double click** on the Block View to get the Schematic View. Notice how the schematic view has changed. The design is now composed of gates from your target library.

5. Save the Design

Once optimization has completed, we can save the design and other related data for the next stage in the IC design process (layout).

To save the netlist as binary (DDC) file, at DV command prompt, type:

```
write -f ddc -hierarchy -out ./netlist/counter.ddc
```

To save a Verilog netlist, type:

```
write -f verilog -hierarchy -out ./netlist/counter.v
```

We also need to write out our design constraints in a format called SDC:

```
write_sdc ./netlist/counter.sdc
```

Now we need to analyze the design to see if we have met our design constraints (area & timing). For that we need to generate reports.

6. Analyze Design

To get the area of your design, type: **report_area**

Take a look at the last two lines:

Total cell area: 236.838003

Total area: 244.004085

The total area includes the cell and wire(net) area. The numbers for your design **might vary** slightly!

To get a clock report, type: **report_clock**

What is the clock period? _____

Does it match your design constrain? _____. It should!

Type: **report_clock -skew**

What is the clock transition? _____. It should!

Generate a timing report, type: **report_timing**

You will get the following report:

```

*****
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : counter
Version: F-2011.09-SP3
Date   : Mon Mar 19 11:36:56 2012
*****

Operating Conditions: WORST   Library: saed90nm_max_pg
Wire Load Model Mode: enclosed

Startpoint: en (input port clocked by clk)
Endpoint:   cout_reg[0]
            (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type:  max

Des/Clust/Port  Wire Load Model  Library
-----
counter         8000             saed90nm_max_pg

Point           Incr            Path
-----
clock clk (rise edge)          0.00      0.00
clock network delay (ideal)    0.00      0.00
input external delay          1.00      1.00 r
en (in)                      0.00      1.00 r
U14/QN (INVX0)                 1.49      2.49 f
U6/Q (A022X1)                 1.19      3.68 f
cout_reg[0]/D (DFFARX1)       0.03      3.70 f
data arrival time                                3.70

clock clk (rise edge)          5.00      5.00
clock network delay (ideal)    0.00      5.00
cout_reg[0]/CLK (DFFARX1)      0.00      5.00 r
library setup time            -0.24      4.76
data required time                                4.76
-----
data required time                                4.76
data arrival time                               -3.70
-----
slack (MET)                        1.06

```

Launch Path

Capture Path

Take a look at the slack value. Your number maybe slightly different from the above (1.06). However it should be a positive value. A positive slack value (MET) indicates that you have met your timing constraints. A negative slack value (VIOLATED) indicates a timing constraint failure. A timing failure will require re-optimizing the design or changing the source code.

Quit DV, click on **File - Exit - OK**.

This completes task 1. Proceed to the next task.

Task 2

In this lab, you will perform synthesis using DC shell interface i.e. the most common way of doing synthesis. You will create a script file containing all the design constraints and another command script file to run the synthesis process in batch mode.

Perform all task under the task2 directory:

cd ~/lab_work/synthesis/task2

The ".synopsys_dc.setup" has already been created with the necessary library and path settings. To run in batch mode, we will need two files - constraints and command file. Lets create the constraints file first. Type the following:

gedit scripts/constraints.tcl

Add in the following constraints to the script file:

```
set_max_area 0

create_clock -period 5 [get_ports clk]

set_clock_transition 0.5 [get_clocks clk]

set_input_delay 1 -clock clk [get_ports "rst en load"]

set_output_delay 1 -clock clk [get_ports cout]

set_load 0.001 [get_ports cout]

set_input_transition 0.8 [get_ports "rst en load"]
```

These are similar constraints to task 1. **Save** the file and **exit gedit**. The file will be saved to the **scripts** directory. Check to see if the file has any syntax errors, at terminal, type:

dcprocheck scripts/constraints.tcl

If you get any errors, its due to typos, refer to the above constraints, and fix the errors. Make sure you get no errors before proceeding to the next step. If there are no errors, dcprocheck will return the following message:

```
Loading snps_tcl.pcx...
Loading syn.pcx...
scanning: /home/sree/lab_work/synthesis/task2/scripts/constraints.tcl
checking: /home/sree/lab_work/synthesis/task2/scripts/constraints.tcl
```

Now create a command file, which will read in the design, apply the constraints, compile the design, generate reports and exit the tool.

gedit scripts/run.tcl

(you can use gedit or any text editor of your choice)

Add in the following commands to the script file:

```
read_verilog source/counter.v  
  
link  
  
source scripts/constraints.tcl  
  
compile  
  
write -f ddc -hierarchy -out netlist/counter.ddc  
  
write -f Verilog -hierarchy -out netlist/counter.v  
  
write_sdc netlist/counter.sdc  
  
redirect -file reports/area.rpt { report_area }  
  
redirect -file reports/timing.rpt { report_timing }  
  
redirect -file reports/clock.rpt { report_clock }  
  
redirect -file -append reports/clock.rpt { report_clock -skew }  
  
exit
```

Save the file and **exit gedit**. The file will be saved to the **scripts** directory. Check to see if the file has any syntax errors, at terminal, type:

```
dcprocheck scripts/run.tcl
```

Script commands summary:

- `read_verilog/link`: Read in the design, and checks to ensure design is loaded properly
- `source`: read in constraints file, and applies the constraints to the current design
- `compile`: optimizes and maps the design to the target technology
- `write`: saves design netlist
- `write_sdc`: saves design constraints file (SDC)
- `redirect -file`: redirects the specific command output to a file instead of the screen.

Now run the synthesis process in batch mode, type:

```
dc_shell -f scripts/run.tcl |tee dc.log
```

Note: `|` is the pipe character (typically above the return/enter key).

The above command will invoke `dc_shell`, which is the shell interface program for DC, and execute the commands in the file `run.tcl`. All the output to screen is also captured in the log file `dc.log`.

Once the synthesis process completes, use `gedit` to view the file, `dc.log`:

`gedit dc.log`

Check to see if log contains any "**Error**". If you have, fix it - its probably caused by typos.

Then use **`gedit`** to view all the report files in the **`reports`** directory.

What is the total area: _____

What is the clock transition time? _____

Is the timing slack positive or negative? _____

Did you meet your timing constraints? _____

This completes task 2. Continue to the next task.

Task 3

In this task, you will have to perform the synthesis task based on the given specification. You can use Design Vision or dc_shell to perform the synthesis operation.

Following are the specifications of the design:

Design name	detector
File name	source/detector.v
Target library	syn90nm_slow.db
Link library	* syn90nm_slow.db
Symbol library	generic.sdb
Library path	lib
Clock period	2ns
Input delay	0.2ns
Output delay	0.1ns
Clock transition time	0.01ns
Input signal transition time	0.1ns
Output capacitance	0.005pf
Area goal	smallest possible size

Based on the above specification, your task is to:

- Create a ".synopsys_dc.setup" file with the appropriate library and path settings.
- Create a constraints file with the name **detector_cons.tcl**.
- Create a command file with the name **run_detector.tcl** that does the following:
 - Reads in the design
 - Applies the above constraints via detector_cons.tcl
 - Optimizes the design
 - Generates a timing, area, **cell** and clock report which must be saved to the **reports** directory
 - Write out a **Verilog** netlist and save it to the **netlist** directory
 - Write out a DDC file and save it to the netlist directory
 - Generates **SDC** file and saves it to the **netlist** directory
 - **Exit** DC/DV when done.
 - All DC run commands and outputs must be capture in a log file.

When done, you will need to hand-over the following files to your lecturer:

- .synopsys_dc.setup
- run_detector.tcl
- detector_cons.tcl
- Area report
- Timing report
- Cell report
- Clock report
- Synthesis run log

Does your design meet its timing constraints? __yes__. What is the slack? __0.091__

This completes task 3. In the next lab, we will cover how to do layout (PnR).

Thank you

Physical Implementation Lab

In this lab you will perform physical implementation operation on a netlist using Synopsys IC Compiler (ICC). Commonly called place & route (or pnr), physical implementation task takes in a gate-level netlist and converts it into a physical layout that can be sent for IC fabrication.

Go to directory task1 under the **lab_work/pnr** directory:

```
cd ~/lab_work/pnr
```

List down all the directories/files under the current directory:

```
ls
```

The directory **ref** contains the standard cell technology library required by ICC. The **netlist** directory contains the gate-level netlist file (**ORC.ddc**) in **DDC** format. This file was generated by Design Compiler using the “write -f ddc ...” command. Do all your work from the **pnr** directory.

1. ICC Setup

Before we can start using ICC, we will need to create a setup file that specifies the standard cell library that we are going to use and its location. For this lab the setup file, **".synopsys_dc.setup"** (note: there is a "." at the beginning of the file name) is provided for you. This is the same file that is used by Design Compiler but now we have additional libraries specified in it.

Using gedit, view the contents fo the **".synopsys_dc.setup"** file. DO NOT MODIFY THIS FILE!

```
gedit .synopsys_dc.setup
```

Browse through the file to view the physical library i.e. Milkyway settings. When done, **exit** gedit.

2. Read Design

Launch ICC in shell mode, open a terminal and type:

```
cd ~/lab_work/pnr
icc_shell
```

At the ICC command prompt, read in the DDC netlist from the netlist directory, type:

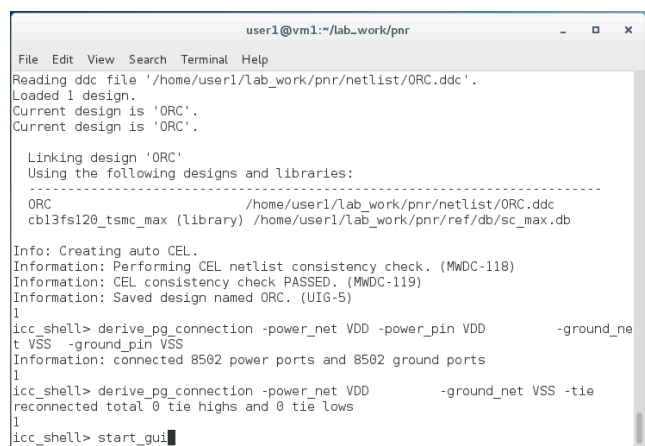
```
import_design -format ddc -top ORC ./netlist/ORC.ddc
```

Next, tie all the power pins of the gates in the netlist to the power nets, VDD and VSS:

```
derive_pg_connection -power_net VDD -power_pin VDD -ground_net VSS -ground_pin VSS
derive_pg_connection -power_net VDD -ground_net VSS -tie
```

For the next step, we will use the GUI, at the ICC prompt, type:

```
start_gui
```

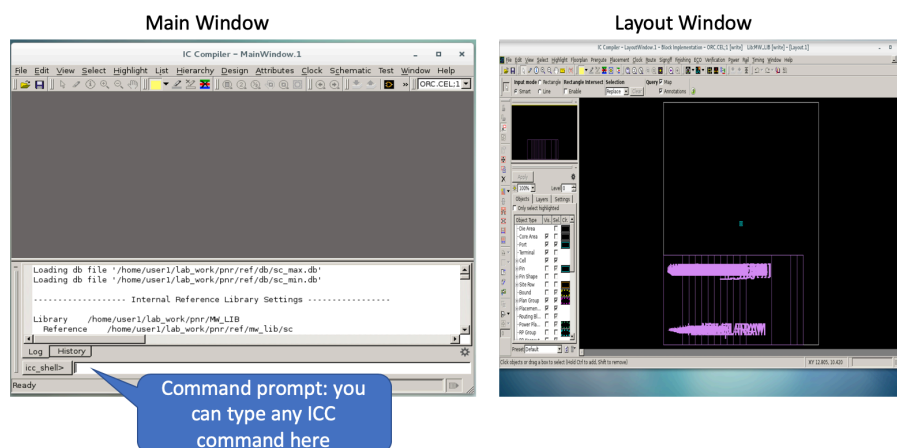


```
user1@vm1:~/lab_work/pnr
File Edit View Search Terminal Help
Reading ddc file '/home/user1/lab_work/pnr/netlist/ORC.ddc'.
Loaded 1 design.
Current design is 'ORC'.
Current design is 'ORC'.

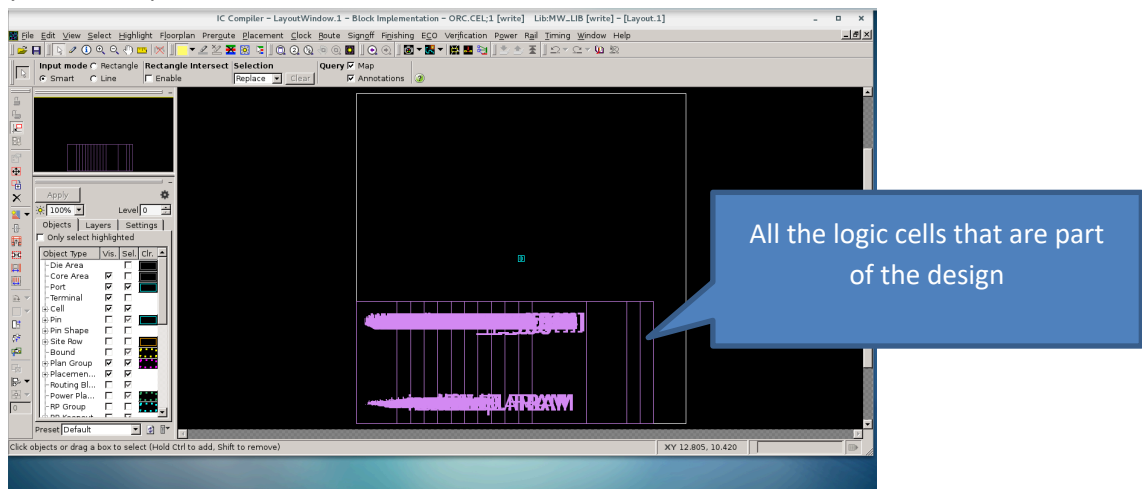
Linking design 'ORC'
Using the following designs and libraries:
-----
ORC /home/user1/lab_work/pnr/netlist/ORC.ddc
cb13fsl20_tsmc_max (library) /home/user1/lab_work/pnr/ref/db/sc_max.db

Info: Creating auto CEL.
Information: Performing CEL netlist consistency check. (MwDC-118)
Information: CEL consistency check PASSED. (MwDC-119)
Information: Saved design named ORC. (UIG-5)
1
icc_shell> derive_pg_connection -power_net VDD -power_pin VDD -ground_net VSS -ground_pin VSS
Information: connected 8502 power ports and 8502 ground ports
1
icc_shell> derive_pg_connection -power_net VDD -ground_net VSS -tie
reconnected total 0 tie highs and 0 tie lows
1
icc_shell> start_gui
```

You will get the ICC GUI. There will be two windows – ICC Main window and Layout window.



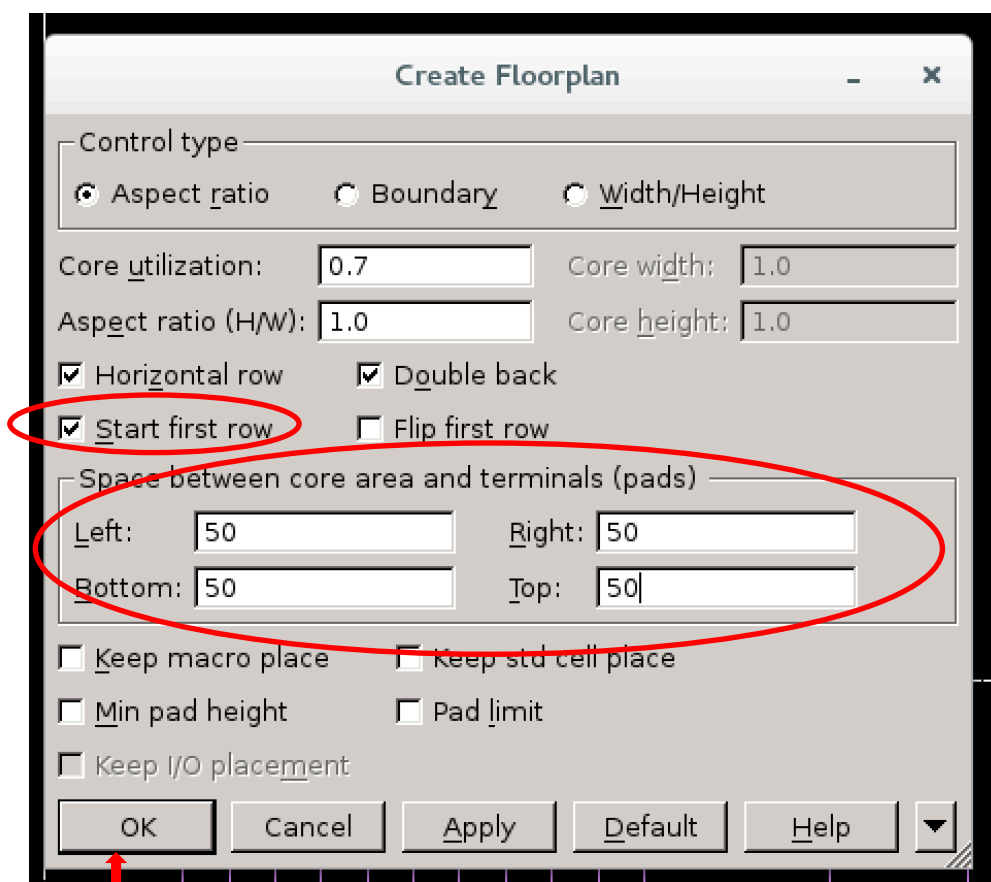
You will do your work from the Layout window. Make that window the active window on your desktop.



3. Floorplan

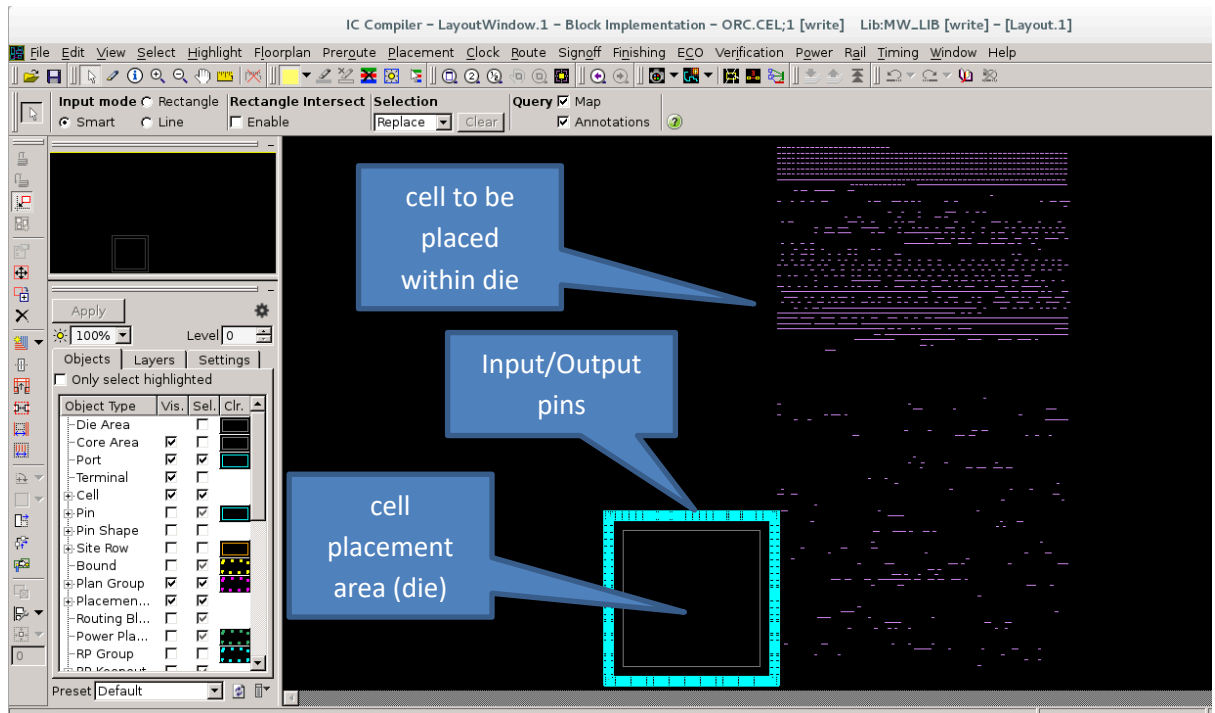
Next up is floorplan. Here is where we specify the die area, pin arrangement and power network.

From the Layout window menu bar, **Click Floorplan – Create Floorplan**. Specify the values as shown in the image below:

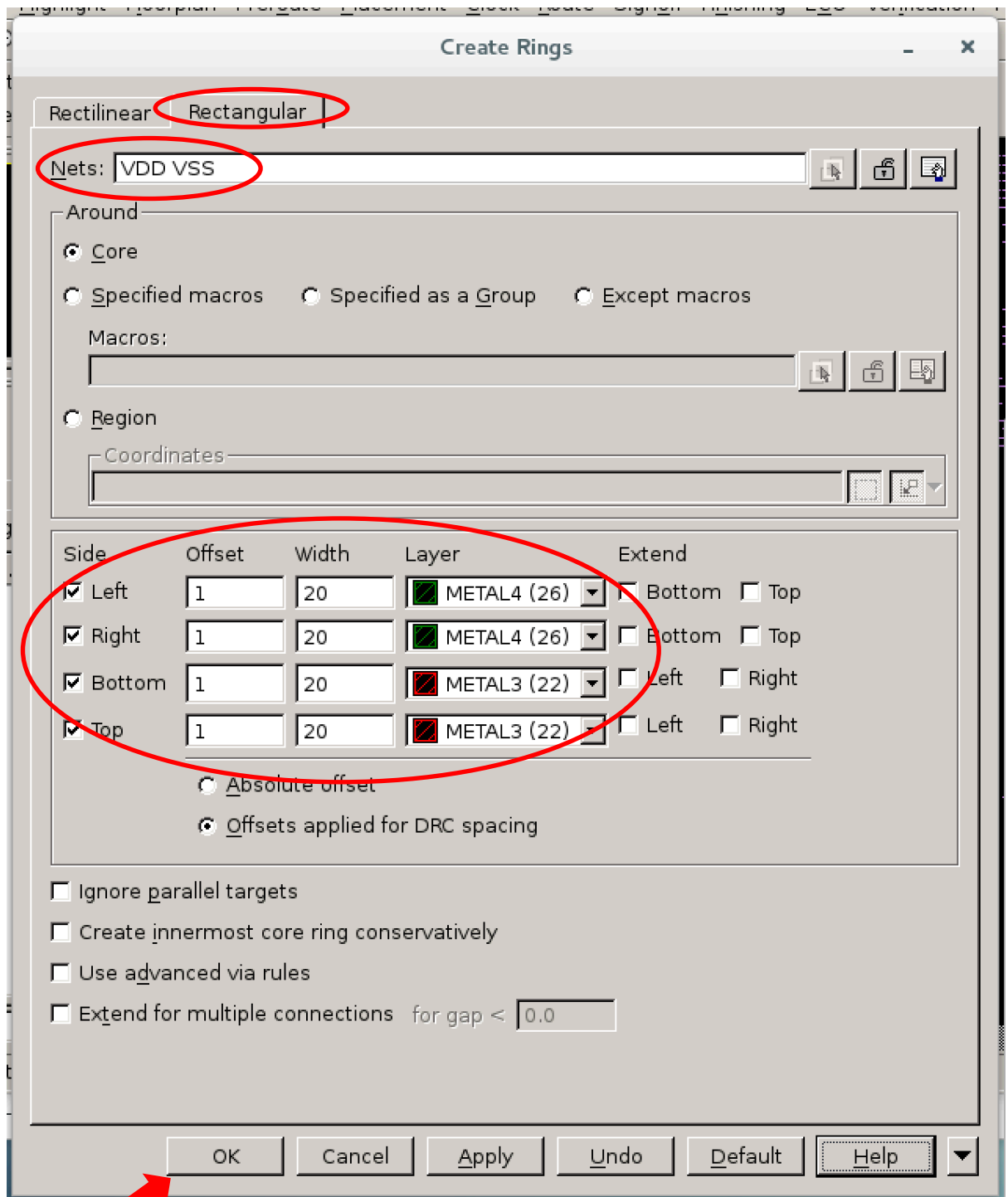


Click **OK**.

Your Layout window will look similar to this:

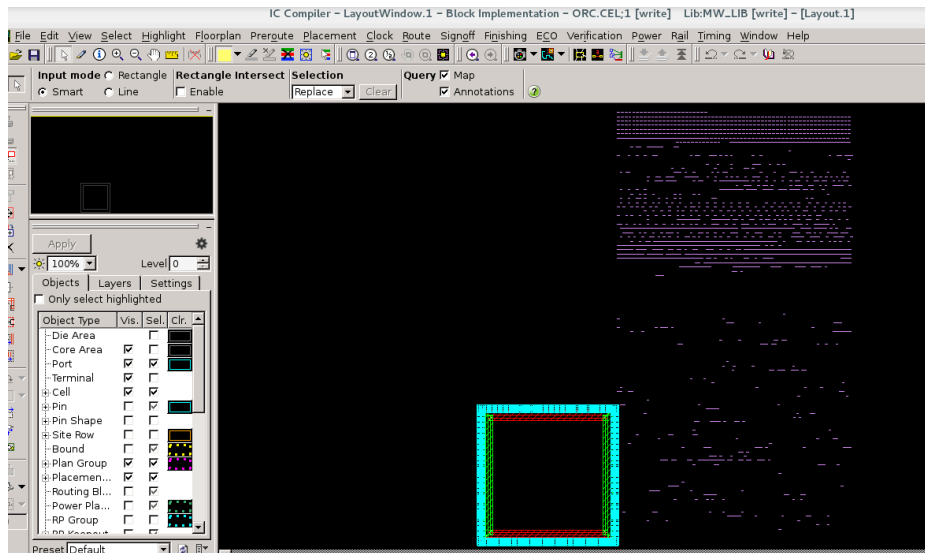


Before we can move all the cells in, we need to create the power (VDD) and ground (VSS) rings (supply network). Click **Preroute – Create Rings**. Key in the values as shown in the image below:



Click **OK**.

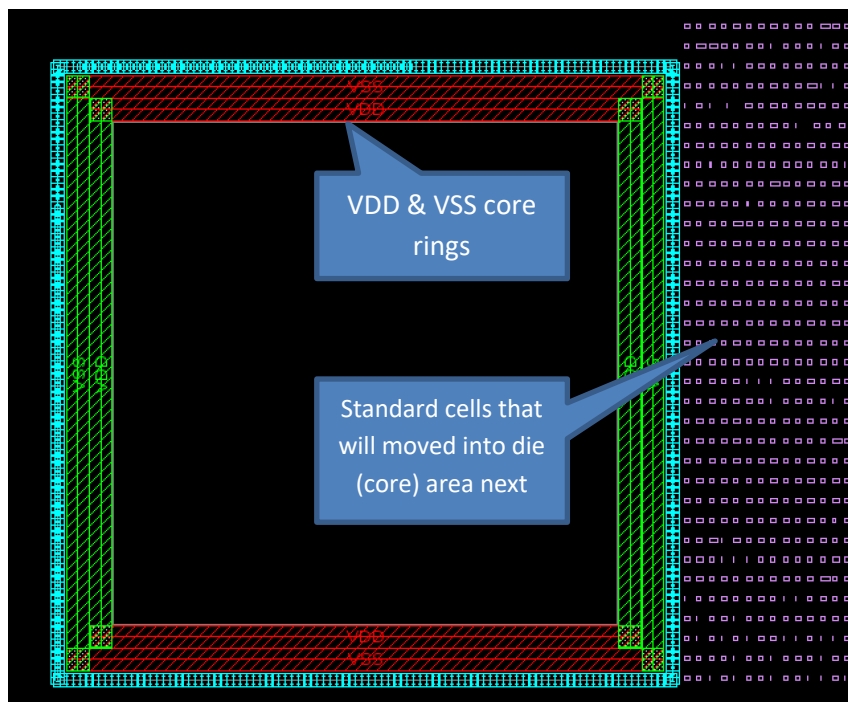
The Layout window should look like this:



Using the magnifier icon on the toolbar, zoom into the rectangle area to have a closer look.

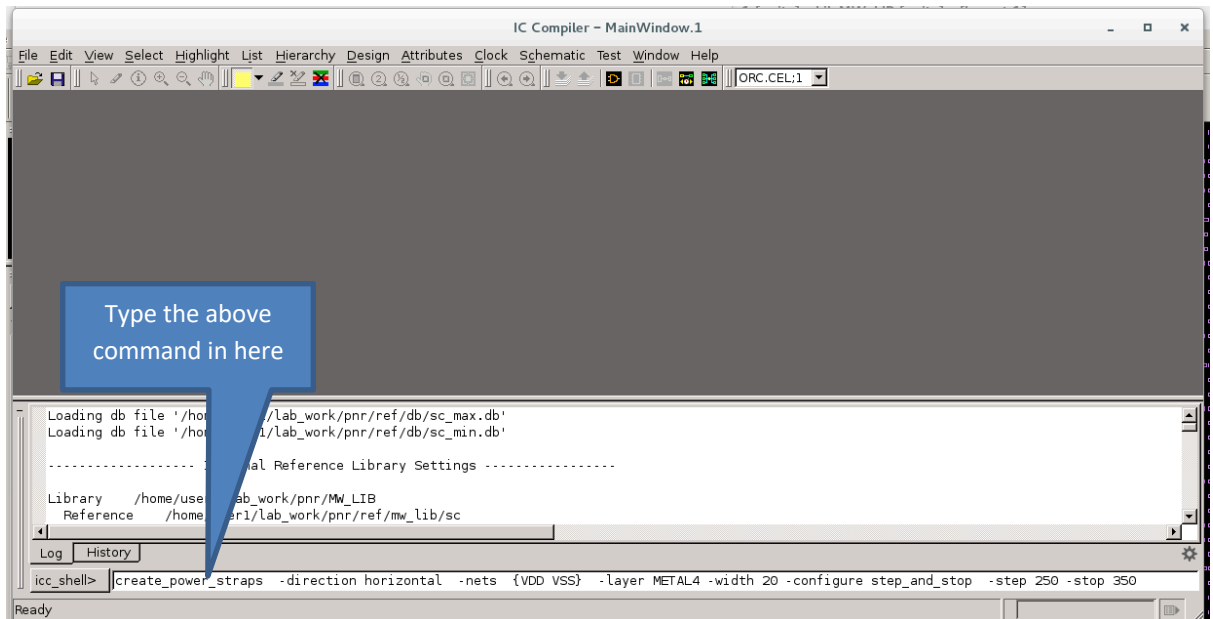


Click on zoom in tool, and draw a rectangle around the die area to zoom in. The zoomed in view is shown below:

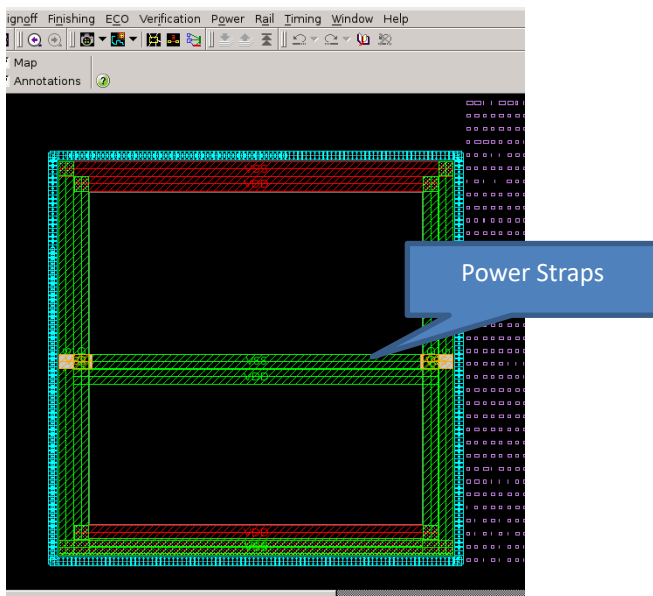


Once the rings have been created, we will create the power straps. Make the Main Window the active window now (bring it to the forefront). At the `icc_shell` prompt, type in the following command:

**create_power_straps -direction horizontal -nets {VDD VSS} -layer METAL4 -width 20
-configure step_and_stop -step 250 -stop 350**



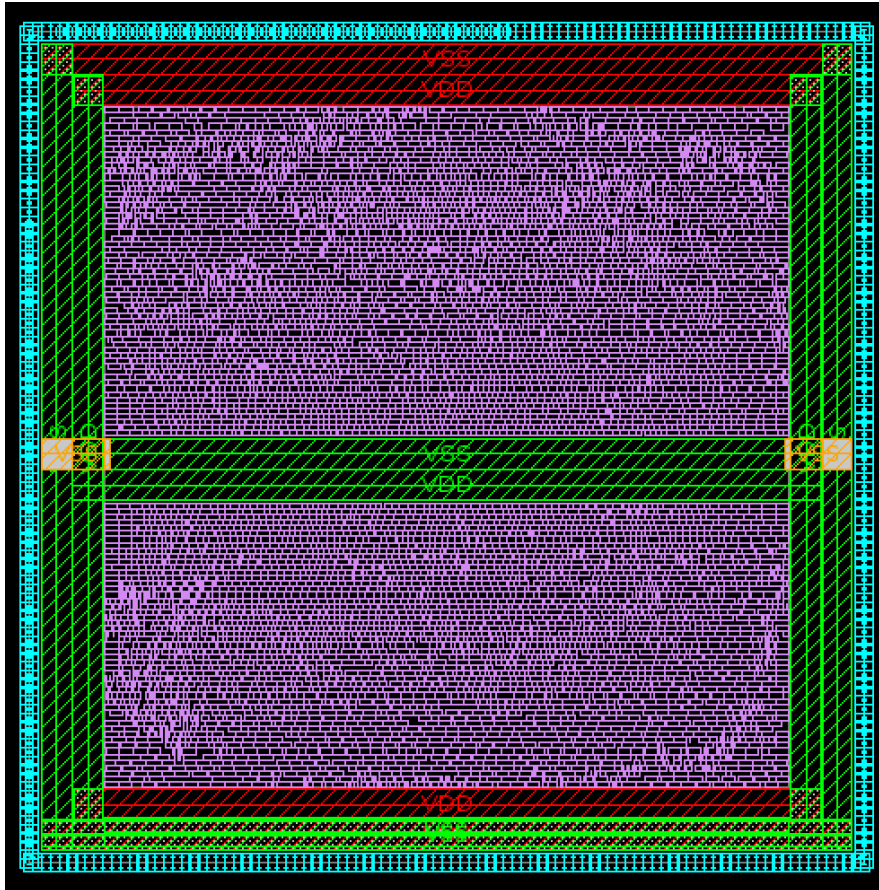
The above command will create the power straps on metal layer 4, as shown in the image below:



Back in the ICC prompt, key in the following commands:

**set_pnet_options -complete {METAL3 METAL4}
derive_pg_connection -power_net VDD -power_pin VDD -ground_net VSS -ground_pin VSS
derive_pg_connection -power_net VDD -ground_net VSS -tie
create_fp_placement -timing_driven -no_hierarchy_gravity**

This will move all the standard cells into the core area. See figure below:



Final step under floor-planning would be to route all the power nets (VDD, VSS), type the following command:

```
preroute_standard_cells -remove_floating_pieces
```

We are now ready for cell placement (within die) and optimization.

4. Placement

Lets optimize the placement of the cells within the die area, type the following commands:

```
set_separate_process_options -placement false
```

```
place_opt
```

```
derive_pg_connection -power_net VDD -power_pin VDD -ground_net VSS -ground_pin VSS
```

```
derive_pg_connection -power_net VDD -ground_net VSS -tie
```

5. Clock Tree Synthesis (CTS)

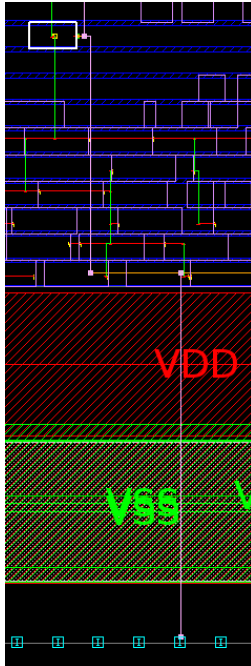
CTS builds a buffer tree for the clock network, to enable the clock signal to drive multiple flip-flops with degrading the signal strength. Type the following commands:

```
remove_clock_uncertainty [all_clocks]
```

```
clock_opt
```

The layout will look a little different now because of the adding in of clock buffers and routing of the clock nets. Try zoom/pan around the layout to view the clock routes.

The image below shows how the clock pin is routed to a clock buffer:

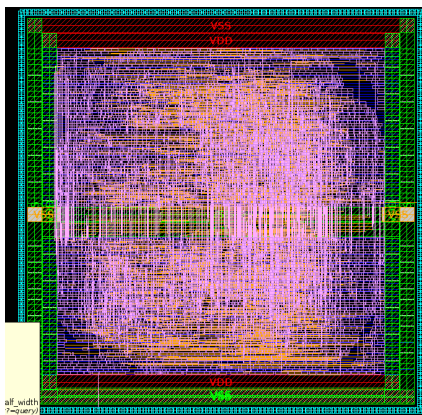


6. Routing

Final step in the physical design process is to route the design. Type:

route_opt

Once completed, your routed design should look like this:



Re-run the following two commands to ensure all power/ground connectivity for any new cells added in during the routing process:

```
derive_pg_connection -power_net VDD -power_pin VDD -ground_net VSS -ground_pin VSS
derive_pg_connection -power_net VDD -ground_net VSS -tie
```

Verify the design is free of any design rule violations (DRC), type:

verify_zrt_route

You should get the following output:

```
Verify Summary:
Total number of nets = 9933, of which 0 are not extracted
Total number of open nets = 0, of which 0 are frozen
Total number of excluded ports = 0 ports of 0 unplaced cells connected to 0 nets
                                0 ports without pins of 0 cells connected to 0
nets
                                0 ports of 0 cover cells connected to 0 non-pg
nets
Total number of DRCs = 0
Total number of antenna violations = no antenna rules defined
Total number of voltage-area violations = no voltage-areas defined
Total number of tie to rail violations = not checked
Total number of tie to rail directly violations = not checked
1
icc_shell>
```

This indicates no
DRC violations

Run a Layout vs. Schematic (LVS) check to ensure there are no shorts/open nets:

verify_lvs

The output should look like this:

```
** Total Floating ports are 0.
** Total Floating Nets are 0.
** Total SHORT Nets are 0.
** Total OPEN Nets are 0.
** Total Electrical Equivalent Error are 0.
** Total Must Joint Error are 0.

-- LVS END : --
Elapsed = 0:00:01, CPU = 0:00:00
Update error cell ...
1
```

This indicates no
LVS errors

Now let's check if our design meets all the timing constraints. But first we need to extract the parasitic in the layout (unwanted resistance and capacitance), type:

extract_rc

To run setup checks, type:

report_timing -delay max -path short

Look at the bottom of the report for the word **"slack"**:

clock network delay (propagated)	0.19	10.19
outY_reg[121]/CP (dfcrq1)	0.00	10.19 r
library setup time	-0.03	10.16
data required time		10.16

data required time		10.16
data arrival time		-10.16

slack (MET)		0.00

slack (MET) indicates you have met the setup timing constraints.

To run hold checks, type:

report_timing -delay min -path short

clock clk (rise edge)	0.00	0.00
clock network delay (propagated)	0.08	0.08
outY_reg[126]/CP (dfcrq1)	0.00	0.08 r
library hold time	0.01	0.09
data required time		0.09

data required time		0.09
data arrival time		-0.50

slack (MET)		0.41

Similarly, **slack (MET)** indicates you have met the hold timing constraints.

To get an overall status of your design including timing constraints, design rules, cell count, type:

report_qor

Note: you can ignore any max fanout violations.

Now its time to save all the data. Save your layout database in Milkyway format:

save_mw_cel -as ORC_routed

Save your post-layout netlist in DDC format:

write -f ddc -hier -out netlist/ORC_postlayout.ddc

Save your post-layout netlist in Verilog format

write_verilog -no_physical_only_cells -no_core_filler_cells ./netlist/ORC_postlayout.v

Save all the design constraints:

write_sdc ./netlist/ORC_postlayout.sdc

Save the parasitic information in SPEF format:

extract_rc

write_parasitics -output ./netlist/ORC_postlayout.spef

The Verilog netlist and SDC/SPEF file will be used by PrimeTime to perform Static Timing Analysis.

This completes ICC lab. Thank you

Static Timing Analysis (STA) using PrimeTime Lab

In this lab, you will perform Static Timing Analysis (STA) on a post-layout design using back-annotated parasitic. Post-layout design is a design which has gone through a place & route process. In this lab, you will use PrimeTime (PT) to perform timing analysis on design created by IC Compiler.

Once you have completed your design layout in IC Compiler, you will write out the following files:

1. Design netlist in ddc or Verilog format
2. Parasitic information in SPEF format
3. Design constraints in SDC format (Optional)

For this lab, we will only use the Verilog netlist and SPEF. We will specify the design constraints manually to PrimeTime.

Go to directory `lab_work/sta_lab/` :

```
cd ~/lab_work/sta_lab
```

List down all the directories/files under the current directory:

```
ls
```

The directory **lib** contains the standard cell technology library required for STA. The **netlist** directory contains the post-layout Verilog netlist files and SPEF file.

1. PT Setup

Before we can start using PT, we will need to create a setup file that specifies the standard cell library that we are going to use and its location. Using **gedit**, create a file called **".synopsys_pt.setup"** (note: there is a "." at the beginning of the file name).

```
gedit .synopsys_pt.setup
```

Add the following library settings to the file:

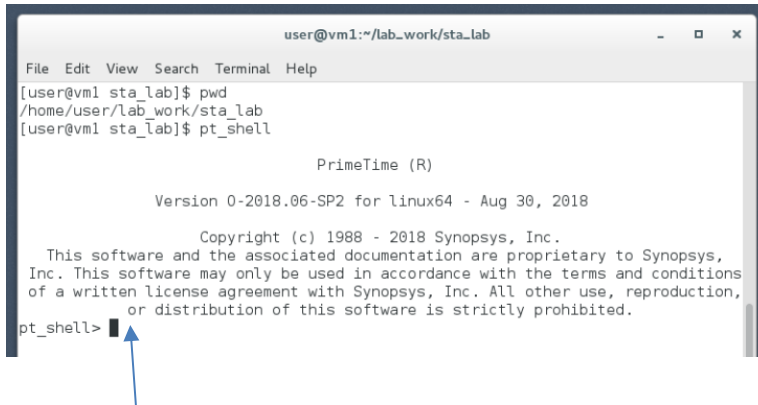
```
set search_path    "$search_path . ./lib"  
set link_path      " * sc_max.db"
```

Save the **".synopsys_pt.setup"** and **exit** gedit. Make sure you save the file in the **lab_work/sta_lab** directory

2. Read & Link Design

Launch PT in shell mode, at the terminal type:

pt_shell



```
user@vm1:~/lab_work/sta_lab
File Edit View Search Terminal Help
[user@vm1 sta_lab]$ pwd
/home/user/lab_work/sta_lab
[user@vm1 sta_lab]$ pt_shell

PrimeTime (R)

Version 0-2018.06-SP2 for linux64 - Aug 30, 2018

Copyright (c) 1988 - 2018 Synopsys, Inc.
This software and the associated documentation are proprietary to Synopsys,
Inc. This software may only be used in accordance with the terms and conditions
of a written license agreement with Synopsys, Inc. All other use, reproduction,
or distribution of this software is strictly prohibited.
pt_shell>
```

Here you can type in any PT commands. We will now verify that your library settings in the ".synopsys_pt.setup" file was applied correctly. Type:

printvar search_path

printvar link_path

Your output should look like this:



```
pt_shell> printvar link_path
link_path      = "* sc_max.db"
pt_shell> printvar search_path
search_path    = ". ./lib"
pt_shell>
```

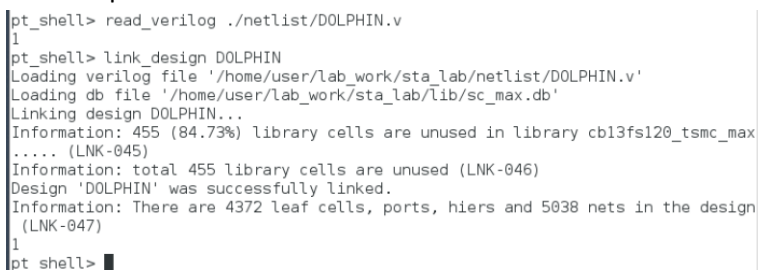
If yes, proceed to the next step, else quit PT (type: exit), and change the setting in the ".synopsys_pt.setup" file as per the specification given above. Then redo the above steps. Do not continue until your setting matches the above values.

Let's read in the post-layout Verilog netlist from the netlist directory, type:

read_verilog ./netlist/DOLPHIN.v

link_design DOLPHIN

Your output will look like this:



```
pt_shell> read_verilog ./netlist/DOLPHIN.v
1
pt_shell> link_design DOLPHIN
Loading verilog file '/home/user/lab_work/sta_lab/netlist/DOLPHIN.v'
Loading db file '/home/user/lab_work/sta_lab/lib/sc_max.db'
Linking design DOLPHIN...
Information: 455 (84.73%) library cells are unused in library cb13fs120_tsmc_max
..... (LNK-045)
Information: total 455 library cells are unused (LNK-046)
Design 'DOLPHIN' was successfully linked.
Information: There are 4372 leaf cells, ports, hiers and 5038 nets in the design
(LNK-047)
1
pt_shell>
```

The top design name is DOLPHIN. When you do “link_design DOLPHIN”, PT sets the current_design to DOLPHIN, and loads in all cells that make up the DOLPHIN design into PT’s memory. To verify the current design, type:

current_design

This should return the value, “**DOLPHIN**” as shown in the following figure:

```
pt_shell> current_design
{"DOLPHIN"}
pt_shell>
```

Since we will be doing hold analysis, we need to specify an additional library that models all the minimum cell delays, type:

set_min_library sc_max.db -min sc_min.db

3. Apply Design Constraints

Now that the design has been successfully loaded into PT, we can specify our timing constraints on the design. In a typical design flow, these will be the same set of constraints as what you would have specified in Design Compiler with some minor changes.

Before we specify the constraints, let’s get the units that is used by PT, type:

report_units

```
Units
-----
Capacitive_load_unit      : 1e-12 Farad
Current_unit              : 1e-06 Amp
Resistance_unit           : 1000 Ohm
Time_unit                 : 1e-09 Second
Voltage_unit              : 1 Volt
1
```

From the resulting output, we can tell that the unit for time is ns and for capacitance is pico-farad.

Specify a clock period of 6ns on the input port, clk. Type:

create_clock -period 6.0 [get_ports clk]

Since this is a post-layout netlist, the clock network has already been built, thus we can instruct PT to calculate the clock latency, skews and transition. This is done by executing the following command:

set_propagated_clock [all_clocks]

Specify the following Interface (input and output port) timing constraints on all the ports except for clock port:

```
set_input_delay 1.0 -clock clk -network_latency_included \  
[remove_from_collection [all_inputs] [get_ports clk]]
```

```
set_output_delay 2.5 -clock clk -network_latency_included [all_outputs]
```

Specify a load of 0.2pf on all output ports:

```
set_load 0.2 [all_outputs]
```

All the inputs except for clock is being driven by the library cell, bufbd4:

```
set_driving_cell -lib_cell bufbd4 \  
[remove_from_collection [all_inputs] [get_ports clk]]
```

The final constrain would be the operating condition to model all PVT effects on the design:

```
set_operating_conditions -analysis_type on_chip_variation cb13fs120_tsmc_max
```

4. Constraints Validation

Verify the clock period is set to 6ns on the port clk, type:

report_clock

The output should look like this:

Attributes:
p - Propagated clock
G - Generated clock
I - Inactive clock

Clock	Period	Waveform	Attrs	Sources	Voltage Config
clk	6.00	{0 3}	p	{clk}	
1	—				

clock period 6
unit is based on library

Attribute P indicates a propagated clock

Clock port

To verify input delay constraints, type:

report_port -input_delay

Browse through the report, and verify it matches your constraints above.

Input Port	Input Delay				Related Clock	Related Pin
	Min Rise	Min Fall	Max Rise	Max Fall		
clk	--	--	--	--	--	--
clk_enable	1.00	1.00	1.00	1.00	clk	--
op1[0]	1.00	1.00	1.00	1.00	clk	--
op1[1]	1.00	1.00	1.00	1.00	clk	--
op1[2]	1.00	1.00	1.00	1.00	clk	--
op1[3]	1.00	1.00	1.00	1.00	clk	--

Min and max delay as expected

No constraints on the clock port

To verify output delay constraints, type:

report_port -output_delay

Browse through the report, and verify it matches your constraints above.

Output Port	Output Delay				Related Clock	Related Pin
	Min		Max			
	Rise	Fall	Rise	Fall		
result[0]	2.50	2.50	2.50	2.50	clk	--
result[1]	2.50	2.50	2.50	2.50	clk	--
result[2]	2.50	2.50	2.50	2.50	clk	--
result[3]	2.50	2.50	2.50	2.50	clk	--
result[4]	2.50	2.50	2.50	2.50	clk	--
result[5]	2.50	2.50	2.50	2.50	clk	--

Min and max delay as expected

To verify input drive, type:

report_port -drive

Browse through the report, and verify it matches your constraints above.

Input Port	Driving Cell		Mult(min/max)	Clock	Attrs(min/max)
	Rise(min/max)	Fall(min/max)			
clk_enable	bufbd4/bufbd4	bufbd4/bufbd4	-- / --	--	--
op1[0]	bufbd4/bufbd4	bufbd4/bufbd4	-- / --	--	--
op1[1]	bufbd4/bufbd4	bufbd4/bufbd4	-- / --	--	--
op1[2]	bufbd4/bufbd4	bufbd4/bufbd4	-- / --	--	--
op1[3]	bufbd4/bufbd4	bufbd4/bufbd4	-- / --	--	--
op1[4]	bufbd4/bufbd4	bufbd4/bufbd4	-- / --	--	--
op1[5]	bufbd4/bufbd4	bufbd4/bufbd4	-- / --	--	--
op1[6]	bufbd4/bufbd4	bufbd4/bufbd4	-- / --	--	--

driving cell is bufbd4

To verify the output port is set to 0.2pf, type:

report_port [all_outputs]

which will give you the following details:

Attributes:				
I - ideal network				
H - HyperScale context override				
Port	Dir	Pin Cap min/max	Wire Cap min/max	Attributes
result[9]	out	0.2000/0.2000	0.0000/0.0000	
result[28]	out	0.2000/0.2000	0.0000/0.0000	
result[4]	out	0.2000/0.2000	0.0000/0.0000	
result[30]	out	0.2000/0.2000	0.0000/0.0000	
result[23]	out	0.2000/0.2000	0.0000/0.0000	
result[16]	out	0.2000/0.2000	0.0000/0.0000	

output port load set to 0.2pf

Finally, to verify the operating condition and analysis type, enter:

report_design

You should get the following report which shows the OC is set to cb13fs120_tsmc_max and the analysis type is on_chip_variation.

Design Attribute	Value

Operating Conditions:	
analysis_type	on_chip_variation
operating_condition_min_name	cb13fs120_tsmc_max
process_min	1.2
temperature_min	125
voltage_min	1.08
tree_type_min	worst_case
operating_condition_max_name	cb13fs120_tsmc_max
process_max	1.2
temperature_max	125
voltage_max	1.08
tree_type_max	worst_case

5. Timing Analysis

Now we are ready to perform timing analysis. We will first use wire load models provided by the foundry libraries to calculate cell and net delays. This method is fast but is not accurate for deep submicron designs. However, it provides a good starting point for doing timing analysis as it provides a general state of the design.

To get the overall state of the design timing, type:

report_analysis_coverage

```
pt_shell> report_analysis_coverage
*****
Report : analysis_coverage
Design : DOLPHIN
Version: 0-2018.06-SP2
Date   : Fri Jan 25 21:12:26 2019
*****

Type of Check      Total      Met      Violated      Untested
-----
setup              1029      1004 ( 98%)      24 (  2%)      1 (  0%)
hold              1029      1028 (100%)       0 (  0%)      1 (  0%)
recovery           54         54 (100%)       0 (  0%)      0 (  0%)
removal           54         54 (100%)       0 (  0%)      0 (  0%)
min_pulse_width   739       685 ( 93%)       0 (  0%)      54 (  7%)
out_setup         32         32 (100%)       0 (  0%)      0 (  0%)
out_hold          32         32 (100%)       0 (  0%)      0 (  0%)
-----
All Checks         2969      2889 ( 97%)      24 (  1%)      56 (  2%)
1
```

As shown in the report above, there are 24 setup violations in the design. This indicates there are 24 timing paths where the capture flip-flop does not meet its setup timing.

To get the worst violator or critical path, type:

report_timing

This will generate a report like this:

```
Startpoint: s3_op1_reg_3_
(rising edge-triggered flip-flop clocked by clk)
Endpoint: s4_op2_reg_30_
(rising edge-triggered flip-flop clocked by clk)
Last common pin: bufbdfG3B1I1_1/Z
Path Group: clk
Path Type: max

Point              Incr      Path
-----
clock clk (rise edge)          0.00      0.00
clock network delay (propagated) 0.82      0.82
s3_op1_reg_3_/CP (sdnrq2)       0.00      0.82 r
s3_op1_reg_3_/Q (sdnrq2)       0.34      1.16 f
...
s4_op2_reg_30_/D (sdnrq4)       5.96      7.12 r
data arrival time              7.12

clock clk (rise edge)          6.00      6.00
clock network delay (propagated) 0.77      6.77
clock reconvergence pessimism   0.01      6.78
s4_op2_reg_30_/CP (sdnrq4)      6.78      6.78 r
library setup time             -0.19      6.59
data required time              6.59

data required time              6.59
data arrival time              -7.12
slack (VIOLATED)                -0.53
```

(Note: the report above has been truncated)

The critical path has a negative slack of 0.53. This indicates the flip-flop, s4_op2_reg_30_ failed to meet its setup time by 0.53ns.

Let's save the critical path start point and end point as variables. We can use this later on for comparison. Type:

```
set start "s3_op1_reg_3_/CP"
```

```
set end "s4_op2_reg_30_/D"
```

You can get the same report as above by typing:

```
report_timing -from $start -to $end
```

As you modify any design constraints or change the netlist, the critical path will change. If we want to compare the timing for the same path with different constraints being applied, using variable to store the start/end points is a very convenient way.

To find out all the violations in the design, type:

```
report_constraints -all
```

This will generate the following report:

```
max_delay/setup ('clk' group)
```

Endpoint	Slack
s4_op2_reg_30_/D	-0.53 (VIOLATED)
s4_op2_reg_28_/D	-0.48 (VIOLATED)
s4_op1_reg_27_/D	-0.45 (VIOLATED)
s4_op2_reg_25_/D	-0.37 (VIOLATED)
s4_op2_reg_27_/D	-0.37 (VIOLATED)
s4_op1_reg_23_/D	-0.36 (VIOLATED)
s4_op1_reg_30_/D	-0.33 (VIOLATED)
s4_op2_reg_29_/D	-0.33 (VIOLATED)
s4_op1_reg_25_/D	-0.29 (VIOLATED)
s4_op1_reg_28_/D	-0.29 (VIOLATED)
s4_op2_reg_23_/D	-0.27 (VIOLATED)
s4_op2_reg_31_/D	-0.26 (VIOLATED)
s4_op1_reg_26_/D	-0.25 (VIOLATED)
s4_op2_reg_22_/D	-0.24 (VIOLATED)
s4_op2_reg_20_/D	-0.21 (VIOLATED)
s4_op1_reg_22_/D	-0.21 (VIOLATED)
s4_op2_reg_26_/D	-0.16 (VIOLATED)
s4_op2_reg_24_/D	-0.16 (VIOLATED)
s4_op1_reg_29_/D	-0.16 (VIOLATED)
s4_op1_reg_21_/D	-0.14 (VIOLATED)
s4_op1_reg_31_/D	-0.13 (VIOLATED)
s4_op1_reg_24_/D	-0.12 (VIOLATED)
s4_op2_reg_21_/D	-0.06 (VIOLATED)
s4_op2_reg_18_/D	-0.02 (VIOLATED)

This report shows the endpoints of the path which are violating our timing constraints. This number is consistent with data given by report_analysis_coverage. If there are other categories of violation, that too will be shown by report_constraints.

To generate a report for checking hold time constraints, type:

```
report_timing -delay min
```



```

Startpoint: operation[0]
           (input port clocked by clk)
Endpoint:  s4_op2_reg_5_
           (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: min

```

Point	Incr	Path

clock clk (rise edge)	0.00	0.00
clock network delay (propagated)	0.00	0.00
input external delay	1.00	1.00 f
operation[0] (in)	0.02	1.02 f
s4_op2_reg_5_/SD (sdnrq1)	0.00	1.02 f
data arrival time		1.02
clock clk (rise edge)	0.00	0.00
clock network delay (propagated)	0.81	0.81
clock reconvergence pessimism	0.00	0.81
s4_op2_reg_5_/CP (sdnrq1)		0.81 r
library hold time	-0.06	0.75
data required time		0.75

data required time		0.75
data arrival time		-1.02

slack (MET)		0.27

As shown in the above report, the path has a positive slack of 0.27, which means there is no hold time violation in this design.

In a typical chip design process, when you encounter any timing violations, you will need to debug the cause of the violations – there are numerous reasons for this:

- Poor design partitioning
- Incorrect constraints
- Missing constraints
- Poor RTL design
- Non-optimal placement of cells, ports or macros
- Path requires more than one clock cycle for data to go through (multicycle path)

and many more. We will fix the violation in a later task. For now, let's continue with our analysis.

The design above was analyzed using wire load model, which is statistical way of calculating net and cell delays based on net fanout. This method is not suitable for DSM designs. The proper way of analyzing timing is by using extracted capacitance and resistance data from your layout design. From the PnR tool (IC Compiler), you can write out a SPEF file, which contains extracted net capacitance and resistance based on the actual routing being done on the design. This SPEF file when back-annotated into PT, provide a very accurate way to calculate cell and net delays which closely matches actual silicon data.

For this lab, the SPEF file is located in the **netlist** directory. To back annotate the parasitic file, type:

```
read_parasitic netlist/DOLPHIN.spef
```

You will get the following output after executing the above command:

```

0 error(s)
Format is SPEF
Annotated nets           :          5037
Annotated capacitances   :          78114
Annotated resistances    :          74264
Reduced coupling capacitances :          48237
Annotated PI models      :              0
Annotated Elmore delays  :              0

```

The above report shows that the SPEF file was successfully back-annotated. To check the quality of the annotation (i.e if there are missing annotations), type:

report_annotated_parasitics

which will generate the following report:

Net Type	Total	Lumped	RC pi	RC network	Not Annotated
Internal nets					
- Pin to pin nets	4932	0	0	4932	0
- Driverless nets	1	0	0	0	1
- Loadless nets	0	0	0	0	0
Boundary/port nets					
- Pin to pin nets	105	0	0	105	0
- Driverless nets	0	0	0	0	0
- Loadless nets	0	0	0	0	0
	5038	0	0	5037	1

The report shows that all nets have back-annotated data on it.

Run the command **report_analysis_coverage**. What type and number of violations are there in the design?

Type of violation: **setup**

Number of violations: **42**

How does this compare with your pre-annotation report? Are there more or less violations now? **More violations now.**

Run the **report_timing** command to answer the following questions:

What is the slack?

-0.96

How does the slack compare with the previous report?

Its worse than before (-0.96 vs -0.53)

Are the start and end-points the same as before?

No. The critical path has changed. Now its:

start point: s3_op2_reg_20_/CP

end point: s4_op2_reg_25_/D

What command will you run to generate a timing report for the same start and end point as per the pre-annotated report? How is the slack compared to pre-annotated results?

report_timing -from \$start -to \$finish

The slack is -0.88 which is worst compared to the pre-annotated value of -0.53.

Provide one reason why the pre-annotated report varies compared to post-annotated report?

Post-annotated design has more accurate R & C values which can be worse then the estimated values provided by the wire-load models used in the pre-annotated design.

You can use "report_qor" to check the summary of report timing, area, and power

6. Debug

Let's take a more detailed look at the critical path. Generate the following report:

report_timing -nets -capacitance -input_pins

This will add two extra columns in the timing report showing the fanout for the cells in the path and also the net fanout. Shown below is partial section of the timing report:

s4_add_189_plus_plus_U60/ZN (xn02d1)			0.27	&	2.69	f
n7069 (net)	1	0.02				
s4_mul_189_mult_mult_U1068/I (invbd4)			0.00	&	2.69	f
s4_mul_189_mult_mult_U1068/ZN (invbd4)			0.15	&	2.84	r
s4_mul_189_mult_mult_n1913 (net)	13	0.13				
s4_mul_189_mult_mult_U54/A1 (nr02d0)			0.00	&	2.84	r
s4_mul_189_mult_mult_U54/ZN (nr02d0)			0.18	&	3.02	f

As can be seen from the report, the largest fanout is 13, but resulting delay is only 0.15ns for the cell, which is relatively small. Trying to fix the timing by inserting buffer or upsizing cells, won't help much in this case. To fix this setup violations, you can try the following:

1. Modify the source code to reduce the path length and redo the synthesis and PnR – but this will take a long time
2. Try re-doing the PnR in IC Compiler
3. Reduce the clock frequency (increase the clock period)
4. Specify the long path as multicycle paths – giving more than once cycle between launch and capture flip-flops.

For this exercise, we use option 4 i.e. specifying multicycle path constraints. To specify multicycle path constraints, we need to know the endpoint names of the capture flip-flops. This can be obtained by running:

report_constraints -all

You will get the following report:

Endpoint	Slack
s4_op2_reg_25_/D	-0.96 (VIOLATED)
s4_op2_reg_28_/D	-0.95 (VIOLATED)
s4_op1_reg_27_/D	-0.95 (VIOLATED)
s4_op1_reg_23_/D	-0.95 (VIOLATED)
s4_op2_reg_30_/D	-0.94 (VIOLATED)
s4_op1_reg_30_/D	-0.94 (VIOLATED)
s4_op2_reg_31_/D	-0.93 (VIOLATED)
s4_op2_reg_27_/D	-0.92 (VIOLATED)
s4_op2_reg_20_/D	-0.91 (VIOLATED)
s4_op2_reg_22_/D	-0.89 (VIOLATED)
s4_op2_reg_29_/D	-0.89 (VIOLATED)
s4_op1_reg_25_/D	-0.87 (VIOLATED)
s4_op1_reg_28_/D	-0.85 (VIOLATED)
s4_op2_reg_23_/D	-0.85 (VIOLATED)
s4_op1_reg_26_/D	-0.84 (VIOLATED)
s4_op2_reg_24_/D	-0.82 (VIOLATED)
s4_op1_reg_22_/D	-0.81 (VIOLATED)
s4_op2_reg_18_/D	-0.80 (VIOLATED)
s4_op1_reg_29_/D	-0.79 (VIOLATED)
s4_op2_reg_26_/D	-0.77 (VIOLATED)
s4_op1_reg_21_/D	-0.75 (VIOLATED)
s4_op2_reg_19_/D	-0.75 (VIOLATED)
s4_op1_reg_24_/D	-0.72 (VIOLATED)
s3_op1_reg_31_/D	-0.71 (VIOLATED)
s4_op1_reg_31_/D	-0.71 (VIOLATED)
s4_op1_reg_18_/D	-0.71 (VIOLATED)
s4_op1_reg_19_/D	-0.71 (VIOLATED)
s4_op1_reg_20_/D	-0.68 (VIOLATED)
s4_op2_reg_21_/D	-0.68 (VIOLATED)
s4_op1_reg_17_/D	-0.62 (VIOLATED)
s3_op1_reg_30_/D	-0.50 (VIOLATED)
s4_op2_reg_17_/D	-0.45 (VIOLATED)
s4_op2_reg_16_/D	-0.38 (VIOLATED)
s3_op2_reg_31_/D	-0.37 (VIOLATED)
s4_op1_reg_16_/D	-0.36 (VIOLATED)
s3_op1_reg_29_/D	-0.34 (VIOLATED)
s4_op2_reg_15_/D	-0.23 (VIOLATED)
s3_op2_reg_30_/D	-0.15 (VIOLATED)
s4_op2_reg_14_/D	-0.14 (VIOLATED)

Using the information above, together with the pattern matching *(match multiple characters) and ?(match one character), we can constrain the design for multicycle path. Note that multicycle path constraints should only be applied to the specific endpoints and not just on any path. Type the following constrain to give these paths 2 cycles for data to go through from the launch point to capture point:

```
set_multicycle_path -setup 2 -to [get_pin "s?_*_reg_*/D"]
```

```
set_multicycle_path -hold 1 -to [get_pin "s?_*_reg_*/D"]
```

To verify this indeed can fix the setup violations, rerun the following reports:

```
report_analysis_coverage
```

```
report_constraints -all
```

```
pt_shell> report_analysis_coverage
*****
Report : analysis_coverage
Design : DOLPHIN
Version: 0-2018.06-SP2
Date   : Sat Jan 26 11:23:57 2019
*****
```

Type of Check	Total	Met	Violated	Untested
setup	1029	1028 (100%)	0 (0%)	1 (0%)
hold	1029	1028 (100%)	0 (0%)	1 (0%)
recovery	54	54 (100%)	0 (0%)	0 (0%)
removal	54	54 (100%)	0 (0%)	0 (0%)
min_pulse_width	739	685 (93%)	0 (0%)	54 (7%)
out_setup	32	32 (100%)	0 (0%)	0 (0%)
out_hold	32	32 (100%)	0 (0%)	0 (0%)
All Checks	2969	2913 (98%)	0 (0%)	56 (2%)

```
1
pt_shell> report_constraints -all
*****
Report : constraint
        -all_violators
        -path_slack_only
Design : DOLPHIN
Version: 0-2018.06-SP2
Date   : Sat Jan 26 11:24:03 2019
*****
```

A blank constrain reports indicated no violations

The design now has no violated timing constraints and can be taped-out.

This completes STA lab. Exit PrimeTime, at the pt_shell prompt, type: **exit**

In this lab, you have seen how to:

1. Read a design into PrimeTime
2. Constrain a design for timing
3. Run timing analysis
4. Debug a timing problem.