MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

VLSI DESIGN LAB MANUAL

Subject Code: ECE 2244

IV Semester B. Tech. in EE (VLSI Design and Technology)

Coordinator

Head of the Department
ECE Department

VLSID LAB Guidelines for Students

1. Be on time and follow the institution's discipline and dress code.
2. All the students are required to come prepared for the experiments to be done in the lab.
3. Students should try to analyze and understand the solved problems and then try to solve the unsolved problems of the experiment in the lab
4. Every student must bring the notebook (Observation copy), Journal (Hardbound record book) regularly to the lab.
5. Students are instructed to maintain a neat notebook (observation copy) in which the results of all the problems solved in the lab should be properly noted down. Also, solutions to the exercise problems of the experiment that they will be conducting should be written before coming to the lab.
6. Students have to get their results verified and observation copies checked by the instructor before leaving the lab.
7. In the Journal (Record book), exercise problems that they have verified in the previous lab, including a brief theory consisting of working of the circuit or an explanation regarding various commands, should be written.
8. Students will be evaluated in every lab based on individual performance, observation copy, write-up in the journal, and involvement in the lab.
9. If the students have not repeated the missed labs or completed all experiments, they will be detained and will not be eligible for the lab-end exam

Evaluation plan

• In-semester Assessment Marks: 60% (50 Marks+10 Marks for Mini Project)

- ✓ Continuous evaluation component (lab sessions 1 to 9): 10 marks each. This will be converted into 50 marks.
- ✓ Assessment is based on the conduction of each experiment, exercise problems, and answering the questions related to the experiment.
- ✓ The mini project of 10 Marks will be evaluated in the last lab (10^{th} lab).

• End semester assessment: 40 % (40 marks): **Write up** (12 Marks), **Conduction** (12 Marks), **Results** (8 Marks) and **Viva-voce** (8 Marks).

List of Experiments

Experiment 1

Introduction to Linux, Cadence environment for simulation of logic gates.

Objective:

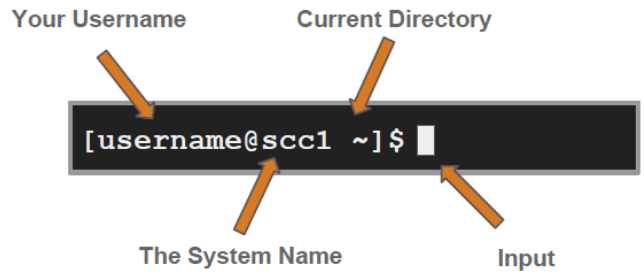
To understand and learn the basic commands required for Linux and Cadence environment for simulation of logic gates.

Theory: Linux is a family of open-source Unix-like operating systems based on the Linux kernel assembled under the model of free and open-source software development and distribution. The operating system (OS) is the software that directly manages a system's hardware and resources, like CPU, memory, and storage (RAM, RAM and other memory devices). Comes in several "distributions" to serve different purposes.

Why Linux

- Free and open-source.
- Powerful for research data centres
- Personal for desktops and phones
- Universal
- Community (and business) driven.

Linux: "prompt"



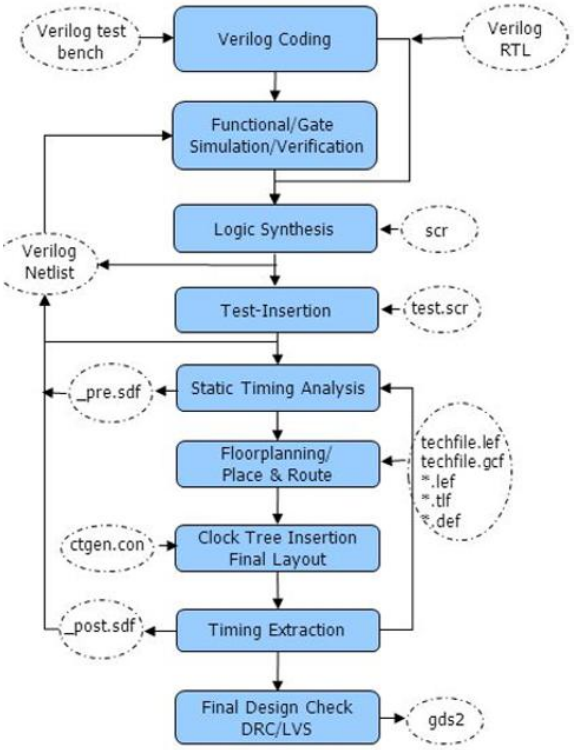
Some basic commands are:

- **ls**: List directory contents.
- **cd**: Change the current directory.
- **pwd**: Print the working directory.
- **mkdir**: Create directories.
- **rm, rmdir**: Remove files and directories.
- **cp, mv**: Copy and move files or directories.

Cadence tools:

Cadence Circuit Simulator refers to simulation tools developed by **Cadence Design Systems**, a leading provider of electronic design automation (EDA) software. These tools are widely used in the semiconductor and electronics industries for designing and verifying integrated circuits (ICs), printed circuit boards (PCBs), and other electronic systems.

The design flow is as follows,



	CADENCE
Synthesis	GENUS
Placement and Route	INNOVUS
Physical Design Verification (DRC, LVS)	CADENCE PVS
RC Extraction	QUANTUS
Static Time Analysis (STA)	TEMPUS
Power Analysis	CADENCE VOLTUS
Simulation	CADENCE XSIM

→Understanding how to start Cadence tools (e.g., virtuoso, icfb) from the command line.

Steps to start Cadence Tool:

1. Creating a Workspace:
    a. In Documents create a folder and name it.
    b. Create a sub-folder and open terminal from the sub-folder.

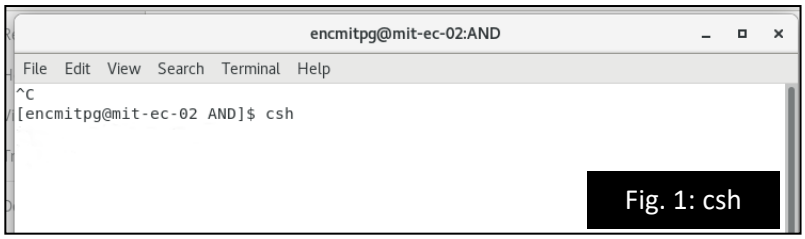


Fig. 1: csh

2. Functional Simulation:
    a. Enter the following command to open the cadence environment.
    csh (Invoke C-Shell)

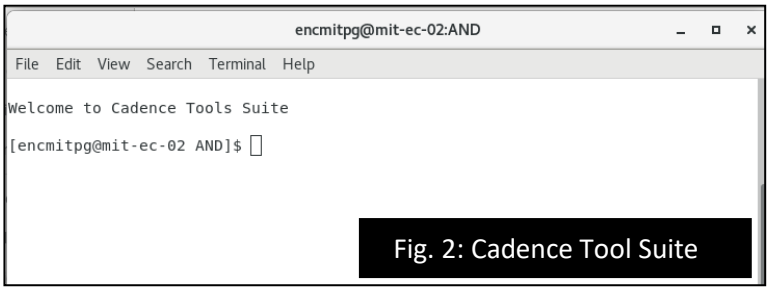


Fig. 2: Cadence Tool Suite

3. Creating Source Codes:

a. In the Terminal, enter gedit <filename>.v
b. A blank document is opened where the source code can be typed.

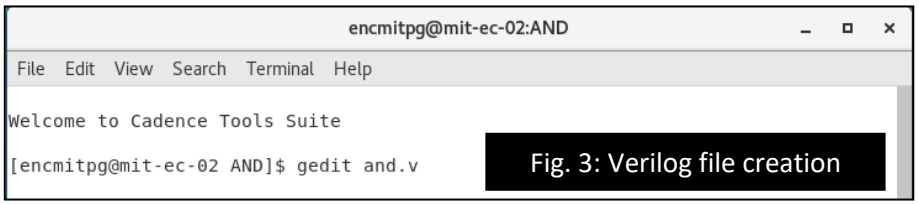


Fig. 3: Verilog file creation

4. Source Code:
   a. Save the file and close the text file.

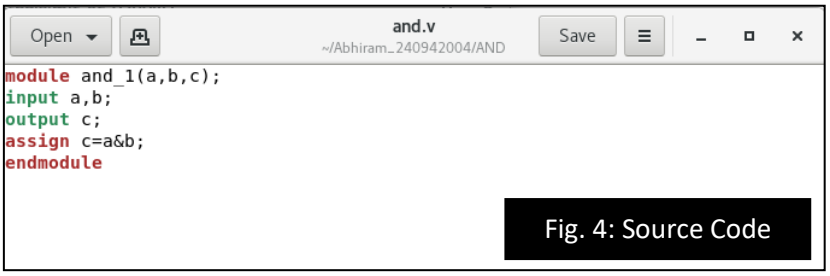


Fig. 4: Source Code

5. Creating Test bench:
   a. Similarly, create the test bench using ***gedit <filename_tb>.v*** to open a new blank document.

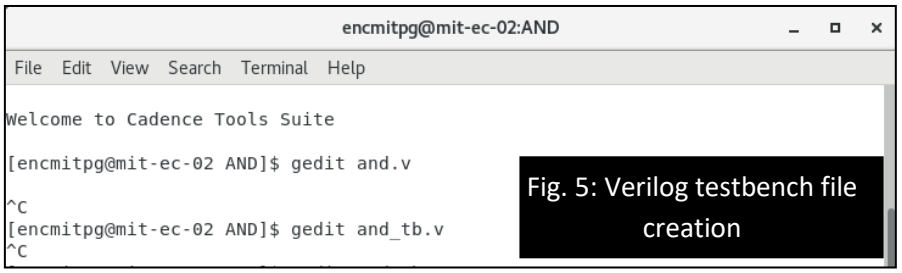


Fig. 5: Verilog testbench file creation

   b. Testbench code:

Fig. 6: Testbench code

   c. Save and close the file.

NC launch environment and incisive simulator:

NC Launch is a graphical user interface (GUI) environment in the Cadence suite that simplifies the management and execution of simulations for digital, analog, or mixed-signal designs. It is particularly

associated with Incisive Simulation tools and provides a streamlined way to configure, run, and analyze simulations.

NC Launch serves as a user-friendly interface for setting up simulation tasks, managing test benches, and visualizing results. It is designed to enhance productivity by abstracting much of the complexity associated with command-line simulation workflows.

6. To Launch Simulation tool
   Write: source /home/install/cshrc

   • *linux:/> nclaunch -new &*
   // "-new" option is used for invoking NCVERILOG for the first time for any design

   • *linux:/> nclaunch &*
   // On subsequent calls to NCVERILOG

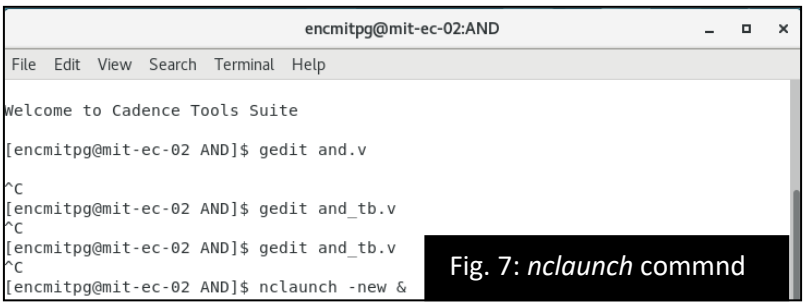


Fig. 7: *nclaunch* commnd

- It will open the *nclaunch* window for functional simulation. We can compile, elaborate and simulate it using Multiple.
- Select "Multiple Step" and then select "Create cds.lib File" as shown in below figure.

Fig. 8: *nclaunch* step

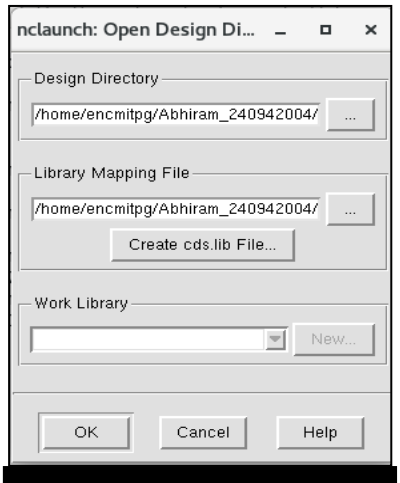


Fig. 9: *nclaunch* after selecting multiple steps

- Click the cds.lib file and save the file by clicking on Save option.
- Save cds.lib file and select the correct option for cds.lib file format based on the HDL Language and Libraries used.

- Select “Don’t include any libraries (Verilog design)” from “New cds.lib file” and click On “OK” as in below figure.
- We are simulating Verilog design without using any libraries.

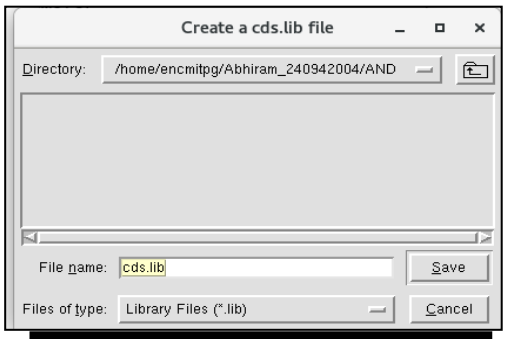


Fig. 10: Create sdc file and save

Fig. 11: Don’t include libraries

• A Click “OK” in the “*nclaunch*: Open Design Directory” window as shown in below figure.

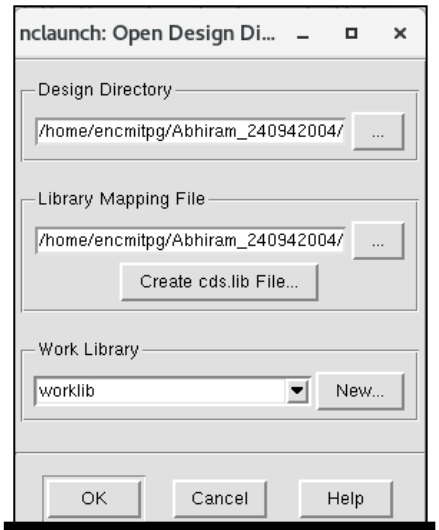


Fig. 12: After saving cds file

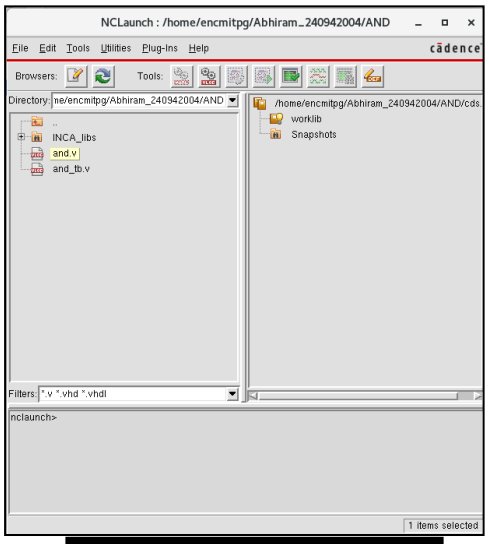


Fig. 13: nclaunch window

• A ‘NCLaunch window’ appears as shown in figure below.
• Left side you can see the HDL files. Right side of the window has worklib and snapshots directories listed.
• Worklib is the directory where all the compiled codes are stored while Snapshot will have output of elaboration which in turn goes for simulation.

✓ To perform the function simulation, the following three steps are involved,
- Compilation,
- Elaboration, and
- Simulation.

Step 1: Compilation: Process to check the correct Verilog language syntax and usage.

- Inputs: Supplied are Verilog design and test bench codes.
- Outputs: Compiled database created in mapped library if successful, generates report else error reported in log file.

Steps for compilation:

1. Create work/library directory (most of the latest simulation tools creates automatically).
2. Map the work to library created (most of the latest simulation tools creates automatically).
3. Run the compile command with compile options.

- Left side select the file and in Tools : launch Verilog compiler with current selection will get enable. Click it to compile the code.

- Worklib is the directory where all the compiled codes are stored while Snapshot will have output of elaboration which in turn goes for simulation.

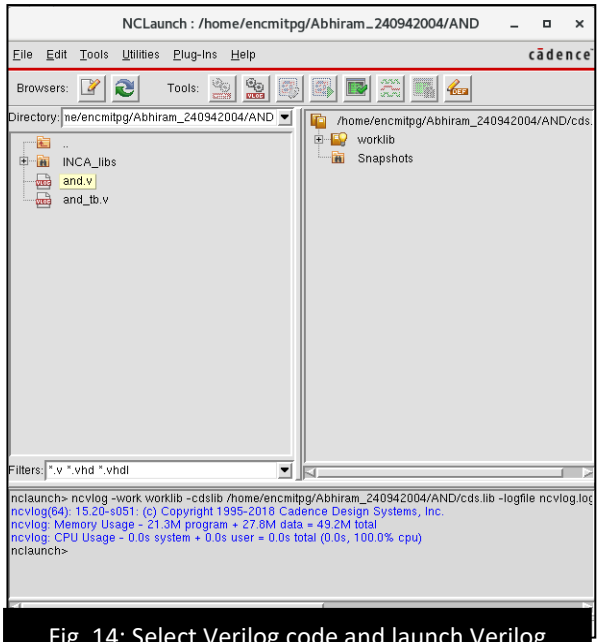


Fig. 14: Select Verilog code and launch Verilog compiler

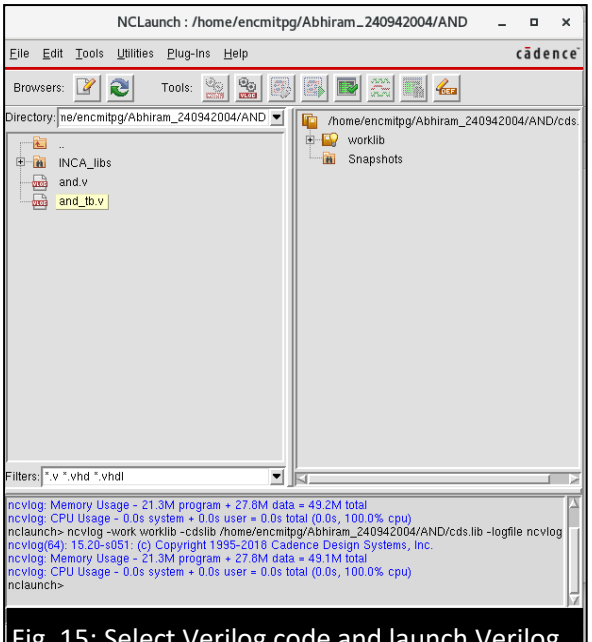
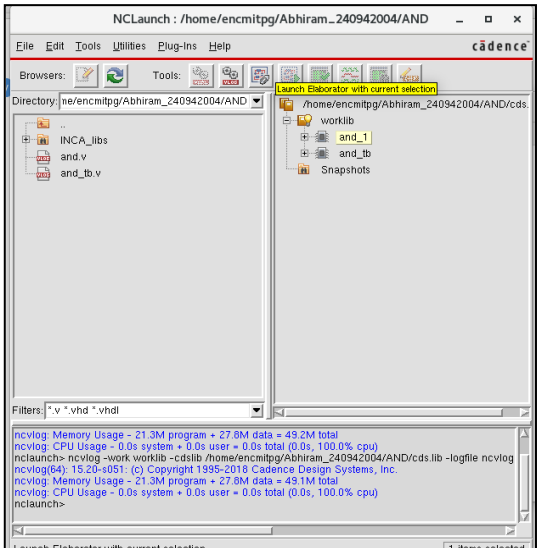


Fig. 15: Select Verilog code and launch Verilog compiler

- After compilation it will come under worklib you can see in right side window.



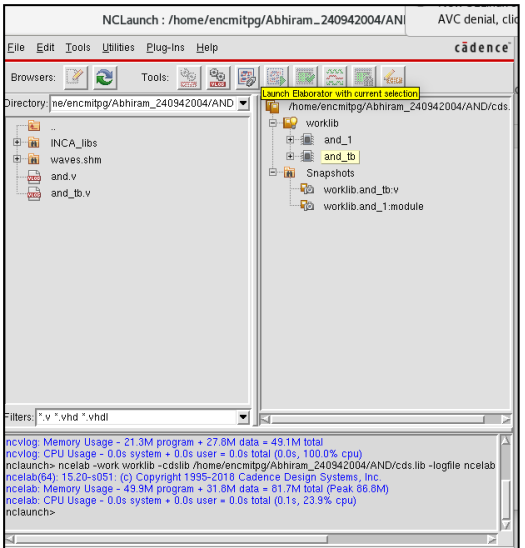


Fig. 17: Compiled Test-bench Database

- Select the test bench and compile it. It will come under worklib. Under Worklib you can see the module and test-bench.
- The cds.lib file is an ASCII text file. It defines which libraries are accessible and where they are located. It contains statements that map logical library names to their physical directory paths. For this Design, you will define a library called “worklib”.

Step 2: Elaboration: To check the port connections in hierarchical design

- Inputs: Top level design/test bench Verilog codes
- Outputs: Elaborate database updated in mapped library if successful, generates report else error reported in log file

Steps for elaboration – Run the elaboration command with elaborate options

1. It builds the module hierarchy.
2. Binds modules to module instances.
3. Computes parameter values.
4. Checks for hierarchical names conflicts.
5. It also establishes net connectivity and prepares all of this for simulation.

- After elaboration the file will come under snapshot. Select the test bench and elaborate it.

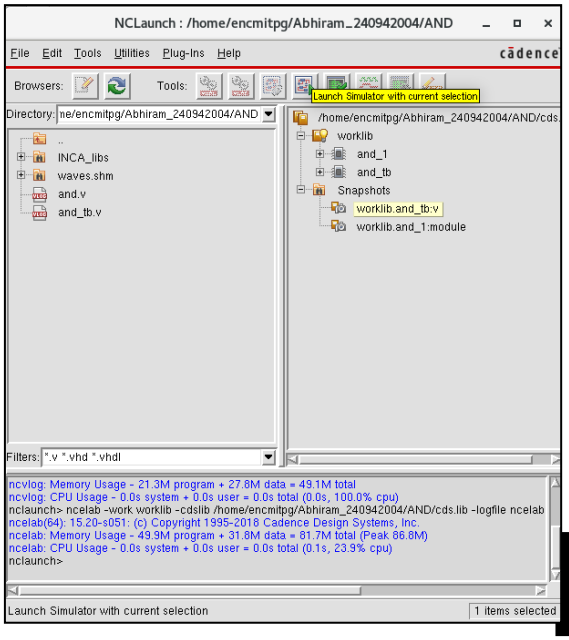


Fig. 18: Elaboration Launch Operation

Step 3: Simulation: Simulate with the given test vectors over a period of time to observe the output behaviour.

- Inputs: Compiled and Elaborated top level module name.
- Outputs: Simulation log file, waveforms for debugging.

Simulation allows to dump design and test bench signals into a waveform.

Steps for simulation – Run the simulation command with simulator options.

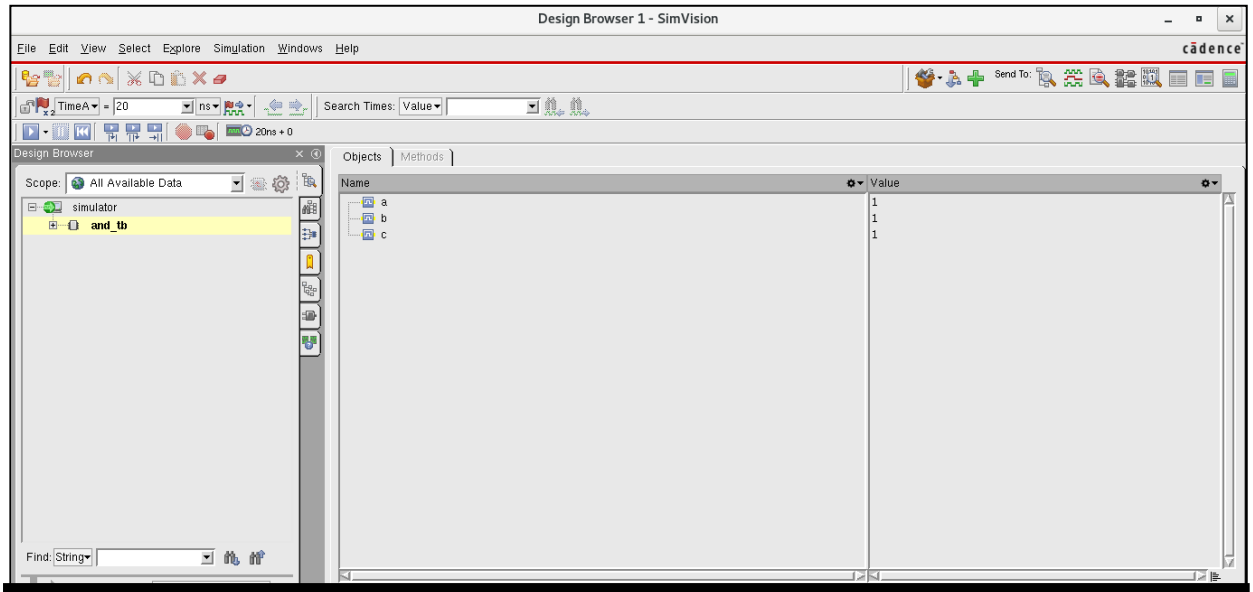


Fig. 19: After launching simulator SimVision window appears, Right click on the tb module and select 'send to simulation window'

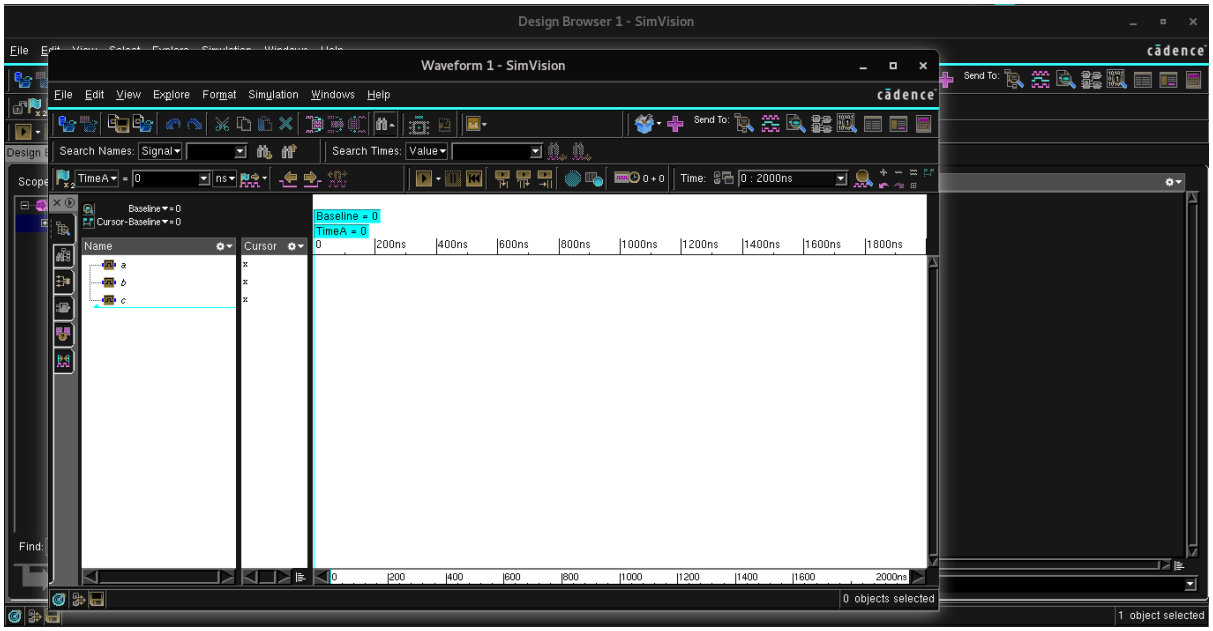


Fig. 20: Click on Run simulation to get waveform

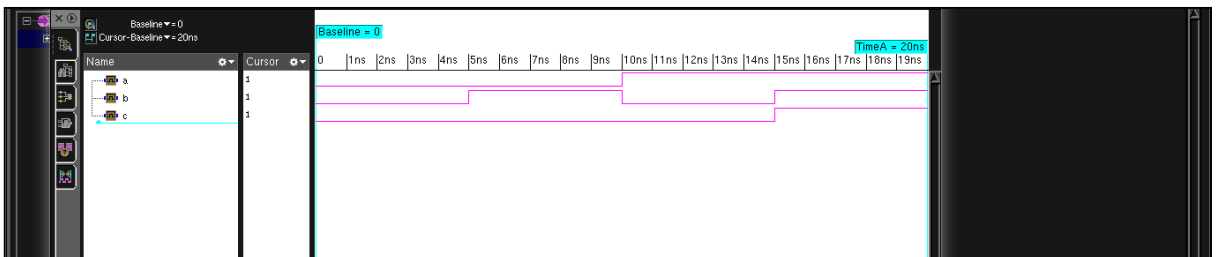


Fig. 21: Run the simulation

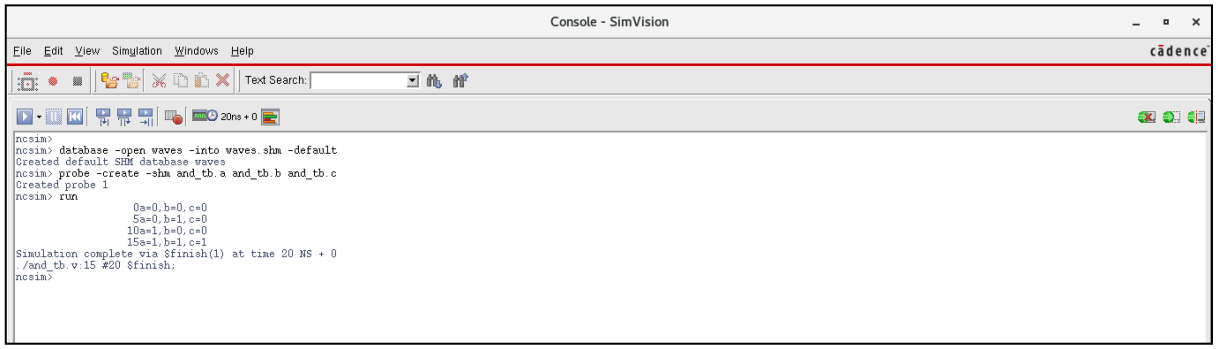


Fig. 22: Open console window to analyze the displayed values

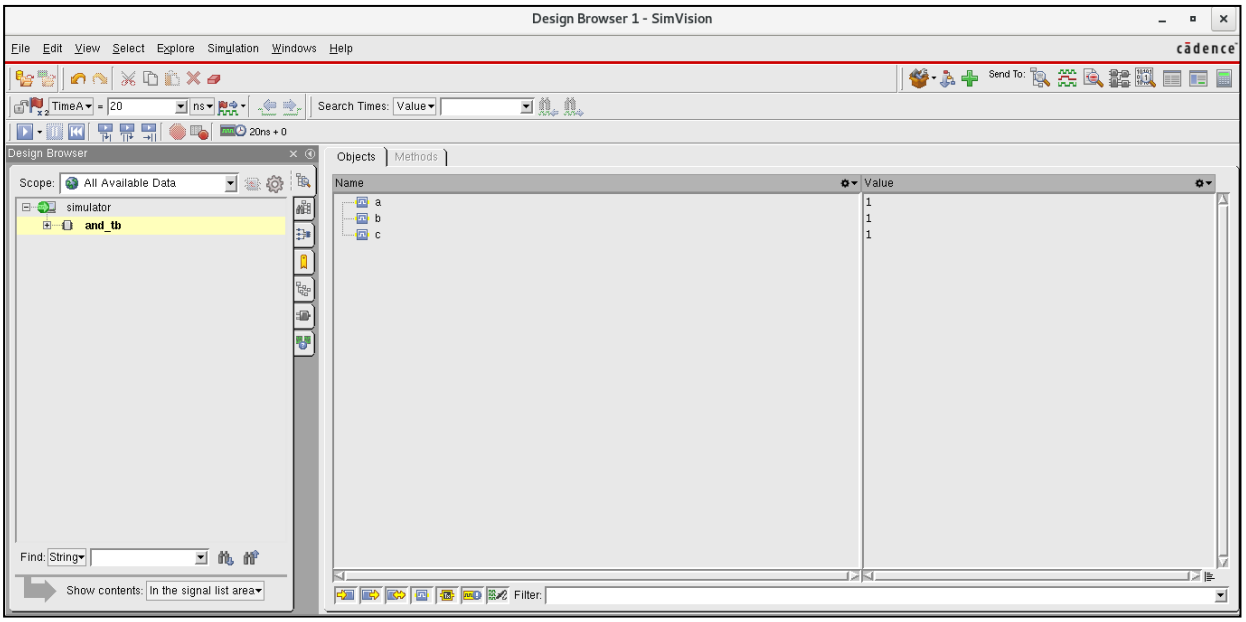


Fig. 23: go to SimVision window File then Exit from the window

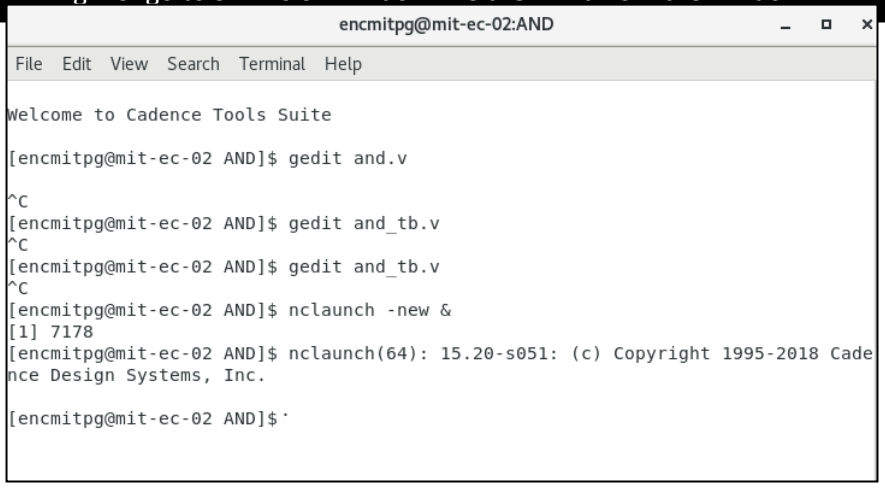


Fig. 24: Press ctrl+C and exit nclaunch

- Instead of nclaunch, design file and testbench can be run using single irun command.

```
[encmitpg@mit-ec-02 AND]$ irun and.v and_tb.v -access +rwc -gui
```

Fig. 25: irun single command to open SimVision

```
encmitpg@mit-ec-02:AND
File  Edit  View  Search  Terminal  Help
[encmitpg@mit-ec-02 AND]$ irun and.v and_tb.v -access +rwc -gui
irun(64): 15.20-s051: (c) Copyright 1995-2018 Cadence Design Systems, Inc.
file: and.v
        module worklib.and_1:v
                errors: 0, warnings: 0
file: and_tb.v
        module worklib.and_tb:v
                errors: 0, warnings: 0
                Caching library 'worklib' ....... Done
        Elaborating the design hierarchy:
        Top level design units:
                and_tb
        Building instance overlay tables: .................... Done
        Generating native compiled code:
                worklib.and_1:v <0x23a0ae19>
                        streams:   0, words:     0
                worklib.and_tb:v <0x53ca028c>
                        streams:   5, words:  5380
        Building instance specific data structures.
        Loading native compiled code:     ..................... Done
        Design hierarchy summary:
                                Instances  Unique
                Modules:             2       2
                Registers:           2       2
                Scalar wires:        3       -
                Initial blocks:      2       2
                Cont. assignments:   0       1
                Pseudo assignments:  2       2
        Writing initial simulation snapshot: worklib.and_tb:v

-----------------------------------
Relinquished control to SimVision...
ncsim>
ncsim> source /home/install/INCISIVE152/tools/inca/files/ncsimrc
ncsim>
```

Fig. 26: Terminal window after successful compilation and elaboration

Exercise Problems

1. Test all the logic gates and analyze the results.

Experiment 2

Implementation of various logic circuits and waveform verifications using NCLAUNCH.

Objective:

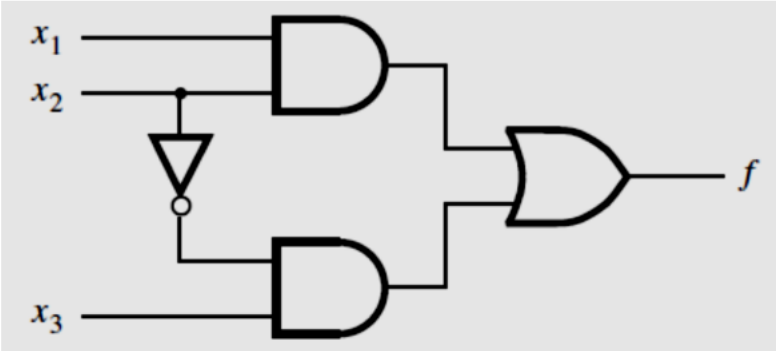
To study and implement various logic circuits and verify their waveforms using cadence nclaunch.

Theory: NC Launch is a graphical user interface (GUI) environment in the Cadence suite that simplifies the management and execution of simulations for digital, analog, or mixed-signal designs. It is particularly associated with Incisive Simulation tools and provides a streamlined way to configure, run, and analyze simulations.

NC Launch serves as a user-friendly interface for setting up simulation tasks, managing test benches, and visualizing results. It is designed to enhance productivity by abstracting much of the complexity

associated with command-line simulation workflows. The details on how to launch NCLAUNCH are described in experiment 1 and are given on page 7.

Example 1: Write Verilog code to implement the following circuit using the continuous assignment.



Truth table:

x1	x2	x3	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

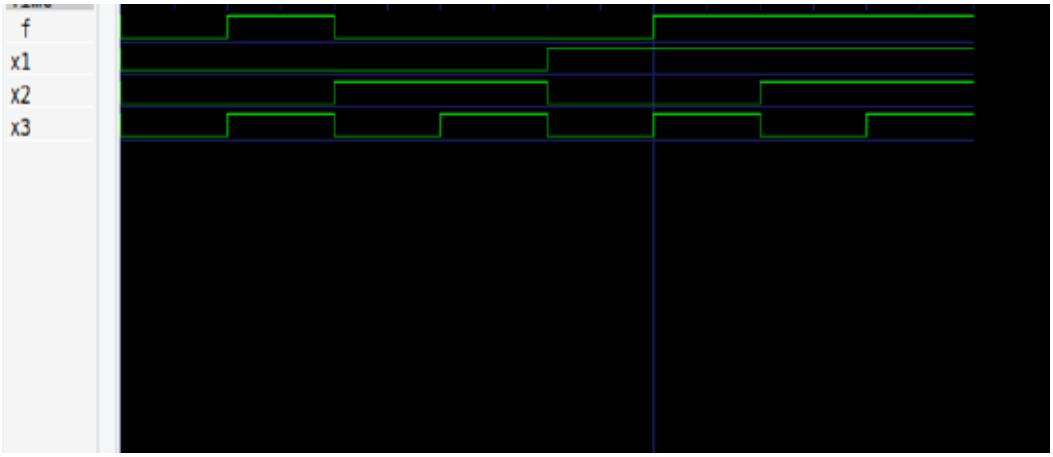
Verilog code:

```
module example1(x1, x2, x3, f);
    input x1, x2, x3;
    output f;
    assign f = (x1 & x2) | (~x2 & x3);
endmodule
```

TestBench Code:

```
module example1_tb();
reg x1, x2, x3;                 //Input
wire f;                         //Output
example1 ex1(x1, x2, x3, f);    //Instantiation of the module
initial
begin
x1=1'b0; x2=1'b0; x3=1'b0;
#20; x1=1'b0; x2=1'b0; x3=1'b1;
#20; x1=1'b0; x2=1'b1; x3=1'b0;

#20; x1=1'b0; x2=1'b1; x3=1'b1;
#20; x1=1'b1; x2=1'b0; x3=1'b0;
#20; x1=1'b1; x2=1'b0; x3=1'b1;
#20; x1=1'b1; x2=1'b1; x3=1'b0;
#20; x1=1'b1; x2=1'b1; x3=1'b1;
#20;
```

Waveform:



Example 2: Write a Verilog code for the full adder and verify the design by simulation.

A full adder is a digital circuit that computes the sum of three binary bits, typically referred to as the input bits A, B, and the carry-in bit Cin. The full adder outputs two binary bits: the sum bit S and the carry-out bit Cout

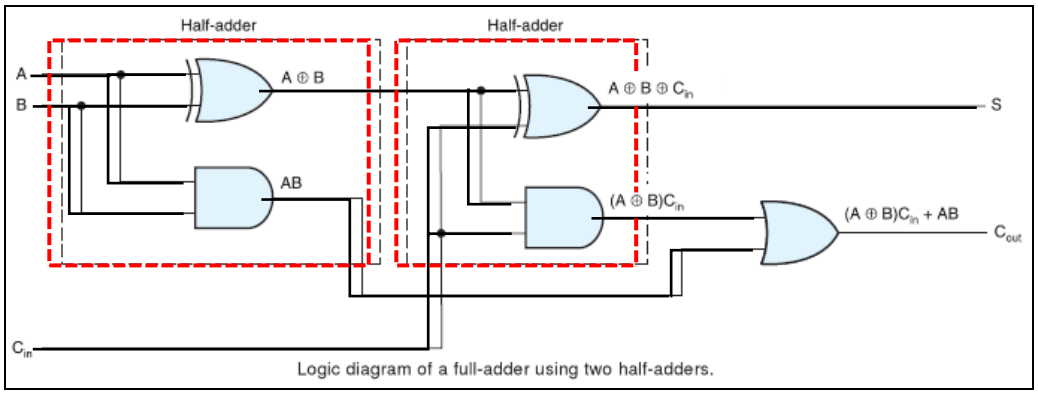
Sum (S): This is the result of adding the three input bits. The sum is given by the equation:

$$S=A \oplus B \oplus Cin$$

where $\oplus$ represents the XOR operation.

Carry-Out (Cout): This is the carry generated from the addition, which is carried over to the next higher bit position in multi-bit binary addition. The carry-out is calculated using:

$$Cout=(A \cdot B)+(B \cdot Cin)+(Cin \cdot A)$$



Logic diagram of a full-adder using two half-adders.

Source Code:

fulladder.v
~/Abhiram_240942004/FULL ADDER

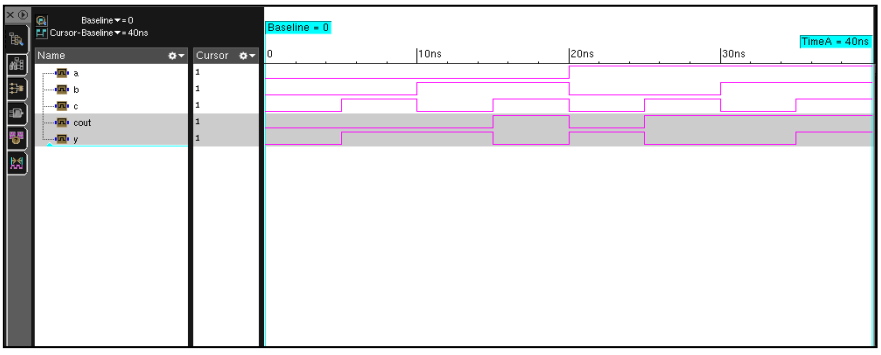
```
module fulladder(a,b,c,y,cout);
input a,b,c;
output y,cout;
assign y = a^b^c;
assign cout = a&b|b&c|a&c;
endmodule
```

Test Bench

fulladder_tb.v
~/Abhiram_240942004/FULL ADDER

```
module fulladder_tb();
reg a,b,c;
wire y,cout;
fulladder uut(a,b,c,y,cout);

initial begin
a=0;b=0;c=0;
#5 a=0;b=0;c=1;
#5 a=0;b=1;c=0;
#5 a=0;b=1;c=1;
#5 a=1;b=0;c=0;
#5 a=1;b=0;c=1;
#5 a=1;b=1;c=0;
#5 a=1;b=1;c=1;
end

initial begin
$monitor($time,"a=%b,b=%b,c=%b",a,b,c);
#40 $finish;
end
endmodule
```

Waveform:



Exercise Problems

1. Write a Verilog code for full subtractor and verify the design by simulation.
2. Implement a 2:4 Decoder using Verilog and verify its output.
3. Write the source and testbench code 4:1 MUX and 1:4 DEMUX.
4. Write the source and testbench code 4-bit priority encoder.
5. Write the source and testbench code to simulate gray to binary code converters

Experiment 3

Introduction of various abstraction levels and simulation of logic circuits.

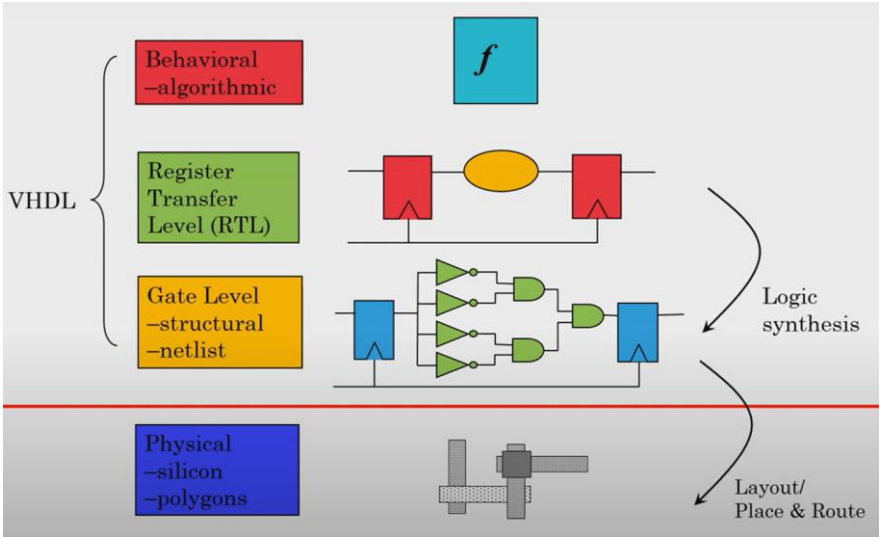
Objective:

To understand the concepts related to different abstract levels and modeling styles for logic circuits and write Verilog Programs using the same.

Theory:

Levels of Abstraction in electronic design and system modeling represent different ways of describing a system based on the amount of detail included. The abstraction level is shown below,

Level Name	Behavioral Representation	Structural Representation
System	Algorithms Truth-Tables	Processors Memories
R.T.L	Register transfers	Registers ALUs
Logic	Boolean Equations	Gates
Transistor	Transfer Function Timing diagrams	Transistors (Analog domain)



Modules are the basic building blocks for digital logic circuit modeling. The module is the principal design entity in Verilog.

Module Declaration: The first line of a module declaration specifies the *module name* and *port list* (arguments). The next few lines specify the *i/o type* (input, output or inout) and *width* of each port.

Syntax:

```
module module_name (port_list);
input [msb:lsb] input_port_list;
output [msb:lsb] output_port_list;
inout [msb:lsb] inout_port_list;

... statements...

endmodule
```

Dataflow modeling: The data-flow model uses signal assignment statements that are **concurrent** (The order of assign statements does not matter). Dataflow modeling uses *continuous assignment statements* with keyword *assign.*

assign Y = Boolean Expression using variables and operators.

A dataflow description is based on function rather than structure and hence uses a number of bit-wise operators.

Bitwise Verilog Operator	**Symbol**
NOT	~
AND	&
OR	|
XOR	^
XNOR	^~ or ~^

A dataflow description is based on function rather than structure and hence uses a number of bit-wise operators.

Example 1: Write a dataflow Verilog code for following digital building blocks and verify the design by simulation: 32:1 mux.

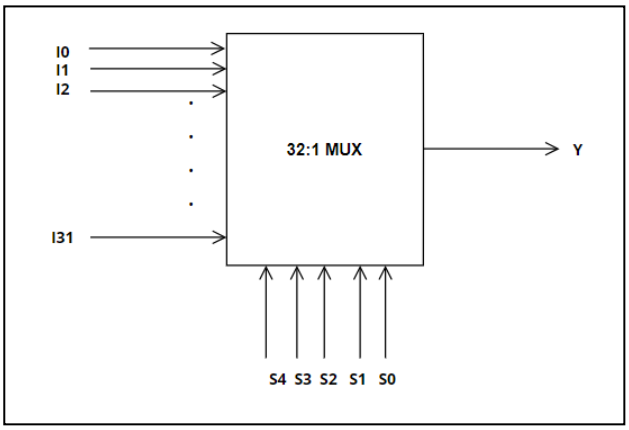
A 32:1 multiplexer (MUX) is a digital circuit that selects one of 32 input signals and forwards the selected input to a single output line based on the value of a set of control signals.

Components of a 32:1 MUX:

- Inputs (I0 to I31): These are the 32 data inputs, each of which can carry a binary signal (0 or 1). The MUX selects one of these inputs to send to the output.
- Control Signals (S0 to S4): These are 5 selection lines or control signals, which determine which of the 32 inputs is connected to the output. Since there are 32 inputs, you need 5 control signals.

- Output (Y): This is the single output line that carries the value of the selected input.

Circuit:



Source Code :

32mux.v
~/Abhiram_240942004/32_1MUX

```
module mux32(i,s,y);
input [31:0]i;
input [4:0]s;
output y;
assign y=i[s];
endmodule
```

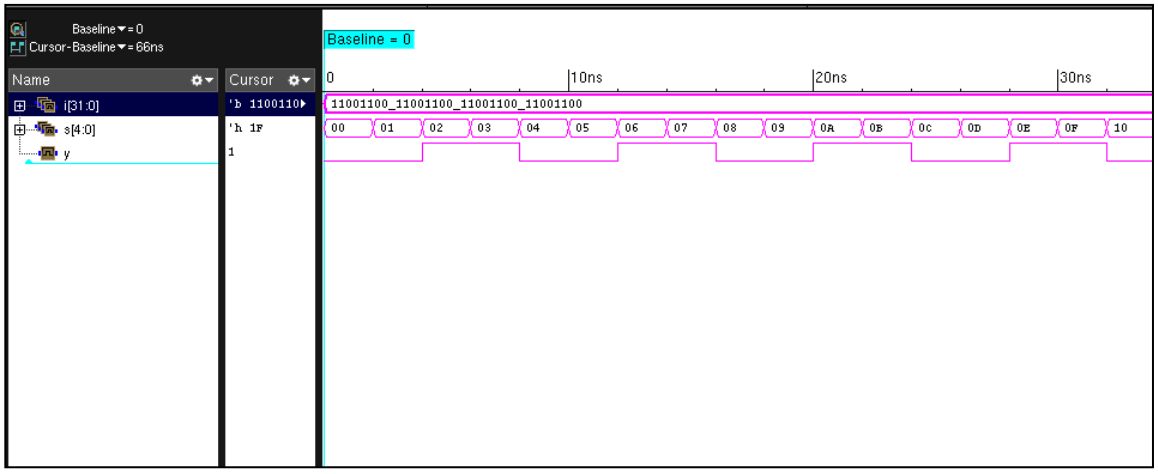
Testbench:

32mux_tb.v
~/Abhiram_240942004/32_1MUX

```
module mux32_tb();
reg [31:0]i;
reg [4:0]s;
wire y;

mux32 uut(.i(i),.s(s),.y(y));

initial begin
i = 32'h000000;
s = 5'b0;
i = 32'hCCCCCCCC; s = 5'd0;
#2 i = 32'hCCCCCCCC; s = 5'd1;
#2 i = 32'hCCCCCCCC; s = 5'd2;
#2 i = 32'hCCCCCCCC; s = 5'd3;
#2 i = 32'hCCCCCCCC; s = 5'd4;
#2 i = 32'hCCCCCCCC; s = 5'd5;
#2 i = 32'hCCCCCCCC; s = 5'd6;
#2 i = 32'hCCCCCCCC; s = 5'd7;
#2 i = 32'hCCCCCCCC; s = 5'd8;
#2 i = 32'hCCCCCCCC; s = 5'd9;
#2 i = 32'hCCCCCCCC; s = 5'd10;
#2 i = 32'hCCCCCCCC; s = 5'd11;
#2 i = 32'hCCCCCCCC; s = 5'd12;
#2 i = 32'hCCCCCCCC; s = 5'd13;
#2 i = 32'hCCCCCCCC; s = 5'd14;
```

```
Open                    32mux_tb.v                    Save
                        ~/Abhiram_240942004/32_1MUX
#2 i = 32'hCCCCCCCC; s = 5'd11;
#2 i = 32'hCCCCCCCC; s = 5'd12;
#2 i = 32'hCCCCCCCC; s = 5'd13;
#2 i = 32'hCCCCCCCC; s = 5'd14;
#2 i = 32'hCCCCCCCC; s = 5'd15;
#2 i = 32'hCCCCCCCC; s = 5'd16;
#2 i = 32'hCCCCCCCC; s = 5'd17;
#2 i = 32'hCCCCCCCC; s = 5'd18;
#2 i = 32'hCCCCCCCC; s = 5'd19;
#2 i = 32'hCCCCCCCC; s = 5'd20;
#2 i = 32'hCCCCCCCC; s = 5'd21;
#2 i = 32'hCCCCCCCC; s = 5'd22;
#2 i = 32'hCCCCCCCC; s = 5'd23;
#2 i = 32'hCCCCCCCC; s = 5'd24;
#2 i = 32'hCCCCCCCC; s = 5'd25;
#2 i = 32'hCCCCCCCC; s = 5'd26;
#2 i = 32'hCCCCCCCC; s = 5'd27;
#2 i = 32'hCCCCCCCC; s = 5'd28;
#2 i = 32'hCCCCCCCC; s = 5'd29;
#2 i = 32'hCCCCCCCC; s = 5'd30;
#2 i = 32'hCCCCCCCC; s = 5'd31;

end

initial begin
$monitor($time,"i=%h,s=%d,y=%b",i,s,y);
#66 $finish;
end
endmodule
```

Waveform:



Example 2: Write a dataflow Verilog code for 4-bit binary to gray code converter and verify the design by simulation.

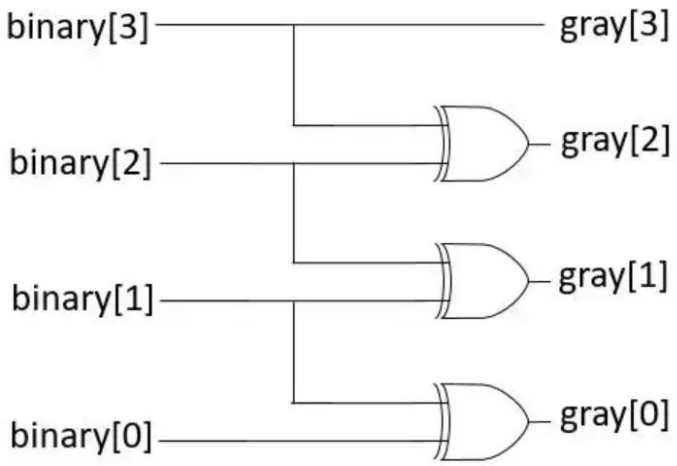
1. A 4-bit Binary to Gray code converter is a digital circuit that converts a 4-bit Binary code input into its equivalent 4-bit gray code output.
2. Components of a 4-bit Binary to Gray Converter:

3. The most significant bit (MSB) of the Gray code is the same as the MSB of the binary number.
4. Each subsequent bit of the Gray code is obtained by XORing the current binary bit with the previous binary bit.

Conversion Formula:

- gray[3]=binary[3]
- gray[2]= binary [3]⊕ binary [2]
- gray[1]= binary [2]⊕ binary [1]
- gray[0]= binary [1]⊕ binary [0]

Circuit:



Source Code:

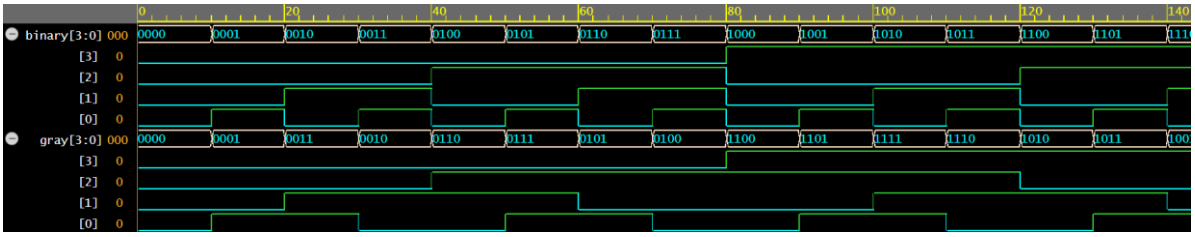
```
module binary_to_gray ( input  [3:0] binary,  output [3:0] gray );
   assign gray[3] = binary[3];                 // MSB remains the same
   assign gray[2] = binary[3] ^ binary[2];   // XOR of bit 3 and bit 2
   assign gray[1] = binary[2] ^ binary[1];   // XOR of bit 2 and bit 1
   assign gray[0] = binary[1] ^ binary[0];   // XOR of bit 1 and bit 0
endmodule
```

Testbench:

```
module tb_binary_to_gray;
   reg  [3:0] binary;
   wire [3:0] gray;
```

```
    // Instantiate the binary_to_gray module
    binary_to_gray uut ( .binary(binary), .gray(gray));
    initial begin
        $monitor("Binary = %b, Gray = %b", binary, gray);
            // Test cases
        binary = 4'b0000; #10;
        binary = 4'b0001; #10;
        binary = 4'b0010; #10;
        binary = 4'b0011; #10;
        binary = 4'b0100; #10;
        binary = 4'b0101; #10;
        binary = 4'b0110; #10;
        binary = 4'b0111; #10;
        binary = 4'b1000; #10;
        binary = 4'b1001; #10;
        binary = 4'b1010; #10;
        binary = 4'b1011; #10;
        binary = 4'b1100; #10;
        binary = 4'b1101; #10;
        binary = 4'b1110; #10;
        binary = 4'b1111; #10;
        $finish;
    end
endmodule
```

Waveform:



Exercise Problems

1. Write a dataflow Verilog code for following digital building blocks and verify the design by simulation: 5:32 Decoder
2. Write a dataflow Verilog code for 4-bit binary multiplier and verify the design by simulation.
3. Write a dataflow Verilog code for a BCD-to-seven-segment decoder and verify the design by simulation.
4. Write a dataflow Verilog code for binary to Excess-3 code and verify the design by simulation.

5. Write a dataflow Verilog code for 8:1 MUX with enable input and verify the design by simulation.

Experiment 4

Simulation of logic circuits using Behavioural Modeling

Objective:

To understand the concepts related to various abstraction levels and write Verilog programs for different logic circuit simulation and validation.

Theory:

To model the behavior of a digital logic circuit using sequential modelling, the following two statements are primarily used:

i) Initial statement
ii) Always statement

Initial statement: An initial statement executes only once. It begins its execution at the start of simulation which is at time t = 0.

Syntax:

initial
[timing_control] procedural_statement

Always statement: An always statement executes repeatedly. Just like the initial statement, an always statement also begins execution at time t = 0.

Syntax:

always
[timing_control] procedural_statement

Only a register data type can be assigned a value in either of these statements. Such a data type retains its value until a new value is assigned. All initial and always statements begin execution at time t = 0 concurrently. If no delays are specified in a procedural assignment, zero delay is the default, that is, assignment occurs instantaneously.

Exercise Problems

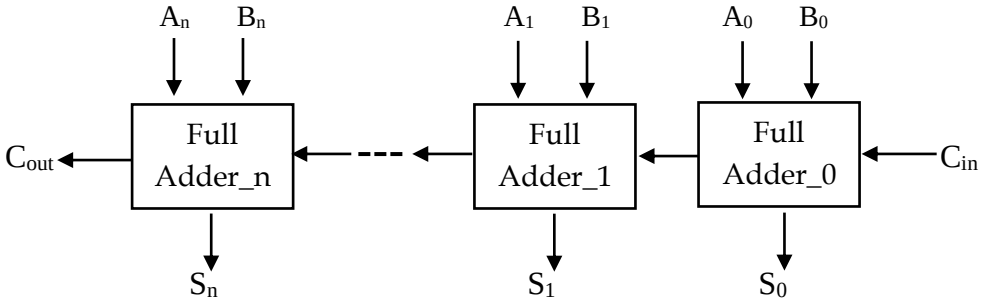
1. **Write the sequential Verilog code for N-bit full adder (assume N = 6 and use for-loop statement) and verify the design by simulation.**
    - An n-bit full adder adds two binary numbers of n bits each, along with a carry input. It performs the binary addition operation across n bits while also accounting for any carry generated from the lower bits.
    - The circuit is built using multiple 1-bit full adders. Each 1-bit full adder adds two input bits (Ai and Bi) along with a carry-in (Cin) from the previous stage. It produces a sum bit (Si) and a carry-out (Cout) that is passed to the next bit.
    - Components:

    Inputs:

        - Two n-bit binary numbers: A = A(n-1), A(n-2), ... A0; B = B(n-1), B(n-2), ..., B0
        - Carry-in (Cin): Typically, the initial carry-in for the LSB is 0.

    Outputs:

- n-bit sum: S = S(n-1), S(n-2), ..., S0
- Carry-out: The carry-out from the most significant bit (Cout) is the final carry result after adding the two numbers.



Source Code:

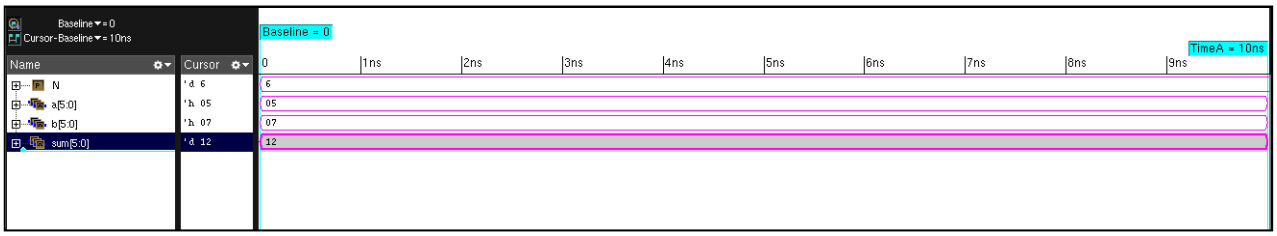
```
module nbitadder(a,b,sum);
parameter N=6;
input [N-1:0]a;
input [N-1:0]b;
output [N-1:0]sum;
wire [N-1:0]cout;
wire carry;
genvar i;
for(i=0;i<N;i=i+1)
begin
if(i==0)
fulladder fulladder_0(a[0],b[0],0,sum[0],cout[0]);
else
fulladder fulladder_0(a[i],b[i],cout[i-1],sum[i],cout[i]);
end
assign carry = cout[N-1];
endmodule

module fulladder(a,b,c,sum,cout);
input a,b,c;
output sum,cout;
assign sum = a^b^c;
assign cout = a&b|b&c|a&c;
endmodule
```

Testbench:

```
module nbitadder_tb();
parameter N = 6;
reg [N-1:0]a;
reg [N-1:0]b;
wire [N-1:0]sum;

nbitadder uut(.a(a),.b(b),.sum(sum));
```

```
initial begin
a = 6'd5;
b = 6'd7;
end

initial begin
$monitor($time,"a=%h,b=%h,sum=%b",a,b,sum);
#10 $finish;
end
endmodule
```

Waveform:



2. **Write a behavioural Verilog code for asynchronous mod-6 counter and verify the design by simulation.**

An asynchronous mod-6 counter is a counter that operates asynchronously, i.e., each flip-flop in the circuit is triggered by the output of the previous flip-flop rather than a common clock. It counts from 0 to 5 (mod-6 operation) and then resets to 0 on the next clock pulse. Below is a detailed explanation of its operation:

Input/Output:

- clk: Clock signal to drive the counter.
- reset: Asynchronous reset signal to initialize the counter.
- q: A 3-bit register to hold the counter value (mod-6 requires 3 bits to count from 0 to 5).

Behavior:

- On the positive edge of the clock or reset:
- If reset is high, the counter is reset to 000.
- If q equals 101 (5 in binary), the counter wraps back to 000 on the next clock edge.
- Otherwise, the counter increments by 1 on each clock edge.

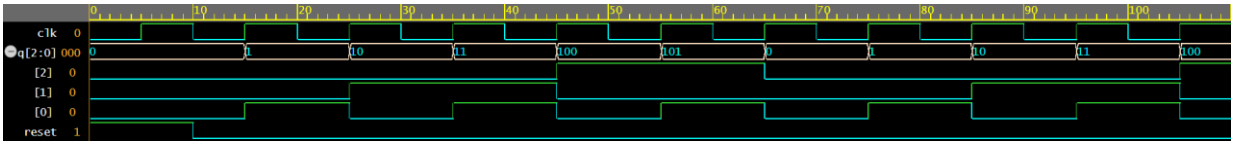
Source Code:

```
module mod6_async_counter (
    input clk,          // Clock input
    input reset,        // Asynchronous reset input
    output reg [2:0] q // 3-bit output to represent counter
value
);
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            q <= 3'b000; // Reset the counter to 0
        end else if (q == 3'b101) begin
            q <= 3'b000; // Reset to 0 when the count
reaches 6 (101 in binary)
        end else begin
            q <= q + 1;  // Increment the counter
        end
    end
endmodule
```

Testbench:

```
module tb_mod6_counter;
    reg clk;              // Clock signal
    reg reset;            // Reset signal
    wire [2:0] q;         // Counter output

    // Instantiate the mod-6 counter
    mod6_async_counter uut (
        .clk(clk),
        .reset(reset),
        .q(q)
    );

    // Generate clock signal
    initial begin
      $dumpfile("dump.vcd"); $dumpvars;
        clk = 0;
        forever #5 clk = ~clk; // Toggle clock every 5 time
units
    end

    // Apply test cases
    initial begin
        $monitor("Time = %0t | Reset = %b | Counter = %b",
$time, reset, q);

        reset = 1; #10;  // Apply reset
        reset = 0; #100; // Remove reset and observe counter
operation

        $finish;
    end
endmodule
```

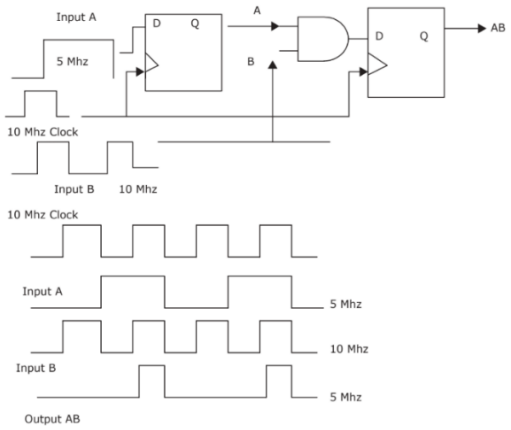
Waveform:



Exercise Problems

6. Write the behavioural Verilog code for synchronous mod 8 counter and verify the design by simulation.
7. Write a behavioural Verilog code and test bench to model and simulate 4 bit synchronous up/ down counter using Cadence tool.

8. Write sequential Verilog code for 4-bit universal shift register and verify the design by simulation.
9. Write sequential Verilog code for Master Slave JK flipflop and verify the design by simulation.
10. Write the behavioural code given circuit with input signals A and B and output AB.



Experiment 5

Simulation of logic circuits using Transistor level modeling

Objective:

To understand the concepts related to structural modeling style and write Verilog programs using the same.

Theory:

In structural modelling, the digital system is described using interconnected components or modules, essentially representing the hardware structure of the system. It closely resembles the actual circuit where logic gates and other hardware modules are interconnected to form the final system. Structural modelling is a lower level of abstraction compared to behavioural modelling, and it describes how components are connected rather than how they behave.

In Verilog, the structural model is defined using module instances and port connections. Modules can be instantiated and connected through nets to represent wires.

NMOS and PMOS Transistors:

- ***NMOS Transistor:*** Conducts when the gate input is high (1), creating a connection between the source and drain.
- ***PMOS Transistor:*** Conducts when the gate input is low (0), connecting the source and drain.

Syntax for NMOS and PMOS Transistors:

In Verilog, switch-level modelling is defined using nmos and pmos keywords. The syntax for instantiating these transistors is:

nmos (drain, source, gate); // For NMOS transistor

pmos (drain, source, gate); // For PMOS transistor

- ***Drain:*** The output of the transistor.
- ***Source:*** The source of the transistor (usually connected to ground for NMOS or vdd for PMOS).
- ***Gate:*** The control signal for the transistor (determines whether the transistor is on or off).

Exercise Problems

Example 1: Write the Verilog code and testbenches for a 3 input CMOS NAND gate using transistor-level modeling.

Circuit diagram:

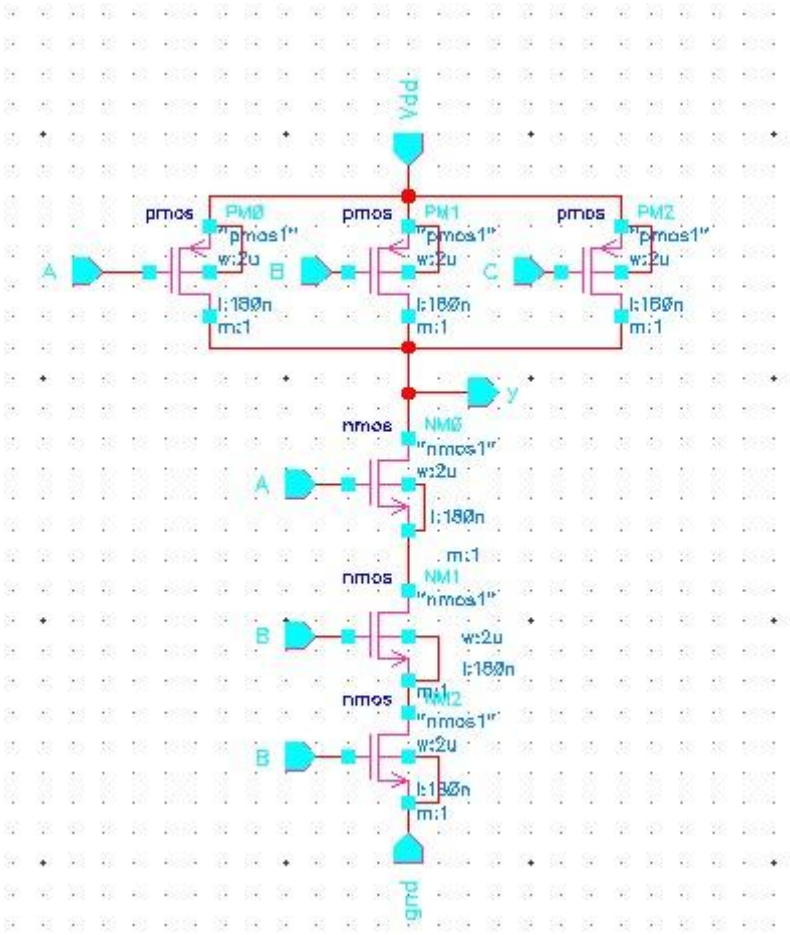


Figure 1: CMOS circuit diagram of three input NAND gate.

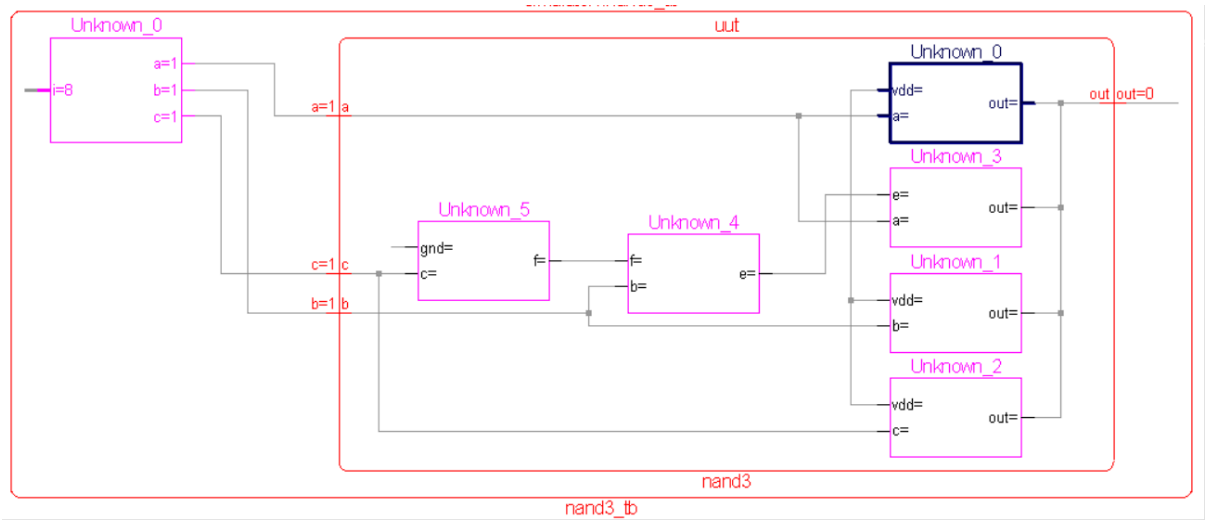


Figure 2: Circuit diagram of three input NAND gate.

Source code:

```
module nand3(out,a,b,c);

output out;

input a,b,c;

supply1 vdd;

supply0 gnd;

pmos m1(out,vdd,a);

pmos m2(out,vdd,b);

pmos m3(out,vdd,c);

nmos m4(out,e,a);

nmos m5(e,f,b);

nmos m6(f,gnd,c);

endmodule
```

Testbench code:

```
module nand3_tb();
reg a,b,c;
wire out;
integer i;
nand3 uut (out,a,b,c);
initial begin
$monitor("Time = %0t, a = %0b, b = %0b,  c = %0b, out = %0b ",$time,a,b,c,b);
a=0;b=0;c=0;
for(i=0;i<=7;i=i+1)
begin
{a,b,c}=i;
#10;
end
#300 $stop;
end
endmodule
```

Simulation and Console results:

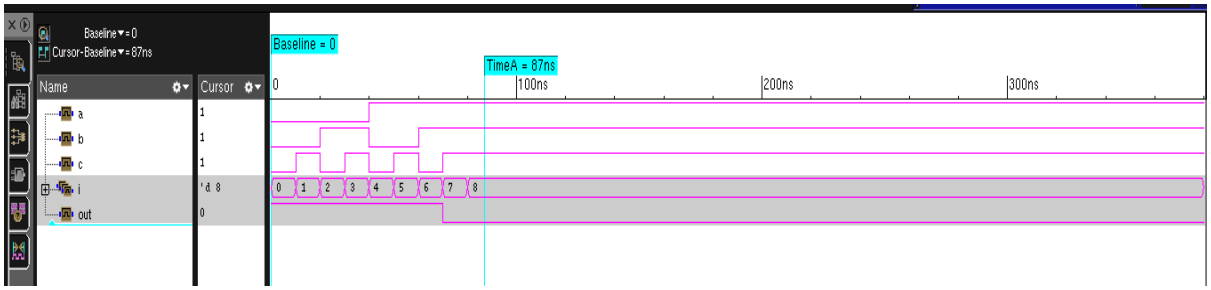


Figure 3: Simulation result of 3 input NAND gate.

```
ncsim>
ncsim> source /home/install/INCISIVE152/tools/inca/files/ncsimrc
ncsim> database -open waves -into waves.shm -default
Created default SHM database waves
ncsim> probe -create -shm nand3_tb.a nand3_tb.b nand3_tb.c nand3_tb.i nand3_tb.out
Created probe 1
ncsim> run
Time = 0, a = 0, b = 0,  c = 0, out = 0
Time = 10, a = 0, b = 0,  c = 1, out = 0
Time = 20, a = 0, b = 1,  c = 0, out = 1
Time = 30, a = 0, b = 1,  c = 1, out = 1
Time = 40, a = 1, b = 0,  c = 0, out = 0
Time = 50, a = 1, b = 0,  c = 1, out = 0
Time = 60, a = 1, b = 1,  c = 0, out = 1
Time = 70, a = 1, b = 1,  c = 1, out = 1
Simulation stopped via $stop(1) at time 380 NS + 0
ncsim>
```

Figure 4: Console result of 3 input nand gate.

Example 2: Write the Verilog code and test benches for a given function (F) using transistor-level modeling.

F = ~E(A +BCD):

Truth table:

A	B	C	D	E	F = ¬E(A + BCD)
0	0	0	0	0	0
0	1	1	1	0	1
1	0	0	0	0	1
1	1	1	1	0	1
0	0	1	1	1	0
0	1	1	0	0	0
1	1	0	1	1	0
1	0	1	0	0	1
0	1	0	0	1	0
1	1	1	1	1	0

Table 1: Truth table for the following expression.

Circuit Diagram:

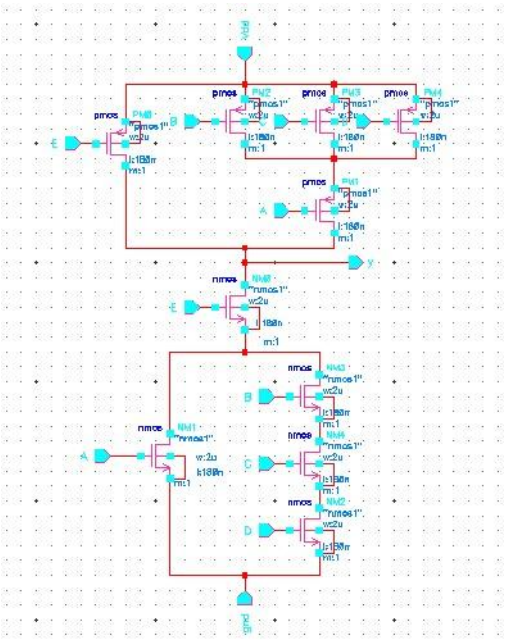


Figure 5: CMOS circuit diagram.

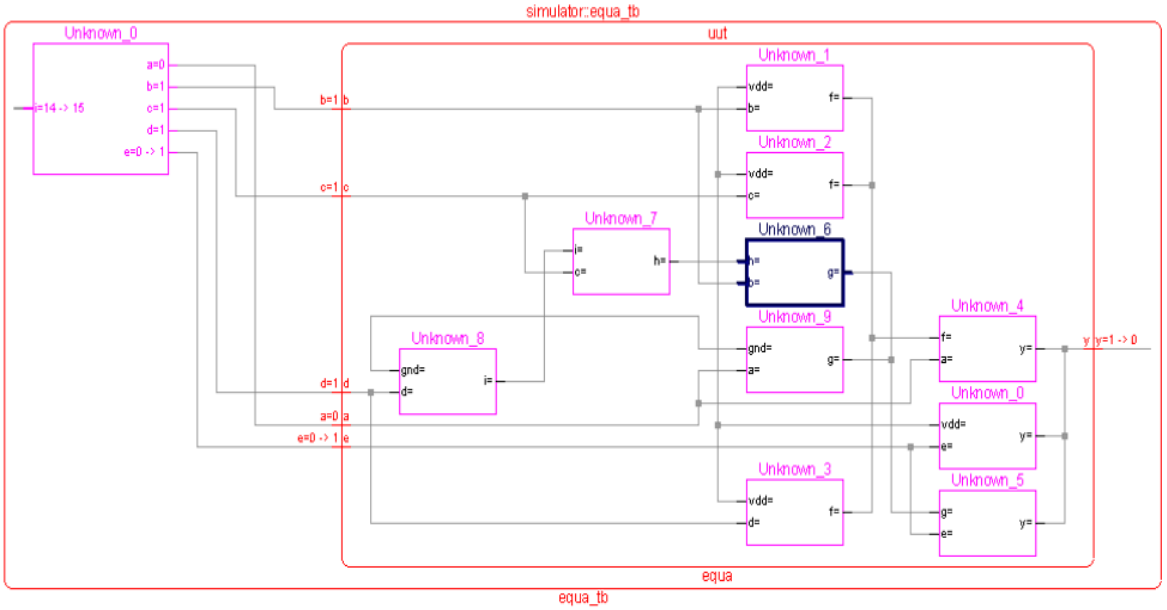


Figure 6: Circuit diagram.

Source code:

```
module equa(a,b,c,d,e,y);
input a,b,c,d,e;
output y;
supply1 vdd;
supply0 gnd;
pmos m1(y,vdd,e);
pmos m2(f,vdd,b);
pmos m3(f,vdd,c);
pmos m4(f,vdd,d);
pmos m5(y,f,a);
nmos m6(y,g,e);
nmos m7(g,h,b);
nmos m8(h,i,c);
nmos m9(i,gnd,d);
nmos m10(g,gnd,a);
endmodule
```

Testbench code:

```
module equa_tb();
reg a,b,c,d,e;
wire y;
integer i;
equa uut (a,b,c,d,e,y);
initial begin
$monitor("Time = %0t, a = %0b, b = %0b,  c = %0b, d = %0b,e = %0b y = %0b
",$time,a,b,c,d,e,y);
a=0;b=0;c=0;d=0;e=0;
for(i=0;i<=32;i=i+1)
begin
{a,b,c,d,e}=i;
#10;
end
#300 $stop;
end
endmodule
```

Simulation and console output:

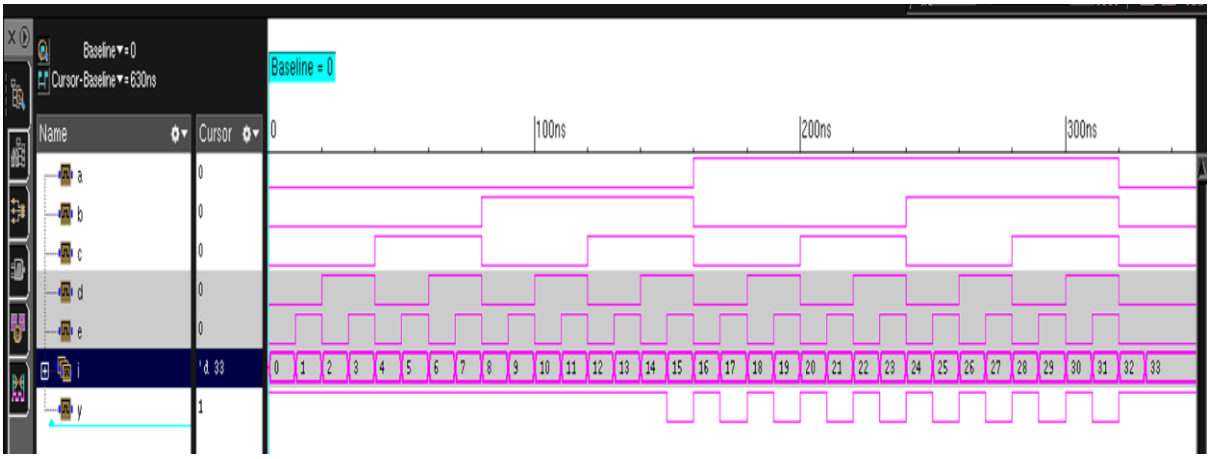


Figure 7: Simulation waveform of the above expression.

```
Created probe 1
ncsim> run
Time = 0,  a = 0,  b = 0,   c = 0,  d = 0, e = 0  y = 1
Time = 10,  a = 0,  b = 0,   c = 0,  d = 0, e = 1  y = 1
Time = 20,  a = 0,  b = 0,   c = 0,  d = 1, e = 0  y = 1
Time = 30,  a = 0,  b = 0,   c = 0,  d = 1, e = 1  y = 1
Time = 40,  a = 0,  b = 0,   c = 1,  d = 0, e = 0  y = 1
Time = 50,  a = 0,  b = 0,   c = 1,  d = 0, e = 1  y = 1
Time = 60,  a = 0,  b = 0,   c = 1,  d = 1, e = 0  y = 1
Time = 70,  a = 0,  b = 0,   c = 1,  d = 1, e = 1  y = 1
Time = 80,  a = 0,  b = 1,   c = 0,  d = 0, e = 0  y = 1
Time = 90,  a = 0,  b = 1,   c = 0,  d = 0, e = 1  y = 1
Time = 100,  a = 0,  b = 1,   c = 0,  d = 1, e = 0  y = 1
Time = 110,  a = 0,  b = 1,   c = 0,  d = 1, e = 1  y = 1
Time = 120,  a = 0,  b = 1,   c = 1,  d = 0, e = 0  y = 1
Time = 130,  a = 0,  b = 1,   c = 1,  d = 0, e = 1  y = 1
Time = 140,  a = 0,  b = 1,   c = 1,  d = 1, e = 0  y = 1
Time = 150,  a = 0,  b = 1,   c = 1,  d = 1, e = 1  y = 0
Time = 160,  a = 1,  b = 0,   c = 0,  d = 0, e = 0  y = 1
Time = 170,  a = 1,  b = 0,   c = 0,  d = 0, e = 1  y = 0
Time = 180,  a = 1,  b = 0,   c = 0,  d = 1, e = 0  y = 1
Time = 190,  a = 1,  b = 0,   c = 0,  d = 1, e = 1  y = 0
Time = 200,  a = 1,  b = 0,   c = 1,  d = 0, e = 0  y = 1
Time = 210,  a = 1,  b = 0,   c = 1,  d = 0, e = 1  y = 0
Time = 220,  a = 1,  b = 0,   c = 1,  d = 1, e = 0  y = 1
Time = 230,  a = 1,  b = 0,   c = 1,  d = 1, e = 1  y = 0
Time = 240,  a = 1,  b = 1,   c = 0,  d = 0, e = 0  y = 1
Time = 250,  a = 1,  b = 1,   c = 0,  d = 0, e = 1  y = 0
Time = 260,  a = 1,  b = 1,   c = 0,  d = 1, e = 0  y = 1
Time = 270,  a = 1,  b = 1,   c = 0,  d = 1, e = 1  y = 0
Time = 280,  a = 1,  b = 1,   c = 1,  d = 0, e = 0  y = 1
Time = 290,  a = 1,  b = 1,   c = 1,  d = 0, e = 1  y = 0
Time = 300,  a = 1,  b = 1,   c = 1,  d = 1, e = 0  y = 1
Time = 310,  a = 1,  b = 1,   c = 1,  d = 1, e = 1  y = 0
Time = 320,  a = 0,  b = 0,   c = 0,  d = 0, e = 0  y = 1
Simulation stopped via $stop(1) at time 630 NS + 0
ncsim>
```

Figure 8: Console waveform of the above expression.

Example 3: Write Verilog code for the following combinational logic function using CMOS logic F=~(ABC+DE).

Truth table:

A	B	C	D	E	F = ¬(ABC + DE)
0	0	0	0	0	1
0	1	1	1	0	1
1	0	1	1	1	0
1	1	1	0	0	0
0	1	0	1	1	0
1	1	0	0	1	1
1	1	1	1	0	0
0	0	1	1	1	0
1	0	0	0	1	1
1	1	1	1	1	0

Table 2: Truth table for the above expression.

Circuit Diagram:

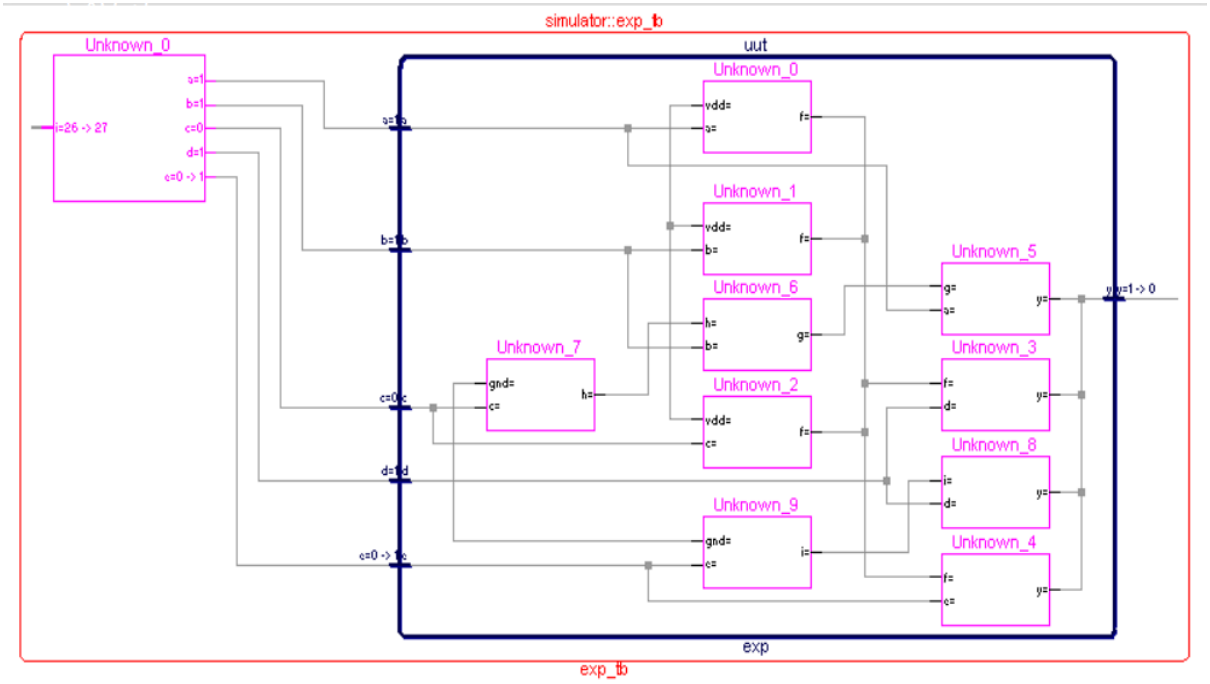


Figure 9: Circuit diagram.

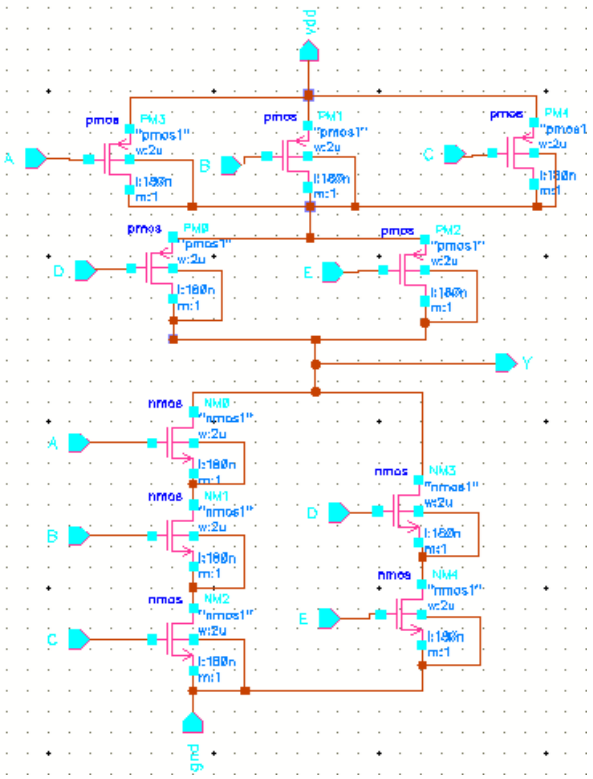


Figure 10: CMOS implementation of the above expression.

Source code:

```
module exp(a,b,c,d,e,y);
input a,b,c,d,e;
output y;
supply1 vdd;
supply0 gnd;
pmos m1(f,vdd,a);
pmos m2(f,vdd,b);
pmos m3(f,vdd,c);
pmos m4(y,f,d);
pmos m5(y,f,e);
nmos m6(y,g,a);
nmos m7(g,h,b);
nmos m8(h,gnd,c);
nmos m9(y,i,d);
nmos m10(i,gnd,e);
endmodule
```

Testbench code:

```
module exp_tb();
reg a,b,c,d,e;
wire y;
```

```
integer i;
exp uut (a,b,c,d,e,y);
initial begin
$monitor("Time = %0t, a = %0b, b = %0b,  c = %0b, d = %0b,e = %0b y = %0b
",$time,a,b,c,d,e,y);
a=0;b=0;c=0;d=0;e=0;
for(i=0;i<=32;i=i+1)
begin
{a,b,c,d,e}=i;
#10;
end
#300 $stop;
end
endmodule
```

Simulation waveform:

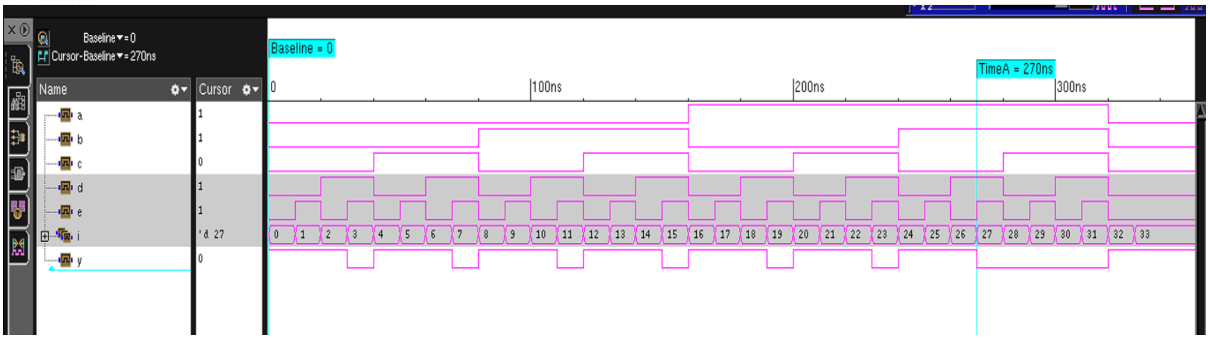


Figure 11: Simulation output.

```
Created probe 1
ncsim> run
Time = 0,  a = 0,  b = 0,   c = 0,  d = 0, e = 0 y = 1
Time = 10,  a = 0,  b = 0,   c = 0,  d = 0, e = 1 y = 1
Time = 20,  a = 0,  b = 0,   c = 0,  d = 1, e = 0 y = 1
Time = 30,  a = 0,  b = 0,   c = 0,  d = 1, e = 1 y = 0
Time = 40,  a = 0,  b = 0,   c = 1,  d = 0, e = 0 y = 1
Time = 50,  a = 0,  b = 0,   c = 1,  d = 0, e = 1 y = 1
Time = 60,  a = 0,  b = 0,   c = 1,  d = 1, e = 0 y = 1
Time = 70,  a = 0,  b = 0,   c = 1,  d = 1, e = 1 y = 0
Time = 80,  a = 0,  b = 1,   c = 0,  d = 0, e = 0 y = 1
Time = 90,  a = 0,  b = 1,   c = 0,  d = 0, e = 1 y = 1
Time = 100,  a = 0,  b = 1,   c = 0,  d = 1, e = 0 y = 1
Time = 110,  a = 0,  b = 1,   c = 0,  d = 1, e = 1 y = 0
Time = 120,  a = 0,  b = 1,   c = 1,  d = 0, e = 0 y = 1
Time = 130,  a = 0,  b = 1,   c = 1,  d = 0, e = 1 y = 1
Time = 140,  a = 0,  b = 1,   c = 1,  d = 1, e = 0 y = 1
Time = 150,  a = 0,  b = 1,   c = 1,  d = 1, e = 1 y = 0
Time = 160,  a = 1,  b = 0,   c = 0,  d = 0, e = 0 y = 1
Time = 170,  a = 1,  b = 0,   c = 0,  d = 0, e = 1 y = 1
Time = 180,  a = 1,  b = 0,   c = 0,  d = 1, e = 0 y = 1
Time = 190,  a = 1,  b = 0,   c = 0,  d = 1, e = 1 y = 0
Time = 200,  a = 1,  b = 0,   c = 1,  d = 0, e = 0 y = 1
Time = 210,  a = 1,  b = 0,   c = 1,  d = 0, e = 1 y = 1
Time = 220,  a = 1,  b = 0,   c = 1,  d = 1, e = 0 y = 1
Time = 230,  a = 1,  b = 0,   c = 1,  d = 1, e = 1 y = 0
Time = 240,  a = 1,  b = 1,   c = 0,  d = 0, e = 0 y = 1
Time = 250,  a = 1,  b = 1,   c = 0,  d = 0, e = 1 y = 1
Time = 260,  a = 1,  b = 1,   c = 0,  d = 1, e = 0 y = 1
Time = 270,  a = 1,  b = 1,   c = 0,  d = 1, e = 1 y = 0
Time = 280,  a = 1,  b = 1,   c = 1,  d = 0, e = 0 y = 0
Time = 290,  a = 1,  b = 1,   c = 1,  d = 0, e = 1 y = 0
Time = 300,  a = 1,  b = 1,   c = 1,  d = 1, e = 0 y = 0
Time = 310,  a = 1,  b = 1,   c = 1,  d = 1, e = 1 y = 0
Time = 320,  a = 0,  b = 0,   c = 0,  d = 0, e = 0 y = 1
Simulation stopped via $stop(1) at time 630 NS + 0
ncsim>
```

Figure 12: Console output.

Exercise Problems

1. Write Verilog code for the following combinational logic function using CMOS logic a) F=~(ABC+DE) , b) XOR gate, c) half adder
2. Write Verilog code for the full adder and verify the output.
3. Write Verilog code for the full substractor and verify the output.
4. Write Verilog code for the 4:1 MUX and verify the output.
5. Write Verilog code for the 2:4 decoder and verify the output.

Experiment-6

Introduction to the synthesis of Digital circuits

Objective:

The main objectives of this lab are:

- Familiarization with synthesis flow.
- Setting up synthesis constraints.
- Generating optimized gate-level netlist and Standard Design Constraints.

Introduction

Synthesis is a process of transforming RTL (a description of a circuit expressed in a language such as Verilog or VHDL written in behavioral modeling or data flow modeling) to technologydependent or independent gate-level netlist including nets, sequential and combinational cells, and their connectivity. The main goal of synthesis is obtaining a gate-level netlist, logic optimization, inserting a clock-gating cell for power reduction, inserting DFT (Design for Testability) cell, and maintaining the logical equivalence between RTL and gate-level netlist. The best output of place and route depend on the synthesis.

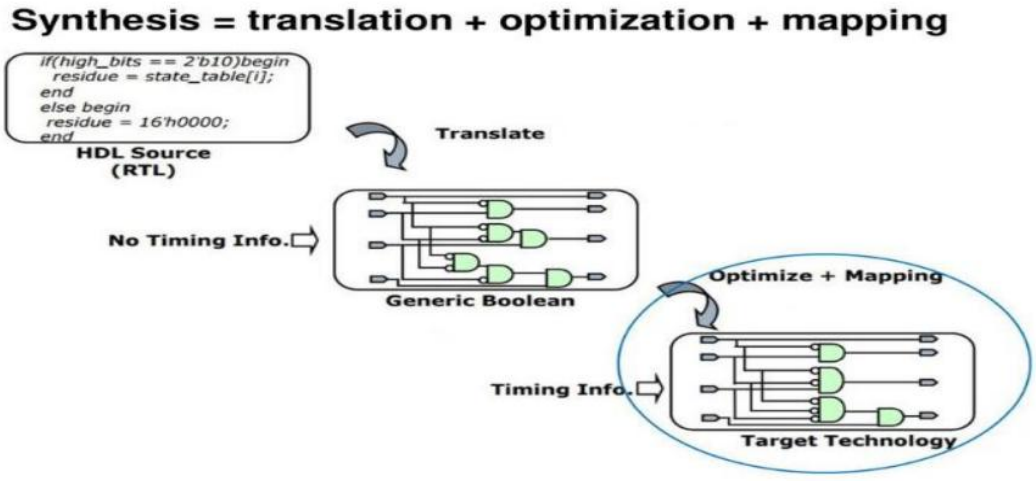


Fig: Steps of Synthesis

Purpose of Logic Circuit Synthesis

- **Automates Design:** Reduces the manual effort required to design circuits by automating the generation of gate-level implementations.
- **Optimization:** Ensures the design meets specifications for performance, power consumption, and area (PPA metrics).

- **Bridges Abstraction Levels**: Converts a functional specification into hardware-level representations.

Steps in Logic Circuit Synthesis

- **High-Level Design Description:**
  - Starts with a functional specification in a **hardware description language (HDL)** such as **Verilog** or **VHDL**.
  - Example: Writing if-else conditions, loops, or operations at a behavioral level.
- **Translation:**
  - The HDL code is converted into an **intermediate representation** (IR) or **Boolean expressions**.
- **Logic Optimization:**
  - Simplifies Boolean expressions using optimization techniques:
    - Reducing the number of gates.
    - Eliminating redundant paths.
  - Ensures the circuit adheres to constraints like timing, area, and power.
- **Technology Mapping:**
  - Maps the optimized Boolean expressions to the **technology library** of standard cells (pre-designed logic gates like AND, OR, NOT, multiplexers, etc.).
  - Each standard cell is characterized by a specific process technology node (e.g., 7nm, 28nm).
- **Generation of Gate-Level Netlist:**
  - Produces a **netlist** that describes the connectivity between gates.

Types of Synthesis

- **Behavioral Synthesis:**
  - Converts high-level algorithms into a circuit at the RTL (Register Transfer Level).
- **Logic Synthesis:**
  - Converts RTL descriptions into a **gate-level design**.
- **Physical Synthesis:**
  - Prepares the gate-level netlist for layout by considering physical constraints such as placement and routing.

Key Components of Synthesis

- **Technology Library:**
  - Contains predefined logic cells, characterized for performance, power, and area.
- **Constraints File:**
  - Specifies design constraints such as:
  - Maximum delay (timing constraints).
  - Target area or power limits.
- **Optimization Objectives:**
  - Ensure the design is:
    - **Fast** (meets timing).
    - **Small** (uses minimal chip area).
    - **Power-efficient** (reduces energy consumption)

Example:

To write a Verilog code for 4bit up/down asynchronous rest counter and its test-bench for verification and do the synthesizing.

Tool Required:

- Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim).
- Synthesis: Genus

Design Information and Bock Diagram:

- An up/down counter is a digital counter which can be set to count either from 0 to MAX_VALUE or MAX_VALUE to 0.
- The direction of the count(mode) is selected using a single bit input. The module has 3 inputs - clk, reset which is active high and an Up Or Down mode input. The output is Counter which is 4 bits in size.
- When Up mode is selected, counter counts from 0 to 15 and then again from 0 to 15.
- When Down mode is selected, the counter counts from 15 to 0 and then again from 15 to 0.
- Changing mode doesn't reset the Count value to zero.
- You must apply high value to reset, to reset the Counter output.

You must perform the function simulation using NCLaunch (Already discuss in previous experiments). This involves three stages :

- Step 1: Compilation- Process to check the correct Verilog language syntax and usage
- Step 2: Elaboration- To check the port connections in hierarchical design
- Step 3: Simulation- Simulate with the given test vectors over a period of time to observe the output behavior.

Verilog code for 4-Bit Up-Down Counter:

Verilog code for 4-Bit Up-Down Counter:

```
//Defining a Timescale for Precision
`timescale 1ns/1ps
//Defining Module
module counter(clk,rst,m,count);
//Defining Inputs and Outputs (4bit)
input clk,rst,m;
output reg [3:0]count;
```

```
//The Block is executed when EITHER of positive edge of clock
//Both are independent events or Neg Edge of Rst arrives
always@(posedge clk or negedge rst)
begin
if(!rst)
count=0;
if(m)
count=count+1;
else
count=count-1;
end
endmodule
```

Creating Test Bench:

- Similarly, create your test bench using gedit .v or .vhdl to open a new blank document (4bup_down_count_tb.v).

Test-bench code for 4-Bit Up-Down Counter:

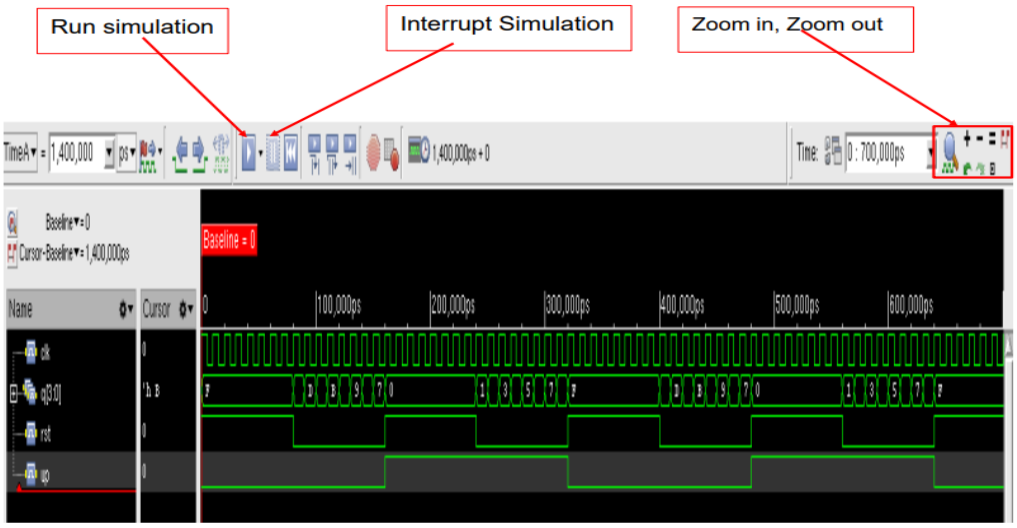
```
`timescale 1ns/1ps          // Creating Time Scale as in Source Code
module counter_test;        // Defining Module Name without Port List
reg clk, rst,m;             // Defining I/P as Registers [to Hold Values]
wire [3:0] count;           // Defining O/P as Wires [To Probe Waveforms]
Initial
begin
clk=0;                      // Initializing Clock and Reset
rst=0;#25;                  // All O/P is 4'b0000 from t=0 to t=25ns.
Rst=1;                      // Up-Down counting is allowed at posedge clk
end
initial
begin
m=1;                        // Condition for Up-Count
#600 m=0;                   // Condition for Down-Count
rst=0;#25;
rst=1;
#500 m=0;
end
counter counter1(clk,m,rst, count);   // Instantiation of Source Code
always #5 clk=~clk;         // Inverting Clk every 5ns
initial
#1400 $finish;              // Finishing Simulation at t=1400ns
endmodule
```

You must perform the function simulation using NCLaunch (Already discuss in previous experiments). This involves three stages :

- Step 1: Compilation- Process to check the correct Verilog language syntax and usage

- Step 2: Elaboration- To check the port connections in hierarchical design
- Step 3: Simulation- Simulate with the given test vectors over a period of time to observe the
  output behavior.

Final Wave form after simulation



Synthesize the design using Constraints and analyse reports, critical path and Max Operating Frequency.

Step 1 : Getting Started

- Make sure you close out all the Incisive tool windows first.
- Synthesis requires three files as follows,
  - Liberty Files (.lib)
  - Verilog/VHDL Files (.v or .vhdl or .vhd)
  - SDC (Synopsis Design Constraint) File (.sdc)

Step 2 : Creating an SDC File

- An SDC file (Synopsys Design Constraints file) is a widely used format in the IC design process to specify design constraints. These constraints guide EDA tools during synthesis, place, and route (P&R), and static timing analysis (STA). The file uses Tcl-based syntax and focuses on defining timing, clocking, and I/O constraints.

Key Features of SDC Files:

1. **Clock Definitions**:
    - Specify the clocks in the design, including their frequency, waveform, and latency.
2. **Input and Output Delays**:
    - Defines timing constraints for external inputs and outputs relative to a clock.
3. **Timing Exceptions**:
    - Includes specific paths that require exceptions, such as false paths or multi-cycle paths.
4. **Design Constraints**:
    - Specifies load, drive strength, and other characteristics to guide synthesis and STA.

Creating SDC File

- In your terminal type “gedit counter_top.sdc” to create an SDC File if you do not have one.
- The SDC File must contain the following commands;

i. create_clock -name clk -period 2 -waveform {0 1} [get_ports "clk"]
ii. set_clock_transition -rise 0.1 [get_clocks "clk"]
iii. set_clock_transition -fall 0.1 [get_clocks "clk"]
iv. set_clock_uncertainty 0.01 [get_ports "clk"]
v. set_input_delay -max 0.8 [get_ports "rst"] -clock [get_clocks "clk"]
vi. set_output_delay -max 0.8 [get_ports "count"] -clock [get_clocks "clk"]
vii. set_input_transition 0.12 [all_inputs]
viii. set_load 0.15 [all_outputs]
ix. set_max_fanout 30.00 [current_design]

i → Creates a Clock named “clk” with Time Period 2ns and On Time from t=0 to t=1.
ii, iii → Sets Clock Rise and Fall time to 100ps.
iv → Sets Clock Uncertainty to 10ps.
v, vi → Sets the maximum limit for I/O port delay to 1ps.

```
create_clock -name clk -period 2 -waveform {0 1} [get_ports "clk"]

set_input_delay -max 0.8 -clock clk [all_inputs]
set_output_delay -max 0.8 -clock clk [all_outputs]

set_input_transition 0.2 [all_inputs]
set_max_capacitance 30 [get_ports]

set_clock_transition -rise 0.1 [get_clocks "clk"]
set_clock_transition -fall 0.1 [get_clocks "clk"]

set_clock_uncertainty 0.01 [get_ports "clk"]

set_input_transition 0.12 [all_inputs]
set_load 0.15 [all_outputs]
set_max_fanout 30.00 [current_design]
```

Step 3: Performing Synthesis

- The Liberty files are present in the below path,
  /home/install/FOUNDRY/digital/nm/dig/lib/ (This can be changed based on defined location)
- The Available technology nodes are 180nm ,90nm and 45nm.
- In the terminal, initialise the tools with the following commands if a new terminal is being used.
  - csh
  - source /home/install/cshrc
  - The tool used for Synthesis is “Genus”. Hence, type “genus -gui” to open the tool.
- The Following are commands to proceed,

1. read_libs /home/install/FOUNDRY/digital/90nm/dig/lib/slow.lib //corner analysis
2. read_hdl counter.v
3. elaborate
4. read_sdc constraints_top.sdc //Reading Top Level SDC
5. set_db syn_generic_effort medium //Effort level to medium for generic, mapping and optimization
6. set_db syn_map_effort medium
7. set_db syn_opt_effort medium
8. syn_generic
9. syn_map
10. syn_opt //Performing Synthesis Mapping and

Optimisation

11. report_timing > counter_timing.rep //Generates Timing report for worst datapath and dumps into file
12. report_area > counter_area.rep //Generates Synthesis Area report and dumps into a file
13. report_power > counter_power.rep //Generates Power Report [Pre-Layout]
14. write_hdl > counter_netlist.v //Creates readable Netlist File
15. write_sdc > counter_sdc.sdc //Creates Block Level SDC

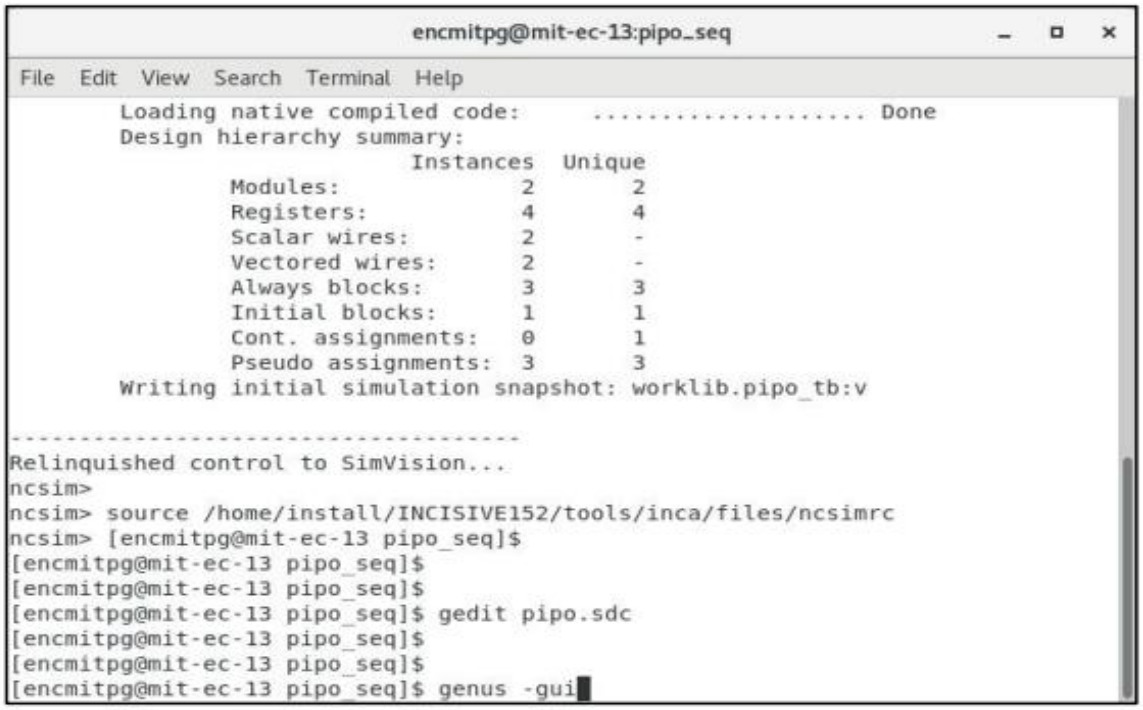


Fig. 31: Launch genus

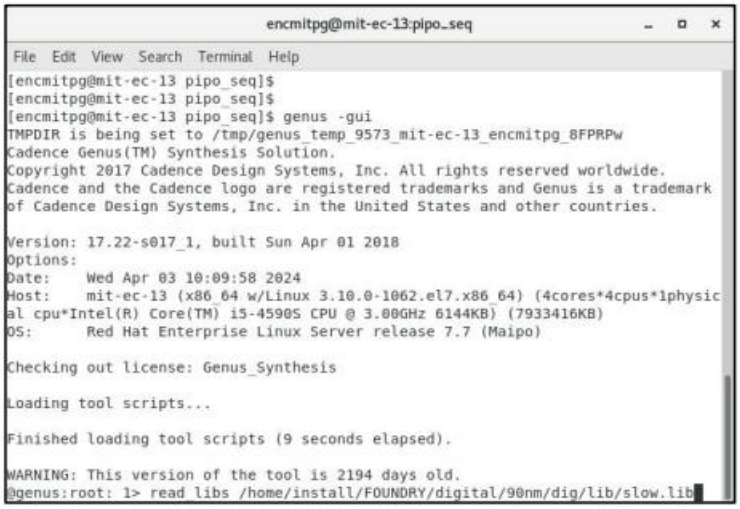


Fig. 32: Read library to specify library path

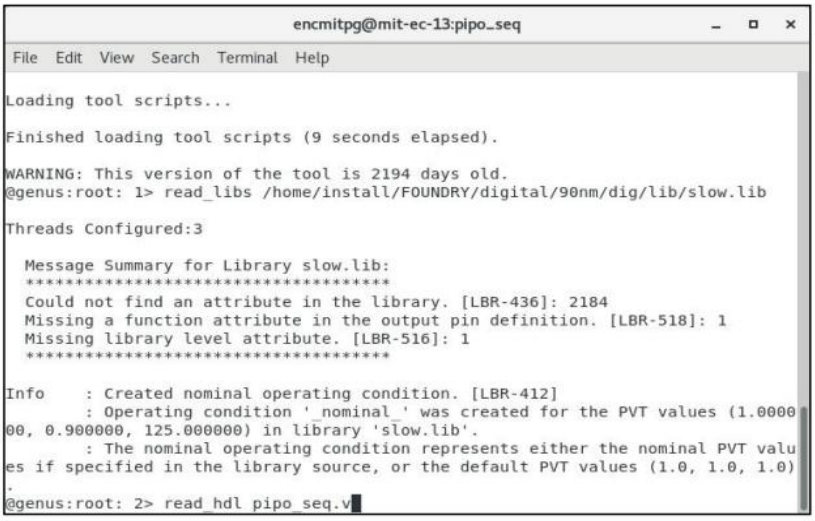


Fig. 33 read the hdl design file

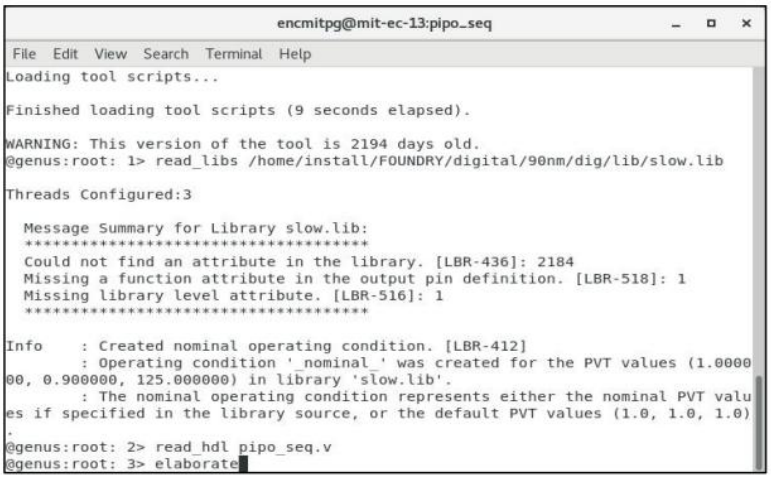


Fig. 34: elaborate

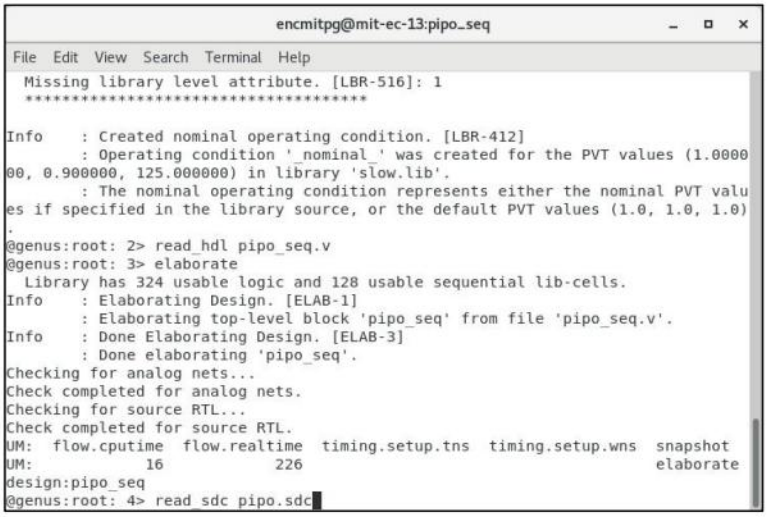


Fig. 35: read constraint file

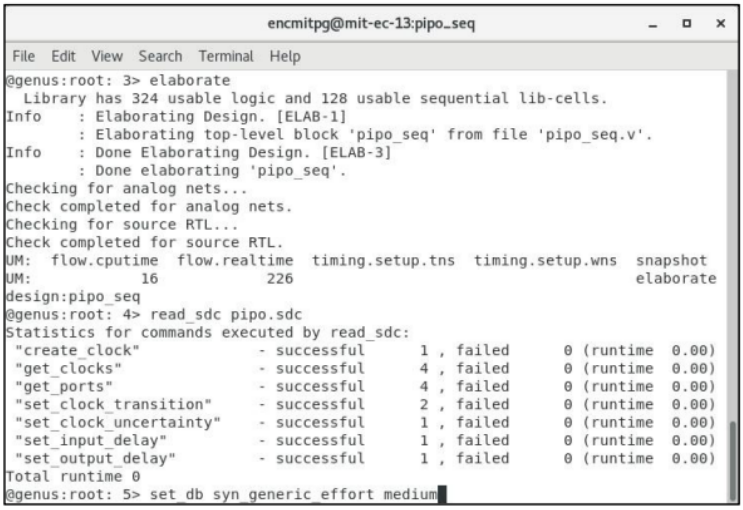


Fig. 36: set effort level for translation (generic)

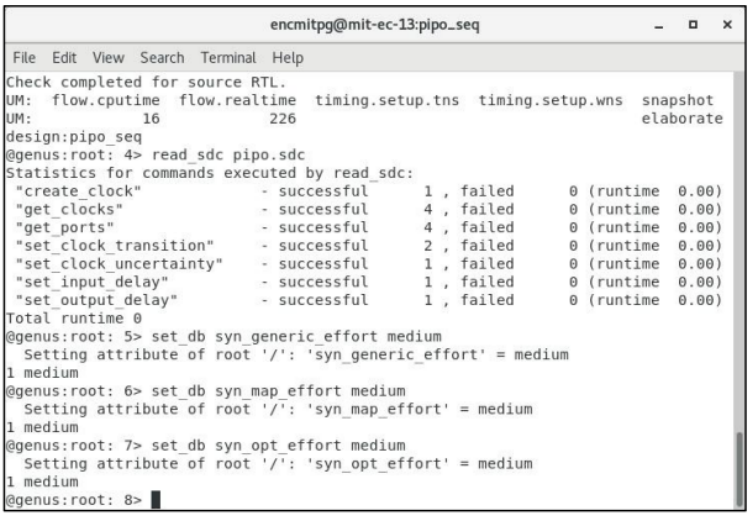


Fig. 37: set effort level for mapping and optimization

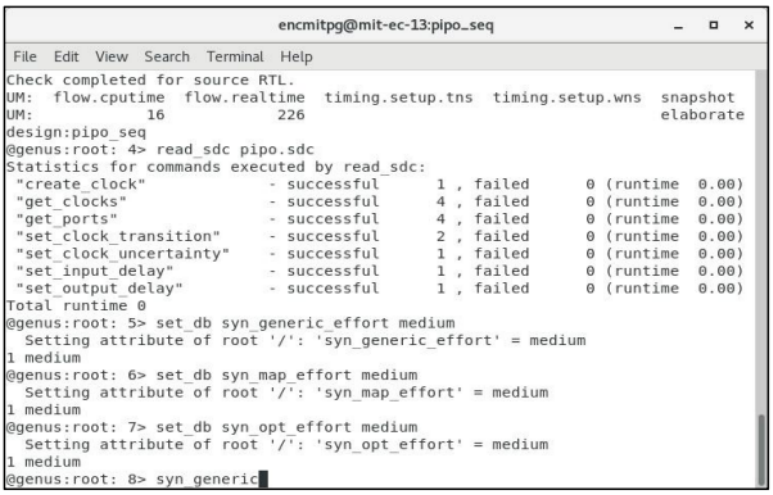


Fig. 38: perform translation

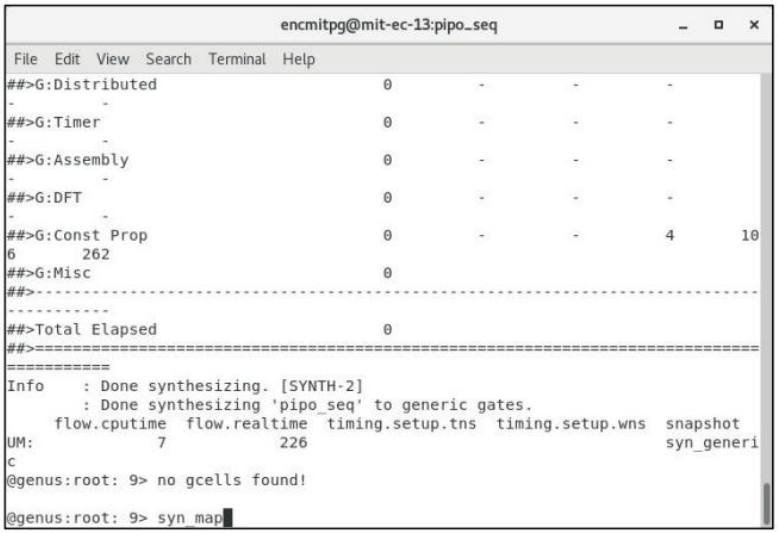


Fig. 39: Perform mapping

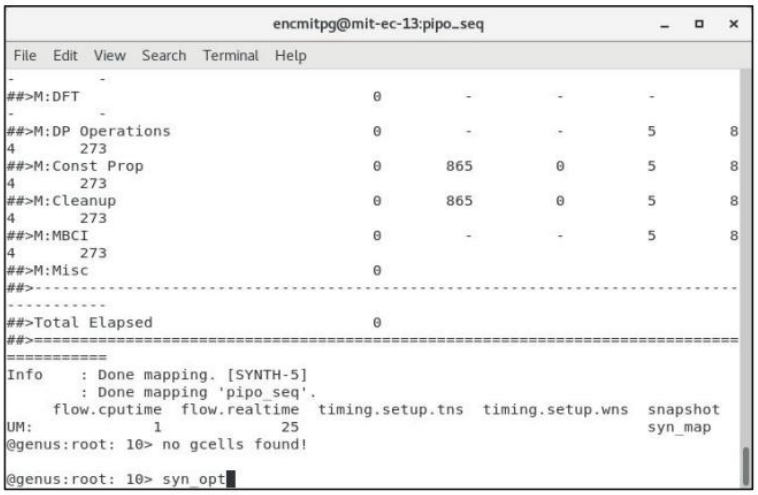


Fig. 40: perform optimization

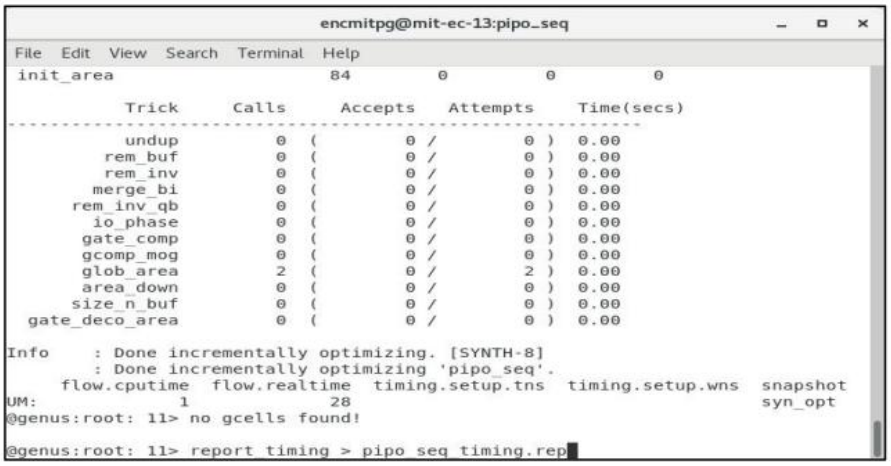


Fig. 41: generate timing report

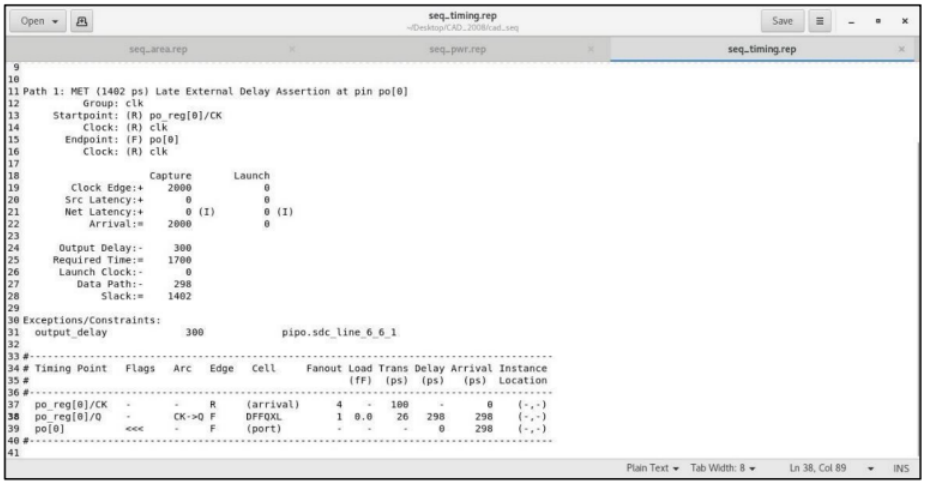


Fig. 42: Timing report

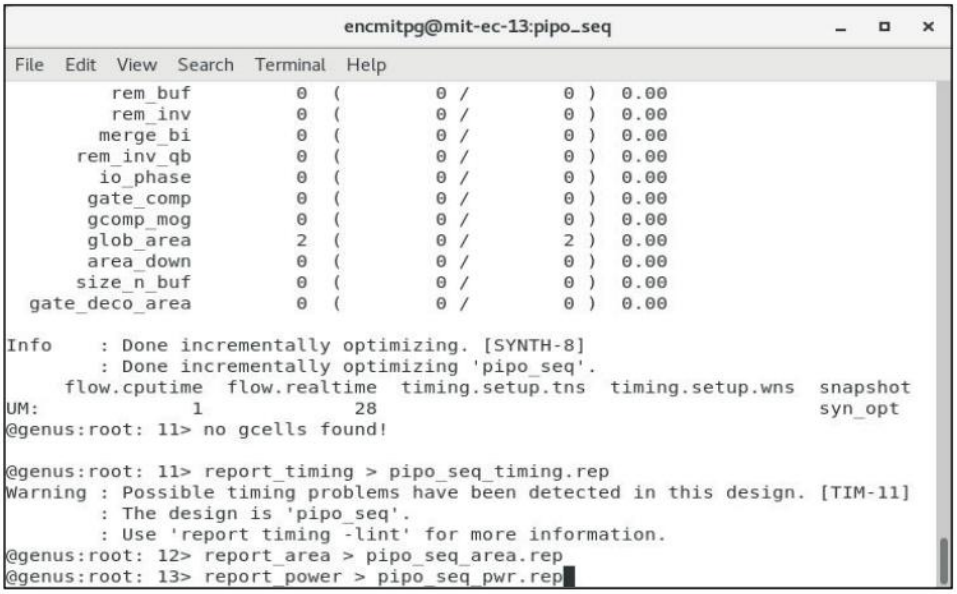


Fig. 43: Generate area and power report

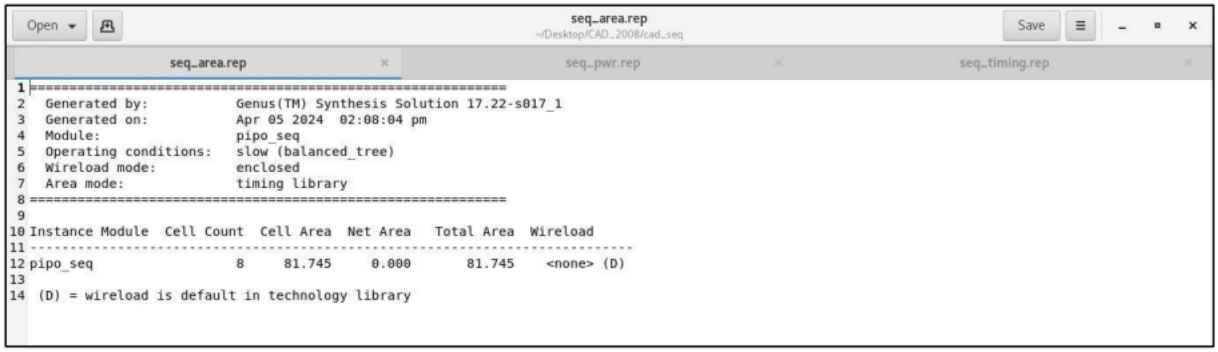


Fig. 44: Area report

Fig. 45: Power report

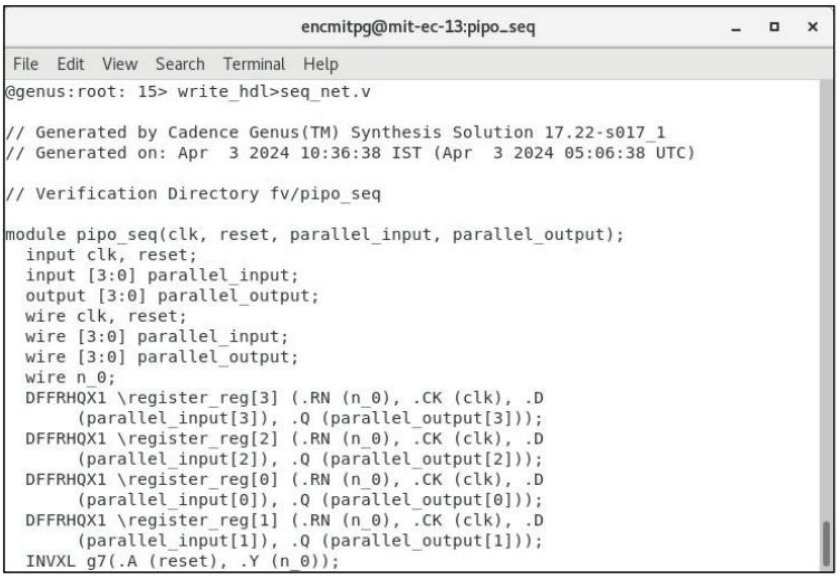


Fig. 46: Generate netlist

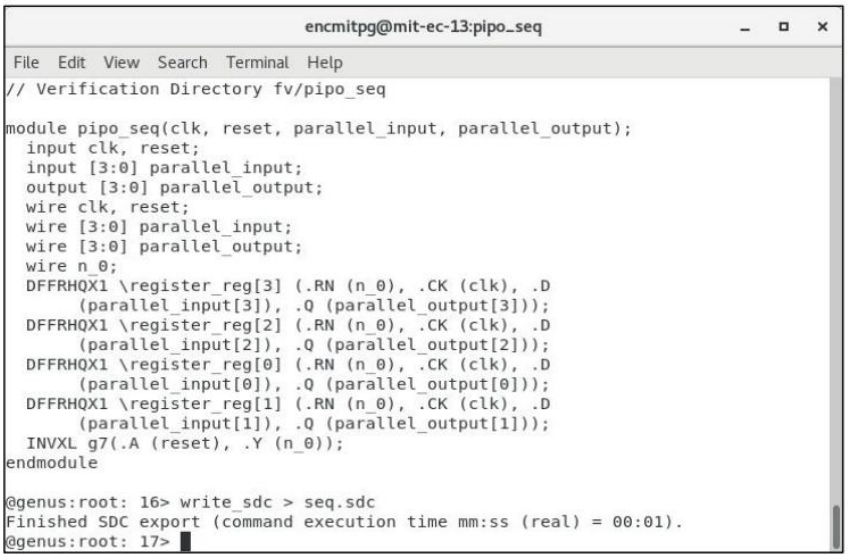


Fig. 47: Generate block level constraint file

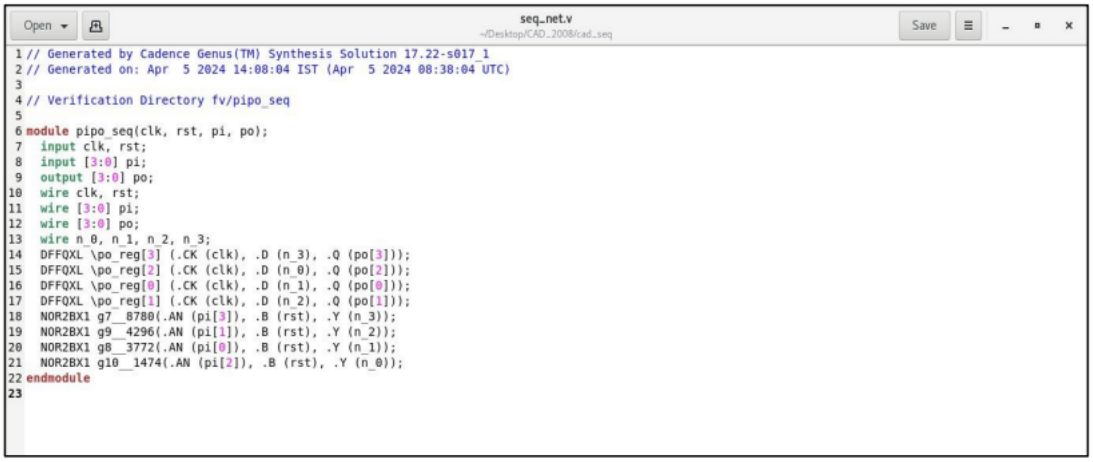


Fig. 48: Generated netlist

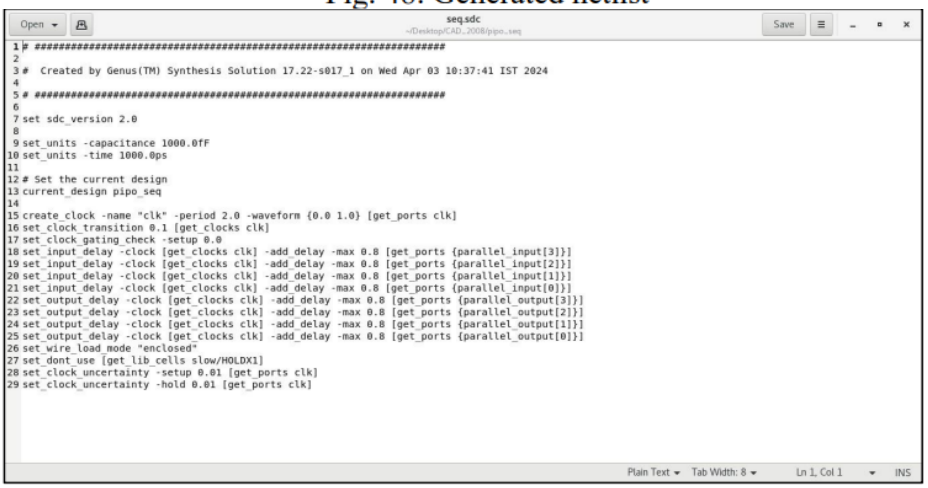


Fig. 49: Generated block level constraint file

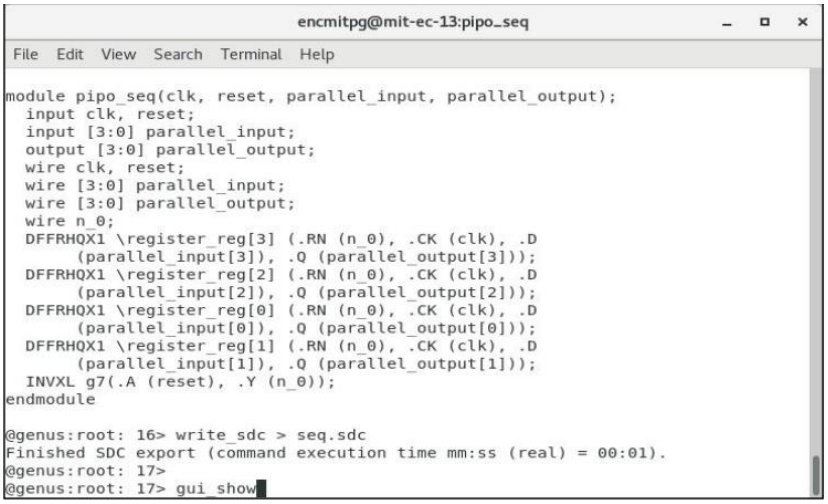


Fig. 50: Schematic generation

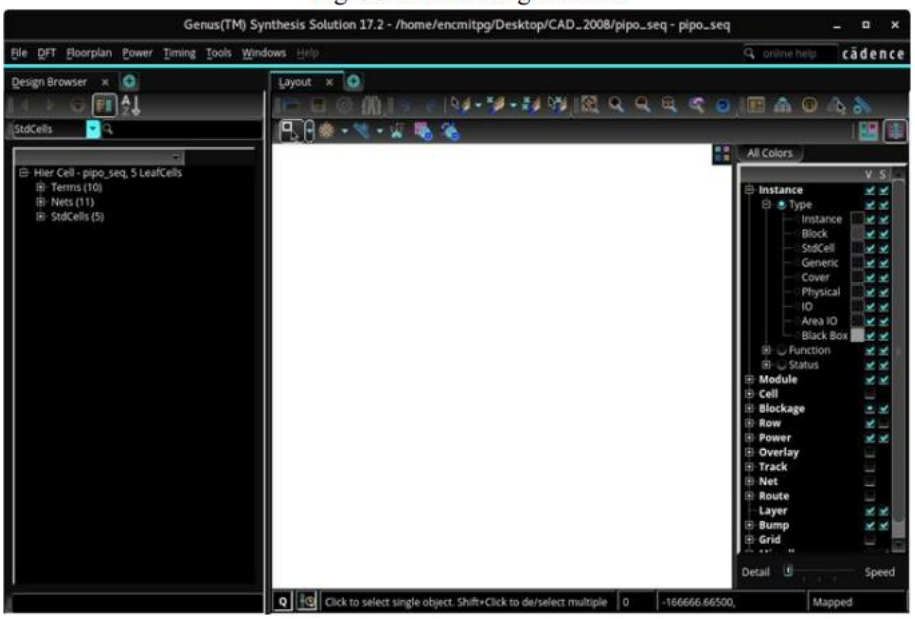
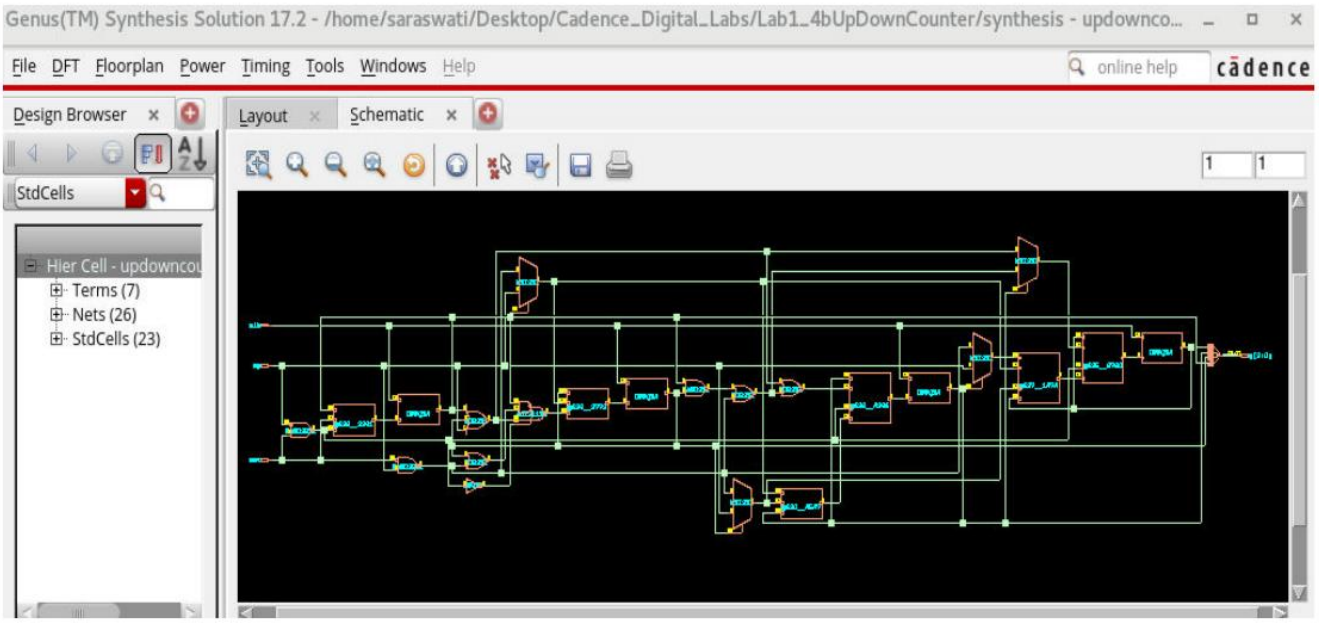


Fig. 51: Genus synthesis solution window appears

Synthesis RTL Schematic : Schematic Capture of 4bit Up-Down Counter



Commands 1-5 are intended for Synthesis process while 11-15 for Generating reports and Outputs.

Note :-
1) report_timing gives you the path with highest failing slack where
Setup Slack = Required Time – Arrival Time.

2) Worst Setup Slack ==> Highest Arrival time ==> Highest Propagation Delay.

3) Maximum Clock Frequency = 1/ (Max Data Path Delay – Min Clock Path Delay + Tsetup)

All the Information can be gathered from report_timing.

4) The Cells given in the netlist can be checked in the .lib files for their properties.

c) Compilation, Simulation and Synthesis of 32-bit Up/Down Counter.

Source Code :

```
`timescale 1ns/1ps                          //Defining a Timescale for Precision
module counter(clk,rst,m,count);            //Defining Module and Port List
input clk,rst,m;                                  //Defining Inputs
output reg [31:0]count;                     //Defining 4-bit Output as Reg type
always@(posedge clk or negedge rst)         //The Block is executed when
begin                                       //EITHER of positive edge of clock
if(!rst)                                    //or Neg Edge of Rst arrives
count=0;                                          // Both are independent events
if(m)
count=count+1;
else
count=count-1;
end
endmodule
```

Test Bench :

```
`timescale 1ns/1ps              //Creating Time Scale as in Source Code
module counter_test;              //Defining Module Name without Port List
reg clk, rst,m;                   //Defining I/P as Registers [to Hold Values]
wire [31:0] count;                //Defining O/P as Wires [To Probe Waveforms]
initial
begin
clk=0;                            //Initializing Clock and Reset
rst=0;#25;                                //All O/P is 4'b0000 from t=0 to t=25ns.
rst=1;                            //Up-Down counting is allowed at posedge clk
end
20


initial
begin
m=1;                                    //Condition for Up-Count
#600 m=0;                               //Condition for Down-Count
rst=0;#25;
rst=1;
#500 m=0;
end
counter counter1(clk,m,rst, count);    //Instantiation of Source Code
always #5 clk=~clk;                       //Inverting Clk every 5ns
initial
#1400 $finish;                           //Finishing Simulation at t=1400ns
endmodule
```

Waveform :

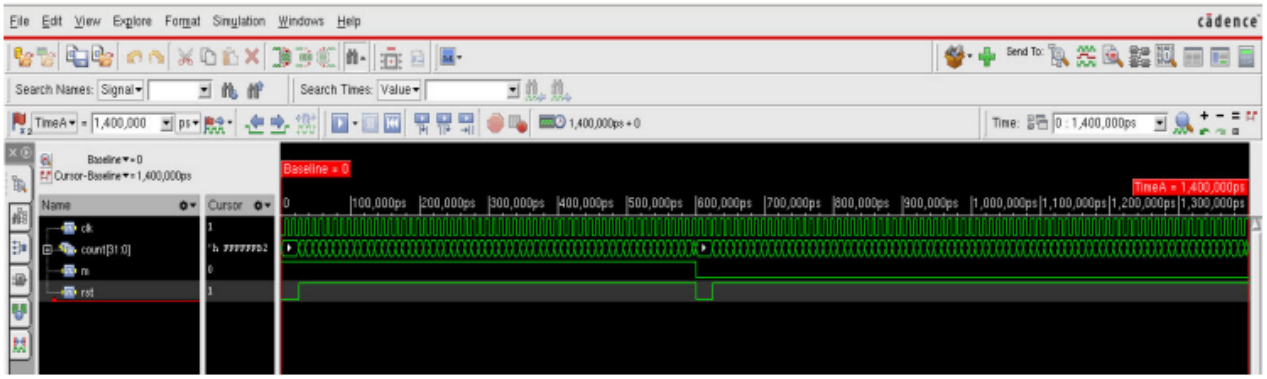


Figure No1.21: Simulation Waveform Window

The procedure for Simulation and Synthesis remains same as mentioned earlier.

- **Post Lab Task**
  - Can synthesis be possible without Constraints?
  - How constraints are important for timing analysis.

Verilog code for Full adder:

You must perform the function simulation using NCLaunch (Already discuss in previous experiments). This involves three stages :

- Step 1: Compilation- Process to check the correct Verilog language syntax and usage
- Step 2: Elaboration- To check the port connections in hierarchical design
- Step 3: Simulation- Simulate with the given test vectors over a period of time to observe the output behavior.

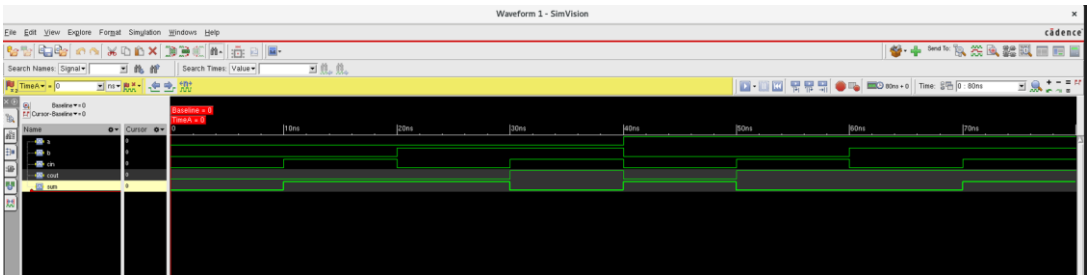
Full Adder Verilog Code:

```
module fa(input a,b,cin,
output sum,cout);

assign sum = a ^ b ^ cin;
assign cout = (a&b)|(b&cin)|(a&cin);

endmodule
```

Test Bench for Full adder:

```
module fa_tb();
reg a,b,cin;
wire sum, cout;

fa uut(.a(a), .b(b), .cin(cin), .sum(sum), .cout(cout));

initial begin
$monitor("Time=%t, a=%b, b=%b, cin=%b, sum=%b, cout=%b", $time,a,b,cin,sum,cout);
 a=0; b=0; cin=0; #10
 a=0; b=0; cin=1; #10
 a=0; b=1; cin=0; #10
 a=0; b=1; cin=1; #10
 a=1; b=0; cin=0; #10
 a=1; b=0; cin=1; #10
 a=1; b=1; cin=0; #10
 a=1; b=1; cin=1; #10

$finish;
end
endmodule
```

Waveform:



Repeat the above steps for synthesis for full adder.

Exercise problem:

1. Perform the above exercises with corner analysis includes fast.
2. Write a verilog code and performed synthesis for MOD 10 counter.
3. To write a verilog code for 4bit Adder and verify the functionality using Test bench and synthesize, Analyse Reports and Netlist, Critical Path and Max Operating Frequency.
    - Please note that the adder does not contain any clock, SDC can be skipped as an Input, but you have to consider the SDC during the synthesis.

Experiment-7

Design, synthesis, and analysis of combinational circuits.

Objective:

The main objectives of this lab are:

- Familiarization with synthesis flow.
- Synthesis the simulation without constraints or with constraints
- Generating optimized gate-level netlist and Standard Design Constraints.

Example: Full Adder Circuit-

- Write a Verilog code for Full-Adder circuit (Any style you can use)
- Simulate with NC launch
- Simulate without NC launch
- Synthesis using TCL command

Design information and block diagram-

A **Full Adder** is a combinational logic circuit that performs the addition of three binary digits: two input bits (A, B) and a carry-in bit (Cin). It produces two outputs:

- **Sum (S):** Represents the least significant bit (LSB) of the addition.
- **Carry-out (Cout):** Represents the most significant bit (MSB), which is carried to the next higher bit position in multi-bit addition.

Boolean Expressions

- **Sum (S):** $S=A \oplus B \oplus Cin$
- **Carry-out (Cout):** $Cout=(A \cdot B)+(B \cdot Cin)+(A \cdot Cin)Cout$

Truth Table of Full Adder

A	B	Cin	Sum(S)	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tool Required:

- Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim).
- Synthesis: Genus

Creating a Workspace:

- In Desktop Create a folder and name it as Digital_Vlsi (give folder name without any space).
- Create a new sub-directory and name it as pipo_seq for the design and open a terminal from the sub-directory.

Functional Simulation:

- Invoke the cadence environment by type the below commands • csh (Invokes C-Shell)
- source /home/install/cshrc (mention the path of the tools) (The path of cshrc could vary depending on the installation destination as /home/install/or /home etc.)

Creating Source Codes:

- In the Terminal, type gedit .v
- A blank document opens up into which the following source code can be typed down.

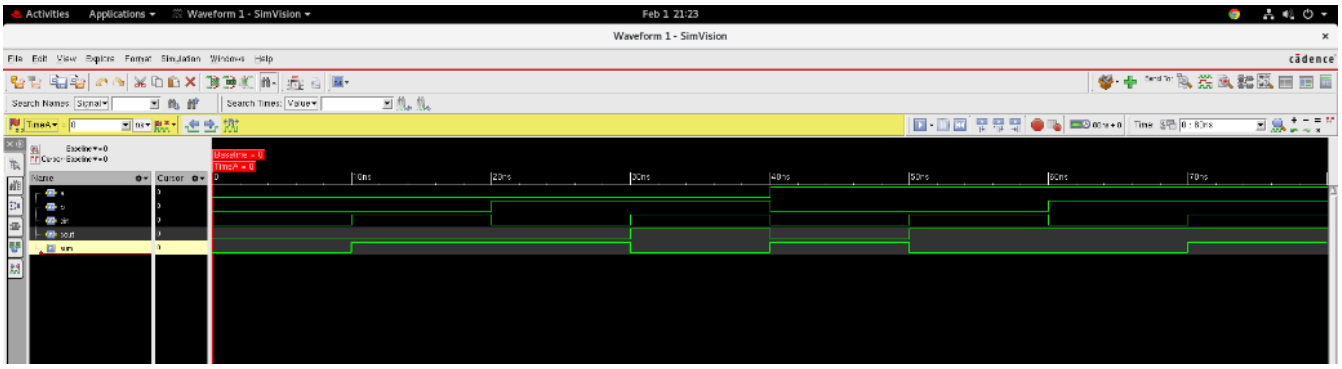
Source Code

```
module fa(input a,b,cin,
output sum,cout);

assign sum = a ^ b ^ cin;
assign cout = (a&b)|(b&cin)|(a&cin);

endmodule
```

Creating Test Bench: Similarly, create your test bench using gedit <filename_tb>.v to open a new blank document.

```
module fa_tb();
reg a,b,cin;
wire sum, cout;

fa uut(.a(a), .b(b), .cin(cin), .sum(sum), .cout(cout));

initial begin
$monitor("Time=%t, a=%b, b=%b, cin=%b, sum=%b, cout=%b", $time,a,b,cin,sum,cout);
 a=0; b=0; cin=0; #10
 a=0; b=0; cin=1; #10
 a=0; b=1; cin=0; #10
 a=0; b=1; cin=1; #10
 a=1; b=0; cin=0; #10
 a=1; b=0; cin=1; #10
 a=1; b=1; cin=0; #10
 a=1; b=1; cin=1; #10

$finish;
end
endmodule
```

Waveform Full Adder-



To Launch Simulation tool

- linux:/> nclaunch -new&
  // "-new" option is used for invoking NCVERILOG for the first time for any design
- linux:/> nclaunch&
  // On subsequent calls to NCVERILOG
- It will invoke the nclaunch window for functional simulation we can compile, elaborate and simulate it using Multiple.

To perform the function simulation, the following three steps are involved Compilation, Elaboration and Simulation.

Step 1: Compilation: Process to check the correct Verilog language syntax and usage

Inputs: Supplied are Verilog design and test bench codes

Outputs: Compiled database created in mapped library if successful, generates report else error reported in log file

Steps for compilation:

1. Create work/library directory (most of the latest simulation tools creates automatically).
2. Map the work to library created (most of the latest simulation tools creates automatically).
3. Run the compile command with compile options.

- Left side select the file and in Tools : launch verilog compiler with current selection will get enable. Click it to compile the code
- Worklib is the directory where all the compiled codes are stored while Snapshot will have output of elaboration which in turn goes for simulation

- After compilation it will come under worklib you can see in right side window.
- Select the test bench and compile it. It will come under worklib. Under Worklib you can see the module and test-bench.
- The cds.lib file is an ASCII text file. It defines which libraries are accessible and where they are located. It contains statements that map logical library names to their physical directory paths. For this Design, you will define a library called “worklib”.

Step 2: Elaboration: To check the port connections in hierarchical design

Inputs: Top level design/test bench Verilog codes

Outputs: Elaborate database updated in mapped library if successful, generates report else error reported in log file

Steps for elaboration – Run the elaboration command with elaborate options

1. It builds the module hierarchy.
2. Binds modules to module instances.
3. Computes parameter values.
4. Checks for hierarchical names conflicts.
5. It also establishes net connectivity and prepares all of this for simulation.

- After elaboration the file will come under snapshot. Select the test bench and elaborate it.

Step 3: Simulation: Simulate with the given test vectors over a period of time to observe the output behaviour.

Inputs: Compiled and Elaborated top level module name.

Outputs: Simulation log file, waveforms for debugging.

Simulation allow to dump design and test bench signals into a waveform.

Steps for simulation – Run the simulation command with simulator options.

RUN WITHOUT NCLAUNCH: Instead of NC Launch, design file and testbench can be run using single irun command. (Use your file name for .V)

```
[encmitpg@mit-ec-13 pipo_seq]$ irun pipo_seq.v pipo_tb.v -access +rwc -gui
```

Synthesize the design using Constraints and analyse reports, critical path and Max Operating Frequency

Step 1: Getting Started

- Make sure you close out all the Incisive tool windows first.
- Synthesis requires three files as follows,
  1. Liberty Files (.lib)
  2. Verilog/VHDL Files (.v or .vhdl or .vhd)
  3. SDC (Synopsis Design Constraint) File (.sdc)

Step 2 : Creating an SDC File

- In your terminal type "gedit counter_top.sdc" to create an SDC File if you do not have one.
- The SDC File must contain the following commands:

Constraint file:

i. create_clock -name clk -period 2 -waveform {0 1} [get_ports "clk"]
ii. set_clock_transition -rise 0.1 [get_clocks "clk"]
iii. set_clock_transition -fall 0.1 [get_clocks "clk"]
iv. set_clock_uncertainty 0.01 [get_ports "clk"]
v. set_input_delay -max 0.3 [get_ports "pi"] -clock [get_clocks "clk"]
vi. set_output_delay -max 0.3 [get_ports "po"] -clock [get_clocks "clk"]

i→ Creates a Clock named "clk" with Time Period 2ns and On Time from t=0 to t=1.
ii, iii → Sets Clock Rise and Fall time to 100ps.
iv → Sets Clock Uncertainty to 10ps.
v, vi → Sets the maximum limit for I/O port delay to 1ps

Step 3 : Performing Synthesis

- The Liberty files are present in the below path, /home/install/FOUNDRY/digital/<Technology_Node_number>nm/dig/lib/
- The Available technology nodes are 180nm ,90nm and 45nm.
- In the terminal, initialise the tools with the following commands if a new terminal is being used.
  - csh
  - source /home/install/cshrc
- The tool used for Synthesis is "Genus". Hence, type "genus -gui" to open the tool.
- The Following are commands to proceed :

```
1. read_libs /home/install/FOUNDRY/digital/90nm/dig/lib/slow.lib
2. read_hdl seq.v
3. elaborate
4. read_sdc pipo.sdc                    //reading top level sdc
5. set_db syn_generic_effort medium     // effort level to medium for generic, mapping
6. set_db syn_map_effort medium
7. set_db syn_opt_effort medium
8. syn_generic
9. syn_map
10. syn_opt                             //Performing Synthesis Mapping and Optimisation
11. report_timing > seq_timing.rep
    //Generates Timing report for worst datapath and dumps into file
12. report_area > seq_area.rep
    //Generates Synthesis Area report and dumps into a file
13. report_power > seq_pwr.rep
    //Generates Power Report [Pre-Layout]
14. write_hdl>seq_net.v                 //Creates readable Netlist File
15. write_sdc > seq.sdc                 //Creates Block Level SDC
16. gui_show
```

Genus Script file with .tcl file Extension

Make sure that the following file need to be present in the directory where the simulation is performed.

```
[150205105@aust synthesis_lab]$ tree
.
|-- EDI_files
|   |-- lef
|   |   `-- gsclib045.lef
|   |-- libs
|   |   |-- fast.lib
|   |   |-- slow.lib
|   |   `-- typical.lib
|   `-- others
|       `-- capTable
|-- input_files
|   `-- alu_4bit.v
`-- synthesis_cmd.tcl

5 directories, 7 files
```

Make sure the following commands are present inside the tcl (name.tcl) file

	Commands	Description
1	set_db init_lib_search_path EDI_files/libs/	Sets the value of a specific attribute. Here we are setting directory name where all the timing libraries are located.
2	set_db library slow.lib	Sets which timing library will be used while mapping
3	set_db lef_library EDI_files/lef/gsclib045.lef	Sets lef file of a target technology
4	set_db hdl_search_path input_files	Sets the directory name where RTL is located
5	read_hdl alu_4bit.v	Loads the design with pre-synthesized RTL
6	elaborate	Creates a design from Verilog module. Undefined modules are labeled as unresolved and treated as blackbox
7	set_top_module alu_4bit	Sets top module name
8	current_design alu_4bit	Changes the current directory in the design hierarchy to the specified design

9	write_hdl > alu_4bit_elaborated.v	Creates a structural netlist using generic/mapped logic
10	create_clock -name clk -period 10 [get_ports clk]	Creates a clock named "clk" having 10ns period in a specific port "clk"
11	set_clock_uncertainty -setup 0.5 [get_clocks clk]	Sets uncertainty value for the clocks while calculating setup
12	set_clock_uncertainty -hold 0.5 [get_clocks clk]	Sets uncertainty value for the clocks while calculating hold
13	set_max_transition 2 [get_ports clk]	Sets maximum allowable transition time for changing logic state to 2ns for data path
14	set_clock_transition -min -fall 0.5 [get_clocks clk]	Sets minimum allowable clock transition time to 0.5ns for switching logic state from high to low for clock path
15	set_clock_transition -min -rise 0.5 [get_clocks clk]	Sets minimum allowable clock transition time to 0.5ns for switching logic state from low to high for clock path
16	set_clock_transition -max -fall 0.5 [get_clocks clk]	Sets maximum allowable clock transition time to 0.5ns for switching logic state from high to low for clock path
17	set_clock_transition -max -rise 0.5 [get_clocks clk]	Sets maximum allowable clock transition time to 0.5ns for switching logic state from low to high for clock path
18	set_clock_groups -name original -group [list [get_clocks clk]]	Defines groups of specific clocks
19	set DRIVING_CELL BUFX8	Defines driving cell name which will drive the input ports of the design
20	set DRIVE_PIN {Y}	Defines driver pin of the driving cell
21	set_driving_cell -lib_cell $DRIVING_CELL -pin $DRIVE_PIN [all_inputs]	Sets driving cell properties for all the input ports
22	set_max_fanout 10 [current_design]	Sets maximum allowable fanout number to 10
23	set_load 0.5 [all_outputs]	Sets load capacitance of the output ports of the design
24	set_operating_conditions slow	Sets operating condition for delay calculation
25	set_input_delay -max 0.5 [all_inputs]	Synthesis tool assumes the data is launched by a positive edge triggered flop from the external logic

		(and the maximum input delay for the setup analysis is 0.5ns)
26	set_output_delay -max 0.5 [all_outputs]	Synthesis tool assumes the data is captured by a positive edge triggered flop in the external logic (and the maximum output delay for the setup analysis is 0.5ns)
27	remove_assign -buffer_or_inverter BUFX16 -design [current_design]	Removes assign statement using BUFX16 cell
28	syn_generic	Performs generic synthesis
29	write_hdl > alu_4bit_generic.v	Creates a structural netlist using generic logic after generic synthesis
30	synthesize -to_mapped	Performs mapping using target timing library
31	write_hdl > alu_4bit_post_synthesis.v	Creates a structural netlist using mapped logic after mapping
32	remove_assigns_without_opt -buffer_or_inverter BUFX12 -verbose	Removes assign statement using BUFX12 cell
33	set_remove_assign_options -buffer_or_inverter BUFX12 -verbose	Sets buffer or inverter cell to remove assign statements
34	write -mapped > alu_4bit_mapped.v	Writes mapped netlist for post-synthesis flow
35	write_sdc > alu_4bit.sdc	Writes constraints file for post-synthesis flow

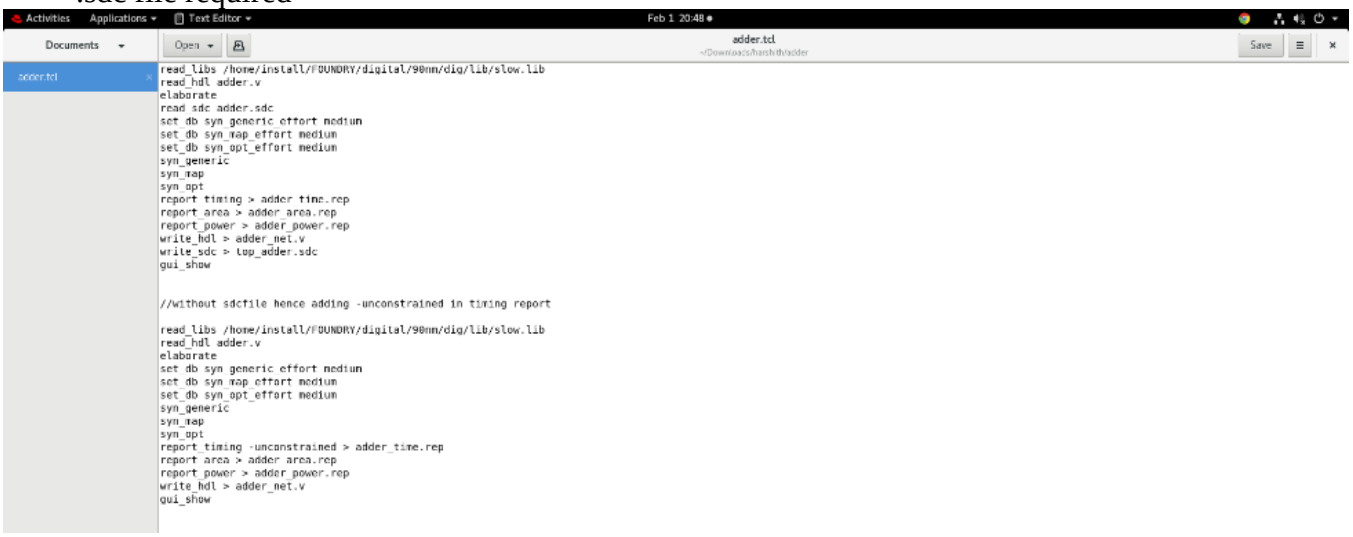
Steps to create tcl file

Fig. 53: Create tcl file

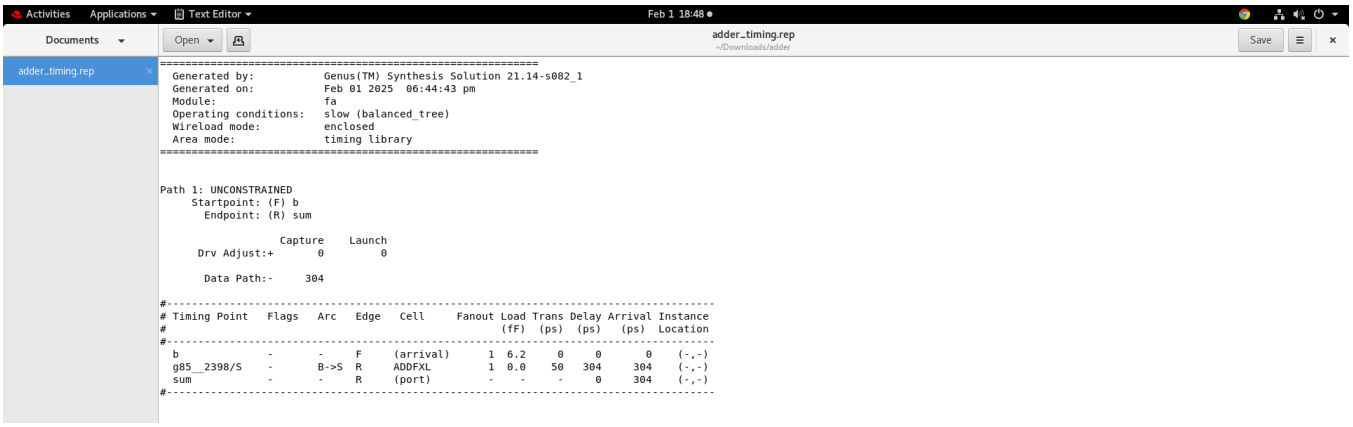
Write a tcl scripting file-

- If the circuits do not contain any clock signal then you can skip the .sdc file using the command "-unconstrained".
- Further if you want to add delay information/ any input information for any circuits then

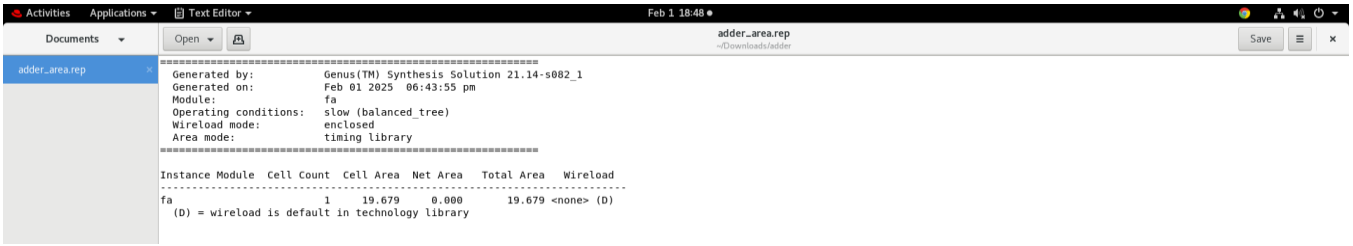
.sdc file required



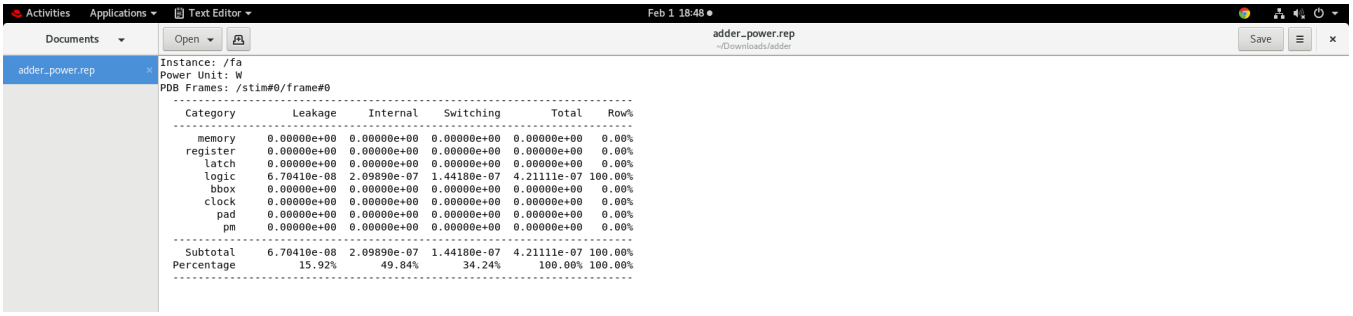
Run the created tcl file-



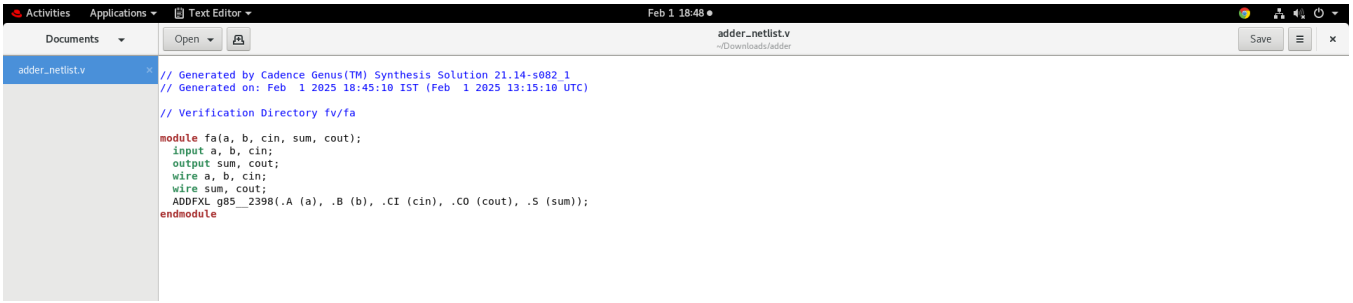
Area report-



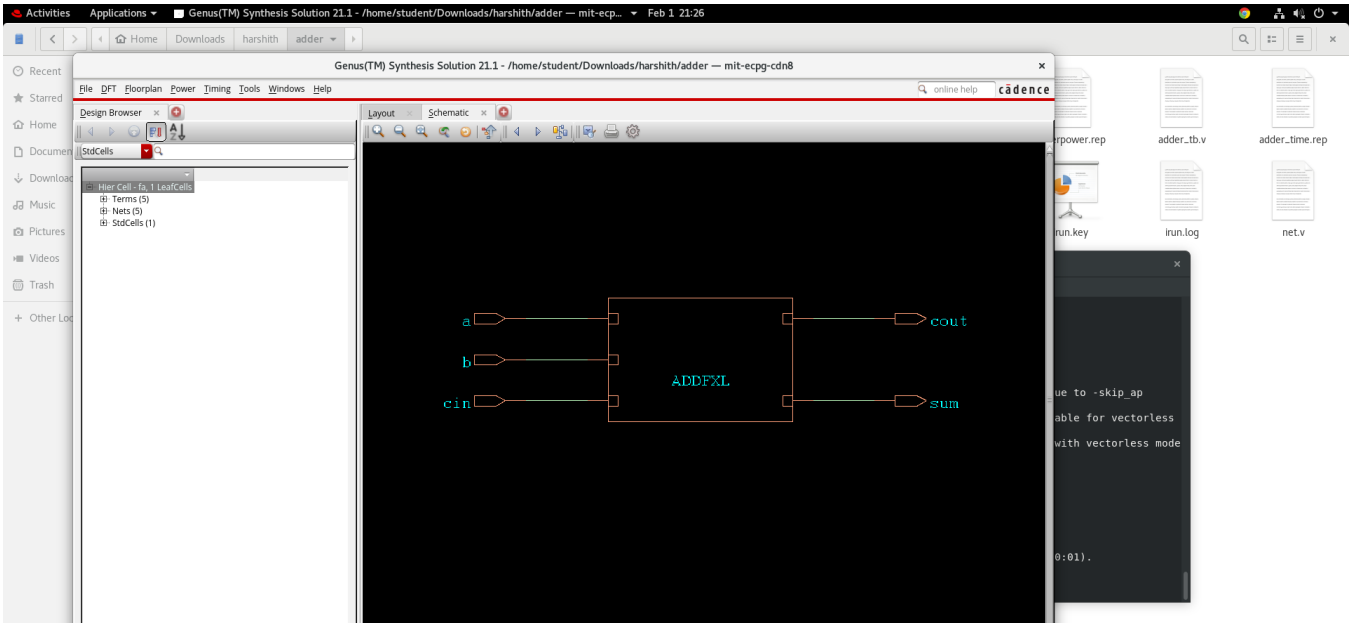
Power consumption report-



Netlist files-



Schematic Diagram-



- To show the synthesized output execute the command gui_show
- The GUI window of the genus synthesis output will be opened. If you click and zoom into each block of the circuit, you will be able to view the gate interconnections inside the block.

Exercise problem:

1. Write a Verilog code for the following expression Q= (AB)'.(A+B)'.C and do the synthesis. Further, calculate the power area and performance with and without using the constraint file.
2. Repeat the first problem by giving a delay to the input signal and checking the PPA.
3. s

Homework

- **Exercise Problem- 1**

  A Line follower Robot uses 2 Infra-Red (IR) sensors to detect the color of the surface. Sensor output is High when white surface is detected and Low on a black surface. Try to model the Robot controller and then implement its hardware? A Line follower Robot uses 2 Infra-Red (IR) sensors to detect the color of the surface. Sensor output is High when white surface is detected and Low on a black surface. Try to model the Robot controller and then implement its hardware?

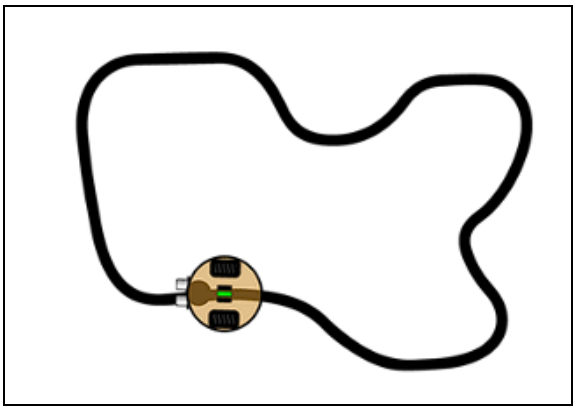


Figure 1: Line Follower

Source Code:

```
module LineFollowerRobot (
    input clk, reset,
    input sensorLeft, sensorRight, // Sensors detecting black or white surfaces
    output reg motorLeft, motorRight
);


parameter STRAIGHT = 2'b00, TURN_LEFT = 2'b01, TURN_RIGHT = 2'b10, STOP = 2'b11;
reg [1:0] state, next_state;

always @(posedge clk or posedge reset) begin
    if (reset)
        state <= STRAIGHT;
    else
        state <= next_state;
end


always @(*) begin

    case (state)
        STRAIGHT: begin
            if (sensorLeft == 1 && sensorRight == 1)
                next_state <= STRAIGHT;
            else if (sensorLeft == 0 && sensorRight == 1)
                next_state <= TURN_LEFT;
            else if (sensorLeft == 1 && sensorRight == 0)
                next_state <= TURN_RIGHT;
            else
                next_state <= STOP;
        end
```

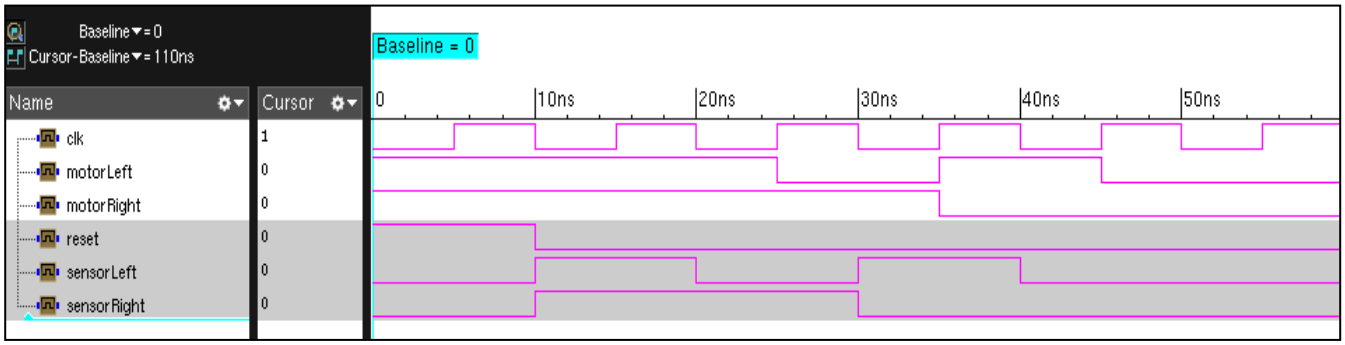
```
        TURN_LEFT: begin
            if (sensorLeft == 0 && sensorRight == 1)
                next_state <= TURN_LEFT;
            else if (sensorLeft == 1 && sensorRight == 0)
                next_state <= TURN_RIGHT;
            else
                next_state <= STOP;
        end
        TURN_RIGHT: begin
            if (sensorLeft == 1 && sensorRight == 0)
                next_state <= TURN_RIGHT;
            else if (sensorLeft == 0 && sensorRight == 1)
                next_state <= TURN_LEFT;
            else
                next_state <= STOP;
        end
        STOP: begin
            if (sensorLeft == 0 && sensorRight == 0)
                next_state <= STOP;
            else
                next_state <= STRAIGHT;
        end
        default: next_state <= STRAIGHT;
    endcase
end
always @(*) begin
    case (state)
        STRAIGHT: begin
            motorLeft <= 1;
            motorRight <= 1;
        end
        TURN_LEFT: begin
            motorLeft <= 0;
            motorRight <= 1;
        end
        TURN_RIGHT: begin
            motorLeft <= 1;
            motorRight <= 0;
        end
        STOP: begin
            motorLeft <= 0;
            motorRight <= 0;
        end
default: begin
            motorLeft <= 1;
            motorRight <= 1;
        end
    endcase
```

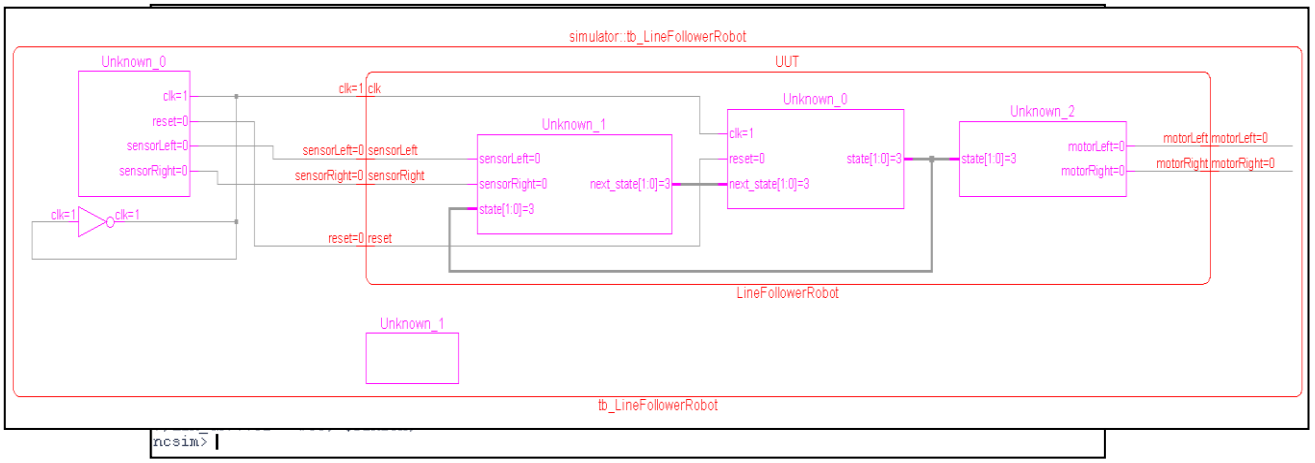
```
end
endmodule
```

Testbench:

```
module tb_LineFollowerRobot();
   reg clk, reset;
   reg sensorLeft, sensorRight;
   wire motorLeft, motorRight;

   LineFollowerRobot UUT (
      .clk(clk),
      .reset(reset),
      .sensorLeft(sensorLeft),
      .sensorRight(sensorRight),
      .motorLeft(motorLeft),
      .motorRight(motorRight)
   );

      always #5 clk = ~clk;

initial begin
      clk=0;
      reset = 0;
sensorLeft = 0;
sensorRight = 0;

reset = 1;#10;
      reset = 0;

      sensorLeft = 1; sensorRight = 1; #10;
      sensorLeft = 0; sensorRight = 1; #10;
      sensorLeft = 1; sensorRight = 0; #10;
      sensorLeft = 0; sensorRight = 0; #10;
#60; $finish;
   end
initial begin
$monitor("Time %t: clk=%b, reset=%b, sensorLeft=%b,
sensorRight=%b,motorLeft=%b,motorRight=%b",
$time,clk,reset,sensorLeft,sensorRight,motorLeft,motorRight);
end
endmodule
```

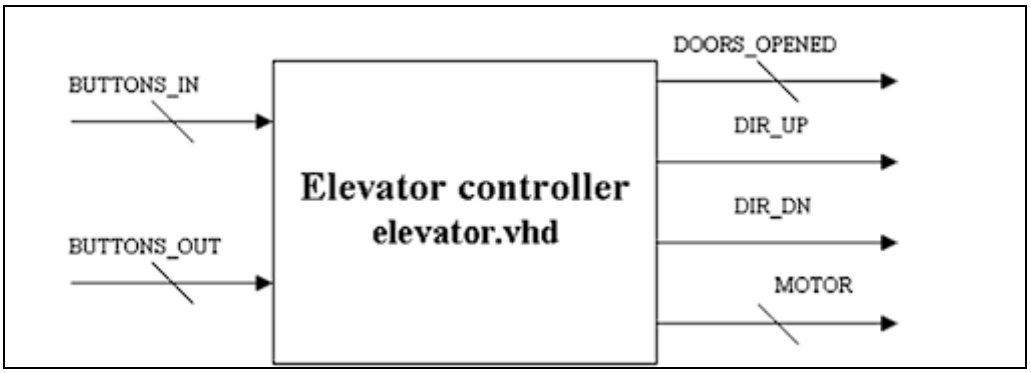
Waveform and Simulation:





- **Exercise Problem- 2**

1. **Model and implement an Elevator control system handling 4 floors (0 to 3)**



Source Code:

```
module elevator(
    input clk,
    input reset,
    input [1:0] current_floor,
    input [3:0] buttons_in,
    input [3:0] buttons_out,
    output reg [3:0] motor,
    output reg dir_up,
    output reg dir_down
);

    reg [3:0] floor_requests;
    reg [1:0] next_floor;

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            motor <= 4'b0000;
            dir_up <= 1'b0;
```

```
    dir_down <= 1'b0;
    next_floor <= current_floor;
    floor_requests <= 4'b0000;
end
else begin

    floor_requests <= buttons_in | buttons_out;

    if (floor_requests[current_floor]) begin
        motor <= 4'b0000;  // Stop at current floor
        dir_up <= 1'b0;
        dir_down <= 1'b0;
    end
    else begin
        if (current_floor == 2'b00 && floor_requests[1]) begin
            next_floor <= 2'b01;
            motor <= 4'b0010;
            dir_up <= 1'b1;
            dir_down <= 1'b0;
        end
        else if (current_floor <= 2'b01 && floor_requests[2]) begin
            next_floor <= 2'b10;
            motor <= 4'b0100;
            dir_up <= 1'b1;
            dir_down <= 1'b0;
        end
        else if (current_floor <= 2'b10 && floor_requests[3]) begin
            next_floor <= 2'b11;
```

```
            motor <= 4'b1000;
            dir_up <= 1'b1;
            dir_down <= 1'b0;
        end

        else if (current_floor >= 2'b01 && floor_requests[0]) begin
            next_floor <= 2'b00;
            motor <= 4'b0001;
            dir_up <= 1'b0;
            dir_down <= 1'b1;
        end

else if (current_floor >= 2'b10 && floor_requests[1]) begin
            next_floor <= 2'b01;
            motor <= 4'b0010;
            dir_up <= 1'b0;
            dir_down <= 1'b1;
        end
        else if (current_floor >= 2'b11 && floor_requests[2]) begin
            next_floor <= 2'b10;
            motor <= 4'b0100;
            dir_up <= 1'b0;
            dir_down <= 1'b1;
        end
        else begin
            motor <= 4'b0000; // No movement if no requests in any direction
            dir_up <= 1'b0;
            dir_down <= 1'b0;
```

```
            end
        end
    end
  end
endmodule
```

Testbench:

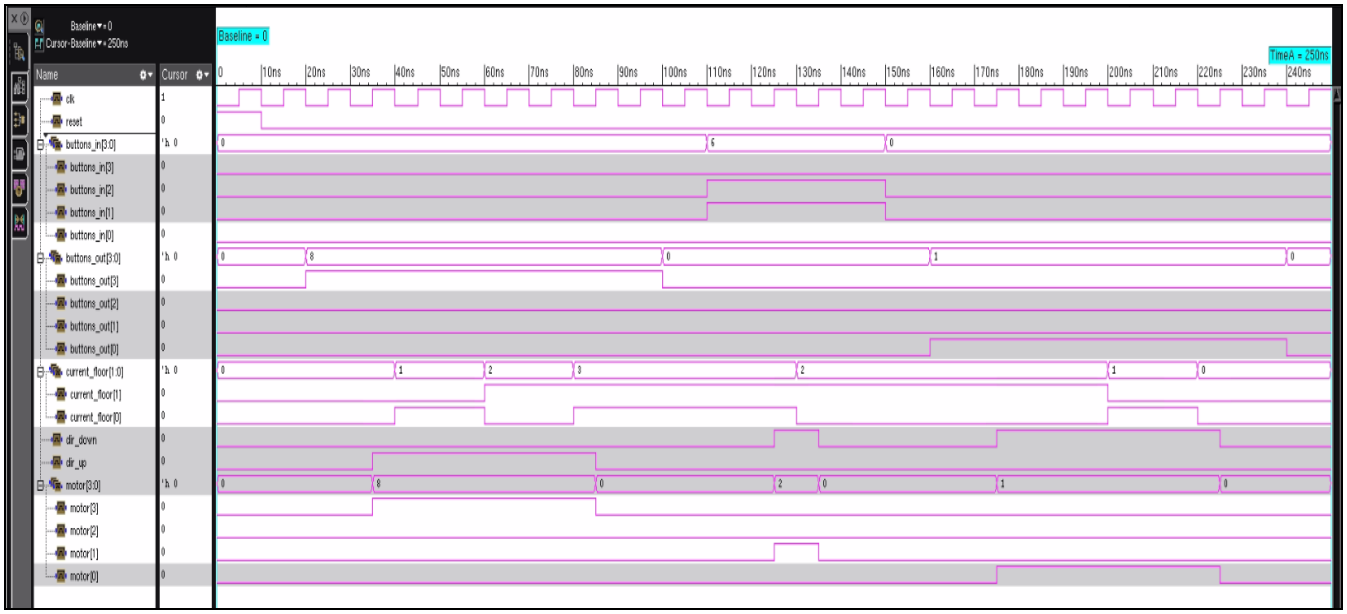
```
module elevator_tb;

  reg clk;
  reg reset;
  reg [1:0] current_floor;
  reg [3:0] buttons_in;
  reg [3:0] buttons_out;
  wire [3:0] motor;
  wire dir_up;
  wire dir_down;


  elevator uut (
    .clk(clk),
    .reset(reset),
    .current_floor(current_floor),

 .buttons_in(buttons_in),
    .buttons_out(buttons_out),
```

```
    .motor(motor),
    .dir_up(dir_up),
    .dir_down(dir_down)
);


always #5 clk = ~clk;

initial begin
    clk = 0;
    reset = 1;
    current_floor = 2'b00;
    buttons_in = 4'b0000;
    buttons_out = 4'b0000;


    #10;
    reset = 0;

    // Request to go to the 3rd floor (external)
    #10;
    buttons_out = 4'b1000;  // Request for 3rd floor
    #20;
    current_floor = 2'b01;  // Move to 1st floor
    #20;
    current_floor = 2'b10;  // Move to 2nd floor
    #20;
    current_floor = 2'b11;  // Reach 3rd floor
```

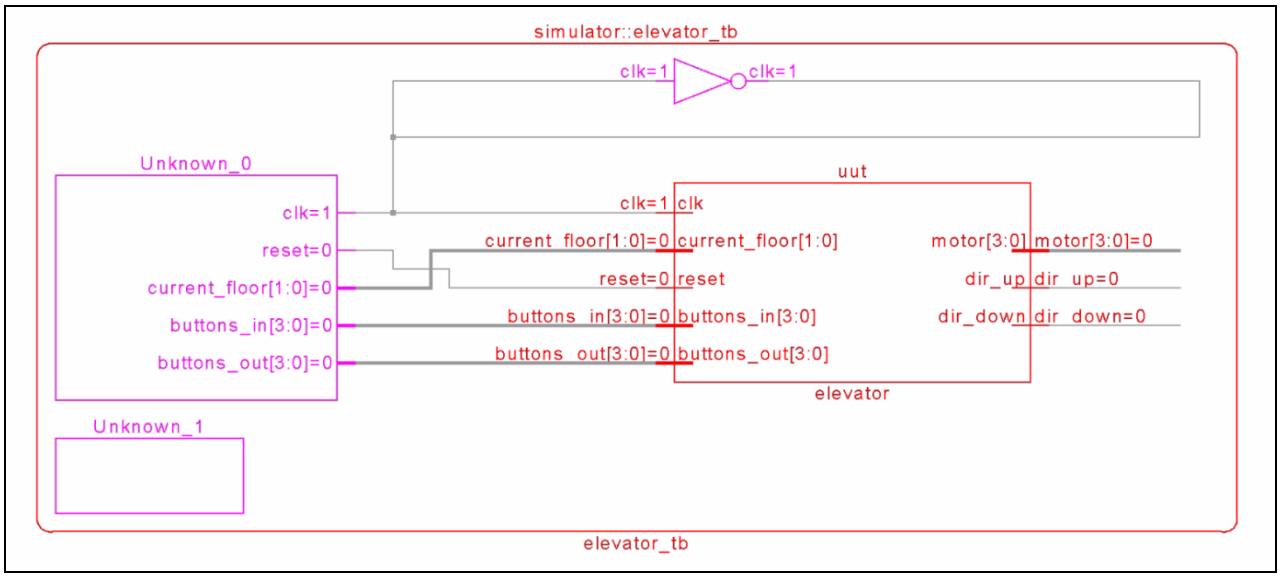
```
    #20;
    buttons_out = 4'b0000;  // Clear requests

    // Request from 1st and 2nd floors (internal)
    #10;
    buttons_in = 4'b0110;   // Requests for 1st and 2nd floors
    #20;
    current_floor = 2'b10;  // Move to 2nd floor
    #20;
    buttons_in = 4'b0000;   // Clear requests

    // Request to go down to ground floor (external)
    #10;
    buttons_out = 4'b0001;  // Request for ground floor
    #20;
    current_floor = 2'b10;  // Move to 2nd floor
    #20;
    current_floor = 2'b01;  // Move to 1st floor
    #20;
    current_floor = 2'b00;  // Reach ground floor
    #20;
    buttons_out = 4'b0000;  // Clear requests


    $finish;
end
```

```
    initial begin
        $monitor("Time=%0t | Floor=%0d | Buttons In=%b | Buttons Out=%b | Motor=%b | Dir_Up=%b | Dir_Down=%b",
                 $time, current_floor, buttons_in, buttons_out, motor, dir_up, dir_down);
    end

endmodule
```

Waveform and Simulation:



```
ncsim>
ncsim> source /home/install/INCISIVE152/tools/inca/files/ncsimrc
ncsim> database -open waves -into waves.shm -default
Created default SHM database waves
ncsim> probe -create -shm elevator_tb.buttons_in elevator_tb.buttons_out elevator_tb.clk eleva
Created probe 1
ncsim> run
Time=0 | Floor=0 | Buttons In=0000 | Buttons Out=0000 | Motor=0000 | Dir_Up=0 | Dir_Down=0
Time=20 | Floor=0 | Buttons In=0000 | Buttons Out=1000 | Motor=0000 | Dir_Up=0 | Dir_Down=0
Time=35 | Floor=0 | Buttons In=0000 | Buttons Out=1000 | Motor=1000 | Dir_Up=1 | Dir_Down=0
Time=40 | Floor=1 | Buttons In=0000 | Buttons Out=1000 | Motor=1000 | Dir_Up=1 | Dir_Down=0
Time=60 | Floor=2 | Buttons In=0000 | Buttons Out=1000 | Motor=1000 | Dir_Up=1 | Dir_Down=0
Time=80 | Floor=3 | Buttons In=0000 | Buttons Out=1000 | Motor=1000 | Dir_Up=1 | Dir_Down=0
Time=85 | Floor=3 | Buttons In=0000 | Buttons Out=1000 | Motor=0000 | Dir_Up=0 | Dir_Down=0
Time=100 | Floor=3 | Buttons In=0000 | Buttons Out=0000 | Motor=0000 | Dir_Up=0 | Dir_Down=0
Time=110 | Floor=3 | Buttons In=0110 | Buttons Out=0000 | Motor=0000 | Dir_Up=0 | Dir_Down=0
Time=125 | Floor=3 | Buttons In=0110 | Buttons Out=0000 | Motor=0010 | Dir_Up=0 | Dir_Down=1
Time=130 | Floor=2 | Buttons In=0110 | Buttons Out=0000 | Motor=0010 | Dir_Up=0 | Dir_Down=1
Time=135 | Floor=2 | Buttons In=0110 | Buttons Out=0000 | Motor=0000 | Dir_Up=0 | Dir_Down=0
Time=150 | Floor=2 | Buttons In=0000 | Buttons Out=0000 | Motor=0000 | Dir_Up=0 | Dir_Down=0
Time=160 | Floor=2 | Buttons In=0000 | Buttons Out=0001 | Motor=0000 | Dir_Up=0 | Dir_Down=0
Time=175 | Floor=2 | Buttons In=0000 | Buttons Out=0001 | Motor=0001 | Dir_Up=0 | Dir_Down=1
Time=200 | Floor=1 | Buttons In=0000 | Buttons Out=0001 | Motor=0001 | Dir_Up=0 | Dir_Down=1
Time=220 | Floor=0 | Buttons In=0000 | Buttons Out=0001 | Motor=0001 | Dir_Up=0 | Dir_Down=1
Time=225 | Floor=0 | Buttons In=0000 | Buttons Out=0001 | Motor=0000 | Dir_Up=0 | Dir_Down=0
Time=240 | Floor=0 | Buttons In=0000 | Buttons Out=0000 | Motor=0000 | Dir_Up=0 | Dir_Down=0
Simulation complete via $finish(1) at time 250 NS + 0
./elevator_tb.v:74          $finish;
ncsim>
```



Experiment - 08

Design, synthesis, and analysis of sequential circuits.

Objective:

The objective of RTL synthesis of a sequential digital circuit is to transform a high-level functional description of the circuit (Verilog) into a lower-level, technology-specific gate-level representation that can be implemented in hardware. Further, students are able to run the simulation using the TCL script for synthesis of any sequential circuit.

Exercise Problem: 4-bit PIPO (parallel – in parallel -out) shift register circuit.

- Write a verilog code for 4-bit PIPO shift register circuit
- Simulate with nclaunch
- Simulate without nclaunch
- Synthesis using tcl command

Modelling Style: Sequential Modelling

Tools required:

- Functional Simulator
- Incisive Simulator
- Synthesis: Genus
- Physical Design: Innovus

Design Information and Bock Diagram

A 4-bit PIPO (Parallel In Parallel Out) shift register is a digital circuit commonly used in digital systems for various purposes such as data storage, parallel-to-serial or serial-to-parallel conversion, and delay elements.

A 4-bit PIPO shift register consists of four flip-flops (memory cells) capable of storing binary data. Each flip-flop stores one bit of data, allowing the register to hold a 4-bit binary word. The register has four parallel input lines, one for each bit. When new data is presented at the input, it is loaded simultaneously into all four flip-flops, updating the contents of the register. The register has four parallel output lines, one for each bit. These output lines allow the current contents of the register to be read out in parallel, providing access to the stored data.

The PIPO shift register typically doesn't perform shifting operations. Instead, it holds the data statically until new data is loaded into it. The "Parallel-In Parallel-Out" nature of the register means that data is both input and output simultaneously, without any shifting.

Applications: PIPO shift registers find applications in various digital systems where parallel data storage and retrieval are necessary. They can be used in microcontrollers, microprocessors, digital signal processing, and communication systems for tasks such as data buffering, temporary storage, and interfacing between parallel and serial data streams.

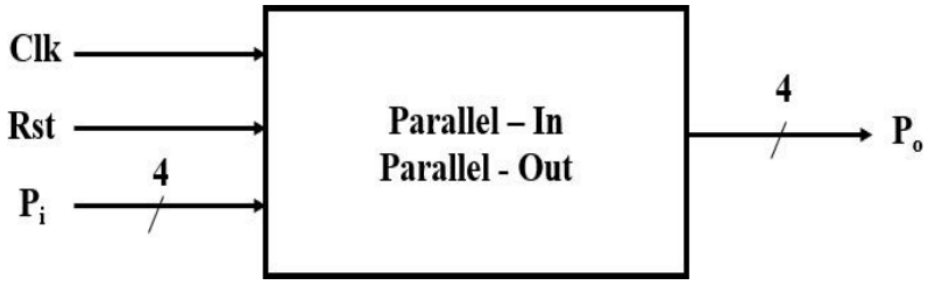


Fig. 1: Block diagram of 4-bit PIPO

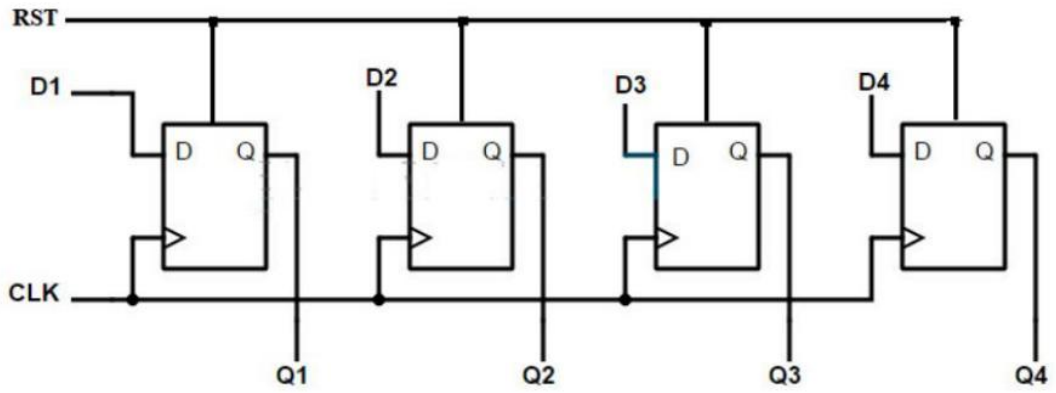


Fig. 2: Block diagram using D flipflop

Tool Required:

- Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim).
- Synthesis: Genus

Creating a Workspace:

- In Desktop Create a folder and name it as Digital_Vlsi (give folder name without any space).
- Create a new sub-directory and name it as pipo_seq for the design and open a terminal from the sub-directory.

Functional Simulation:

- Invoke the cadence environment by type the below commands • csh (Invokes C-Shell)
- source /home/install/cshrc (mention the path of the tools) (The path of cshrc could vary depending on the installation destination as /home/install/or /home etc.)

Creating Source Codes:

- In the Terminal, type gedit .v
- A blank document opens up into which the following source code can be typed down.

Source Code

```
 1 module pipo_seq(clk,rst, pi, po);
 2 input clk,rst;
 3 input [3:0] pi;
 4 output [3:0] po;
 5 wire [3:0] pi;
 6 reg [3:0] po;
 7
 8 always @(posedge clk)
 9
10 begin
11 if (rst)
12 po<= 4'b0000;
13 else
14 po <= pi;
15 end
16
17 endmodule
```

Creating Test Bench: Similarly, create your test bench using gedit <filename_tb>.v to open a new blank document.

```
 1 module pipo_tb;
 2 reg clk;
 3 reg rst;
 4 reg [3:0] pi;
 5 wire [3:0] po;
 6
 7 pipo_seq uut (.clk(clk),.rst(rst),.pi(pi),.po(po) );
 8
 9 initial begin
10 clk = 0;
11 rst = 0;
12 pi = 4'b0001;
13 #5 rst=1'b1;
14 #5 rst=1'b0;
15 #10 pi=4'b1001;
16 #10 pi=4'b1010;
17 #10 pi=4'b1011;
18 #10 pi=4'b1110;
19 #10 pi=4'b1111;
20 #10 pi=4'b0000;
21 end
22
23 always #5 clk = ~clk;
24
25 initial #100 $finish;
26 initial
27 $monitor ("Time: %0t, pi: %b, po: %b", $time, pi, po);
28
29 endmodule
```

To Launch Simulation tool

- linux:/> nclaunch -new&
  // "-new" option is used for invoking NCVERILOG for the first time for any design
- linux:/> nclaunch&
  // On subsequent calls to NCVERILOG
- It will invoke the nclaunch window for functional simulation we can compile, elaborate and simulate it using Multiple.

To perform the function simulation, the following three steps are involved Compilation, Elaboration and Simulation.

Step 1: Compilation: Process to check the correct Verilog language syntax and usage

Inputs: Supplied are Verilog design and test bench codes

Outputs: Compiled database created in mapped library if successful, generates report else error reported in log file

Steps for compilation:

1. Create work/library directory (most of the latest simulation tools creates automatically).
2. Map the work to library created (most of the latest simulation tools creates automatically).
3. Run the compile command with compile options.

- Left side select the file and in Tools : launch verilog compiler with current selection will get enable. Click it to compile the code
- Worklib is the directory where all the compiled codes are stored while Snapshot will have output of elaboration which in turn goes for simulation

- After compilation it will come under worklib you can see in right side window.
- Select the test bench and compile it. It will come under worklib. Under Worklib you can see the module and test-bench.
- The cds.lib file is an ASCII text file. It defines which libraries are accessible and where they are located. It contains statements that map logical library names to their physical directory paths. For this Design, you will define a library called "worklib".

Step 2: Elaboration: To check the port connections in hierarchical design

Inputs: Top level design/test bench Verilog codes

Outputs: Elaborate database updated in mapped library if successful, generates report else error reported in log file

Steps for elaboration – Run the elaboration command with elaborate options

1. It builds the module hierarchy.
2. Binds modules to module instances.
3. Computes parameter values.
4. Checks for hierarchical names conflicts.
5. It also establishes net connectivity and prepares all of this for simulation.

- After elaboration the file will come under snapshot. Select the test bench and elaborate it.

Step 3: Simulation: Simulate with the given test vectors over a period of time to observe the output behaviour.

Inputs: Compiled and Elaborated top level module name.

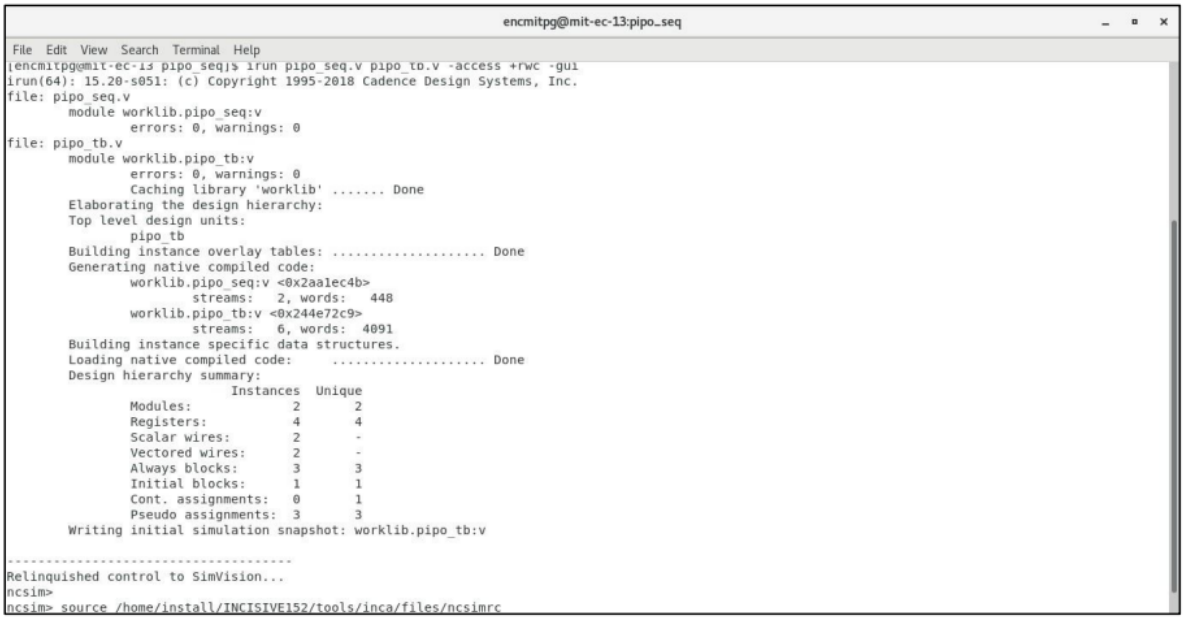
Outputs: Simulation log file, waveforms for debugging.

Simulation allow to dump design and test bench signals into a waveform.

Steps for simulation – Run the simulation command with simulator options.

RUN WITHOUT NCLAUNCH: Instead of nclaunch, design file and testbench can be run using single irun command.

```
[encmitpg@mit-ec-13 pipo_seq]$ irun pipo_seq.v pipo_tb.v -access +rwc -gui
```



Synthesize the design using Constraints and analyse reports, critical path and Max Operating Frequency

Step 1: Getting Started

- Make sure you close out all the Incisive tool windows first.
- Synthesis requires three files as follows,
  1. Liberty Files (.lib)
  2. Verilog/VHDL Files (.v or .vhdl or .vhd)
  3. SDC (Synopsis Design Constraint) File (.sdc)

Step 2 : Creating an SDC File

- In your terminal type “gedit counter_top.sdc” to create an SDC File if you do not have one.
- The SDC File must contain the following commands:

Constraint file:

i. create_clock -name clk -period 2 -waveform {0 1} [get_ports "clk"]
ii. set_clock_transition -rise 0.1 [get_clocks "clk"]
iii. set_clock_transition -fall 0.1 [get_clocks "clk"]
iv. set_clock_uncertainty 0.01 [get_ports "clk"]
v. set_input_delay -max 0.3 [get_ports "pi"] -clock [get_clocks "clk"]
vi. set_output_delay -max 0.3 [get_ports "po"] -clock [get_clocks "clk"]

i→ Creates a Clock named "clk" with Time Period 2ns and On Time from t=0 to t=1.
ii, iii → Sets Clock Rise and Fall time to 100ps.
iv → Sets Clock Uncertainty to 10ps.
v, vi → Sets the maximum limit for I/O port delay to 1ps

Step 3 : Performing Synthesis

- The Liberty files are present in the below path, /home/install/FOUNDRY/digital/<Technology_Node_number>nm/dig/lib/
- The Available technology nodes are 180nm ,90nm and 45nm.
- In the terminal, initialise the tools with the following commands if a new terminal is being used.
  - csh
  - source /home/install/cshrc
- The tool used for Synthesis is "Genus". Hence, type "genus -gui" to open the tool.
- The Following are commands to proceed :

1. read_libs /home/install/FOUNDRY/digital/90nm/dig/lib/slow.lib
2. read_hdl seq.v
3. elaborate
4. read_sdc pipo.sdc //reading top level sdc
5. set_db syn_generic_effort medium // effort level to medium for generic, mapping
6. set_db syn_map_effort medium
7. set_db syn_opt_effort medium
8. syn_generic
9. syn_map
10. syn_opt //Performing Synthesis Mapping and Optimisation
11. report_timing > seq_timing.rep
    //Generates Timing report for worst datapath and dumps into file
12. report_area > seq_area.rep
    //Generates Synthesis Area report and dumps into a file
13. report_power > seq_pwr.rep
    //Generates Power Report [Pre-Layout]
14. write_hdl>seq_net.v //Creates readable Netlist File
15. write_sdc > seq.sdc //Creates Block Level SDC
16. gui_show

Genus Script file with .tcl file Extension

Make sure that the following file need to be present in the directory where the simulation is performed.

```
[150205105@aust synthesis_lab]$ tree
.
|-- EDI_files
|   |-- lef
|   |   `-- gsclib045.lef
|   |-- libs
|   |   |-- fast.lib
|   |   |-- slow.lib
|   |   `-- typical.lib
|   `-- others
|       `-- capTable
|-- input_files
|   `-- alu_4bit.v
`-- synthesis_cmd.tcl

5 directories, 7 files
```

Make sure the following commands are present inside the tcl (name.tcl) file

	Commands	Description
1	set_db init_lib_search_path EDI_files/libs/	Sets the value of a specific attribute. Here we are setting directory name where all the timing libraries are located.
2	set_db library slow.lib	Sets which timing library will be used while mapping
3	set_db lef_library EDI_files/lef/gsclib045.lef	Sets lef file of a target technology
4	set_db hdl_search_path input_files	Sets the directory name where RTL is located
5	read_hdl alu_4bit.v	Loads the design with pre-synthesized RTL
6	elaborate	Creates a design from Verilog module. Undefined modules are labeled as unresolved and treated as blackbox
7	set_top_module alu_4bit	Sets top module name
8	current_design alu_4bit	Changes the current directory in the design hierarchy to the specified design

9	write_hdl > alu_4bit_elaborated.v	Creates a structural netlist using generic/mapped logic
10	create_clock -name clk -period 10 [get_ports clk]	Creates a clock named "clk" having 10ns period in a specific port "clk"
11	set_clock_uncertainty -setup 0.5 [get_clocks clk]	Sets uncertainty value for the clocks while calculating setup
12	set_clock_uncertainty -hold 0.5 [get_clocks clk]	Sets uncertainty value for the clocks while calculating hold
13	set_max_transition 2 [get_ports clk]	Sets maximum allowable transition time for changing logic state to 2ns for data path
14	set_clock_transition -min -fall 0.5 [get_clocks clk]	Sets minimum allowable clock transition time to 0.5ns for switching logic state from high to low for clock path
15	set_clock_transition -min -rise 0.5 [get_clocks clk]	Sets minimum allowable clock transition time to 0.5ns for switching logic state from low to high for clock path
16	set_clock_transition -max -fall 0.5 [get_clocks clk]	Sets maximum allowable clock transition time to 0.5ns for switching logic state from high to low for clock path
17	set_clock_transition -max -rise 0.5 [get_clocks clk]	Sets maximum allowable clock transition time to 0.5ns for switching logic state from low to high for clock path
18	set_clock_groups -name original -group [list [get_clocks clk]]	Defines groups of specific clocks
19	set DRIVING_CELL BUFX8	Defines driving cell name which will drive the input ports of the design
20	set DRIVE_PIN {Y}	Defines driver pin of the driving cell
21	set_driving_cell -lib_cell $DRIVING_CELL -pin $DRIVE_PIN [all_inputs]	Sets driving cell properties for all the input ports
22	set_max_fanout 10 [current_design]	Sets maximum allowable fanout number to 10
23	set_load 0.5 [all_outputs]	Sets load capacitance of the output ports of the design
24	set_operating_conditions slow	Sets operating condition for delay calculation
25	set_input_delay -max 0.5 [all_inputs]	Synthesis tool assumes the data is launched by a positive edge triggered flop from the external logic

		(and the maximum input delay for the setup analysis is 0.5ns)
26	set_output_delay -max 0.5 [all_outputs]	Synthesis tool assumes the data is captured by a positive edge triggered flop in the external logic (and the maximum output delay for the setup analysis is 0.5ns)
27	remove_assign -buffer_or_inverter BUFX16 -design [current_design]	Removes assign statement using BUFX16 cell
28	syn_generic	Performs generic synthesis
29	write_hdl > alu_4bit_generic.v	Creates a structural netlist using generic logic after generic synthesis
30	synthesize -to_mapped	Performs mapping using target timing library
31	write_hdl > alu_4bit_post_synthesis.v	Creates a structural netlist using mapped logic after mapping
32	remove_assigns_without_opt -buffer_or_inverter BUFX12 -verbose	Removes assign statement using BUFX12 cell
33	set_remove_assign_options -buffer_or_inverter BUFX12 -verbose	Sets buffer or inverter cell to remove assign statements
34	write -mapped > alu_4bit_mapped.v	Writes mapped netlist for post-synthesis flow
35	write_sdc > alu_4bit.sdc	Writes constraints file for post-synthesis flow

```
# Setting library and RTL paths
set_db / .init_lib_search_path {lib}
set_db / .init_hdl_search_path {rtl}

# Read Lib, RTL and SDC files
set_db / .library "slow.lib"
set DESIGN updowncounter
read_hdl "4bup_down_count.v"
elaborate $DESIGN
check_design -unresolved
read_sdc constraints_top.sdc

# Setting effort medium
set_db syn_generic_effort medium
set_db syn_map_effort medium
set_db syn_opt_effort medium
syn_generic
syn_map
syn_opt

write_hdl > upDowncounter_netlist.v
write_sdc > upDowncounter_sdc.sdc
# PPA Reports
report_power > upDowncounter_power.rpt
report_gates > upDowncounter_gatecount.rpt
report_timing > upDowncounter_timing.rpt
report_area > upDowncounter_area.rpt
report_qor -levels_of_logic -power -exclude_constant_nets > upDowncounter_qor.rpt
gui_show
```

Steps to create tcl

Fig. 53: Create tcl file

- Copy the commands from genus folder created in the pipo_seq folder

Fig. 54: TCL file

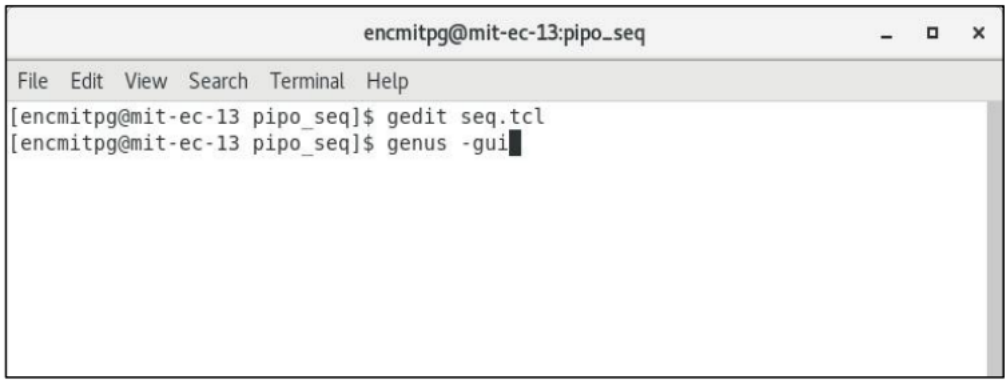


Fig. 55: Open genus

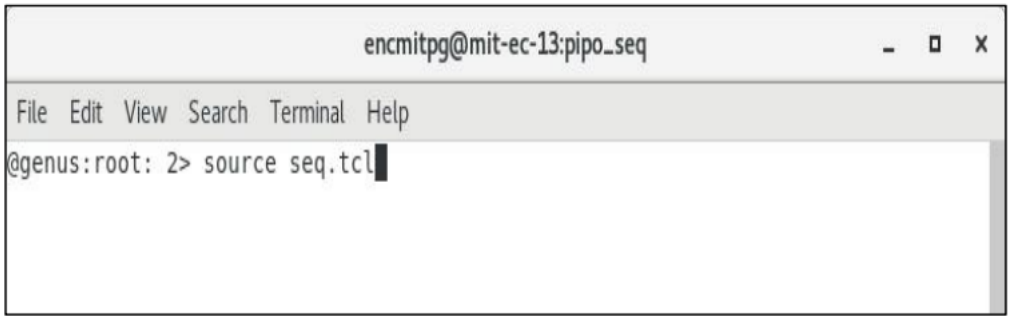


Fig. 56: Source tcl file

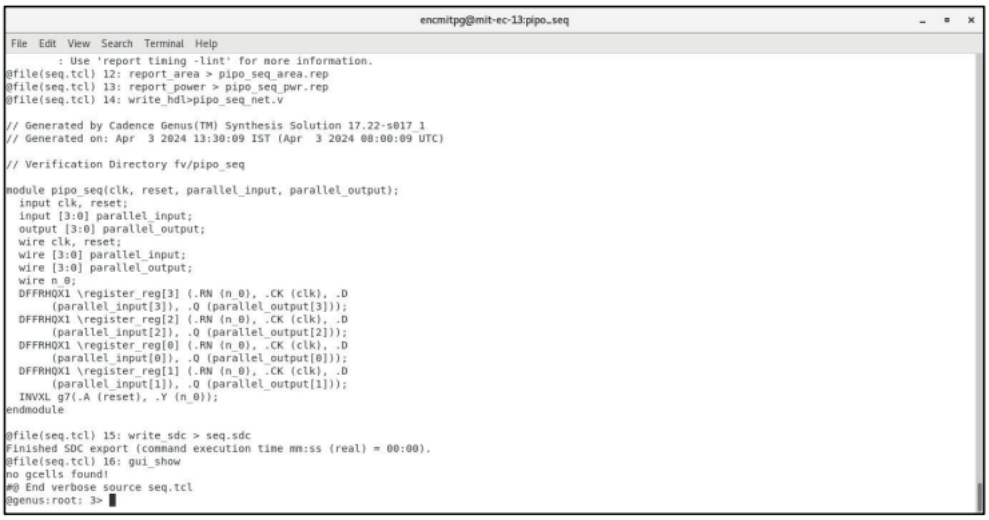


Fig. 57: End genus by giving exit command

- To show the synthesized output execute the command gui_show
- The GUI window of the genus synthesis output will be opened. If you click and zoom into each block of the circuit, you will be able to view the gate interconnections inside the block.
- **Post Lab Task**
  - Is the testbench module synthesizable?
  - Why is the operating condition of synthesis slow?
  - What is Standard Design Constraints (SDC)?
  - What do LEF and LIB files contain?
  - List the functions of buffer cells in synthesis.
  - Check the function of commands report_power, report_gates, report_timing in Genus

Exercise problem:

4. Do the following points for 4 bit universal shift register
   a. Understand the use and application of 4 bit universal shift register.
   b. Write sequential Verilog code for 4-bit universal shift register and verify the design by simulation.
   c. Do the synthesis and calculate the power area and performance without using the tcl command

d. Repeat the above problem with TCL command line.

5. Write the Verilog code for Master Slave JK flipflop and do same as exercise problem 1.
6. Write the Verilog code for synchronous mod 8 counter and do same as exercise problem 1.

Experiment- 9

Introduction to Physical Design

Objective:

- To understand the fundamental steps of physical design: floor planning and placement.
- To learn how to use a physical design tool (e.g., Cadence Innovus or Synopsys IC Compiler) for a simple circuit.
- To analyze area utilization, cell placement, and power planning.

Equipment and Software Required

- **Software Tools:**
  - Cadence Innovus / Synopsys IC Compiler / OpenROAD.
- **Design Files:**
  - Gate-level netlist (e.g., a simple combinational circuit).
  - Technology library (e.g., 45nm or 28nm).
  - Constraint file (SDC).

- Workstation or system with the above software installed.

Theory

Cadence: Cadence Design Systems offers a suite of tools specifically tailored for digital VLSI (Very Large-Scale Integration) design, aiding engineers in designing, verifying, and implementing complex digital circuits efficiently.

1. Design Entry: Cadence tools provide various methods for entering the design, including schematic entry and hardware description languages (HDL) such as Verilog, VHDL.

2. Simulation: Cadence offers powerful simulation tools for verifying digital designs at different abstraction levels. Engineers can perform functional simulations to ensure correct logic behavior, timing simulations to validate timing constraints, and mixed-signal simulations to verify interactions between digital and analog components.

3. Synthesis: Synthesis tools provided by Cadence enable the translation of RTL (Register Transfer Level) code written in HDLs into gate-level representations suitable for implementation. These tools optimize the design for area, power, and timing constraints while preserving the functional behavior specified in the RTL code.

4. Verification: Cadence tools include comprehensive verification solutions for ensuring the correctness of digital designs. This includes formal verification techniques, simulation-based verification methodologies, and advanced debugging capabilities to detect and fix design issues efficiently.

5. Physical Design: Cadence tools support the physical implementation of digital designs, including floor planning, placement, routing, and timing closure. These tools help engineers optimize the layout for area, power, and performance goals while meeting design constraints.

6. Design for Test (DFT): Cadence tools include features for incorporating testability into digital designs, such as scan chains, built-in self-test (BIST) structures, and test pattern generation.

7. Integration: Cadence tools are integrated into a unified design environment, allowing seamless data exchange between different stages of the design flow. This integration enhances productivity and streamlines the design process by providing a cohesive platform for design entry, simulation, synthesis, verification, and physical implementation.

Incisive Simulator: Incisive is a high-performance simulation tool by Cadence, offering comprehensive verification capabilities for digital designs. It supports advanced simulation techniques like RTL simulation, gate-level simulation, and assertion-based verification, ensuring thorough verification of complex designs.

Genus Synthesis: Genus is a next-generation RTL synthesis tool from Cadence, designed to address the challenges of advanced process nodes and complex design requirements. It offers innovative synthesis algorithms for optimizing power, performance, and area (PPA) metrics while maintaining design hierarchy and meeting timing constraints.

Innovus Implementation System: Innovus is a leading physical implementation tool by Cadence, offering advanced capabilities for place-and-route, timing closure, and power optimization. It leverages advanced algorithms and technologies to deliver optimal results for designs targeting advanced process nodes, enabling faster time-to-market and improved silicon

quality.

Example: 4-bit PIPO (parallel – in parallel -out) shift register circuit

A 4-bit PIPO (Parallel In Parallel Out) shift register is a digital circuit commonly used in digital systems for various purposes such as data storage, parallel-to-serial or serial-to-parallel conversion, and delay elements.

A 4-bit PIPO shift register consists of four flip-flops (memory cells) capable of storing binary data. Each flip-flop stores one bit of data, allowing the register to hold a 4-bit binary word. The register has four parallel input lines, one for each bit. When new data is presented at the input, it is loaded simultaneously into all four flip-flops, updating the contents of the register. The register has four parallel output lines, one for each bit. These output lines allow the current contents of the register to be read out in parallel, providing access to the stored data.

The PIPO shift register typically doesn't perform shifting operations. Instead, it holds the data statically until new data is loaded into it. The "Parallel In Parallel Out" nature of the register means that data is both input and output simultaneously, without any shifting.

Applications: PIPO shift registers find applications in various digital systems where parallel data storage and retrieval are necessary. They can be used in microcontrollers, microprocessors, digital signal processing, and communication systems for tasks such as data buffering, temporary storage, and interfacing between parallel and serial data streams.

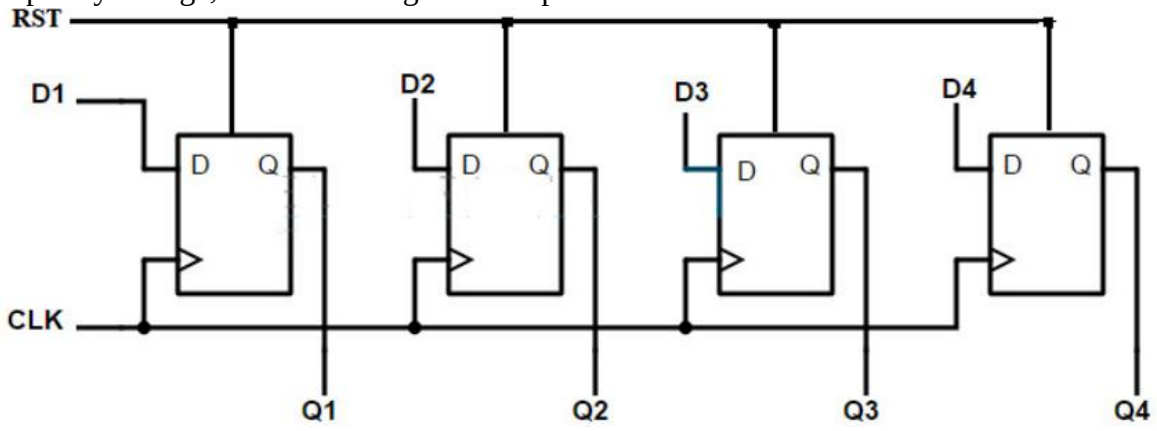


Fig. 2: Block diagram using D flipflop

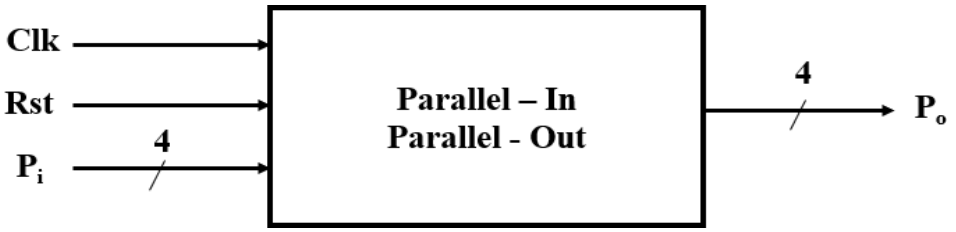
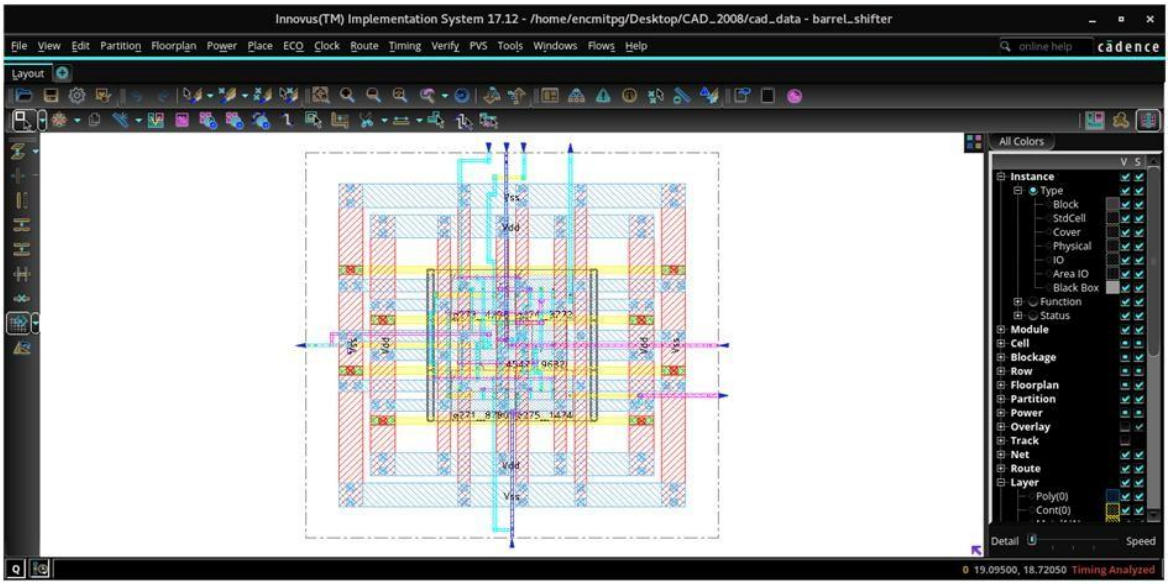


Fig. 1: Block diagram of 4-bit PIPO

Final Layout :-


Innovus(TM) Implementation System 17.12 - /home/encmitpg/Desktop/CAD_2008/cad_data - barrel_shifter
File View Edit Partition Floorplan Power Place ECO Clock Route Timing Verify PVS Tools Windows Flows Help
online help
cadence
Layout
All Colors
Instance
Type
Block
StdCell
Cover
Physical
IO
Area IO
Black Box
Function
Status
Module
Cell
Blockage
Row
Floorplan
Partition
Power
Overlay
Track
Net
Route
Layer
Poly(0)
Cont(0)
Detail
Speed
Vss
Vdd
0 19.09500, 18.72050 Timing Analyzed

Creating a Workspace:

- In Desktop, Create a folder and name it Digital_VlSI (give folder name without any space).
- Create a new sub-directory and name it pipo_seq for the design and open a terminal from the sub-directory.

Functional Simulation:

Invoke the cadence environment by typing the below commands

- csh (Invokes C-Shell)
- source /home/install/cshrc (mention the path of the tools)

(The path of cshrc could vary depending on the installation destination as /home/install/or /home etc.)

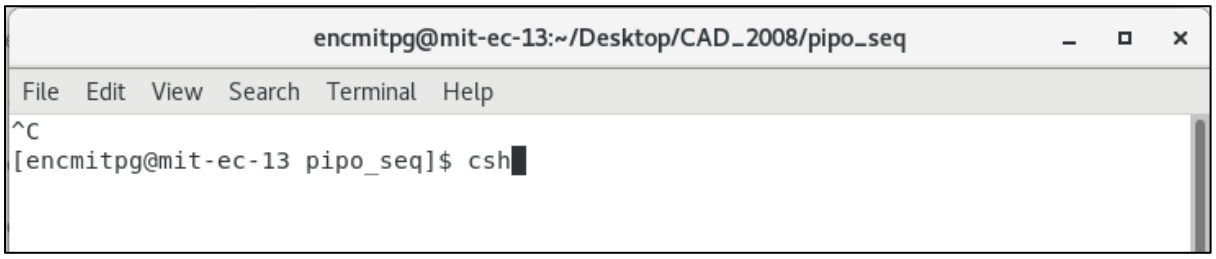


Fig. 3: csh

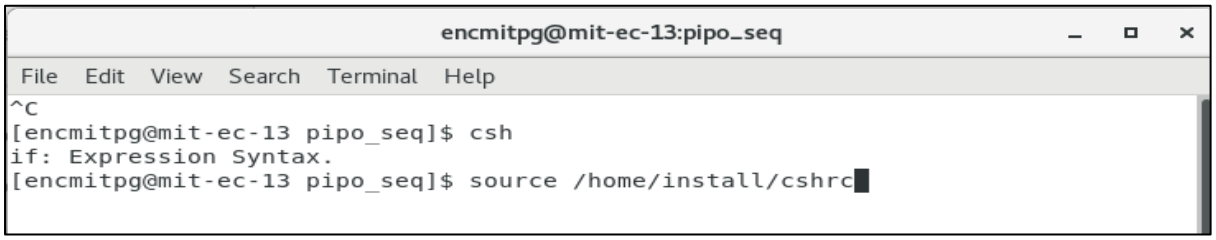


Fig. 4: source/cshrc

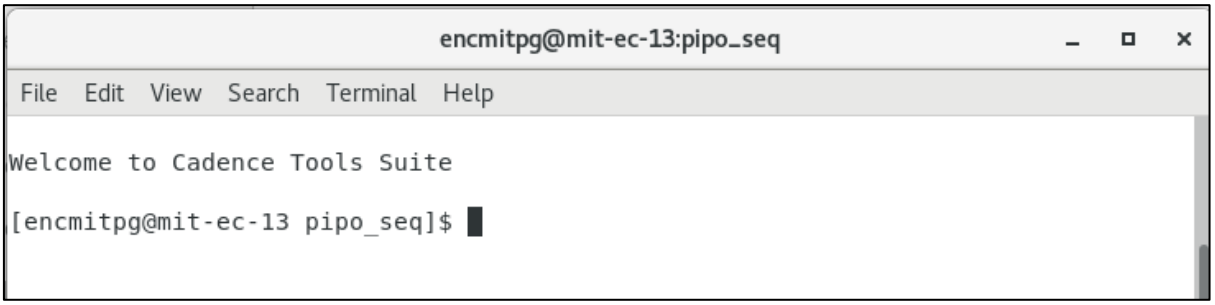


Fig. 5: Cadence tool suite

Creating Source Codes

- In the Terminal, type gedit <filename>.v
- A blank document opens up into which the following source code can be typed down.

Fig. 6: Verilog file creation

Source Code:

```
1 module pipo_seq(clk,rst, pi, po);
2 input clk,rst;
3 input [3:0] pi;
4 output [3:0] po;
5 wire [3:0] pi;
6 reg [3:0] po;
7
8 always @(posedge clk)
9
10 begin
11 if (rst)
12 po<= 4'b0000;
13 else
14 po <= pi;
15 end
16
17 endmodule
```

Fig. 7: source code

- Use Save option or Ctrl+S to save the code or click on the save option from the top most right corner and close the text file.

Creating Test Bench:

- Similarly, create your test bench using gedit <filename_tb>.v to open a new blank document.

Fig. 8: Verilog testbench file creation

Testbench code:

```
 1 module pipo_tb;
 2 reg clk;
 3 reg rst;
 4 reg [3:0] pi;
 5 wire [3:0] po;
 6
 7 pipo_seq uut (.clk(clk),.rst(rst),.pi(pi),.po(po) );
 8
 9 initial begin
10 clk = 0;
11 rst = 0;
12 pi = 4'b0001;
13 #5 rst=1'b1;
14 #5 rst=1'b0;
15 #10 pi=4'b1001;
16 #10 pi=4'b1010;
17 #10 pi=4'b1011;
18 #10 pi=4'b1110;
19 #10 pi=4'b1111;
20 #10 pi=4'b0000;
21 end
22
23 always #5 clk = ~clk;
24
25 initial #100 $finish;
26 initial
27 $monitor ("Time: %0t, pi: %b, po: %b", $time, pi, po);
28
29 endmodule
```

Fig. 9: Testbench code

- Click on the Save option and it will look like the below window and then close the file.

To Launch Simulation tool

- linux:/> nclaunch -new&

// "-new" option is used for invoking NCVERILOG for the first time for any design

- linux:/> nclaunch&

// On subsequent calls to NCVERILOG

- It will invoke the nclaunch window for functional simulation we can compile,elaborate and simulate it using Multiple.

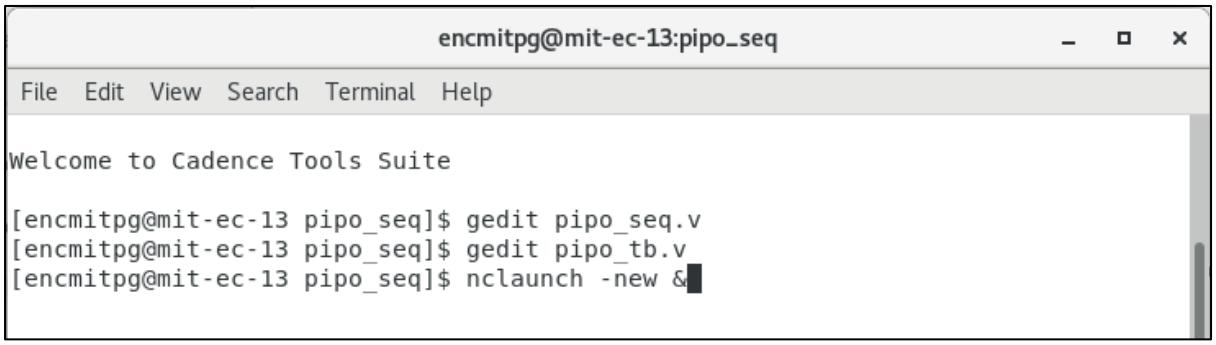


Fig. 10: nclaunch cmmd

- Select Multiple Step and then select "Create cds.lib File" as shown in below figure
- Click the cds.lib file and save the file by clicking on Save option

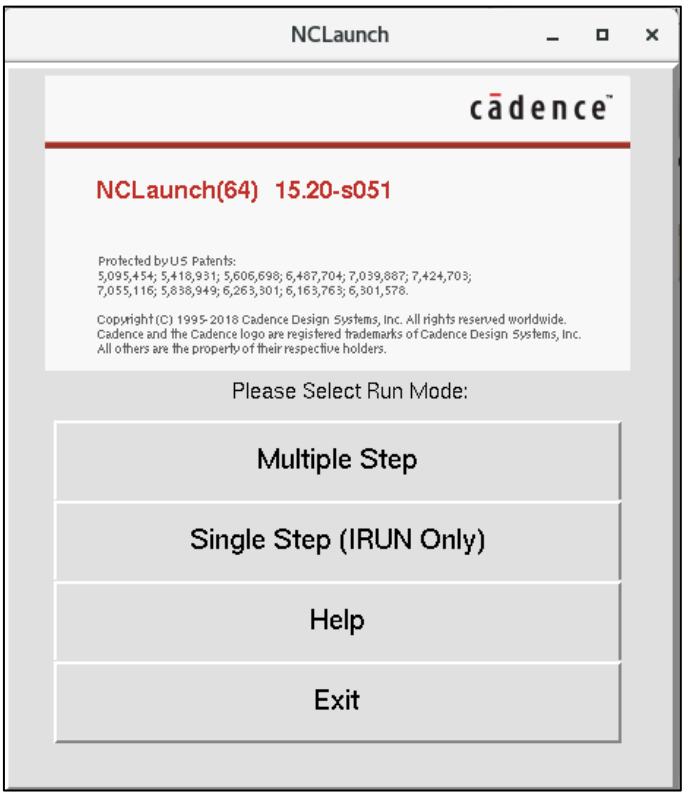


Fig. 11: nclaunch step selection

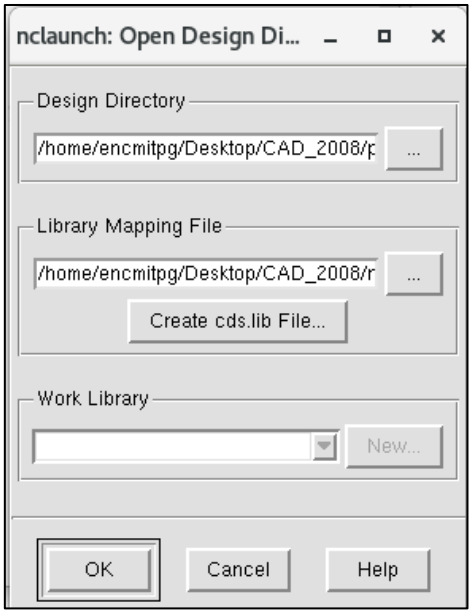


Fig. 12: nclaunch after selecting multiple steps

- Save cds.lib file and select the correct option for cds.lib file format based on the HDL Language and Libraries used.
- Select “Don’t include any libraries (verilog design)” from “New cds.lib file” and click on “OK” as in below figure
- We are simulating verilog design without using any libraries

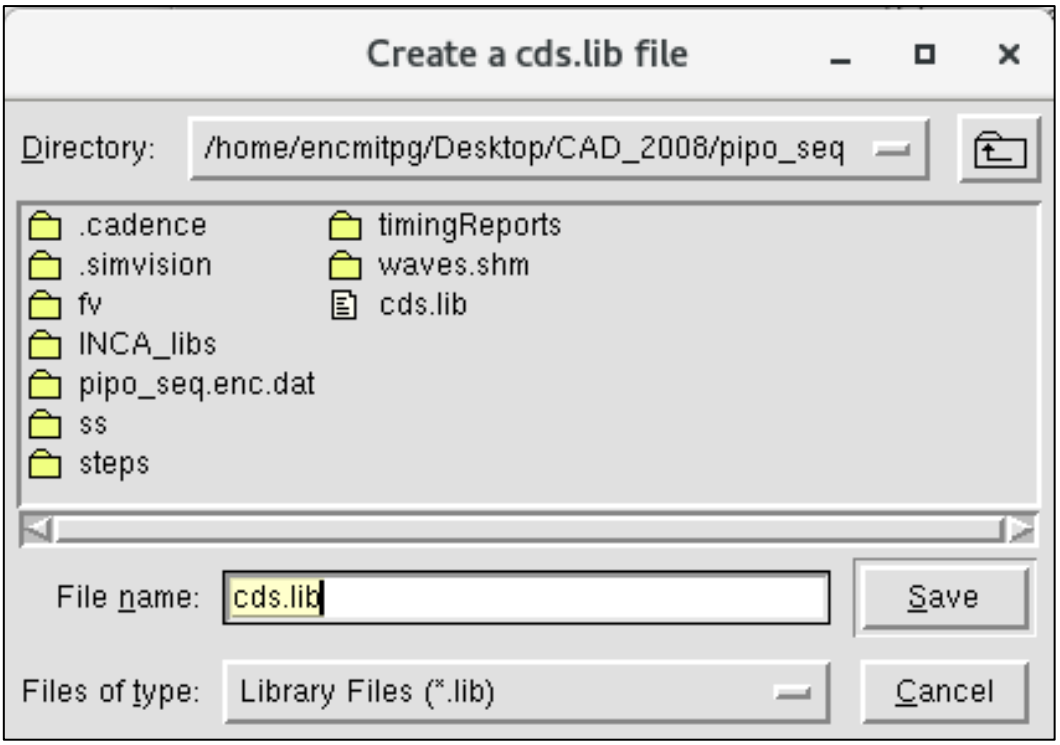


Fig. 13: Create sdc file and save

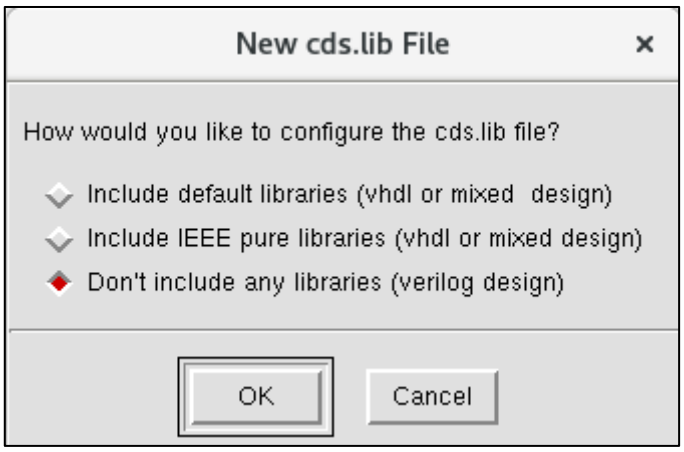


Fig. 14: Don't include libraries

- A Click "OK" in the "nclaunch: Open Design Directory" window as shown in below figure.

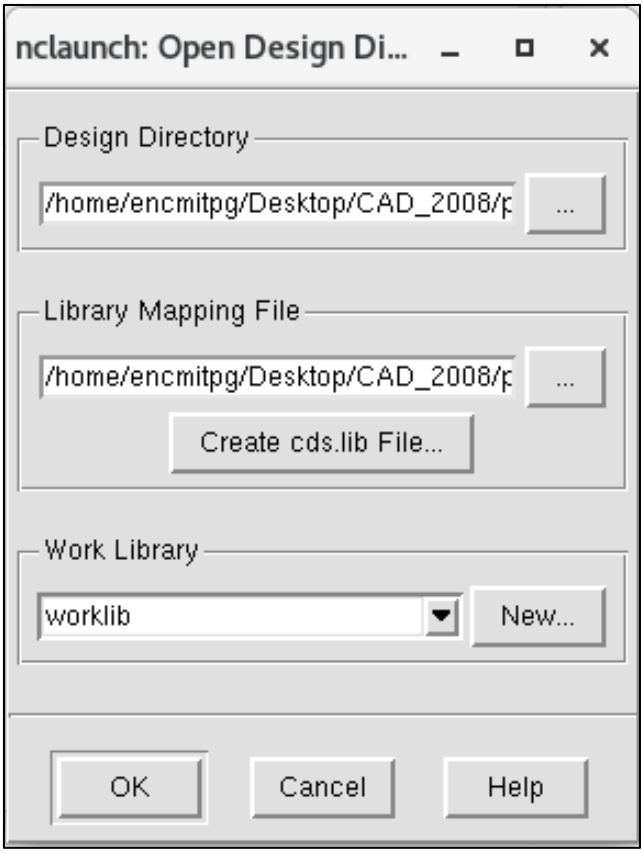


Fig. 15: After saving cds file

- A 'NCLaunch window' appears as shown in figure below
- Left side you can see the HDL files. Right side of the window has worklib and snapshots directories listed.
- Worklib is the directory where all the compiled codes are stored while Snapshot will have output of elaboration which in turn goes for simulation

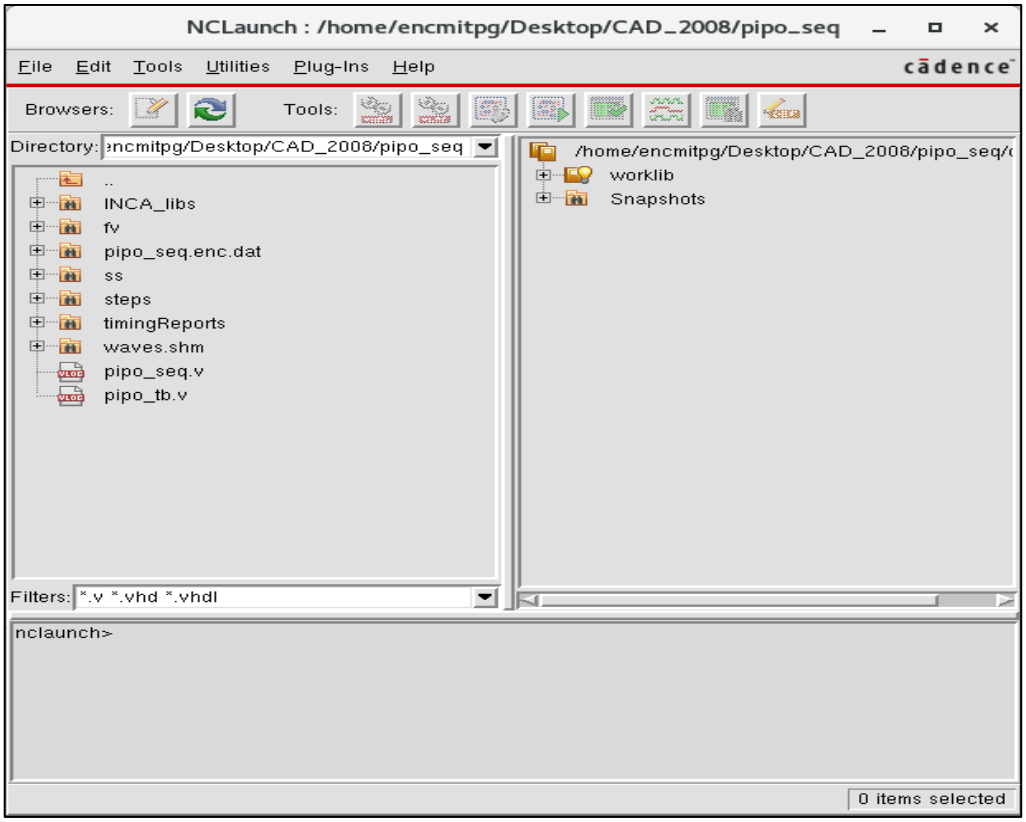


Fig. 16: nclaunch window

To perform the function simulation, the following three steps are involved Compilation, Elaboration and Simulation.

Step 1: Compilation: Process to check the correct Verilog language syntax and usage

Inputs: Supplied are Verilog design and test bench codes

Outputs: Compiled database created in mapped library if successful, generates report else error reported in log file

Steps for compilation:

1. Create work/library directory (most of the latest simulation tools creates automatically).
2. Map the work to library created (most of the latest simulation tools creates automatically).
3. Run the compile command with compile options.

- Left side select the file and in Tools : launch verilog compiler with current selection will get enable. Click it to compile the code
- Worklib is the directory where all the compiled codes are stored while Snapshot will have output of elaboration which in turn goes for simulation

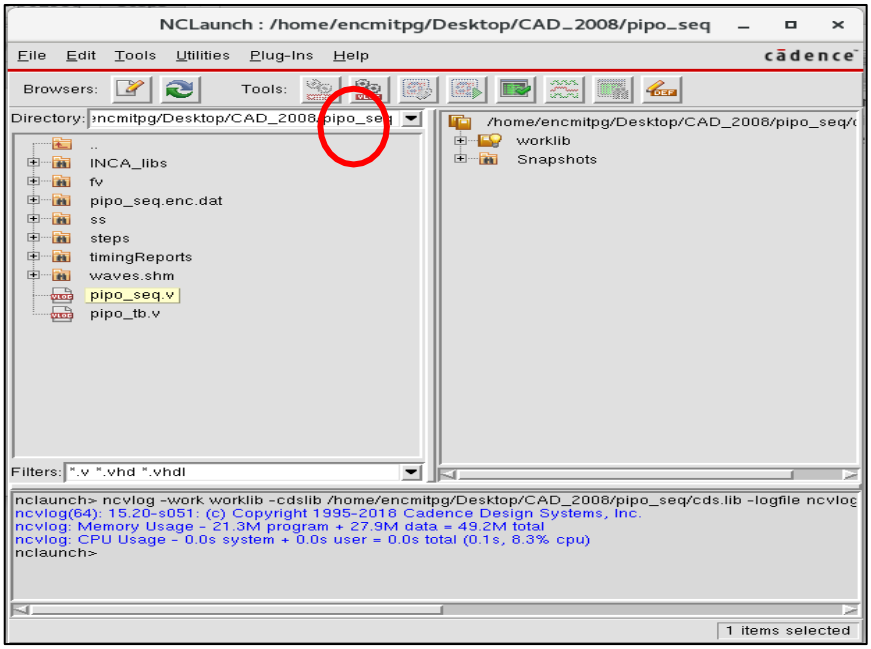


Fig. 17: select Verilog code and launch Verilog compiler

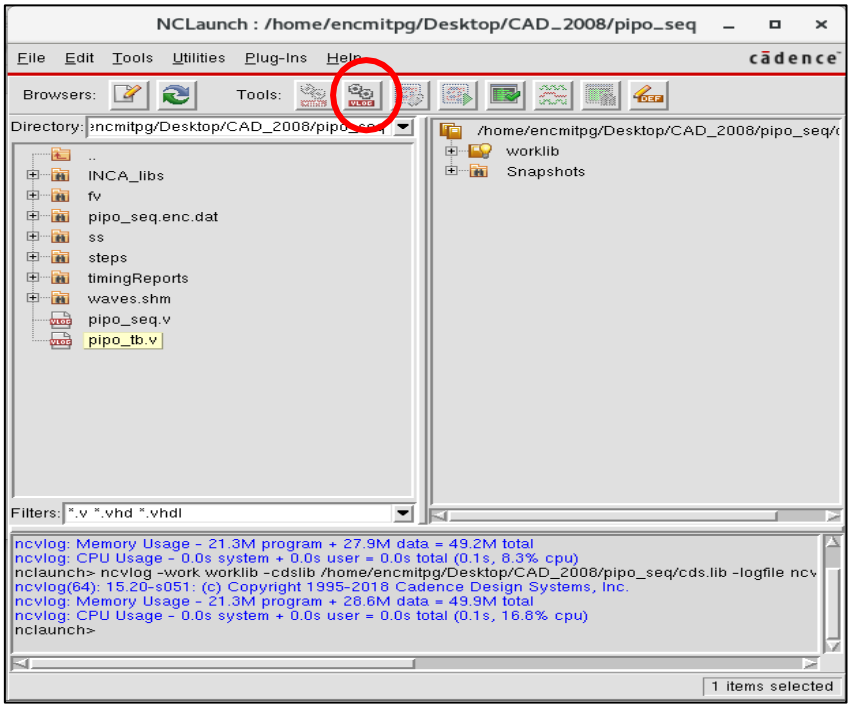


Fig. 18: select Verilog code and launch Verilog compiler

- After compilation it will come under worklib you can see in right side window.

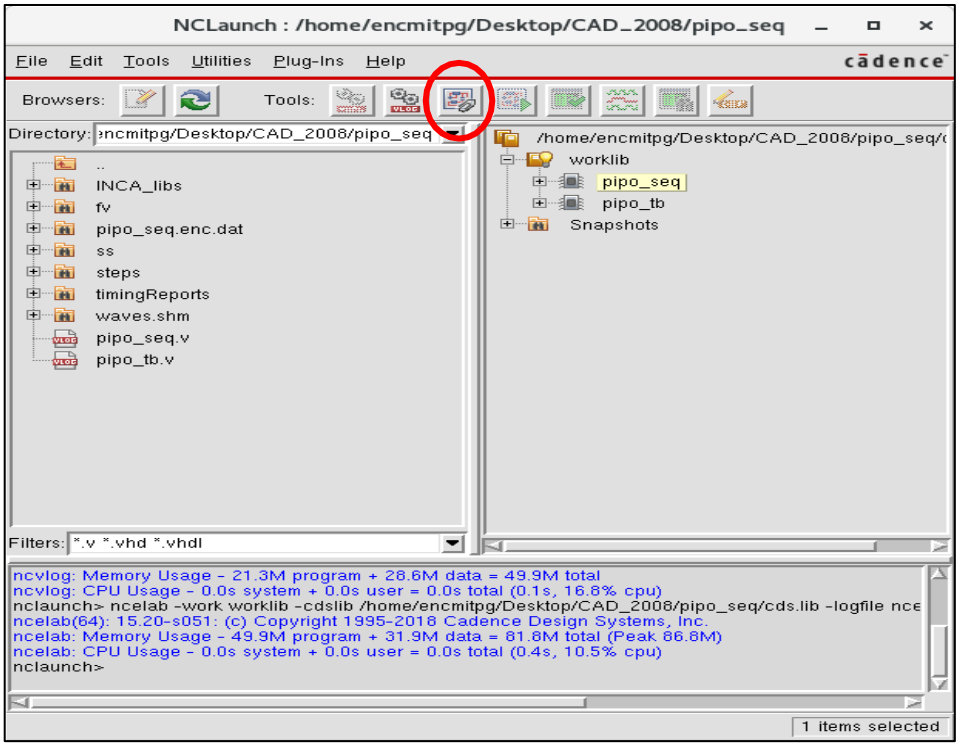


Fig. 19: Compiled database in worklib

- Select the test bench and compile it. It will come under worklib. Under Worklib you can see the module and test-bench.

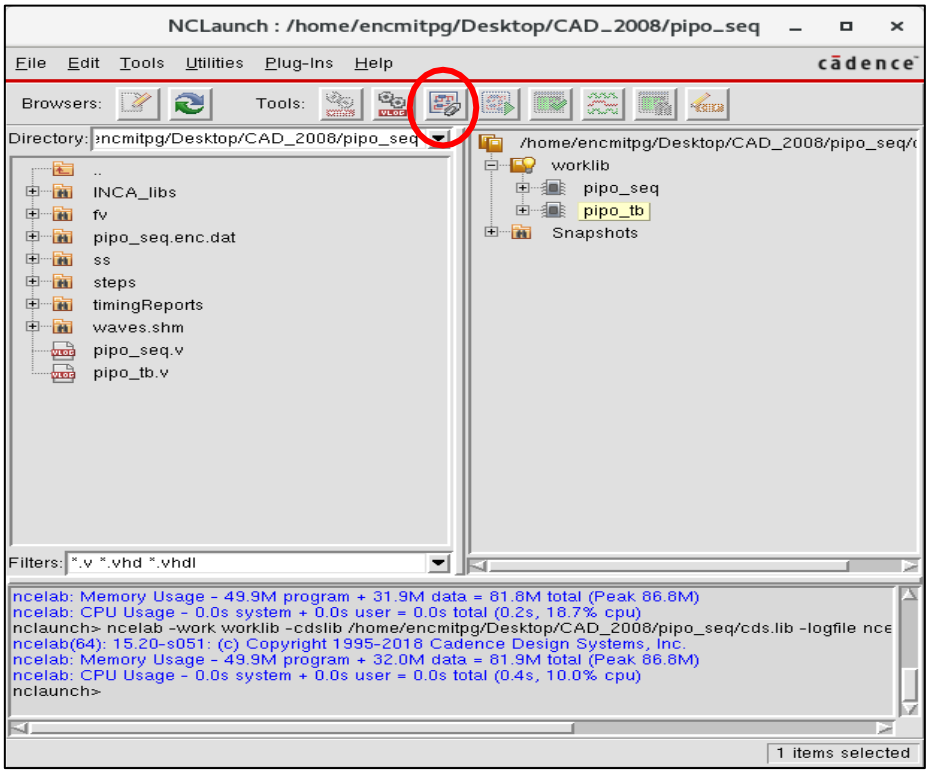


Fig. 20: Compiled Test-bench Database in Worklib

- The cds.lib file is an ASCII text file. It defines which libraries are accessible and where they are located. It contains statements that map logical library names to their physical directory paths. For this Design, you will define a library called “worklib”

Step 2: Elaboration: To check the port connections in hierarchical design

Inputs: Top level design/test bench Verilog codes

Outputs: Elaborate database updated in mapped library if successful, generates report else error reported in log file

Steps for elaboration – Run the elaboration command with elaborate options

1. It builds the module hierarchy.
2. Binds modules to module instances.
3. Computes parameter values.
4. Checks for hierarchical names conflicts.
5. It also establishes net connectivity and prepares all of this for simulation.

- After elaboration the file will come under snapshot. Select the test bench and elaborate it.

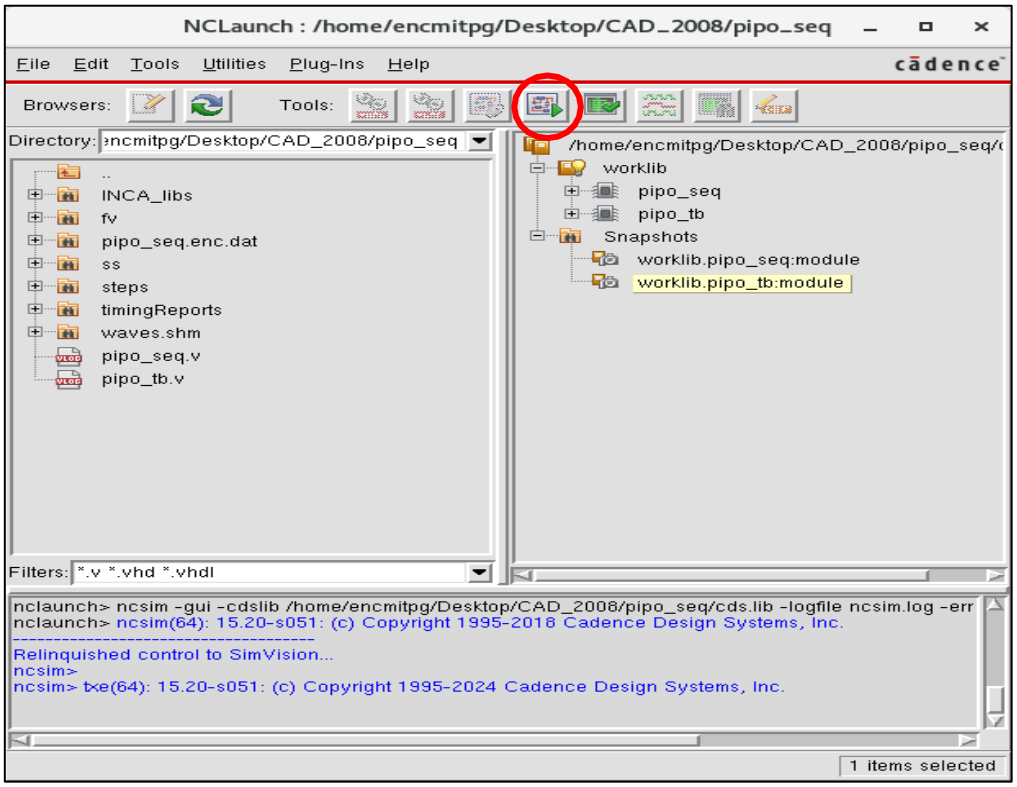


Fig. 21: Elaboration Launch Operation

Step 3: Simulation: Simulate with the given test vectors over a period of time to observe the output behavior.

Inputs: Compiled and Elaborated top level module name.

Outputs: Simulation log file, waveforms for debugging.

Simulation allows to dump design and test bench signals into a waveform.

Steps for simulation – Run the simulation command with simulator options.

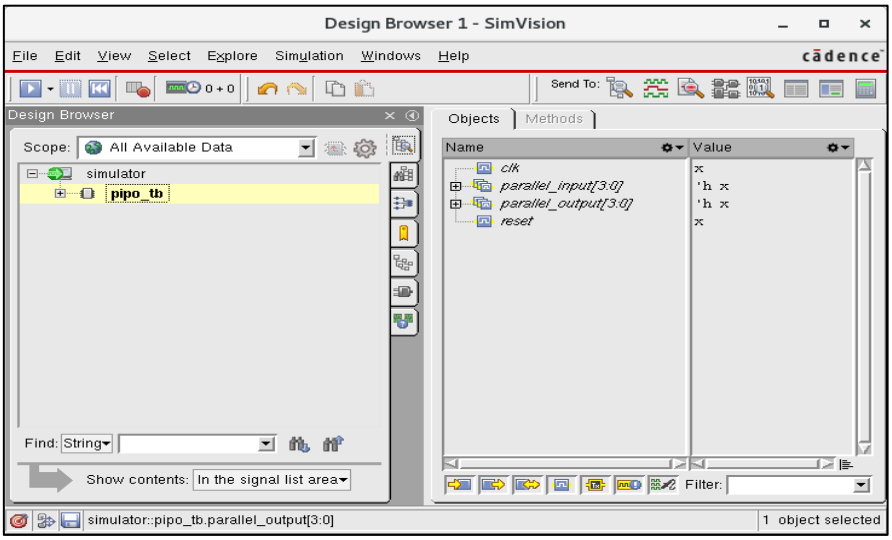


Fig. 22: After launching simulator SimVision window appears, Right click on the tb module and select 'send to simulation window'

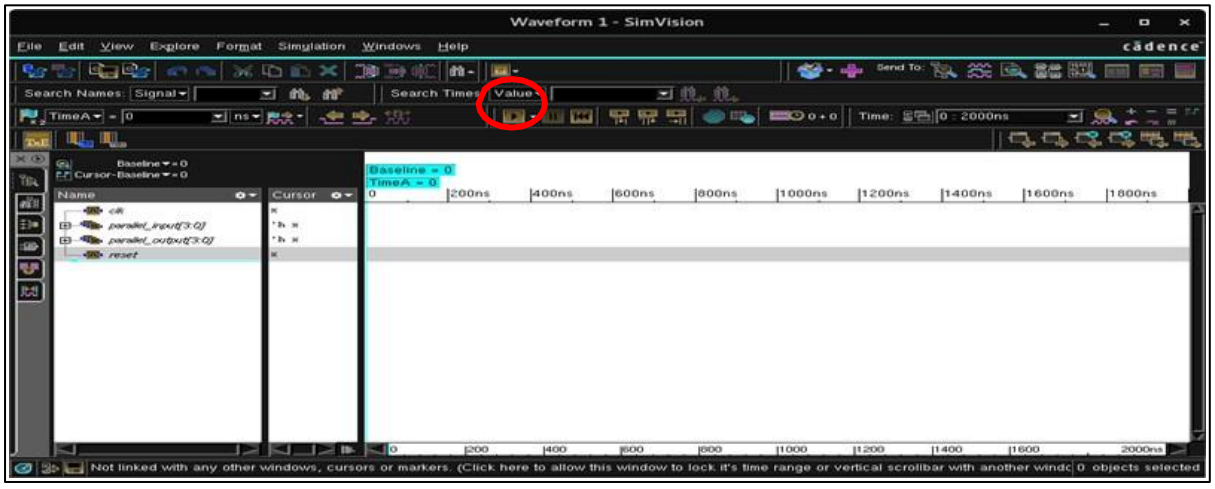


Fig. 23: Click on Run simulation to get waveforms

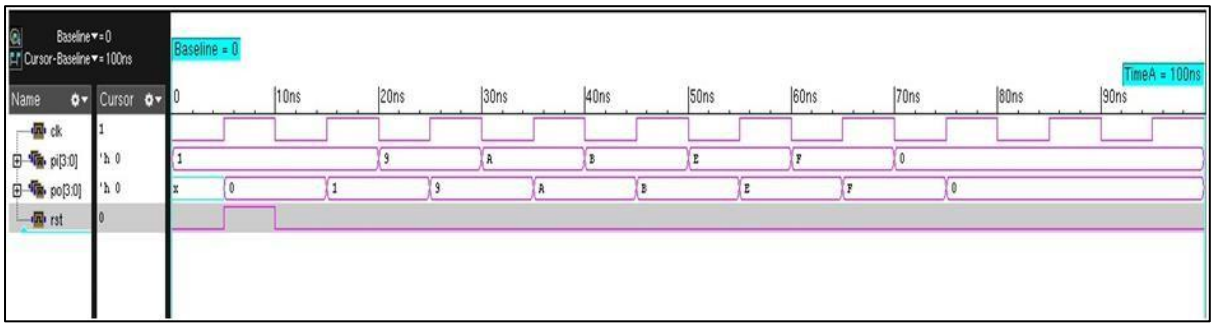


Fig. 24: Run the simulation

```
Time: 0, pi: 0001, po: xxxx
Time: 5, pi: 0001, po: 0000
Time: 15, pi: 0001, po: 0001
Time: 20, pi: 1001, po: 0001
Time: 25, pi: 1001, po: 1001
Time: 30, pi: 1010, po: 1001
Time: 35, pi: 1010, po: 1010
Time: 40, pi: 1011, po: 1010
Time: 45, pi: 1011, po: 1011
Time: 50, pi: 1110, po: 1011
Time: 55, pi: 1110, po: 1110
Time: 60, pi: 1111, po: 1110
Time: 65, pi: 1111, po: 1111
Time: 70, pi: 0000, po: 1111
Time: 75, pi: 0000, po: 0000
```

Fig. 25: Open console window to analyse the displayed values

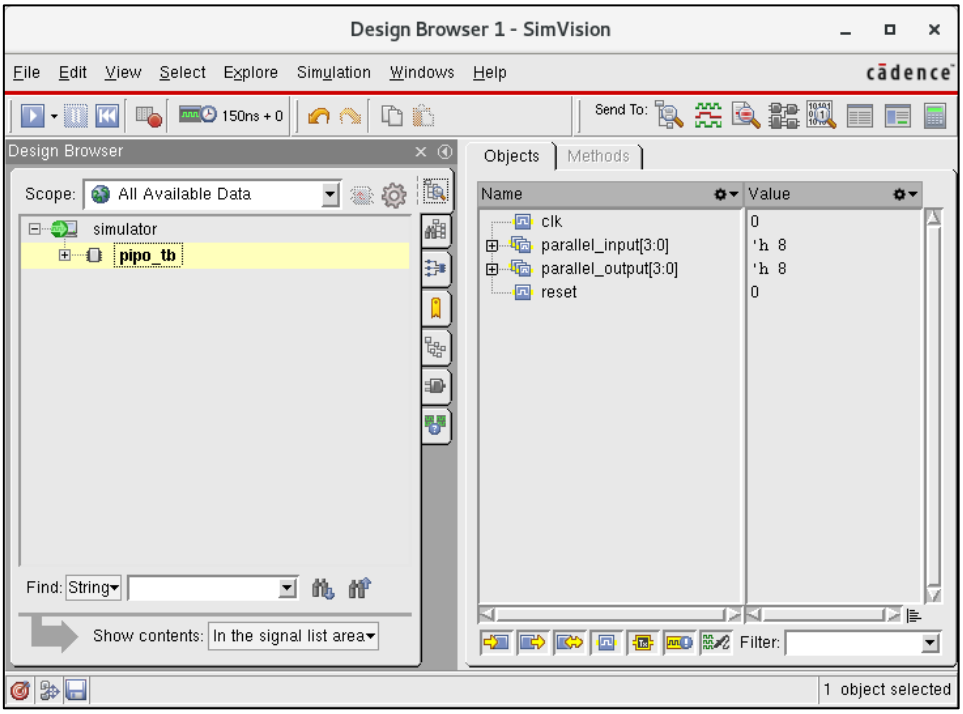


Fig. 26: go to SimVision window File then Exit from the window

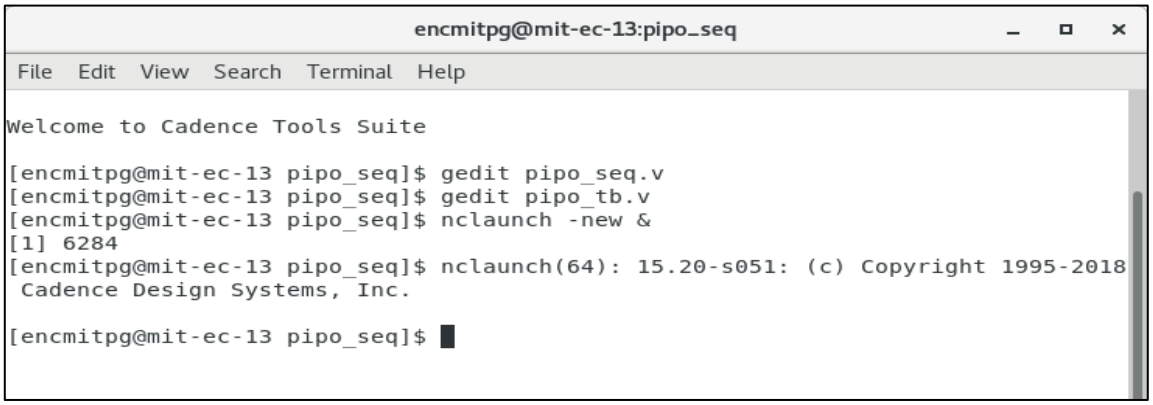


Fig. 27: Press ctrl+C and exit nclaunch

- Instead of nclaunch, design file and testbench can be run using single irun command.

```
[encmitpg@mit-ec-13 pipo_seq]$ irun pipo_seq.v pipo_tb.v -access +rwc -gui
```

Fig. 28: irun single command to open SimVision

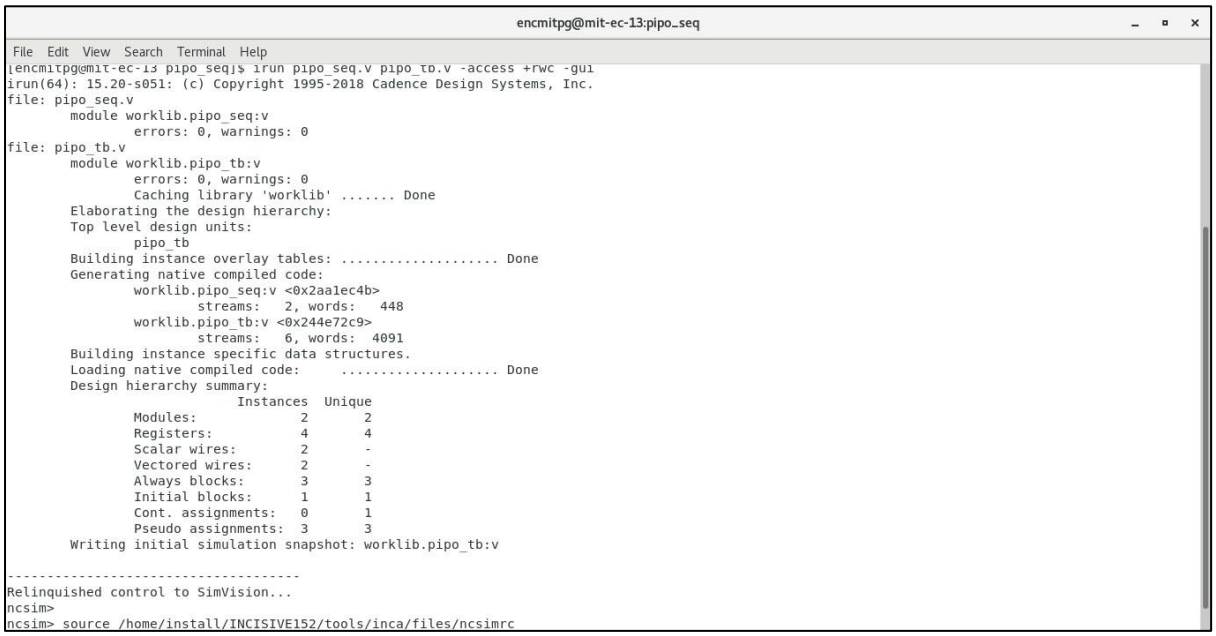


Fig. 29: Terminal window after successful compilation and elaboration

Synthesize the design using Constraints and analyse reports, critical path and Max Operating Frequency

Step 1: Getting Started

- Make sure you close out all the Incisive tool windows first.
- Synthesis requires three files as follows,
  1. Liberty Files (.lib)
  2. Verilog/VHDL Files (.v or .vhdl or .vhd)
  3. SDC (Synopsis Design Constraint) File (.sdc)

Step 2 : Creating an SDC File

- In your terminal type “gedit counter_top.sdc” to create an SDC File if you do not have one.
- The SDC File must contain the following commands:

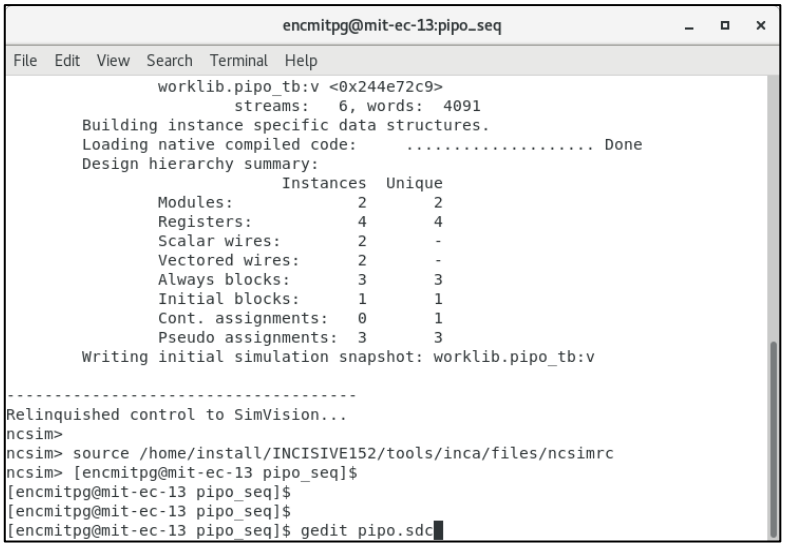


Fig. 30: Create a constraint file

Constraint file:

i. create_clock -name clk -period 2 -waveform {0 1} [get_ports "clk"]
ii. set_clock_transition -rise 0.1 [get_clocks "clk"]
iii. set_clock_transition -fall 0.1 [get_clocks "clk"]
iv. set_clock_uncertainty 0.01 [get_ports "clk"]
v. set_input_delay -max 0.3 [get_ports "pi"] -clock [get_clocks "clk"]
vi. set_output_delay -max 0.3 [get_ports "po"] -clock [get_clocks "clk"]

i→ Creates a Clock named “clk” with Time Period 2ns and On Time from t=0 to t=1.
ii, iii → Sets Clock Rise and Fall time to 100ps.
iv → Sets Clock Uncertainty to 10ps.
v, vi → Sets the maximum limit for I/O port delay to 1ps

Step 3 : Performing Synthesis

- The Liberty files are present in the below path, /home/install/FOUNDRY/digital/<Technology_Node_number>nm/dig/lib/
- The Available technology nodes are 180nm ,90nm and 45nm.
- In the terminal, initialise the tools with the following commands if a new terminal is being used.
  - csh
  - source /home/install/cshrc
- The tool used for Synthesis is “Genus”. Hence, type “genus -gui” to open the tool.
- The Following are commands to proceed :

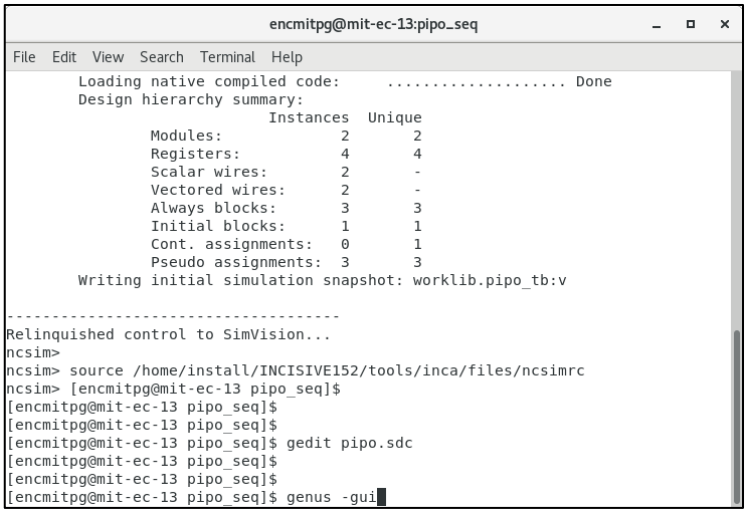


Fig. 31: Launch genus

1. read_libs /home/install/FOUNDRY/digital/90nm/dig/lib/slow.lib
2. read_hdl seq.v
3. elaborate
4. read_sdc pipo.sdc //reading top level sdc
5. set_db syn_generic_effort medium // effort level to medium for generic, mapping
6. set_db syn_map_effort medium
7. set_db syn_opt_effort medium
8. syn_generic
9. syn_map
10. syn_opt //Performing Synthesis Mapping and Optimisation
11. report_timing > seq_timing.rep
    //Generates Timing report for worst datapath and dumps into file
12. report_area > seq_area.rep
    //Generates Synthesis Area report and dumps into a file
13. report_power > seq_pwr.rep
    //Generates Power Report [Pre-Layout]
14. write_hdl>seq_net.v //Creates readable Netlist File
15. write_sdc > seq.sdc //Creates Block Level SDC
16. gui_show

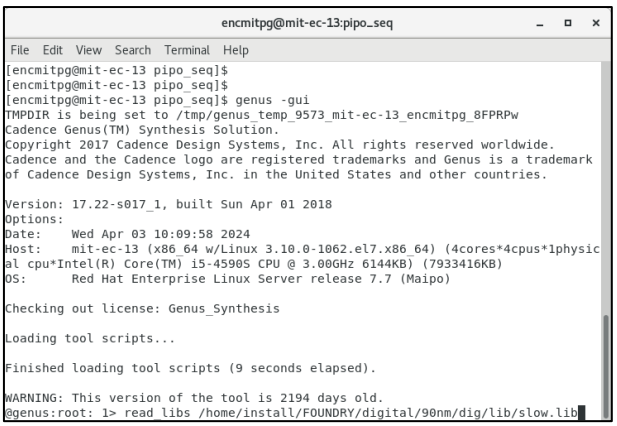


Fig. 32: Read library to specify library path

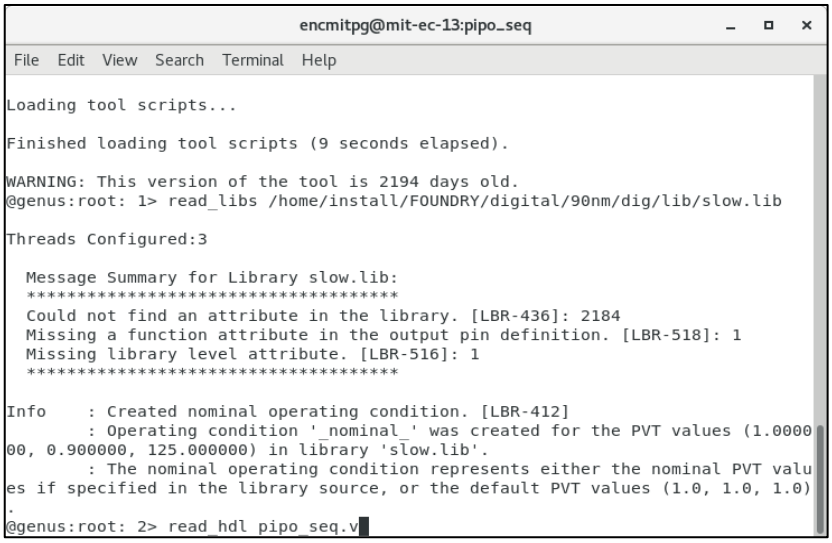


Fig. 33 read the hdl design file

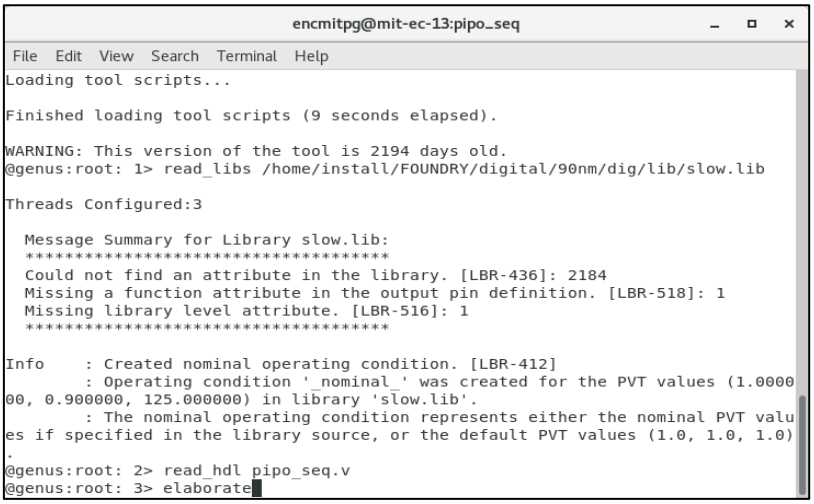


Fig. 34: elaborate

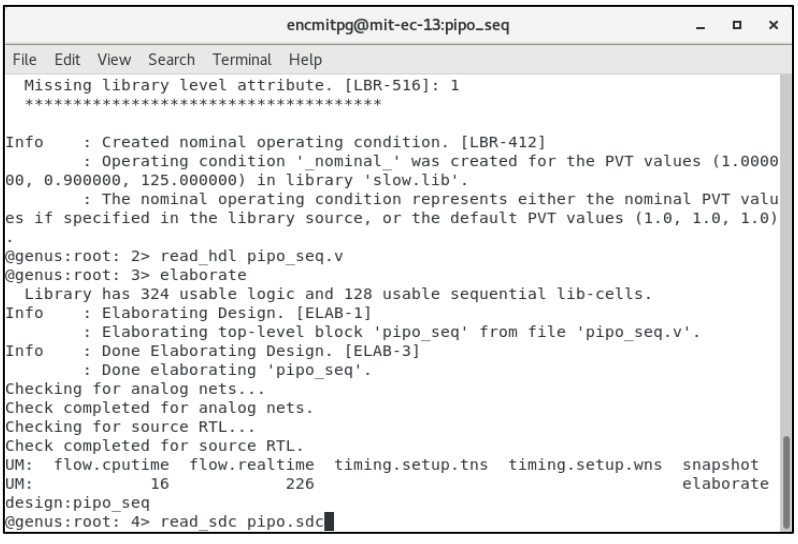


Fig. 35: read constraint file

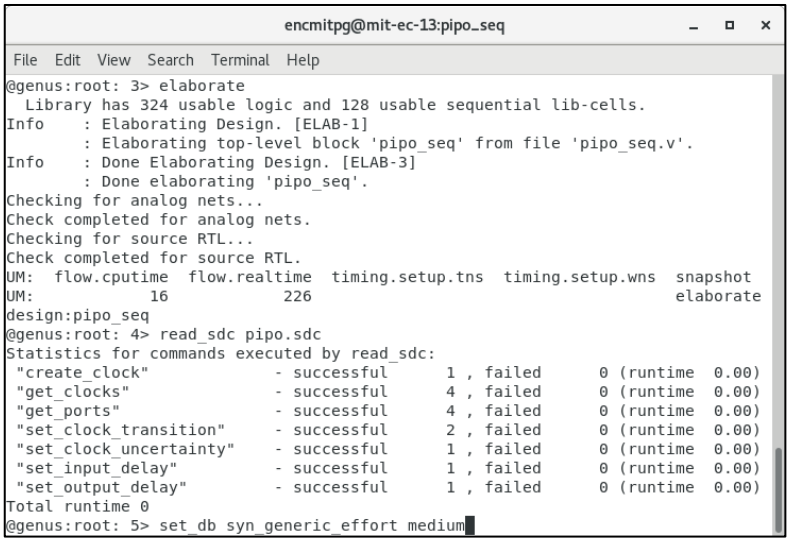


Fig. 36: set effort level for translation (generic)

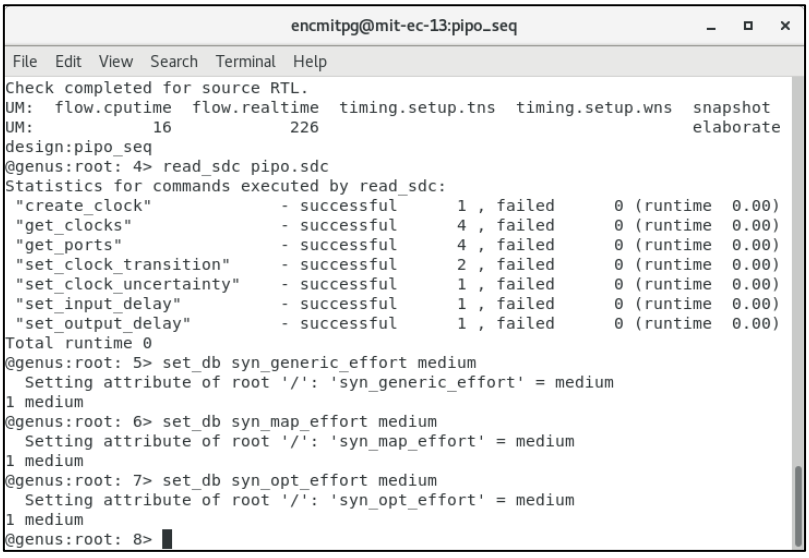


Fig. 37: set effort level for mapping and optimization

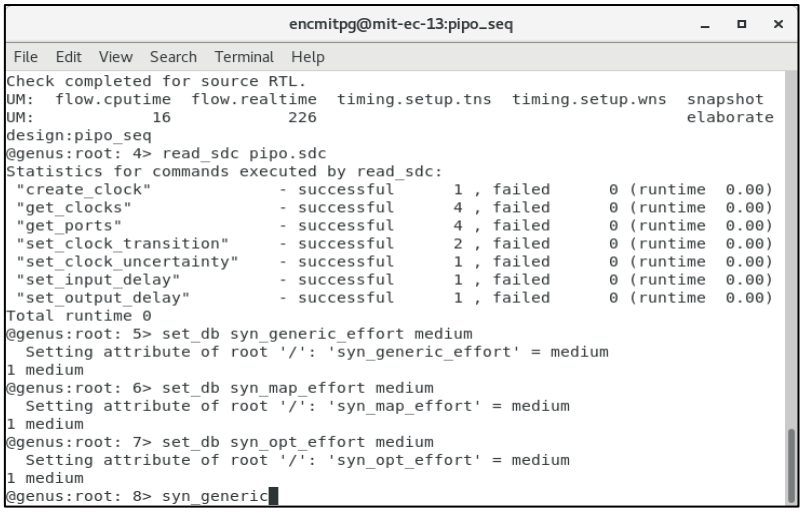


Fig. 38: perform translation

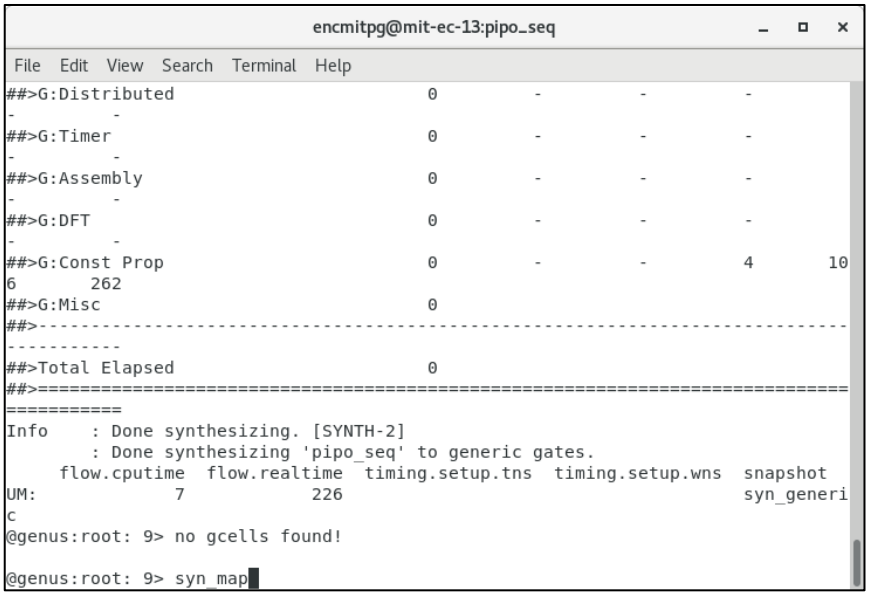


Fig. 39: Perform mapping

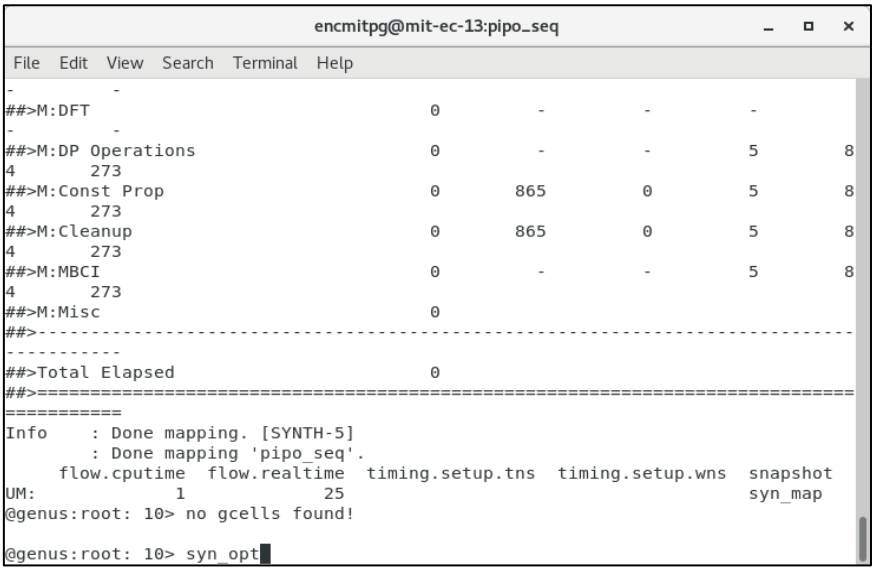


Fig. 40: perform optimization

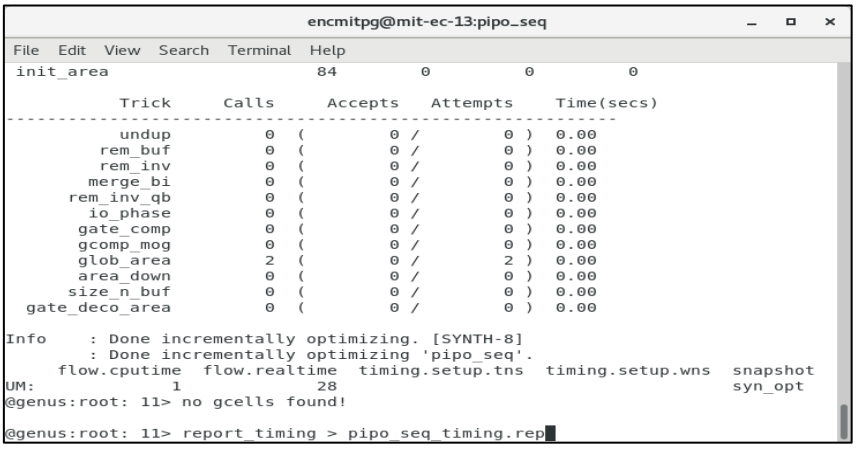


Fig. 41: generate timing report

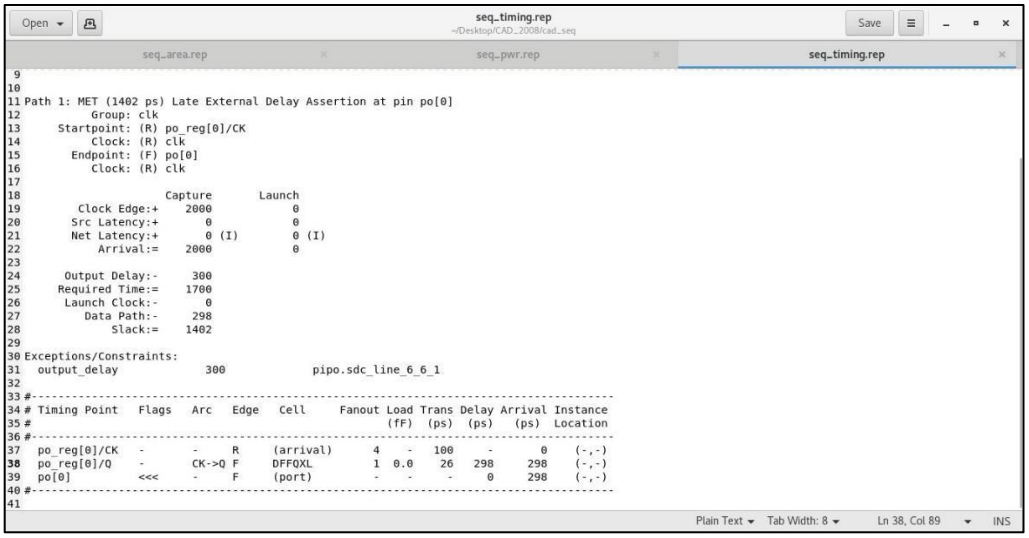


Fig. 42: Timing report

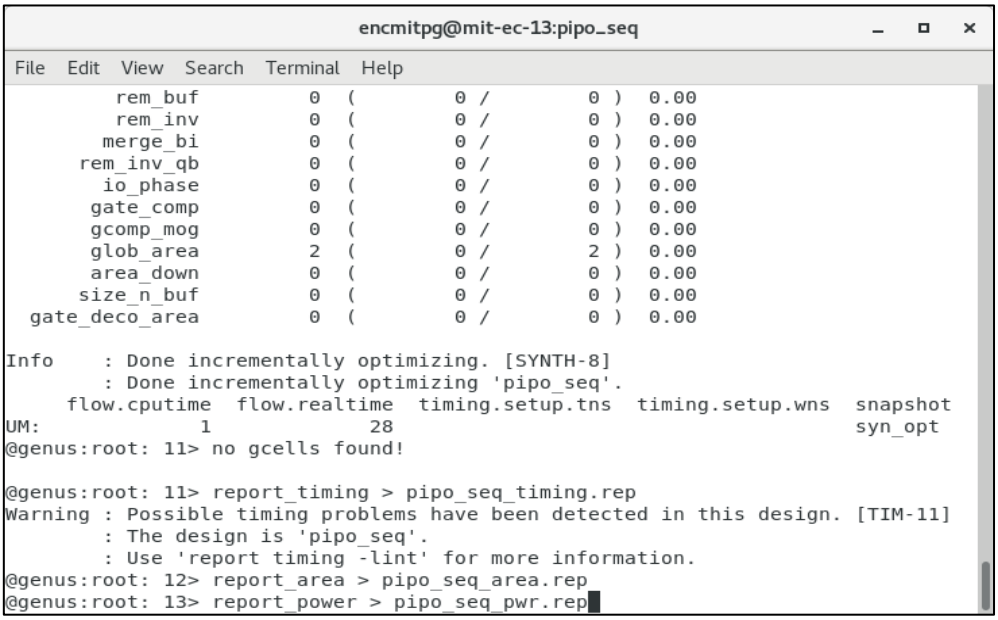


Fig. 43: Generate area and power report

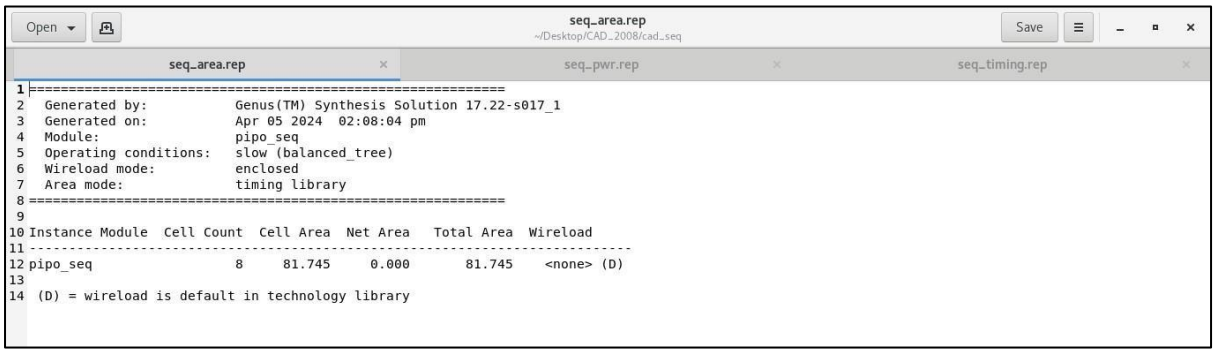


Fig. 44: Area report

Fig. 45: Power report

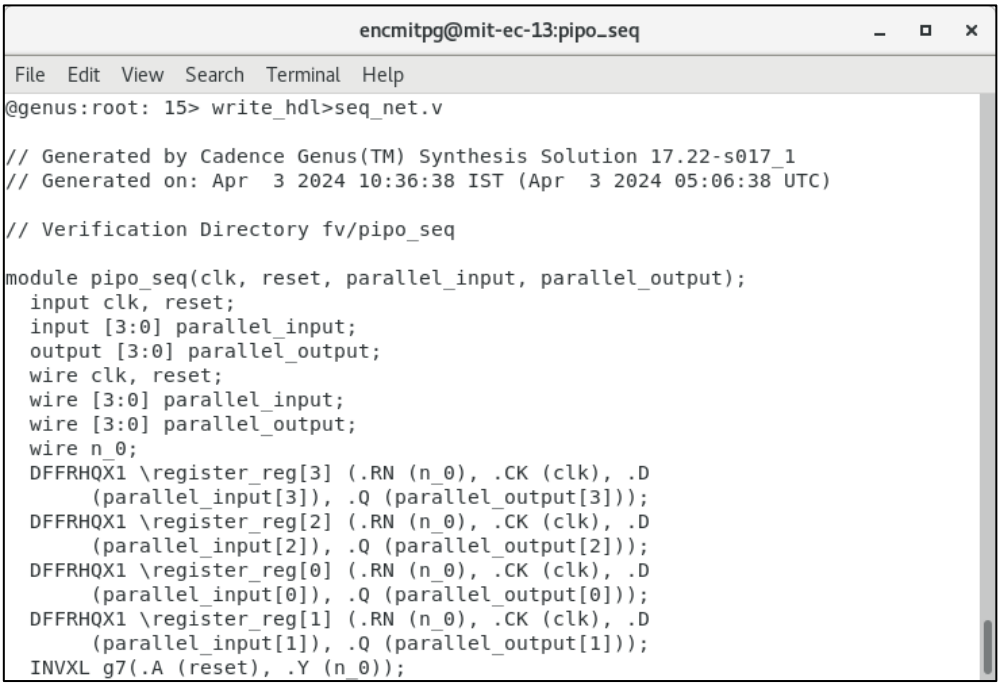


Fig. 46: Generate netlist

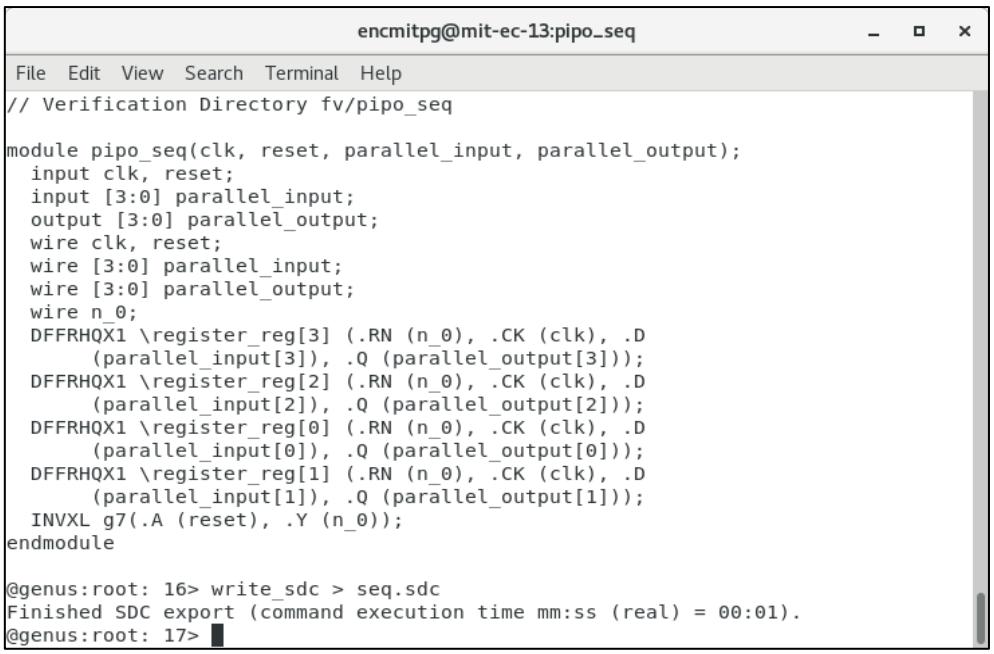


Fig. 47: Generate block level constraint file

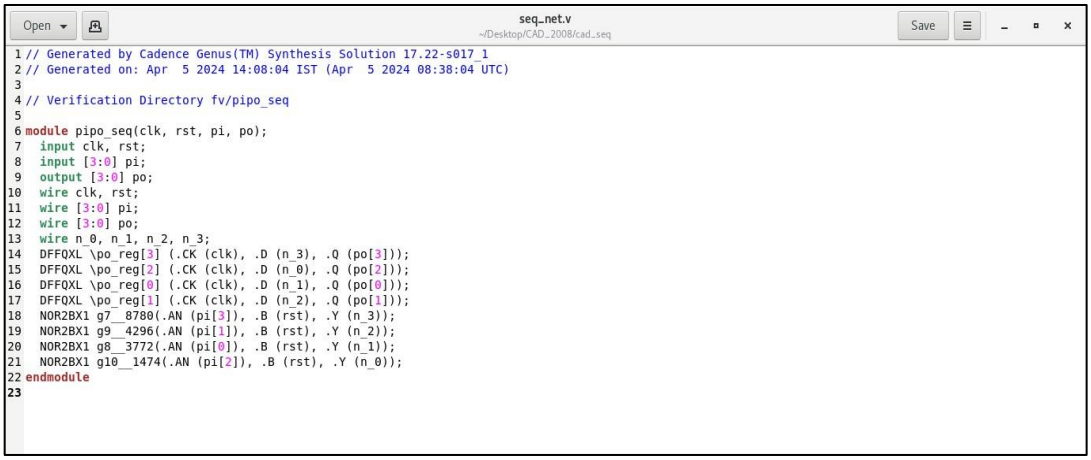


Fig. 48: Generated netlist

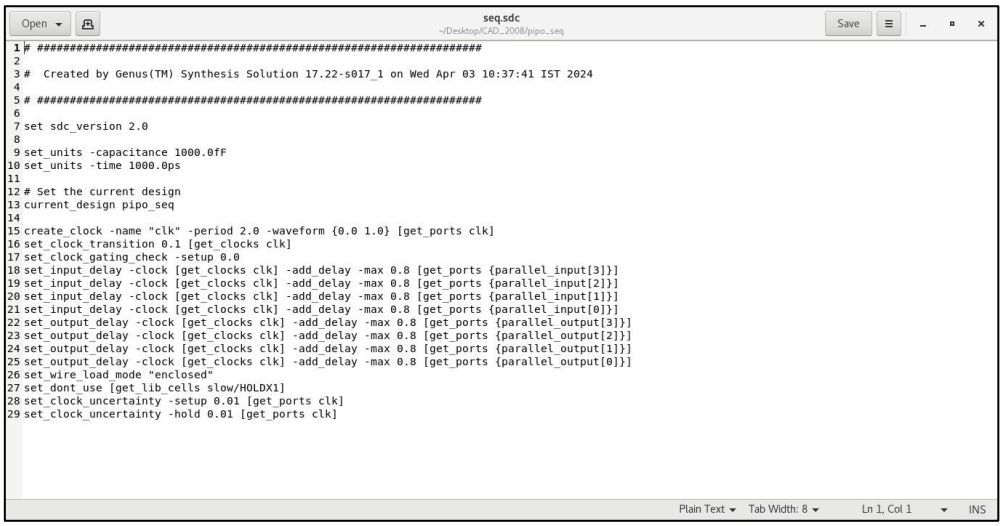


Fig. 49: Generated block level constraint file

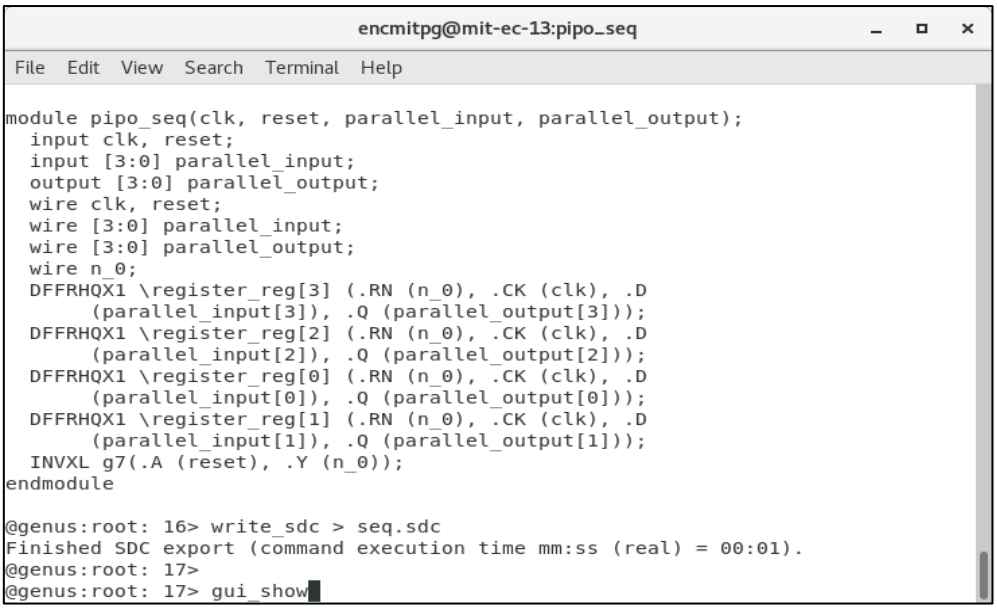


Fig. 50: Schematic generation

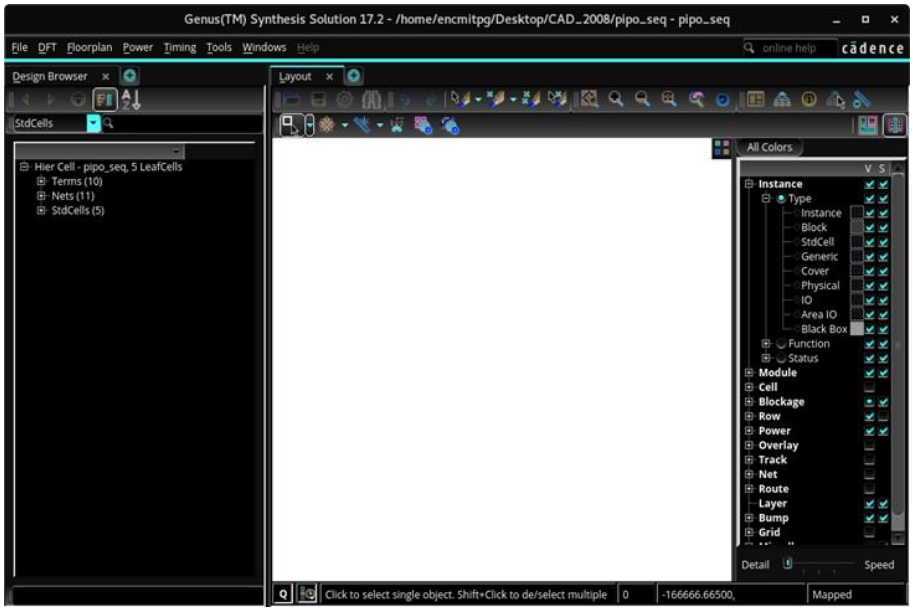


Fig. 51: Genus synthesis solution window appears

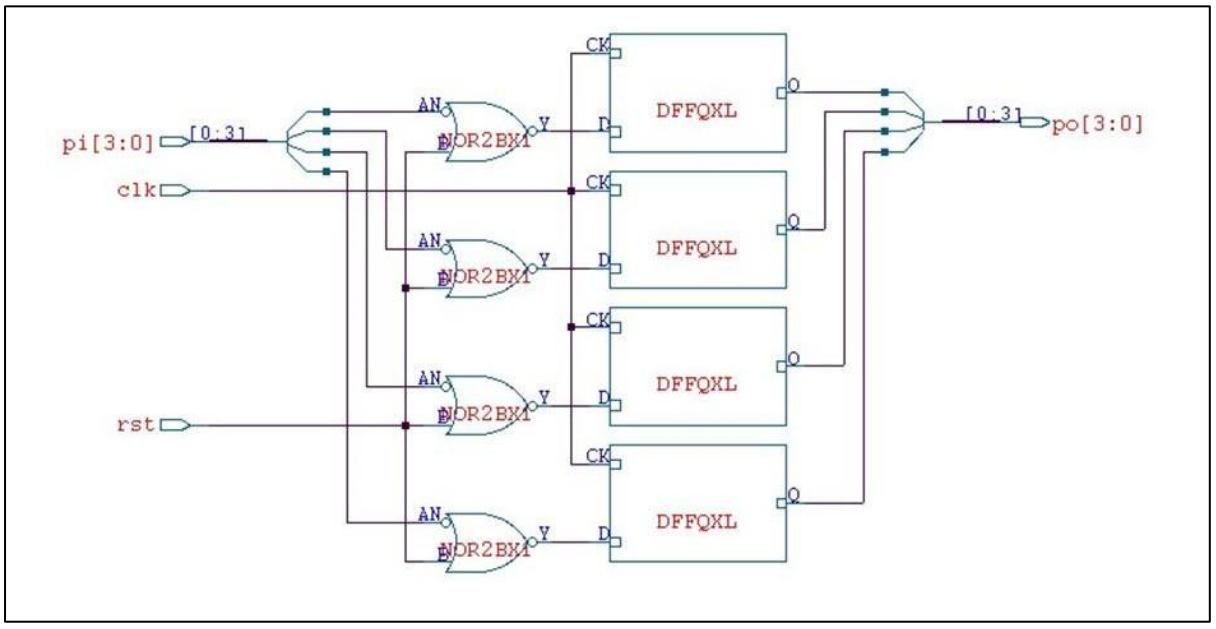


Fig. 52: Synthesis RTL Schematic

Note :-

1. report_timing gives you the path with highest failing slack where
2. Setup Slack = Required Time – Arrival Time.
3. Worst Setup Slack ==> Highest Arrival time ==> Highest Propagation Delay.
4. Maximum Clock Frequency = 1/ (Max Data Path Delay – Min Clock Path Delay + Tsetup)
5. All the Information can be gathered from report_timing.
6. The Cells given in the netlist can be checked in the .lib files for their properties.

Steps to create tcl file

Fig. 53: Create tcl file

- Copy the commands from genus folder created in the pipo_seq folder

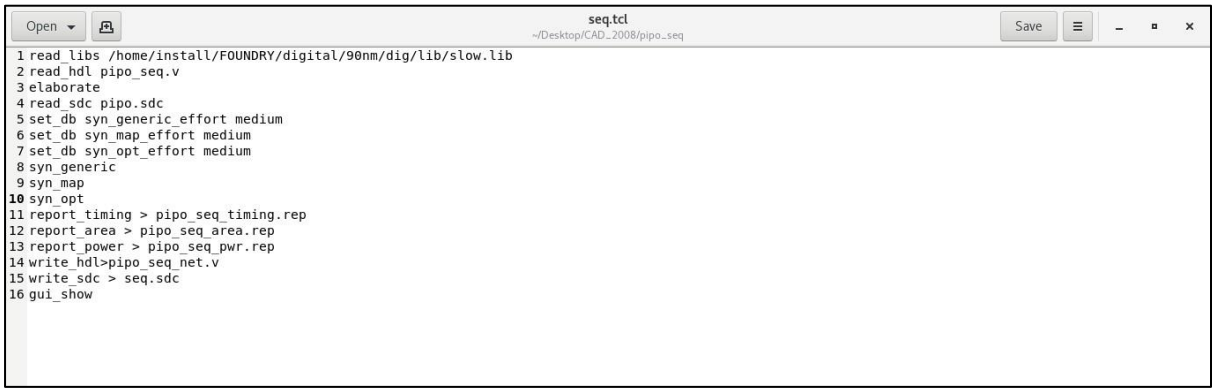


Fig. 54: TCL file

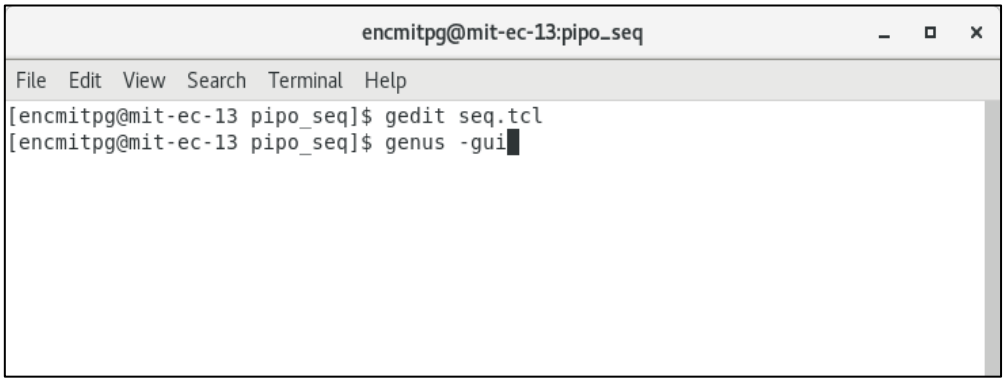


Fig. 55: Open genus

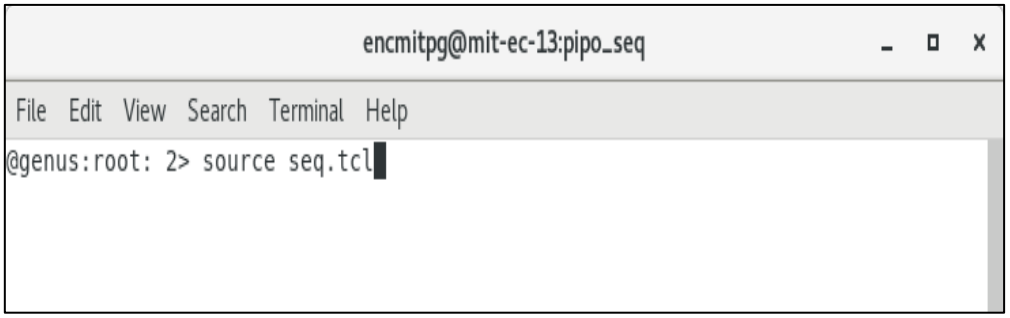


Fig. 56: Source tcl file

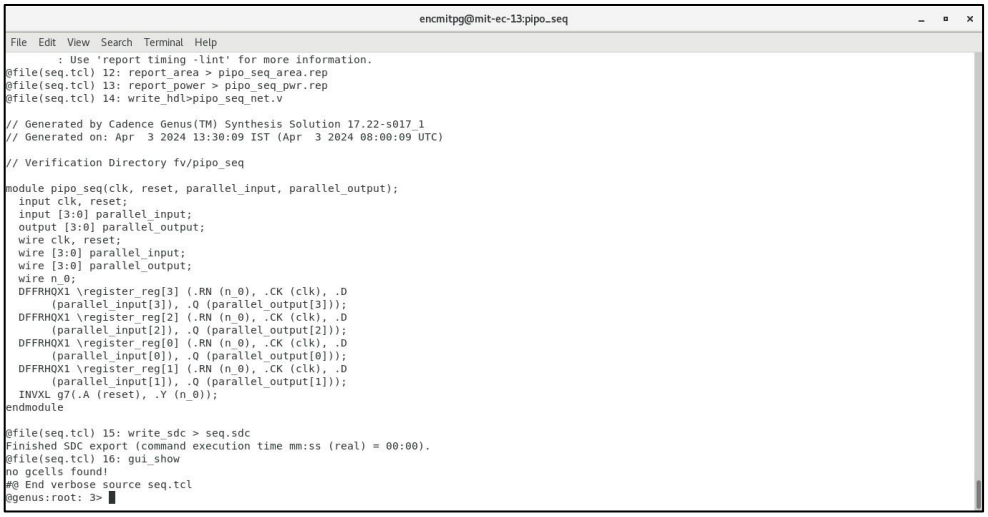


Fig. 57: End genus by giving exit command

Physical Design:

Tool Required:

- Functional Simulation: Incisive Simulator (ncvlog, ncelab, ncsim)
- Synthesis: Genus
- Physical Design: Innovus

Physical Design involves 5 stages as following:

After Importing Design,

- Floor Planning
- Power Planning
- Placement
- CTS (Clock Tree Synthesis)
- Routing Module

Else, if you would like to import your design using GUI, open the Innovus tool and from the

- GUI, go to File → Import Design.
- A new pop-up window appears.
- First load the netlist. You can browse for the file and select “Top cell : Auto Assign”.

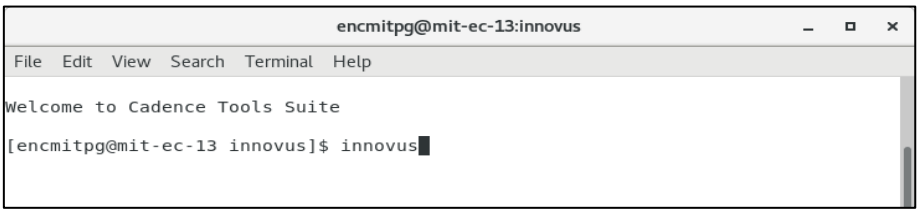


Fig. 58: Terminal to open innovus window

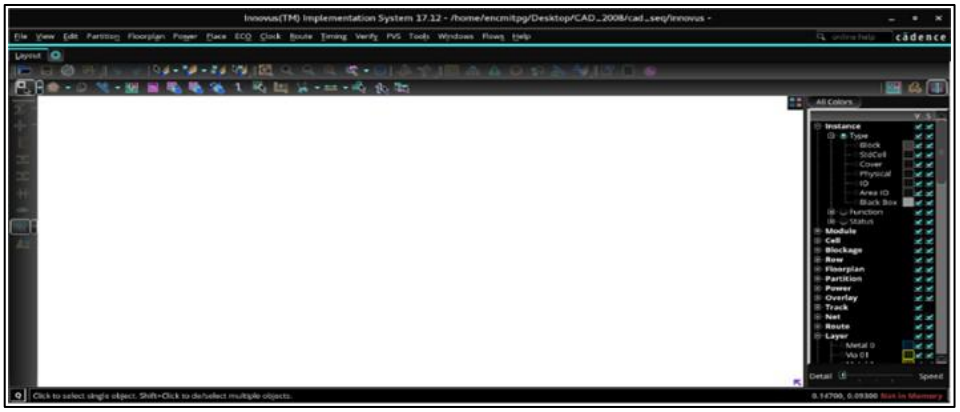


Fig. 59: Innovus window

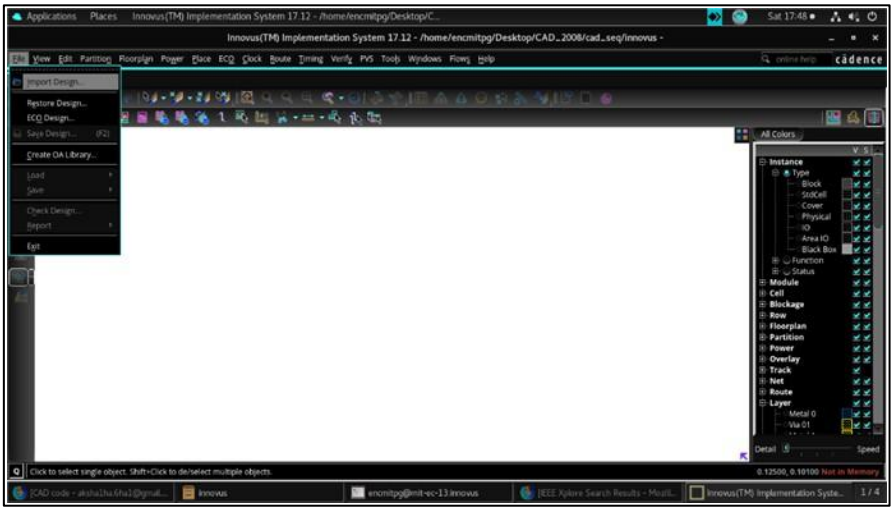


Fig. 60: Click import design

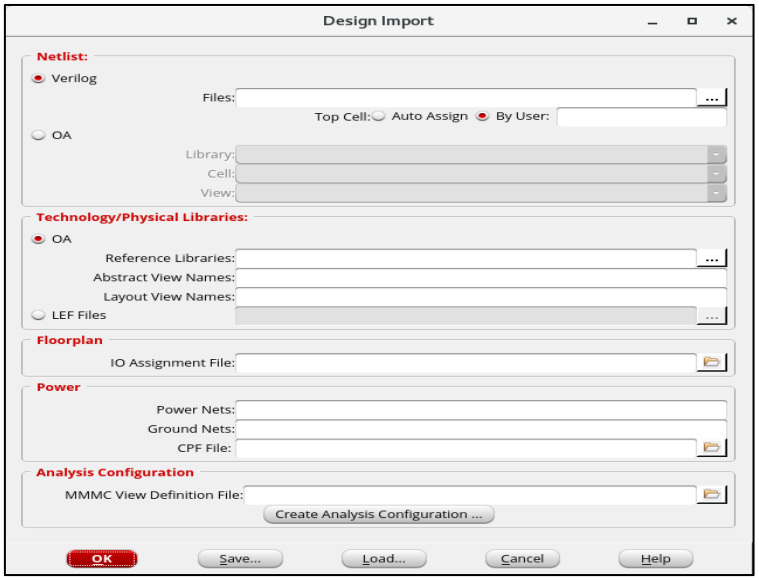


Fig. 61: Design import window

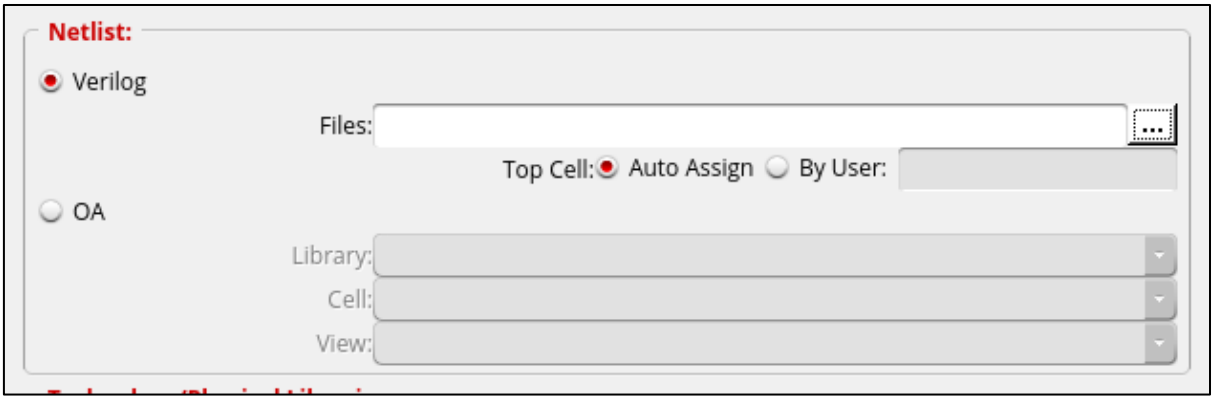


Fig. 62: Select auto design

Fig. 63: Under Verilog section click on browse option to select the netlist file

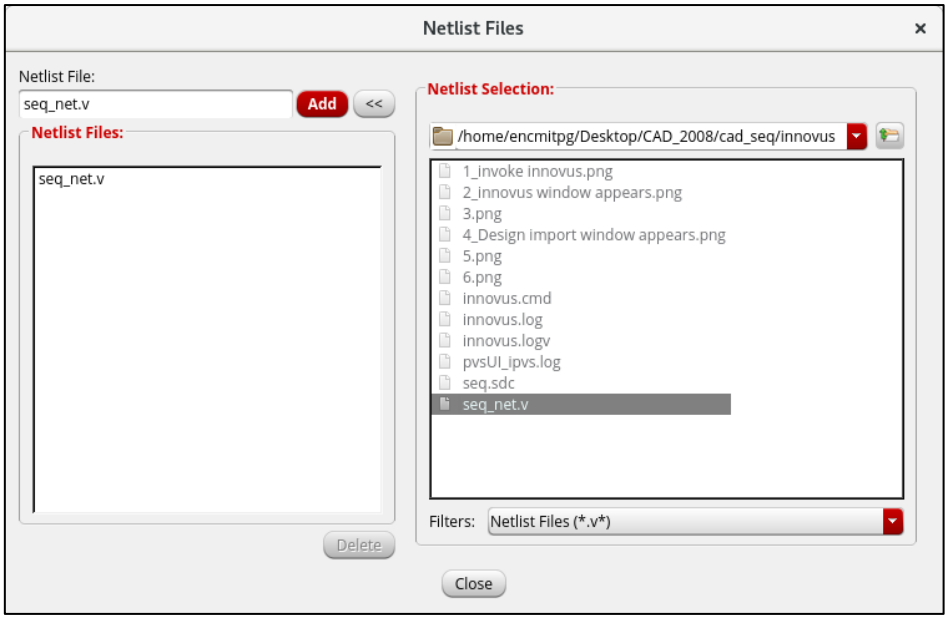


Fig. 64: Select the netlist file generated in the synthesis step

Similarly select your lef files from /home/install/FOUNDRY/digital/90nm/dig/lef/ as shown below

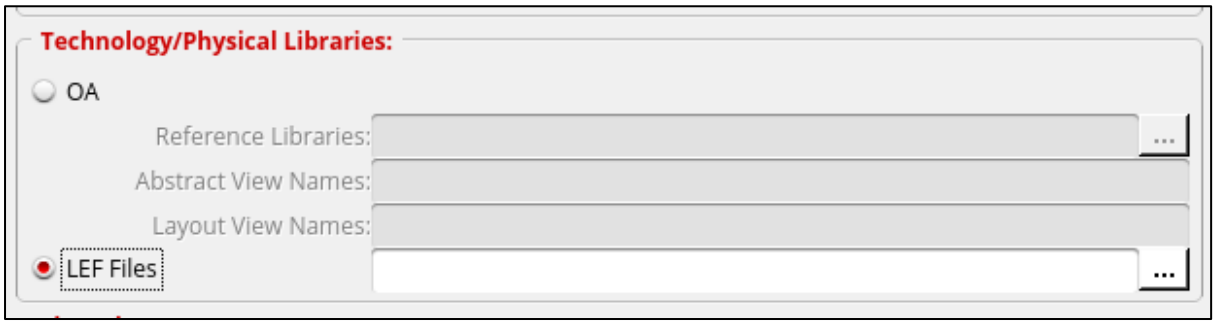


Fig. 65: select LEF files and click on browse option to add LEF files

Fig. 66: click on ">>" button to add LEF files

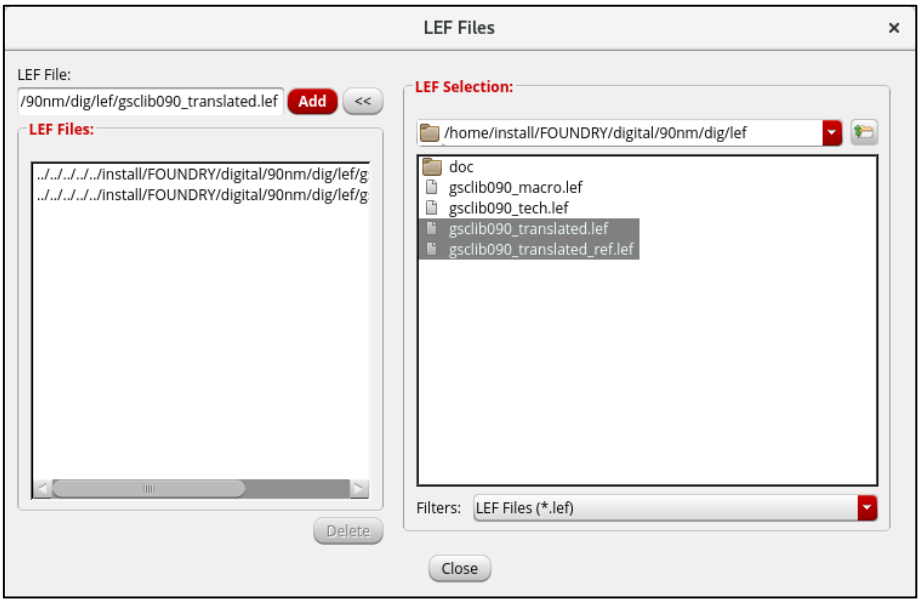


Fig. 67: Select both the LEF files

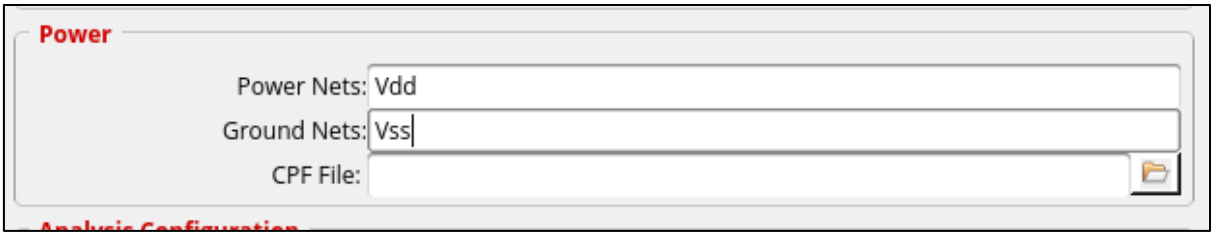


Fig. 68: Power nets and ground nets

In order to load the Liberty File and SDC, create delay corners and analysis view, select the "Create Analysis Configuration" option at the bottom.

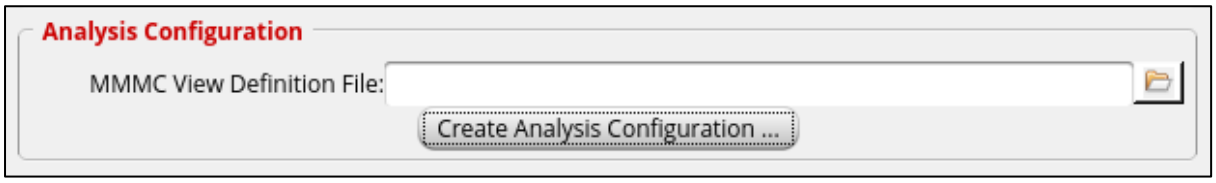


Fig. 69: click on create analysis configuration

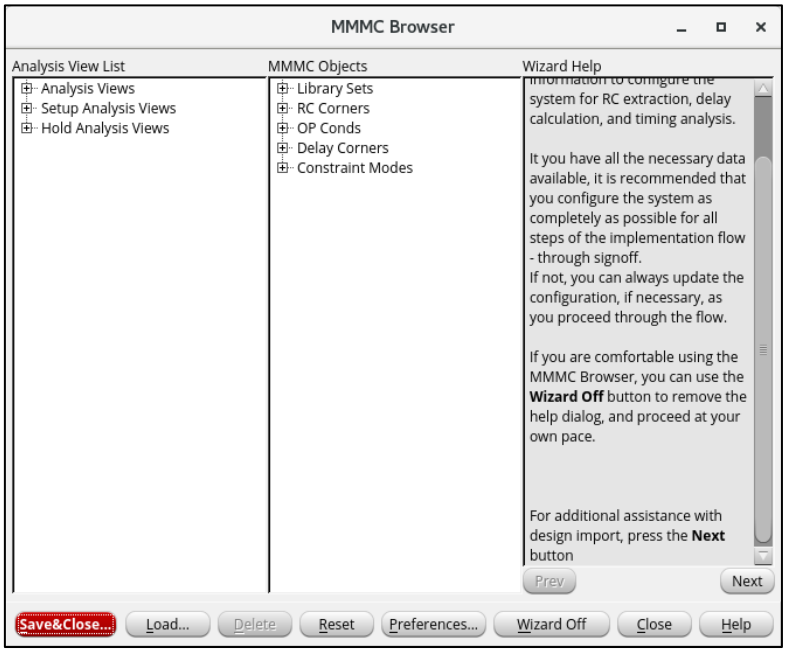


Fig. 70: MMMC Browser window

The order of adding the MMMC Objects is as follows.

1. Library Sets
2. RC Corners
3. Delay Corners
4. Constraints (SDC

- Once all of them are added, Analysis Views are created and assigned to Setup and Hold.
- In order to add any of the objects, make a right click on the corresponding label → Select New.

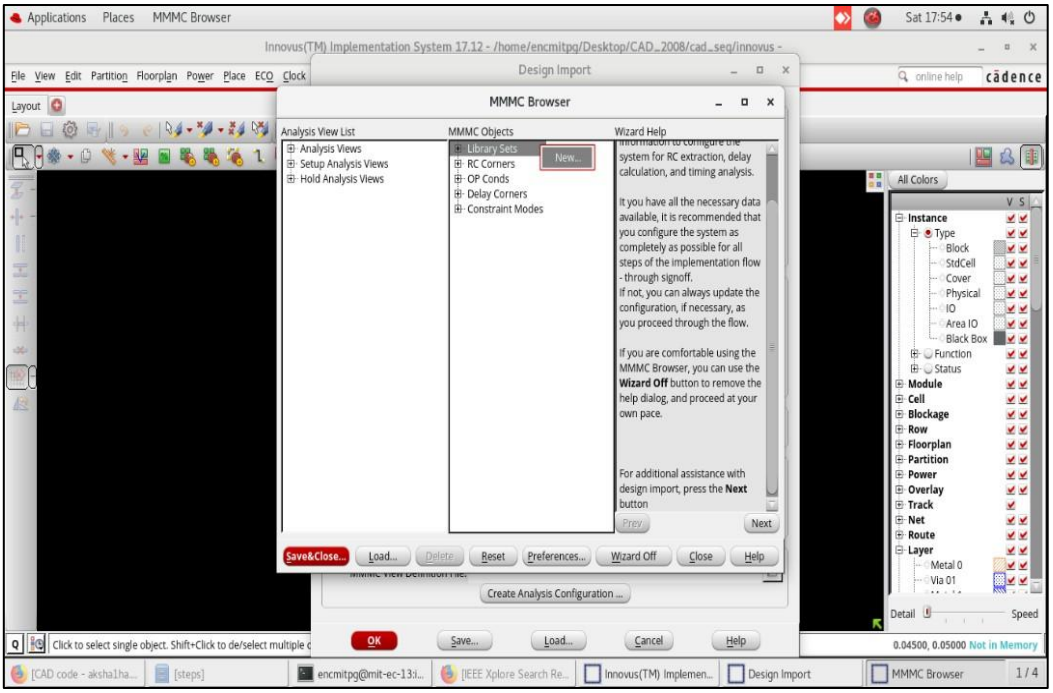


Fig. 71: Right click on library sets -> click on new

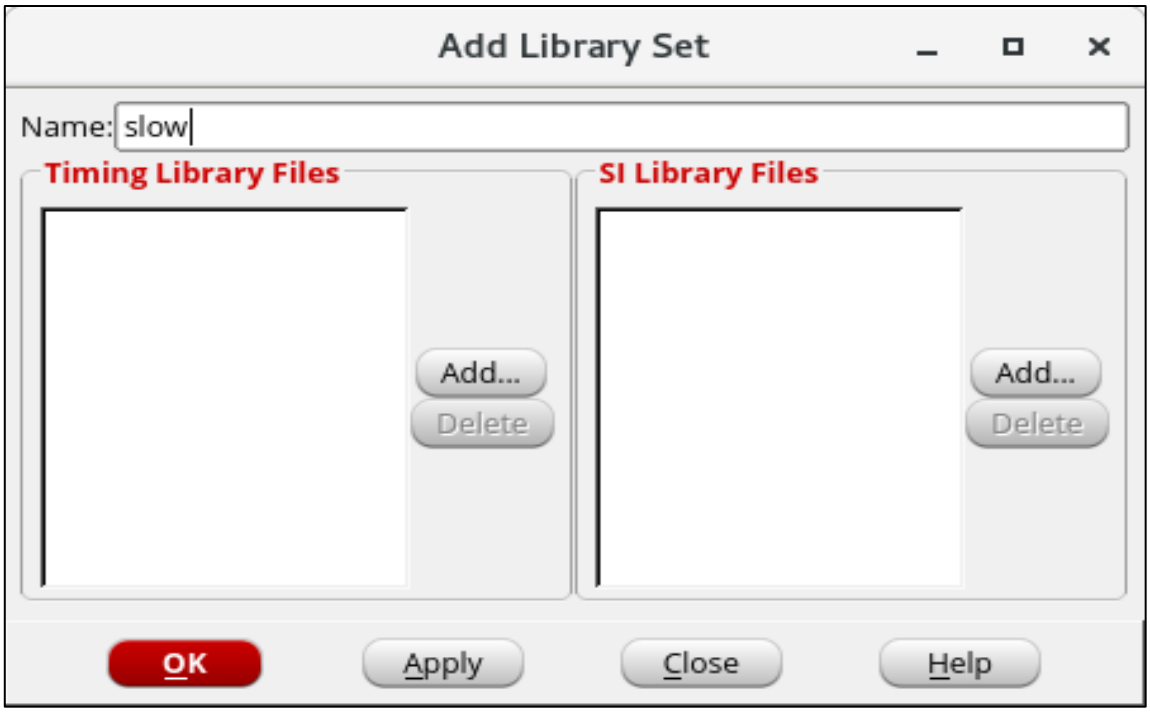


Fig. 72: Give name as slow and add slow.lib file

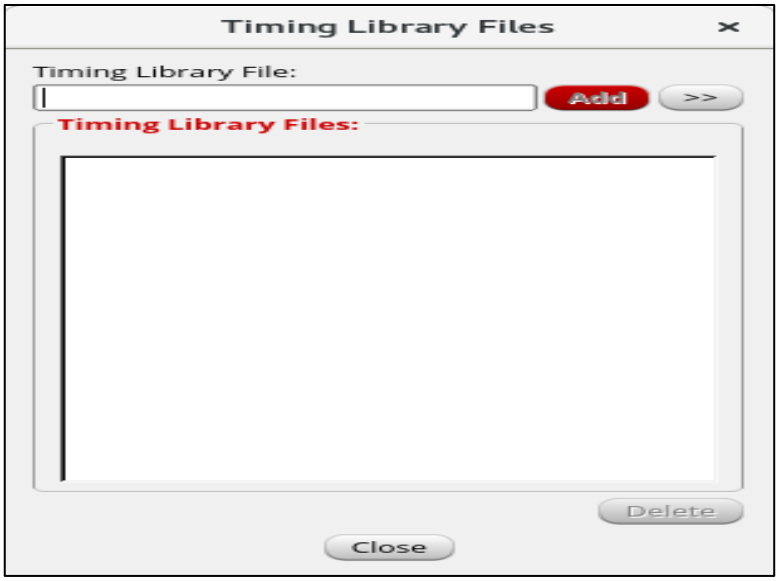


Fig. 73: click on “>>” button to add library file

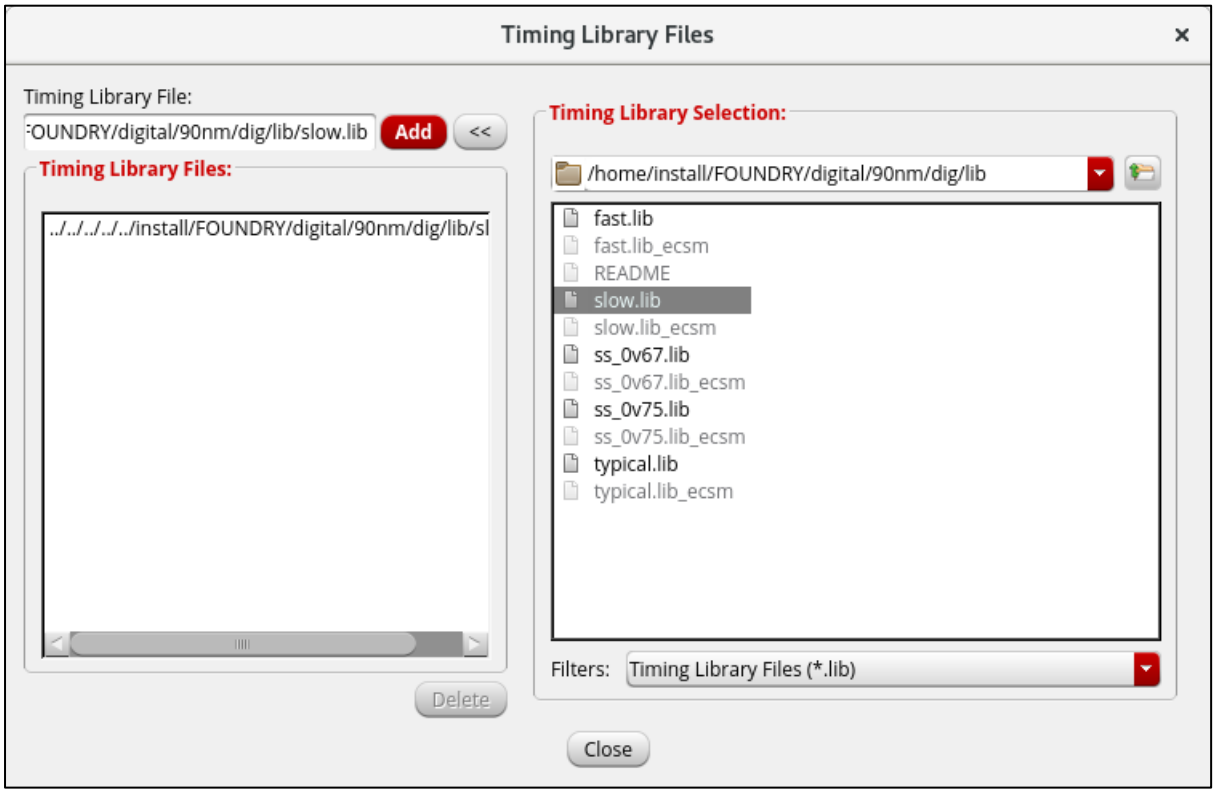


Fig. 74: select slow.lib file

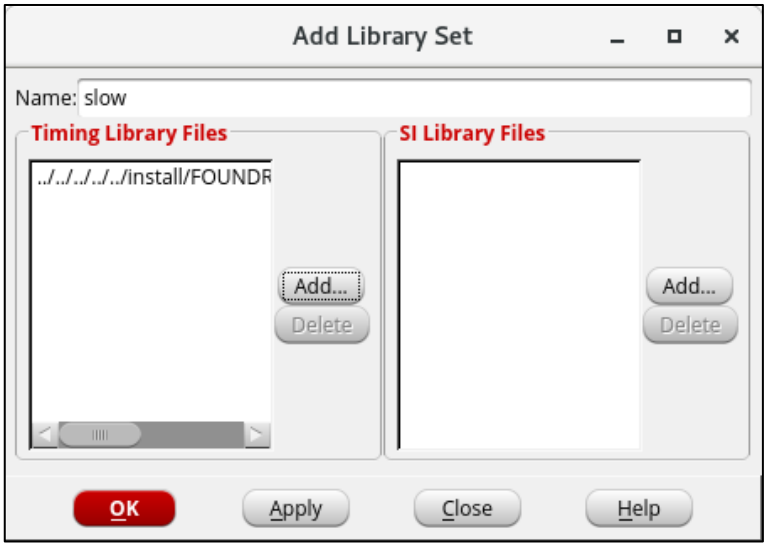


Fig. 75: click on add button

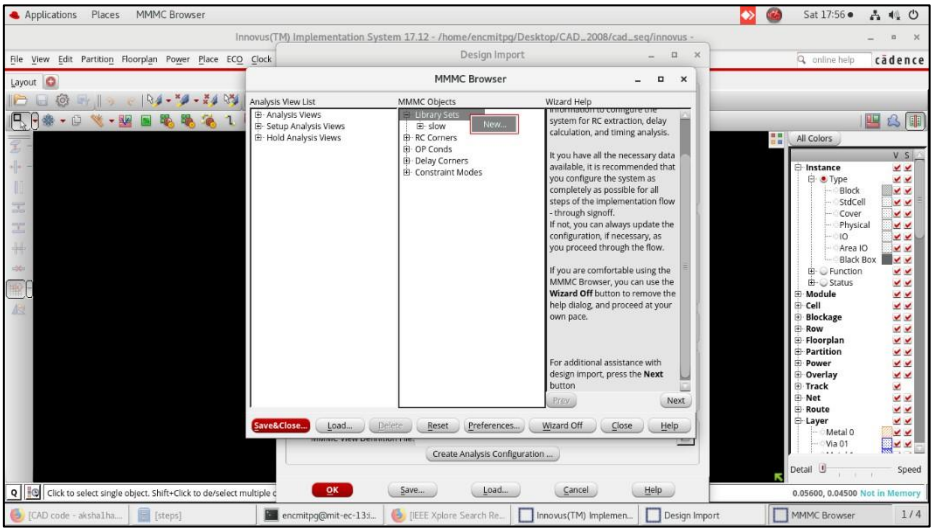


Fig. 76: Right click on library sets then click on new

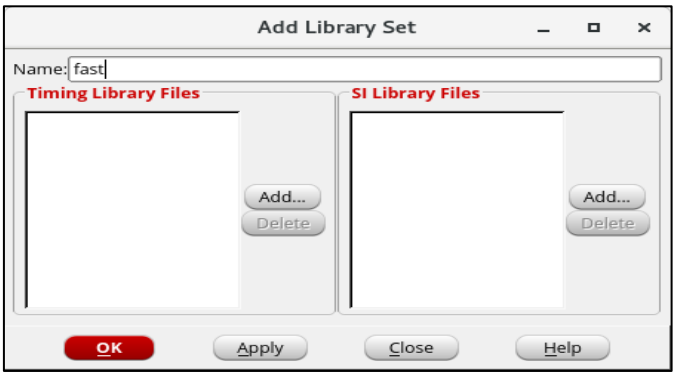


Fig. 77: Give name as fast

Fig. 78: click on “>>” button to add fast lib

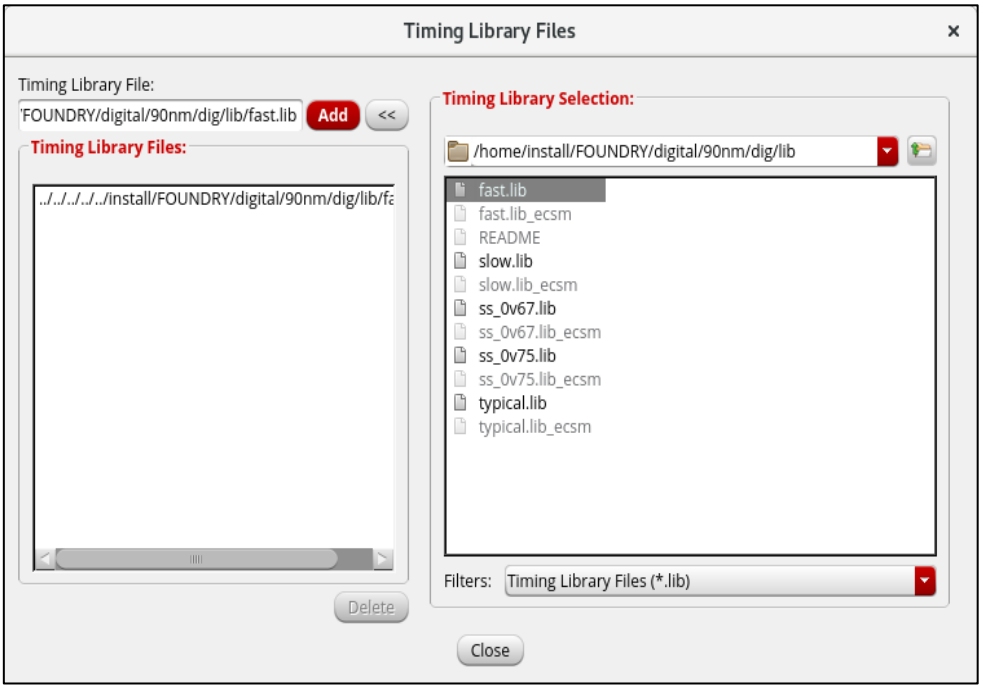


Fig. 79: select fast.lib

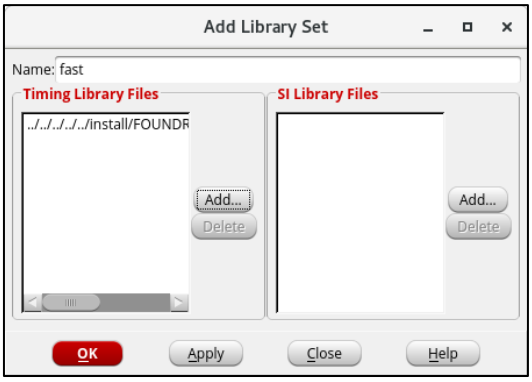


Fig. 80: Click on add button

Adding RC Corners can also be done in a similar process. The temperature value can be found under the corresponding liberty file. Also, cap table and RC Tech files can be added from Foundry where available.

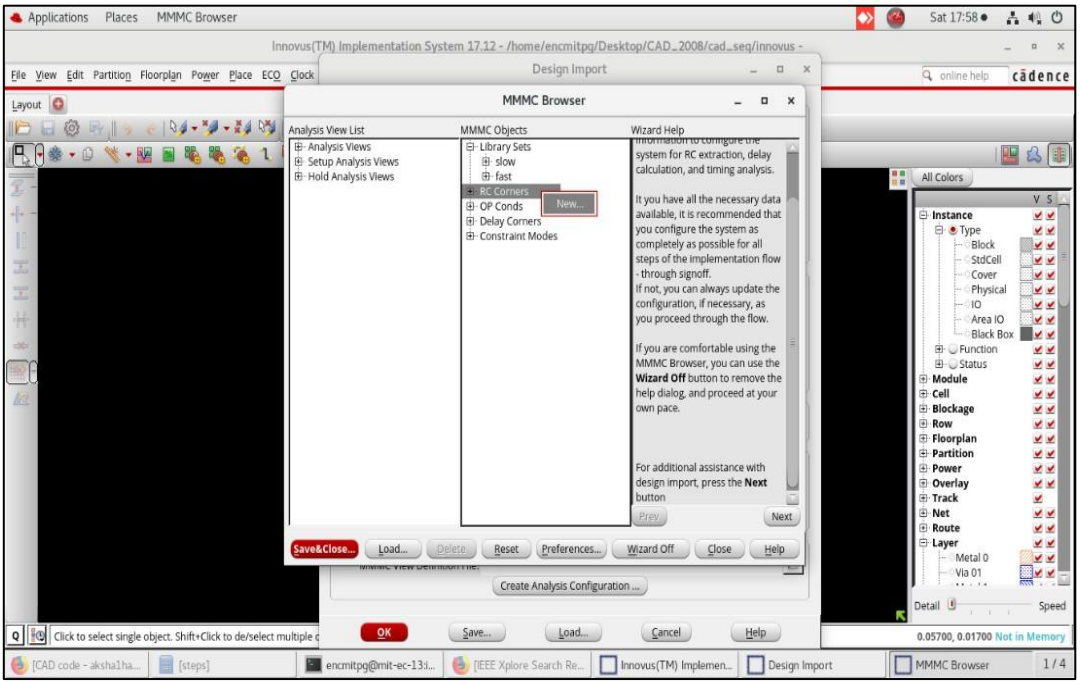


Fig. 81: Right click on RC corner -> new

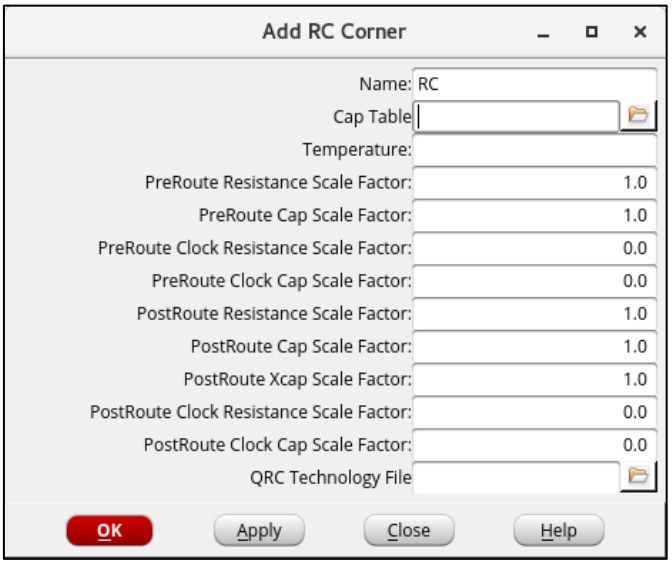


Fig. 82: click on browse option to select cap table

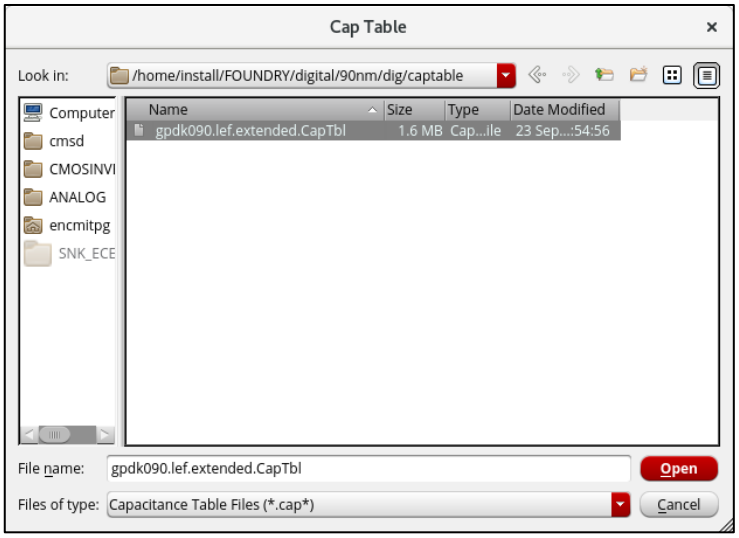


Fig. 83: select cap table

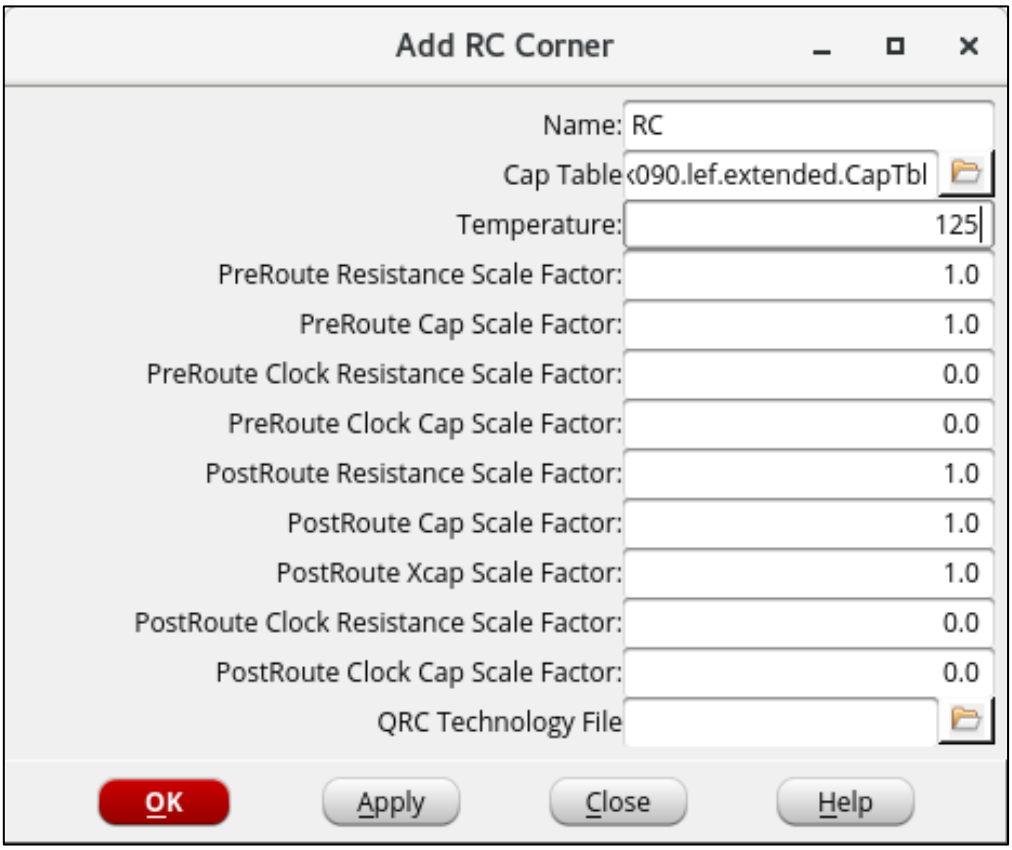


Fig. 84: Give temperature as 125

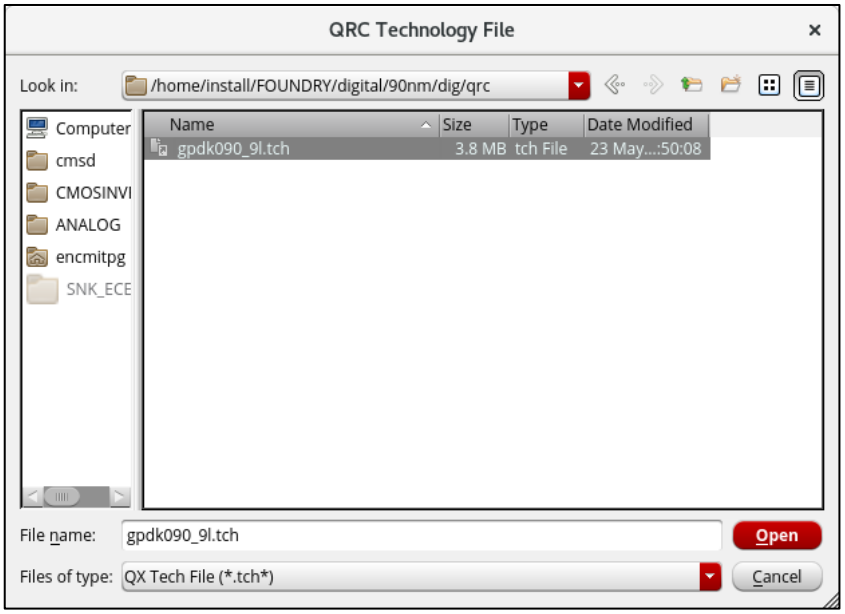


Fig. 85: select qrc file

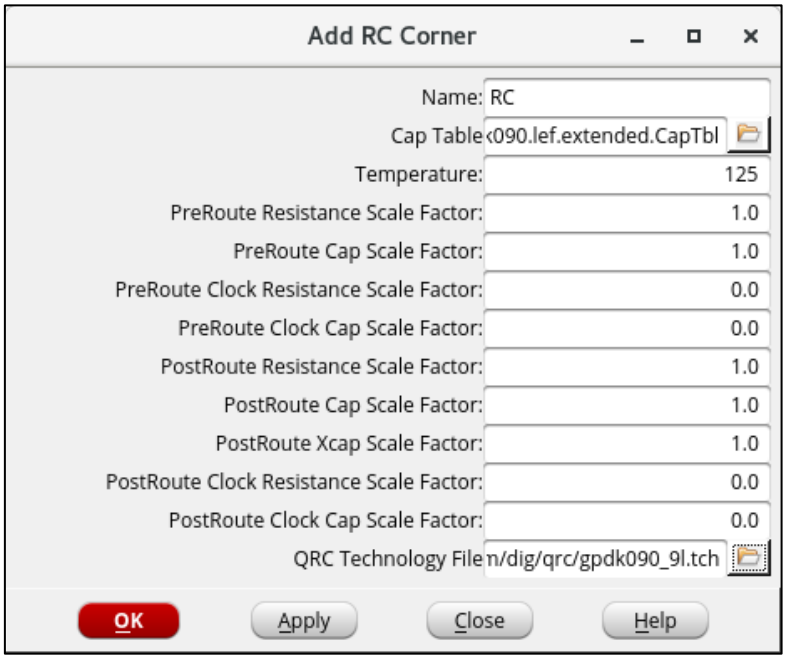


Fig. 86: Click OK

- In the similar way, add minimum RC corner by giving temperature as ‘0’.
- Delay Corners are formed by combining Library Sets with RC Corners.

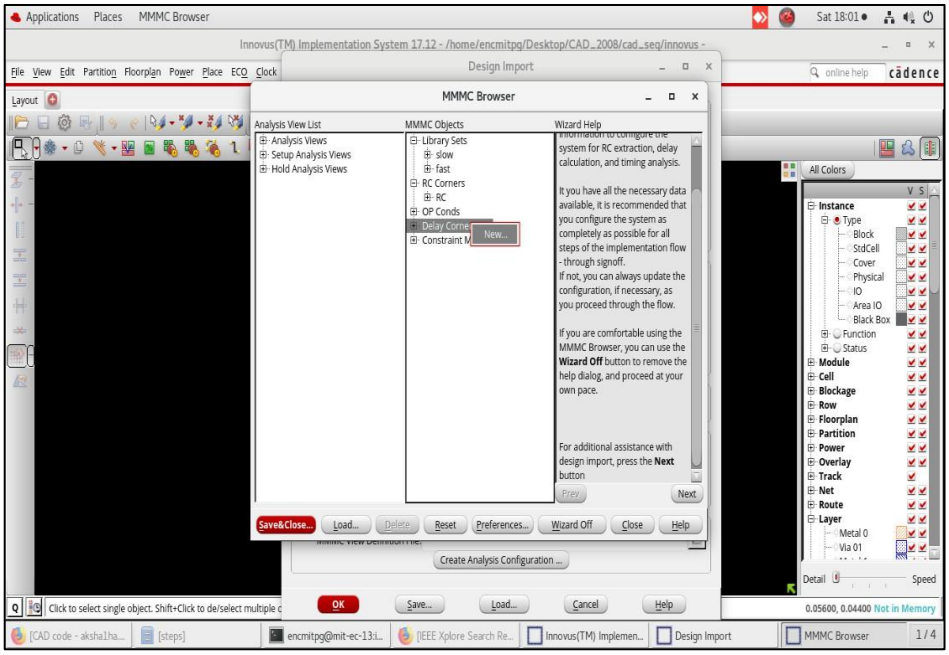


Fig. 87: Right click on delay corner -> new

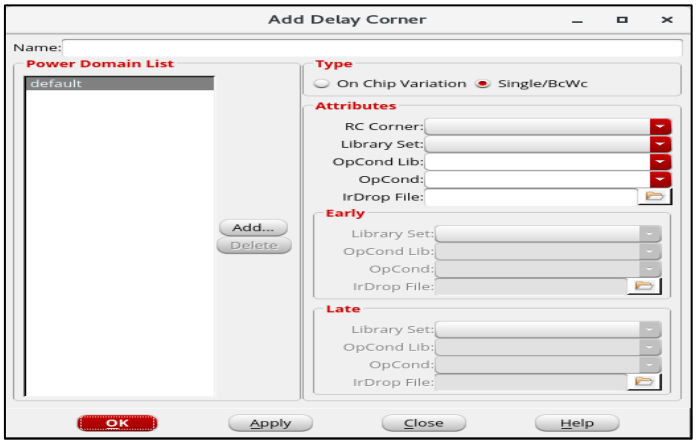


Fig. 88: Delay corner window

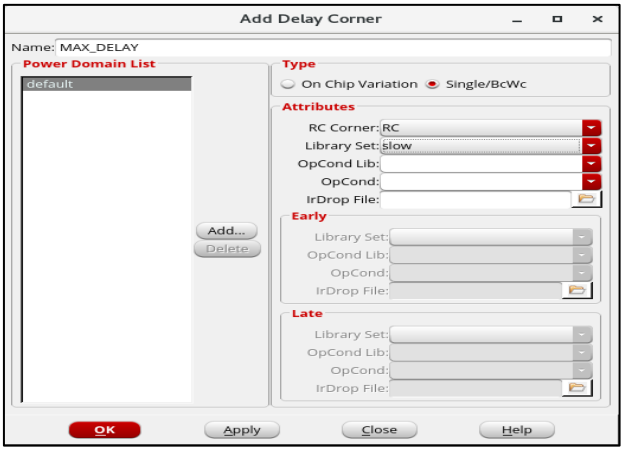


Fig. 89: Select RC_max and slow library set for maximum delay

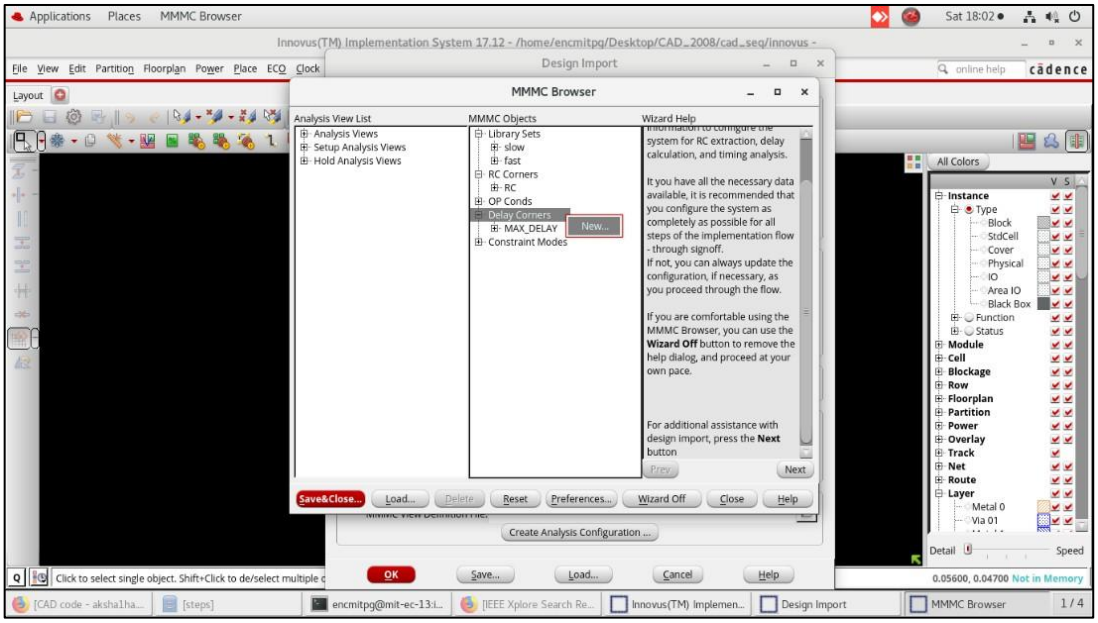


Fig. 90: Right click on delay corner -> new

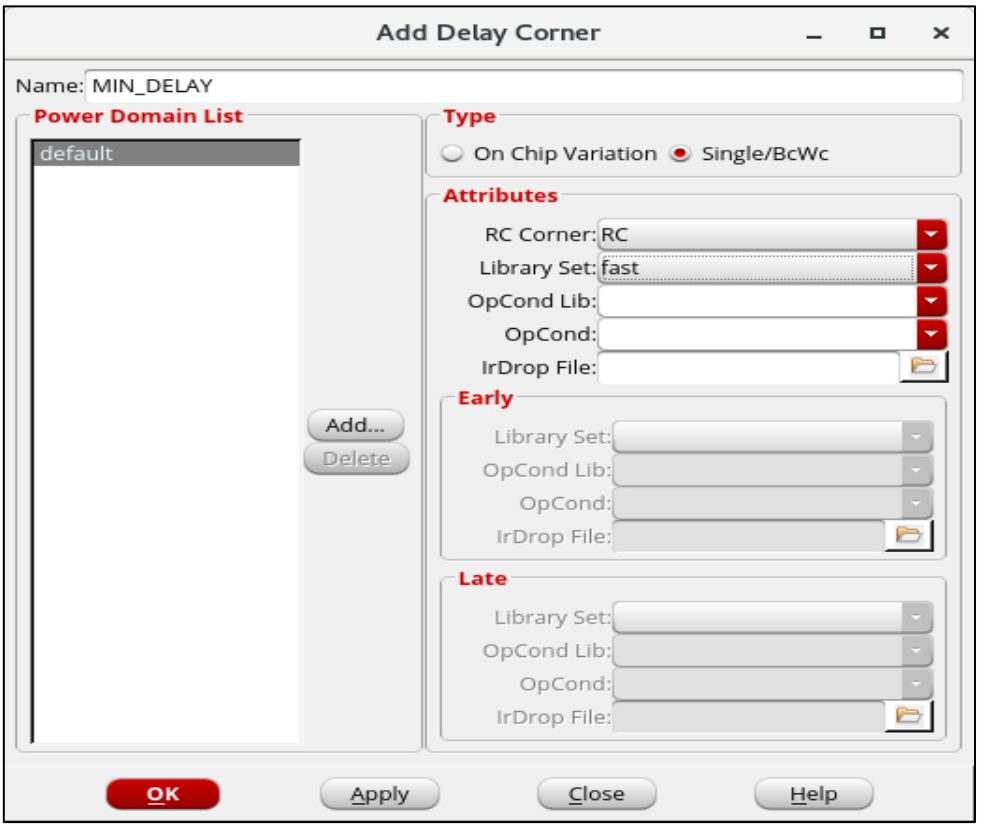


Fig. 91: select RC_min and fast library set for minimum delay

- SDC can be read in under the MMMC Object of “Constraints”.

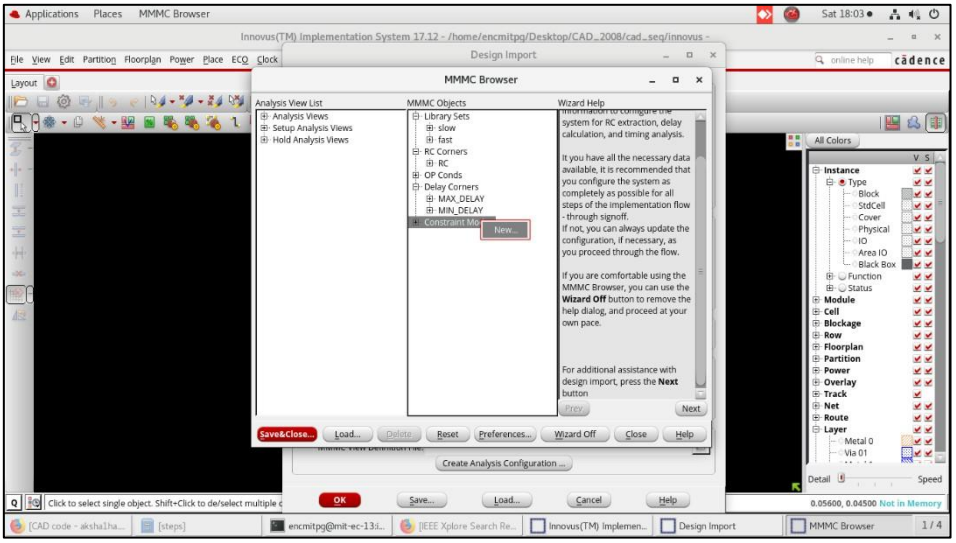


Fig. 92: Right click on constraints file -> new

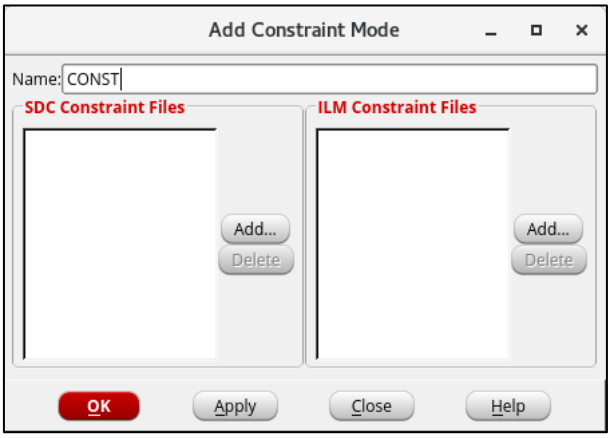


Fig. 93: Give name as CONST and click on add to select the constraint file

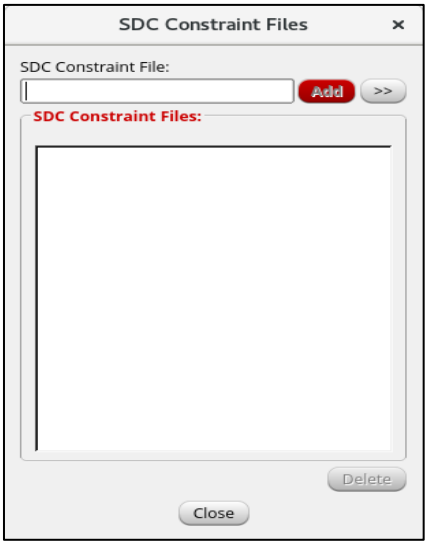


Fig. 94: Click on “>>”

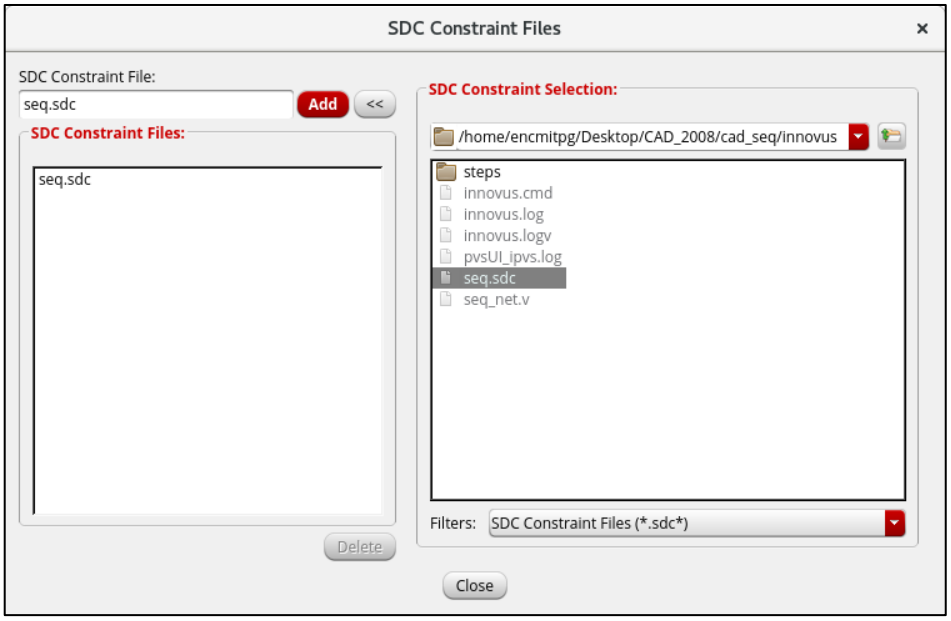


Fig. 95: Select the constraint file

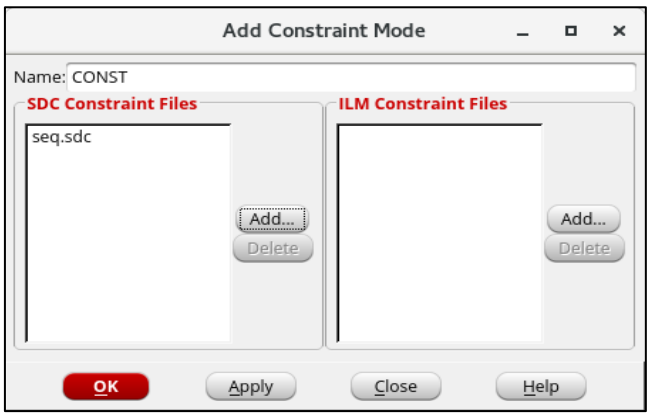


Fig. 96: click on ADD

Analysis Views are formed from combinations of SDC and Delay Corner.

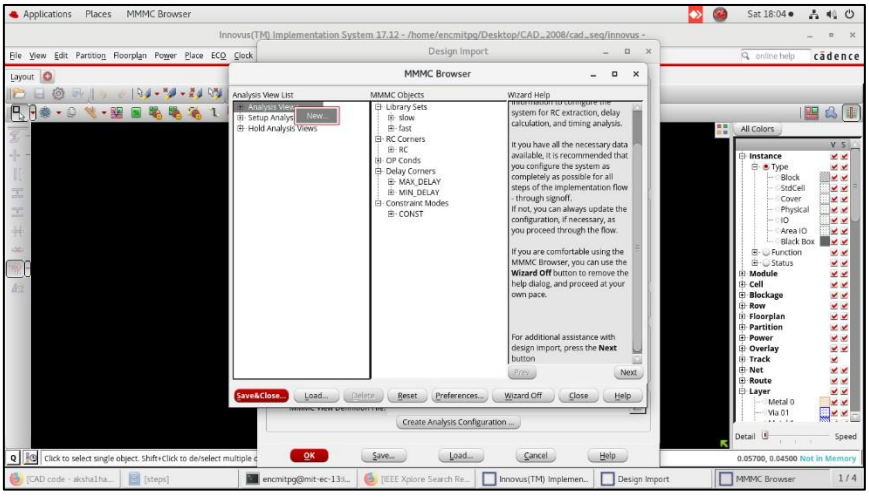


Fig. 97: Right click on Analysis View -> new

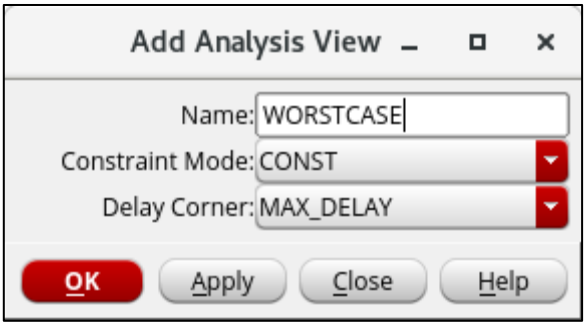


Fig. 98: Worst_case scenario

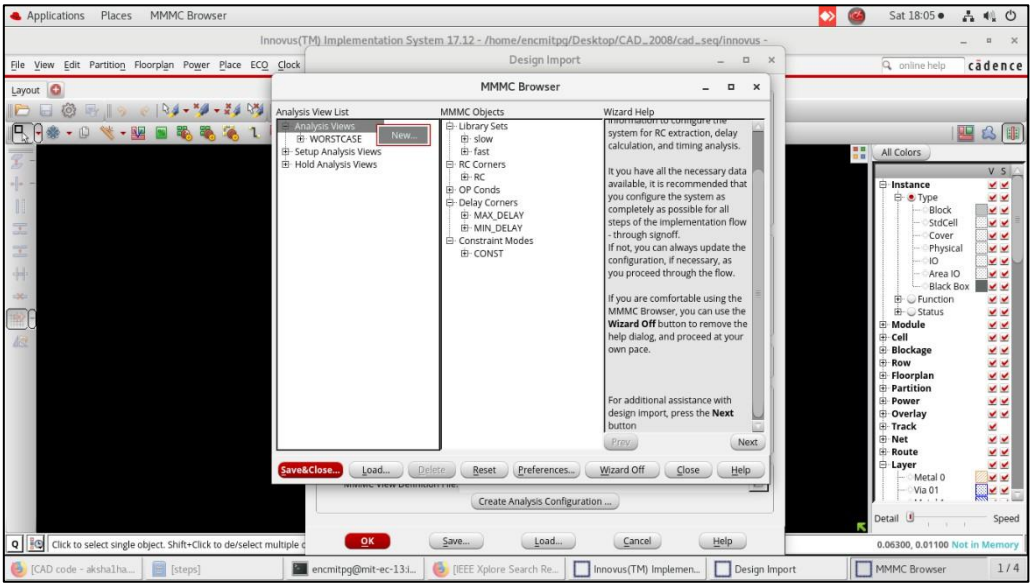


Fig. 99: Right click on Analysis window -> new

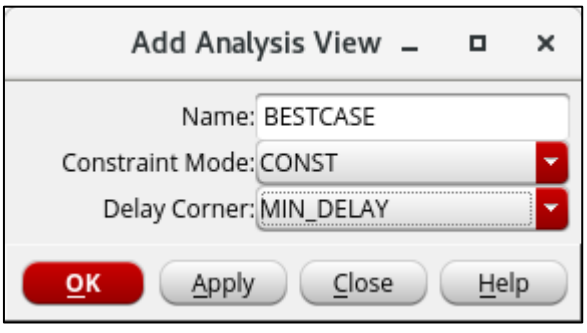


Fig. 100: Best_ case scenario

- Once “Best” and “Worst” Analysis views are created, assign them to Setup and Hold

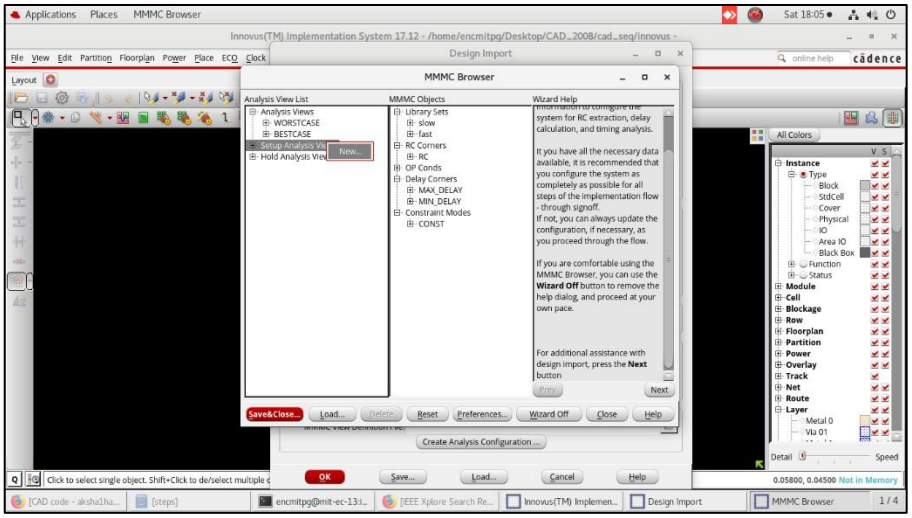


Fig. 101: Right click on setup analysis -> new

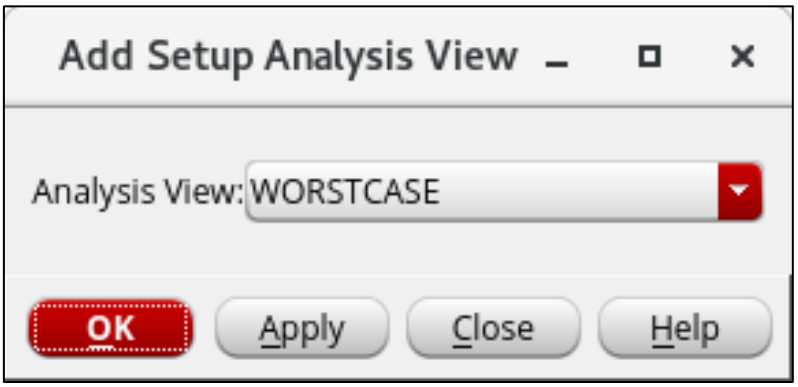


Fig. 102: Select worstcase

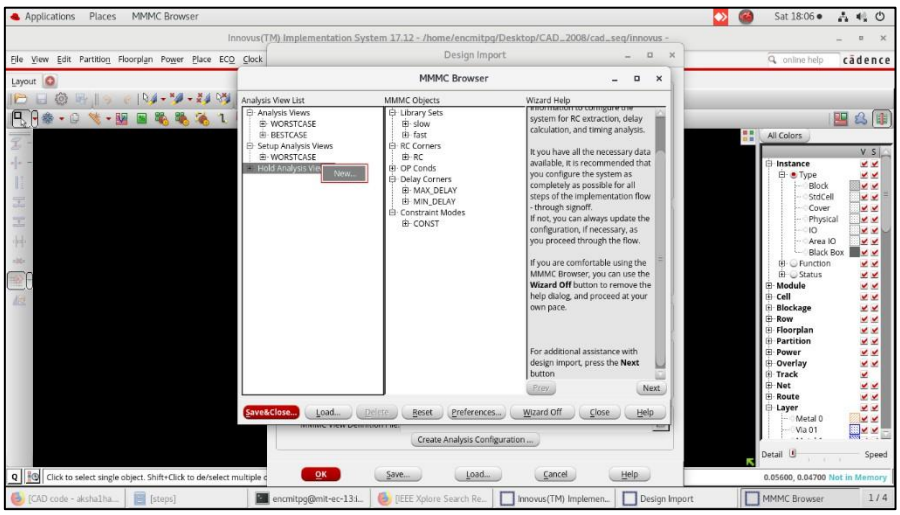


Fig. 103: Right click on Hold analysis -> new

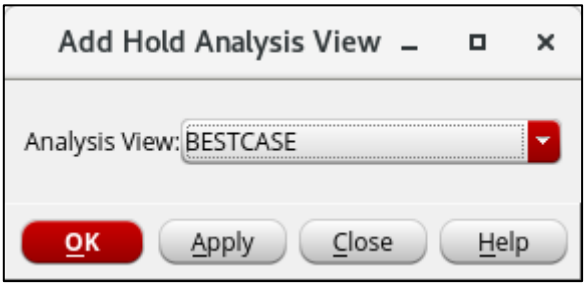


Fig. 104: Select Bestcase

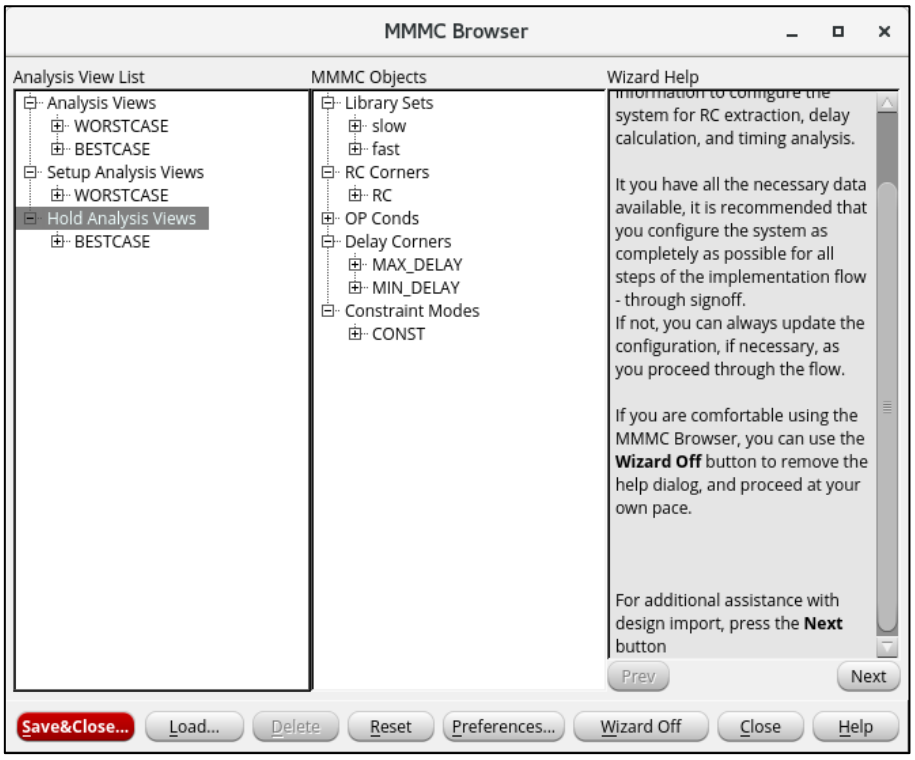


Fig. 105: In MMMC Browser window, click on save and close

- Once all the process is done, click on “Save & Close” and save the script generated with any name of your choice.
- Make sure the file extension remains .view or .tcl
- After saving the script, go back to Import Design window and Click “OK” to load your design.

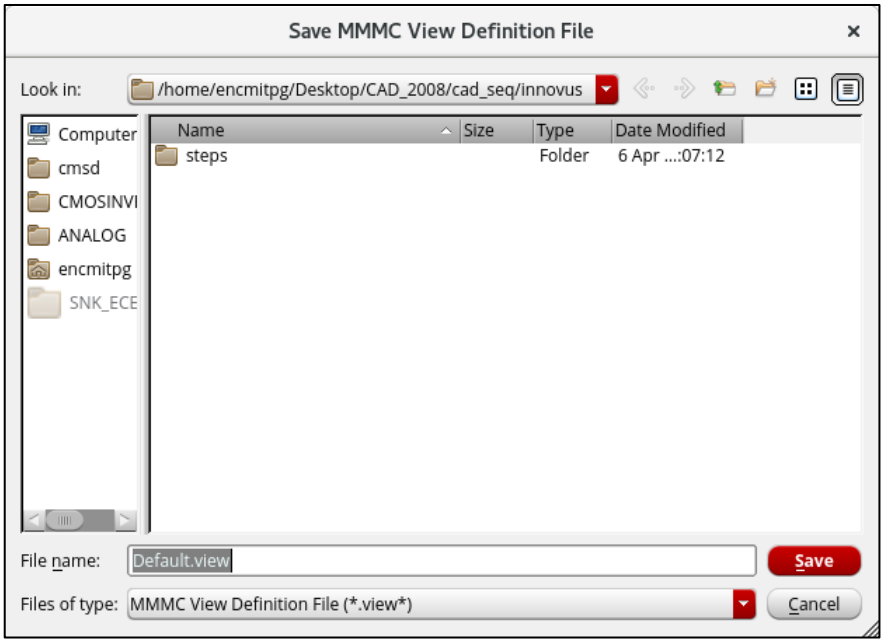


Fig. 106: Click on save

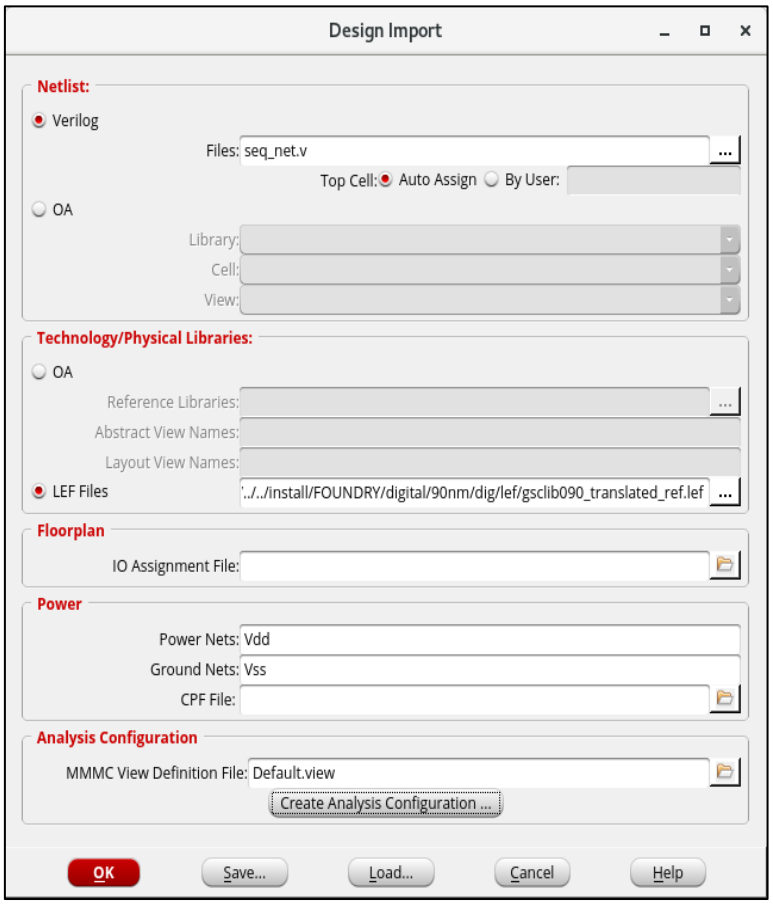


Fig. 107: Click on save

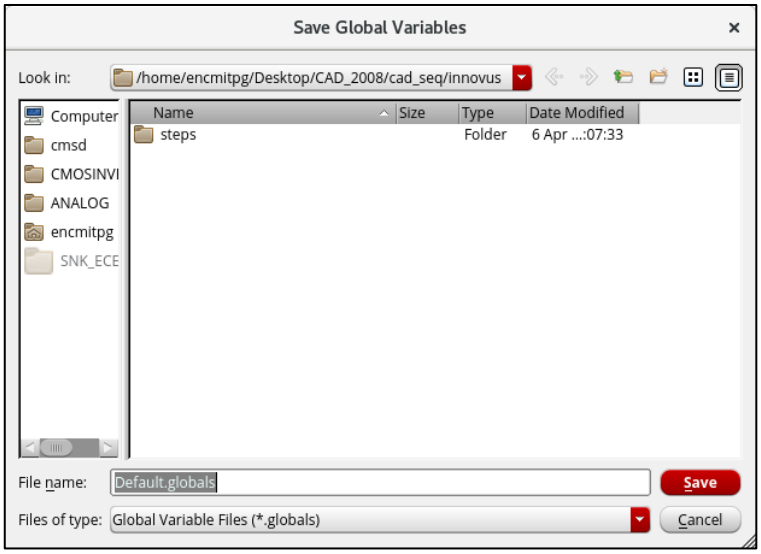


Fig. 108: Click on save and then give Ok

- A rectangular or square box appears in your GUI if and only if all the inputs are read properly.
- If the box does not appear, check for errors in your log (Either on terminal or log file from pwd)

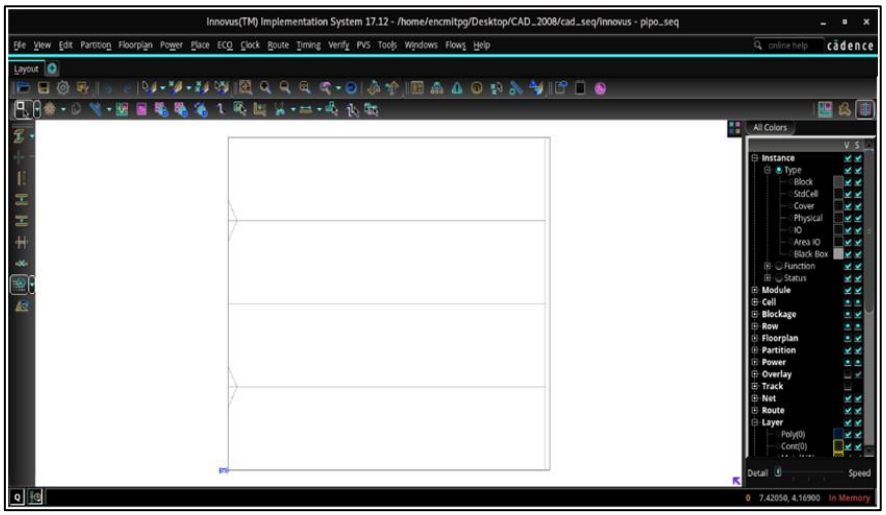


Fig. 109: Innovus window

- The internal area of the box is called “Core Area”.
- The horizontal lines running along the width of Core are “Standard Cell Rows”. Every alternate of them are marked indicating alternate VDD and VSS rows.
- This setup is called “Flipped Standard Cell Rows”

Floorplan

Steps under Floorplan :

1. Aspect Ratio [Ratio of Vertical Height to Horizontal Width of Core]
2. Core Utilisation [The total Core Area % to be used for Floor Planning]
3. Channel Spacing between Core Boundary to IO Boundary

- Select Floorplan → Specify Floorplan to modify/add concerned values to the above Factors.
- On adding/modifying the concerned values, the core area is also modified.

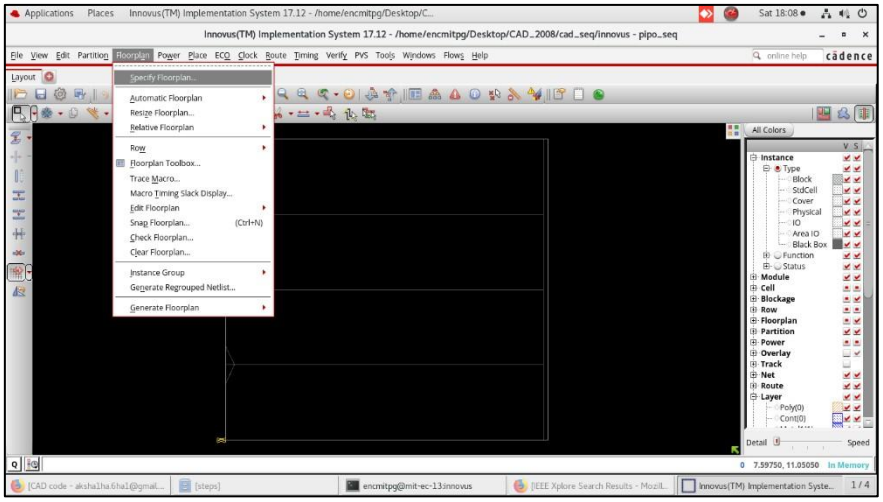


Fig. 110: click on floor plan -> specify floorplan

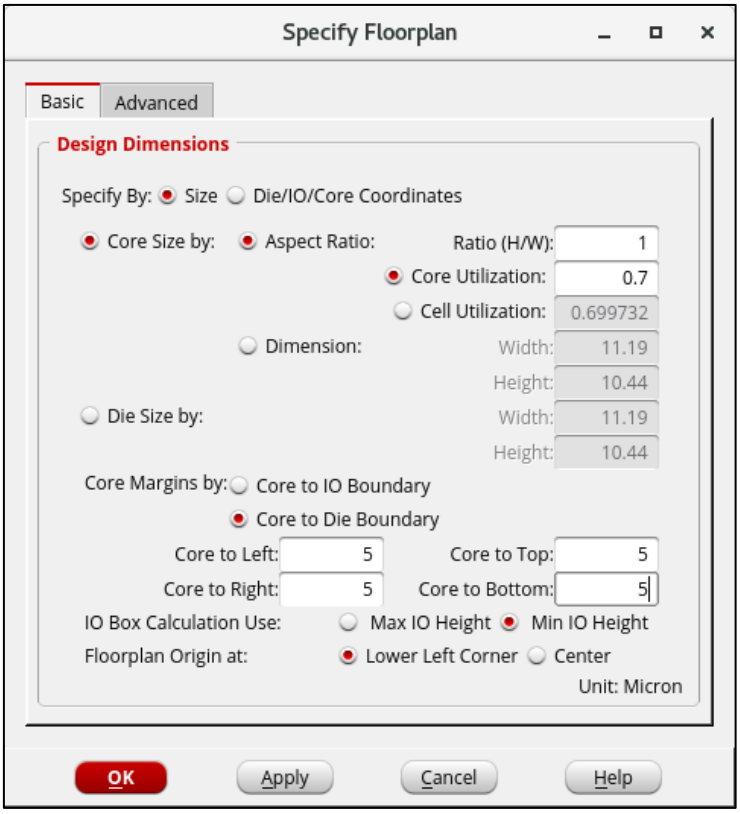


Fig. 111: Specify floorplan window

- The Yellow patch on the Left Bottom are the group of “Unassigned pins” which are to be placed along the IO Boundary along with the Standard Cells [Gates]

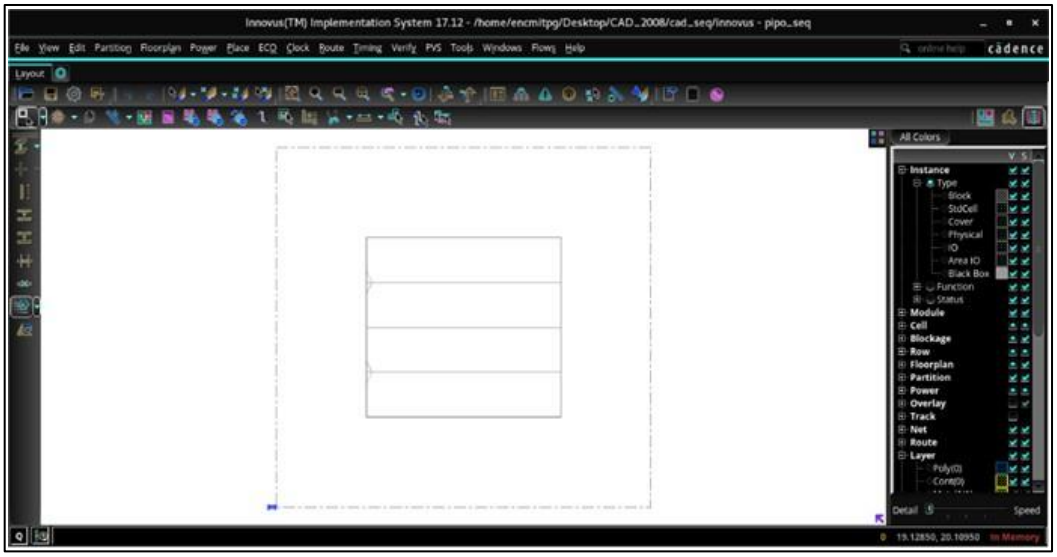


Fig. 112: Innovus window after floorplan

Power Planning

Steps under Power Planning :

1. Connect Global Net Connects
2. Adding Power Rings
3. Adding Power Strings
4. Special Route

Under Connect Global Net Connects, we create two pins, one for VDD and one for VSS connecting them to corresponding Global Nets as mentioned in Globals file / Power and Ground Nets.

- Select Power → Connect Global Nets.. to create “Pin” and “Connect to Global Net” as shown and use “Add to list”.
- Click on “Apply” to direct the tool in enforcing the Pins and Net connects to Design and then Close the window.

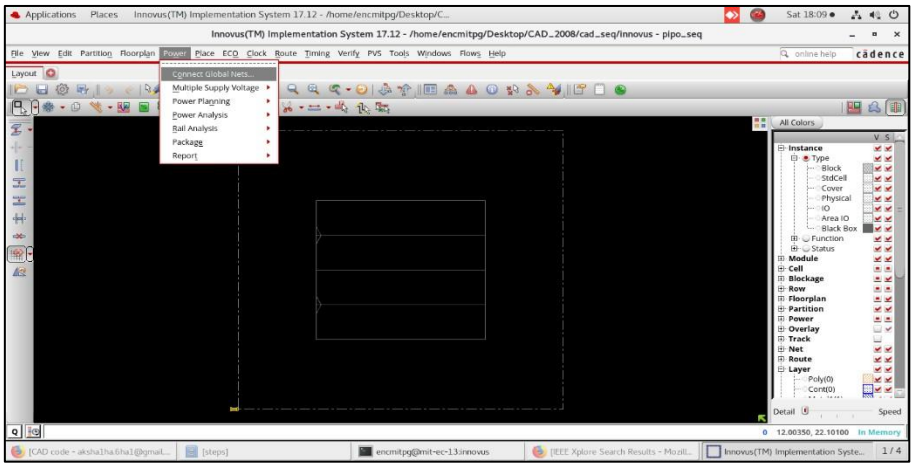


Fig. 113: Click on power -> connect global nets

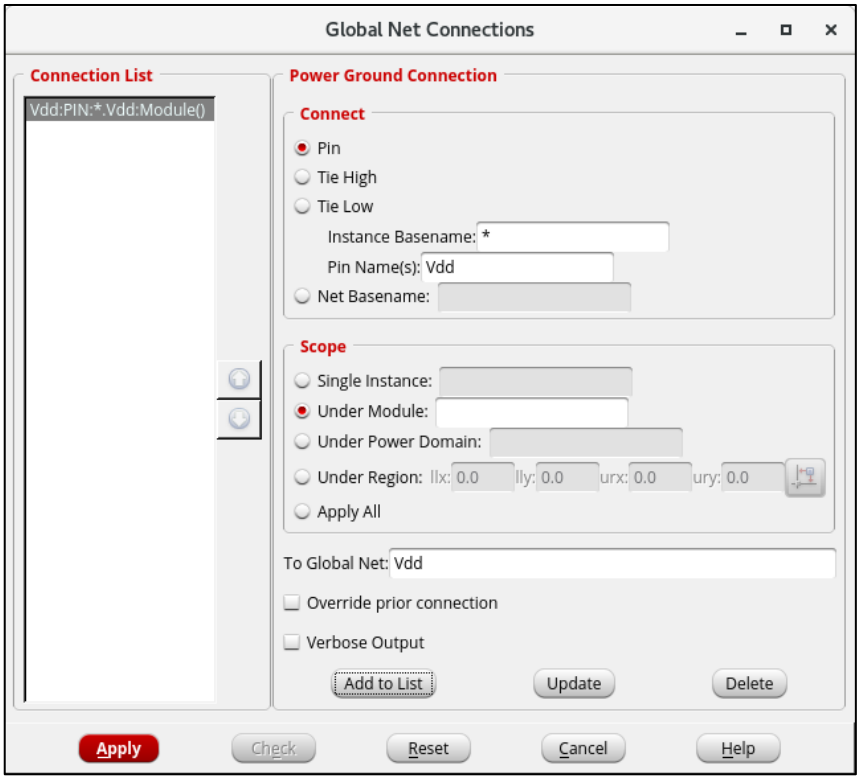


Fig. 114: Global net connection window adding Vdd power net

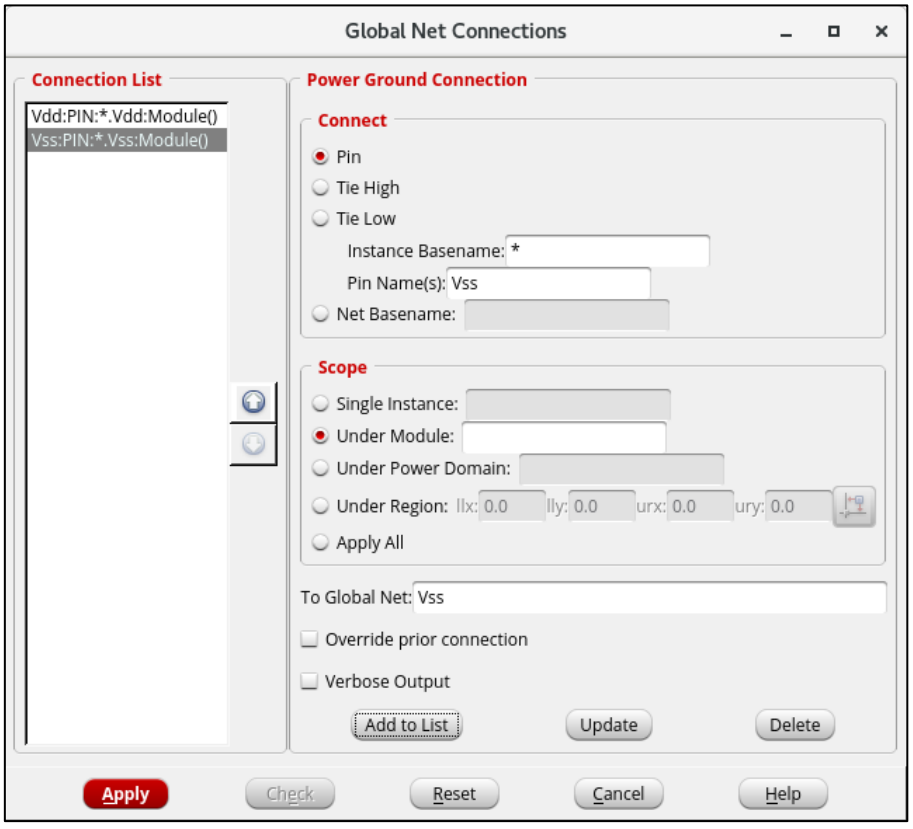


Fig. 115: Global net connection window adding Vss power net

In order to Tap in Power from a distant Power supply, Wider Nets and Parallel connections improve efficiency. Moreover, the cells that would be placed inside the core area are expected to have shorter Nets for lower resistance.

- Hence Power Rings [Around Core Boundary] and Power Stripes [Across Core Boundary] are added which satisfies the above conditions.
- Select Power → Power Planning → Add Rings to add Power rings 'around Core Boundary'

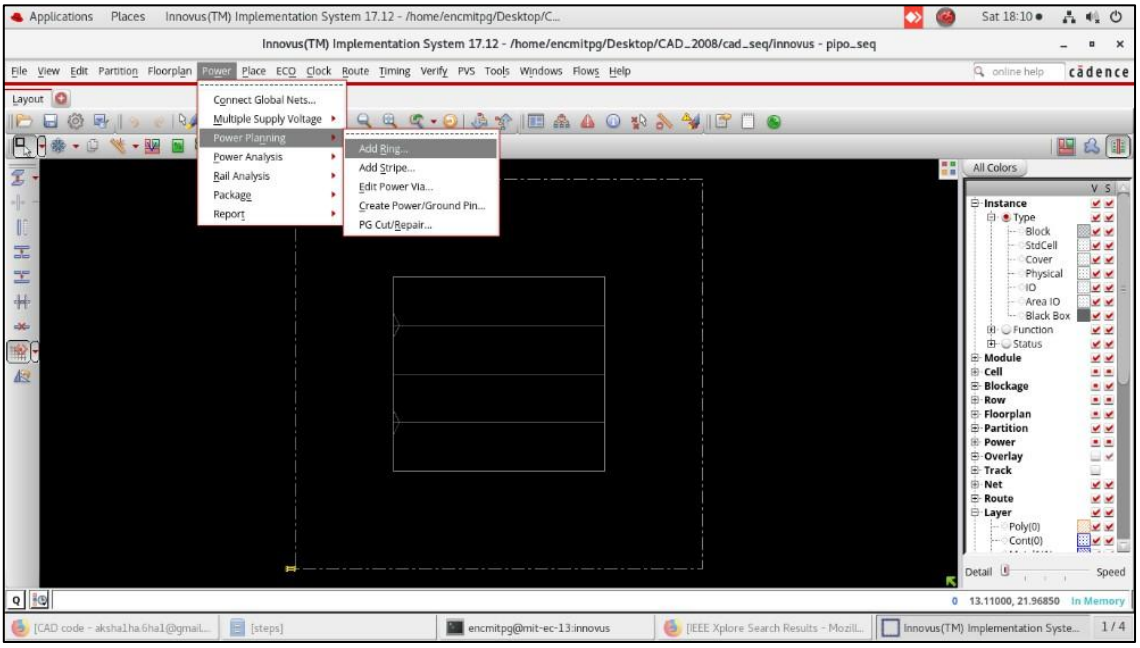


Fig. 116: Click on power planning -> Add ring

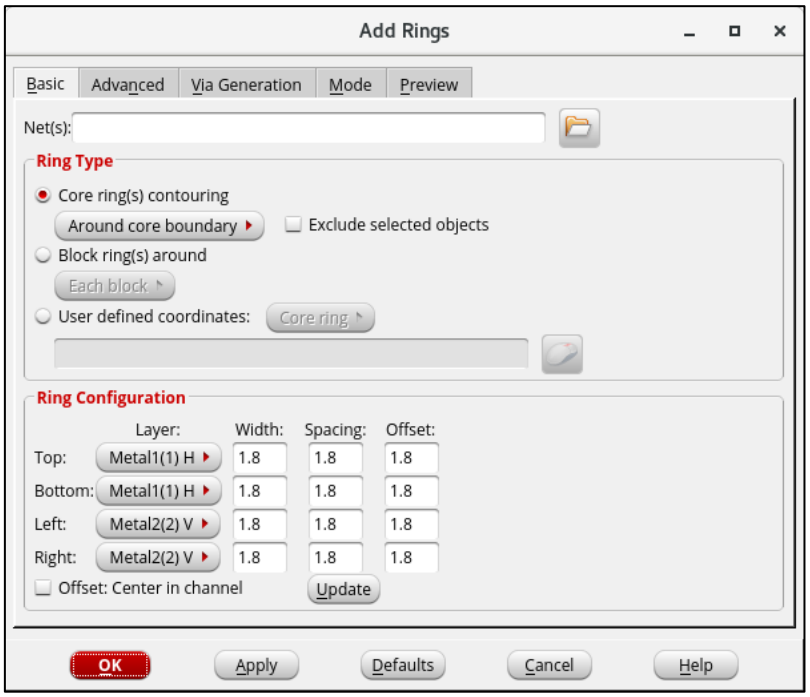


Fig. 117: Add Rings window

- Select the Nets from Browse option OR Directly type in the Global Net Names separated by a space being Case and Spelling Sensitive.
- Select the Highest Metals marked ‘H’ [Horizontal] for Top and Bottom and Metals marked ‘V’ [Vertical] for Right and Bottom. This is because Highest metals have Highest Widths and thus Lowest Resistance.
- Click on Update after the selection and “Set Offset : Centre in Channel” in order to get the Minimum Width and Minimum Spacing of the corresponding Metals and then Click “OK”.
- Similarly, Power Stripes are added using similar content to that of Power Rings.

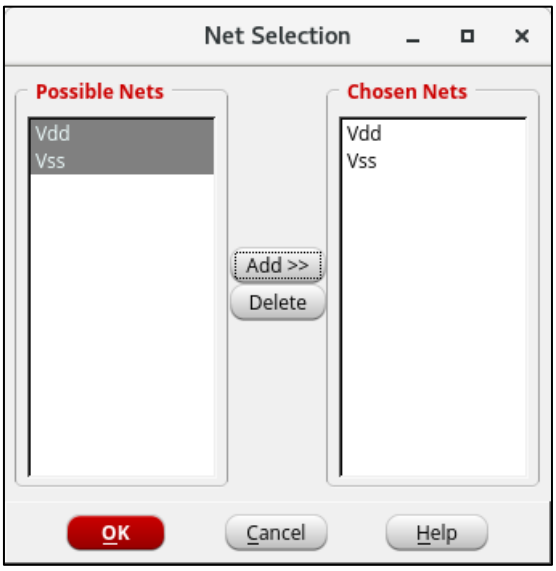


Fig. 118: Net selection window

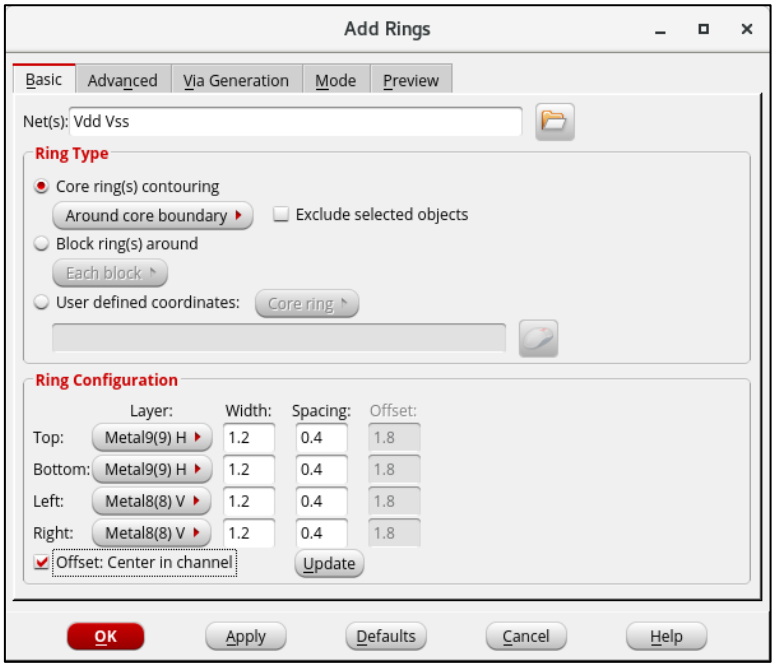


Fig. 119: Add rings window after selecting nets and choosing width

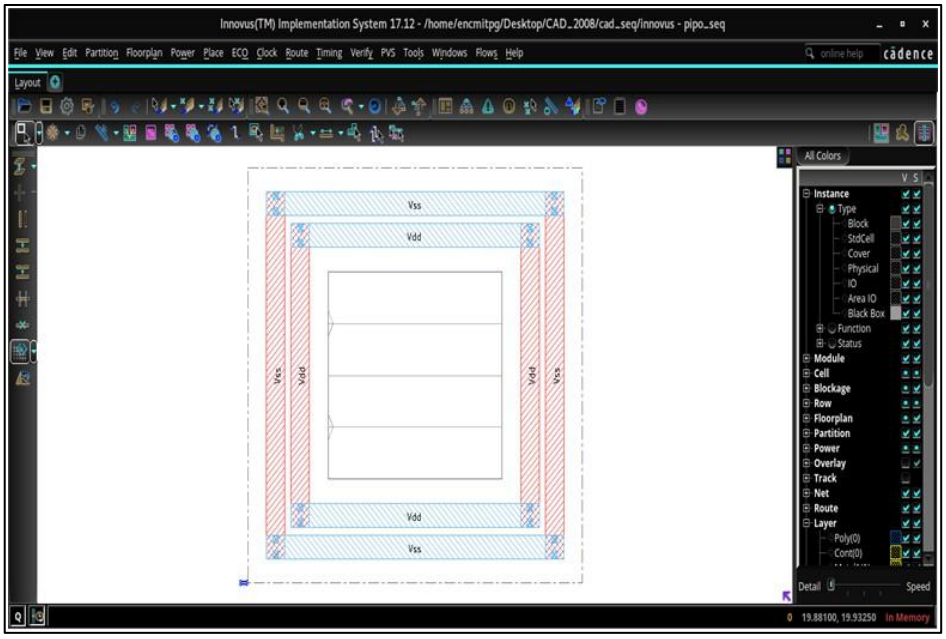


Fig. 120: Innovus window after adding rings

Factors to be considered under Power Stripes :

1. Nets
2. Metal and It's Direction
3. Width and Spacing [Updated]
4. Set to Set Distance = (Minimum Width of Metal + Min. Spacing) x 2

- On adding Power Stripes, The Power mesh setup is complete as shown. However, There are no Vias that could connect Metal 9 or Metal 8 directly with Metal 1 [VDD or VSS of Standard Cells are generally made up of Metal 1].
- The connection between the Highest and Lowest Metals is done through Stacking of Vias done using "Special Route".
- To perform Special Route, Select Route → Special Route → Add Nets → OK.
- After the Special Route is complete, all the Standard Cell Rows turn to the Color coded for Metal 1 as shown below.
- The complete Power Planning process makes sure Every Standard Cell receives enough power to operate smoothly.

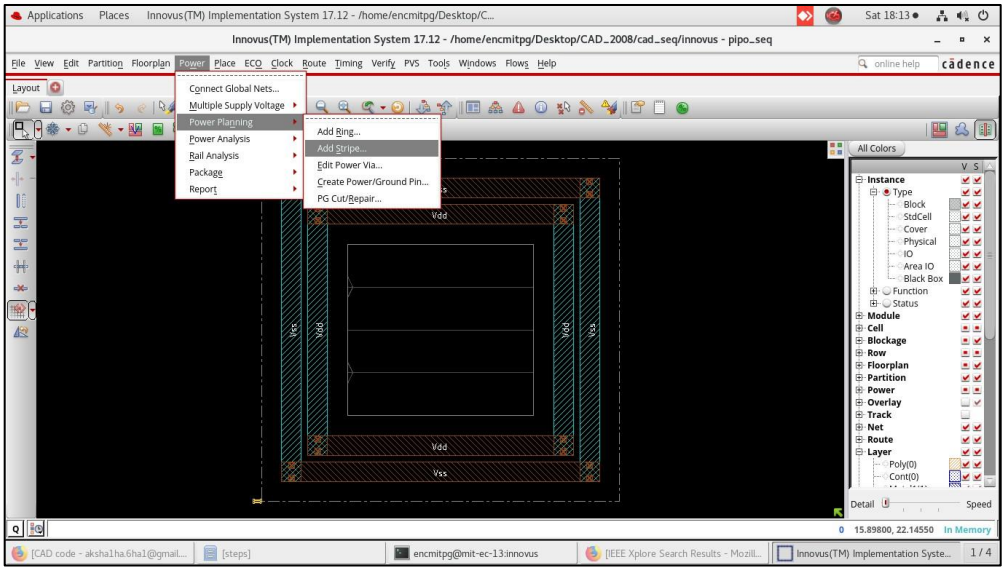


Fig. 121: Click on power -> power planning -> Add stripes

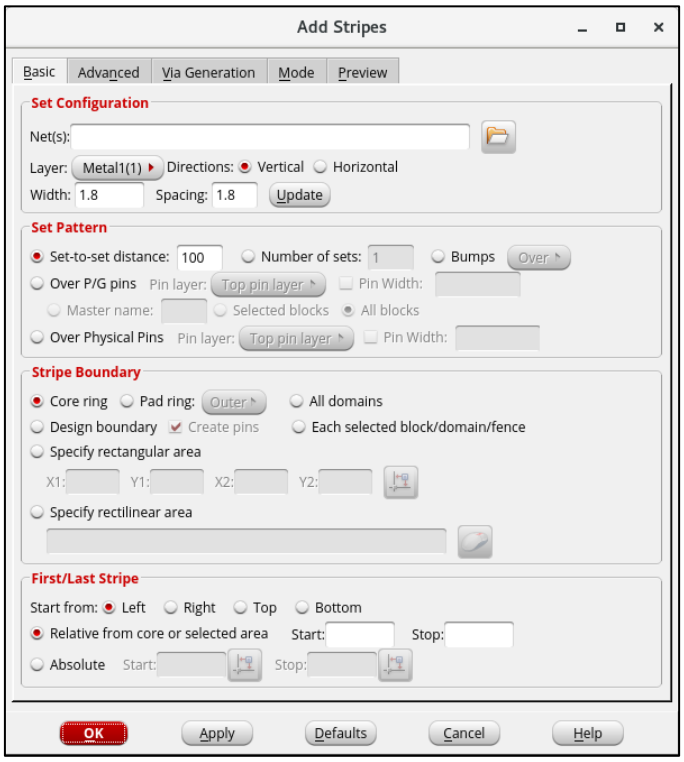


Fig. 122: Add stripes window

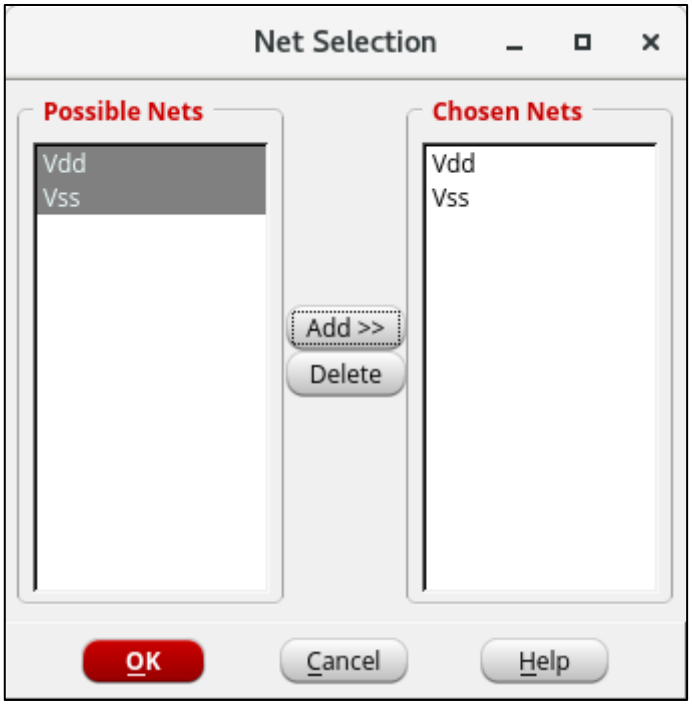


Fig. 123: Select nets

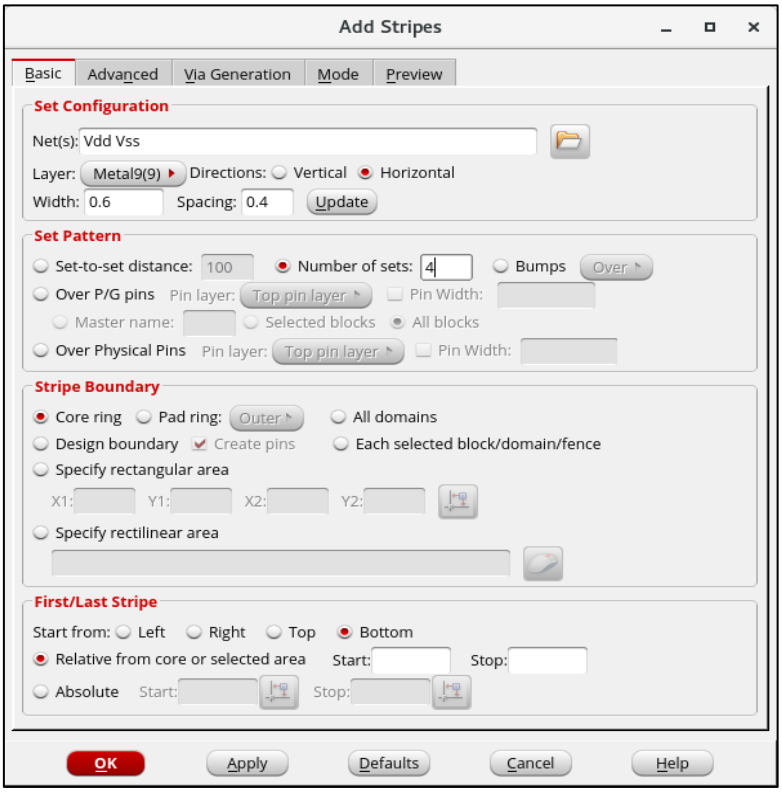


Fig. 124: select layer as metal 9 and directions as horizontal

Then click ‘OK’. Repeat again same steps for metal 8.

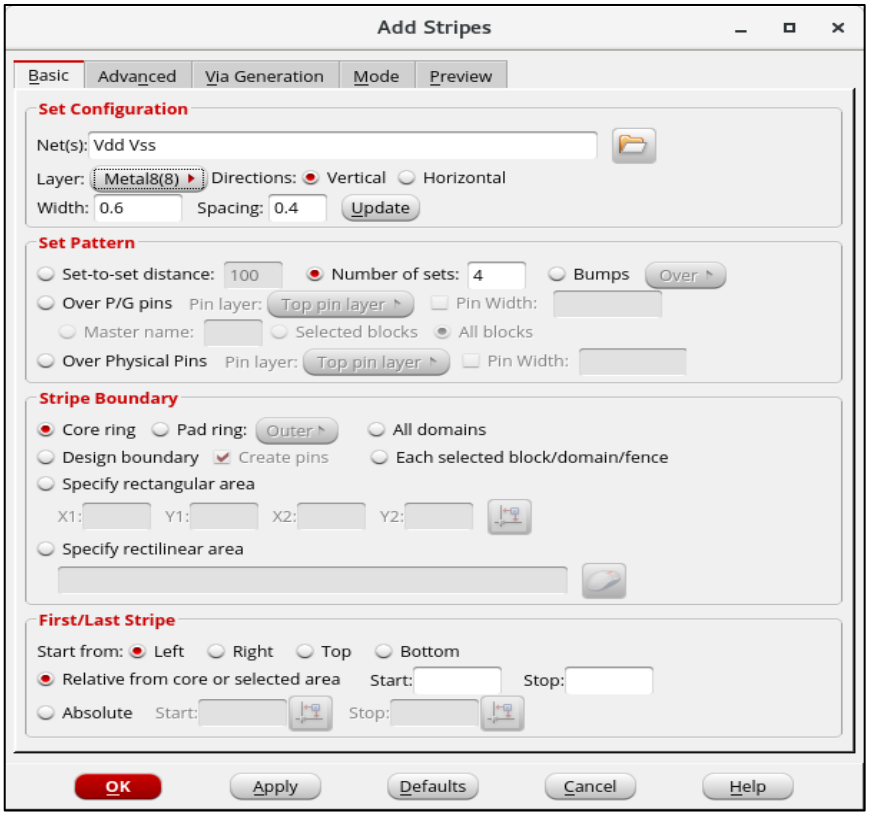


Fig. 125: select layer as metal 8 and direction as vertical

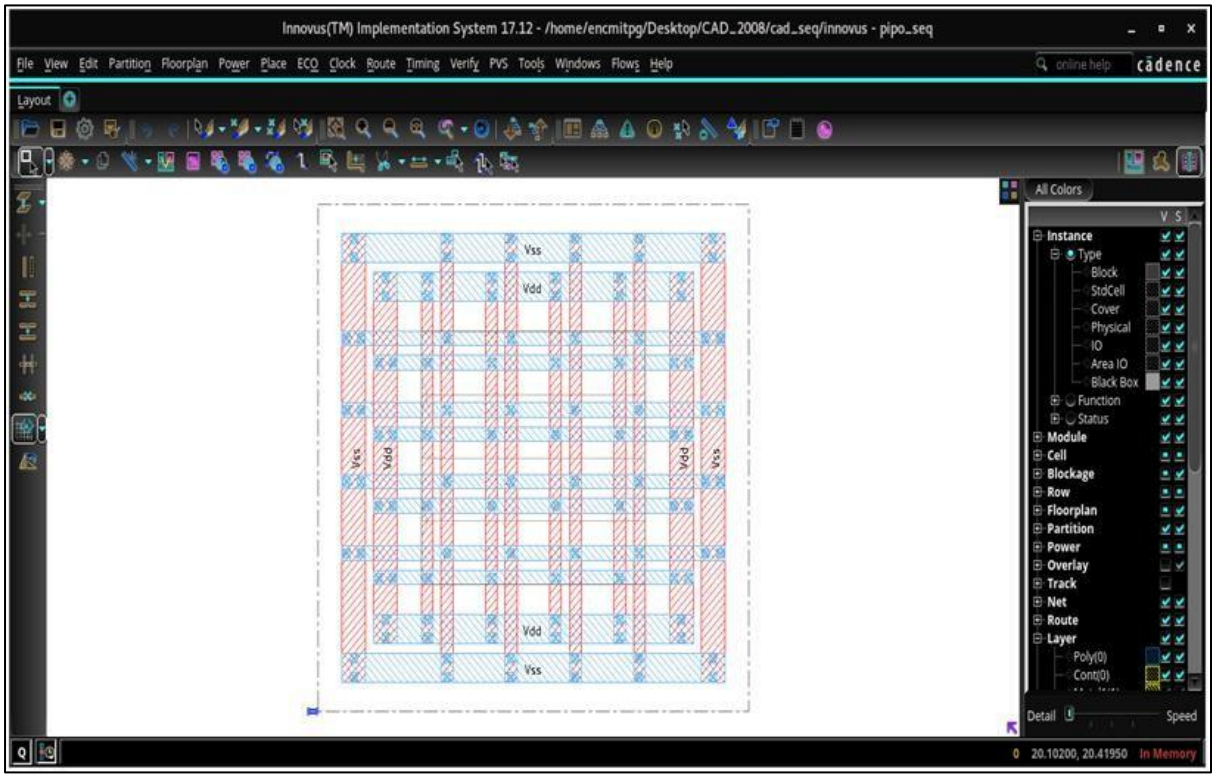


Fig. 126: Window after power planning

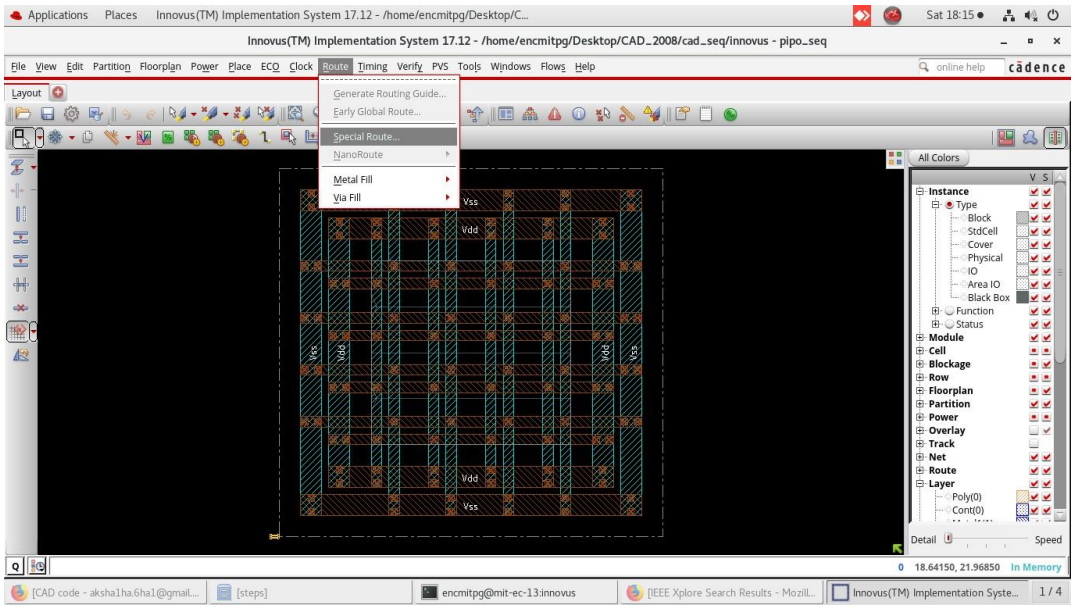


Fig. 127: Route -> special Route

Fig. 128: SRoute window

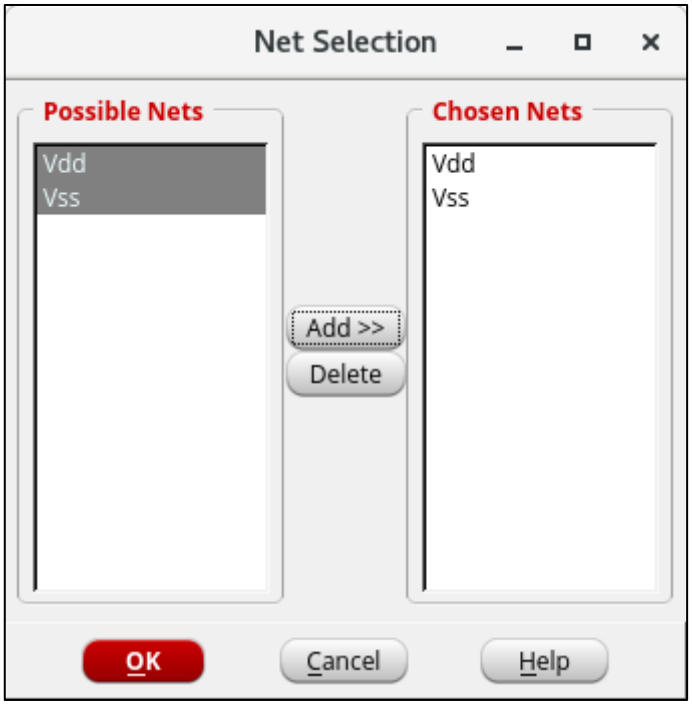


Fig. 129: Add both the nets

Fig. 130: click ‘ok’

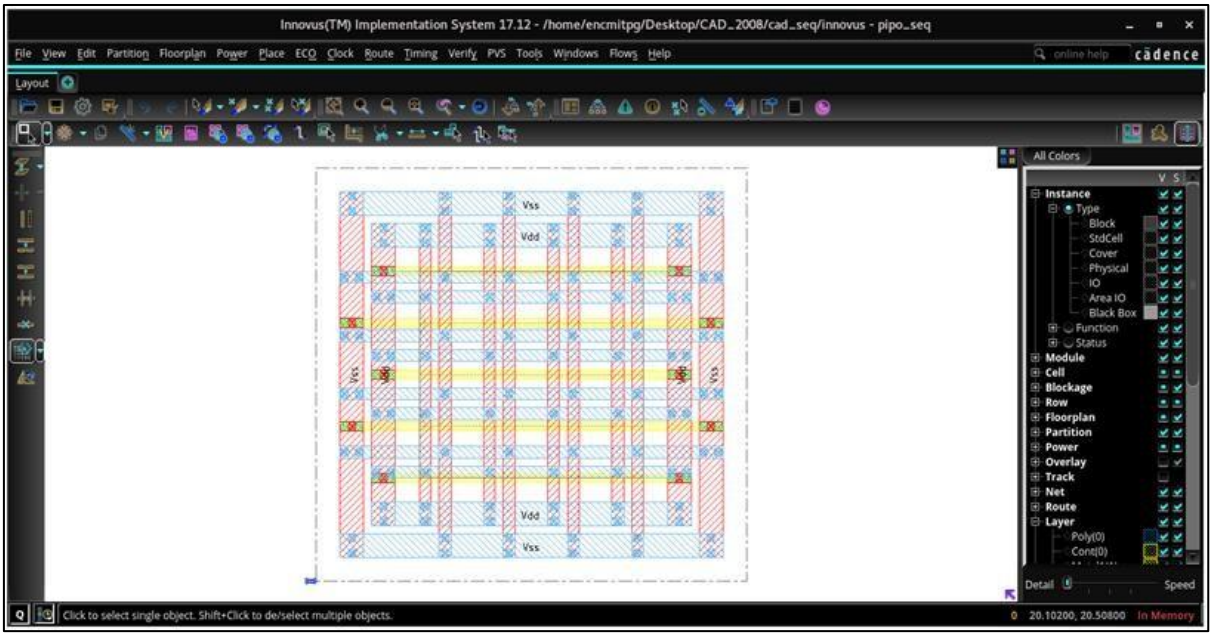


Fig. 131: Pre-placement module

After Power Planning, a few Physical Cells are added namely, End Caps and Well Taps.

1. End Caps : They are Physical Cells which are added to the Left and Right Core Boundaries acting as blockages to avoid Standard Cells from moving out of boundary.
2. Well Taps : They act like Shunt Resistance to avoid Latch Up effects.

- To add End Caps, Select Place → Physical Cell → Add End Caps and “Select” the FILL’s from the available list.
- Higher Fills have Higher Widths. As shown Below, The End Caps are added below your Power Mesh.

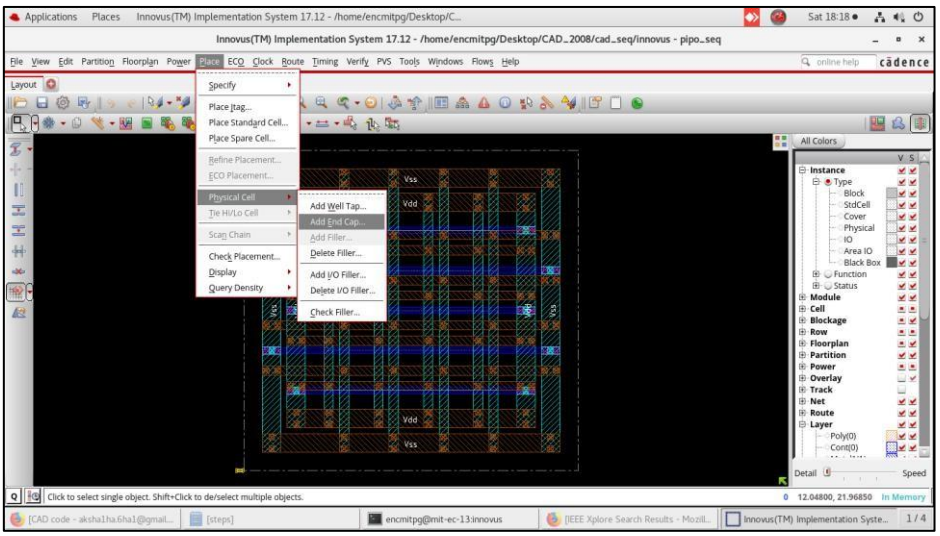


Fig. 131: click on place -> physical cell -> add endcap

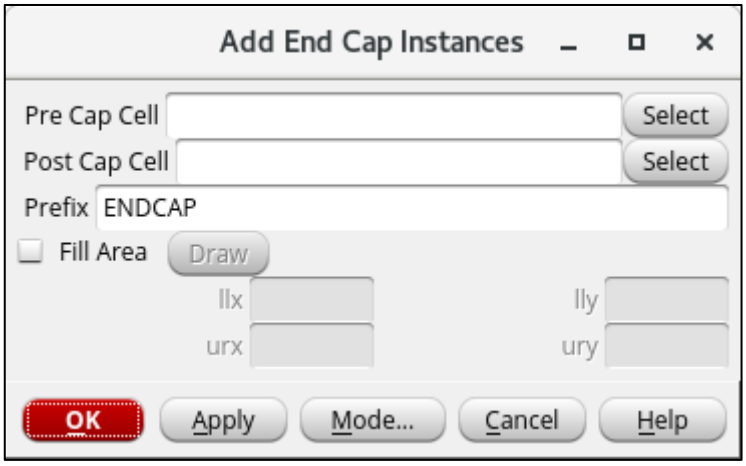


Fig. 132: endcap instances window, click select to add fillers for precap cell

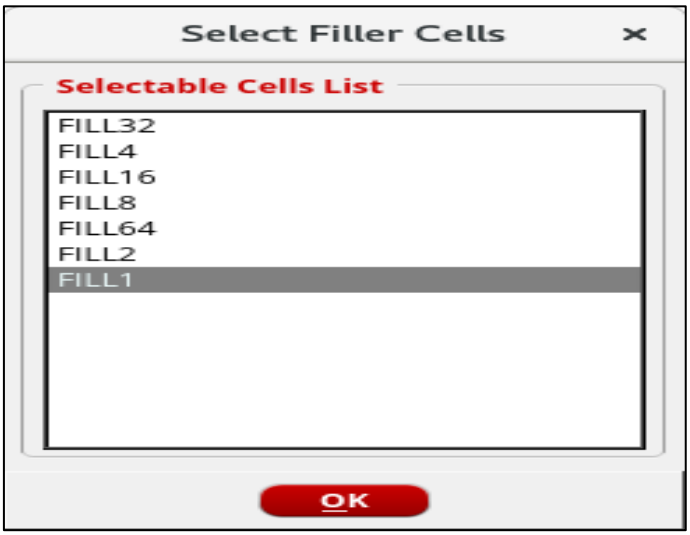


Fig. 133: select Fill1 and give ‘ok’

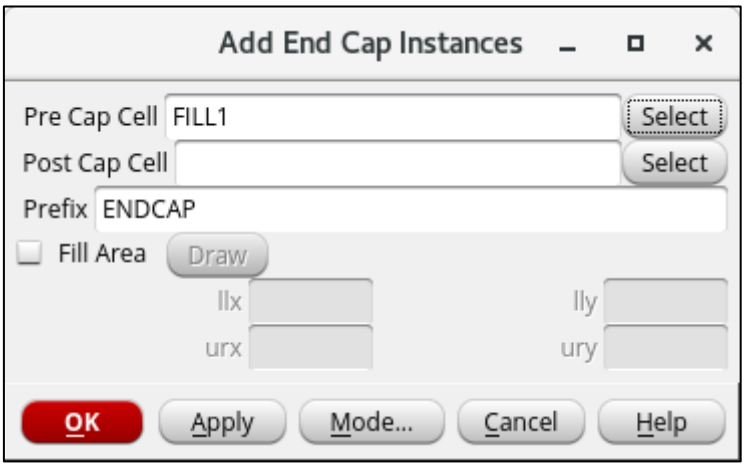


Fig. 134: Click select to add fillers for post cap cell

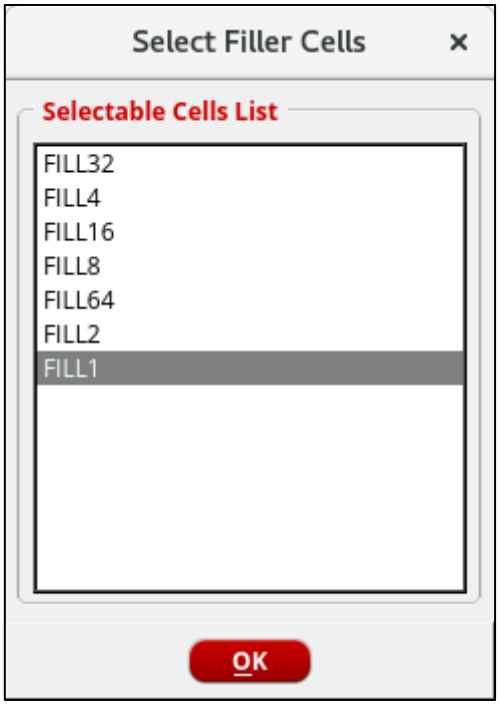


Fig. 135: select fill1

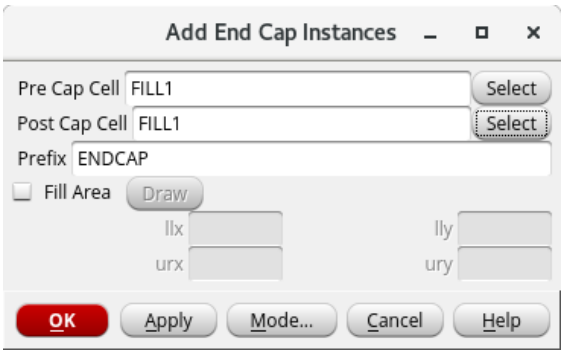


Fig. 136: click apply and then ‘ok’

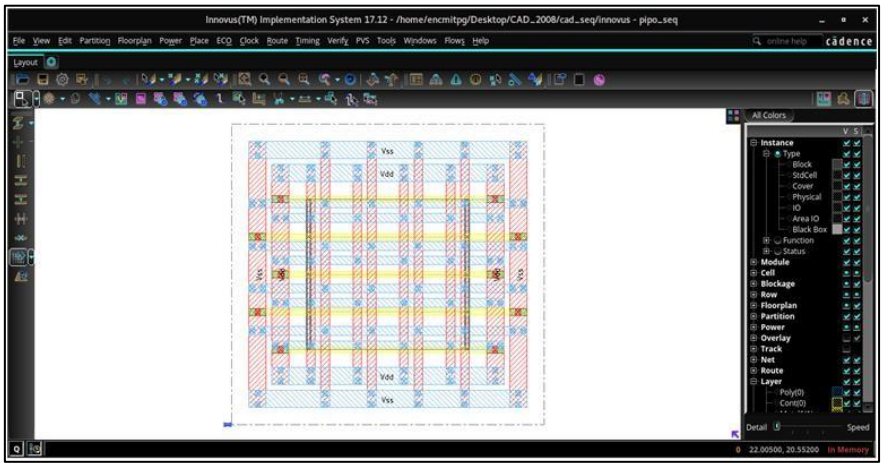


Fig. 137: Innovus window after adding end caps

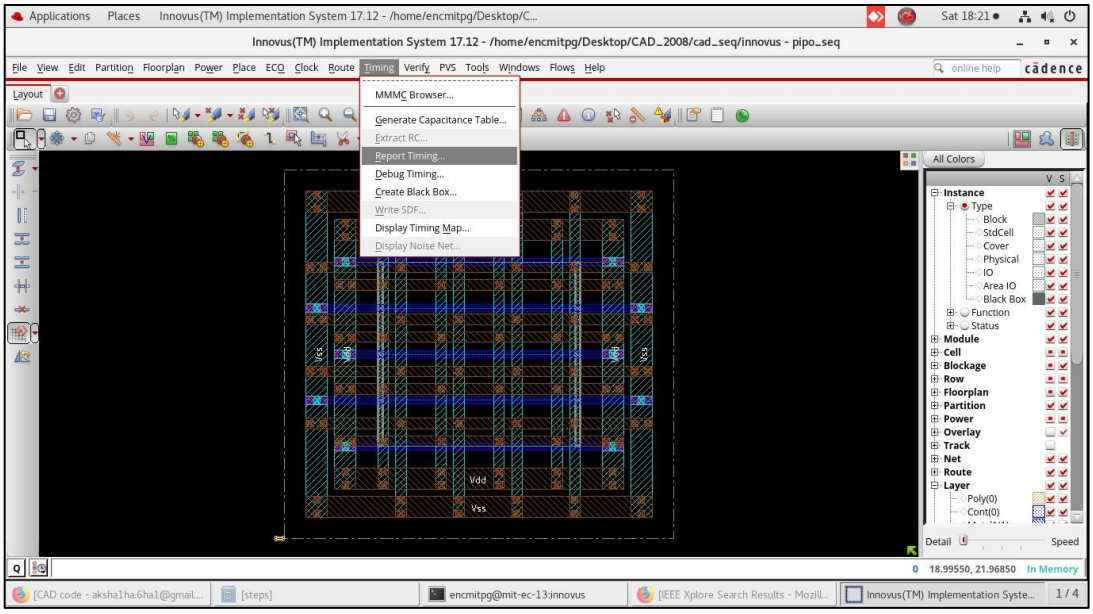


Fig. 138: Click on timing -> report timing

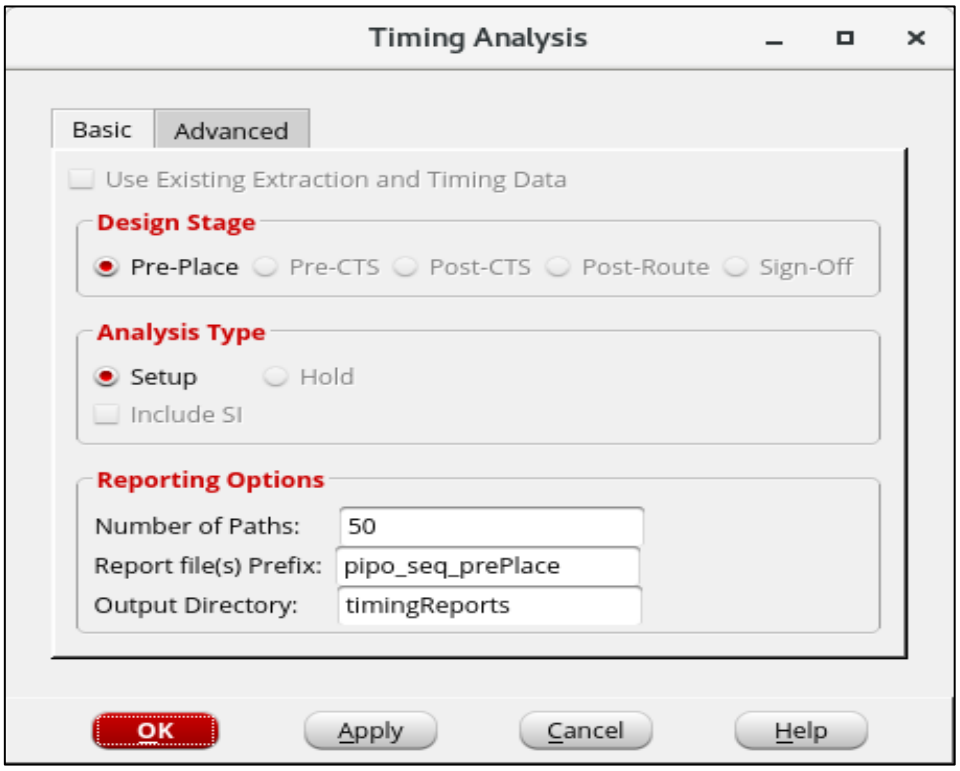


Fig. 139: Select design stage as Pre-place and analysis type as setup

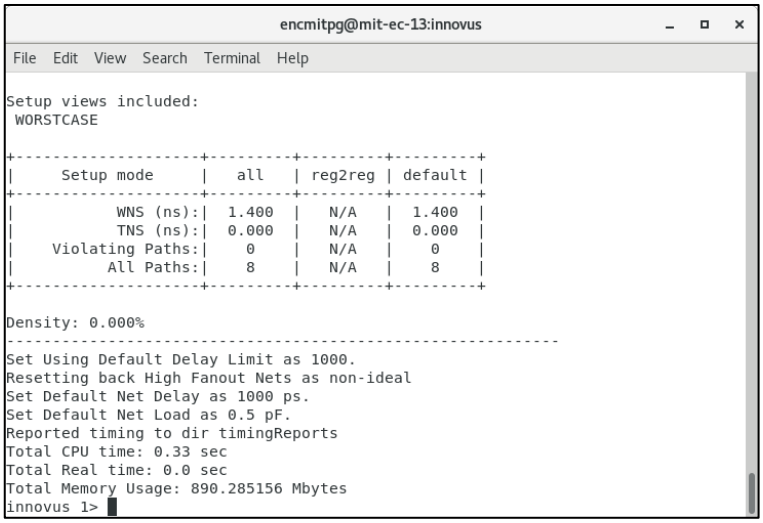


Fig. 140: Pre-placement timing report (terminal window)

- The Placement stage deals with Placing of Standard Cells as well as Pins.
- Select Place → Place Standard Cell → Run Full Placement → Mode → Enable ‘Place I/O Pins’ → OK → OK .

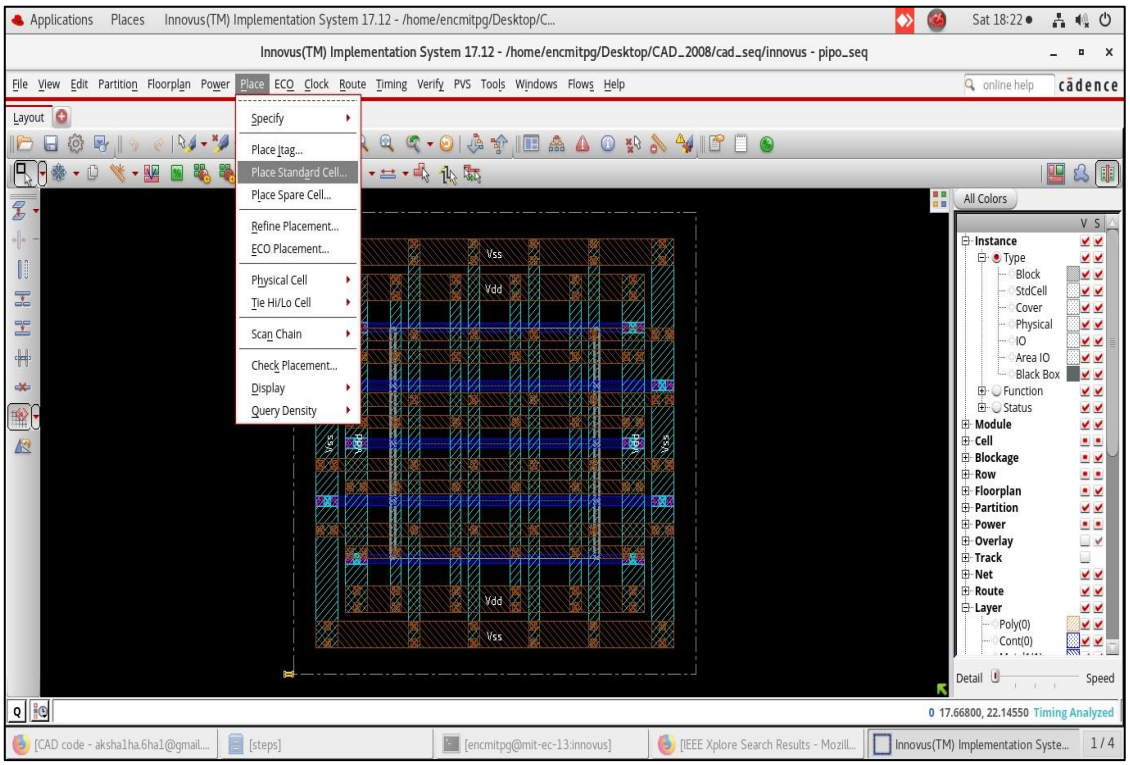


Fig. 141: Place -> Place standard cell

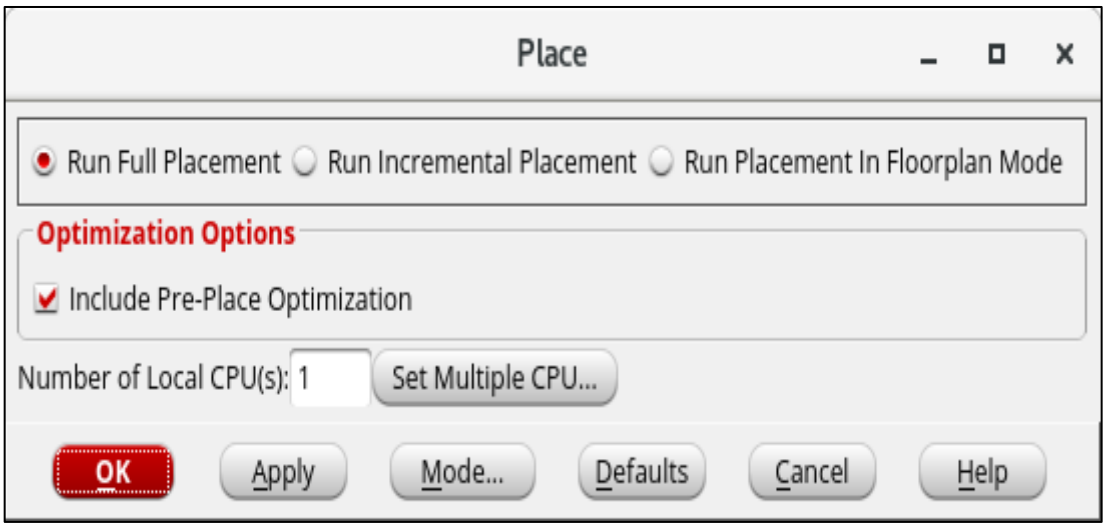


Fig. 142: Click on mode

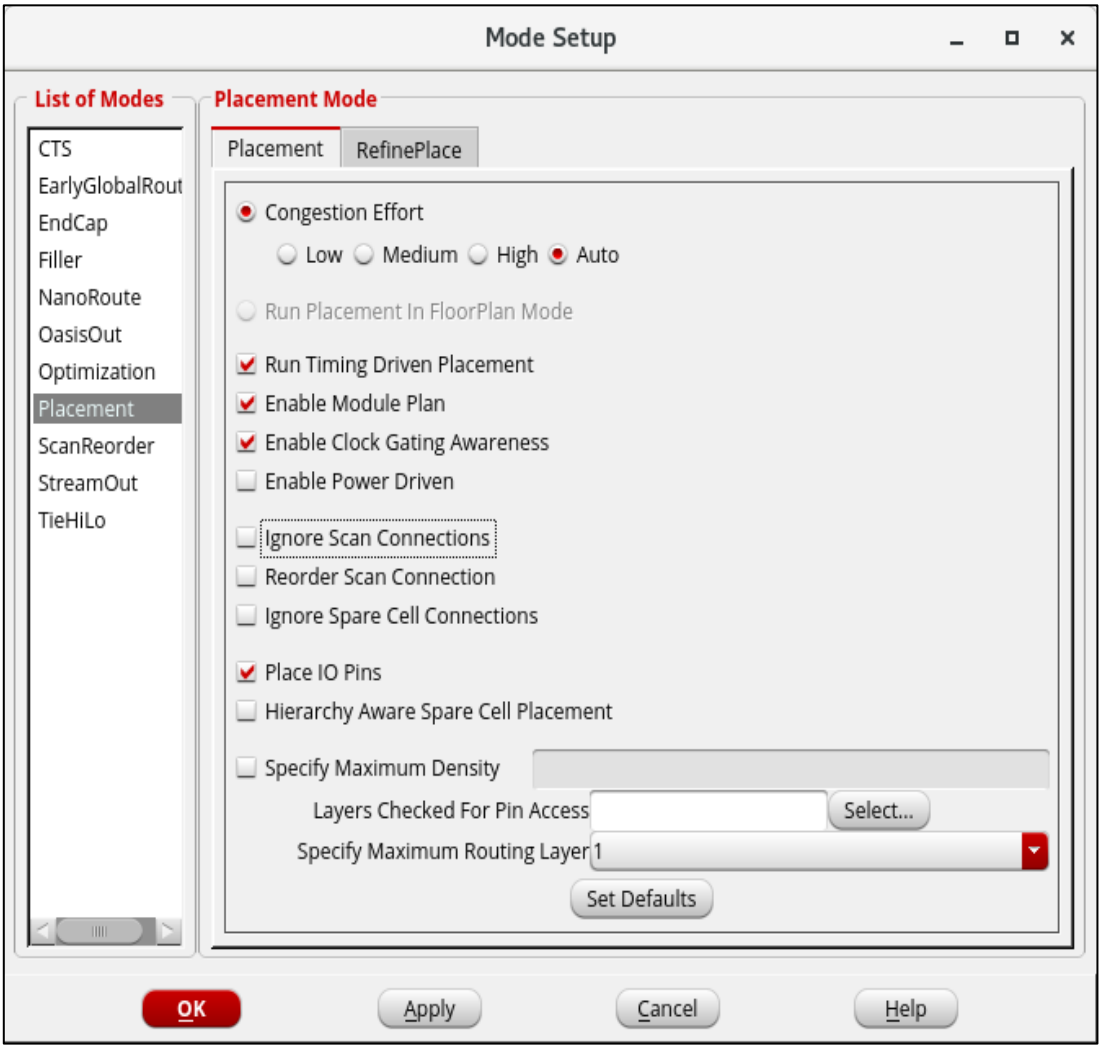


Fig. 143: select place IO pins

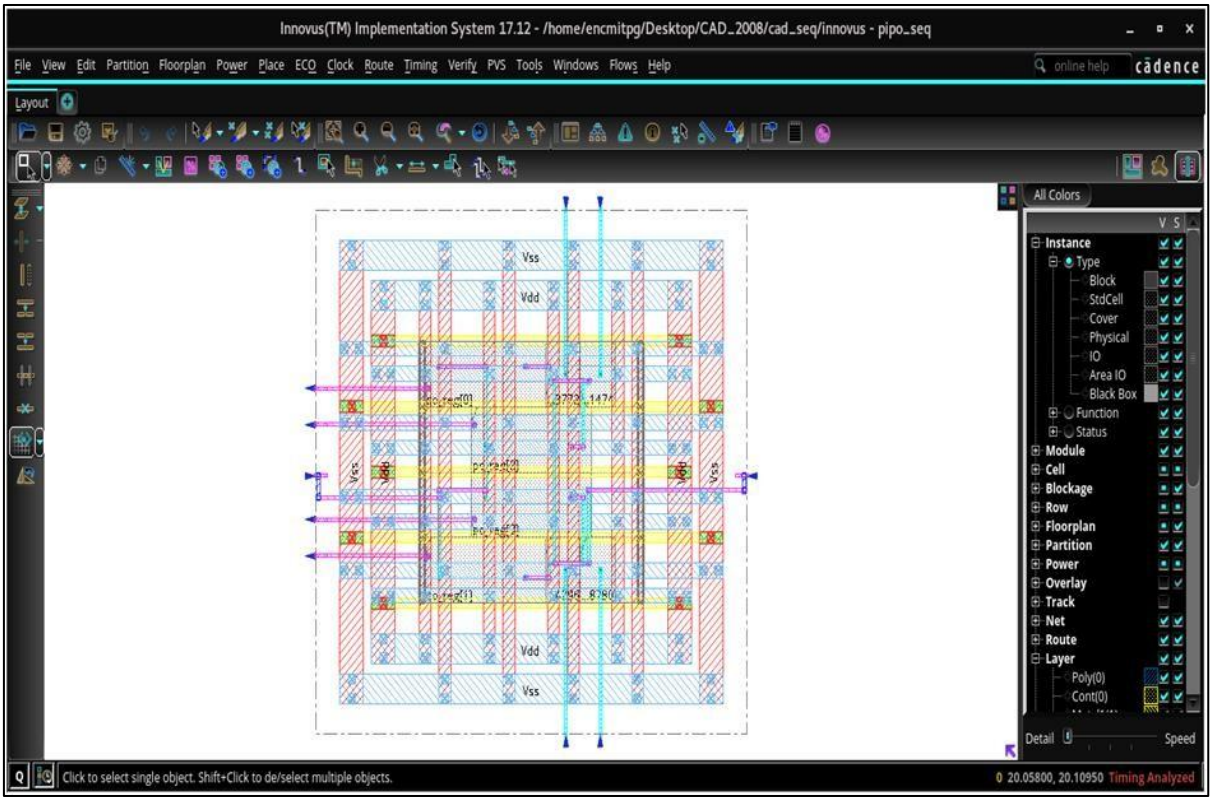


Fig. 144: Innovus window after placement

- All the Standard Cells and Pins are placed as per the communication between them, i.e., Two communicating Cells are placed as close as possible so that shorter Net lengths can be used for connections as Shorter Net Lengths enable Better Timing Results.

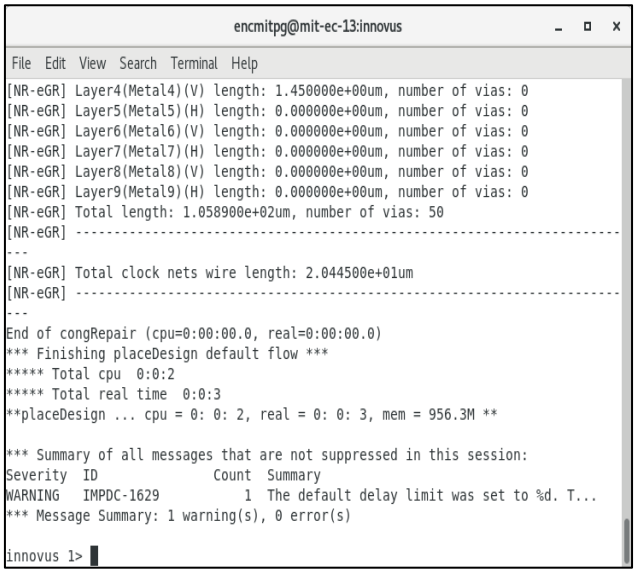


Fig. 145: Terminal window after placement

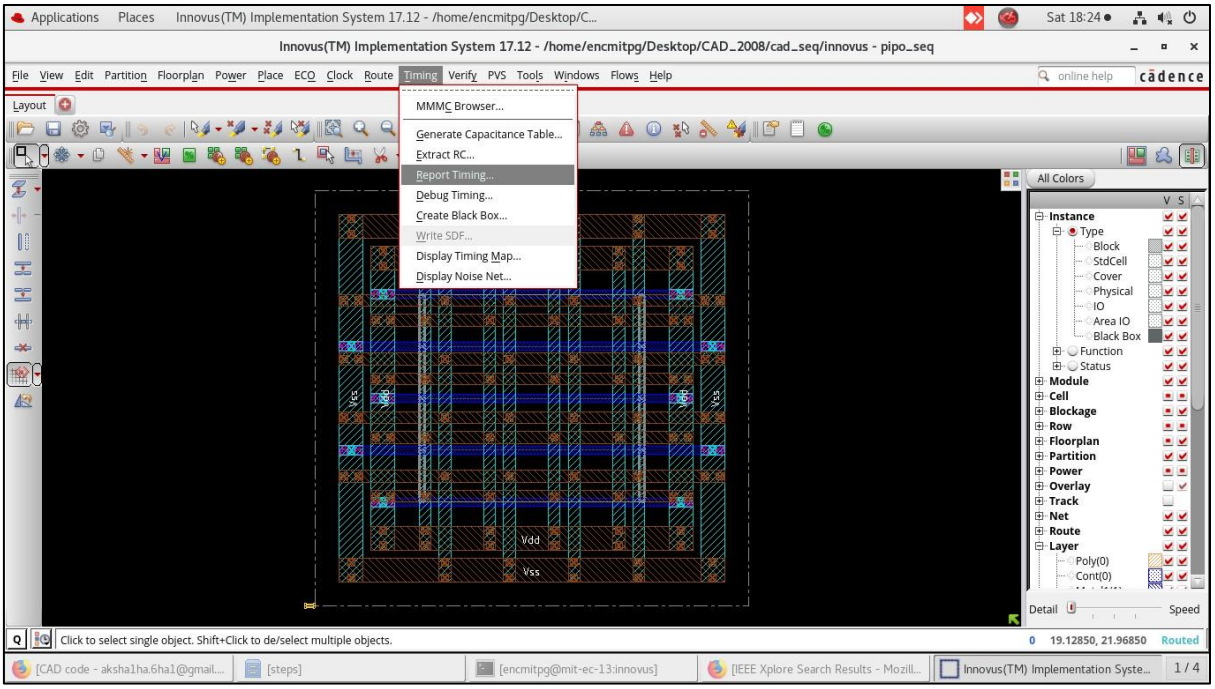


Fig. 146: click on timing -> report timing

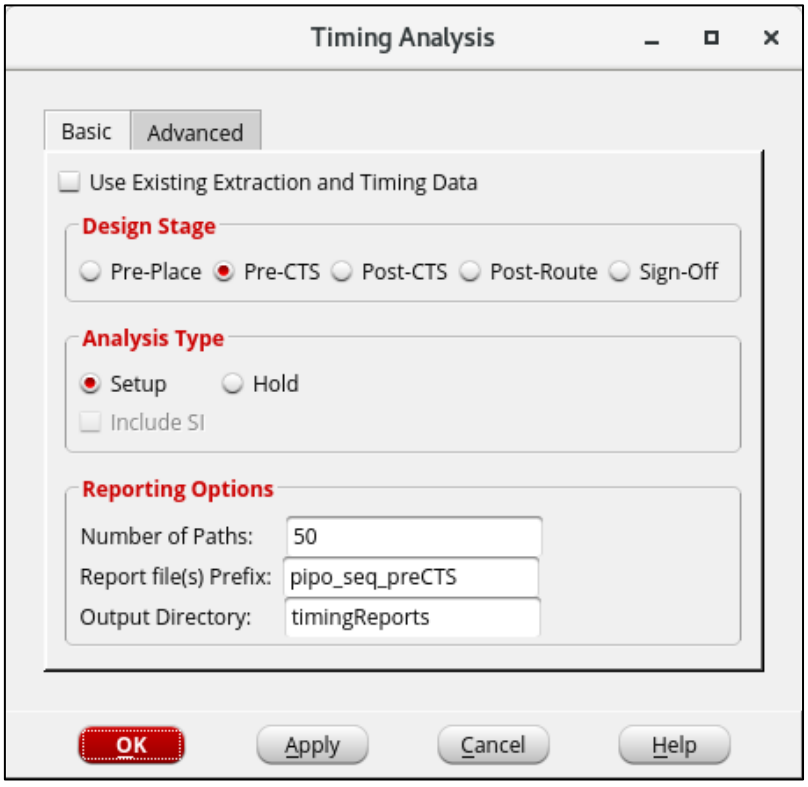


Fig. 147: Select design stage as Pre-CTS and analysis type as Setup

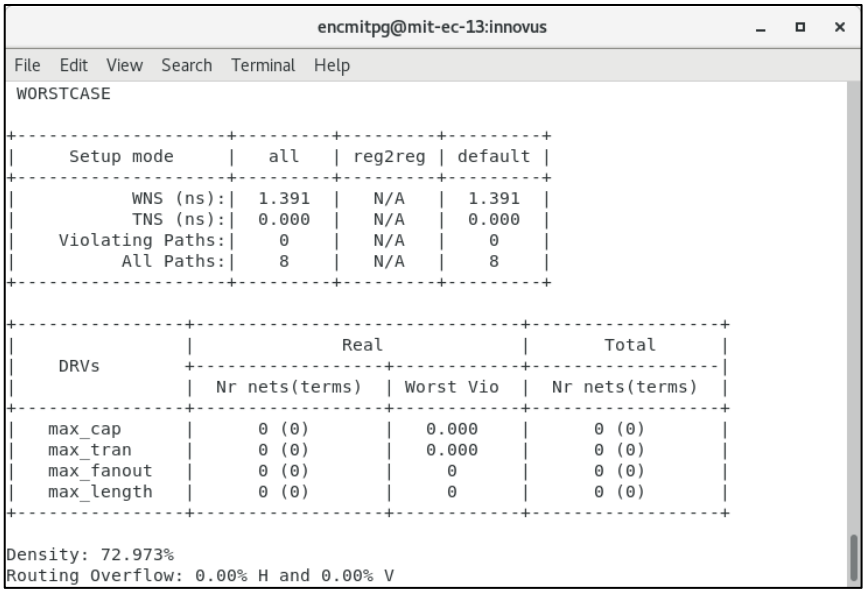


Fig. 148: Timing report after placement

- The CTS Stage is meant to build a Clock Distribution Network such that every Register (Flip Flop) acquires Clock at the same time (Atleast Approximately) to keep them in proper communication.

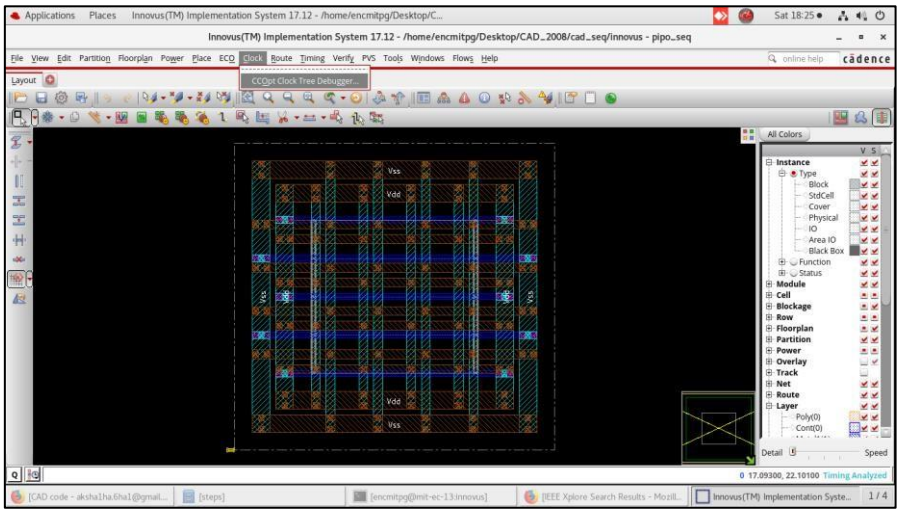


Fig. 149: click on clock -> CCopt clock tree debugger

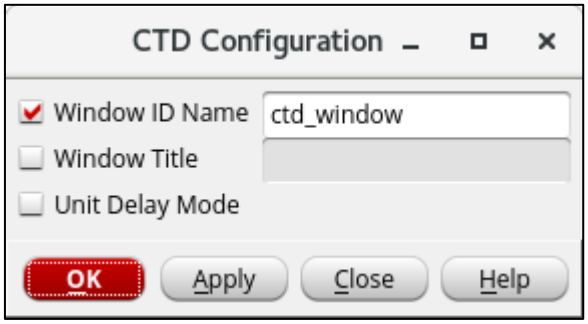


Fig. 150: CTD config window

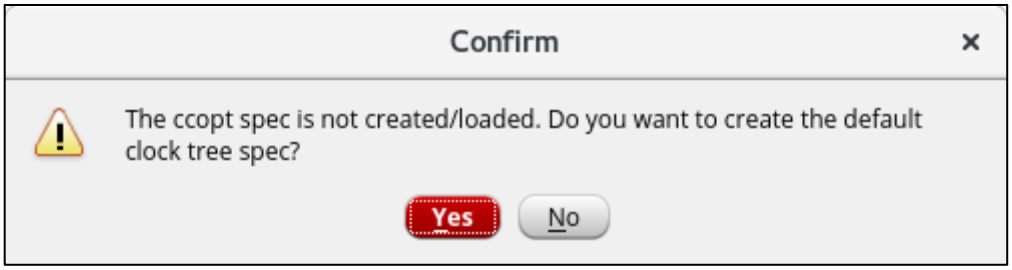


Fig. 151: Click on yes

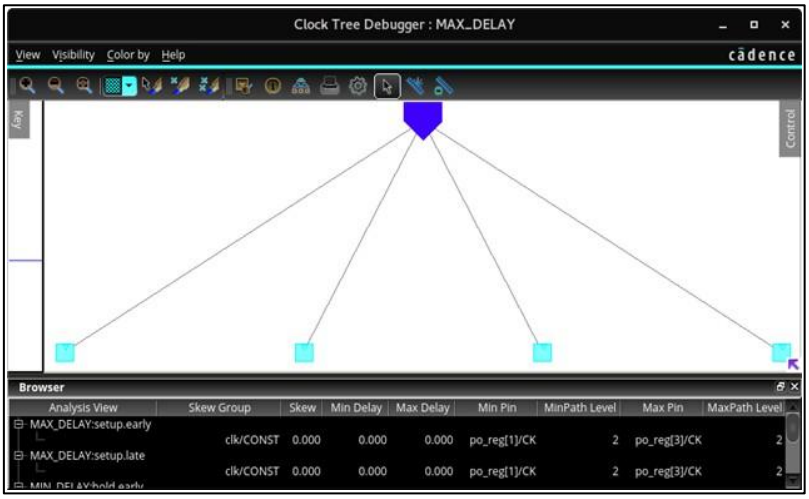


Fig. 152: Clock tree Debugger window

- The Red Boxes are the Clock Pins of various Flip Flops in the Design while Yellow Pentagon on the top represents Clock Source.
- The Clock Tree is built with Clock Buffers and Clock Inverters added to boost up the Clock Signal.

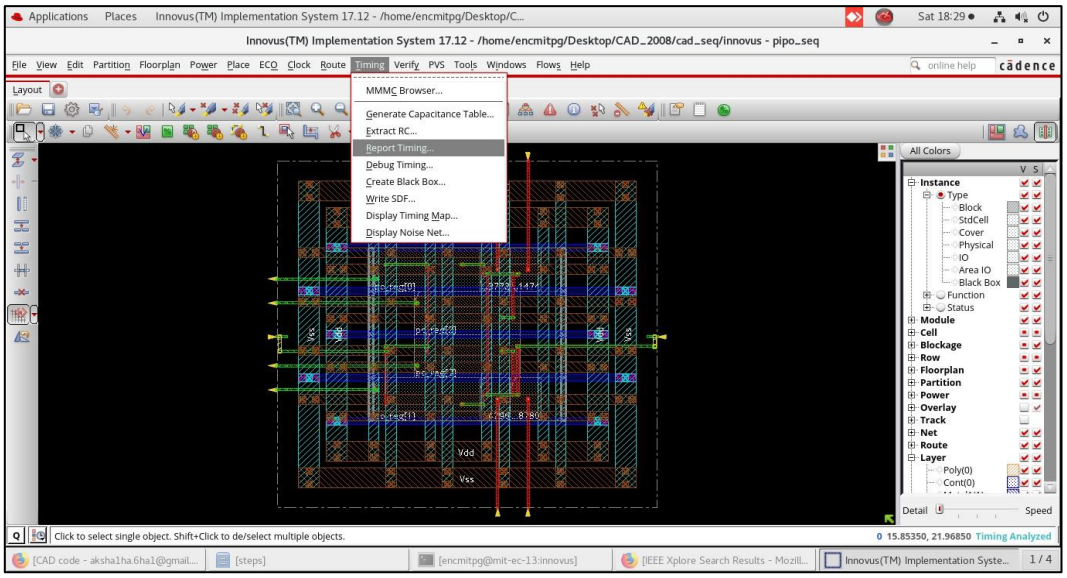


Fig. 153: Click on timing -> report timing

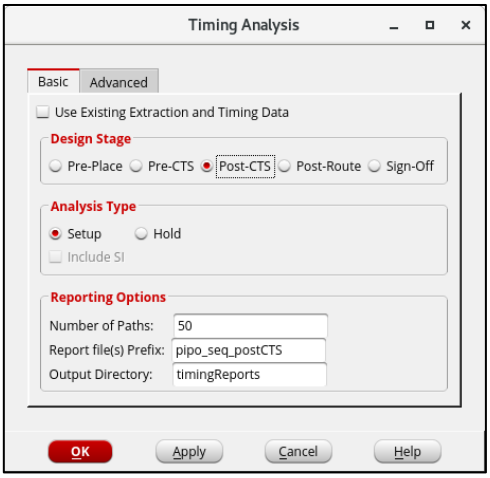


Fig. 154: setup timing analysis

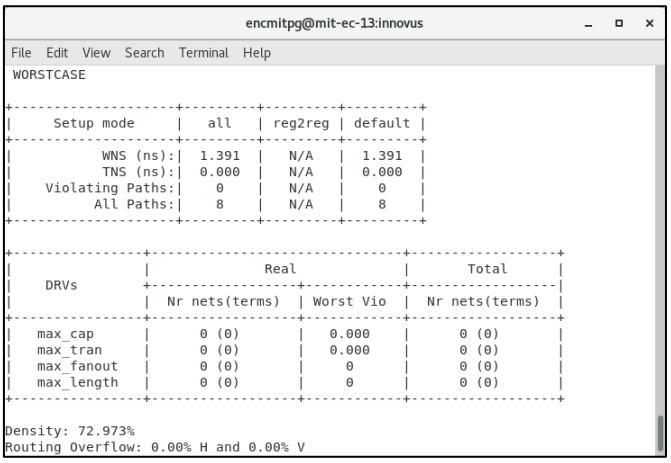


Fig. 155: setup timing report (terminal window)

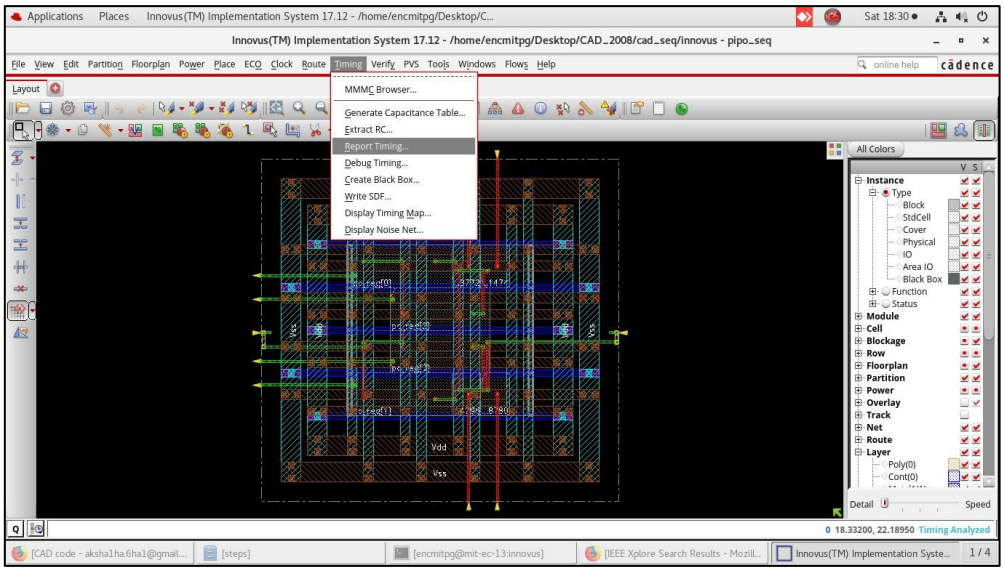


Fig. 156: click on timing -> report timing

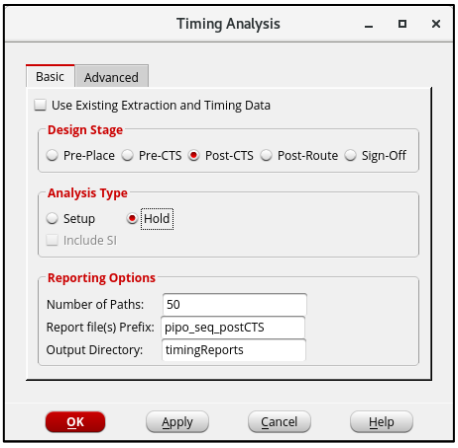


Fig. 157: Hold timing analysis

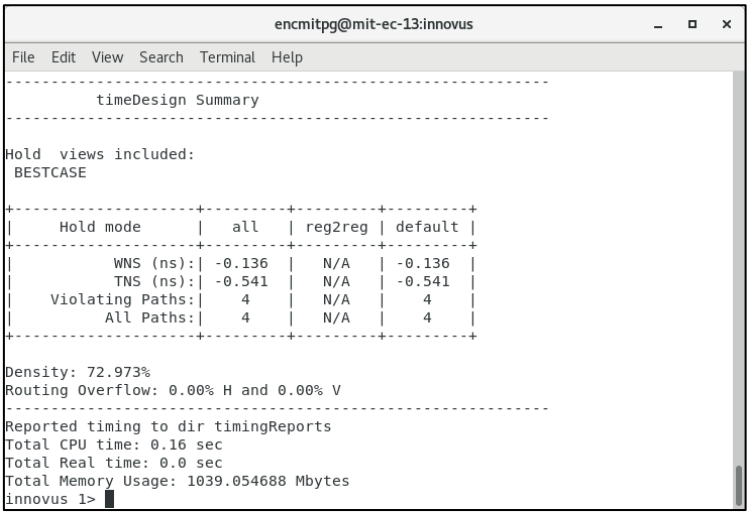


Fig. 158: hold analysis timing report with violations (terminal window)

In case of any Violating paths, the design could be optimized in the following way.

- To optimize the Design, Select ECO → Optimize Design → Design Stage [PreCTS] → Optimization Type – Setup → OK
- After you run the optimization, the terminal displays the latest Timing report and updated area and power reports can be checked.
- This step Optimizes your design in terms of Timing, Area and Power. You can Generate Timing, Area, Power in similar way as above report Post – Optimization to compare the Reports

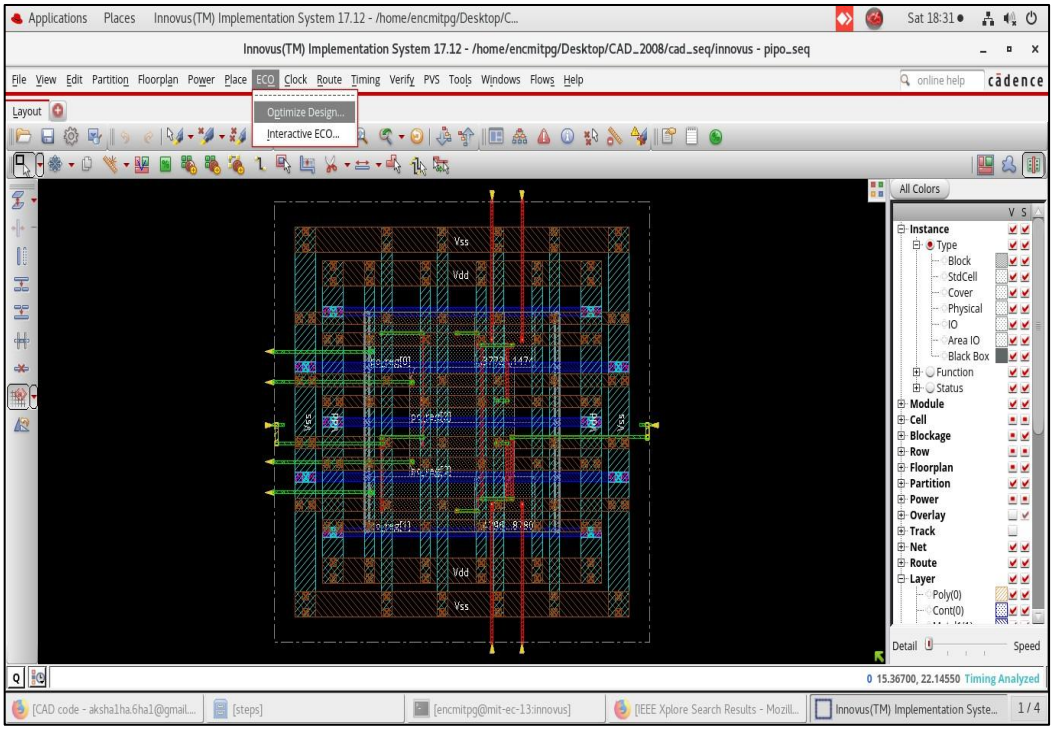


Fig. 159: Click on ECO -> Optimize design

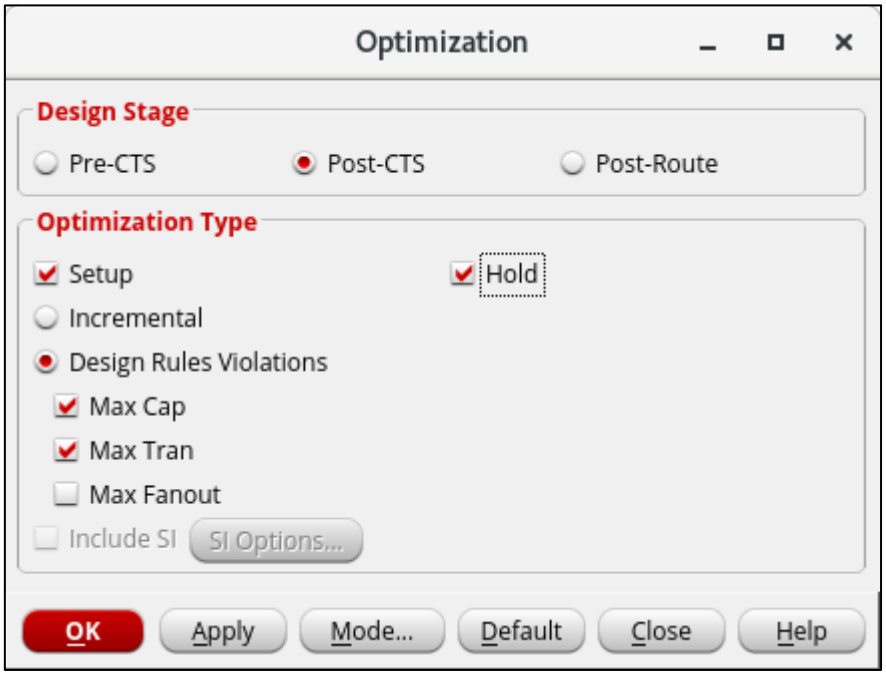


Fig. 160: Optimization window

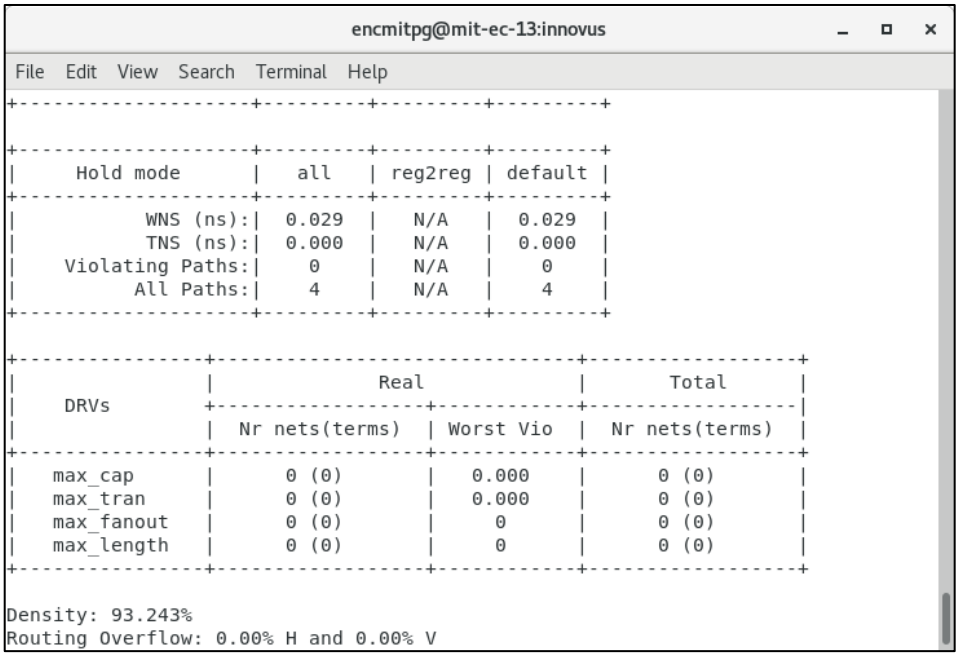


Fig. 161: Timing report after optimization (terminal window)

Routing :

1. All the net connections shown in the GUI till CTS are only based on the Logical connectivity.
2. These connections are to be replaced with real Metals avoiding Opens, Shorts, Signal Integrity [Cross Talks], Antenna Violations etc.
3. To run Routing, Select Route → Nano Route → Route and enable Timing Driven and SI Driven for Design Physical Efficiency and Reliability

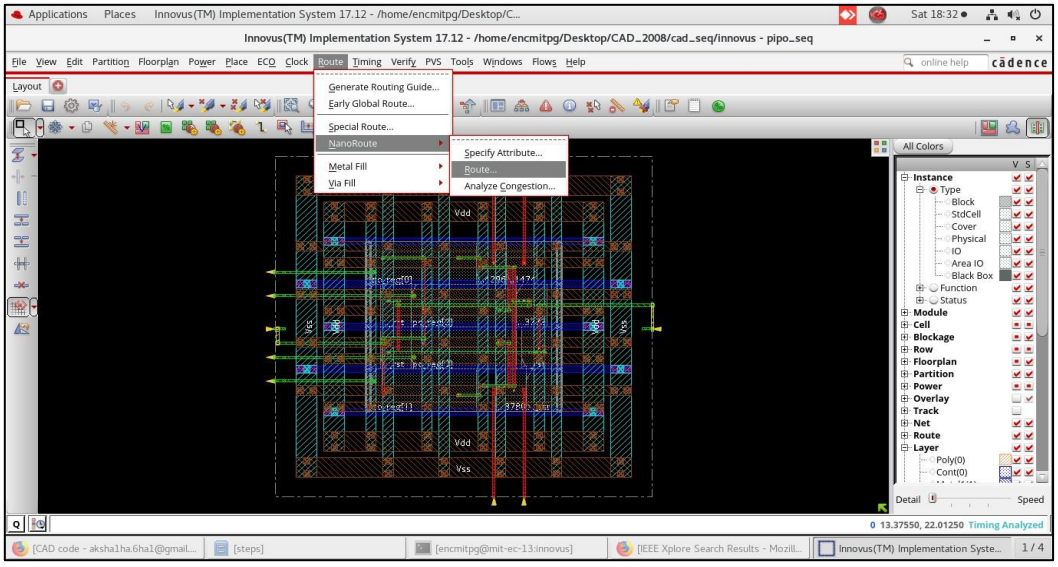


Fig. 162: Click on Route -> Nano Route -> Route

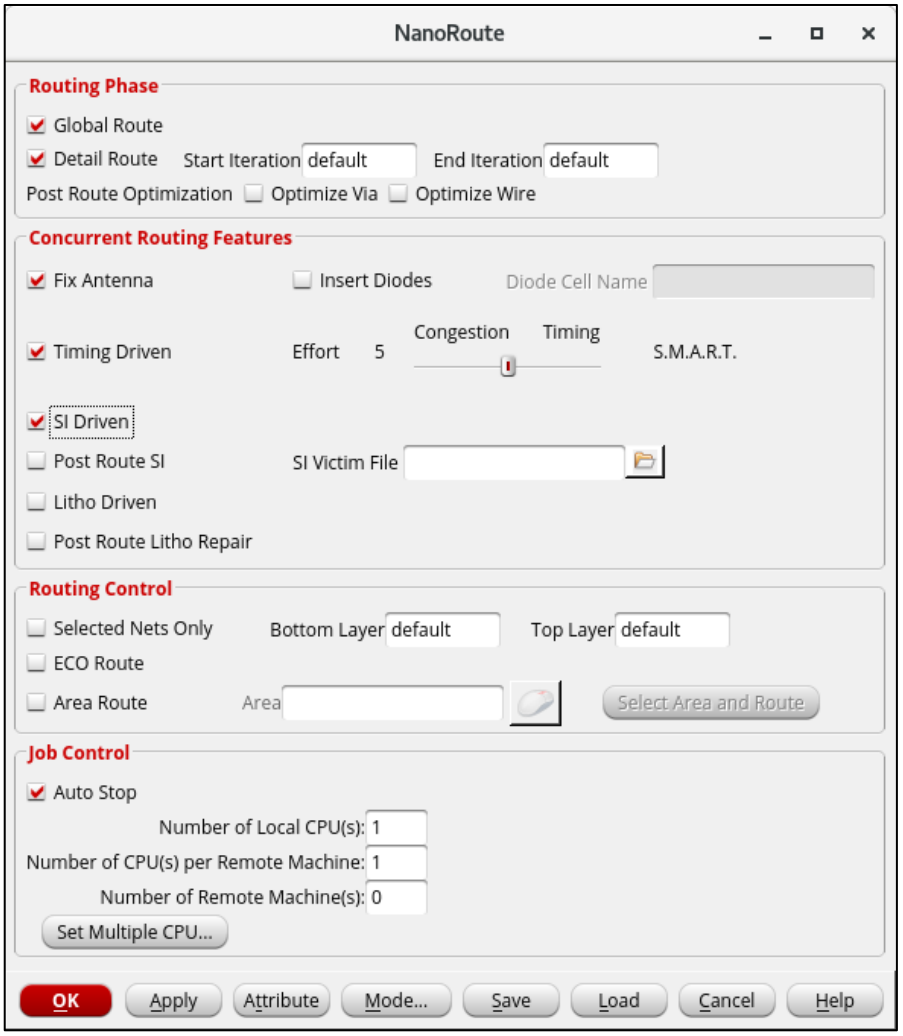


Fig. 163: Nano route window

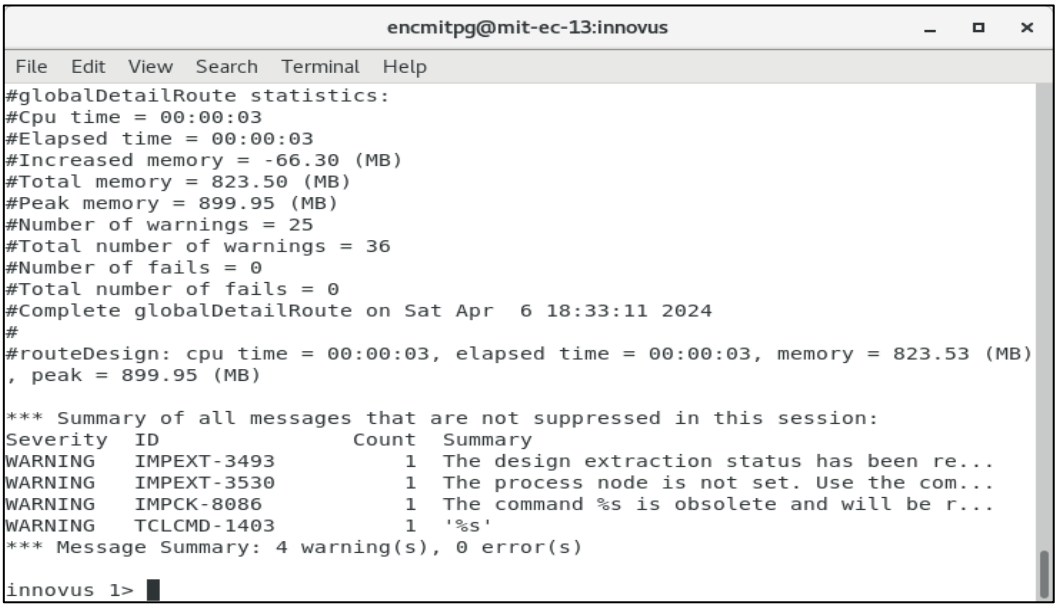


Fig. 164: Terminal window after nano route

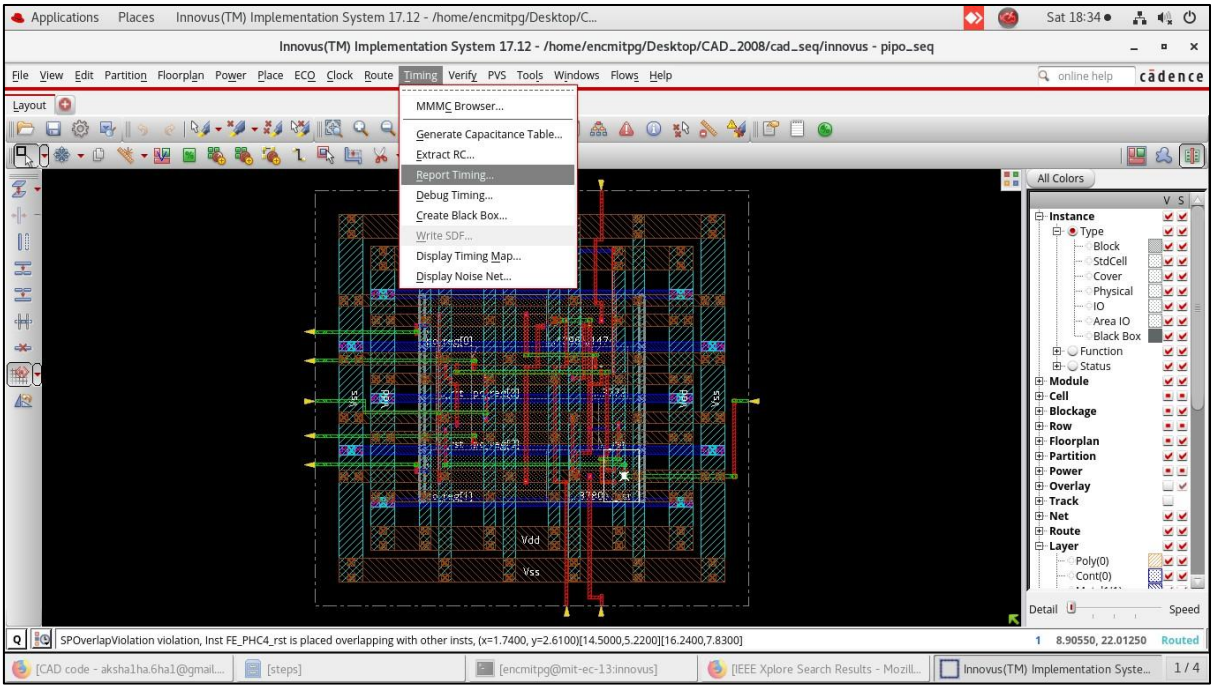


Fig. 165: Click on timing -> Report timing

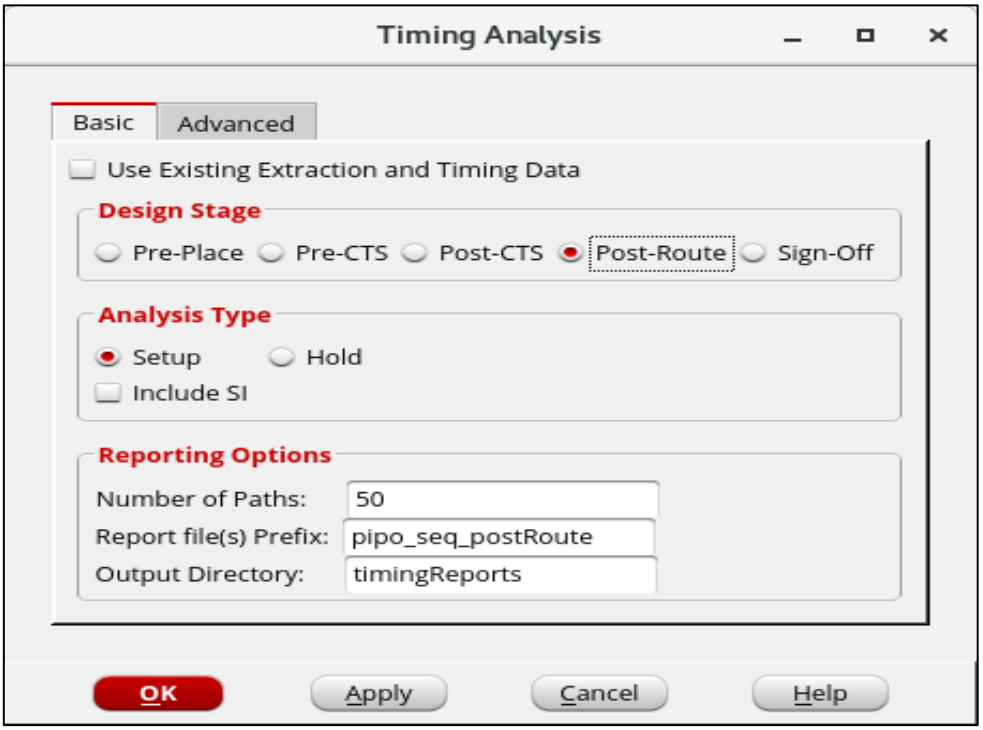


Fig. 166: Setup timing Analysis

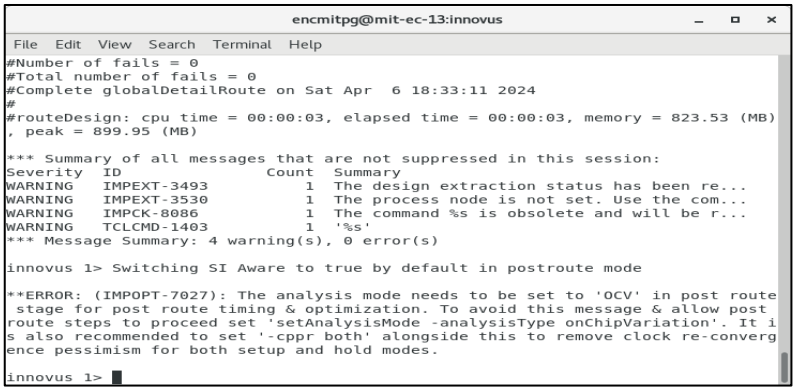


Fig. 167: terminal window showing error for post route analysis

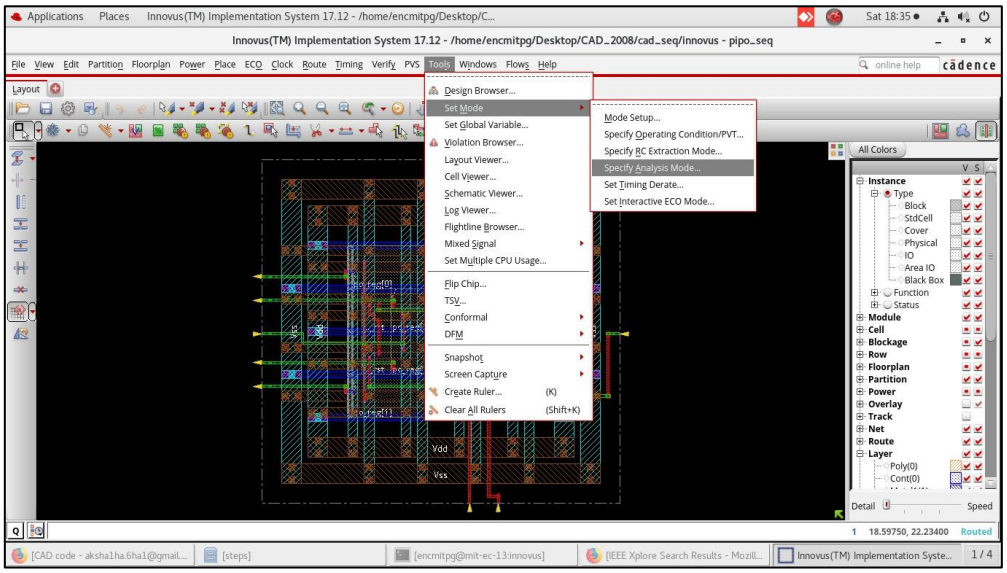


Fig. 168: Click on tools -> set mode -> specify analysis mode

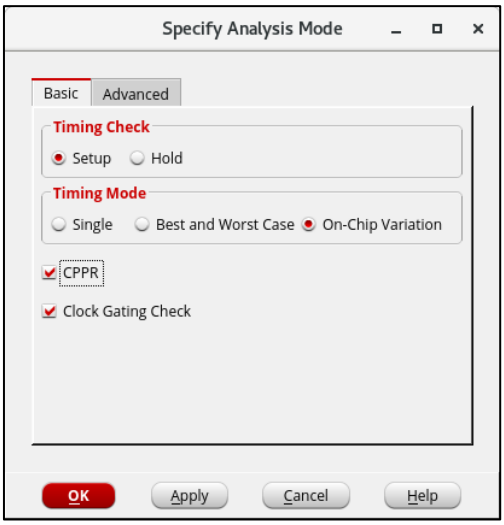


Fig. 169: select On-chip variation and CPPR

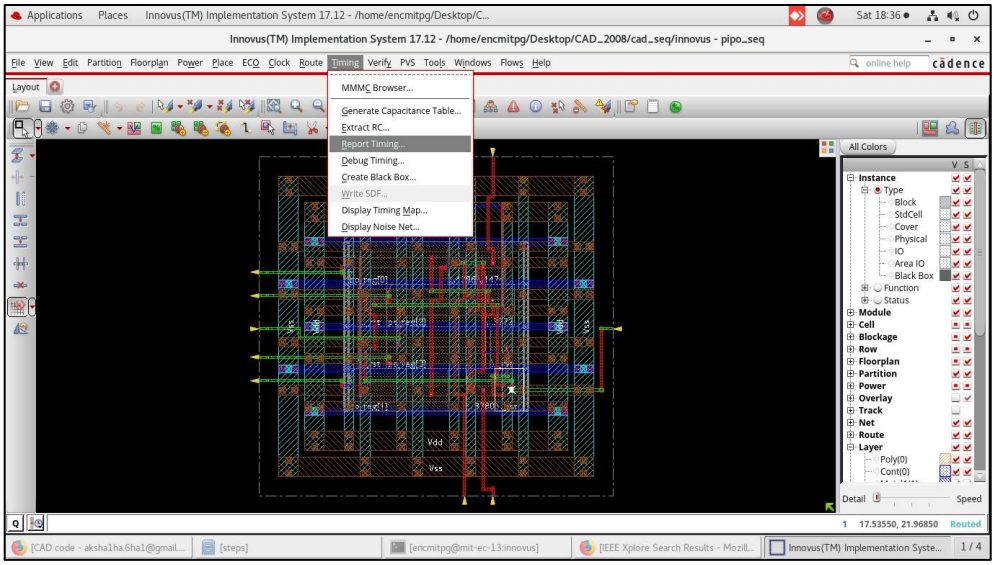


Fig. 170: click on Timing -> report timing

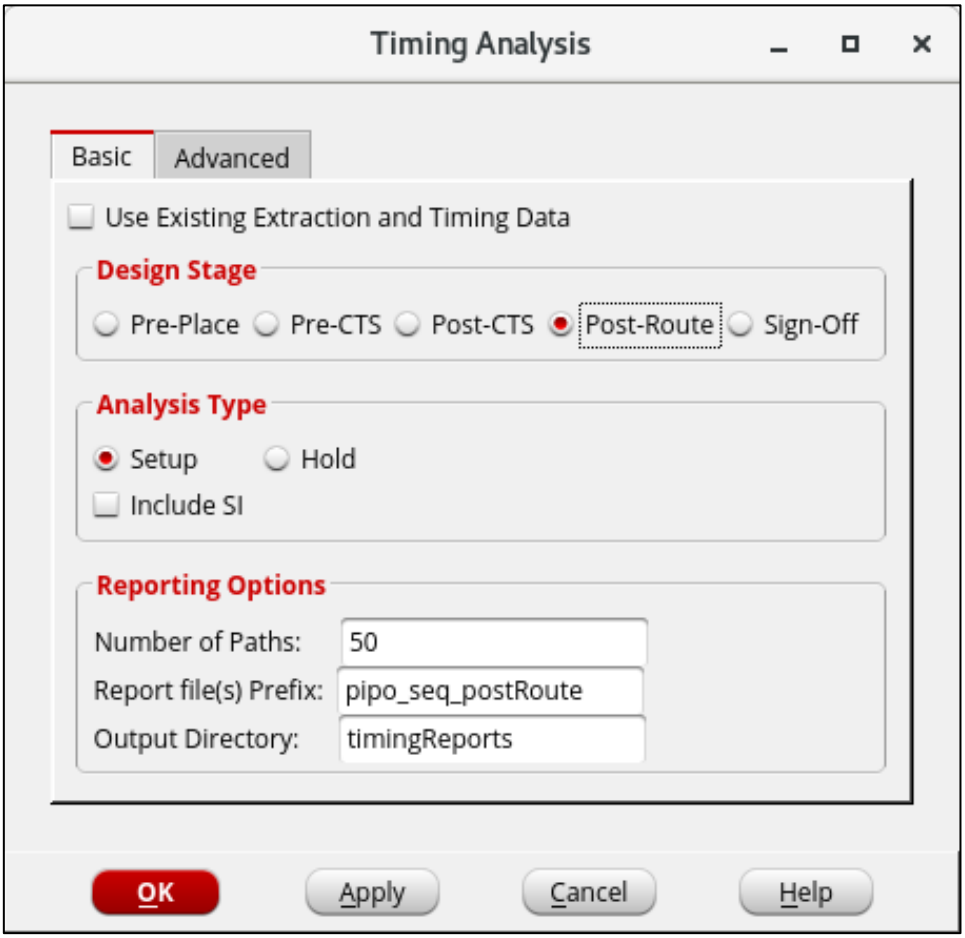


Fig. 171: Setup analysis

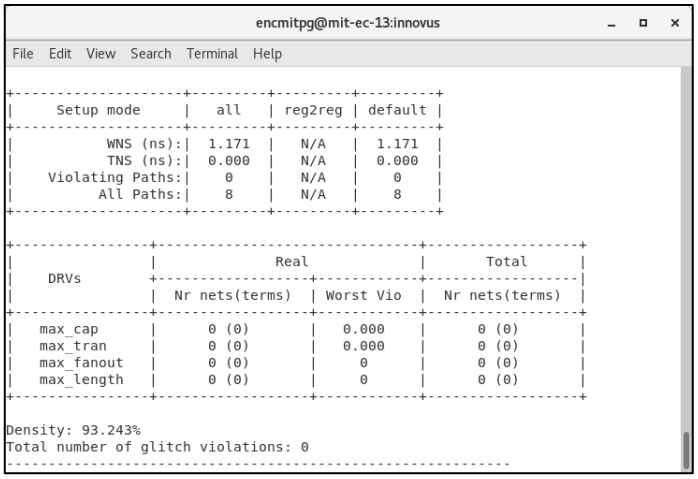


Fig. 172: setup timing report (terminal window)

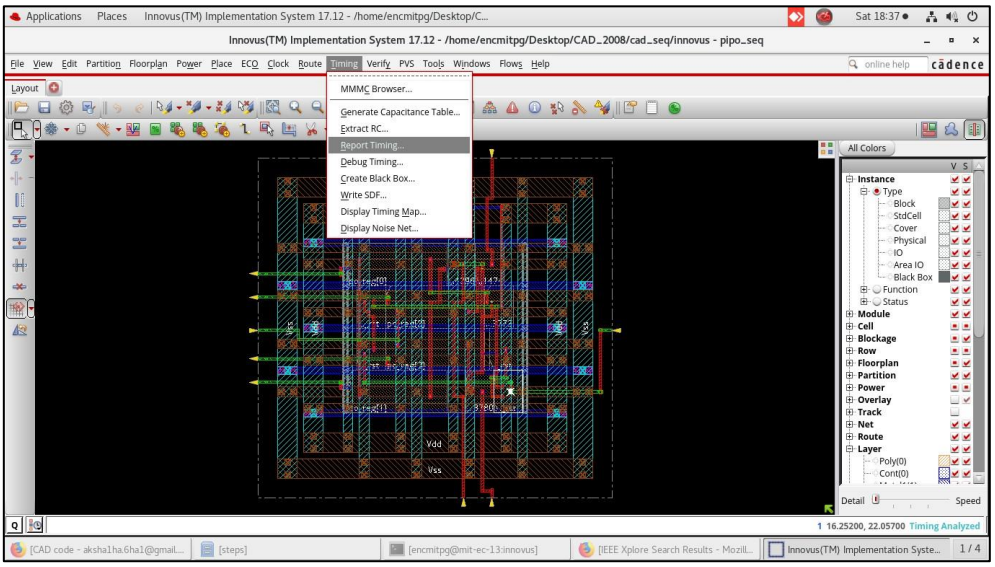


Fig. 173: Click on Timing -> Report timing

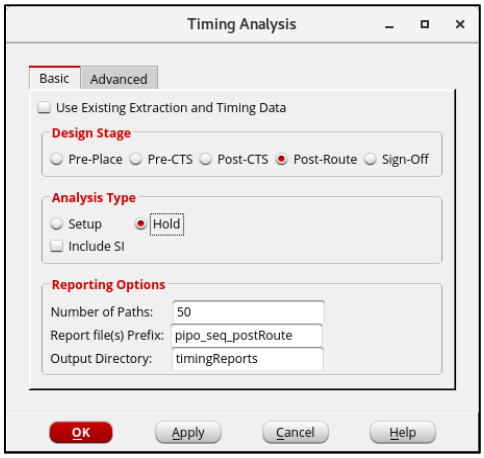


Fig. 174: Hold Time analysis

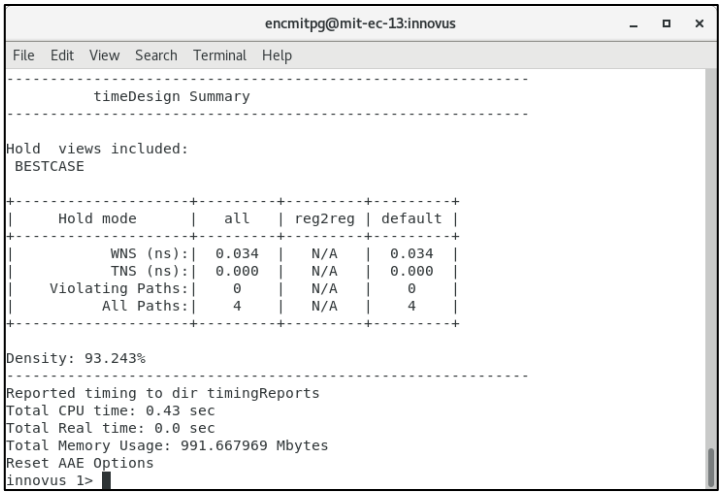


Fig. 175: Hold timing report with violating paths (terminal window)

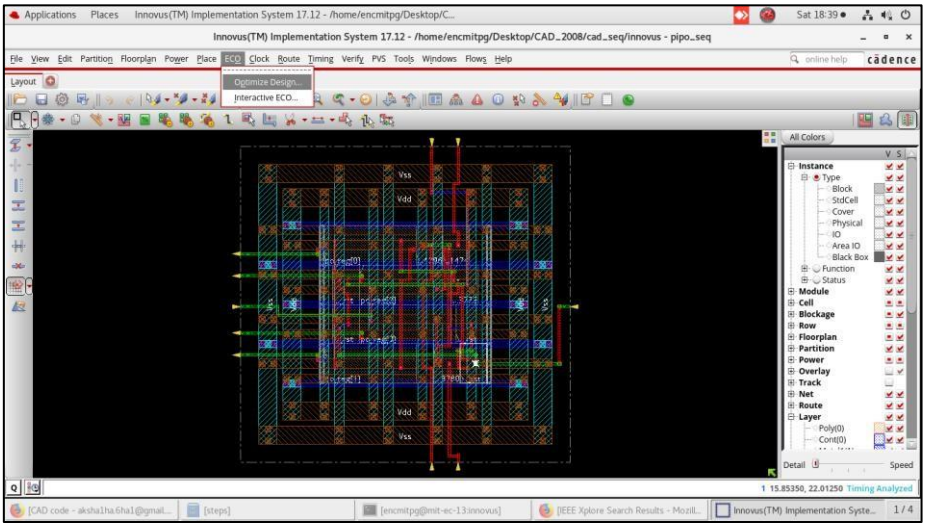


Fig. 176: Click on ECO -> Optimize design

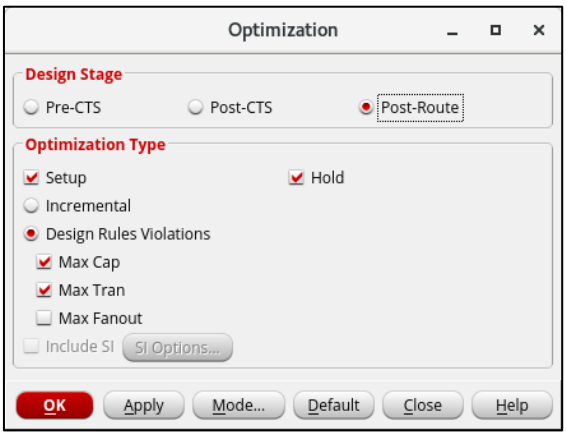


Fig. 177: Optimization window

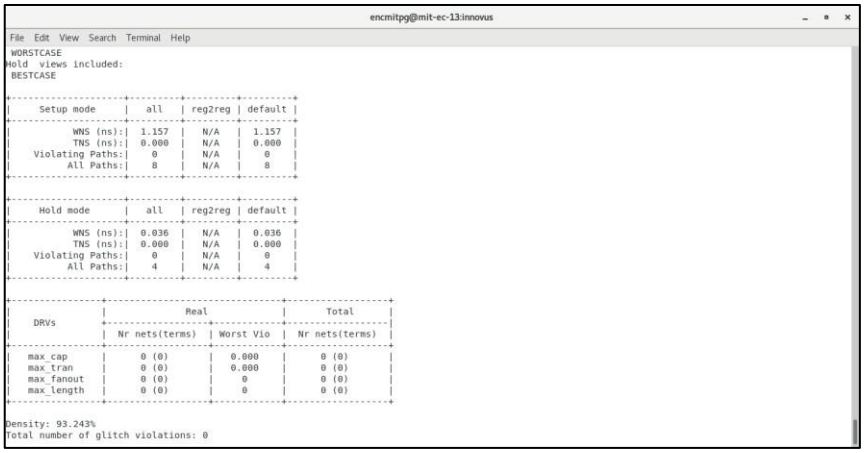


Fig. 178: Hold Timing report (terminal window)

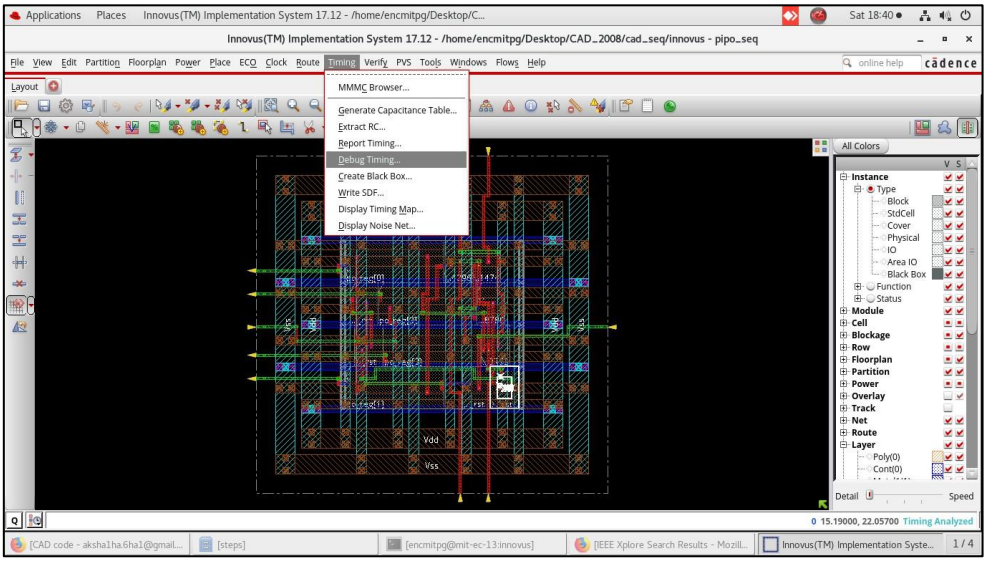


Fig. 179: Click on Timing -> Debug Timing

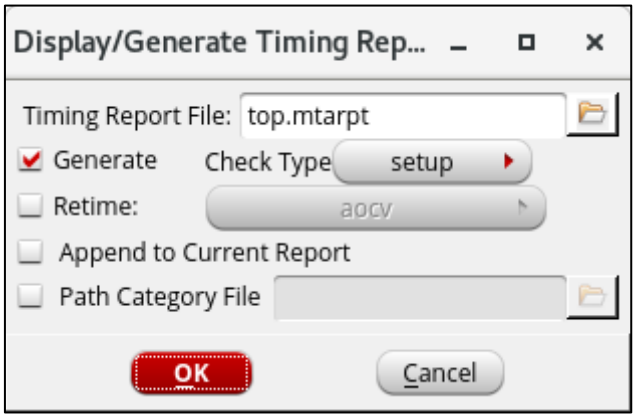


Fig. 180: Select check type as set up

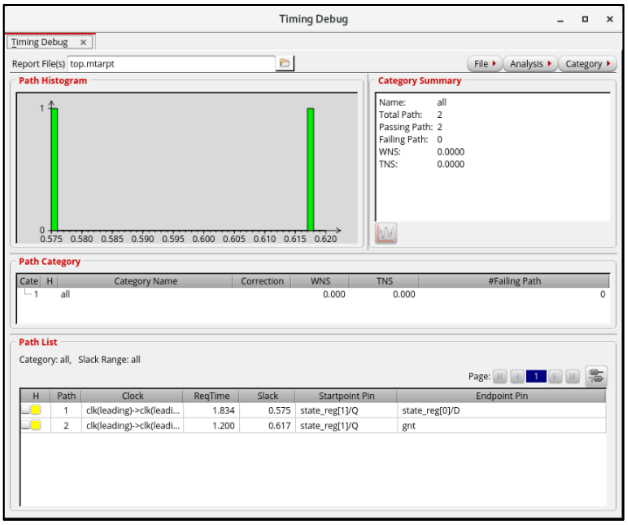


Fig. 181: Timing Debug window (setup)

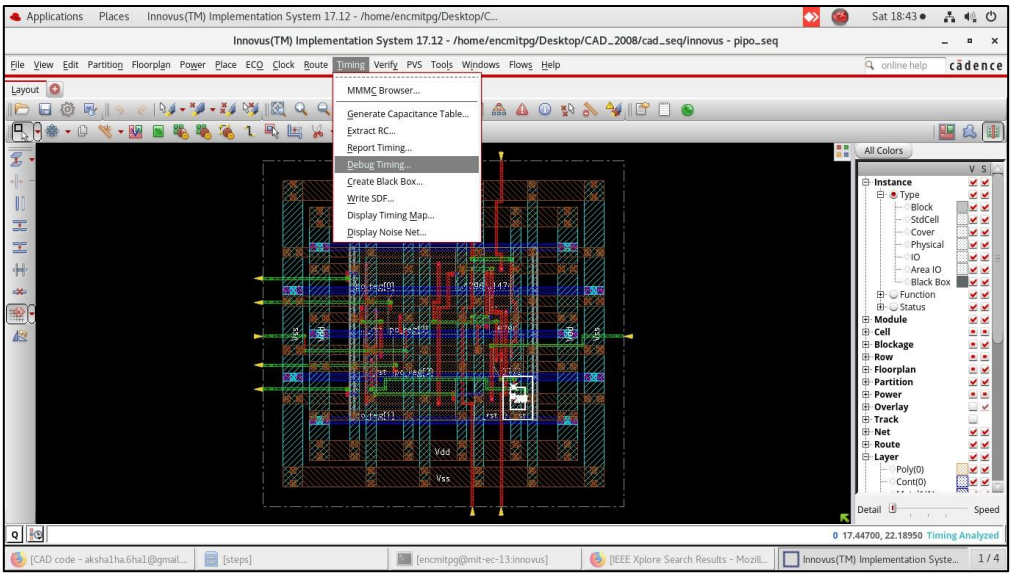


Fig. 182: Click on Timing -> Debug timing

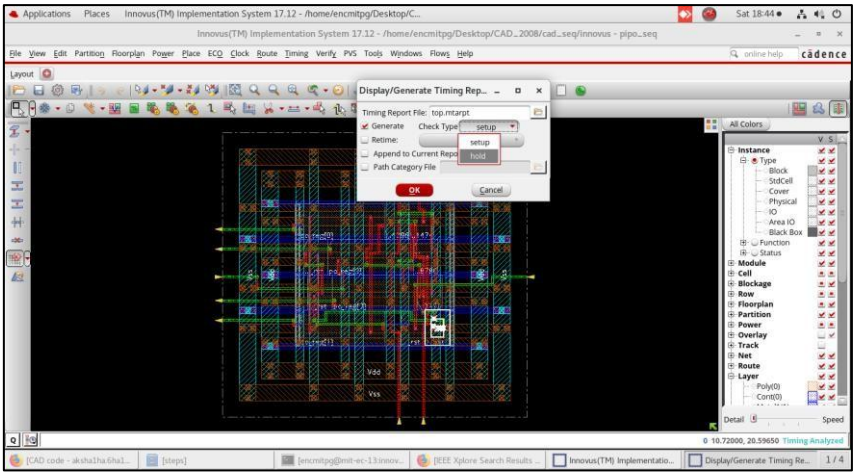


Fig. 183: Select check type as hold

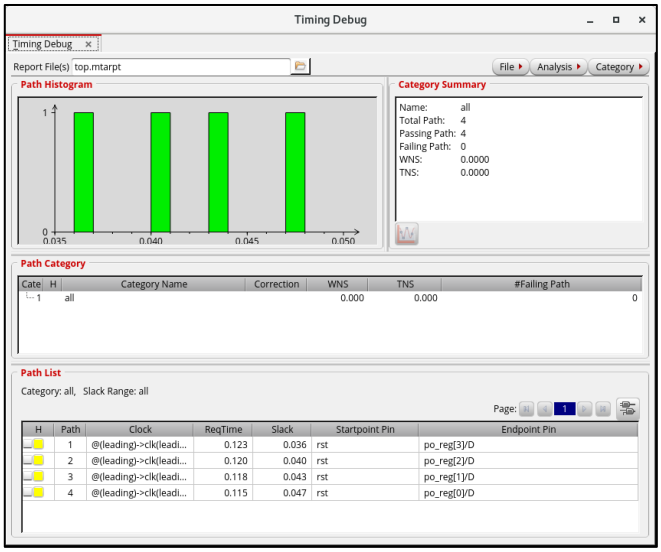


Fig. 184: Timing debug window (hold time)

- ~~Save~~ Design:

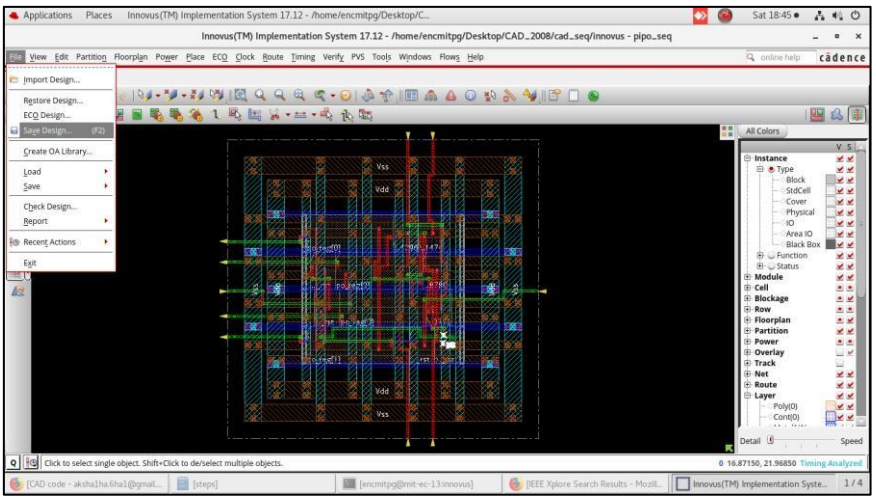


Fig. 185: File ->Save Design

Fig. 186: Select data type as Innovus and give file name

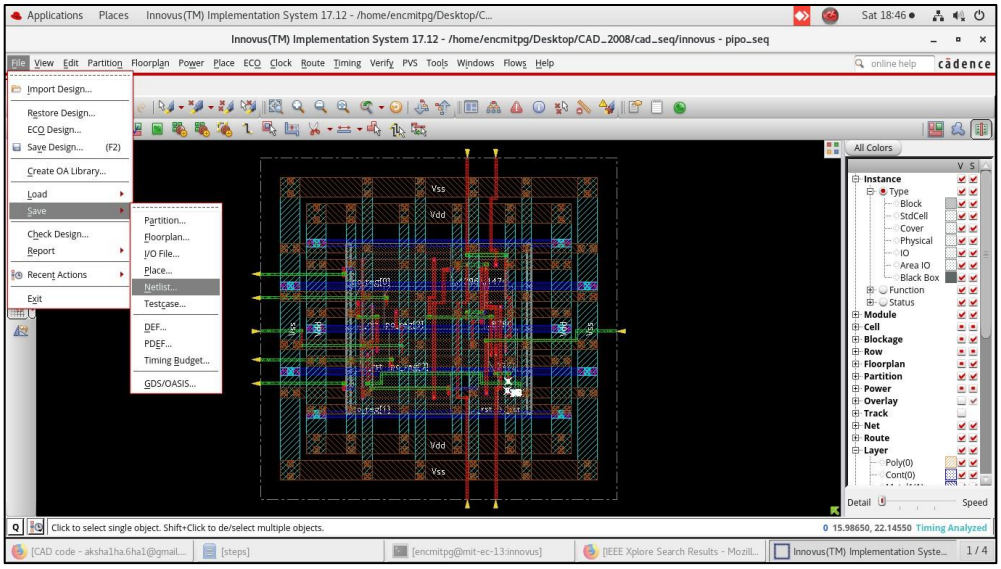


Fig. 187: File -> save -> netlist

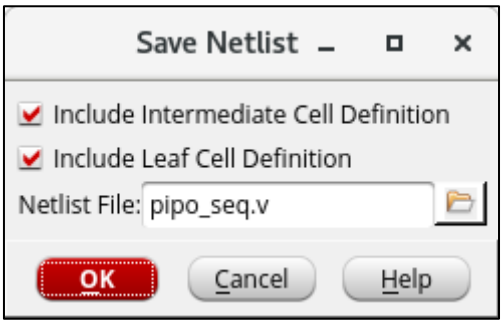


Fig. 188: Give netlist name and then click ‘ok’

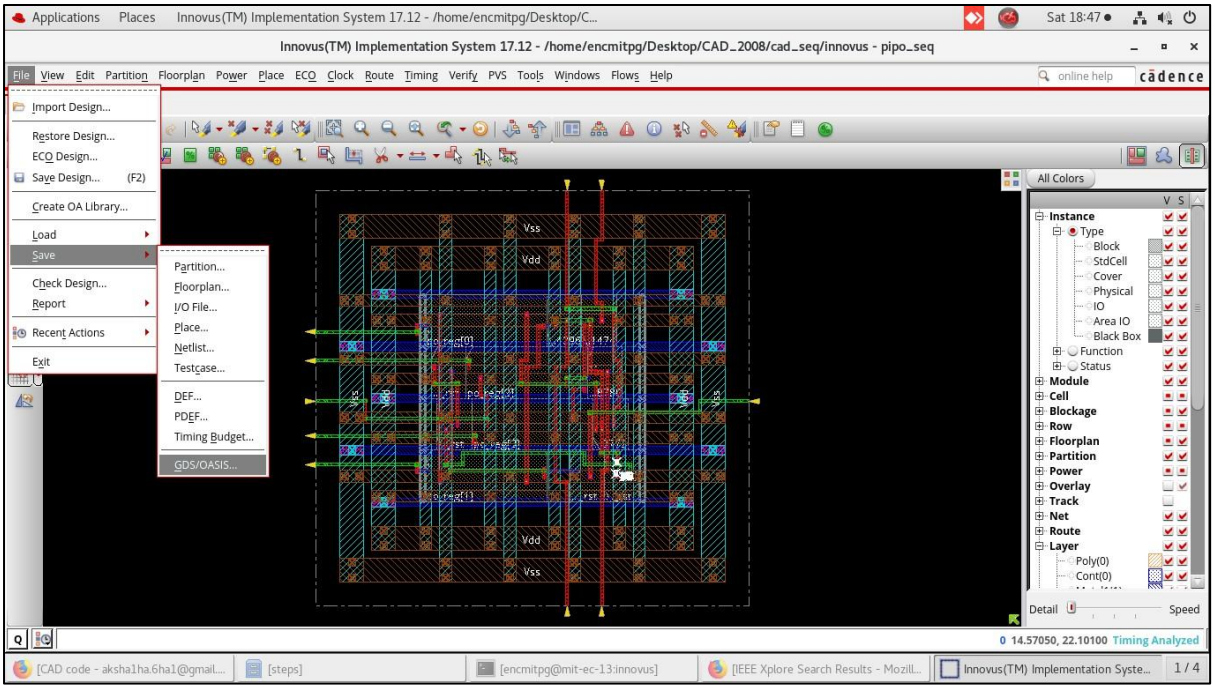


Fig. 189: File -> save -> GDS/OASIS

Fig. 190: Give output file name and then click ‘ok’

Exercise:

1. Design a 4-bit synchronous counter
   a. Write a Verilog code and perform synthesis for the MOD 10 counter.
   b. Do the physical design (Generate a Layout) of the a.
2. 4-bit Barrel shifter