Ahsanullah University of Science and Technology

Department of Electrical and Electronic Engineering

LABORATORY MANUAL

FOR

ELECTRICAL AND ELECTRONIC SESSIONAL COURSES

Student Name :

Student ID :

Course No. : EEE 4232

Course Title : VLSI II Lab.

For the students of

Department of Electrical and Electronic Engineering

4th Year, 2nd Semester

Table of Contents

Lab-0: Overview of VLSI-II Laboratory

Objective

The main objectives of this lab are:

- Familiarization with Application Specific Integrated Circuits (ASIC) design flow.
- Overview of the VLSI-II lab.

Introduction

To design very large-scale integrated circuits some frontend and backend processes needed to be acomplished. The processes can be represented as a flow chart to show the life cycle of a chip which is called Application Specific Integrated Circuits (ASIC) design flow. A typical ASIC design flow is shown below.

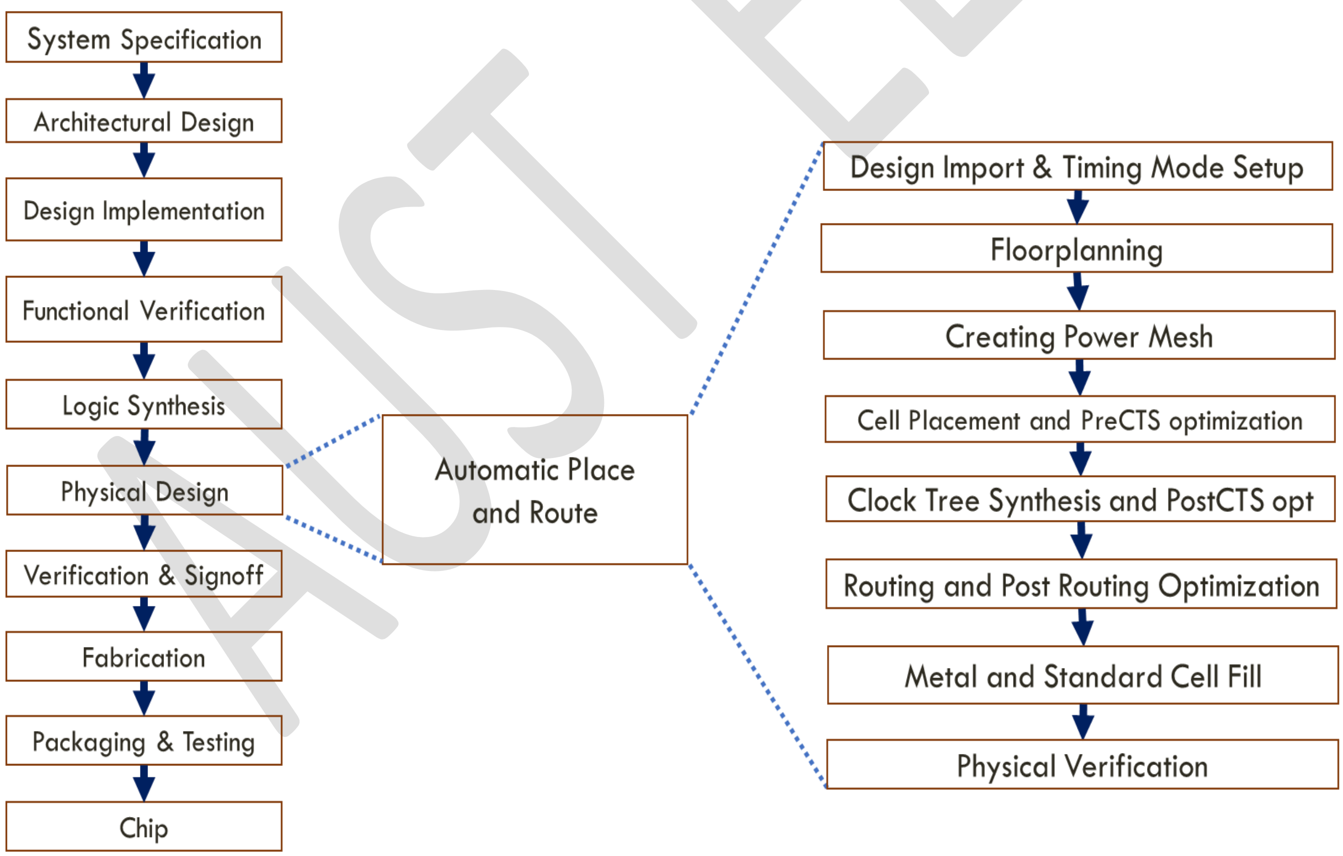


Figure: ASIC design flow

System Specification

Design functionality, performance factors (speed, power, latency, throughput, dimension, data size), cost, I/O requirements etc are clearly stated at this stage.

Architectural Design

Determines required different architecture blocks to implement the design to maximize the performance factors. It also determines the algorithm for optimized connection of the blocks and formal verification is performed.

Design Implementation

The system can be designed in two ways: analog design and digital design. In the analog design process, circuit blocks are designed at the transistor level. On the other hand, the synthesizable RTL description of the device is programmed using Hardware Description Language (HDL) in the digital design process. HDL Programming can be easily implemented for any modern complex device as it gives the advantage of simulating and verifying the design output and functionality efficiently.

Functional Verification and Testing

Functional simulation is performed in this stage, and the logic of the system is verified using timing simulation and test vectors. If the functionality doesn't match the Function should be designed again

Logic Synthesis

The process of translating the RTL into a gate-level netlist is called Synthesis. In this process, the design is optimized, and technology mapping or library binding is done. The gate-level netlist must undergo formal verification to prove that RTL and netlist are equivalent.

Physical Design

Physical Design is the process of transforming a circuit description into a physical layout that describes the position of cells and routes for the interconnections between them. The physical design consists of the following steps.

- **Design Import & Timing Mode Setup**
- **Floorplanning**
- **Creating Power Mesh**
- **Cell Placement and PreCTS optimization**
- **Clock Tree Synthesis and PostCTS opt**
- **Routing and Post-Routing Optimization**
- **Metal and Standard Cell Fill**

Verification and Signoff

Verification would either be just before the tapeout stage of the chip or the stage where design is again taken back through the same flow for optimization. The following verifications are usually performed in this stage.

- **Design Rule Check (DRC):** It checks design rules such as shapes/size/spacing and many other complex rules of each metal layer.
- **Layout vs Schematic (LVS):** It checks whether the design layout is equivalent to its schematic.
- **Antenna Rule Check (ARC):** Checks for a large area of metals that might affect the manufacturing process.
- **Electrical Rule Check (ERC):** The methodology used to check the robustness of a design both at schematic and layout levels against various electronic design rules.

After all verifications, post-processing is applied where the physical layout data is translated into an industry-standard format called **GDSII**. The GDSII file is sent to the semiconductor foundry to convert it into mask data which is called **tapeout**. GDS II is a database file format that is the industry standard for data exchange of integrated circuit or IC layout artwork. It is a binary file format representing planar geometric shapes, text labels, and other information about the layout in hierarchical form. It is also referred as Graphic Design System.

Fabrication

The mask of physical design is sent to factories called fabs(clean room). Several masks are used in turn, each one reproducing a layer of the completed design Masks are used to create a specific pattern of each material in a sequential manner and create a complex pattern of several layers Introduction

For fabricating an IC in the clean room following steps are performed.

- Wafer Preparation
- Oxidation
- Lithography (Photoresist & Masking)
- Etching
- Dopant Incorporation (Diffusion & Ion Implantation)
- Crystal Epitaxial Growth
- Deposition
- Isolation
- Cleaning

Packaging & Testing

After fabricating the chip in a clean room, it should pass some specific tests before commercial use. If all test is confirmed it is packaged and sent to the consumer.

Chip

The final output of the process is a chip.

EDA Files

Liberty Timing File (.lib file)

ASCII representation of the timing and power parameters associated with any cell in particular semiconductor technology. Types of lib file Fast lib, Slow lib, and Typical lib. Basic differences among those libraries are Nominal voltage Nominal temperature cell leakage Power Capacitance, Fall power, Rise power, and Timing.

Library Exchange Format (.lef file)

LEF is a specification file for representing the physical layout of an IC in an ASCII format. It contains library information for a class of designs. It mainly contains Layer information, Via information, Placement site type and origin, and Macrocell definitions.

SDC (Standard Design Constraint)

The Standard Design Constraint format is used to specify the design intent, including the timing, power and area constraints for a design.

Cap table

Cap table contains information of parasitic Resistance and Capacitance which is used to model the interconnect of a design.

Cdb (Celtic Database)

For signal integrity analysis besides lib files, the tool required the .cdb files also. The main issues of concern for signal integrity are Ringing, Crosstalk, Ground bounce, Distortion, Signal loss, Power supply noise.

Commonly used EDA Tools

Function	Tools
Analog Design	Cadence Virtuoso, HSPice, LTSpice
Cell Layout Design	Cadence Virtuoso Layout Suit
RTL Coding	Cadence NCSim, ModelSim, Quartus
Synthesis	Cadence Genus, Yosys Open Synthesis Suite
Physical System Design and STA	Cadence Encounter, Innovus
Verification	Cadence Assura, Mentor graphic Calibre

Probable List of Lab Tasks

The following processes of VLSI ASIC design flow will be covered in the upcoming classes.

Front End Process

- Verilog HDL programming language.
- Functional Verification using Verilog Testbench.
- Modeling Sequential Systems and FSM using Verilog.
- Synthesis

Backend Process

- Physical Design
- Static Timing Analysis
- Physical Verification and Power Analysis

Assessment Procedure and Marks Distribution (Tentative)

Assessment Type	Percentage
i) Continuous Performance	10
ii) Lab Test-1	20
iii) Lab Test-2	25
iv) Assignment	15
v) Project	30
Total	**100**

Lab-1: Introduction to Verilog HDL Programming

Objective

The main objectives of this lab are:

- Basic terminology of Verilog HDL programming.
- Familiarization with different levels of Abstraction in Verilog HDL.
- Simulating Verilog HDL using ModelSim.

Introduction

A system or chip can be designed in two ways: analog design and digital design. In the analog design process, circuit blocks are designed at the transistor level. Nowadays high performing chips are designed with more smarter functions and that has increased the density of the transistor in a chip. In VLSI (Very Large-Scale Integration) technology chips are designed with more than 100,000 transistors. So it is not easy to design and verify such a complex system in an analog process. In the digital design process, according to the functionality of a chip, a synthesizable RTL description of the system is modeled using the Hardware Description Language (HDL). HDL gives the advantage of simulating and verifying the design output and functionality easily before they were fabricated on chips. For a long time, programming languages such as FORTRAN, Pascal, and C were used to describe sequential computer programs after that Hardware Description Languages (HDLs) came into existence to model the concurrency processes found in hardware elements. Some common HDLs are Verilog, System Verilog, VHDL, VerilogA.

Verilog Module

Modules are the building blocks of the Verilog design. Modules can be embedded within other modules, and a higher level module can communicate with its lower-level modules using their input and output ports. A module should be enclosed within a module and **endmodule** keywords. The following figure shows the structure of any Verilog module.

```
module module_name [(port_name{, port_name})];
    [parameter declarations]
    [input declarations]
    [output declarations]
    [inout declarations]
    [wire or tri declarations]
    [reg or integer declarations]
    [function or task declarations]
    [assign continuous assignments]
    [initial block]
    [always blocks]
    [gate instantiations]
    [module instantiations]
endmodule
```

Port Types

Port provides the interface by which a module can communicate with the internal and external environment. Based on the direction of the signal Verilog language allows three types of ports. Ports can be declared as follows.

Type of Port	Verilog Keyword
Input port	input
Output port	output
Bidirectional port	inout

Data Types

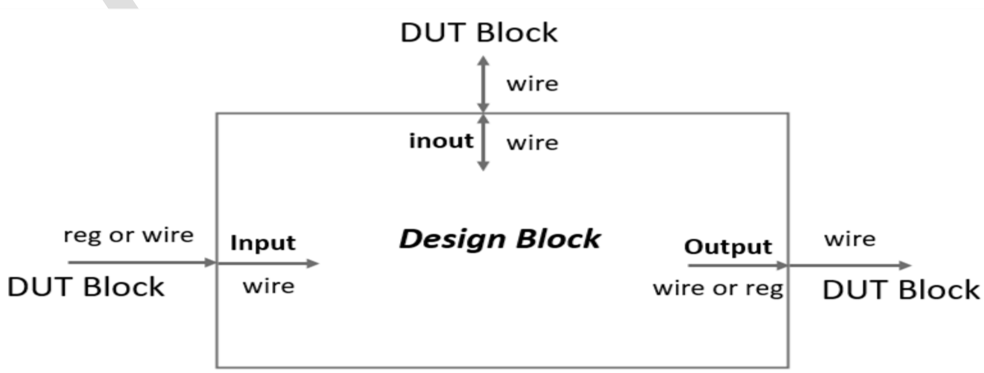
Verilog language has two primary data types called Nets and Registers.

1. **Nets**
    - Represents structural connections between components.
    - Declared as ‘wire’.
    - By default, one bit.
    - All port declaration are implicitly declared as wire in Verilog
2. **Registers**
    - Represents the variables used to store data.
    - Declared as ‘reg’.
    - Stores/holds the last assigned value until it is changed.
    - Must use register data type if a signal is assigned in procedural

In Verilog, **“parameter”** is used to declare constants and does not belong to any other data type such as register or net data types. A constant expression refers to a constant number or previously defined parameter. We cannot modify parameter values at runtime, but we can modify a parameter value using the **“defparam”** statement. In modern RTL design, “**localparam**” is used to declare constants.

Port Connection Rule

Verilog simulator shows violations if port connection rules are violated.



1. **Input**
    - Internal input ports must always be **net (wire)** type.
    - External input ports can be connected to **reg** or **net** type.

2. **Output**
    - Internal output ports can be either **reg** or **net** type.
    - External outputs must be **net** type.

3. **Inouts**
    - Internally and externally inout ports must be **net** type.
    - They are bidirectional.eg-power, ground, etc.

4. **Width Matching**

    It is legal to connect internal and external items of different sizes when inter-module port connections. However, a warning is typically issued that the width does not match.

5. **Unconnected Ports**

    Verilog allows ports to remain unconnected. For example, a full adder module has three inputs (A, B, C) and two outputs (sum, carry). So, if we don't want to use any of the inputs or outputs during the submodule call, we simply ignore that by keeping the place blank. Example if a module is full_add(A, B, C, SUM, Carry) during the submodule call if we want to ignore the C input can write as full_add a1(x,y, ,z,l)

Literals

Literals are used for representing constant numbers. The syntax for a constant is shown below.

<size>' <sign><base> <number>

* The number of binary bits the number is comprised of.
*Default is 32 bit

*Indicates if the number is signed.
*Either s or S.
*Not case sensitive.
*Default is unsigned

*Radix of the number.
*Binary: b or B
*Octal: o or O
*Hexadecimal: h or H
*Decimal: d or D
*Not case sensitive.
*Default is decimal.

Number according to base.

Example 01

The following example demonstrates the Verilog syntax for different literals and data types.

```
parameter a,b,c,d,e,f,g,h; // declaration of multiple variables of parameter type
reg [7:0]i;          // reg type variable declaration which can store up to 8-bit
reg[7:0]j;           // reg type variable declaration which can store up to 8-bit
a=549;               // decimal number 549, no size specified
b=4'bx;              //4-bit unknow value xxxx
c=8'hfx;             // 8-bit number equivalent to 8b1111_xxx
d='h8FF;             // hex number, no size specified
e=5'd3;              // 5-bit decimal number 00011
f=8'b00001011;       //8-bit binary number 00001011
g=8'b0000_1011;      // "_" is a separator used to improve the readability of 8-bit number 00001011
h=8'b1011;           //8-bit binary number 00001011
i=4'sb1011;          // 4-bit positive signed number 00001011
j= - 4'sb1011;       //initializes with 1011 then for negative sign 2s complement is performed which
                     is 0101 then 4 zeros are padded for signed value 00000101
```

Example 01 is not a complete Verilog Module it just demonstrates the syntax

Verilog Operators

To represent the functionality of a digital system different operators such as logical, bitwise, etc. operators must be used. In the following table, different Verilog operators are shown.

Table demonstrating different operators

{ }	concatenation	~	bit-wise NOT
+ - * / **	arithmetic	&	bit-wise AND
%	modulus	|	bit-wise OR
> >= < <=	relational	^	bit-wise XOR
!	logical NOT	^~ ~^	bit-wise XNOR
&&	logical AND	&	reduction AND
||	logical OR	|	reduction OR
==	logical equality	~&	reduction NAND
!=	logical inequality	~|	reduction NOR
===	case equality	^	reduction XOR
!==	case inequality	~^ ^~	reduction XNOR
? :	conditional	<<	shift left
		>>	shift right

Example 02

The following example demonstrates the basic logical syntax of basic logical operation used in digital system representation. We can represent the logical expressions in two ways called **Gate Instantiations** and **Continuous Assignment.**

```
module gates(A,B, Yn,Ya,Yo,Yx, Zn,Za,Zo,Zx);
// Gate Instantiations
output Yn,Ya,Yo,Yx;
input A, B;
not g1(Yn,A);
and g2(Ya,A,B);
or g3(Yo,A,B);
xor g4(Yx,A,B);
// Continuous Assignment
output Zn,Za,Zo,Zx;
assign Zn=~A;
assign Za=A&B;
assign Zo=A|B;
assign Zx=A^B;
endmodule
```

Verilog Modeling Styles

Digital systems are generally modeled in four ways called **Switch-level modeling, Gate level or structural modeling, Data flow modeling (DFM),** and **Behavioral modeling**.

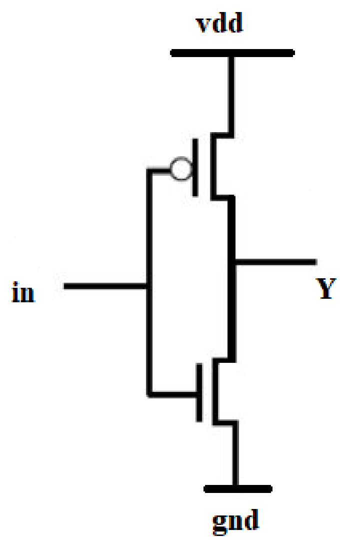
N.B: RTL is a combination of Data Flow and Behavior Modeling styles. The logic synthesis tool can generate a gate-level netlist from RTL.

1. Switch level Modeling

This method provides mechanisms for modeling MOS transistors using Verilog. This modeling style is used in very specific cases, for designing leaf cells in a hierarchical design. Switch-level modeling is not detailed enough to catch many of the problems.

Example 03

The following example demonstrates the Verilog HDL code of an CMOS inverter using the switch level abstraction.



CMOS Inverter

```
module inv_cmos(in,Y);
input in;
output Y;
supply1 vdd;
supply0 gnd;
pmos p1(Y,vdd,in);
nmos n1(Y,gnd,in);
endmodule
```

2. Gate level or structural modeling

In this method, a system is designed using predefined gates or user-defined primitives. It is white box modeling because every design is visible inside the design. It is the lower level of abstraction.

Example 04

The following example demonstrates the Verilog HDL code of a two to one multiplexer module using the gate level abstraction.

```
/*
Steps for Gate Level Modeling
I. Develop the Boolean function of output
II.Draw the logic diagram.
III.Connect the gates with nets(wires).
*/
module mux_2to1(s,Io,I1,Y);
input s,Io,I1;
output Y;
wire w1,w2,w3;
not (w1,s);
and (w2,Io,w1);
and (w3,s,I1);
or (Y,w2,w3);
endmodule
```

3. Data flow modeling (DFM)

In this method, a system is designed by specifying the data flow between input and output. It uses continuous assignment statements to drive a value on a net or wire. It is a higher level of abstraction than the gate level. It may be either black-box modeling or white-box modeling depending on the design complexity.

Example 05

The following example demonstrates the Verilog HDL code of a two to one multiplexer module using the data flow modeling.

```
/*
Steps for Data Flow Modeling
I.Obtain the relation between output and input.
II.Impalement the logical relation using "assign" statement.
*/
module mux_2to1(s,Io,I1,Y);
input s,Io,I1;
output Y;
wire w1,w2,w3;
assign w1=~s;
assign w2=Io & w1;
assign w3=s & I1;
assign Y=w2 | w3;
endmodule
```

4. Behavioral modeling

In this method, a system is designed and implemented in terms of a design algorithm based on the behavior of the design and its performance. Verilog behavioral code must be inside procedural statements/blocks only. It is the highest level of abstraction. It is also known as black-box modeling.

Procedural Block

There are two types of procedural blocks in Verilog called **“Initial”** and **“always”** blocks. Procedural blocks are evaluated in the order in which they appear in the code that’s why it is also known as sequential statements. Procedural statements assign values to reg, integer, real or time variables. Procedural blocks cannot assign values to nets.

a) “initial” Block

- Statements inside the initial block are executed only once.
- Executes at time zero.
- Used in Test bench

b) “always” Block

- Sensitivity list or list of signals that directly affect the output result must be defined in always block.
- Whenever the value of a signal in the sensitivity list changes then the statements inside the always block is executed.

```
always @ (sensitivity_list)
begin
        [procedural assignment statement]
        [if-else statement]
        [case statement]
        [while, repeat and for loops]
        [task and function calls]
end
```

Example 06

The following example demonstrates the Verilog HDL code of a two to one multiplexer module using the behavioral modeling style. The always procedural block is used here to set the output of multiplexer(y) whenever any of the inputs (Io and I1) or selection input (s) changes.

```
/*
Steps for Behavioral Modeling
I.Develop a behavioral algorithm (like ‘C’ programming).
```

```
II.According to the algorithm insert the behavioral statements inside the appropriate procedural
block
*/
module mux_2to1(s,Io,I1,Y);
input s,Io,I1;
output reg Y;
always@ (s,Io,I1)  //if we use always @* The * operator will automatically identify all sensitive variables.
begin
	if(s==0)
		Y=Io;
	else
		Y=I1;
end
endmodule
```

Hierarchical Modeling

A Hierarchical methodology is used to design simple components to construct more complex components There are two design approaches when writing code in a hierarchical style called **Top-Down** and **Bottom-Up** methodology.Typically, designers use these two approaches side-by-side to construct complex circuits.

1. Top-Down Methodology

In a top-down design methodology, we define the top-level block and identify the sub-blocks necessary to build the top-level block. We further subdivide the sub-blocks until we come to leaf cells, which are the cells that cannot further be divided.

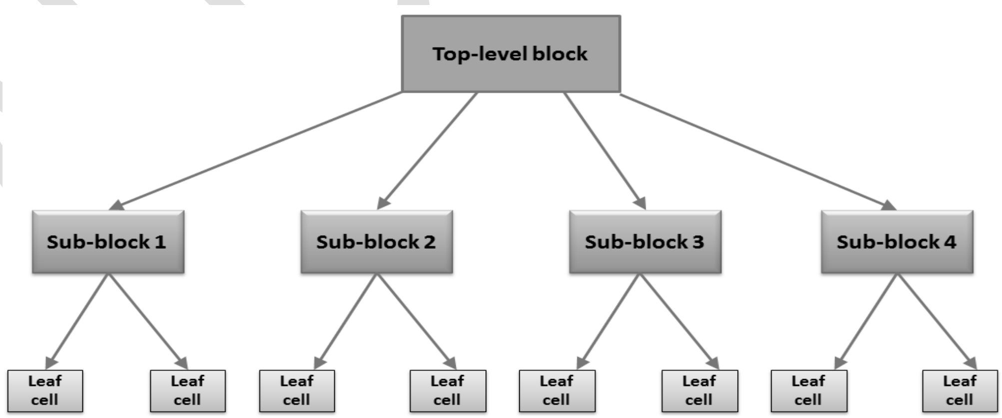


*Figure: Block representation of **Top-Down** methodology*

2. Bottom-Up Methodology

In a bottom-up design methodology, we first identify the building blocks that are available to us. We build bigger cells, using these building blocks. These cells are then used for higher-level blocks until we build the top-level block in the design.

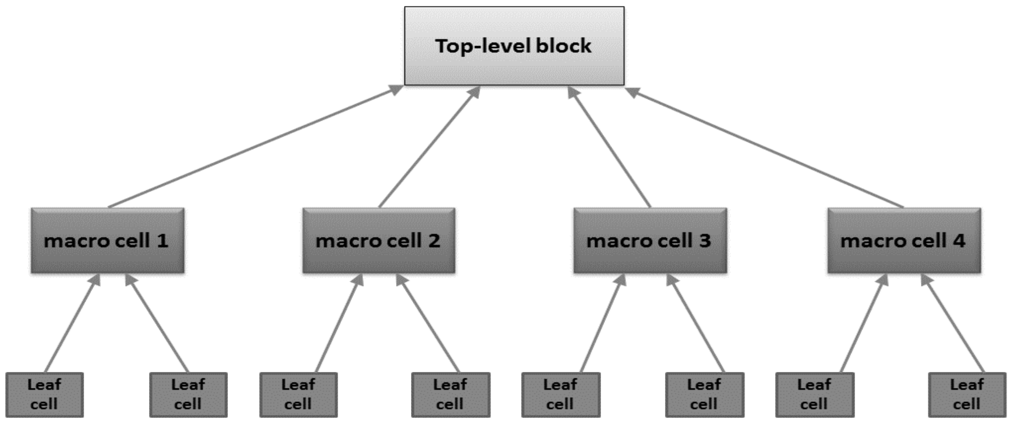


*Figure: Block representation of **Bottom-Up** methodology*

Example 07

The following example demonstrates the Verilog HDL code of a full adder following the **Hierarchical Modeling** style. In the design, the half adder is constructed from the predefined logic gates and then the half adder instance is used twice to design the full adder. This creates two instances in the same module.

```
module Full_Adder(A,B,Cin,sum,carry); // Top module
input A,B,Cin;
output sum,carry;
wire s1,c1,c2;
Half_Adder sm1(s1,c1,A,B);
Half_Adder sm2(sum,c2,s1,Cin);
or o1(carry,c1,c2);
endmodule

module Half_Adder(s,c,x,y);  // macro cell
input  x,y;
output s,c;
xor s1(s,x,y);   // predefined primitive or leaf cells
and c1(c,x,y);
endmodule
```

N.B. One module can be instantiated to another module without maintaining the I/O sequence using the **Namely Wise Instantiation** method (.*currentmodule_variable*(*submodule_variable*)).

Blocking and Non-Blocking Assignment

Blocking **(=)** and non-blocking **(<=)** assignments are provided to control the execution order within an always block. All the previous examples of combinational circuits used blocking assignments. But if the subsequent assignments depend on the results of preceding assignments non-blocking assignments needed to be used. The following examples demonstrates the use of blocking and non blocking assignments.

Example 08

In the following example, we have tried to design a shift register module named **shift_reg** using the blocking assignment.

```
module shift_reg(clock,W,Q);
input clock,W;
output reg[3:0]Q;
always@(posedge clock)
begin
        Q[3]=w;
        Q[2]=Q[3];
        Q[1]=Q[2];
        Q[0]=Q[1];
end
endmodule
```

Now let us try to realize the output of Example 07 for that let us consider Initially Q=0000 and W=1. Now for the first two positive edges of the clock, the output will be following.

```
Output
//After the first positive edge of the clock
        Q[3]=W=1;
        Q[2]=Q[3]=1;
        Q[1]=Q[2]=1;
        Q[0]=Q[1]=1;
//After the second positive edge of the clock
        Q[3]=W=1;
        Q[2]=Q[3]=1;
        Q[1]=Q[2]=1;
        Q[0]=Q[1]=1;
```

Now from the output, we can notice that the output is always the same. For a shift registrar, we know that the output will propagate bit-wise sensing each clock trigger but in the design of Example 08 that is absent due to the use of blocking assignment as the variable update is executed in the order they are coded. It should be noted that the blocking assignment blocks the execution of the next statement till the current statement is executed. So, it can be said that blocking assignment is useful for combinational circuits.

Example 09

In the following example, we have modified the **shift_reg** module of Example 08 by replacing the **blocking** assignment with "**non-blocking**".

```
module shift_reg(clock,W,Q);
input clock,W;
output reg[3:0]Q;
always@(posedge clock)
begin
	Q[3]<=w;
	Q[2]<=Q[3];
	Q[1]<=Q[2];
	Q[0]<=Q[1];
end
endmodule
```

Now let us try to realize the output of Example 08 for that let us consider Initially Q=0000 and W=1. Now for the first two positive edges of the clock, the output will be following.

Output

```
//After the first positive edge of the clock
	Q[3]=W=1
	Q[2]=Q[3]=0
	Q[1]=Q[2]=0
	Q[0]=Q[1]=0
//After the second positive edge of the clock
	Q[3]=W=0
	Q[2]=Q[3]=1
	Q[1]=Q[2]=0
	Q[0]=Q[1]=0
```

Now from the output, we can notice that the output is propagating bit-wise by sensing each clock trigger after using the blocking assignment as the variable update process is executed in parallel. In this code execution of the next statement is not blocked due to the execution of the current statement. This method is useful for modeling sequential circuits and generating concurrent statements.

There are three types of assignments in Verilog, **continuous** *(assign)*, **blocking** *(=)*, and **non blocking** *(<=)*.

Example 10

The following example demonstrates the Verilog HDL code of a D Latch

```
module D_FF(clock,D,Q);
input clock,D;
output reg Q;
always@(*)
      if(clock)
          Q<=D;
endmodule
```

Example 11

The following example demonstrates the Verilog HDL code of a D flip-flop. A D flip-flop is a 1-bit data storage device that saves one-bit data depending on its input D and clock pulse. When a clock edge is triggered, whatever input is present in D goes to the output Q.

```
module D_FF(clock,D,Q);
input clock,D;
output reg Q;
always@(posedge clock)
       Q<=D;
endmodule
```

Example 12

The following example demonstrates the Verilog HDL code of a 4 to 2 priority encoder with a valid bit. In the example, the **casex** statement is used. In Verilog, there are three types of variations in case. The **case**, **casex** and **casez** all do bit-wise comparisons between the selecting *case expression* and individual case item statements. In the **case** statement, the values **x** or **z** in an alternative are checked for an exact match with the same values in the controlling expression. On the other hand, **casex** ignores any bit position containing an 'x' or 'z'. The **casez** statement only ignores bit positions with a 'z'.

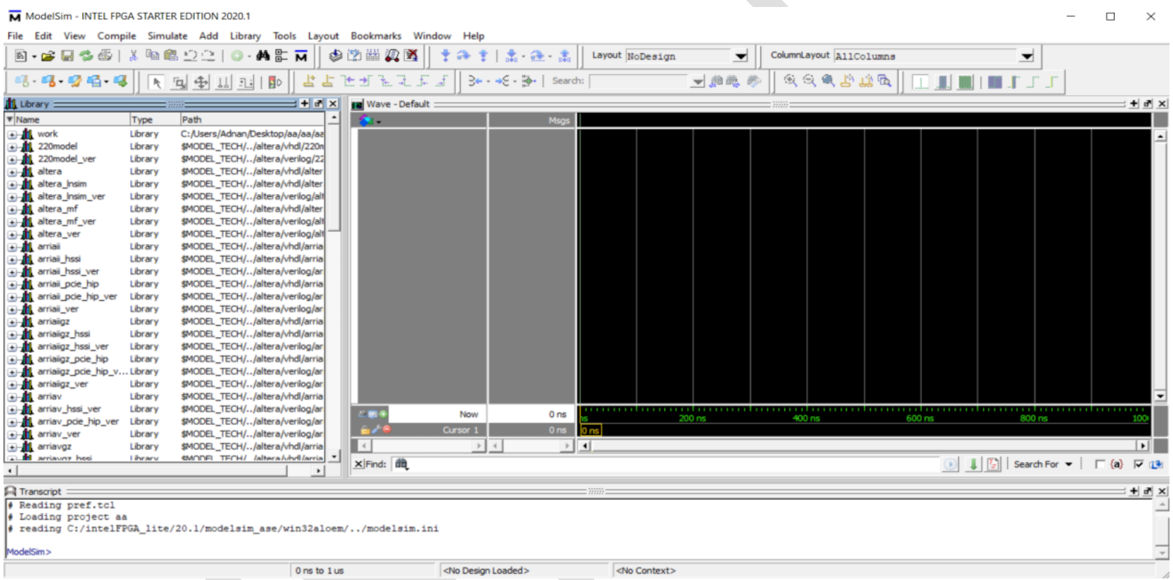
```
module p_encoder_4to2(D,Y,V);
input [3:0]D;          //declaring variable for input
output reg [1:0]Y;  //declaring variable for output
output reg V;        //declaring the variable for valid bit
always@ *
begin
          casex(D)
                    4'b0001:
                    begin
                              Y=2'b00; V=1;
```

```
            end
            4'b001x:
            begin
                    Y=2'b01; V=1;
            end
            4'b01xx:
            begin
                    Y=2'b10; V=1;
            end
            4'b1xxx:
            begin
                    Y=2'b11; V=1;
            end
            default:
            begin
                    Y=2'bx; V=0;
            end
    endcase
end
endmodule
```

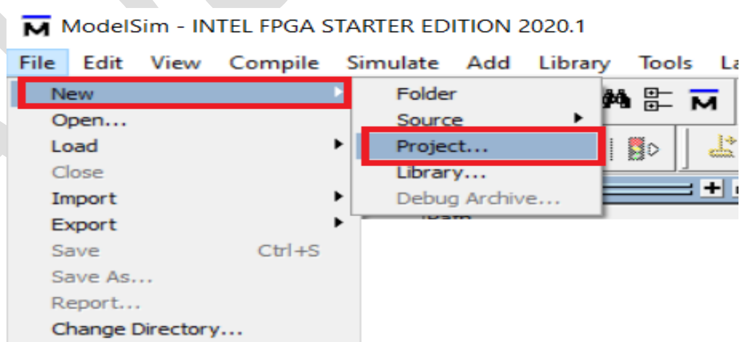
Simulating Verilog HDL

1. Find the following icon on your PC and double-click on the icon to run the software. (**Search:** *ModelSim - Intel FPGA Starter Edition Model Technology ModelSim - Intel FPGA Edition vsim 2020.1 (Quartus Prime 20.1))*

2. The following window will pop up.

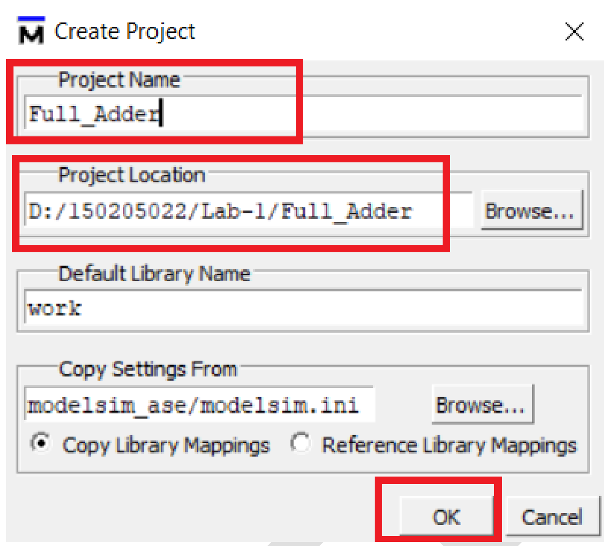


3. Execute **File → New → Project.** The **Create Project** window will appear.

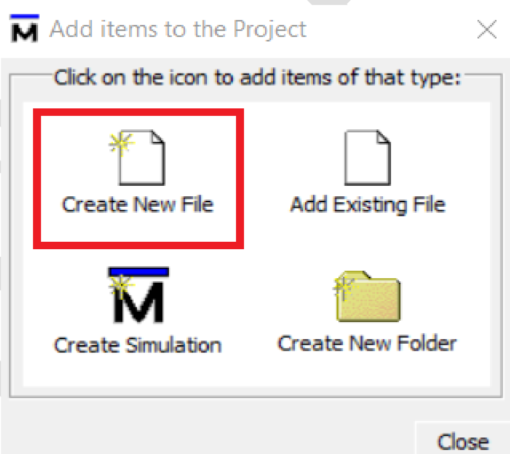


4. In the **Create Project** window change the **Project Location** to your directory (e.g. D:/150205022/Lab-1/Full_Adder) and give a name in the **Project Name** field. After that click on the **OK** button.

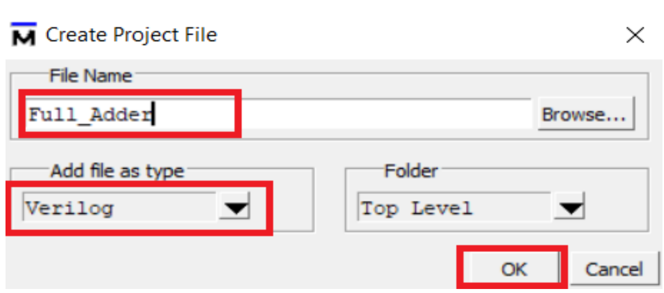
[Project name must be same as the top module]



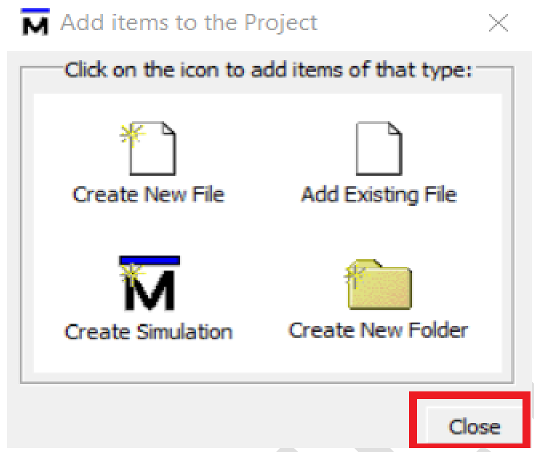
5. The **Add items to the Project** window will appear**.** Select the **Crete New File** button.



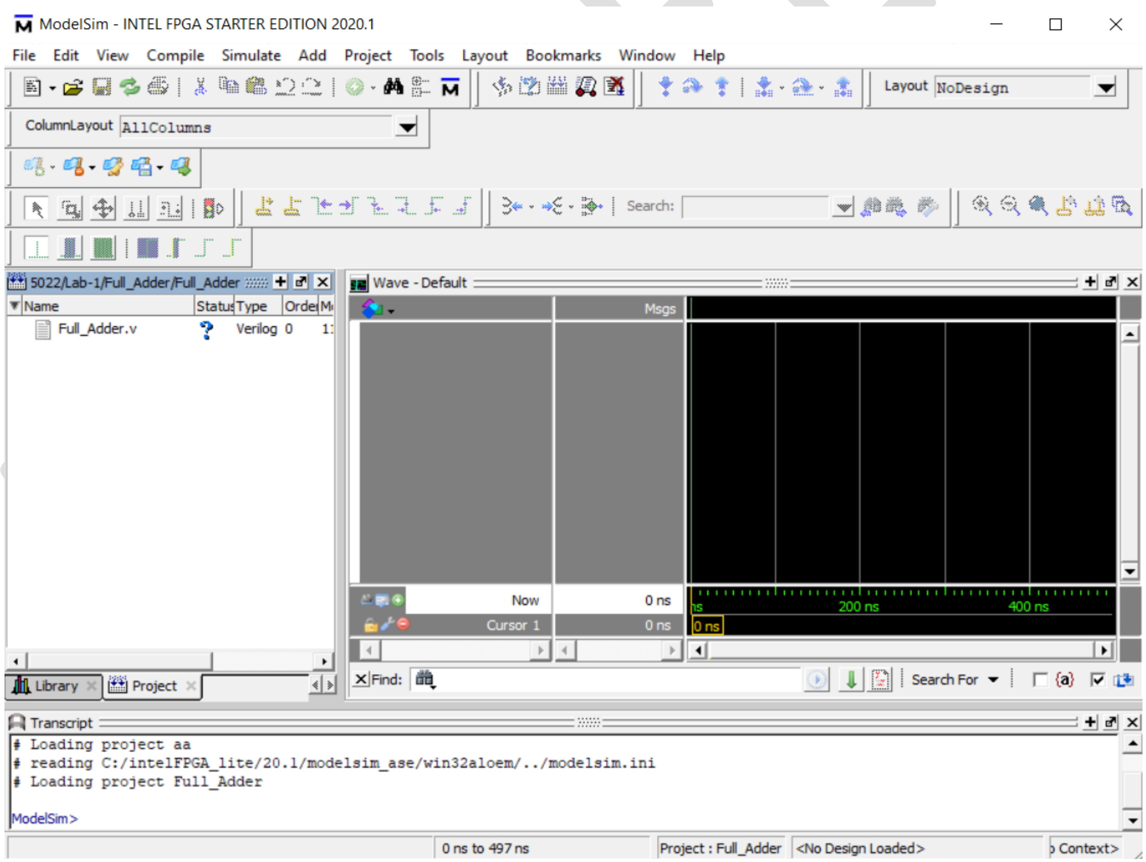
6. In the **Create Project File** window fill up the **File Name** field which must be identical to the project name and top module name. Also, select Verilog from the **Add file as type** dropdown menu. And ten click **OK** button.



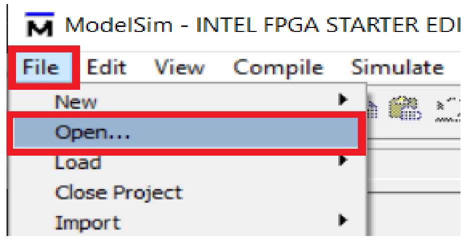
7. The **Add items to the Project** window will appear again**.** Click on the **Close** button.



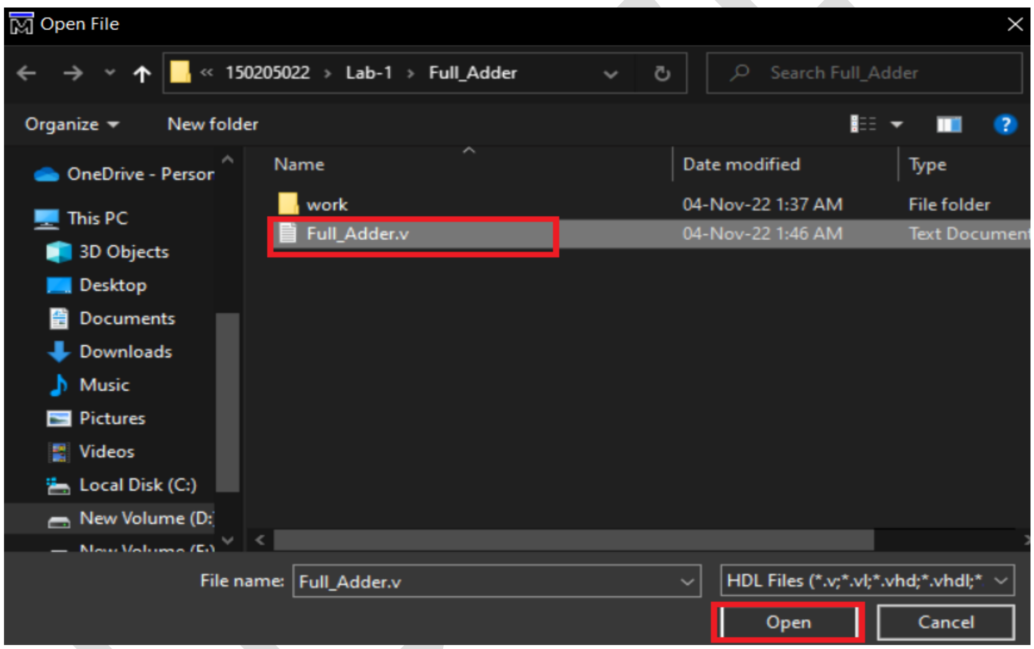
8. Now the ModelSim window will look like the following figure.



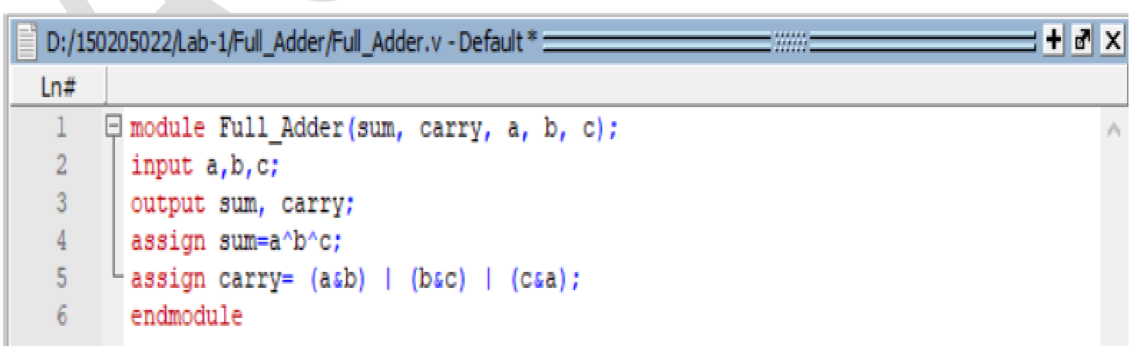
9. Now to open the editor window execute **File → Open…**



10. From the appeared file browser select your Verilog file(.v format)

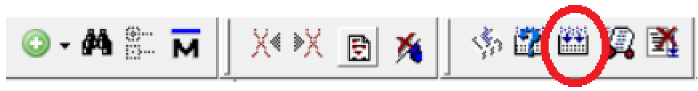


11. In the editor window write the Verilog module of your design. And save using the shortcut executing **Ctrl+S** every time.

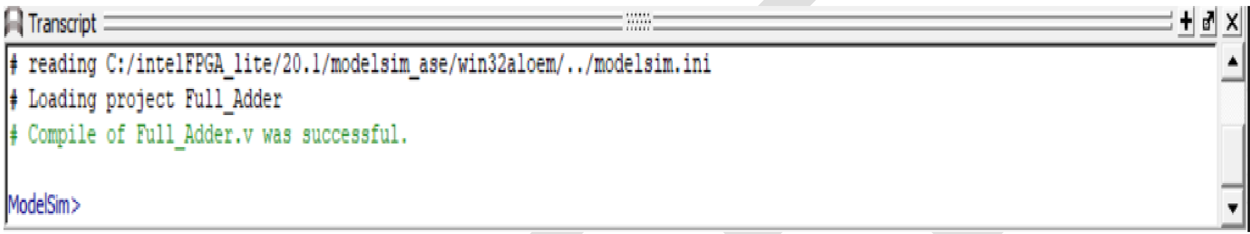


12. Now click on the **Compile All** icon for compiling the design.

[alternatively, execute **Compile → Compile All**]



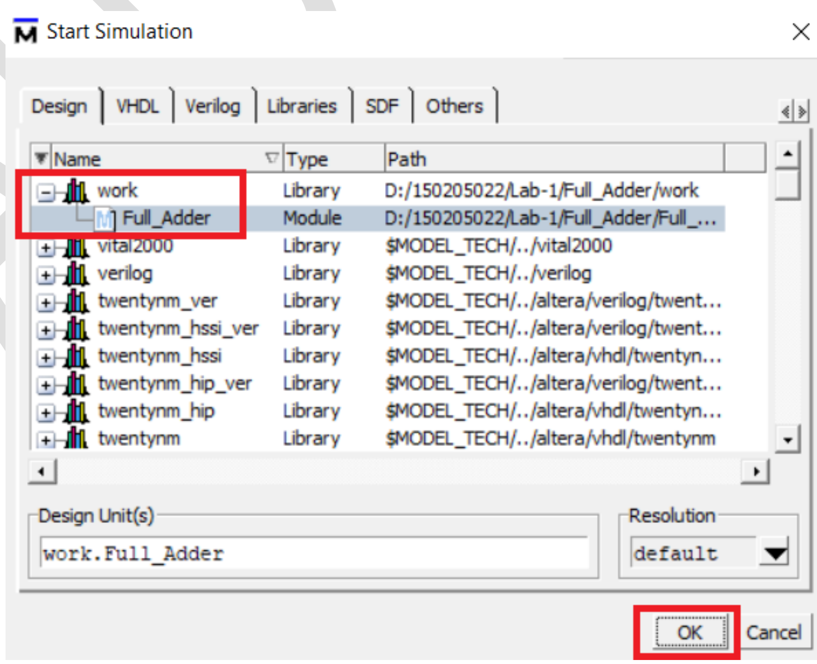
13. After successful compilation you will get the following message will appear in the **Transcript** window.



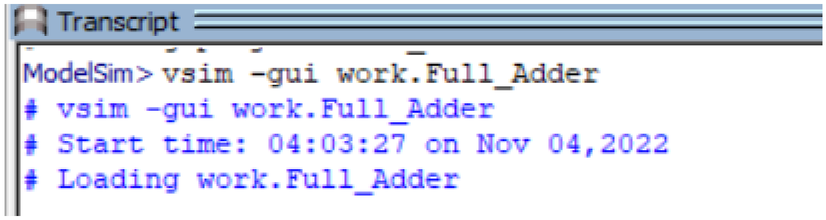
14. Now to simulate the design click on the **Simulate** icon.

[alternatively, execute **Simulate→ Start Simulation..**]

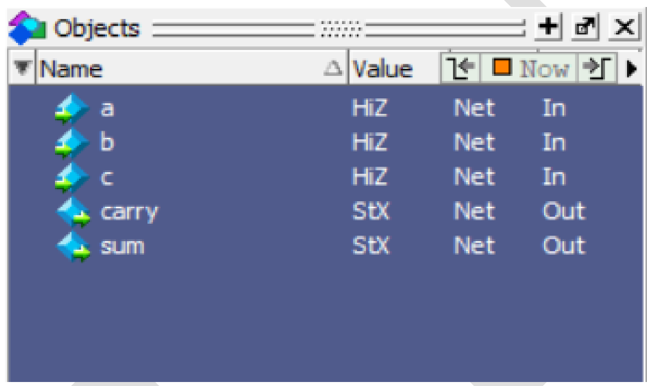
15. The **Start Simulation** window will appear. From the **Design** tab, execute **work → <click on your project module name>** and click on the **OK** button.



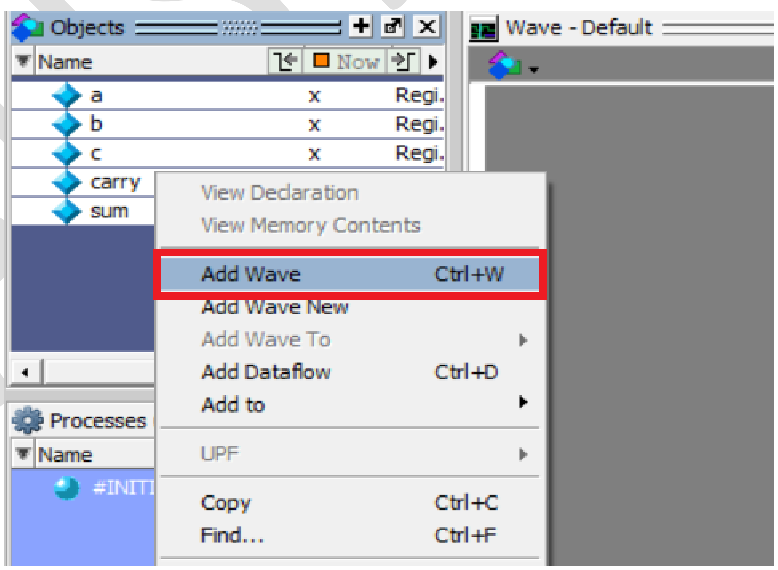
16. The following message will appear in the transcript if everything is done correctly.



17. The input and output variables defined in the Verilog will appear in the **Objects** window.

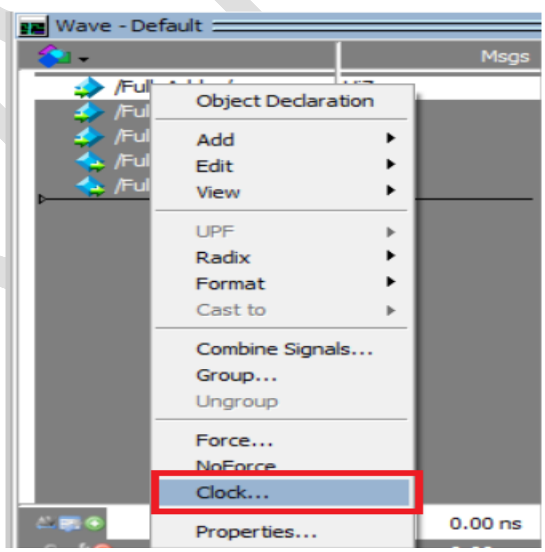


18. Now go to the **Wave** window and select all the input and output variables of the **Objects** window and by right-clicking on your mouse execute **Add Wave** to place them in the **Wave** window.

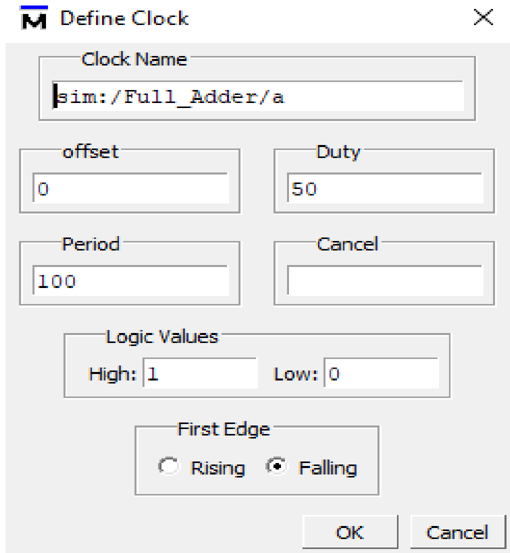


19. All the input and output variables will be placed on the **wave** window and the wave window will look like the following.

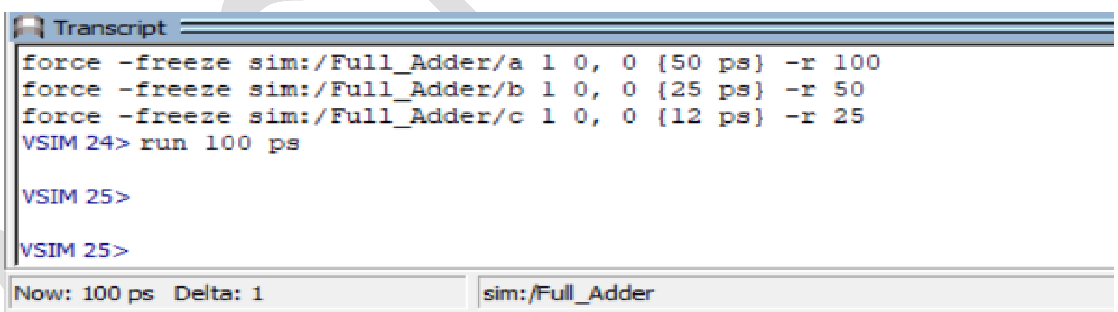
20. Now apply clock to each input variable. Right-clicking any input variable and from the popped-up menu execute **Modify → Clock.**



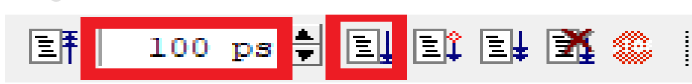
21. The **Define Clock** window will appear. Set parameters as per your requirement keep in mind all the units are in picoseconds by default.



22. After defining all the input clocks, to evaluate the outputs write **run 100 ps** on the **Transcript** of ModelSim. Then the simulation will be performed for 100 ps.

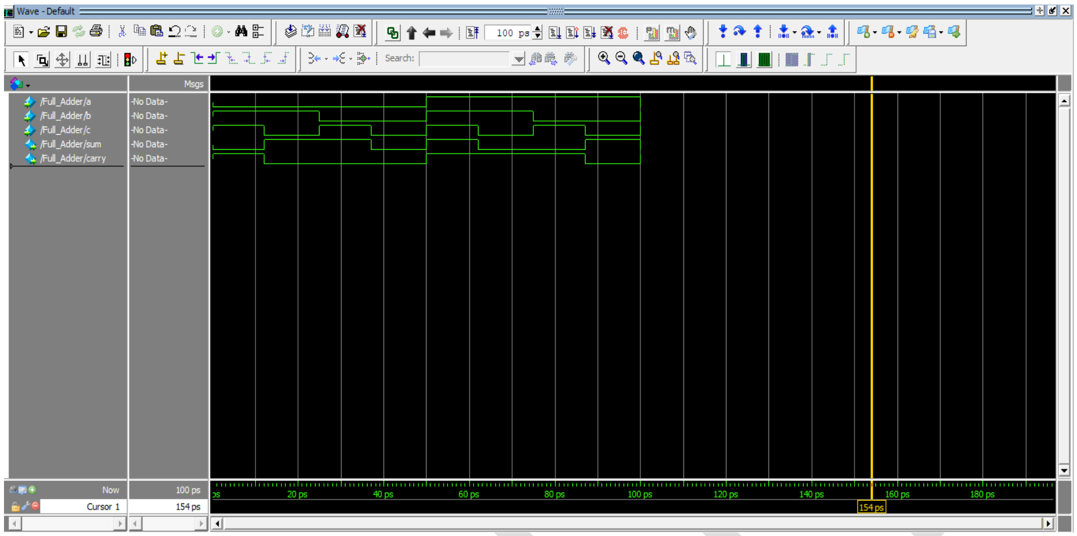


Alternatively, we can run the wave output using the **Run** icon by typing the **Run length**



[Give run length according to your requirement.]

23. The wave window will look like the following figure after simulation.

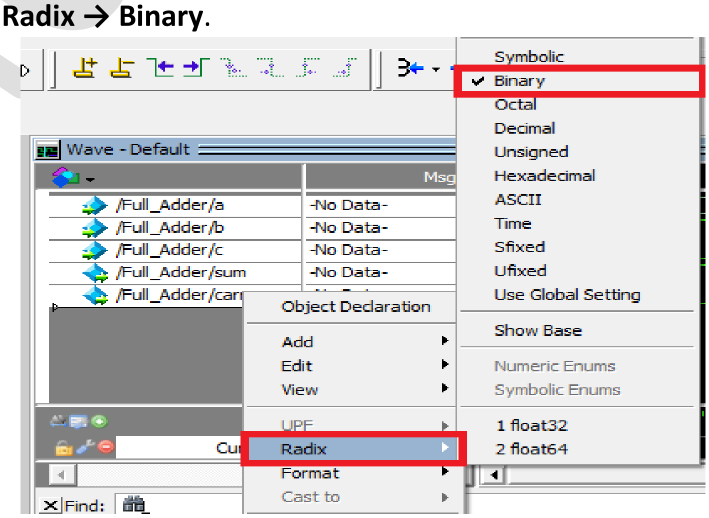


If you need to change the clock pulse you must reset all the clocks before changing clocks otherwise the inputs and outputs will change after the previous run time which is not a convenient way to represent the inputs and outputs. The command **"restart"** is used in the transcript for resetting all the clock. Alternatively, restart can be performed by executing **Simulate → Restart**

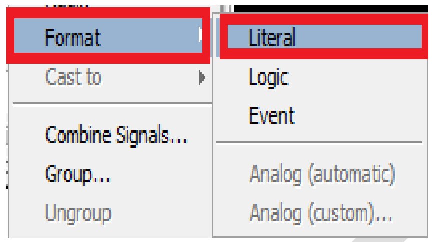
Showing Binary values on the Wave

Sometimes it is hard to verify the functionality of a digital system from the wave. For easy functional verification, we can read the binary values from the wave of ModelSim by doing the following steps.

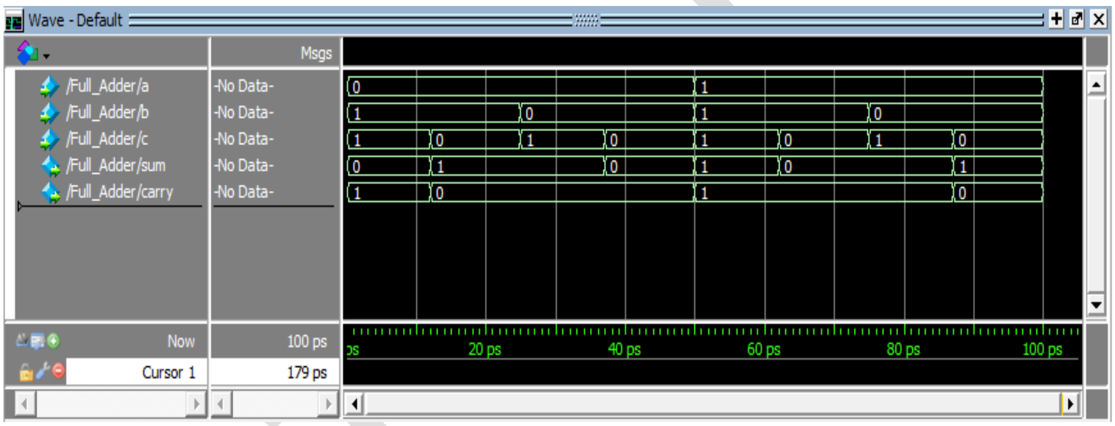
I. Select all the input and output variables on the clock and right-click on the mouse and execute **Radix → Binary**.



II. After changing the **Radix,** change the **Format** type similarly by selecting all input and output variables on the wave by right-clicking on the mouse and then executing **Format → Literal**.

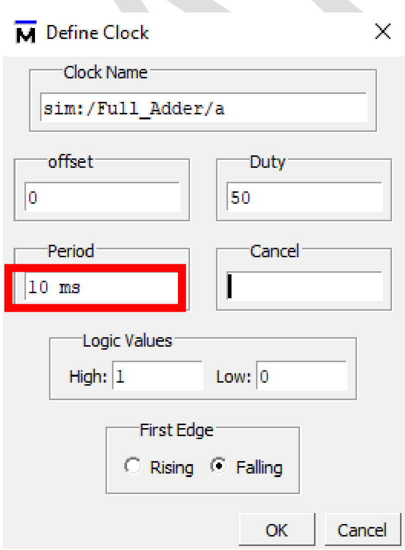


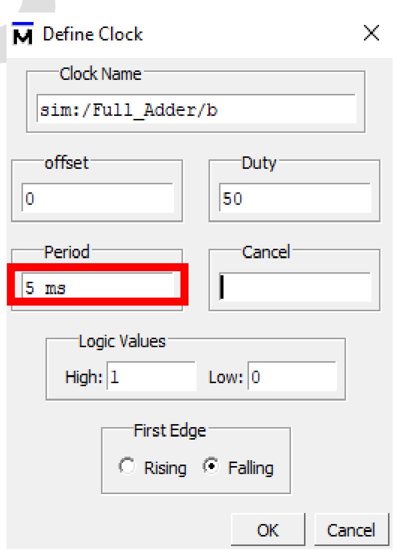
III. Now on the wave, binary values will be displayed which can be easily analyzed.

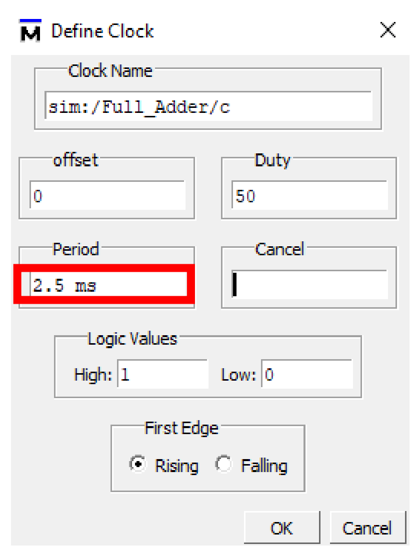


Changing Clock Unit

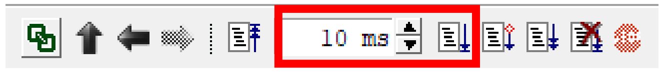
In step 21 it is mentioned that ModelSim's default timing unit is picosecond. But in some cases, we may need to define clocks in other units. Let us consider, that we need to define the period of a, b, and c as 10ms, 5ms, and 2.5ms respectively. Now define the clock a, b, and c as shown in the below figures.



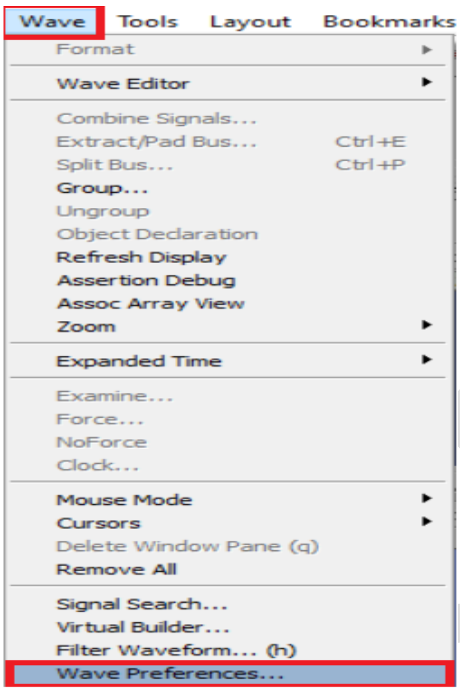


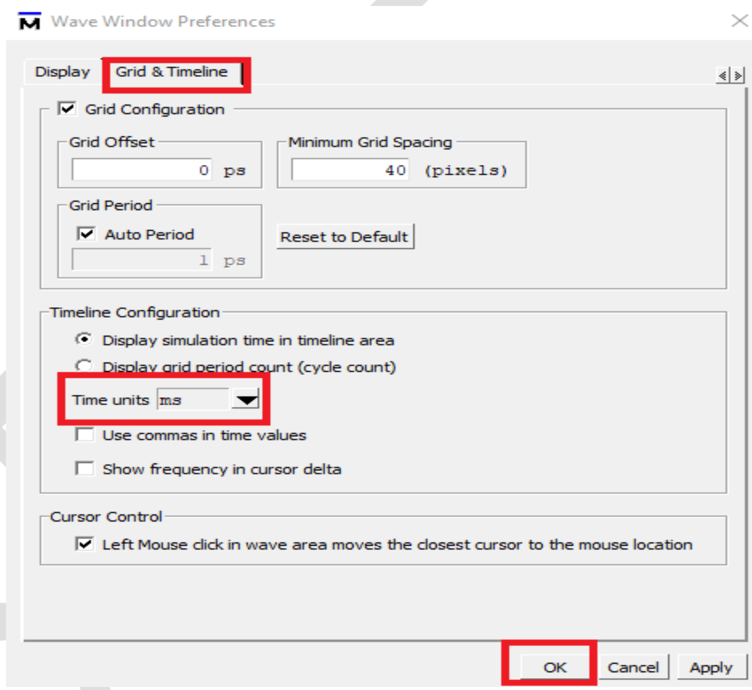


To view output for all the input combinations the run length should be equal to the maximum period.

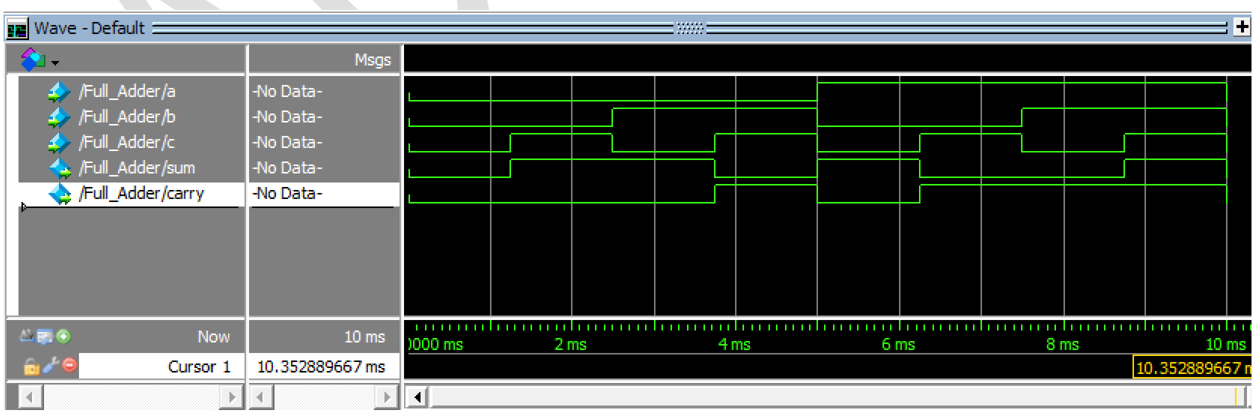


As all the units are in milliseconds, for easy visualization we can change the time units of the wave grid by executing **Wave → Wave Preferences → Grid & Timeline → Time units → ms**.





Now the ModelSim wave window will look like the following figure.



Similarly, for femtoseconds, nanoseconds, and microseconds, we can use **fs**, **ns,** and **ms** respectively

Post Lab Tasks

1. Test the functionality of each example (4-13) using the ModelSim wave.
2. Design three input NOR gate using the switch level abstraction.
3. Design a 4-bit Carry Look Ahead adder using the concept of hierarchical modeling.
4. Design a BCD adder using the behavioural modeling technique.

Lab-2: Introduction to Functional Verification Using Verilog Testbench.

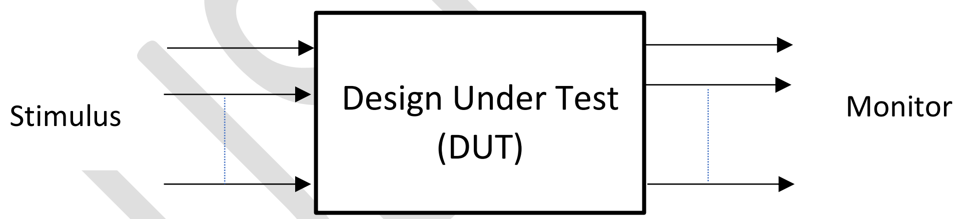
Objective

The main objectives of this lab are:

- Familiarization with test bench module.
- Learning different techniques for generating test vectors
- Verifying combinational circuits imposing test vectors.

Introduction

The test bench is an automated way of verifying and validating a digital design. A test bench is a procedural block that executes only once. Particularly the "initial" procedural block is used for the test bench. Only for repeated clock generation, the "always" procedural block is used. Test bench generates clock, reset, and the required test vectors for a given design under test (DUT) and hence by monitoring the output functionality of the design is verified. During synthesizing a design, a test bench is not required it is required during simulation only.



Block of Design Under Test

Rules of Testbench

I. Define timescale using the command " **`timescale <unit>/<precision**> ".
II. Instantiate the top module in the test bench module.
III. Declare the input and output of design as **"reg"** and **"wire"** type respectively in the test bench module.
IV. Specify the test vectors for different delays using the command **"#<time_delaye>"**.
V. Use **"$display()"** or "**$monitor()"** commands to show outputs for the given test vectors in the transcript.
VI. The "**initia**l" procedural block must be declared at least once.
VII. Terminate testbench using the command "**$finish"**.
VIII. Monitor the outputs for functional verification using the transcript and wave.

Example 01

The following example demonstrates the Verilog HDL code of a half adder.

```
module HA(A,B,S,C);
input A,B;
output S,C;
assign S=A^B;
assign  C=A&B;
endmodule
```

The following Verilog HDL code demonstrates the **Testbench Module** of the half adder of Example 01 for random test inputs.

```
`timescale 1ns/1ps
module HA_TB;
reg a,b;
wire s,c;
HA Ha_dut(a,b,s,c);
initial
begin
        #0 a=0; b=0;
        #5 a=0; b=1;
        #5 a=1; b=0;
        #5 a=1; b=1;
        #5 $finish;
end
endmodule
```

The previous **Testbench Module** of half adder can only generate test vectors for a certain interval but not periodic. The following **Testbench Module** of half adder. The forever loop-like procedural block **“always”** is used to generate periodic inputs.

```
`timescale 1ns/1ps
module HA_TB;
reg a,b;
wire s,c;
HA Ha_dut(a,b,s,c);
initial
begin
        a=0; b=0;
end

always
        #10 a=~a;    // for time period 20 unit
always
```

```
        #5 b=~b; // for time period 10 unit
initial
        #20 $finish;
end
endmodule
```

Example 02

The following example demonstrates the Verilog HDL code of a full adder.

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

The following Verilog HDL code demonstrates the **Testbench Module** of the full adder of Example 02 for random test inputs.

```
`timescale 1ns/100ps
module Full_Adder_TB;
reg a,b,c;
wire sum, carry;
Full_Adder FA_DUT(sum,carry,a,b,c);
initial
begin
        $monitor($time, " a=%b, b=%b, c=%b, sum=%b, carry=%b", a ,b, c, sum, carry);
        #0 a=0; b=0; c=1;
        #5 b=1;
        #5 a=0; b=1; c=1;
        #5 $finish;
end
endmodule
```

Example 03

The following example demonstrates the Verilog HDL code of a 2 to 4 decoder.

```
module decoder_2to4(s,e,y);
input [1:0] s;
input e;
output reg [3:0]y;
integer k;
always@ (s,e)
begin
	for (k=0;k<=3;k=k+1)
		begin
		if ((s==k) && (e==1))
			y[k]=1;
		else
			y[k]=0;
		end
end
endmodule
```

The following Verilog HDL code demonstrates the **Testbench Module** of the 2 to 4 decoder of Example 03.

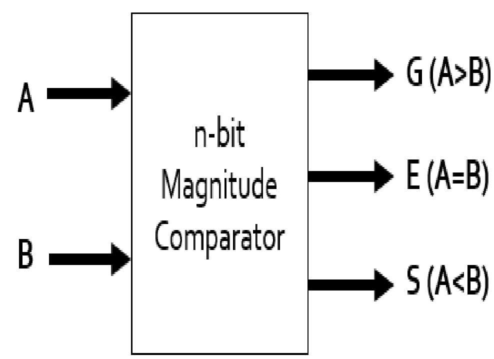
```
`timescale 1ns/1ps
module decoder_2to4_TB;
reg [1:0]s; reg e;
wire [3:0]y;
decoder_2to4 dut(s,e,y);
initial
begin
	$monitor($time, " e=%b, s=%b, y=%b", e ,s, y);
	e=0;
	#5 e=1; s=2'b00;
	#5 s=2'b01;
	#5 s=2'b10;
	#5 s=2'b11;
	#5 s=2'b00;
	#5 s=2'b01;
	#5 s=2'b10;
	#5 s=2'b11;
	#5 $finish;
end
endmodule
```

The following Verilog HDL code demonstrates another **Testbench Module** to verify the 2 to 4 decoder of Example 03 which is efficient than the previous one.

```
`timescale 1ns/1ps
module decoder_2to4_TB;
reg [1:0]s; reg e;
wire [3:0]y;
integer i,j;
decoder_2to4 dut2(s,e,y);
initial
begin
        e=0;
        $monitor($time, "e=%b, s=%b, y=%b", e ,s, y);
        for (j=1;j<=2;j=j+1)
        begin
                for (i=2'b00;i<=2'b11;i=i+1)
                begin
                        #5 e=1; s=i;
                end
        end
        #5 $finish;
end
endmodule
```

Example 04

A magnitude comparator is a combinational circuit that compares the magnitude of two n-bit numbers A and B. The comparison of two numbers is an operation that determines whether one number is greater than, less than, or equal to the other number. The outcome of the comparison is specified by three binary variables G, E, and S that indicate whether A>B, A=B, and A<B respectively. In a magnitude comparator at a time, only one output variable can be logically high.



Block diagram of a magnitude comparator

The following example demonstrates the Verilog HDL code of a 2-bit magnitude comparator. The module has 2 inputs A and B each are 2-bit numbers
When,

A>B outputs G=1, E=0 ,S=0
A=B outputs G=0, E=1, S=0
A<B outputs G=0, E=0, S=1

```
module mag_comp_2bit(A,B,G,E,S);
input [1:0]A,B;             // declaring 2-bit input variables A and B
output reg G,E,S;
always@*                // * symbol means the sensitivity list will be detected automatically
begin
        if (A>B)
        begin
                G=1'b1;
                E=1'b0;
                S=1'b0;
        end
        else if (A==B)
        begin
                G=1'b0;
                E=1'b1;
                S=1'b0;
        end
        else
        begin
                G=1'b0;
                E=1'b0;
                S=1'b1;
        end
end
endmodule
```

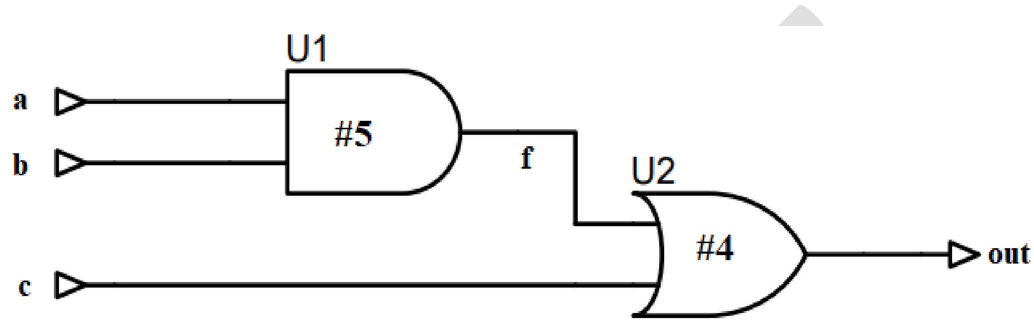
The following Verilog HDL code demonstrates another Testbench module to verify the 2-bit magnitude comparator of Example 04.

```
`timescale 1ns/1ps
module mag_comp_2bit_TB;
reg [1:0]A,B;
wire G,E,S;
integer i,j;
mag_comp_2bit dut(A,B,G,E,S);
initial
begin
        $monitor($time, " A=%b, B=%b, G=%b, E=%b, S=%b", A, B, G ,E, S);
        for (j=2'b00;j<=2'b11;j=j+1)
        begin
                A=j;
                for (i=2'b00;i<=2'b11;i=i+1)
                begin
                        #5 B=i;
                end
        end
```

```
        #0 $finish;
end
endmodule
```

Example 05

The following example demonstrates the Verilog HDL code of delayed gates




```
`timescale 1ns/1ps
module delay_gate(a,b,c,f,out);
input a,b,c;
output out,f;
and #(5) U1(f,a,b);
or  #(4) U2(out,f,c);
endmodule
```

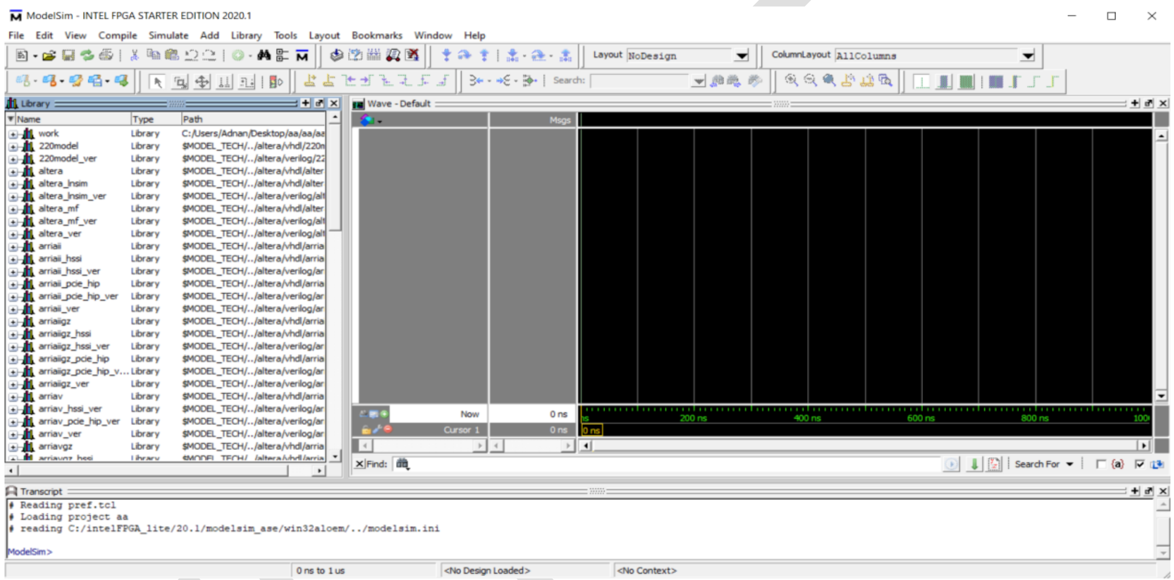
The following Verilog HDL code demonstrates another Testbench module to verify the logic arrangement shown in Example 05.

```
`timescale 1ns/1ps
module delay_gate_TB;
reg a,b,c;
wire out,f;
delay_gate dut(a,b,c,f,out);
initial
begin
        a=1'b0; b=1'b0; c=1'b0;
        #10 a=1'b1; b=1'b1; c=1'b1;
        #10 a=1'b1; b=1'b0; c=1'b0;
        #20 $finish;
end
endmodule
```

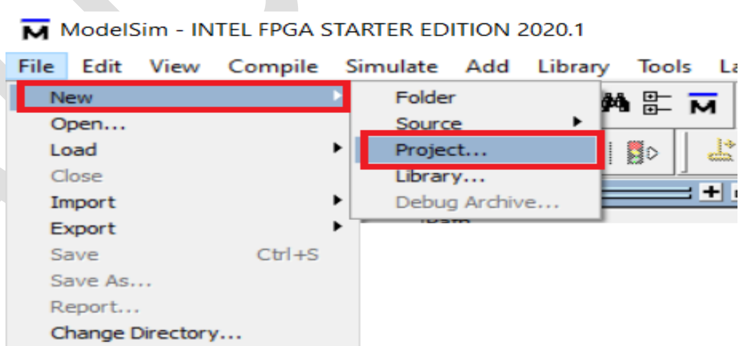
Simulating Testbench

24. Find the following icon on your PC and double-click on the icon to run the software. (*ModelSim - Intel FPGA Starter Edition Model Technology ModelSim - Intel FPGA Edition vsim 2020.1 (Quartus Prime 20.1)*)

25. The following window will pop up.

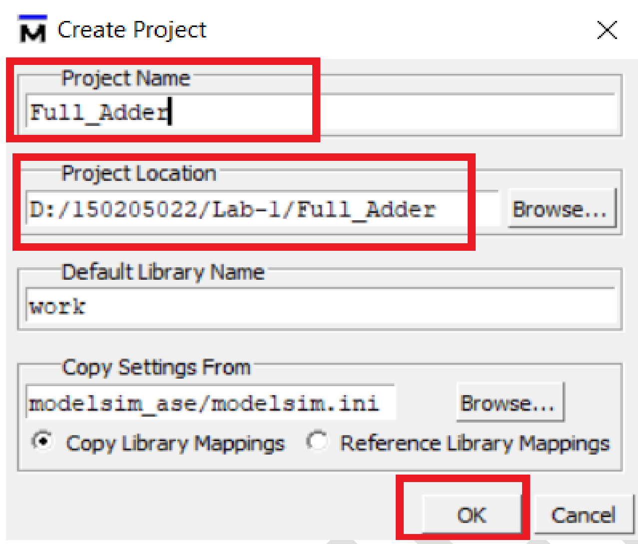


26. Execute **File → New → Project.** The **Create Project** window will appear.

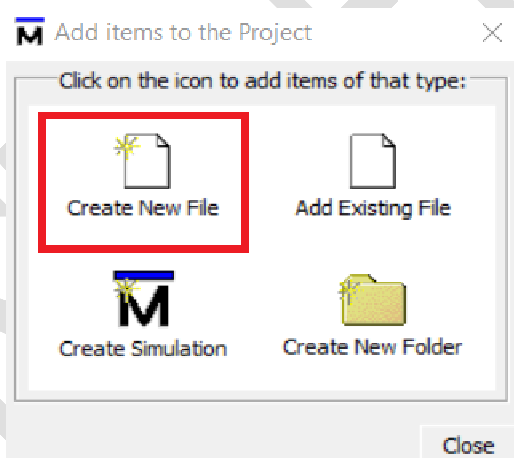


27. In the **Create Project** window change the **Project Location** to your directory (e.g. D:/150205022/Lab-1/Full_Adder) and give a name in the **Project Name** field. After that click on the **OK** button.

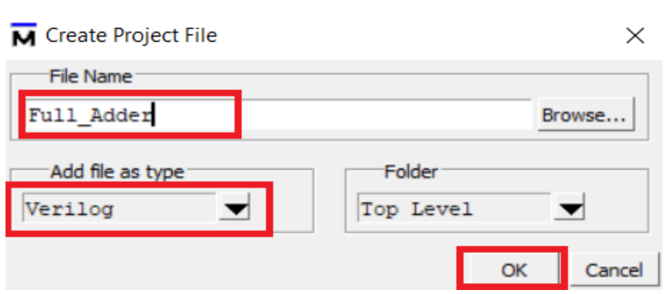
[Project name must be same as the top module]



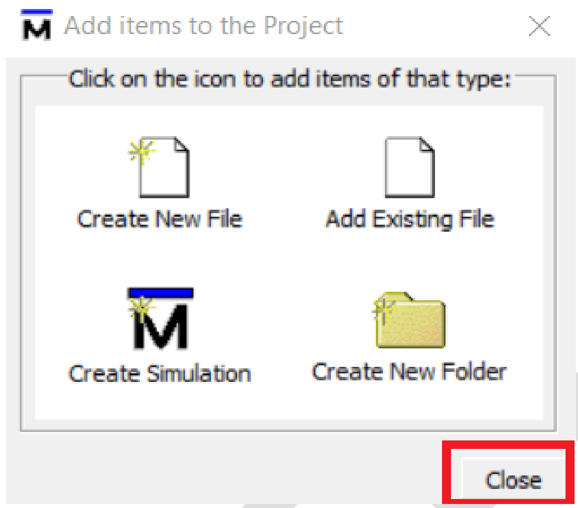
28. The **Add items to the Project** window will appear**.** Select the **Crete New File** button.



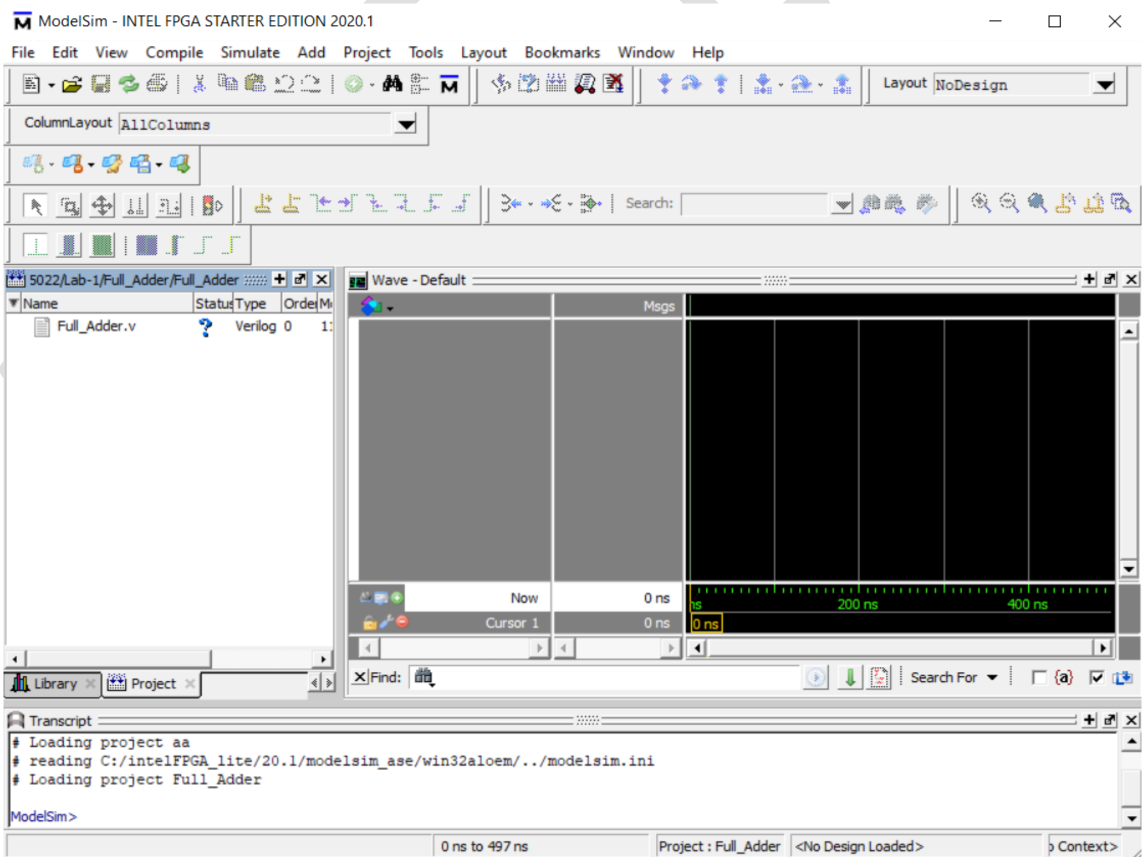
29. In the **Create Project File** window fill up the **File Name** field which must be identical to the project name and top module name. Also, select Verilog from the **Add file as type** dropdown menu. And then click the **OK** button.



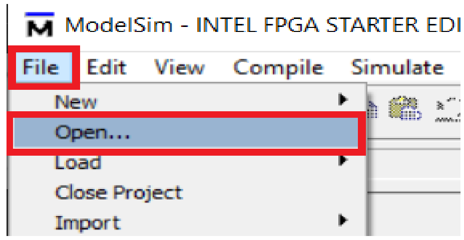
30. The **Add items to the Project** window will appear again**.** Click on the **Close** button.



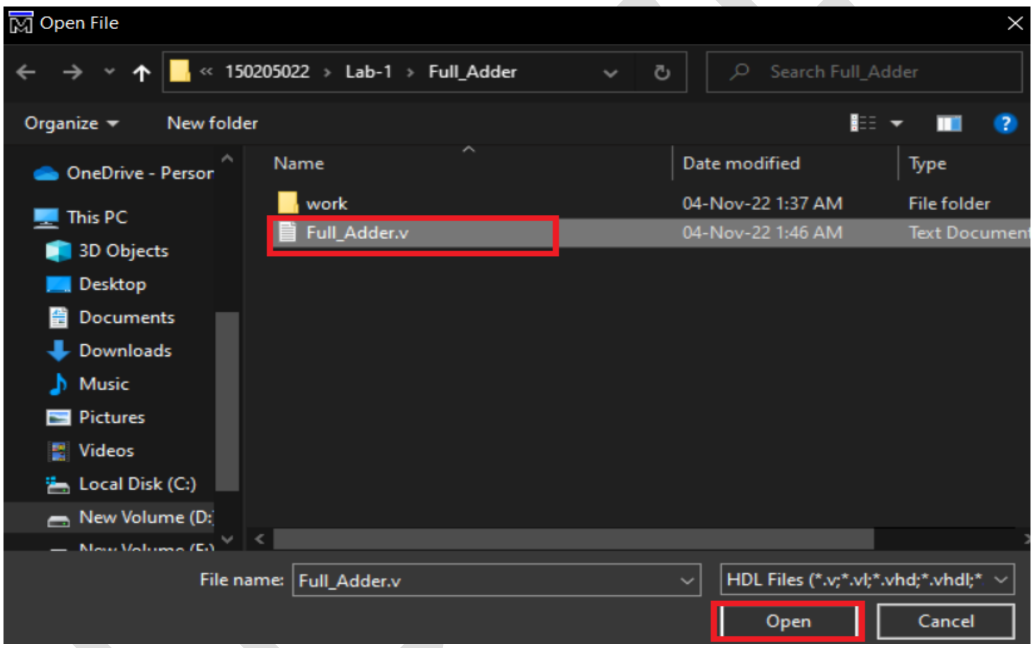
31. Now the ModelSim window will look like the following figure.



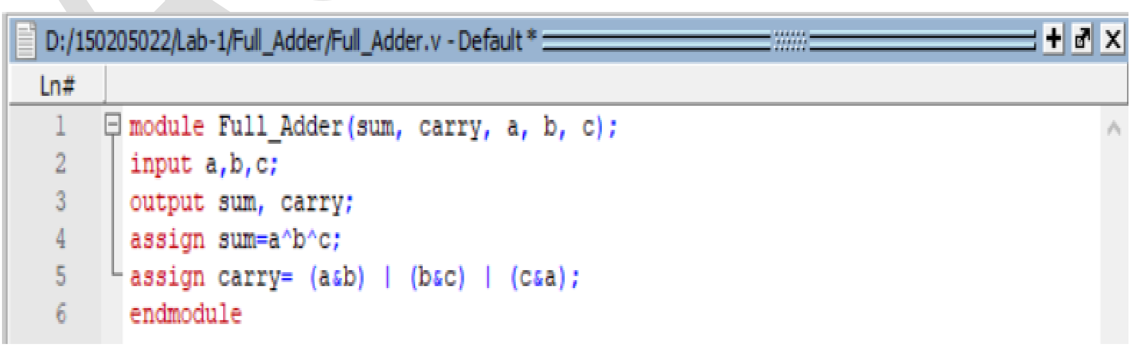
32. Now to open the editor window execute **File → Open…**



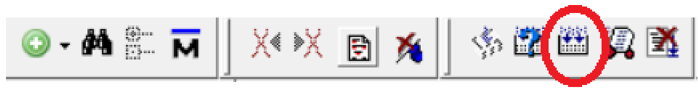
33. From the appeared file browser select your Verilog file (.v format)



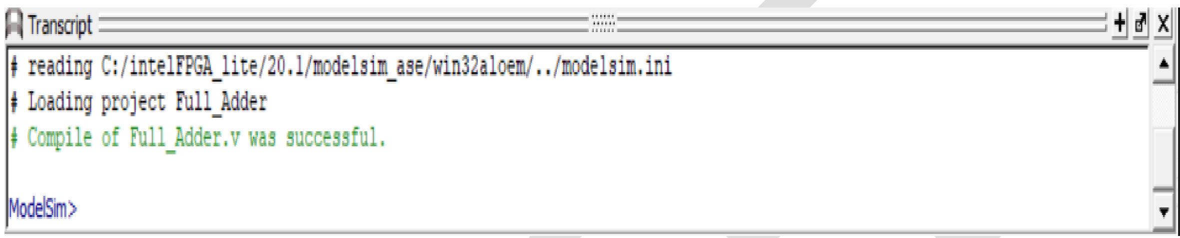
34. In the editor window write the Verilog module of your design. And save using the shortcut executing **Ctrl+S** every time.



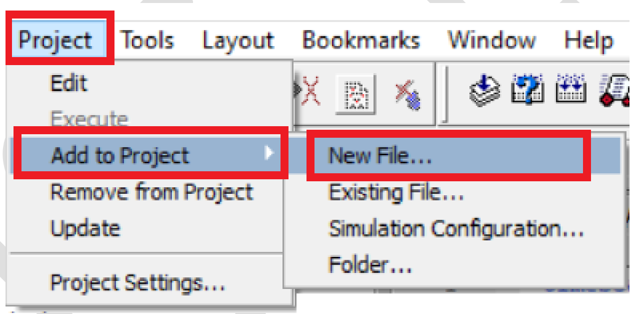
35. Now click on the **Compile All** icon for compiling the design.
[alternatively, execute **Compile → Compile All**]



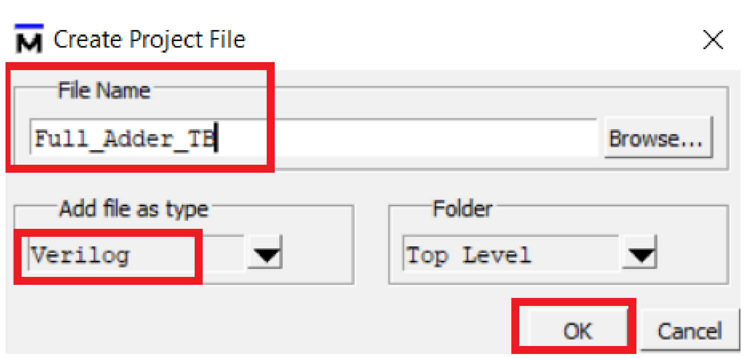
36. After successful compilation you will get the following message will appear in the **Transcript** window.



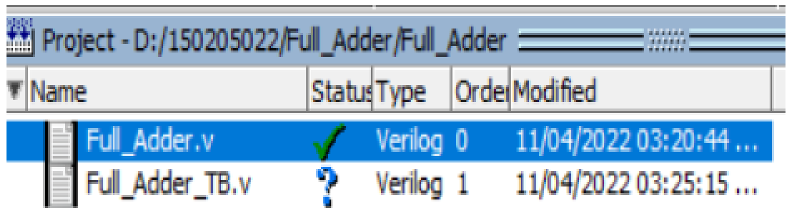
37. Now, to write the test bench code create a new Verilog file at first click on the project window then execute **Project→ Add to Project → New File...**



38. Now in the **Create Project File** window, fill up the **File Name** field which will be our test bench module name. Also, select **Verilog** from the **Add file as type** dropdown menu. And then click the **OK** button.



39. Now there will be two files under the project.



40. Open the testbench file following step 10 and write the testbench code in the editor.

D:/150205022/Full_Adder/Untitled-1.v - Default *

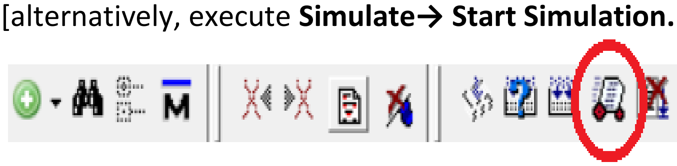
```
`timescale 1ns/100ps
module Full_Adder_TB;
reg a,b,c;
wire sum, carry;
Full_Adder FA_DUT(.sum(sum),.carry(carry),.a(a),.b(b),.c(c));
initial
begin
        $monitor($time, " a=%b, b=%b, c=%b, sum=%b, carry=%b", a ,b, c, sum, carry);
        #0 a=0; b=0; c=1;
        #5 b=1;
        #5 a=1; b=1; c=1;
        #5 $finish;
end
endmodule
```

41. Now click on the **Compile All** icon for compiling the design.

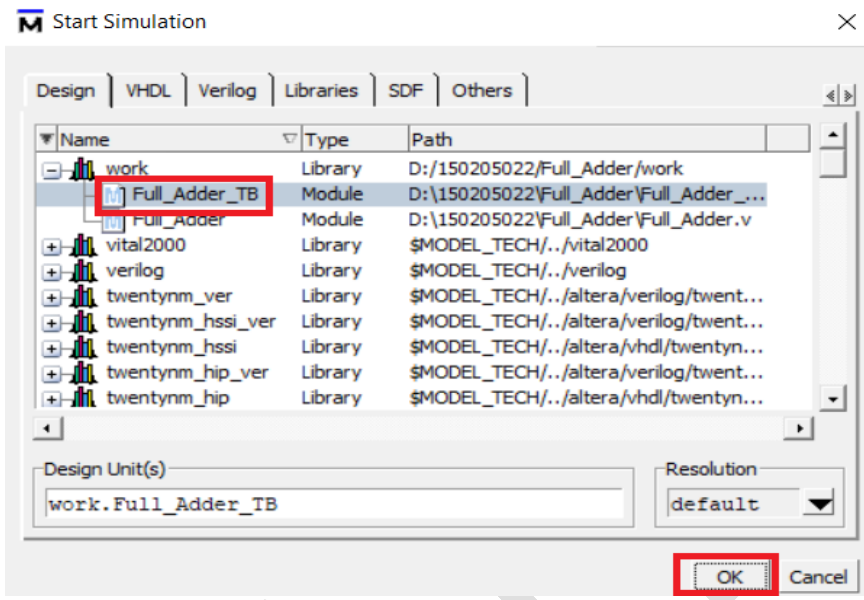
[alternatively, execute **Compile → Compile All**]

42. Now to simulate the design click on the **Simulate** icon.

[alternatively, execute **Simulate→ Start Simulation..**]



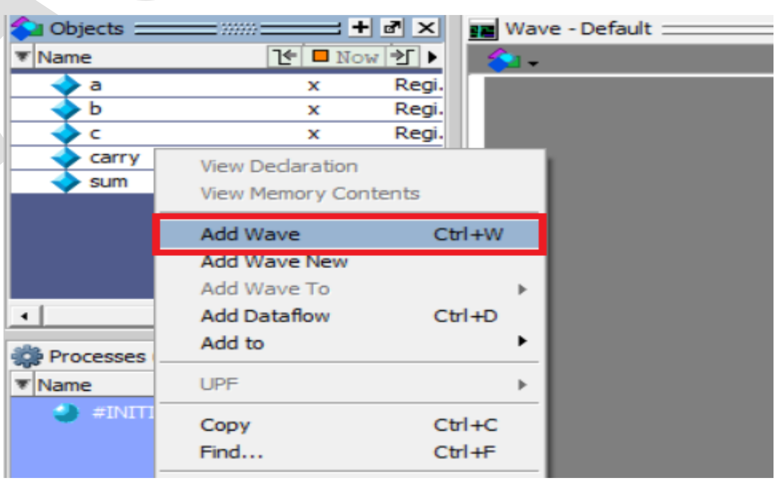
43. The **Start Simulation** window will appear. From the **Design** tab execute **work → <click on your test bench module>** and click on the **OK** button.



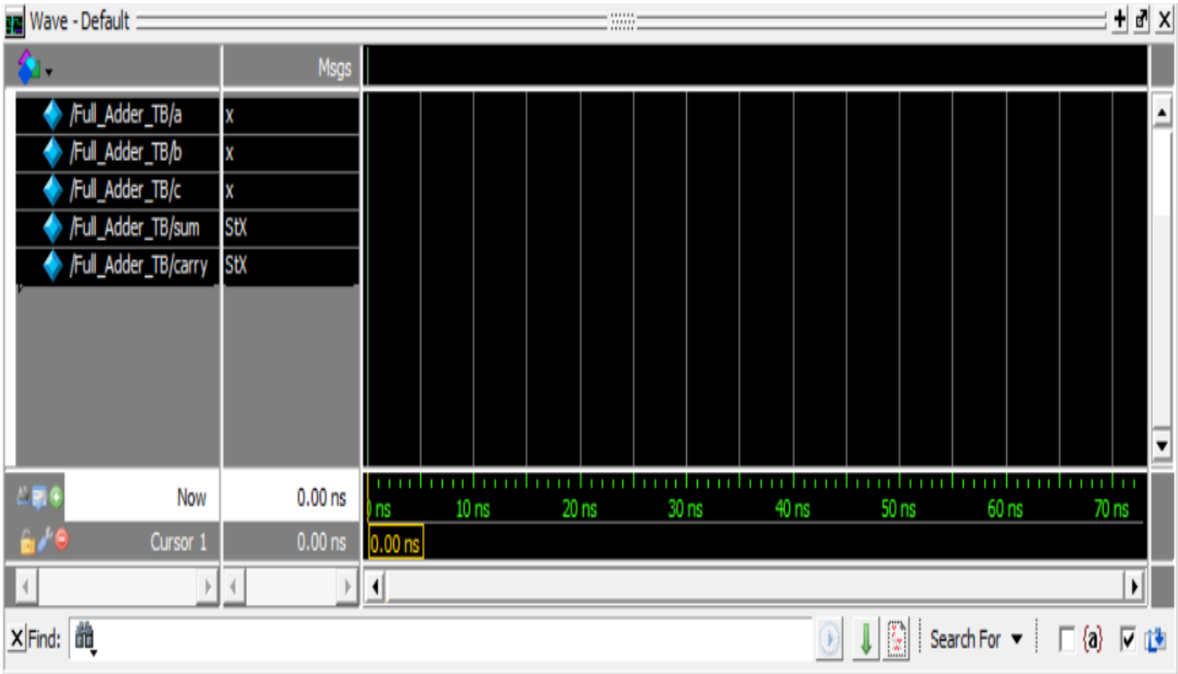
44. The following message will appear in the transcript if everything is done correctly.

```
# End time: 03:44:35 on Nov 04,2022, Elapsed time: 0:04:38
# Errors: 0, Warnings: 2
# vsim -gui work.Full_Adder_TB
# Start time: 03:44:35 on Nov 04,2022
# Loading work.Full_Adder_TB
# Loading work.Full_Adder
```

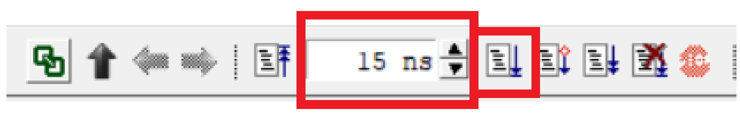
45. Graphically the functionality of the design can be checked from the wave window of the **ModelSim** Simulator. Execute **view → wave** if it doesn't appear automatically. Now go to the **Wave** window and select all the input and output variables of the **Objects** window and by right-clicking on your mouse execute **Add Wave** to place them in the **Wave** window.



46. All the input and output variables will be placed on the **wave** window and the wave window will look like the following figure.

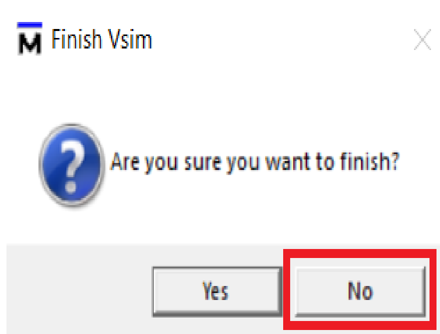


47. Now to evaluate the outputs write **run 15 ns** on the Transcript of ModelSim. Alternatively, we can run the wave output using the **Run** icon by typing the **Run length.**

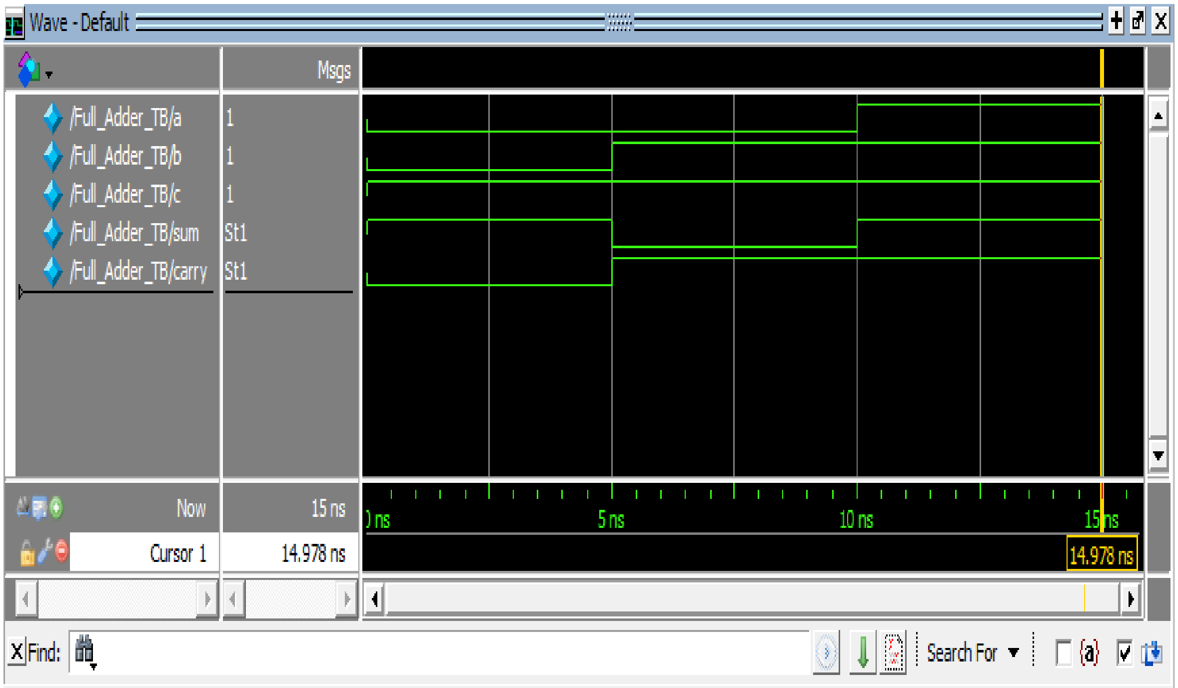


[Give run length according to your requirement.]

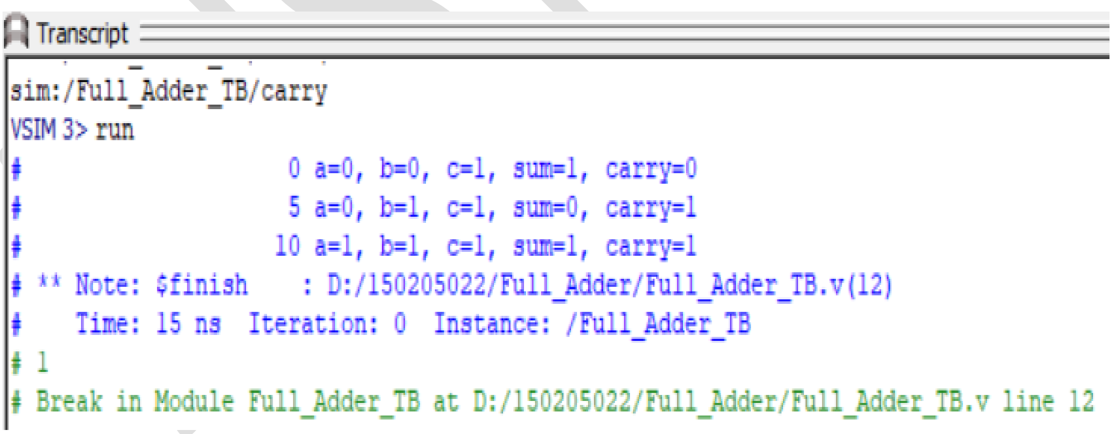
48. The **Finish Vsim** window will appear. Click **No** otherwise the ModelsSim will be closed immediately.



49. Now for the given test vectors the functionality of the design can be verified from the wave output generated by the **ModelSim** simulator.

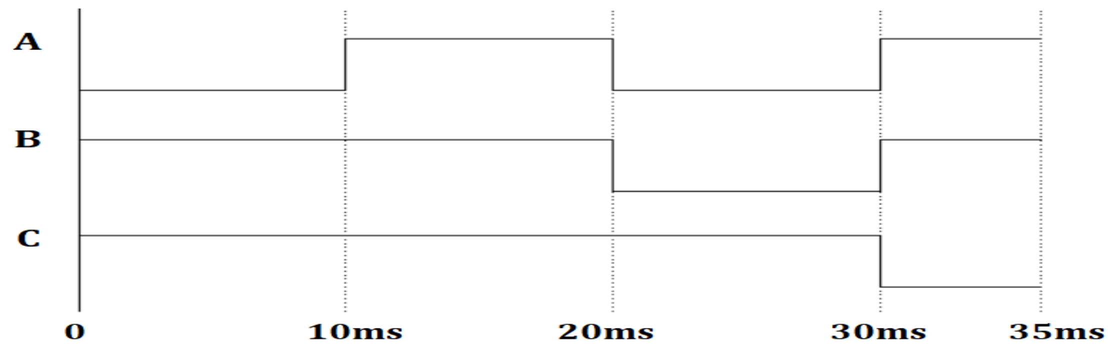


50. Functionality of the design can also be verified from the transcript generated by the **ModelSim** simulator. Execute **view → Transcript** if it doesn't appear automatically.



Post Lab Tasks

1. Write a testbench program to test a full adder circuit with the signal shown below.



2. Differentiate between -
   a. **$finish** and **$stop** command.
   b. **$monitor** and **$display** command.
3. Is it possible to check the functionality of a sequential circuit from the transcript only?
4. Can we use the **"always"** procedural block in the Testbench module?
5. Is it possible to generate periodic stimuli in the testbench? If possible, generate the signals of task-1 for two periods.

Lab-3: Modeling Sequential Systems and Finite State Machine Using Verilog HDL

Objective

The main objectives of this lab are:

- Functional verification of sequential circuits using Verilog Testbench.
- Modeling finite state machine and its functional verification using Verilog Testbench.

Introduction

A digital system can be either in the form of combinational logic or sequential logic. In combinational logic, the output of a circuit depends only on the presently applied inputs. On the other hand, the output of a sequential circuit depends on the applied input and the present states. Most practical digital systems are sequential. To design a digital system, the behavioral abstraction is used as a reference to create and refine a synthesizable register transfer level (RTL) abstraction that captures the desired functionality required by the design specification.

Example 01

Flip-flops are the building blocks of sequential circuits. In the following example, the Verilog HDL code of a positive edge-triggered T flip-flop with reset is demonstrated.

```
module T_FF(T,clk,reset,Q);
input T,clk,reset;
output reg Q;
always@(posedge clk)
begin
if(reset==0)
begin
   if (T)
      Q<=~Q;
   else
      Q<=Q;
end
else
   Q<=0;
end
endmodule
```

Testbench Module of Example 01

The following Verilog HDL code demonstrates the **Testbench Module** of the T flip-flop of Example 01.

```
`timescale 1ns/1ps
module T_FF_TB;
reg T,clk,reset;
wire Q;
T_FF dut(T,clk,reset,Q);
initial
begin
	T=0; clk=0;  reset=1;
end
always
	#2 clk=~clk;
initial
begin
	#6 reset=0; T=1;
	#4 reset=1; T=1;
	#4 reset=1; T=0;
	#2 reset=0;  T=0;
	#2 $finish;
end
endmodule
```

Example 02

The following example demonstrates the Verilog HDL code of a positive edge-triggered JK flip-flop with clear.

```
module JK_FF(clk,J,K,Q,clear);
input clk,J,K,clear;
output reg Q;
always@ (posedge clk)
begin
if(clear==0)
begin
	if (J==0 && K==0)
		Q<=Q;
	else if (J==0 && K==1)
			Q<=0;
	else if (J==1 && K==0)
		Q<=1;
	else
```

```
                Q<=~Q;
end
else
        Q=0;
end
endmodule
```

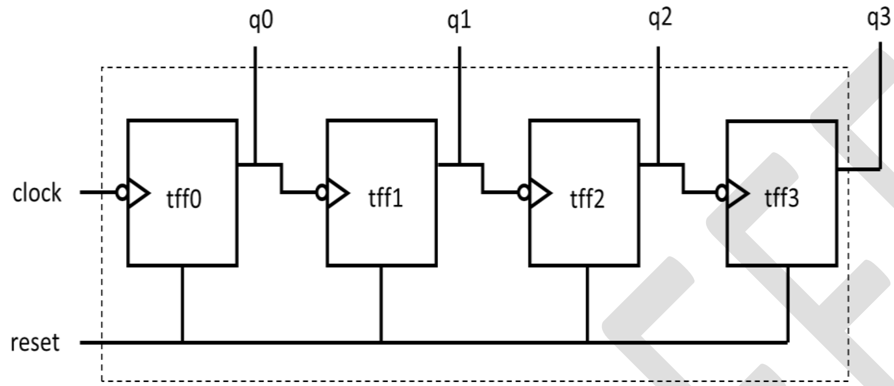
Testbench Module of Example 02

The following Verilog HDL code demonstrates the **Testbench Module** of the JK flip-flop of Example 02.

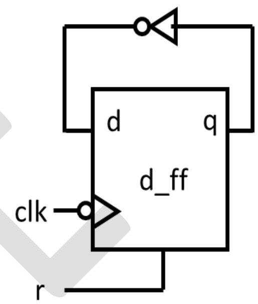
```
`timescale 1ns/1ps
module JK_FF_TB;
reg clk,J,K,clear;
wire Q;
JK_FF JK_dut(clk,J,K,Q,clear);
initial
begin
        clk=0; J=0; K=1;clear=0;
end
always
        #2 clk=~clk;
initial
begin
        #2 clear=1; J=1; K=0;
        #4 clear=0; J=0; K=1;
        #4 J=1;
         #4 J=0;
        #4 $finish;
End
endmodule
```

Example 03

In this example a 4-bit ripple carry counter will be designed using the submodule of a T flip-flop and each T filp-flop is designed using leaf module of D flip-flop. The block representation of the ripple carry counter is shown below.



4-bit ripple carry counter



T flip flop using D flip flop

The following example demonstrates the Verilog HDL code of a 4-bit asynchronous ripple-carry country as shown in the following block diagram.

```
module rc_counter(q,clock,reset);
output [3:0] q;
input clock,reset;
t_ff tff0 (q[0], clock, reset);
t_ff tff1 (q[1], q[0], reset);
t_ff tff2 (q[2], q[1], reset);
t_ff tff3 (q[3], q[2], reset);
endmodule

module t_ff (q,clk,r);//T-Flip-Flop
output q;
input clk,r;
wire d;
d_ff dff1(q,d,clk,r);
not n1(d,q);
endmodule

module d_ff (q,d,clk,r);//D-Flip-Flop
output reg q;
input d,clk,r;
always @(posedge r or negedge clk)
begin
        if (r)
                q<=1'b0;
```

```
          else
                    q<=d;
end
endmodule
```

Testbench Module of Example 03

The following Verilog HDL code demonstrates the **Testbench Module** of the 4-bit asynchronous ripple-carry counter of Example 03.

```
`timescale 1ns/1ps
module rc_counter_TB;
reg clk, res;
wire [3:0]q;
rc_counter rc_counter_dut(q,clk,res);
initial
begin
          clk=0;
end
always
          #5 clk=~clk;
initial
begin
          $monitor($time, " clk=%b, res=%b, q=%b", clk, res, q);
          res=1;
          #15   res=0;
          #180  res=1;
          #10   res=0;
          #20  $stop;
end
endmodule
```

Example 04

The following example demonstrates the Verilog HDL code of a simple 8-bit accumulator. The module is designed in such a way that when reset=0 the output is set to 0 and when reset=1 the output adds the input.

```
module accu(in, acc, clk, reset);
input [7:0] in;
input clk, reset;
output reg [7:0]acc;
always @(posedge clk)
begin
      if (reset)
```

```
			acc<=0;
		else
			acc<=acc+in;
end
endmodule
```

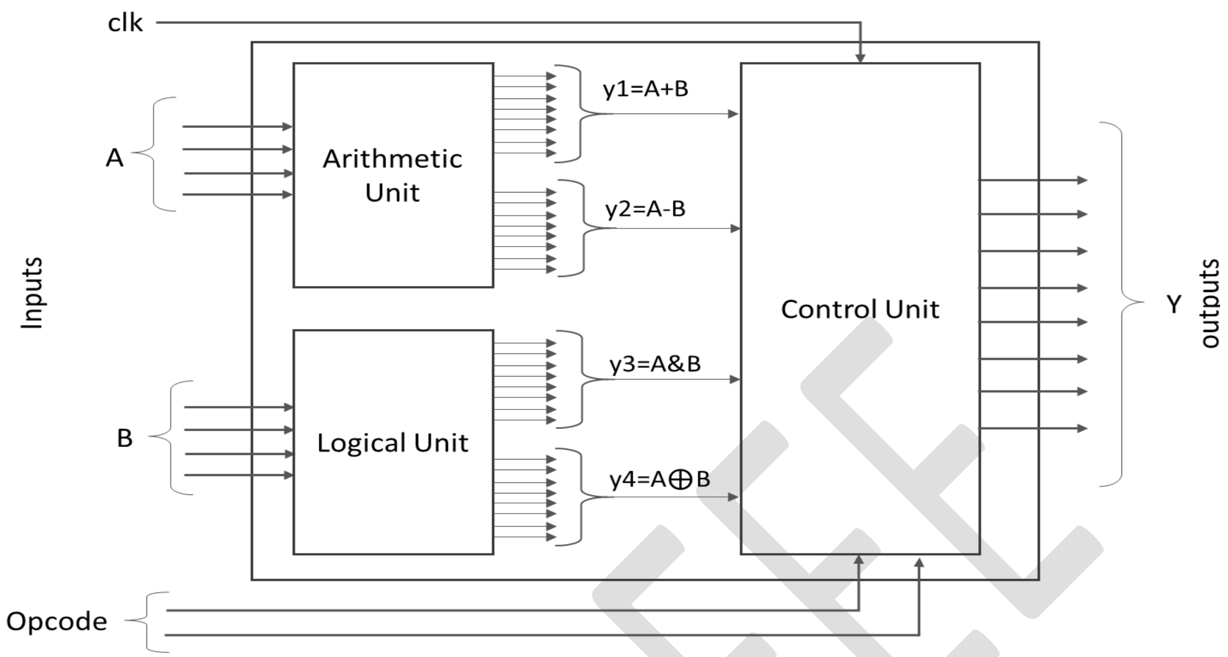
Testbench Module of Example 04

The following Verilog HDL code demonstrates the **Testbench Module** of the accumulator of Example 04.

```
`timescale 1ns/1ps
module accu_TB;
reg [7:0] in;
reg clk,reset;
wire [7:0] out;
accu dut(in, out, clk, reset);
initial
	clk = 1'b0;
always
	#5 clk = ~clk;
initial
begin
	#0 reset<=1; in<=1;
	#5 reset<=0;
	#50 $finish;
end
endmodule
```

Example 05

In this example, a 4-bit Arithmetic Logic Unit (ALU) shown in the following figure will be designed using Verilog HDL. The top module of the ALU is ***alu_4bit*** and it is designed using three sub-modules: ***logical_unit, arithmetic_unit,*** and ***control_unit***. In the design, the two 4-bit inputs **A** and **B** are fed to the inputs of ***arithmetic_unit*** and ***logical_unit*** modules to perform two different arithmetic operations and two different logical operations according to the function table given below. Thus the **arithmetic_unit** and **logical_unit generates** four outputs y1,y2,y3, and y4 which are fed to the inputs of ***control_unit*** module which generates the 8-bit output **Y** from the y1,y2,y3 and y4 depending on its 2-bit **Opcode** input. The output **Y** is also sensitive to the positive edge of the **clk** input.



The function table of the ALU is given below.

Function Table

Opcode	Output (Y)	Description of function
00	A+B	Add A to B
01	A-B	Subtract B from A
10	A&B	Bitwise AND
11	A⊕B	Bitwise XOR

The following Verilog HDL code demonstrates the ALU mentioned in Example 05.

```
module alu_4bit(A,B,Y,clk,Opcode);
input [3:0]A,B;
input [1:0]Opcode;
input clk;
output [7:0]Y;
wire [7:0]y1,y2,y3,y4;
arithmetic_unit sm1(A,B,y1,y2);
logical_unit sm2(A,B,y3,y4);
control_unit sm3(y1,y2,y3,y4,clk,Opcode,Y);
endmodule

module arithmetic_unit(x,y,y1,y2);
input [3:0]x,y;
output reg[7:0]y1,y2;
```

```
always@(x,y)
begin
        y1<=x+y;
        y2<=x-y;
end
endmodule

module logical_unit(x,y,y3,y4);
input [3:0]x,y;
output [7:0]y3,y4;
assign y3=x&y;
assign y4=x^y;
endmodule

module control_unit(y1,y2,y3,y4,clk,Opcode,Y);
input [7:0]y1,y2,y3,y4;
input [1:0]Opcode;
input clk;
output reg[7:0]Y;
always@(posedge clk)
begin
        if(Opcode ==2'b00)
                Y<=y1;
        else if(Opcode ==2'b01)
                Y<=y2;
        else if(Opcode ==2'b10)
                Y<=y3;
        else if(Opcode ==2'b11)
                Y<=y4;
        else
                Y<=0;
end
endmodule
```

Testbench Module of Example 05

The following Verilog HDL code demonstrates the **Testbench Module** of the 4-bit ALU of Example 05.

```
`timescale 1ns/1ps
module alu_4bit_TB;
reg [3:0]A,B;
reg [1:0]Opcode;
reg clk;
wire [7:0]Y;
alu_4bit dut(A,B,Y,clk,Opcode);
initial
```

```
begin
		clk = 1'b0; Opcode=2'b00; A=4'b0100; B=4'b1100;
end
always
		#2.5 clk = ~clk;
initial
begin
		#5 Opcode<=2'b01; A=4'b1000; B=4'b0111;
		#5 Opcode<=2'b10; A=4'b1111; B=4'b1011;
		#5 Opcode<=2'b11; A=4'b1001; B=4'b1010;
		#5 $finish;
end
endmodule
```

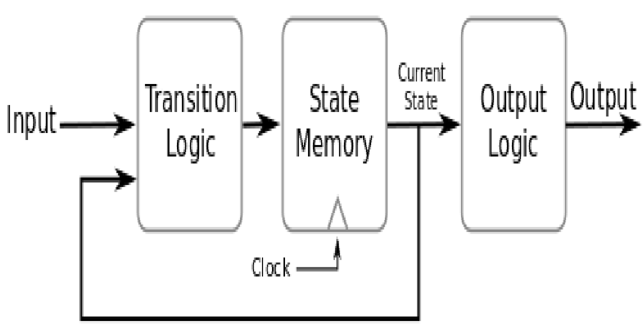
Finite State Machine Design

In a sequential circuit, outputs depend not only on the applied input values but also on the internal state. The internal state also changes with time. As the number of states in a sequential circuit is finite it is also referred to as a Finite State Machine (FSM). FSMs need memory to hold the current state and logic devices to determine the next state. Elevators(lift), vending machines, traffic signal systems, password generators etc. are examples of FSM.

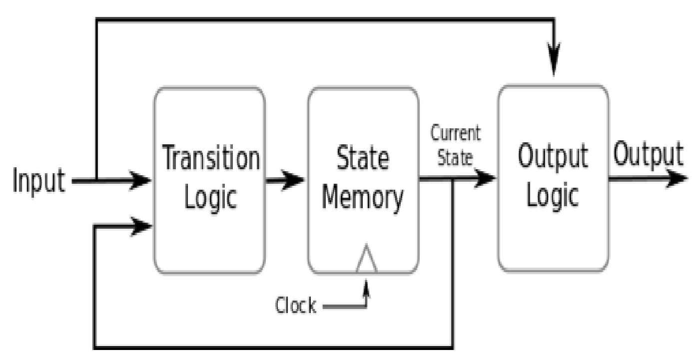
There are two types of finite state machines called the Mealy machine and the Moore Machine.

In Mealy machines, the output is a function of the current state and inputs. In Moore machines, the output is a function of only the current state. To design FSMs, we need to find the state transition diagram or the state table.

FSMs are modeled in Verilog with an always block defining the state registers and combinational logic defining the next state and output logic.



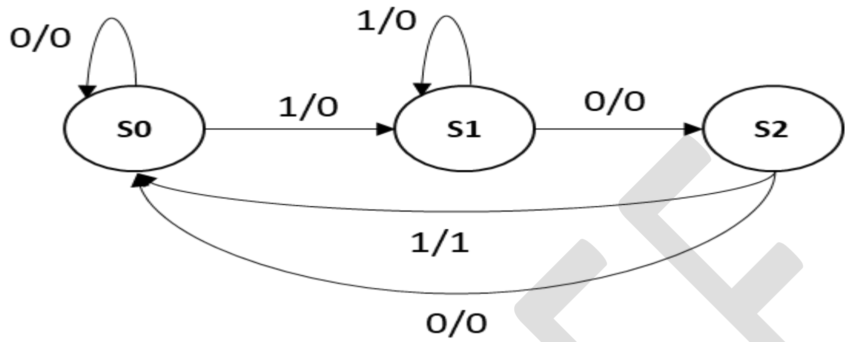
Moore Machine



Mealy Machine

Example 06

In this example, the Verilog HDL code of a Mealy machine is demonstrated that generates output ‘1’ when sequence 101 is detected in a bitstream.



State Transition Diagram

The following Verilog HDL code demonstrates the sequence detector mentioned in Example 06.

```
module seq_101(i,clk,out);
input i,clk;
output reg out;
localparam S0=2'b00, S1=2'b01, S2=2'b10;
reg [1:0]state;
always@ (posedge clk)
begin
case (state)
S0: begin
          out<=i?0:0;
        state<=i?S1:S0;
    end
S1: begin
          out<=i?0:0;
        state<=i?S1:S2;
    end
S2: begin
          out<=i?1:0;
       state<=i?S0:S0;
      end
default:
       begin
            out<=0;
          state<=S0;
        end
endcase
end
endmodule
```

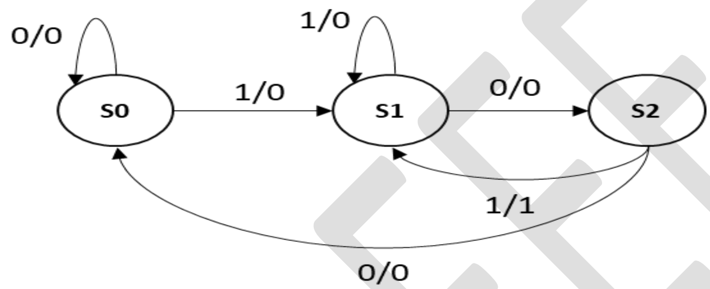
Testbench Module of Example 06

The following Verilog HDL code demonstrates the **Testbench Module** of the sequence detector of Example 06 where 0101001010-bit stream is generated.

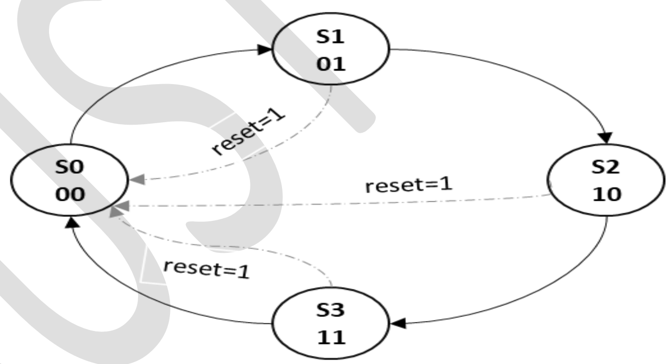
```
`timescale 1ns/1ps
module seq_TB;
reg i,clk;
wire out;
seq dut(i,out,clk);
initial
   clk=0;
always
   #2 clk=~clk;
initial
begin
   #0 i=0;
   #5 i=1; #4 i=0; #4 i=1; #4 i=0;
   #4 i=0; #4 i=1; #4 i=0; #4 i=1;
   #4 i=0;
   #4 $finish;
end
endmodule
```

Post Lab Tasks

1. Design a negative edge triggered D flip flop with reset and verify its functionality using testbench.
2. In Example 5 how many 1s will be generated at the output if the input bitstream is 01010100? Verify your answer using testbench.
3. Write a Verilog program to implement the digital system represented by the following state transition diagram of a Mealy machine. Assume that system has input and output variables **in** and **Y.** The system functions when the positive edge of the clock is detected.



4. Design a Mealy machine to detect the 010 sequences hence verifying its functionality using testbench.
5. The state transition diagram of a two-bit counter is shown below. Assuming that each state changes when a positive edge clock is detected. Design and verify the system using Verilog HDL.



Lab-4: Introduction to Unix Shell

Objective

The main objectives of this lab are:

- Logging into the Cadence software installed Linux server.
- To get started with the Linux environment.
- To comprehend the file and directory management using shell command.
- To get familiar with Vim text editor.

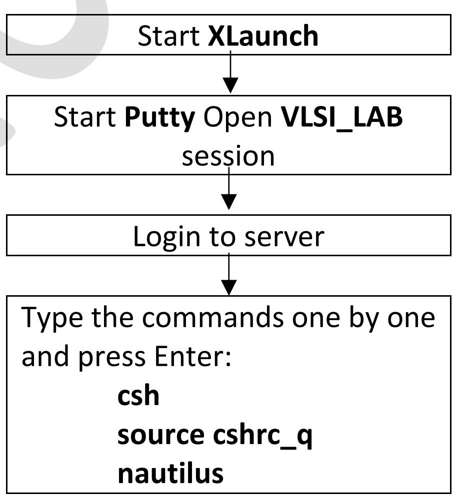
Introduction

Electronic Device Automation (EDA) tools are required to run for a long time which consumes a huge amount of memory (RAM), runs in multiple threads/processes and are multiuser programs. For that, Unix or Linux is the ideal choice to run (EDA) tools.

In the upcoming lab classes, we will use different cadence tools preinstalled on the Linux server. For that, we have to login into your student account from the Windows operating system based computer allocated for the student use.

Steps to Login into Linux Server

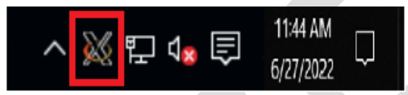
The following flowchart summarizes the steps to login into the Linux server.



The detailed instructions are given below

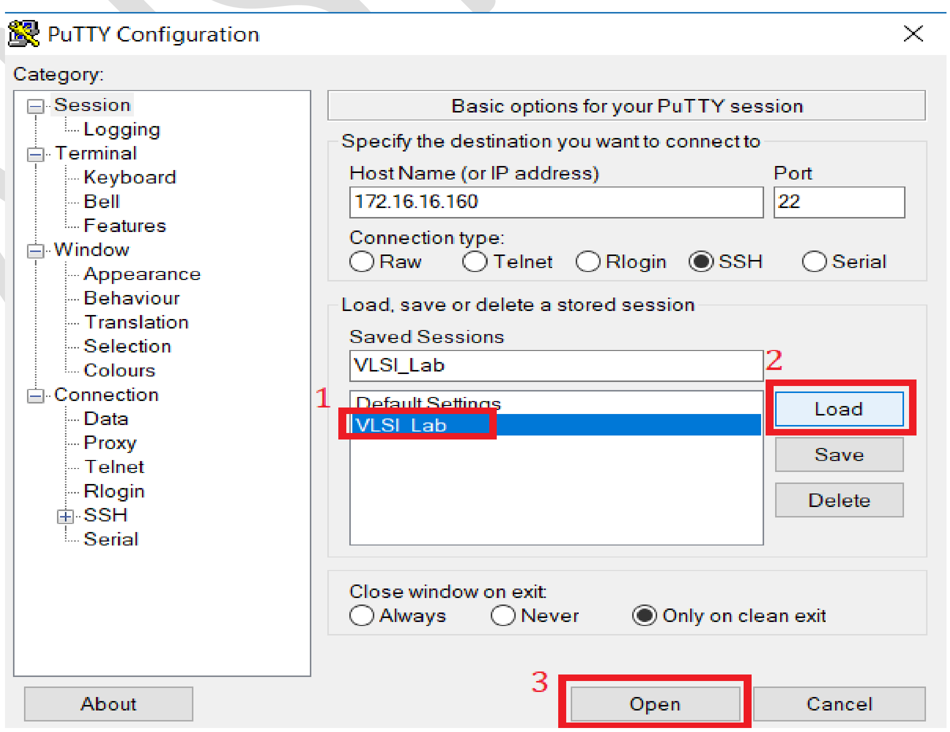
1. Find the Desktop shortcut icon for **XLaunch**. Double-click on it. Click **Next, Next, Next**, **Finish** (in that order) in the windows that pop up one after another.

After it starts, you will see the Xming icon at the bottom right corner of your Desktop screen.



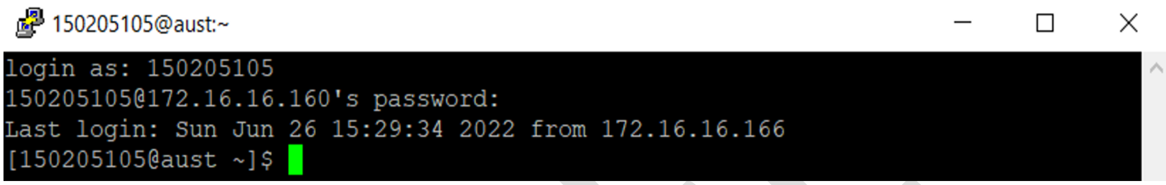
2. Find the icon for Putty. Double click on it to open it. 'Putty Configuration' window will pop-up

3. Select **VLSI_LAB** under the **'Saved Sessions'** category. Click **Load** and then click **Open.**

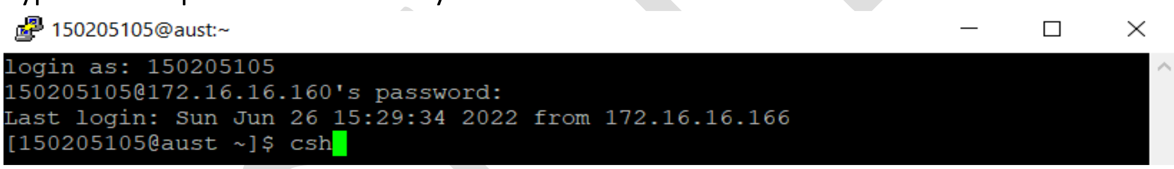


4. Now you will see a Terminal window which prompts you for login.

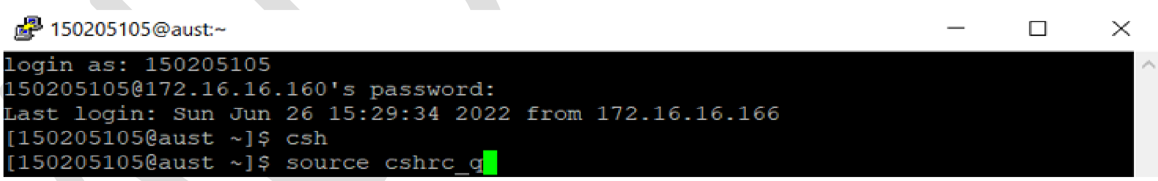
5. Log in to your workstation using user ID and password. Your user name and your password will be your student ID. When you are typing your password, the command window will not display the characters you type in, so make sure you are typing the right password. After logging in to your account, Terminal window should look like the following:



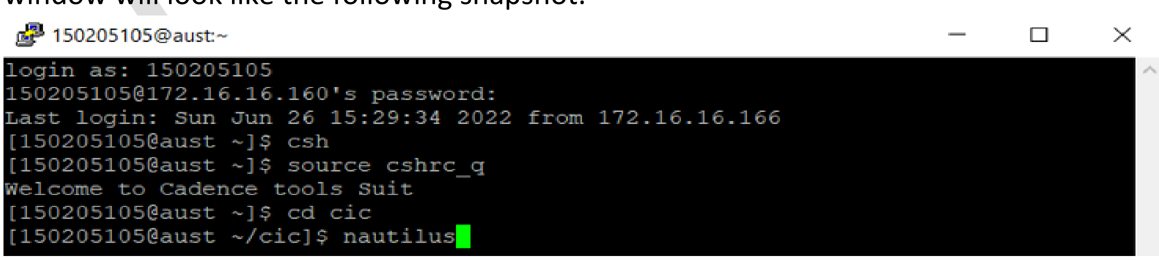
6. Type **csh** and press the '**Enter'** key.



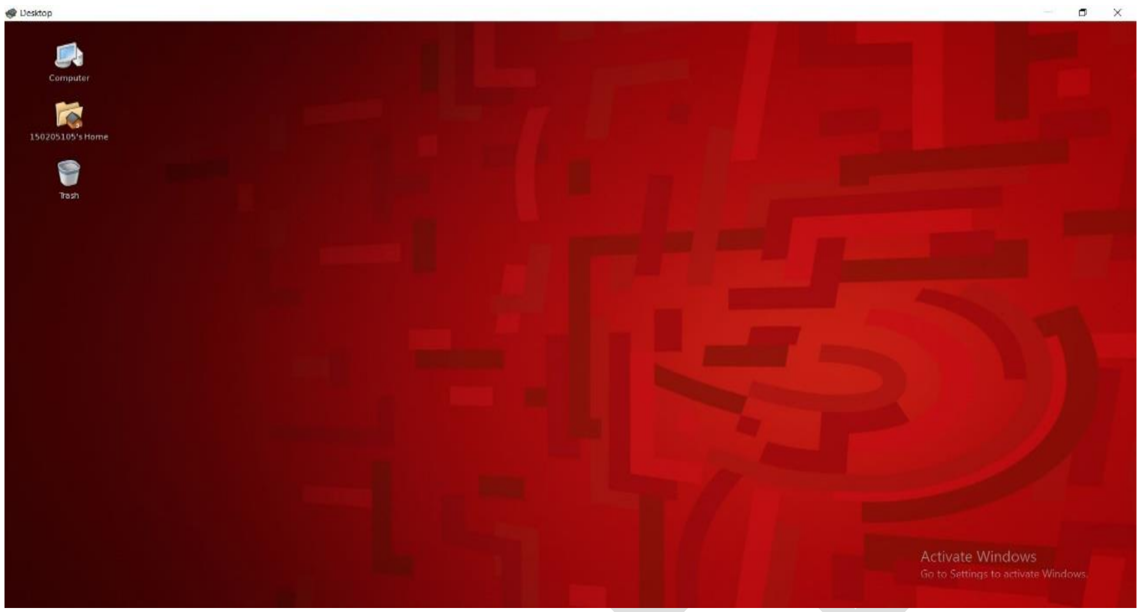
7. Then type **source cshrc_q** and press the '**Enter**' key. The following message will be displayed in the Terminal window: Welcome to Cadence tools Suite That means you can use Cadence tools now.



8. Finally Type **nautilus** and press **the 'Ente**r' key to enter the GUI of your account. The GUI window will look like the following snapshot.

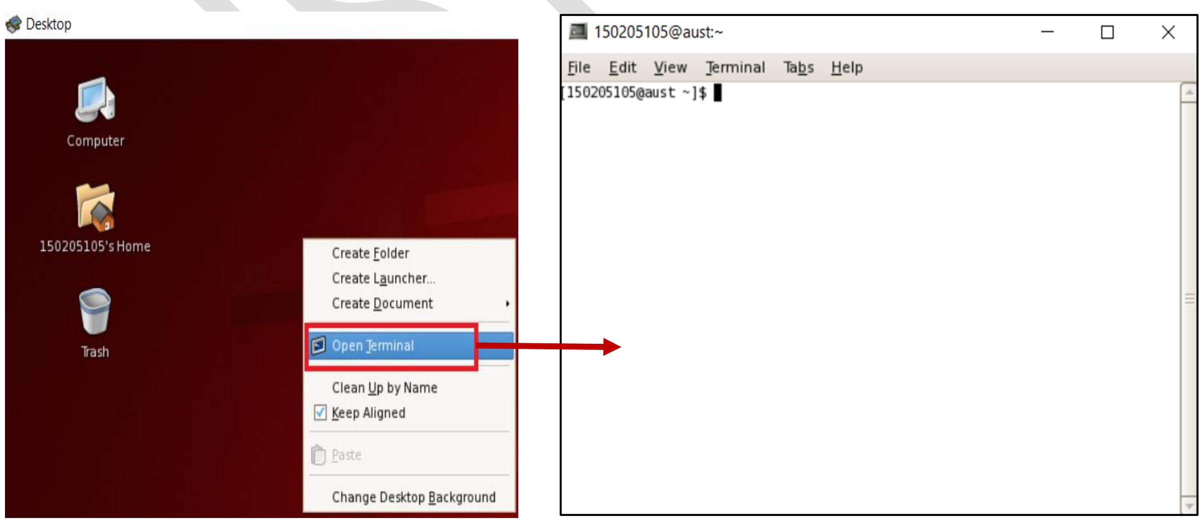


9. The GUI of your account will look like the following window



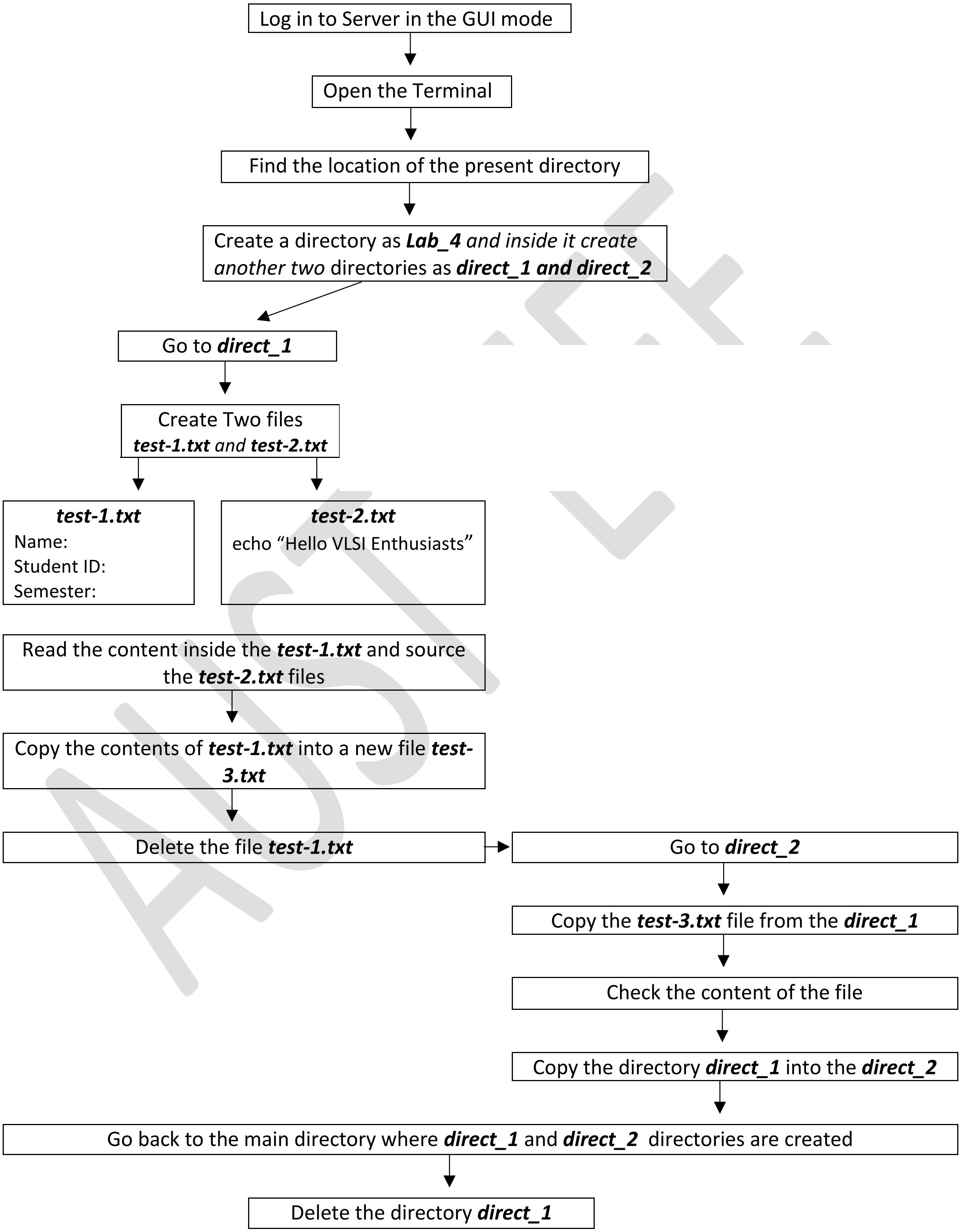
Terminal in Unix

1. Right-click on the blank space of your Linux desktop a window will pop up and then select **Open_Terminal** .



Lab Task

The following flow chart summarizes the tasks to be performed in Lab-04.



Directory Management in Unix

Command	Description	Syntax
pwd	print name of current/working directory.	*pwd*
ls	lists directory contents.	*ls*
ls -ltr	lists directory contents by arranging them according to time by using the **-ltr** switch.	*ls -ltr*
tree	Show the file hierarchy inside a directory	*tree*
mkdir	make directories.	*mkdir <directory_name>*
cd	Change directory.	*cd <directory_path>*
cd ~/ cd ~	Goes to the home directory	cd ~/ cd ~
cd .. cd ../	Goes to the previous directory.	cd .. cd ../
cd ../../	Goes two directories back.	cd ../../

Vim Editor in Unix

Command	Description	Syntax
touch	Creates a file.(Extension can be .txt, .v, .tcl, etc)	*touch test.txt*
press insert/ins	Enables the INSERT mode	The commands of the vim editor can be executed after pressing the **Esc** key.
:w	Writes/saves the text file.	
:q	Quits from vim editor.	
:wq	Writes the text and then quits the vim editor.	
:wq!	Forcefully writes and quits the vim editor through bang(!)	
:set nu	Shows the line numbers.	
:<line no>	The cursor moves to the specified line no.	
:set nu!	Removes the line numbers.	
:/*xyz*	Used to search all the "*xyz*" from the beginning (Use n to move from one to another)	
:?xyz	Used to search all the "*xyz*" from the bottom	
:%s=*x*=*y*=g	Replaces all x with y	
u	Undo	
Press Ctrl+R	Redo	

Reading and sourcing a file

Command	Description	Syntax
cat	Checks the content inside a file.	*cat <file_name>*
source ./	Reads and executes commands from the file.	*source <file_name>* *./<file_name>*

Files and directory manipulation in Unix

Command	Description	Syntax
cp	Copies files and directories.	*cp <source_file> <destination_file>*
rm	Remove files.	*rm <directory_name>*
rmdir	Removes empty directories.	*rmdir <directory_name>*
rm -rf	Removes directories containing files by force recursive using force recursive switch.	*rm -rf <directory_name>*
mv	Moves one or more files and directories to a given location (if the location is not defined. it renames files on the current location).	*mv<source_file> <destination_dir>*

Other Useful Commands

Command/Key	Description
history	Prints the previous commands executed in the bash terminal. (Syntax: history)
man	Shows the documentation of any command (Syntax: man pwd)

Shortcut Keys

Command/Key	Description
Up/Down Arrow keys	Scrolls through command history.
Tab key	Used to complete the command you are typing.
Ctrl + Shift + C	Copies the highlighted command to the clipboard.
Shift + Insert	Pastes the contents of the clipboard.
Ctrl + L	Clears the terminal

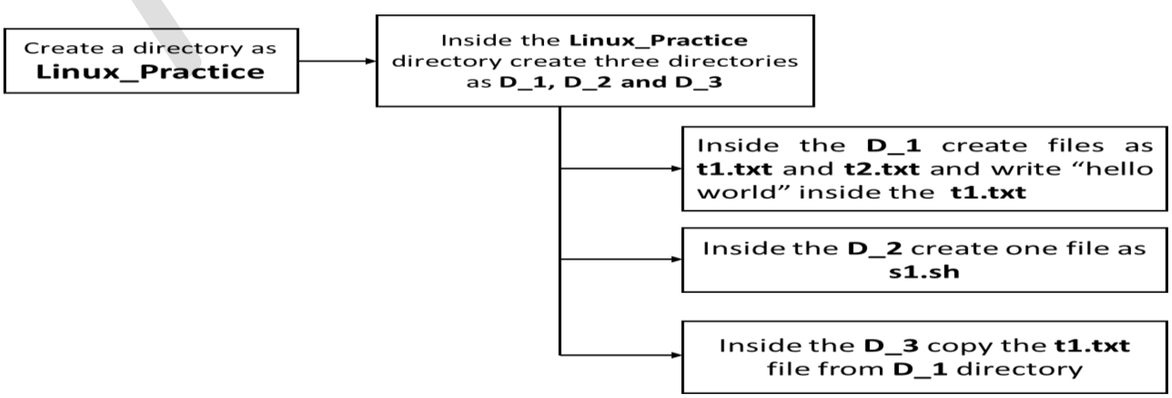
Bash Script

Bash scripts are typically used for handling directories and files, not for coding. But it can be useful for scripting with various arithmetic use cases and scenarios. Bash only supports integer arithmetic, so if we need to perform calculations with floating-point numbers, have to use separate utility in bash. There are several ways and syntax of performing arithmetic operations, using conditions and loops in bash. The below code is just a simple demonstration of arithmetic operations, *if..else..* statement, *for* loop and array declaration in bash. A bash script can be written using the vim editor and it should be saved with the extension ***.sh*** . The commands inside the script can be executed by sourcing the script.

```
a=10
read -p "enter b:" b                #Stores user's input in b variable by prompting in display using -p
sum=$(($a+$b))
sub=$(($a-$b))
mult=$(($a*$b))
div=$(($a/$b))
echo "sum=$sum
$a-$b=$sub
$a*$b=$mult
$a/$b= $div"
                                    #Using if..else.. statement to find whether a is even or odd
if [ $(($a%2)) == "0" ]
then
echo "$a is even"
else
echo "$a is odd"
fi

c=($sum $sub $mult $div)            #storing different variables in array c
elements=${#c[@]}                   #Counts the no of elements present in the array c

for((i=0;i < $elements;i++))        #Using for loop to display all the elements present in array c
do
echo "${c[i]}"
done
```

Post Lab Tasks

1. How fractional values can be handled in bash?
2. Write a bash script to perform the following arithmetic operation.
$$y = \sin(5) + e^3 + \sqrt{3} + 2^3$$
3. Write a bash script that will show your root and home location whenever it is sourced.
4. Write a bash script that will create the following hierarchy in your home.



Lab-5: Synthesis using Genus Synthesis Solution

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 technology-dependent or independent gate-level netlist including nets, sequential and combinational cells, and their connectivity. The main goal of synthesis are 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.

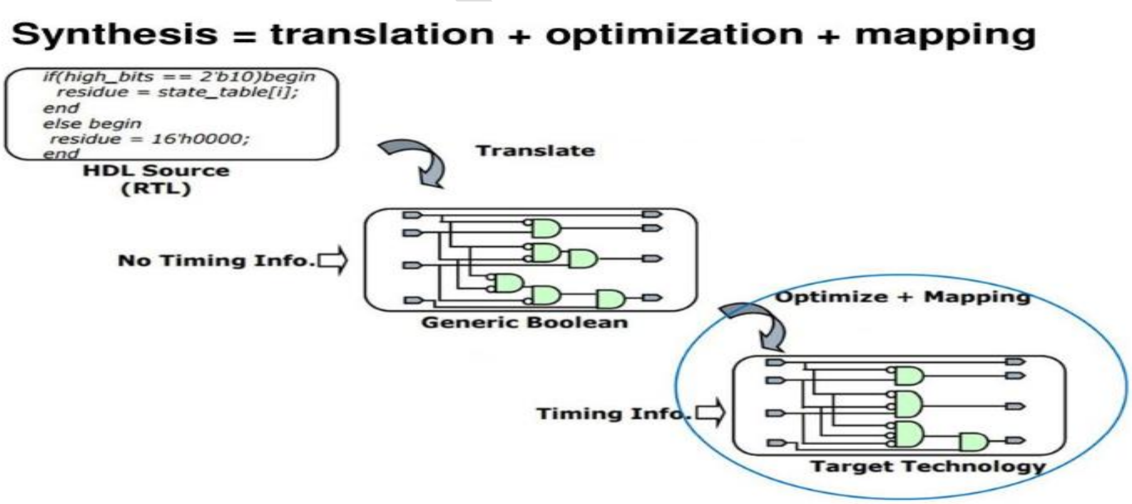


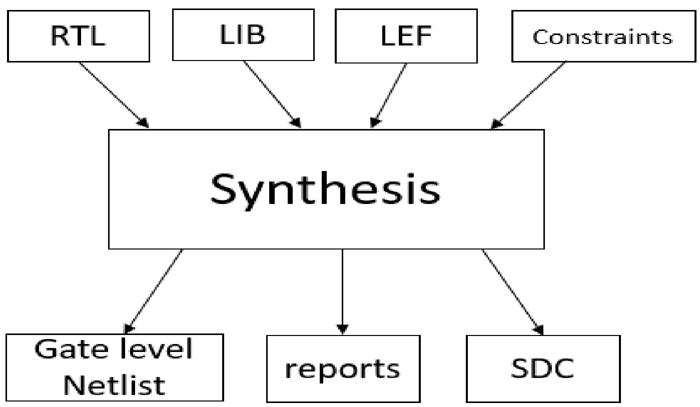
Fig: Steps of Synthesis

Synthesis tools perform the following three steps to meet all the goals.

- **Translation:** Converts RTL into basic Boolean equation form which is technology-independent representation.
- **Optimization:** Performs two types of optimizations.
    - **Logic optimization**
        - Detecting identical cell

        - Optimize multiplexer
        - Remove unused cell and net
        - Reduced word size of the cell
    - **Design optimization**
        - Reduced WNS (Worst Negative Slack) and TNS (Total Negative Slack)
        - Power and area optimization
        - Attempting to meet DRV (Design Rule Violation: Max Fanout, Max Transition, Max Capacitance)
- **Mapping:** Technology-independent Boolean logic equations are mapped to technology-dependent library logic gates based on design constraints, and available gates in the technology library.

Input and Output files of Physical Design



- **Input Files**

Technology-Related Files

I. Technology file containing names, physical and electrical characteristics of metal layers, and design rules (.lef)
II. Timing and functionality information of the standard cell (.lib)

Design Related Files

I. Post Synthesized or Gate Level Netlist (.v)
II. Standard Design Constraints containing all timing and design limitations (.sdc)

- **Output Files**

I. Post-synthesized and optimized gate-level netlist (.v)
II. Standard Design Constraints (.sdc)

Lab Task

In this lab, we will perform synthesis on the RTL of a 4-bit ALU designed and verified in Lab-3 (Example 5).

1. Log in to the server in the GUI mode and source the Cadence license file.
   [**Xlaunch (enable SSH)→putty (load server IP) → login → csh→ source ~/cshrc_q→ nautilus]**

2. In the GUI mode of your account open a terminal by executing **right click on mouse** → **open terminal.**

3. Create a directory at your home ***lab_5***
   First check you are at the home using the command ***pwd***

[150205105@aust ~]$ pwd

   Then create the directory using the ***mkdir*** command.

[150205105@aust ~]$ *mkdir lab_5/*

4. Check whether the directory is created or not using the following command

[150205105@aust ~]$ *ls -ltr*

5. Got the directory ***lab_5*** executing the command **cd lab_5/**

[150205105@aust ~]$ *cd lab_5/*

6. Copy the necessary files from the root into the ***lab_5*** directory by executing the following command.

[150205105@aust lab_5 *lab_5***]$** *source /physicalDesignLab.sh*

7. Go to the copied directory ***synthesis_lab.***

[150205105@aust lab_5]$ *cd synthesis_lab*

8. Make sure the following directories and the files are present in the **synthesis_lab** directory using the command ***tree.***

[150205105@aust synthesis_lab]$ *tree*

```
[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
```

9. Open the **synthesis_cmd.tcl** file using the **Vim** editor.

[150205105@aust synthesis_lab]$ *vi synthesis_cmd.tcl*

10. Make sure the following commands are present inside the **synthesis_cmd.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

11. After checking the **synthesis_cmd.tcl** close the **Vim** editor by executing **Esc → :q**

12. Now make sure you are in the **synthesis_lab** directory. And launch the Genus tool using the command ***genus***.

[150205105@aust synthesis_lab]$ *genus*

13. If the Genus tool is successfully launched, the following text will be shown in the terminal.

```
Cadence Genus Synthesis Solution, Version 15.10-s019_1, built Nov  4 2015

Copyright 2015 Cadence Design Systems, Inc. All rights reserved worldwide.
Cadence and the Cadence logo are registered trademarks and Genus is a trademark
of Cadence Design Systems, Inc. in the United States and other countries.

Options:

Checking out license: Genus_Synthesis
Sourcing GUI preferences file /home/Fall18/150205105/.cadence/genus/gui.tcl ...
WARNING: This version of the tool is 2426 days old.
genus@root:> source synthesis_cmd.tcl
```

14. Now source the **synthesis_cmd.tcl** file to perform the synthesis of the RTL present in the **input_files** directory.

genus@root> *source* synthesis_cmd.tcl

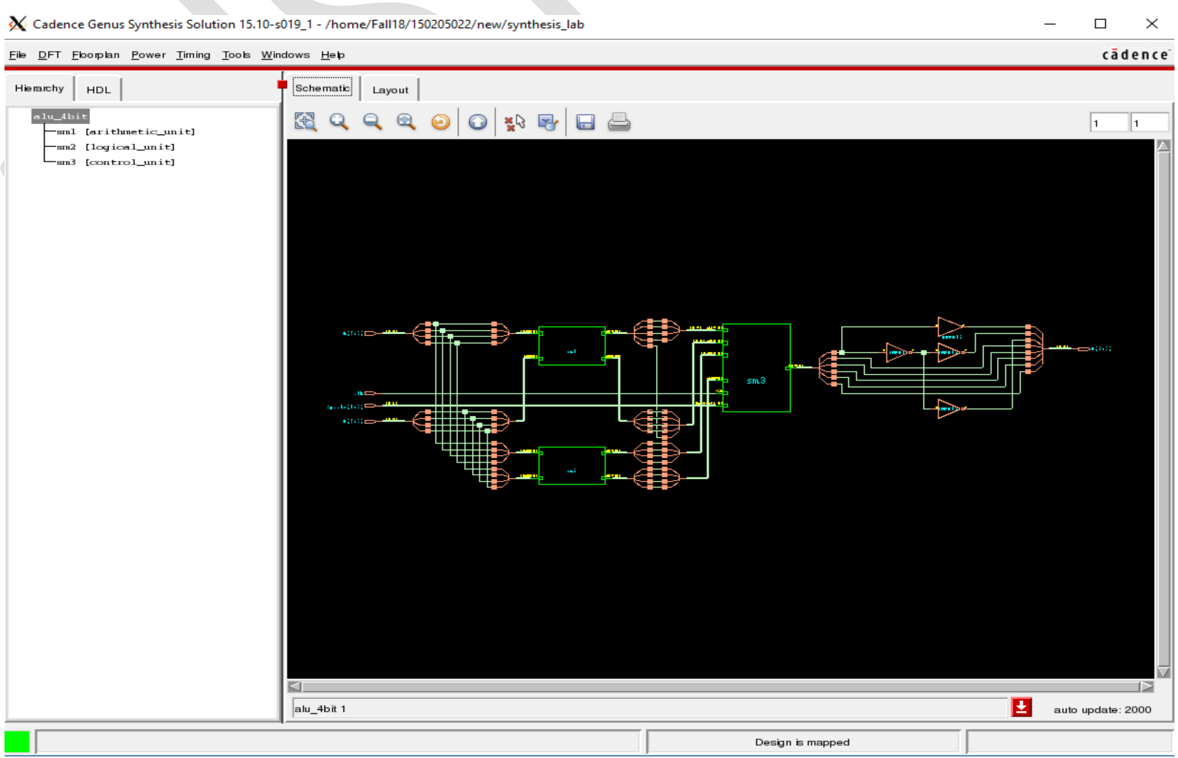
15. After successfully execution of **synthesis_cmd.tcl** file, the Genus tool will show that the SDC file export is finished.

```
Info    : Done incrementally optimizing. [SYNTH-8]
        : Done incrementally optimizing 'alu_4bit'.
     flow.cputime  flow.realtime  timing.setup.tns  timing.setup.wns  snapshot
UM:             0              1                0 ps       infinity ps  synthesize
Finished SDC export (command execution time mm:ss (real) = 00:00).
genus@design:alu_4bit>
```

16. Now to show the synthesized output execute the command **gui_show**.

genus@design:alu_4bit> *gui_show*

17. 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.



18. Close the GUI window and exit the Genus tool using the exit command

.

```
genus@design:alu_4bit> exit
```

19. Now you will be in the **synthesis_lab** directory**.** Check the files inside the directory using the ***ls -ltr*** command and make sure ***alu_4bit.sdc*** and ***alu_4bit _mapped.v*** files are present in the directory which will be used for place and route in the upcoming labs.

```
[150205105@aust synthesis_lab]$ ls -ltr
total 84
-rwxr-xr-x 1 150205105 150205105  1357 Nov  6 12:38 synthesis_cmd.tcl
drwxr-xr-x 2 150205105 150205105  4096 Nov  6 12:38 input_files
drwxr-xr-x 5 150205105 150205105  4096 Nov  6 12:38 EDI_files
drwxr-xr-x 3 150205105 150205105  4096 Nov  6 12:40 fv
-rw-rw-r-- 1 150205105 150205105  7804 Nov  6 12:40 alu_4bit_elaborated.v
-rw-rw-r-- 1 150205105 150205105  8033 Nov  6 12:40 alu_4bit_generic.v
-rw-rw-r-- 1 150205105 150205105  2895 Nov  6 12:40 alu_4bit.sdc
-rw-rw-r-- 1 150205105 150205105  5434 Nov  6 12:40 alu_4bit_post_synthesis.v
-rw-rw-r-- 1 150205105 150205105  5434 Nov  6 12:40 alu_4bit_mapped.v
-rw-rw-r-- 1 150205105 150205105    36 Nov  6 12:41 genus.cmd
-rw-rw-r-- 1 150205105 150205105 26573 Nov  6 12:42 genus.log
[150205105@aust synthesis_lab]$
```

Post Lab Task

1. Is the testbench module synthesizable?
2. Why the operating condition of synthesis is slow?
3. What is Standard Design Constraints (SDC)?
4. What do **LEF** and **LIB** files contain?
5. List the functions of buffer cells in synthesis.
6. Check the function of commands ***report_power***, ***report_gates***, ***report_timing*** in Genus.

Lab-6: Physical Design Using Encounter Digital Implementation System (Part 1)

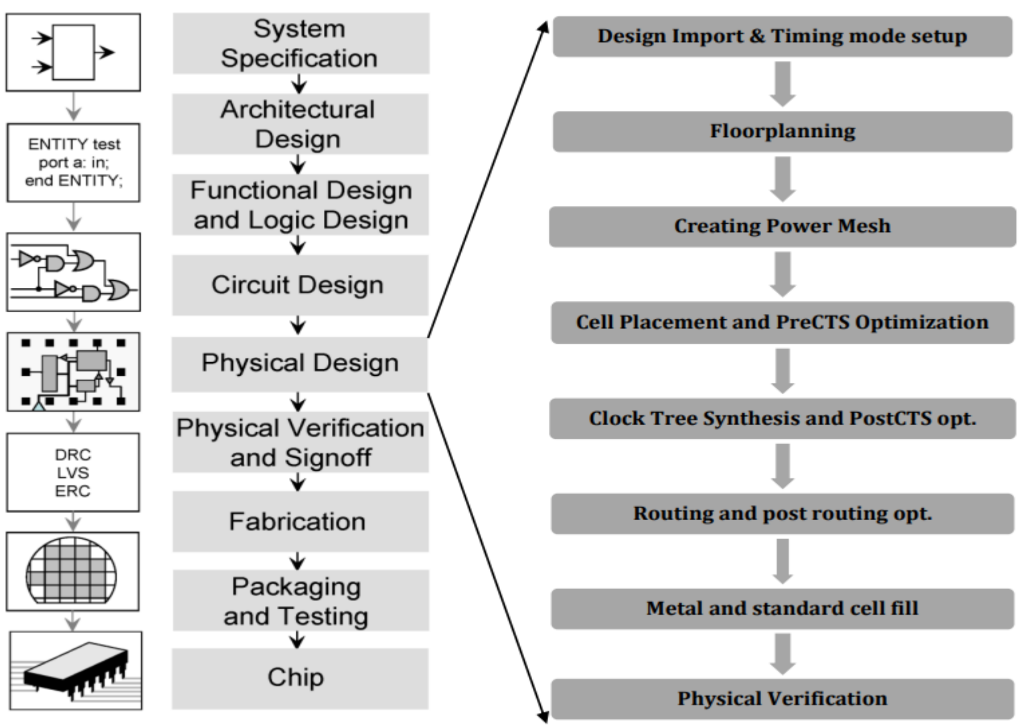
Objective

The main objectives of this lab are:

- Familiarization with Physical Design flow.
- Familiarization with MMMC(Multi-Mode Multi-Corner).
- Familiarization with chip Floorplan.

Introduction

Back-end Design or Physical Design involves the placement of standard cell, macro, and making physical connections between pins using metal layers(routing) to meet the design power, performance, and area (PPA) goals. Physical Design flow uses the technical libraries that are provided by the fabrication houses. These technology files provide information regarding the type of Silicon wafer used, the standard cells used, and the layout rules. Physical design is followed by verification after all verifications post-processing is applied where the data is translated into an industry-standard format called **GDSII**.



ASIC design flow showing the physical design tasks

Physical Design is the process of transforming a circuit description into a physical layout that describes the position of cells and routes for the interconnections between them. In this stage, standard cells are placed on a defined floorplan, and route the wire to connect the standard cells. That is why we call this automatic Place and Route (PnR). Goals for each stage of PnR are given below.

- **Floorplan**
  I. Define the width and height of the core and die. (core defines the area where core Logic cells are placed).
  II. Define locations of preplaced cells (blocks or macros, placed based on connectivity)
  III. Surround pre-placed cells with Decoupling capacitors.

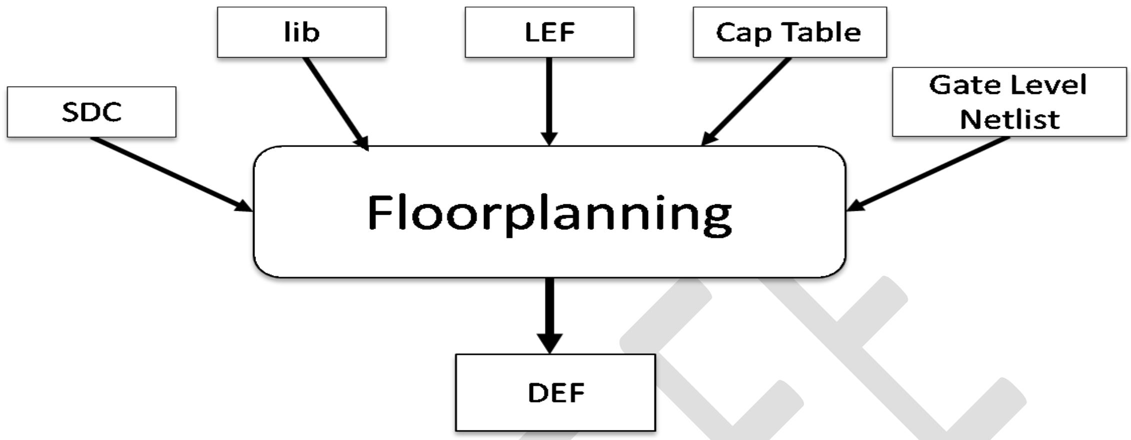
- **Power Plan**
  I. Power grid network is created to distribute power to each part of the design equally.
  II. To connect the power network to every instance by considering IR drops and EM (Electromigration)
  III. Reduce dynamic and static power dissipation.

- **Placement**
  I. Minimizes congestion and makes the design routable
  II. Timing, power, and area optimization
  III. Reduces cell density, pin density, and congestion hot-spots
  IV. Minimal DRV violations

- **Clock Tree Synthesis (CTS)**
  I. Meeting the constraints written in the SDC file
  II. Meeting clock tree targets (Min skew and insertion delay (latency))
  III. Controls buffer/inverter level used in the clock network

- **Routing**
  I. Minimizes total interconnect/wire length
  II. Minimizes critical path delay
  III. Completes the connection without increasing total area and minimizes the number of layer changes
  IV. Reduces cross-talk noise
  V. Meeting Setup and hold timing margin

Input and Output files of Physical Design



- **Input Files**

  Technology-Related Files

  i) **Library Exchange Format file (.lef):** Contains technology information and an abstract view of standard cells

  ii) **Liberty Timing file (.lib)**: ASCII representation of timing, power parameter, and functionality information associated with cells of a particular technology node

  Design Related Files

  i) Post Synthesized or Gate Level Netlist (.v)

  ii) Standard Design Constraints containing all timing and design limitations (.sdc)

- **Output Files**

  i) Post APR Netlist (APR refers to Automatic Place and Route)

  ii) DEF (Design Exchange Format)

In this lab, our main task is to understand and initialize the MMMC (multi-mode multi-corner) and design an efficient floorplan for our synthesized RTL of lab-6. We will perform the rest of the steps and physical verification in the next lab.

Lab Task

Launching Encounter Tool

1. Log in to the server in the GUI mode and source the Cadence license file.

   [**Xlaunch (enable SSH)→putty (load server IP) → login → csh→ source ~/cshrc_q→ nautilus]**

2. In the GUI mode of your account open a terminal by executing **right-click on mouse → open terminal.**

3. First check you are at the home using the command ***pwd***

[150205105@aust ~]$ pwd

   Then create the directory using the ***mkdir*** command.

[150205105@aust ~]$ *mkdir lab_6/*

4. Check whether the directory is created or not using the following command

[150205105@aust ~]$ *ls -ltr*

5. Go to the directory ***lab_6*** by executing the command ***cd lab_6/***

[150205105@aust ~]$ *cd lab_6/*

6. Copy the directory ***pnr_lab*** from the root into the ***lab_6*** directory

[150205105@aust lab_6]$ *cp /pnr_lab . -rf*

7. Go to the copied directory ***pnr_lab.***

[150205105@aust lab_6]$ *cd pnr_lab*

8. Copy the synthesized netlist ***alu_4bit_mapped.v*** and post-synthesis sdc file ***alu_4bit.sdc*** that you created in ***lab 5*** using the following commands.

[150205105@aust pnr_lab]$ *cp ~/lab_5/synthesis_lab/alu_4bit_mapped.v input_files /*
[150205105@aust pnr_lab]$ *cp ~/lab_5/synthesis_lab/ alu_4bit.sdc input_files /*

9. Make sure the following directories and the files are present in the ***pnr_lab*** directory using the command ***tree.***

[150205105@aust *pnr_lab*]$ *tree*

```
[150205105@aust pnr_lab]$ tree
.
|-- EDI_files
|   |-- lef
|   |   `-- gsclib045.lef
|   |-- libs
|   |   |-- fast.lib
|   |   |-- slow.lib
|   |   `-- typical.lib
|   `-- others
|       `-- capTable
`-- input_files
    |-- alu_4bit.sdc
    `-- alu_4bit_mapped.v

5 directories, 7 files
```

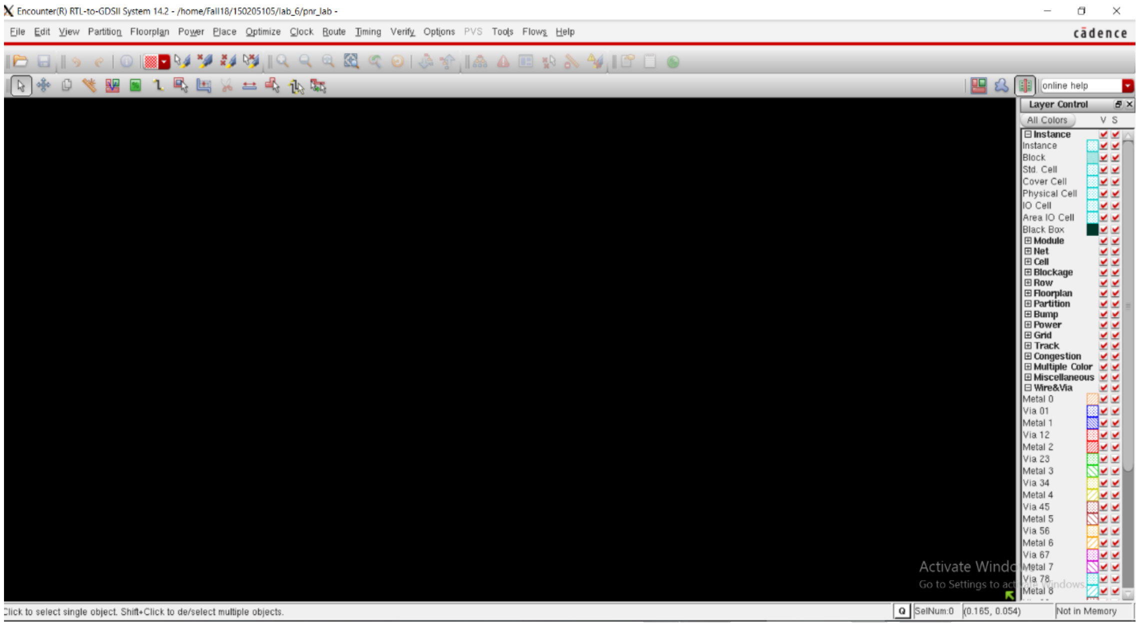
10. Now make sure you are in the ***pnr_lab*** directory. And launch the Encounter tool using the command ***encounter***.

[150205105@aust pnr_lab]$ *encounter*

11. If the **encounter** tool is successfully launched, the following text will be shown in the terminal and the black GUI window of the encounter will appear on your screen.

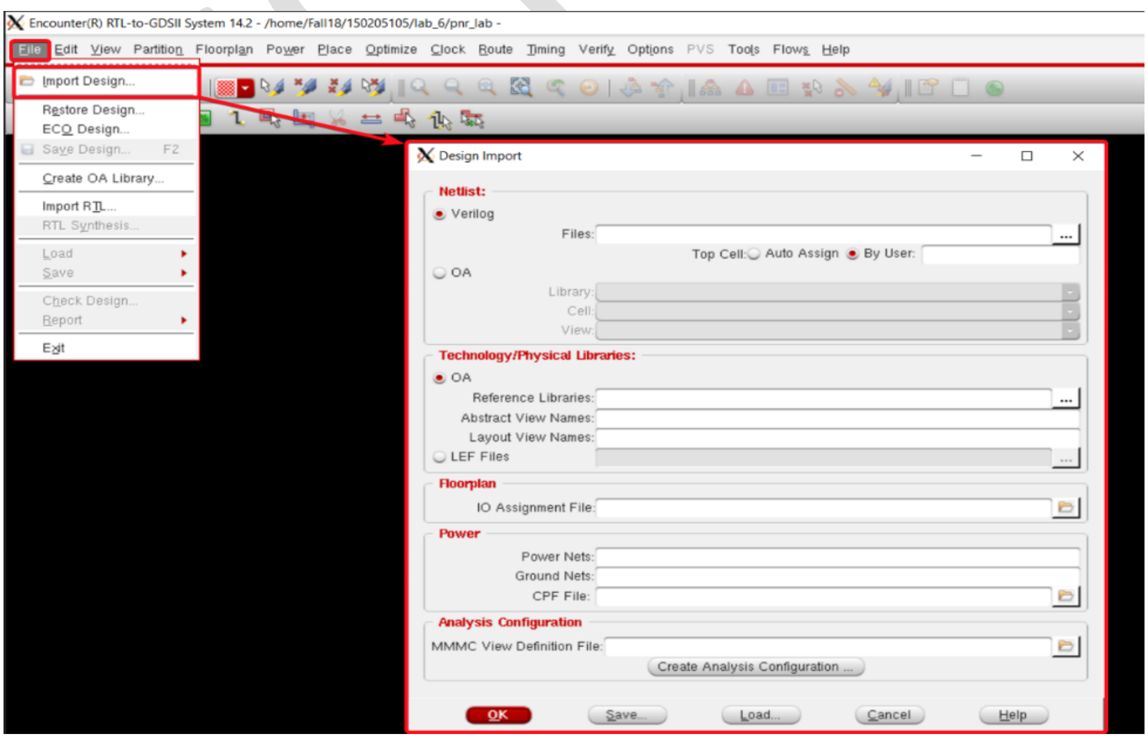
```
*                                                                   *
*                 Cadence Design Systems, Inc.                      *
*                    2655 Seely Avenue                              *
*                   San Jose, CA 95134,  USA                        *
*                                                                   *
*                                                                   *
*********************************************************************

@(#)CDS: Encounter v14.20-p004_1 (64bit) 11/05/2014 14:06 (Linux 2.6.18-194.el5)
@(#)CDS: NanoRoute v14.20-p014 NR141101-0648/14_20-UB (database version 2.30, 246.6.1) {superthreading v1.24}
@(#)CDS: CeltIC v14.20-p001_1 (64bit) 10/15/2014 03:59:12 (Linux 2.6.18-194.el5)
@(#)CDS: AAE 14.20-p007 (64bit) 11/05/2014 (Linux 2.6.18-194.el5)
@(#)CDS: CTE 14.20-p003_1 (64bit) Oct 27 2014 04:09:59 (Linux 2.6.18-194.el5)
@(#)CDS: CPE v14.20-p003
@(#)CDS: IQRC/TQRC 14.1.2-s148 (64bit) Mon Sep 29 16:54:36 PDT 2014 (Linux 2.6.18-194.el5)
@(#)CDS: OA 22.50-p007 Tue Sep 30 00:05:09 2014
@(#)CDS: SGN 10.10-p124 (19-Aug-2014) (64 bit executable)
@(#)CDS: RCDB 11.5
--- Starting "Encounter v14.20-p004_1" on Sun Jul 31 16:40:47 2022 (mem=94.8M) ---
--- Running on aust (x86_64 w/Linux 2.6.18-348.el5) ---
This version was compiled on Wed Nov 5 14:06:30 PST 2014.
Set DBUPerIGU to 1000.
Set net toggle Scale Factor to 1.00
Set Shrink Factor to 1.00000

**INFO:  MMMC transition support version v31-84

encounter 1>
```

The following Encounter GUI window will appear.

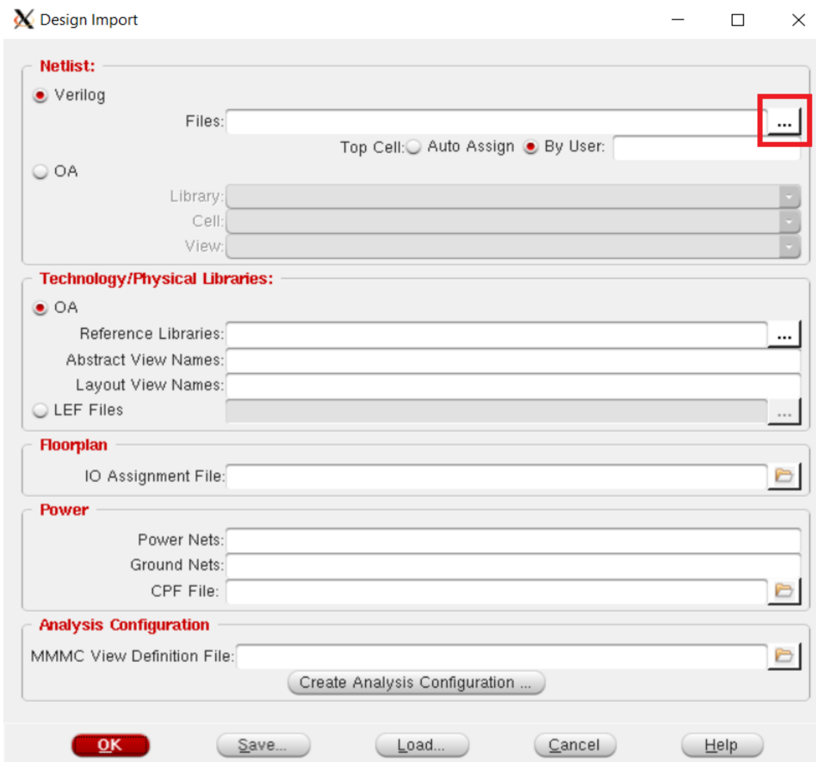


Design Import & Timing mode setup

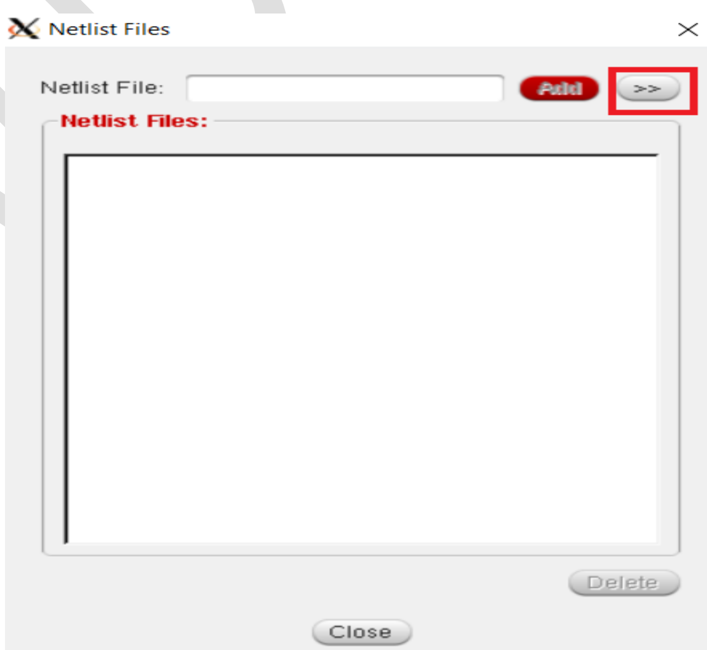
12. Now from the Encounter GUI window, launch the **Design Import** window by executing **File → Import Design**



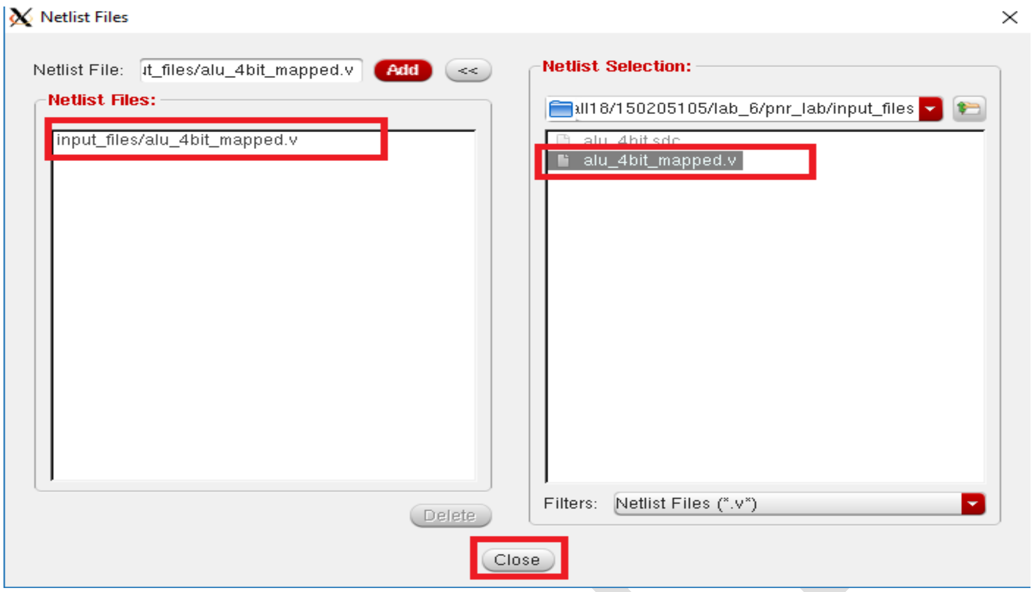
13. In the **Design import** window, select the **Verilog** option under the **Netlist** section and then click on the three dots **(...)** button for importing the synthesized netlist file to the database.



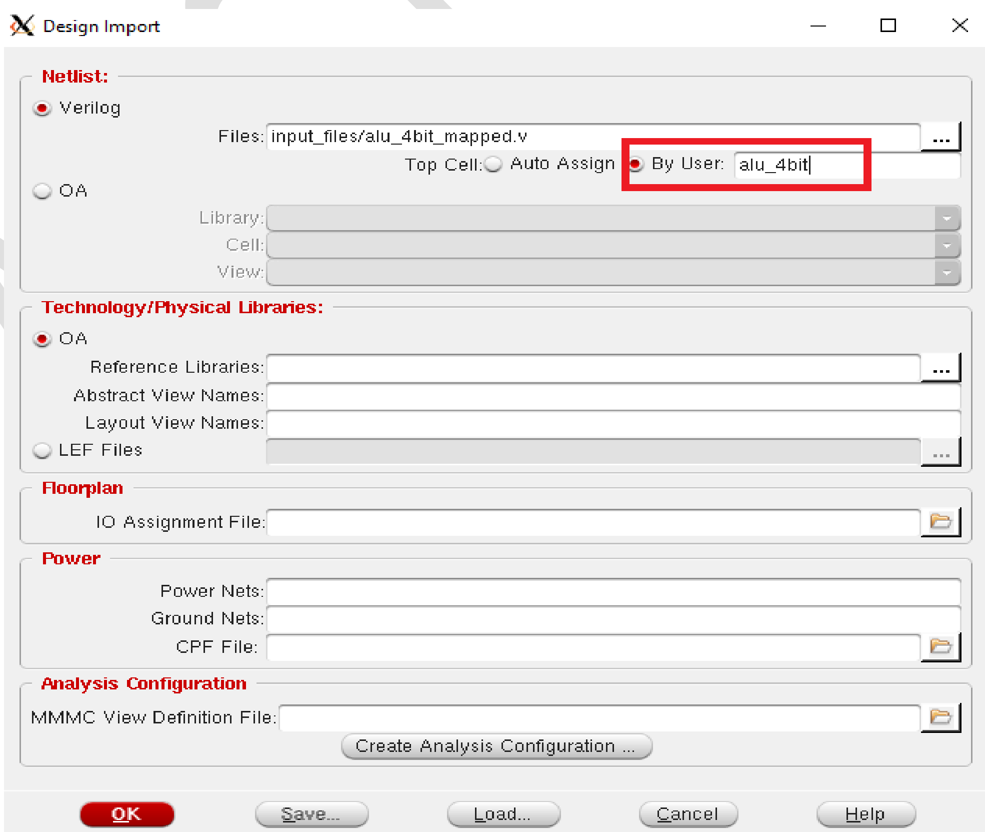
14. Now the **Netlist File** window will appear, click on the double arrow button **(>>)**.



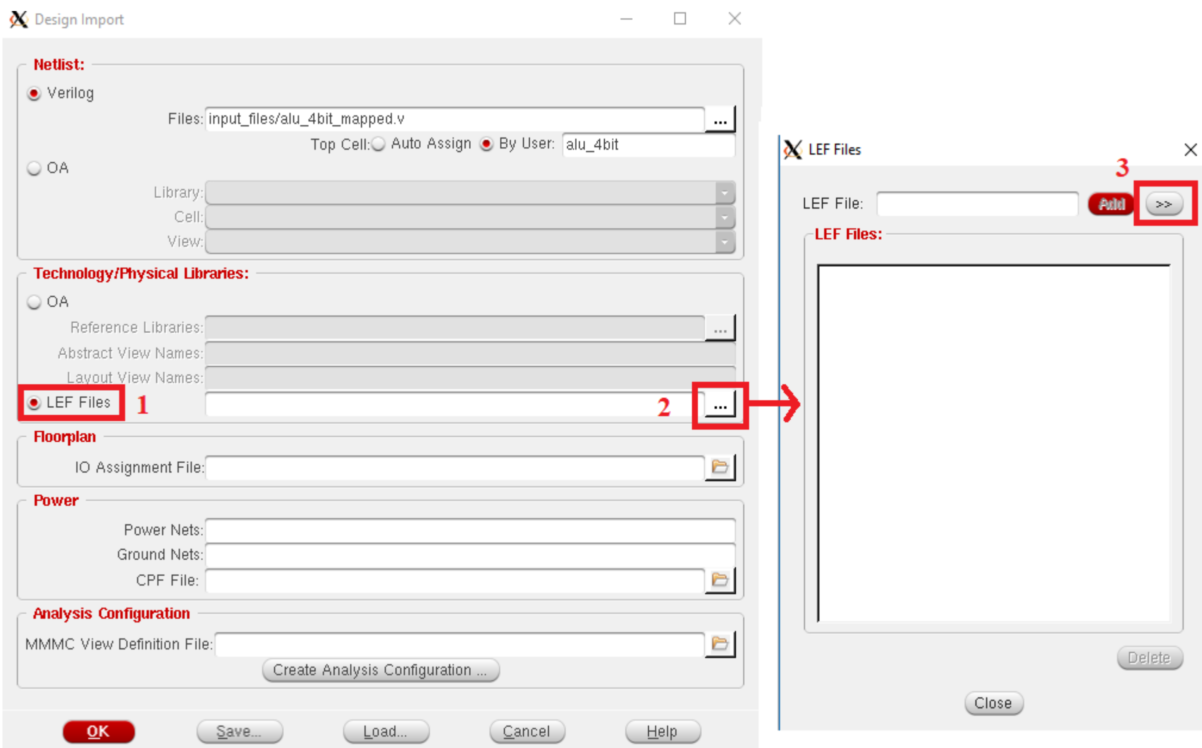
15. In the newly appeared **Netlist Files** window, select the synthesized netlist file ***alu_4bit_mapped.v*** from the ***input_files*** directory. Then click on the **Close** button.



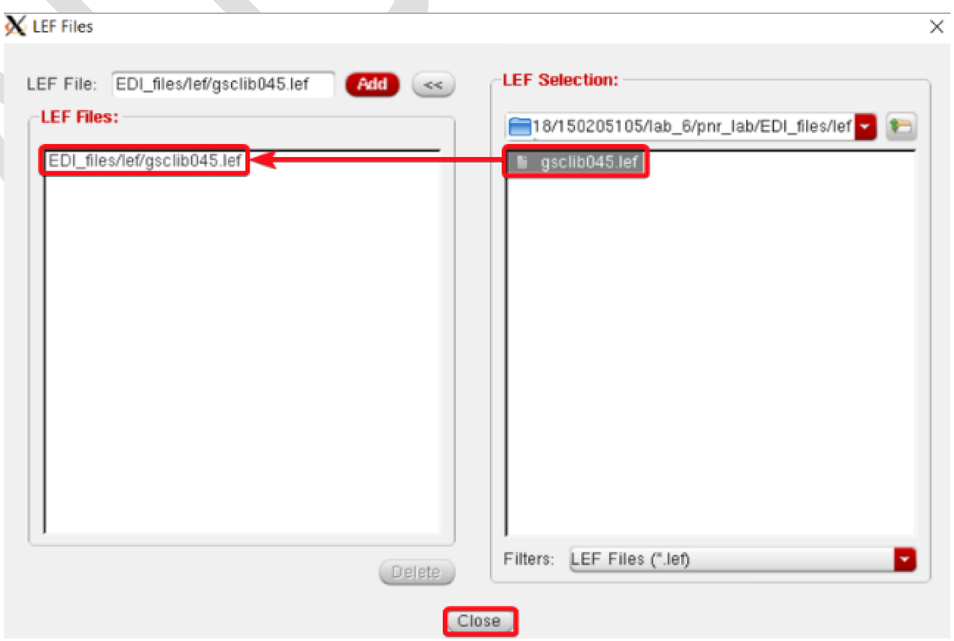
16. If the netlist importing is successful, you will be in the **Design Import** window again. Now to define the **Top Cell** name select the **By User** option and provide the cell name ***alu_4bit*** in the blank field as shown in the following figure.



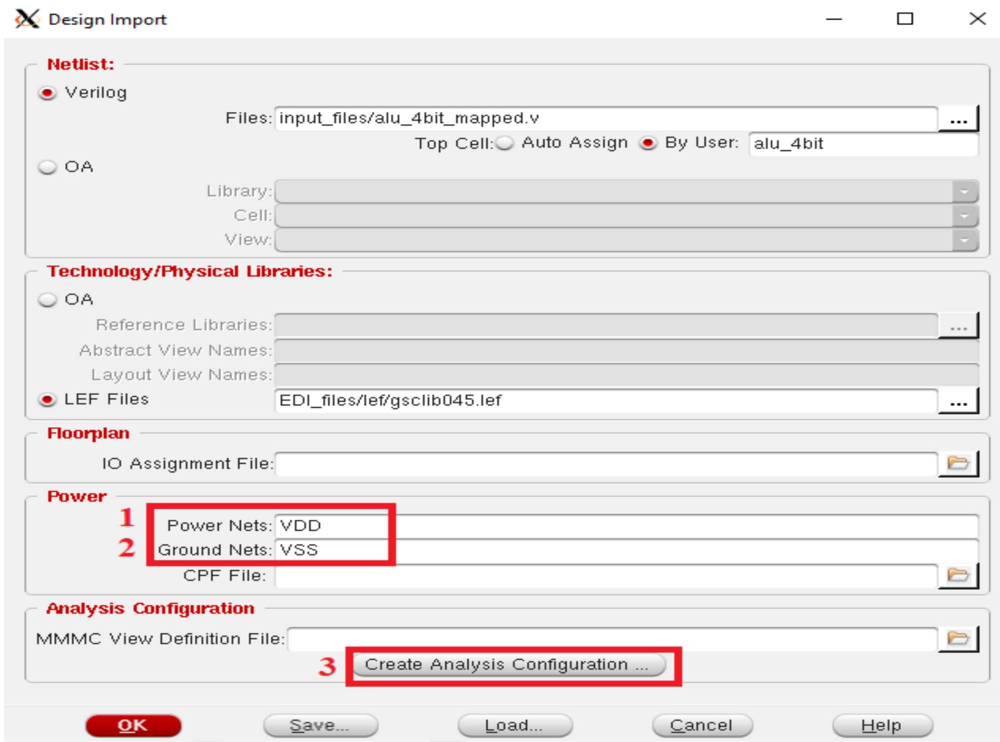
17. For adding lef files to the database, select the **LEF Files** option under the **Technology/Physical Libraries** section of the **Design Import** window. Click on the three dots **(…)** button beside the **LEF Files** option**.** Then on the appeared **LEF Files** window click on the **arrow (>>)** button.



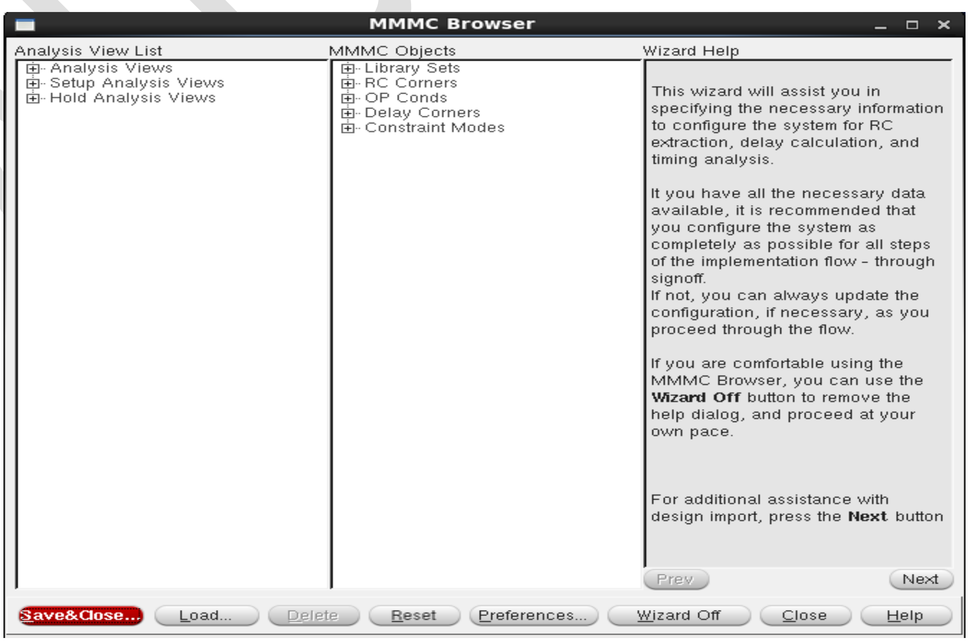
18. In the **LEF Files** window find and select the lef file ***gsclib045.lef*** from the ***EDI_file/lef*** directory and then click on the **Close** button.



19. Now, on the **Power** section of the **Design Import** window, write **Power Nets** name as **VDD** and **Ground Nets** name as **VSS** as shown in the below figure. After that click on the **Create Analysis Configuration** option for creating the MMMC file.



20. The following blank **MMMC Browser** window will appear. We will set MMMC objects and will create appropriate analysis views for our physical design.



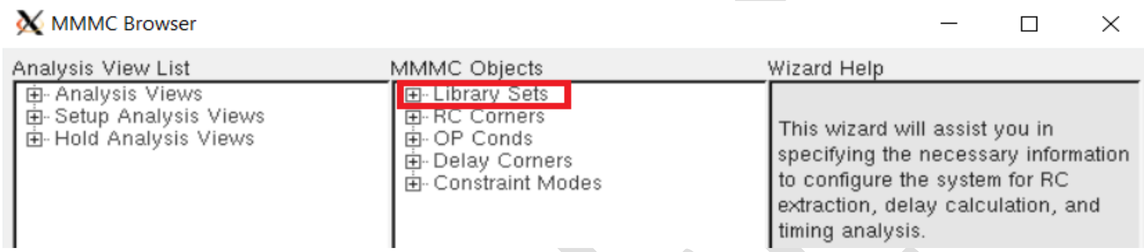
Library Sets

We will create two **Library Sets** using slow and fast timing library files as shown in Table 1.

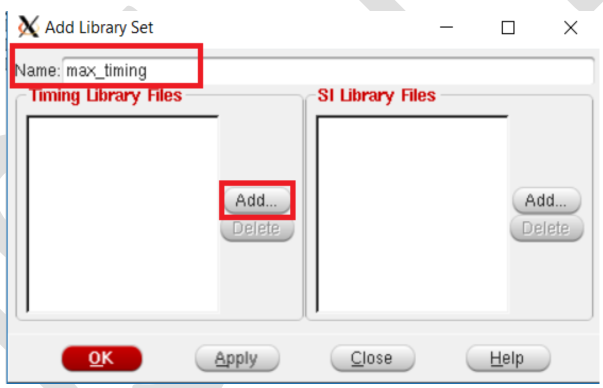
Table-1

Name	Timing library file Directory
max_timing	EDI_files/libs/slow.lib
min_timing	EDI_files/libs/fast.lib

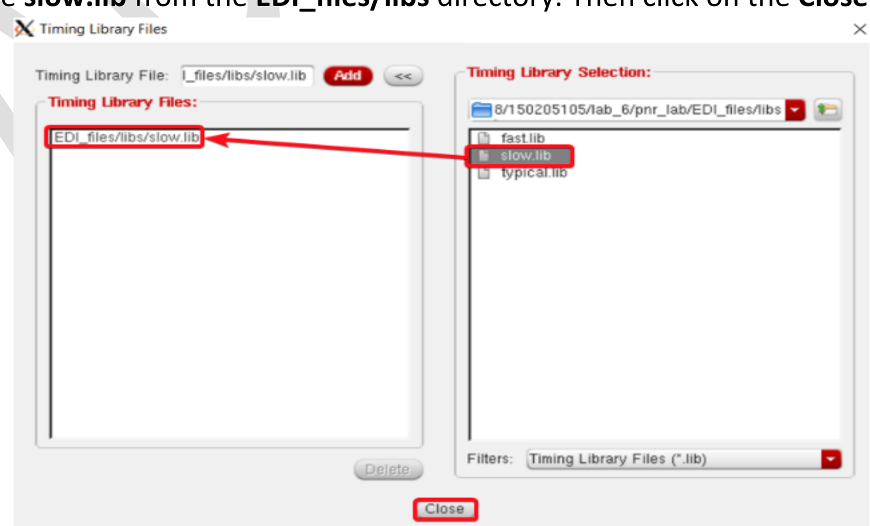
21. To create a library sets, double click on the **Library Sets** option of the **MMMC Browser** to launch the **Add Library Set** window.



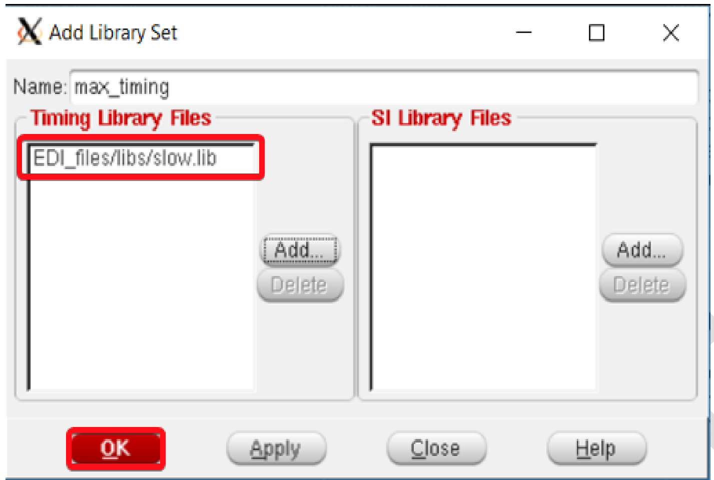
22. In the **Add Library Set** window, write ***max_timing*** in the name field and click on the **Add** button.



23. The **Timing Library Files** window will appear. Click on the double arrow **(>>)** button and select the **slow.lib** from the **EDI_files/libs** directory. Then click on the **Close** button.

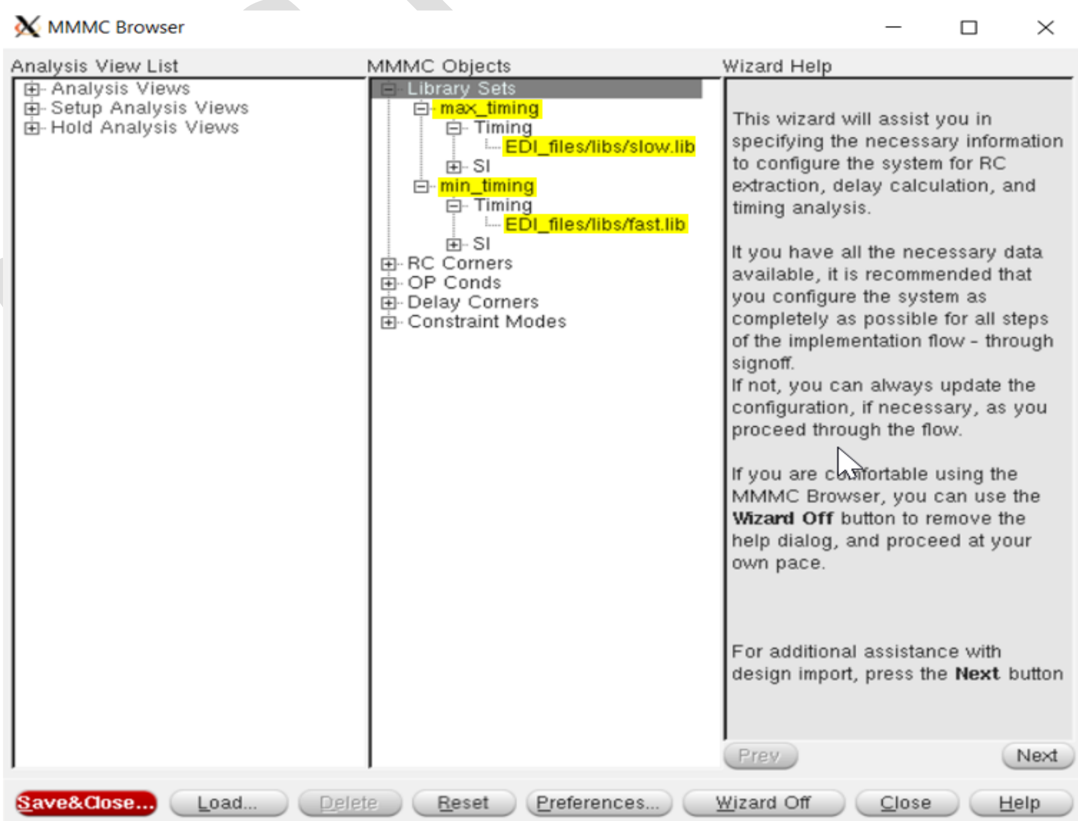


24. Now in the **Add Library Set** window select the **OK** button. A library set will be created named ***max_timing*** which contains **EDI_files/libs/slow.lib**



25. Follow steps 21-24 to create the ***min_timing*** library set by selecting the ***fast.lib****.*

26. After successfully creating two libraries the MMMC browser will look like the below figure.



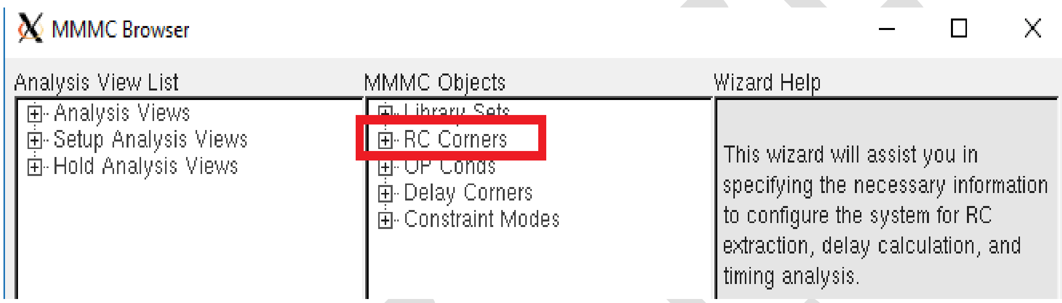
RC Corners

Now, we will create an **RC Corner** using the Cap Table file as shown in Table 2.

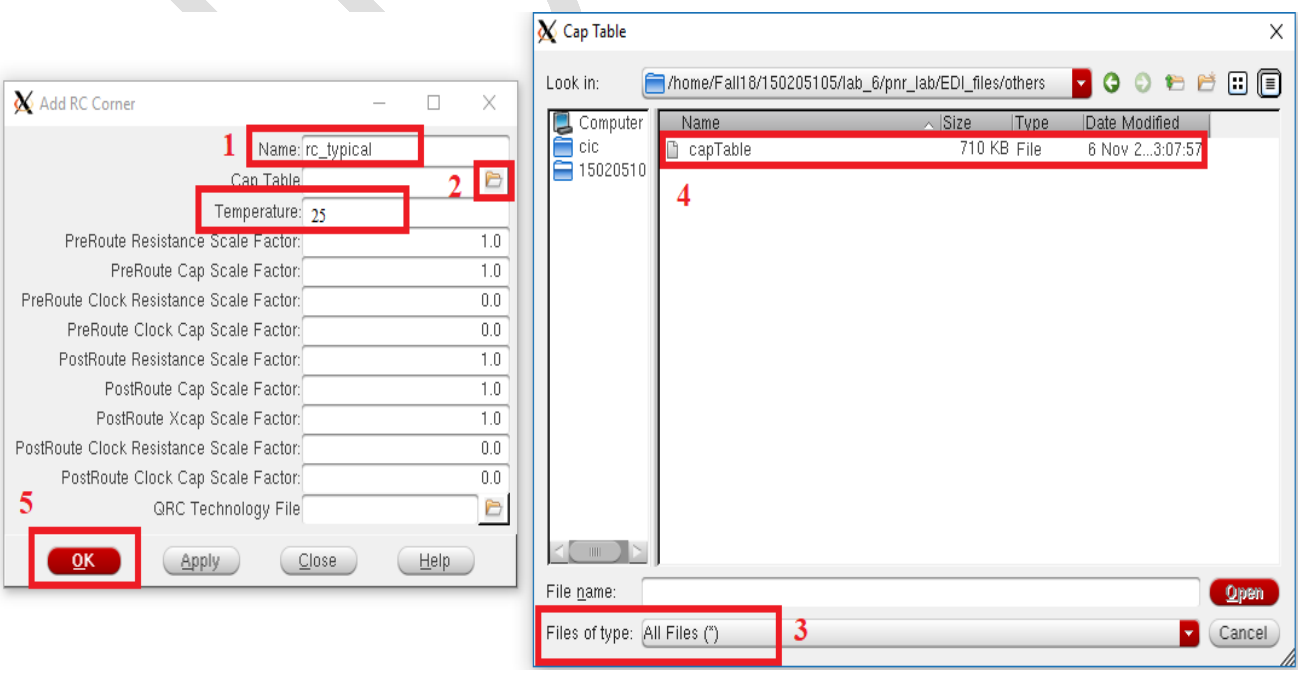
Table-2

Name	Cap Table	Temperature
rc_typical	EDI_files/others/capTable	25

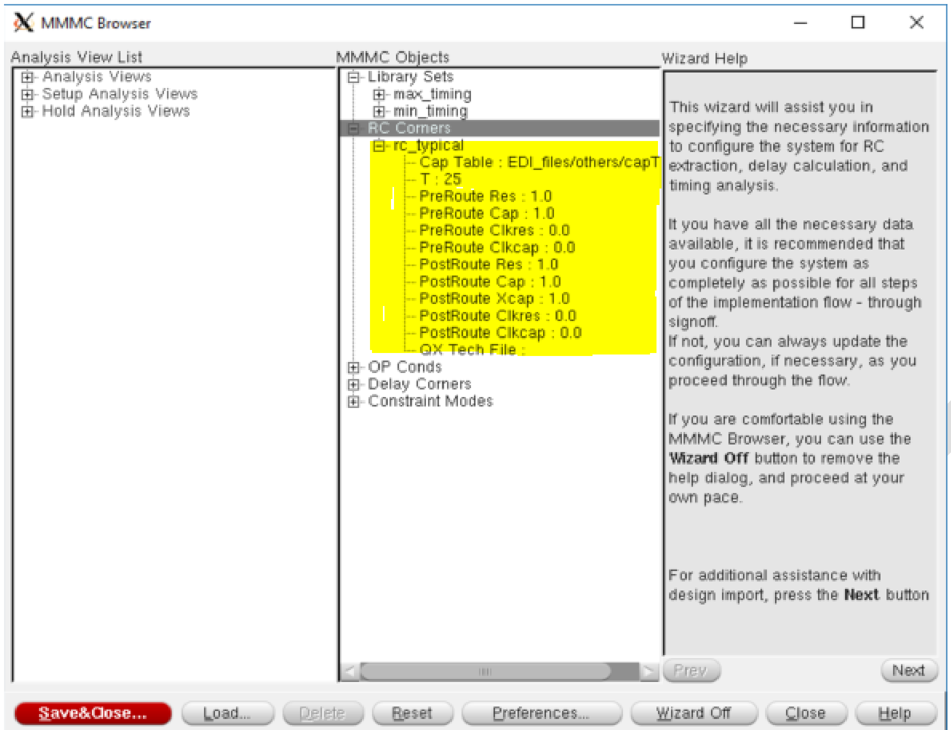
27. To create an RC corner, double click on the **RC Corners** option of the **MMMC Browser** to launch the **Add RC Corner** window.



28. In the **Add RC Corner** window, write ***rc_typical*** in the **Name** field and **25** in the **Temperature** field and select the **Cap Table** from the location ***EDI_files/others/capTable.*** After that click on the **OK** button. If the capTable file is not found in the mentioned location select the **File of type** as **All Files(*)** from the **Cap Table** window.



29. After successfully creating the **RC Corner** the **MMMC browser** will look like the below figure.



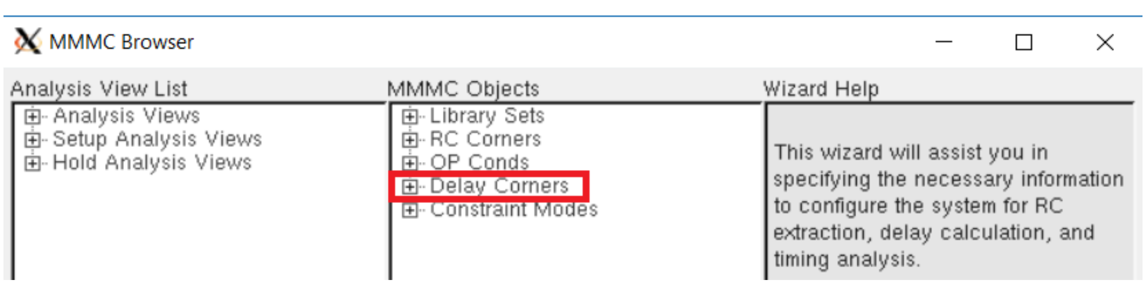
Delay Corners

Now, we will create two different **Delay Corners** using the ***max_timing*** and ***min_timing*** library sets and the ***rc_typical*** RC Corner as shown in Table-3.

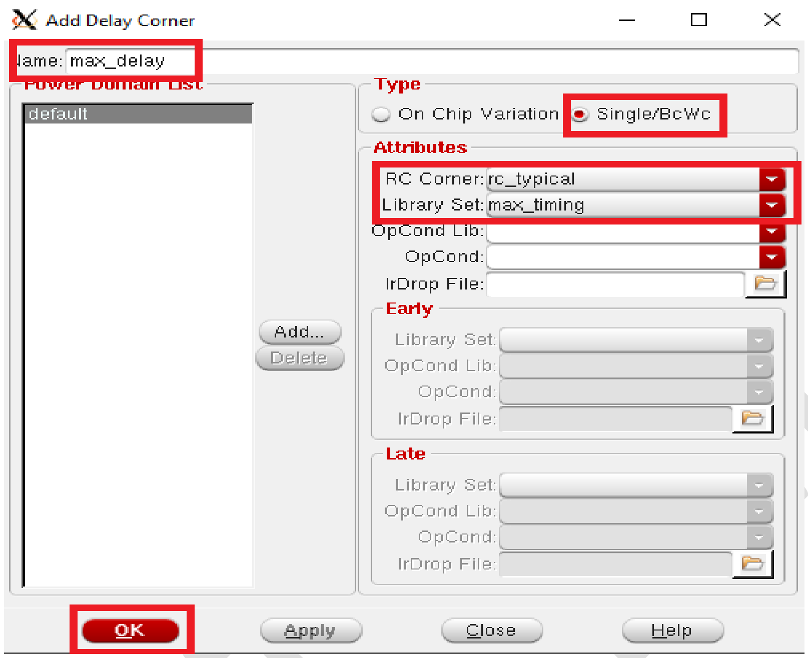
Table-3

Name	Type	RC Corner	Library Set
max_delay	Single Bc/Wc	rc_typical	max_timing
min_delay	Single Bc/Wc	rc_typical	min_timing

30. To create a delay corner, double click on the **Delay Corners** option of the **MMMC Browser** to launch the **Add Delay Corner** window.

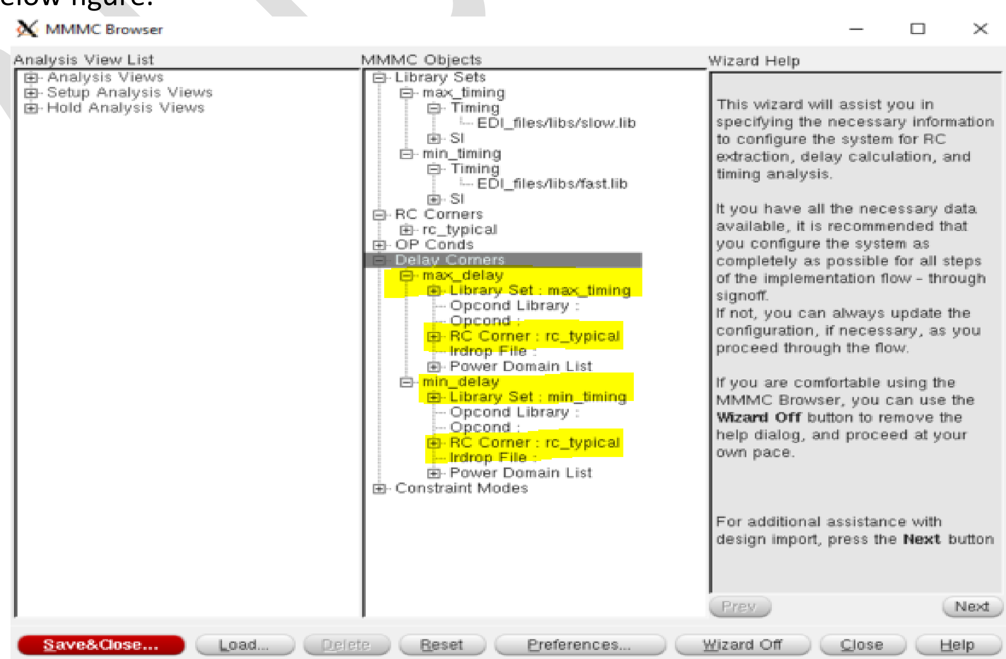


31. In the **Add Delay Corner** window, write ***max_delay*** in the name field, select the type **Single Bc/Wc,** choose ***rc_typical*** from the **RC Corner** option, and ***max_timing*** from the **Library Set** option. Then, click on the **OK** button.



32. Follow steps 27-28 and create the ***min_delay*** delay corner by selecting the ***min_timing*** from **Library Set** and ***rc_typical*** from RC Corner.

33. After successfully creating all two delay corners, the **MMMC Browser** will look like the below figure.



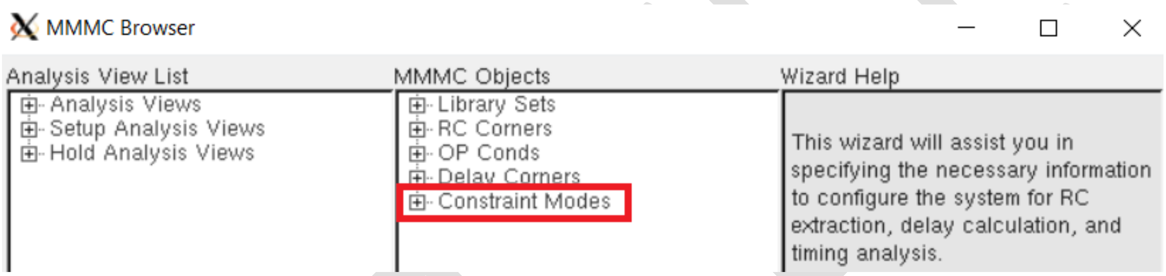
Constraint Mode

In this part, a constraint mode will be created from the post synthesis constraint file(SDC file) using the **Constraint Mode** option as shown in Table-4.

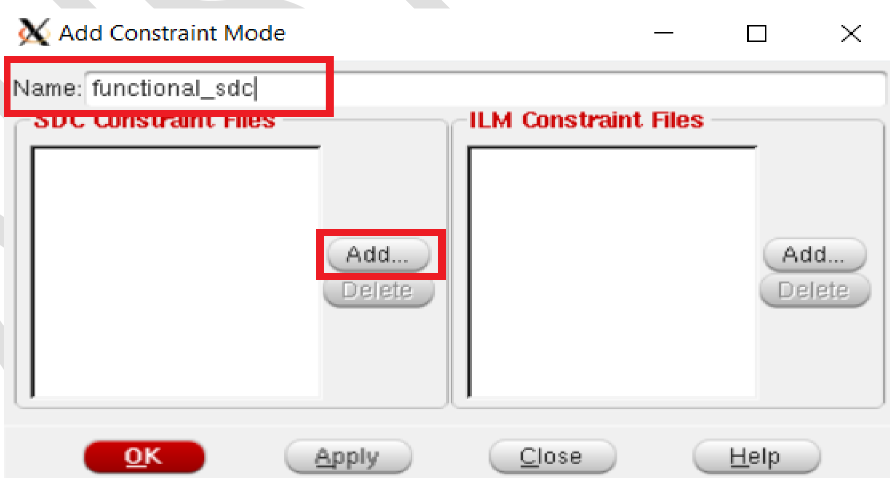
Table-4

Name	SDC constraint files
fuctional_sdc	Input_files/alu_4bit.sdc

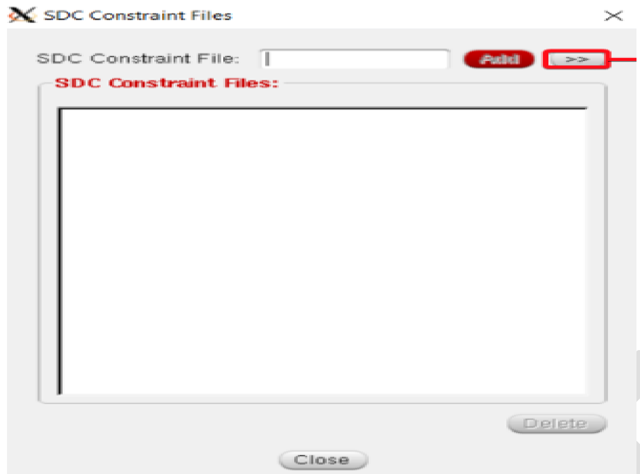
34. Double click on the **Constraint Modes** option of the **MMMC Browser** to open **Add Constraint Mode** window.



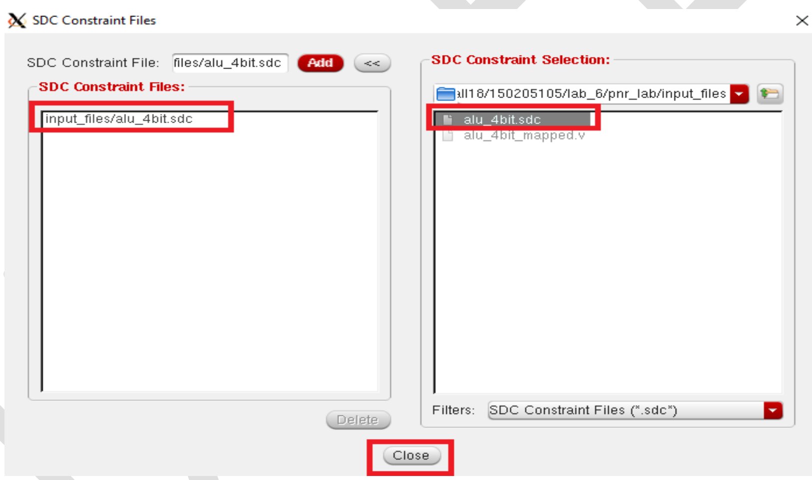
35. **In the Add Constraint Mode** window, write **fuctional_sdc** on the name field and click on the **Add** button.



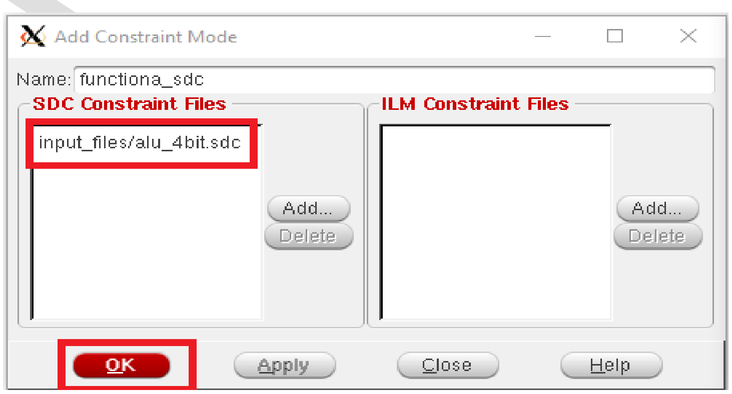
36. Now the **SDC Constraints Files** window will appear. Click on the double arrow button (**>>**)



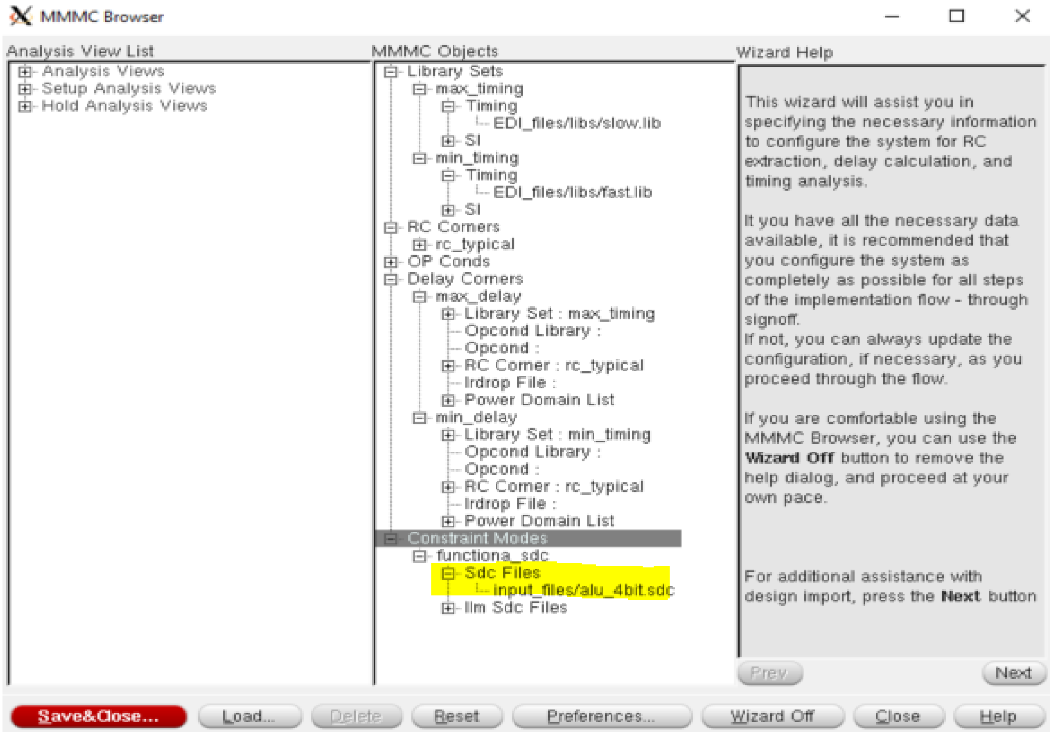
37. Select the ***alu_4bit.sdc*** from the **input_files** directory. After that click on the **Close** button.



38. Then click **OK** on **Add Constraint Mode** window.



39. After successfully creating the constraint mode, a constraint mode named ***functional_sdc*** is created in the **MMMC Browser**.



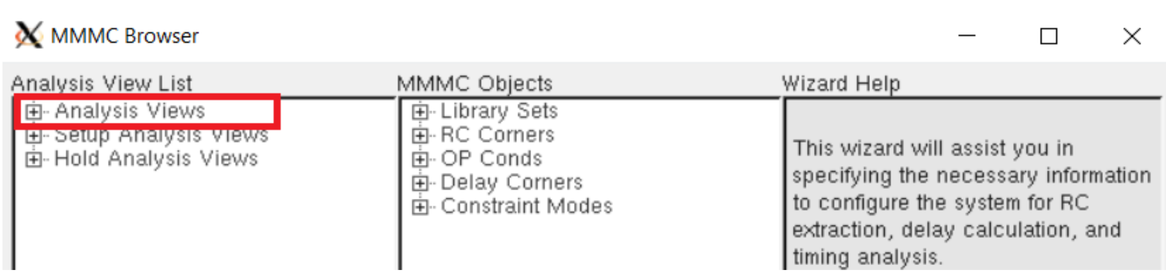
Analysis Views

We will create two different **Analysis Views** using the previously created ***max_delay*** and ***min_delay*** delay corners and constraint mode ***fuctional_sdc*** as shown in Table-5.

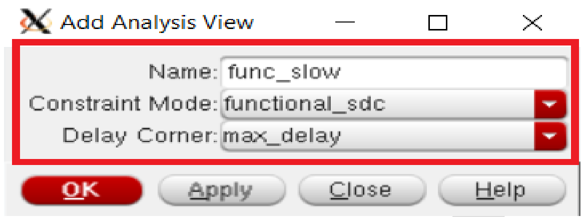
Table-5

Name	Constraint Mode	Delay Corner
func_slow	fuctional_sdc	max_delay
func_fast	fuctional_sdc	min_delay

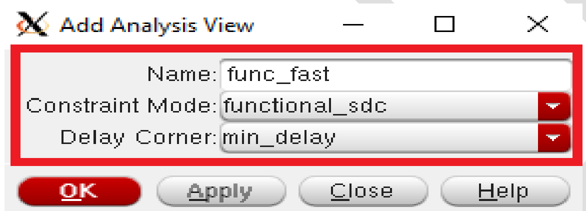
40. To create an analysis view double, click on the **Analysis Views** option of the **MMMC Browser** to launch the **Add Analysis View** window.



41. In the **Add Analysis View** window write ***func_slow*** in the name field**,** select ***functional_sdc*** from the **Constraint Mode** option, ***max_delay*** from the **Delay Corner** option, and after that press **Ok**.



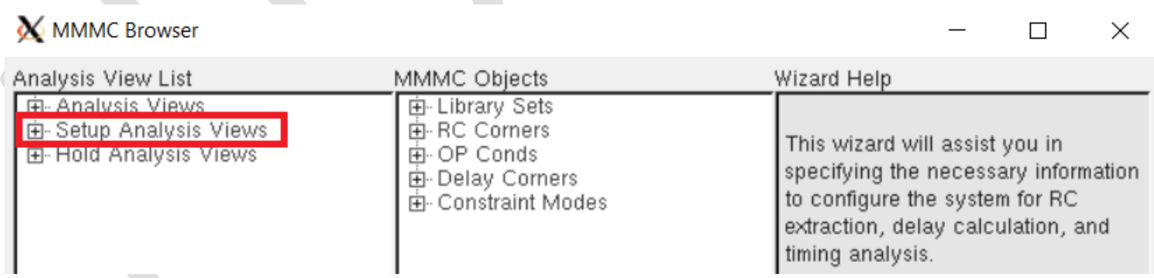
42. Follow steps 35-36 and create the ***func_fast*** analysis view by selecting the ***functional_sdc*** from the **Constraint Mode** option and ***max_delay*** from the **Delay Corner** option.



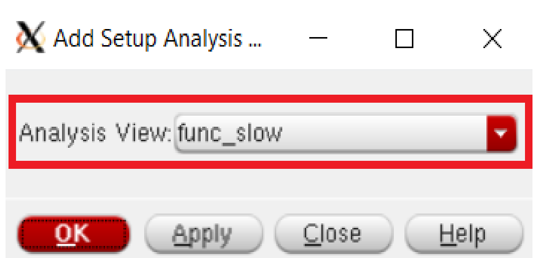
Setup and Hold Analysis View

We will specify setup and hold analysis views using the ***func_slow*** and ***func_fast*** analysis views created in the previous steps.

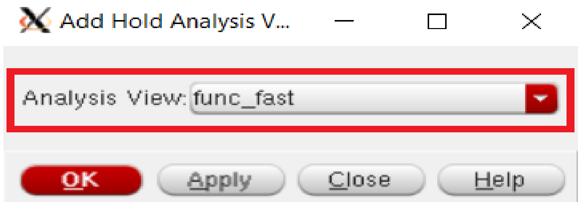
43. To specify the setup analysis view, double click on the **Setup Analysis View** option on the **MMMC Browser** to launch the **Add Setup Analysis**.



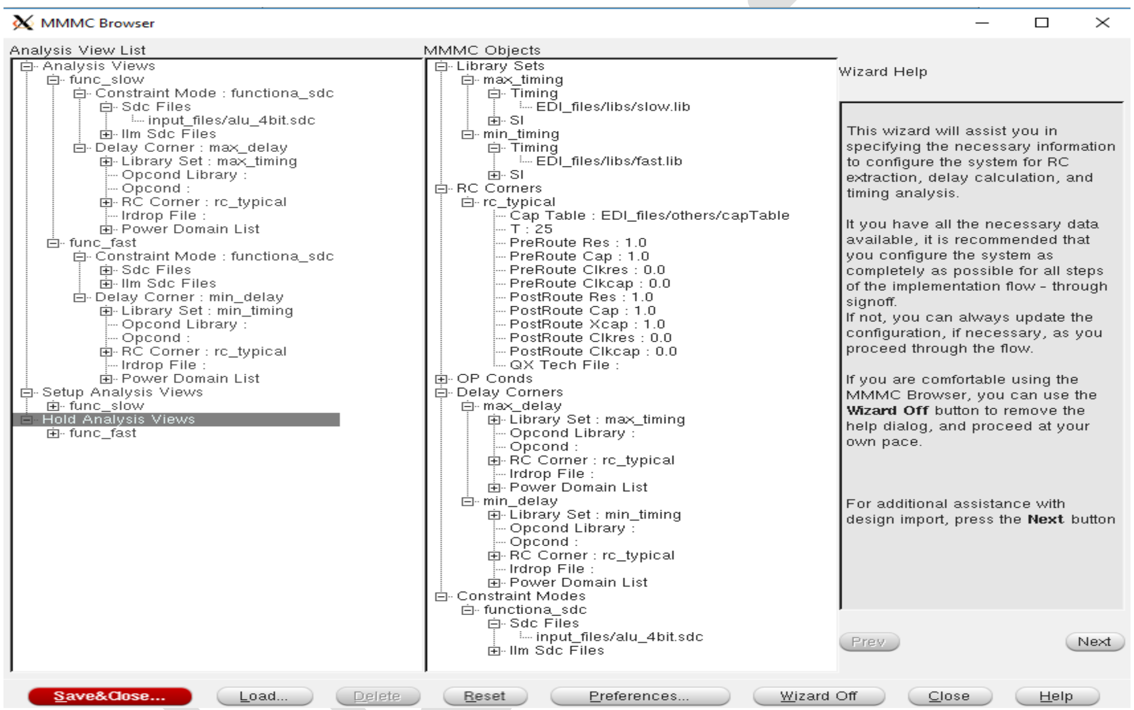
44. In the **Add Setup Analysis View** window select the ***func_slow*** from the **Analysis View** and press **Ok**.



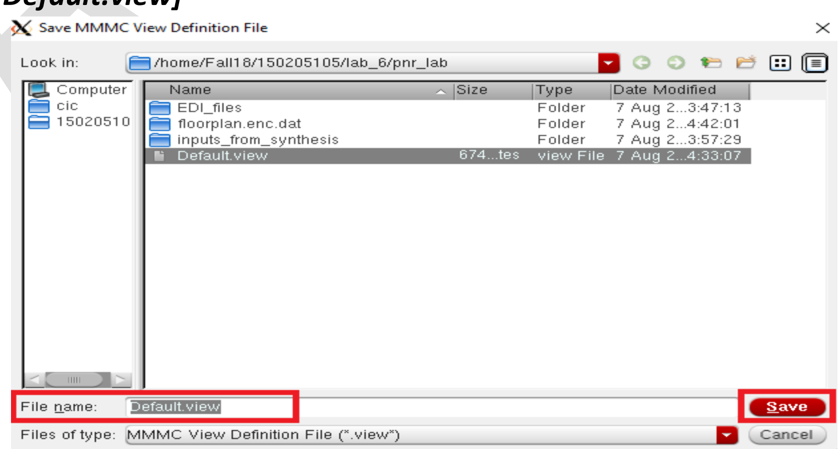
45. Following steps 38-39 and specify the **Hold Analysis View** option by selecting the ***func_fast*** **Analysis View.**



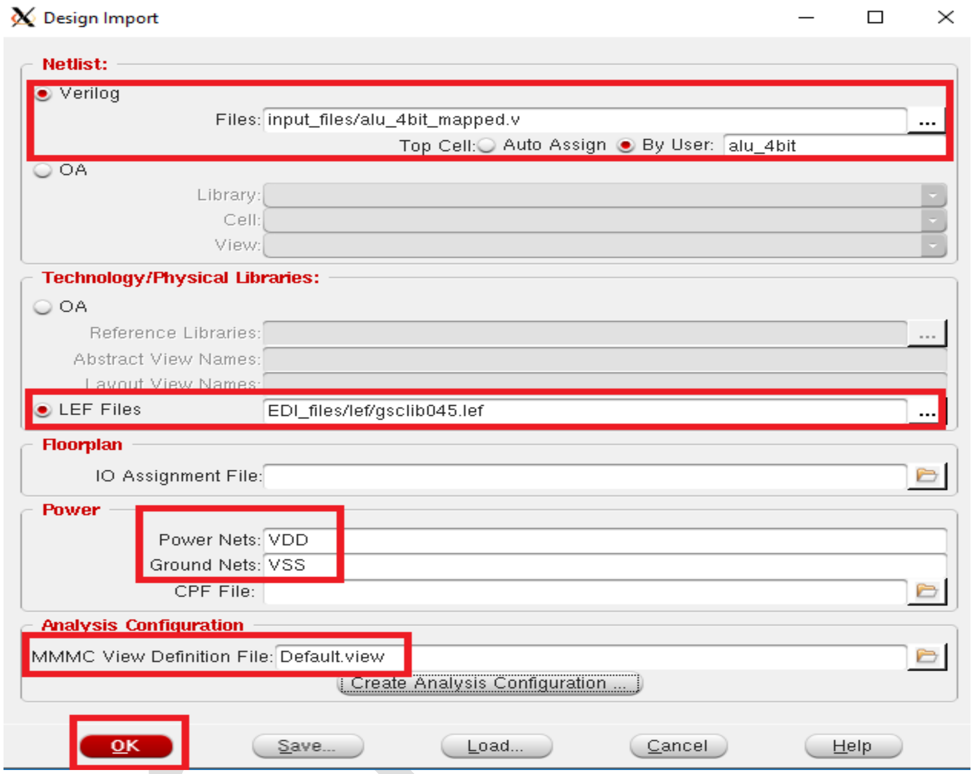
46. After adding all analysis views, make sure your **MMMC Browser** looks like the below figure, and then click on the **Save&Close** button.



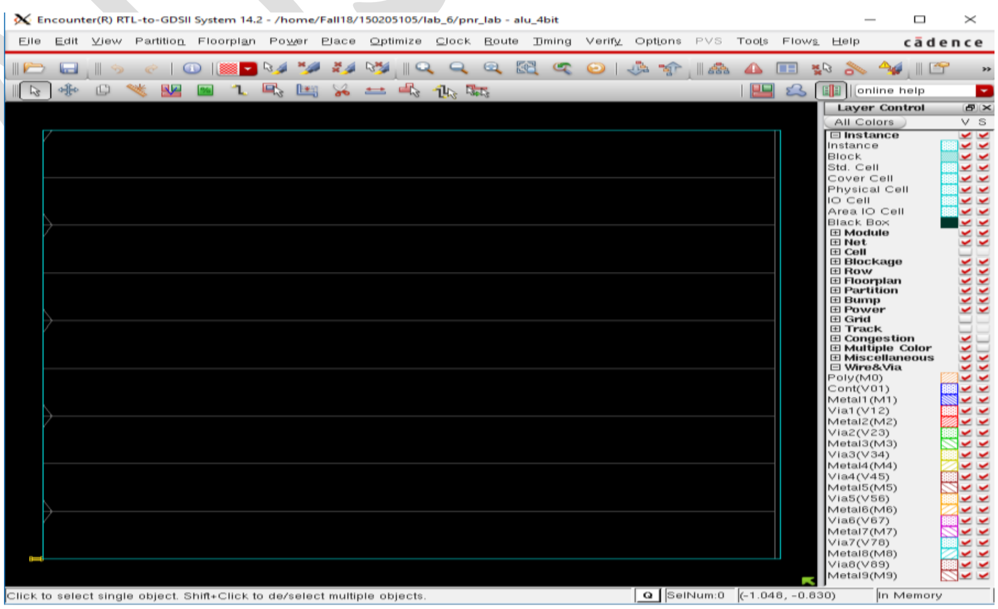
47. The **Save MMMC Browser View Definition File** window will appear. To save all the steps of the MMMC browser provide a file name and click on the **Save** button.*[Here, we used the name **Default.view**]*



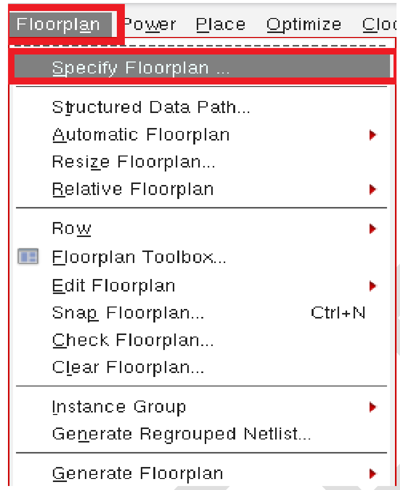
48. Now, make sure your final **Design Import** window looks like the following figure and then click on the **OK** button.



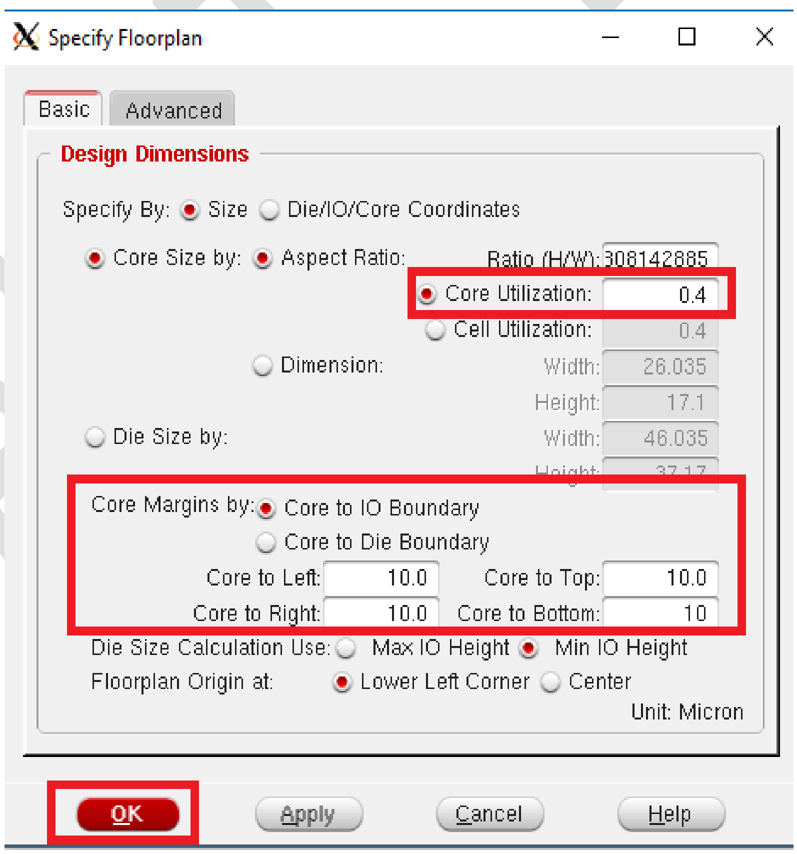
49. An Encounter window will appear on your screen. The window has multiple rows like the following figure which ensure that the database has been created perfectly and ready for floorplanning.



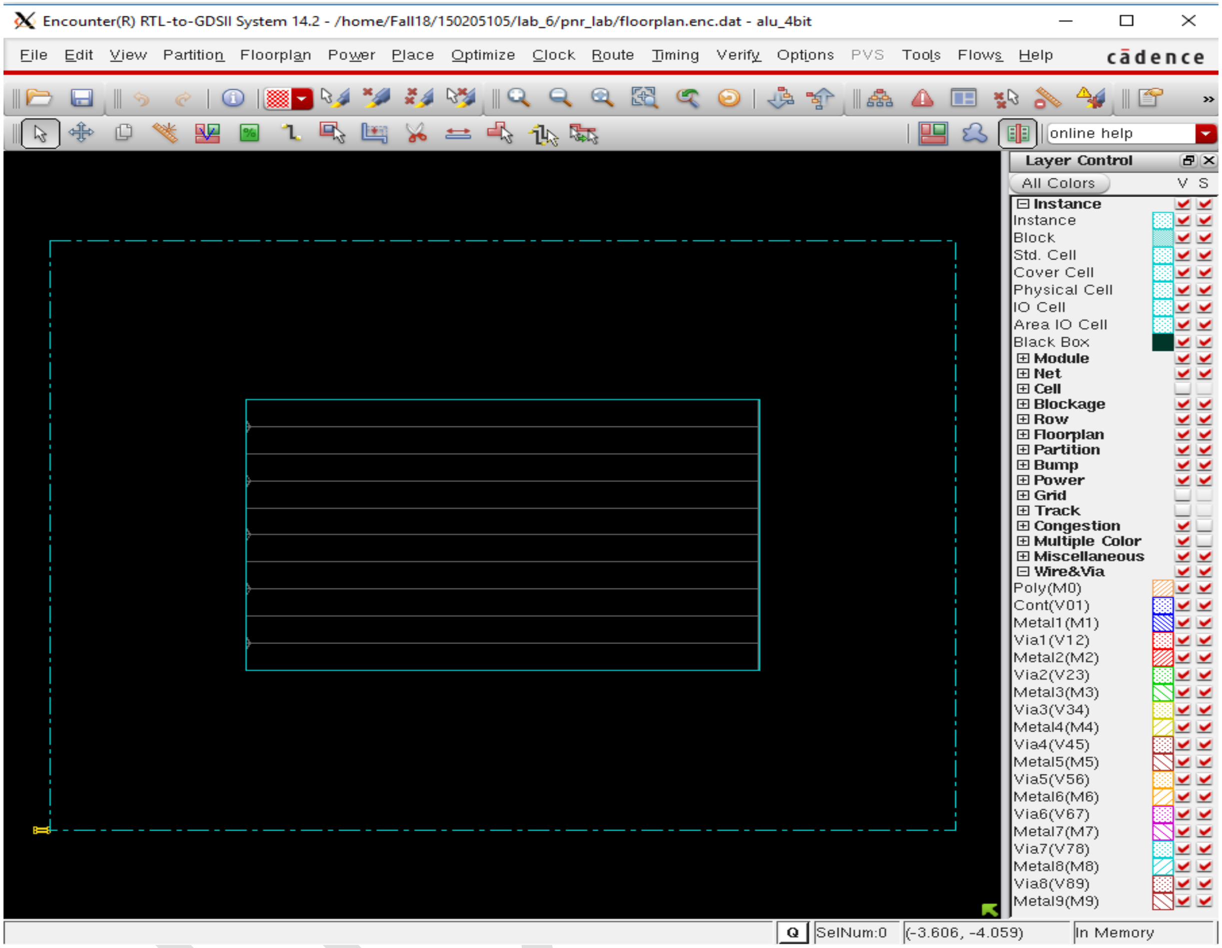
50. For floorplanning, execute **Floorplan → Specify Floorplan**.



51. Now in the **Specify Floorplan** window, set **Core Utilization** 0.4 and select the option **Core to IO Boundary** from the **Core Margin By** section and put 10 to all the four blank spaces (**Core to Left, Core to Top, Core to Right, Core to Bottom**). No need to change the rest of the value.



52. After successfully specifying all the values, the following floorplan will appear on the encounter window.



53. Now save the design as an encounter database file using the following command in the encounter terminal.

encounter 1> *saveDesign floorplan.enc*

Post Lab Task

1. What are the functions of the **MMMC** browser?
2. What does **Cap Table** contain?
3. What are the core area and die area?
4. What is the concept of rows in the floor plan?
5. What is constraint mode and how does it control the whole ASIC design?
6. What are the **PVT** corner and **RC** corner?
7. How is utilization calculated?
8. Why do we check the setup in the slow corner and hold in the fast corner?
9. Check all the options of **Specify Floorplan** window.

Lab-7A: Physical Design Using Encounter Digital Implementation System (Part 2)

Objective

The main objectives of this lab are:

- Familiarization with power mesh creation.
- Familiarization with standard cell placement techniques.

Lab Task

In the last lab, we prepared the design import settings and created a floorplan for our design. In this lab, we will perform the rest of the stages of PnR for completing our physical design

1. Log in to the server in GUI mode and source the Cadence license file.
   [**Xlaunch (enable SSH)→putty (load server IP) → login → csh→ source ~/cshrc_q→ nautilus]**

2. In the GUI mode of your account open a terminal by executing **right-click on mouse → open terminal.**
3. First check you are at the home using the command ***pwd***

   [150205105@aust ~]$ pwd

4. Go to the directory ***lab_6*** executing the command ***cd lab_6/***

   [150205105@aust ~]$ *cd lab_6/*

5. Go to the directory ***pnr_lab*** where you have done ***lab_6*** experiment and saved your design up to the floorplan

   [150205105@aust lab_6]$ *cd pnr_lab*

6. Make sure that the ***floorplan.enc*** encounter database are present in the ***pnr_lab*** directory using the command ***ls -ltr***

   [150205105@aust pnr_lab]$ *ls -ltr*

7. Now make sure you are in the ***pnr_lab*** directory. And launch the Encounter tool using the command ***encounter***.

   [150205105@aust pnr_lab]$ *encounter*

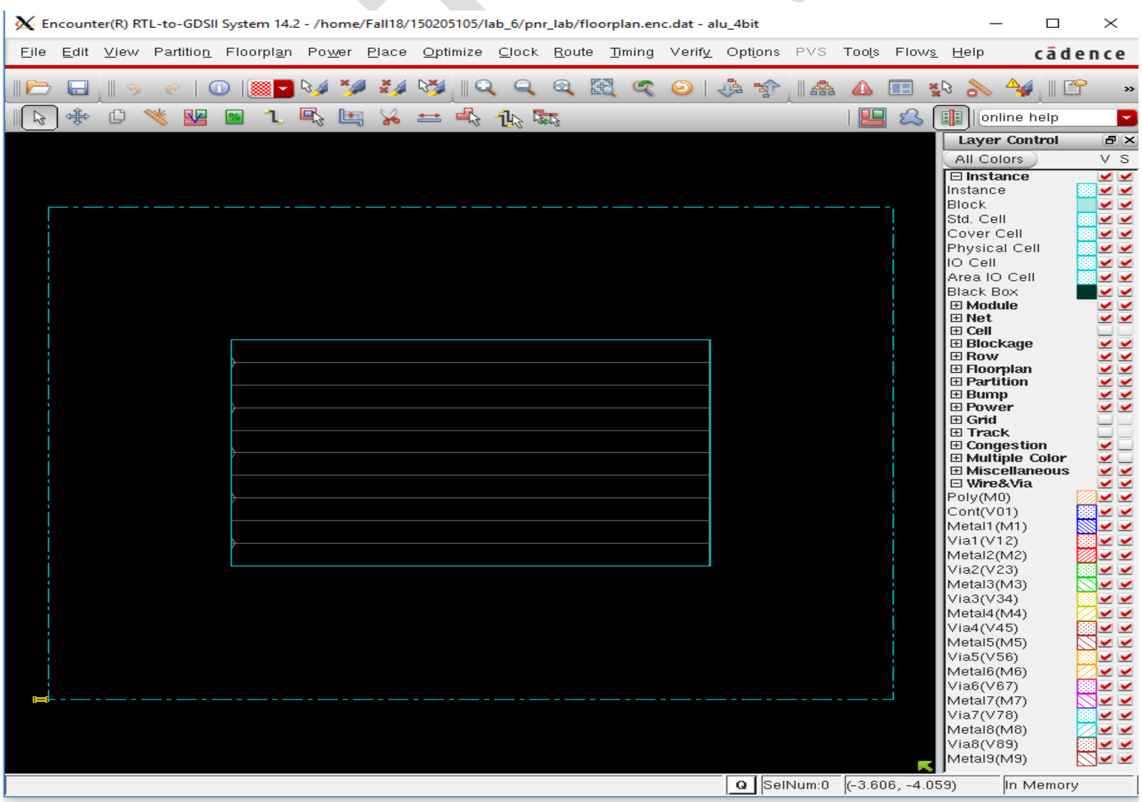
8. If the **encounter** tool is successfully launched, the following text will be shown in the terminal.

```
*                                                                    *
*                   Cadence Design Systems, Inc.                     *
*                      2655 Seely Avenue                             *
*                     San Jose, CA 95134,   USA                      *
*                                                                    *
*                                                                    *
**********************************************************************

@(#)CDS: Encounter v14.20-p004_1 (64bit) 11/05/2014 14:06 (Linux 2.6.18-194.el5)
@(#)CDS: NanoRoute v14.20-p014 NR141101-0648/14_20-UB (database version 2.30, 246.6.1) {superthreading v1.
@(#)CDS: CeltIC v14.20-p001_1 (64bit) 10/15/2014 03:59:12 (Linux 2.6.18-194.el5)
@(#)CDS: AAE 14.20-p007 (64bit) 11/05/2014 (Linux 2.6.18-194.el5)
@(#)CDS: CTE 14.20-p003_1 (64bit) Oct 27 2014 04:09:59 (Linux 2.6.18-194.el5)
@(#)CDS: CPE v14.20-p003
@(#)CDS: IQRC/TQRC 14.1.2-s148 (64bit) Mon Sep 29 16:54:36 PDT 2014 (Linux 2.6.18-194.el5)
@(#)CDS: OA 22.50-p007 Tue Sep 30 00:05:09 2014
@(#)CDS: SGN 10.10-p124 (19-Aug-2014) (64 bit executable)
@(#)CDS: RCDB 11.5
--- Starting "Encounter v14.20-p004_1" on Sun Jul 31 16:40:47 2022 (mem=94.8M) ---
--- Running on aust (x86_64 w/Linux 2.6.18-348.el5) ---
This version was compiled on Wed Nov 5 14:06:30 PST 2014.
Set DBUPerIGU to 1000.
Set net toggle Scale Factor to 1.00
Set Shrink Factor to 1.00000

**INFO:  MMMC transition support version v31-84

encounter 1>
```

9. Now from the encounter terminal, open ***floorplan.enc*** database using the following command.

encounter 1> *source floorplan.enc*

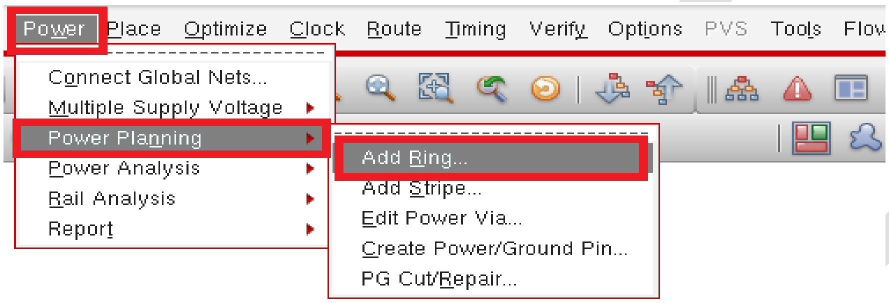
The following floorplan window will appear on your encounter window.



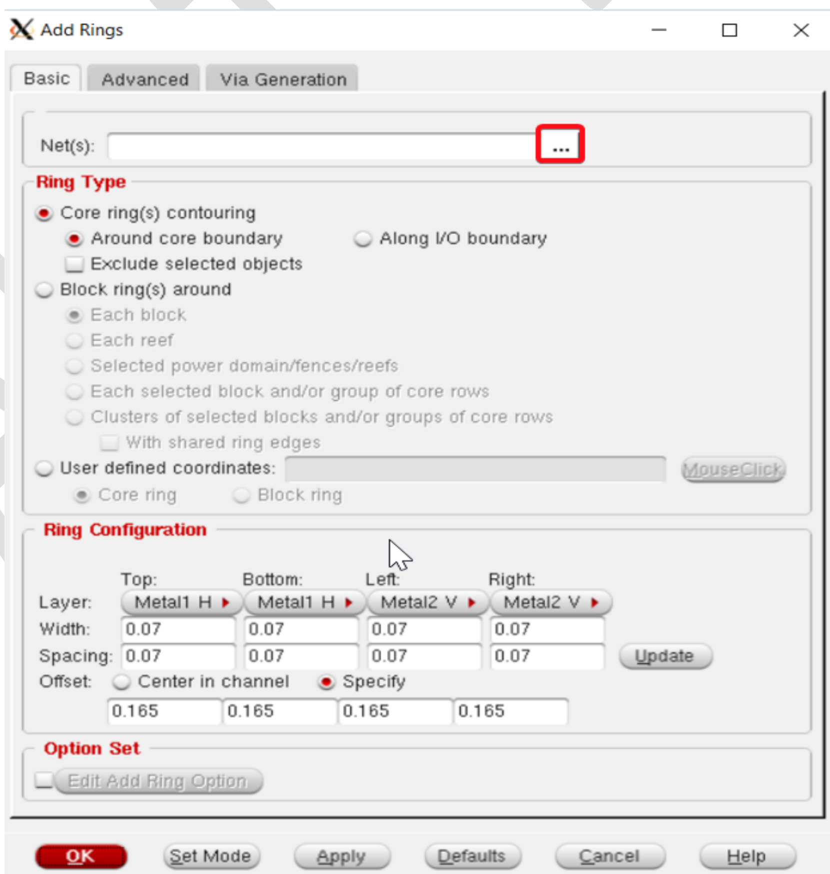
Power Mesh

After restoring ***floorplan.enc*** database to the encounter tool, now the next step is power mesh creation where we will add ***Ring, Stripes***, and ***SRoute*** to the design.

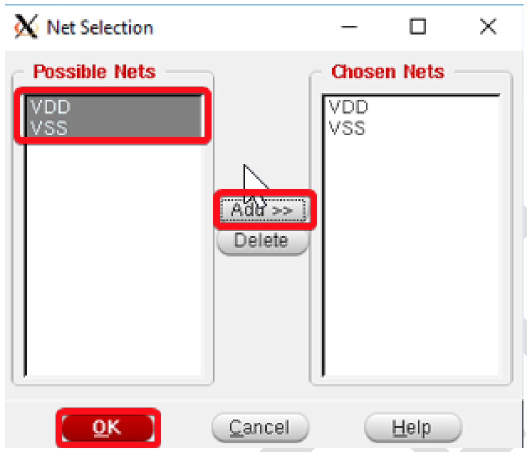
10. Now to add ring to the design, select ***Add Ring*** option by executing ***Power → Power Planning → Add Ring***.



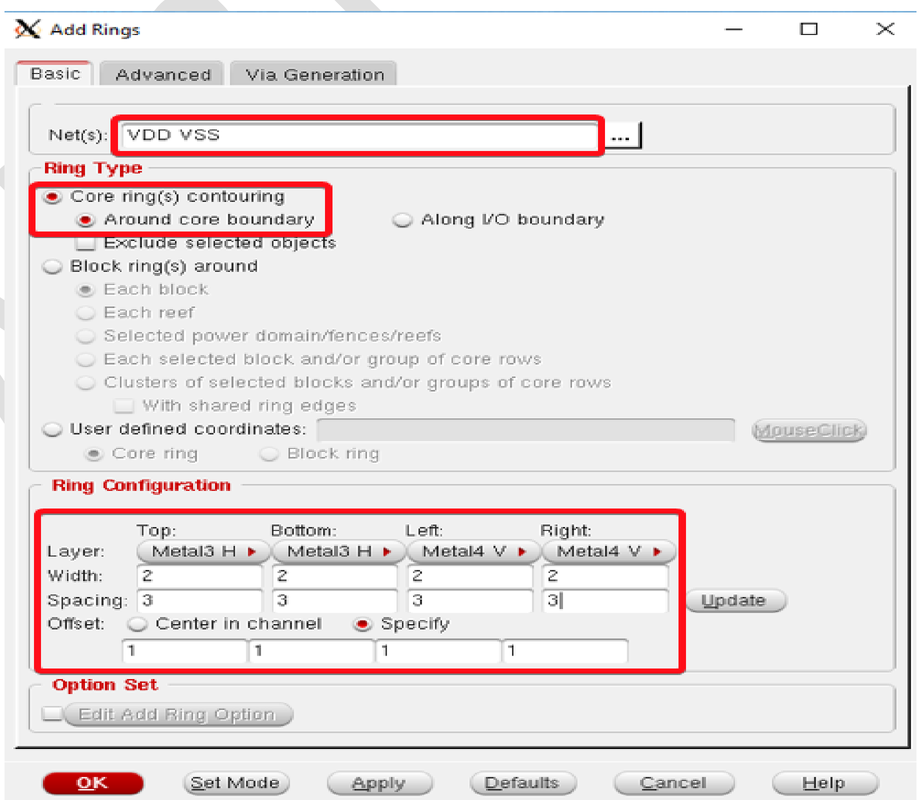
11. In the appeared ***Add Ring*** window, select the ***Basic*** tab and click on the three dots ***(…)*** button beside the ***Net(s)*** field.



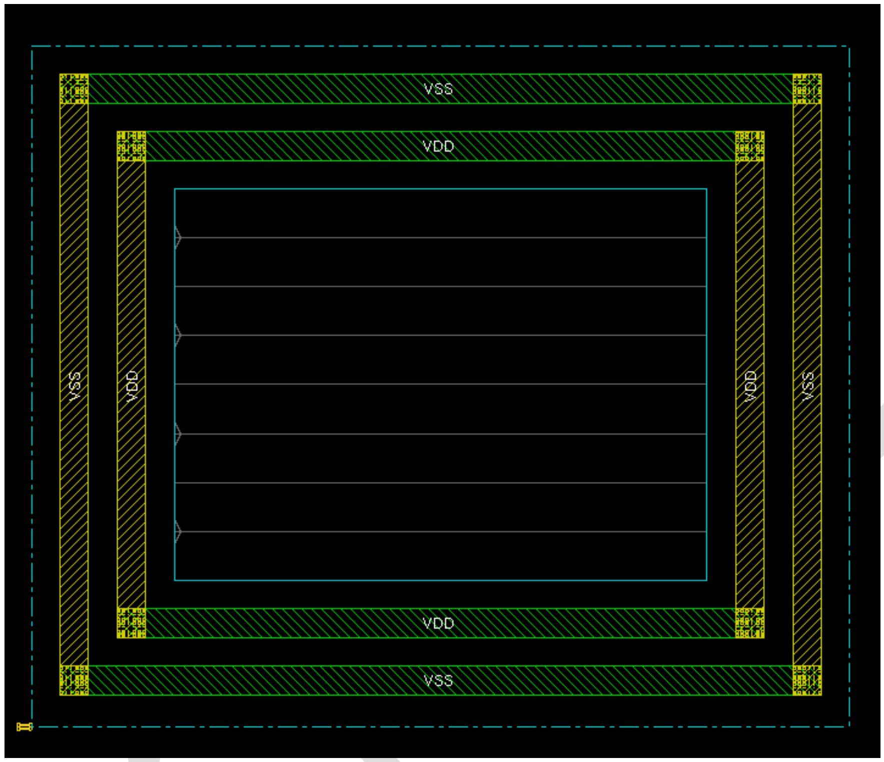
12. From the appeared **Net Selection** window, select both ***VDD*** and ***VSS*** and then click on ***Add*** button. After that click on the ***OK*** button.



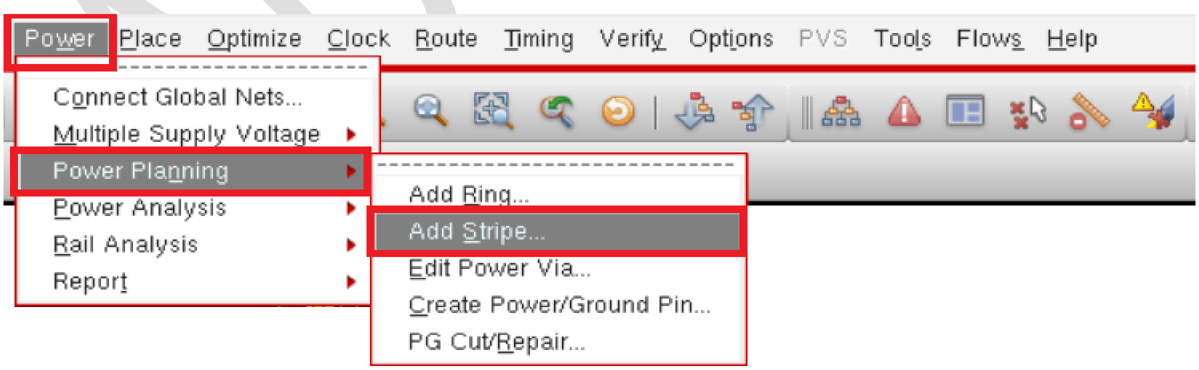
13. After that in the **Basic** tab of **Add Rings** window, check that the ***Around core boundary*** (under the ***Core ring(s) contouring*** option) is selected. Then under the ***Ring Configuration*** section, put all the values on the blank field as same as the below figure. Make sure that the Layer on the Top and Bottom must be ***Metal3 H*** and the Layer on the Left and Right must be ***Metal4 V.***



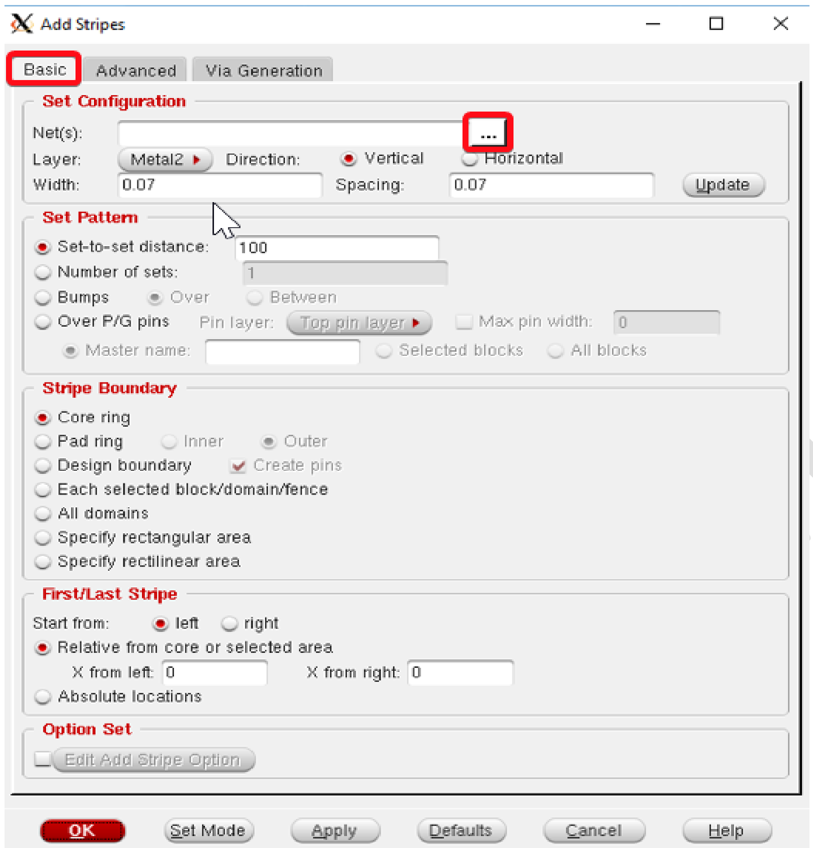
14. Now click on the **OK** button of the **Add Rings** window. You will see the following encounter window where the ring encloses the core area.



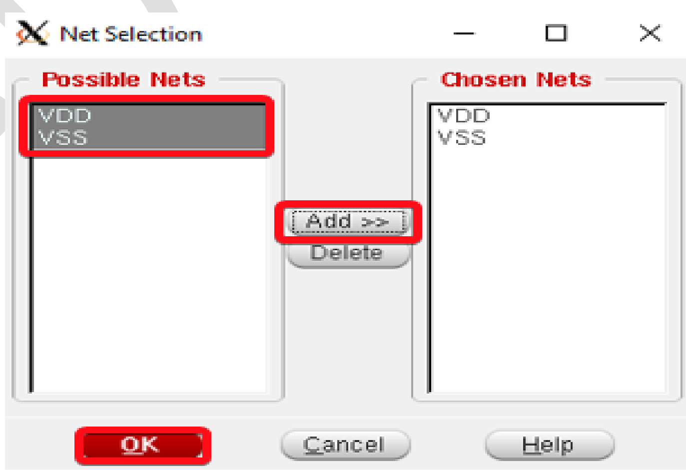
15. To add stripes to the design, select ***Add Stripes*** option by executing ***Power → Power Planning → Add Stripe.***



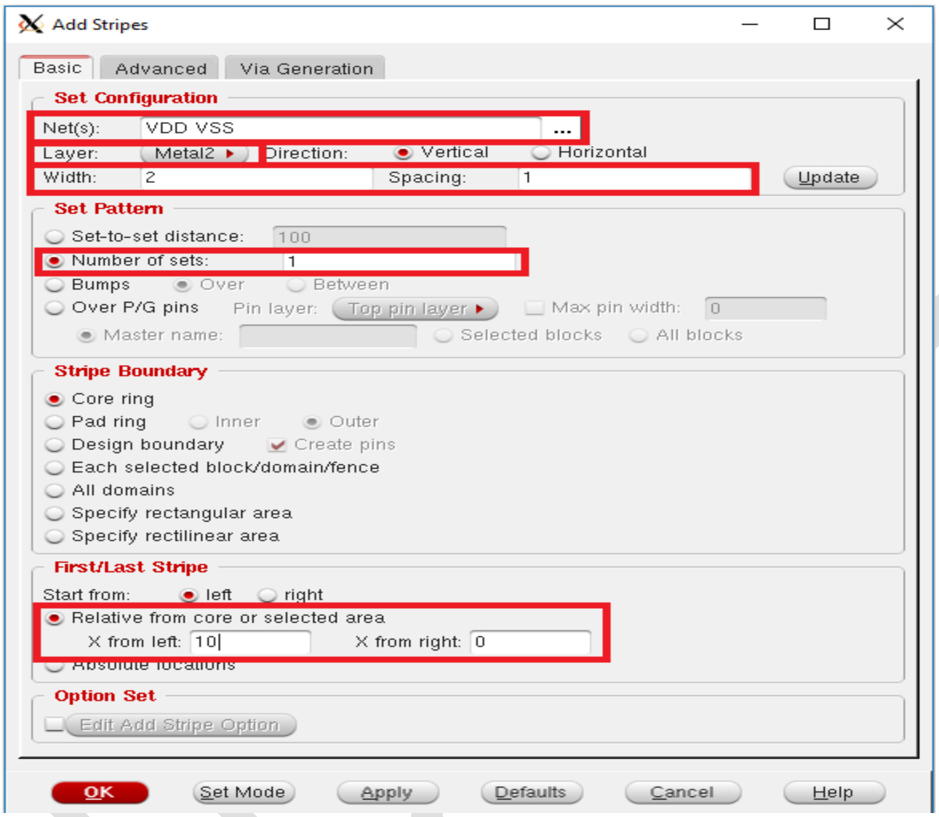
16. In the appeared ***Add Stripes*** window, select the ***Basic*** tab and click on the three dots ***(…)*** button for selecting power nets that will be used as stripes in the design.



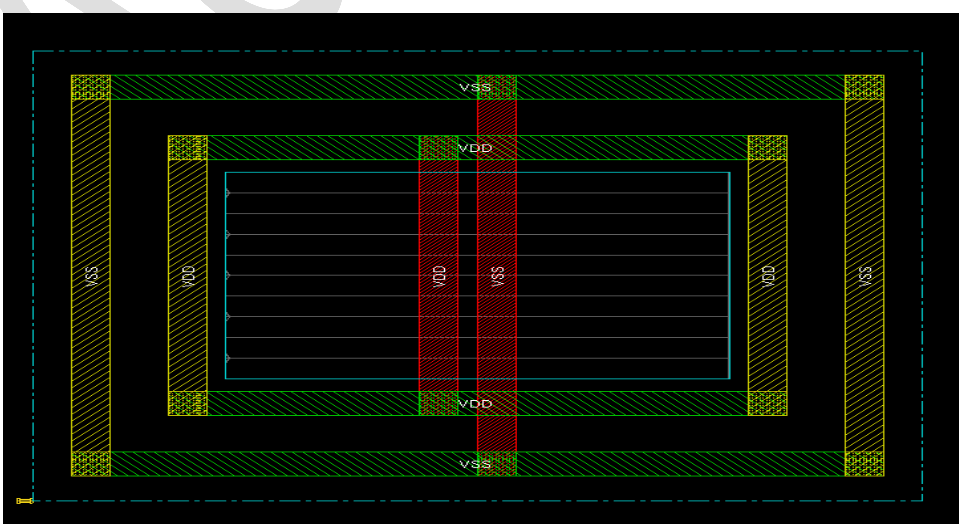
17. From the appeared **Net Selection** window, select both ***VDD*** and ***VSS*** and then click on ***Add*** button. After that click on the ***OK*** button.



18. After that in the **Basic** tab of **Add Stripes** window, select the ***Metal2*** layer and ***Vertical direction*** options. Provide Metal2 stripes with a ***Width** of **2um*** and ***Spacing*** between stripes will be ***1um***. Now under the **Set Pattern** subsection select the ***Number of sets*** option and put the value ***1*** on the blank field. Also, in the blank field of **X from left** option under the **First/Last Stripe** subsection put the value **10**.

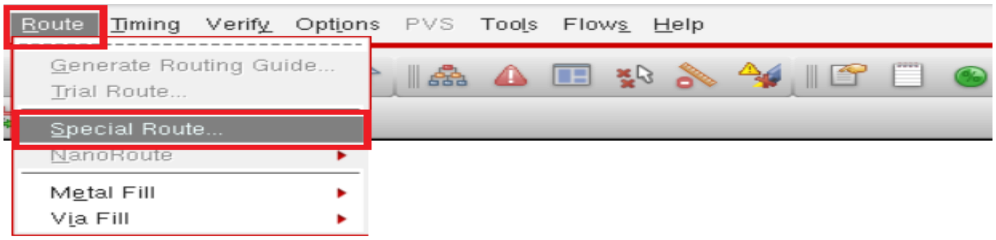


19. Now click on the **OK** button of the **Add Stripes** window. You will see the stripes as well as the ring on the design like the following figure.

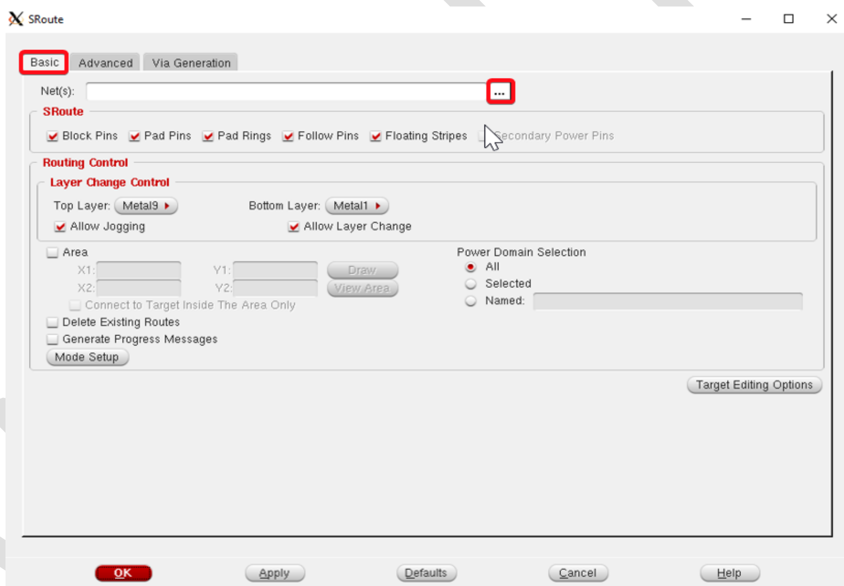


Power Planning: SRoute

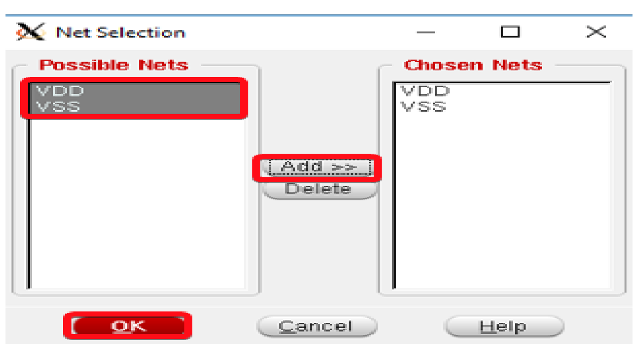
20. Now to deliver the power supply to the core circuit we need to perform **SRoute** (special route). Select the ***Special Route*** option by executing ***Route*** **→** ***Special Route***.



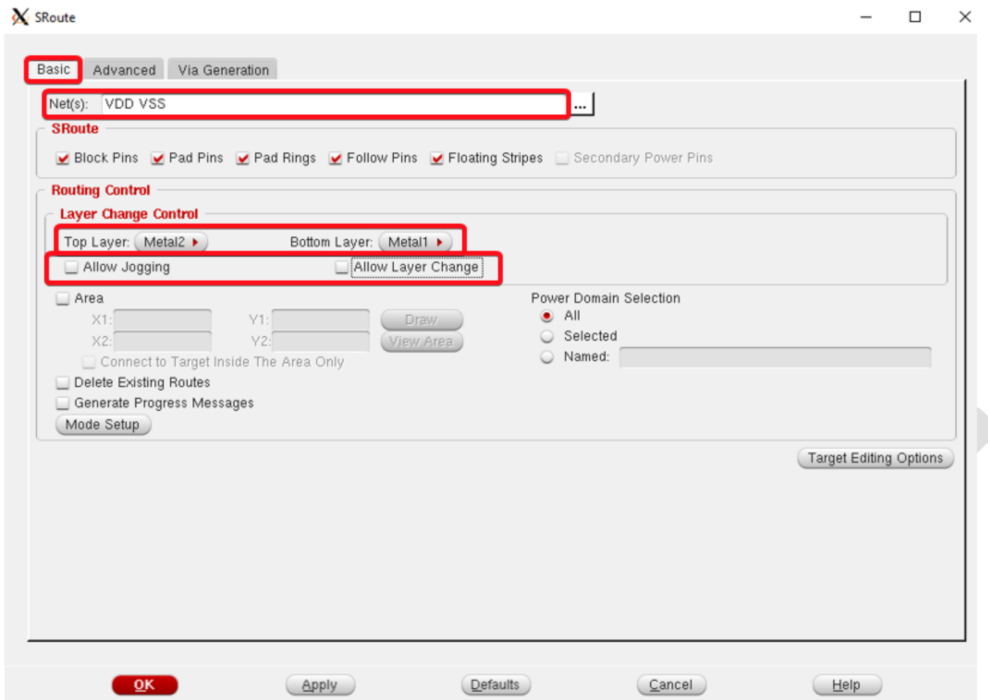
21. In the appeared ***SRoute*** windows, select the ***Basic*** tab and click on the three dots **(...)** button for selecting the power nets name that you created on ***Import Design*** browser.



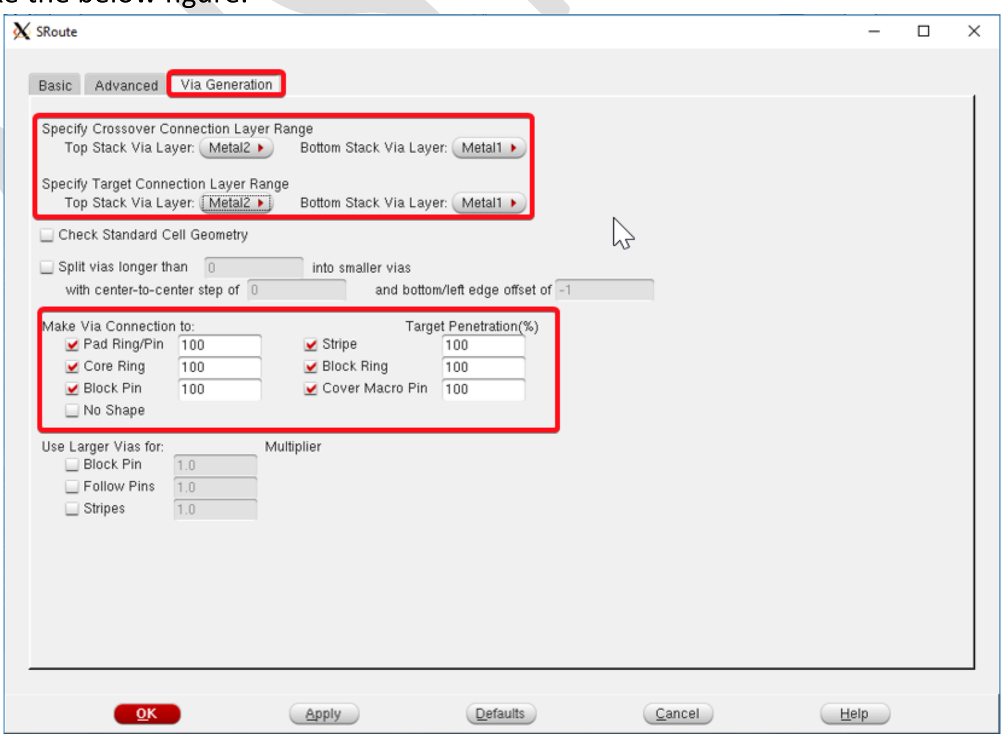
22. From the appeared **Net Selection** window, select both ***VDD*** and ***VSS*** and then click on ***Add*** button. After that press the ***OK*** button.



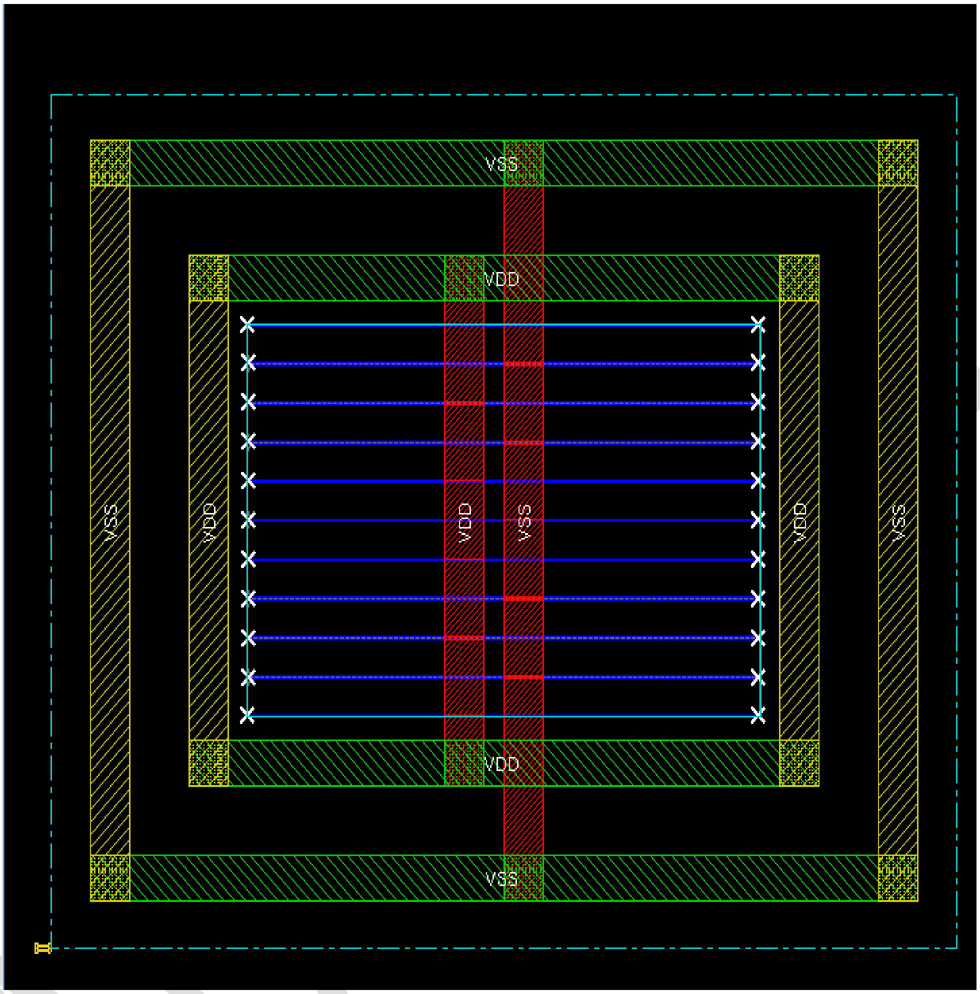
23. Now under the **Basic** tab of the **SRoute** window, choose ***Metal2*** in ***Top Layer*** and ***Metal1*** *in **Bottom Layer***. Make sure that ***Allow Jogging*** and ***Allow Layer Change*** options remain unchecked.



24. Now under the **Via Generation** tab of the **SRoute** window, choose ***Top Stack Via: Metal2*** and ***Bottom Stack Via: Metal1*** options. Make sure that your ***Via Generation*** tab will look like the below figure.



25. After Completing all the tasks on **SRoute** window, click on the ***OK*** button. The following figure will appear.

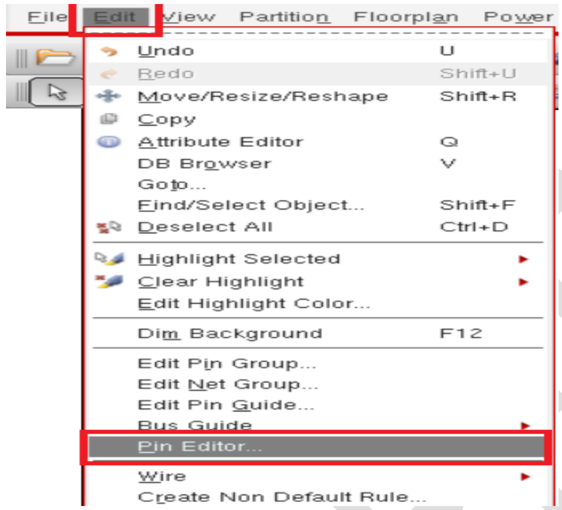


26. This ends our power planning stage. Now save the post your post ***SRoute*** design using the following command.

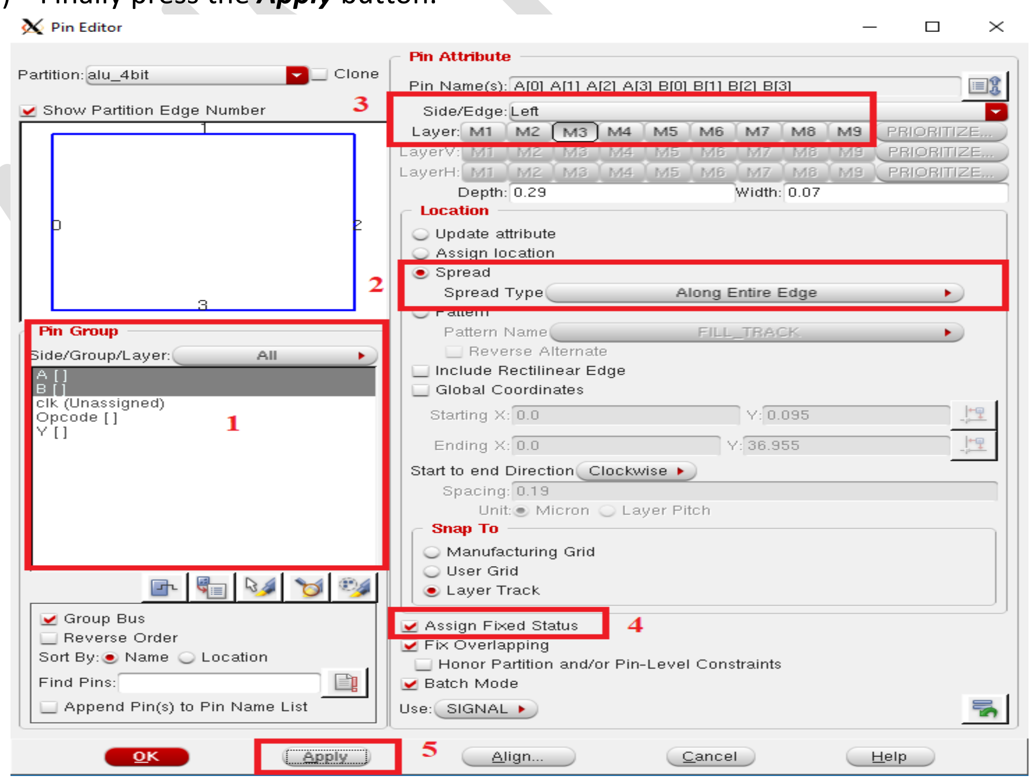
encounter 2> *saveDesign power_plan.enc*

Pin Placement

27. After power mesh creation, all the pins of the design need to be placed around the die boundary. For that, select ***Pin Editor*** by executing ***Edit → Pin Editor.***



28. The steps for assigning pins to the left side are given below,
    a) At first select ***A[] and B[]*** pins the from ***Pin Group***.
    b) Next select ***Spread*** and ***Spread type: Along Entire Edge*** from the ***Location*** *section*.
    c) Then select ***Side/Edge: Left*** from ***Pin Attribute*** section.
    d) Also select ***Layer: M3*** from ***Pin Attribute***. [select M4 for top and bottom pins]
    e) After that, check the ***Assign Fixed Status*** option.
    f) Finally press the ***Apply*** button.

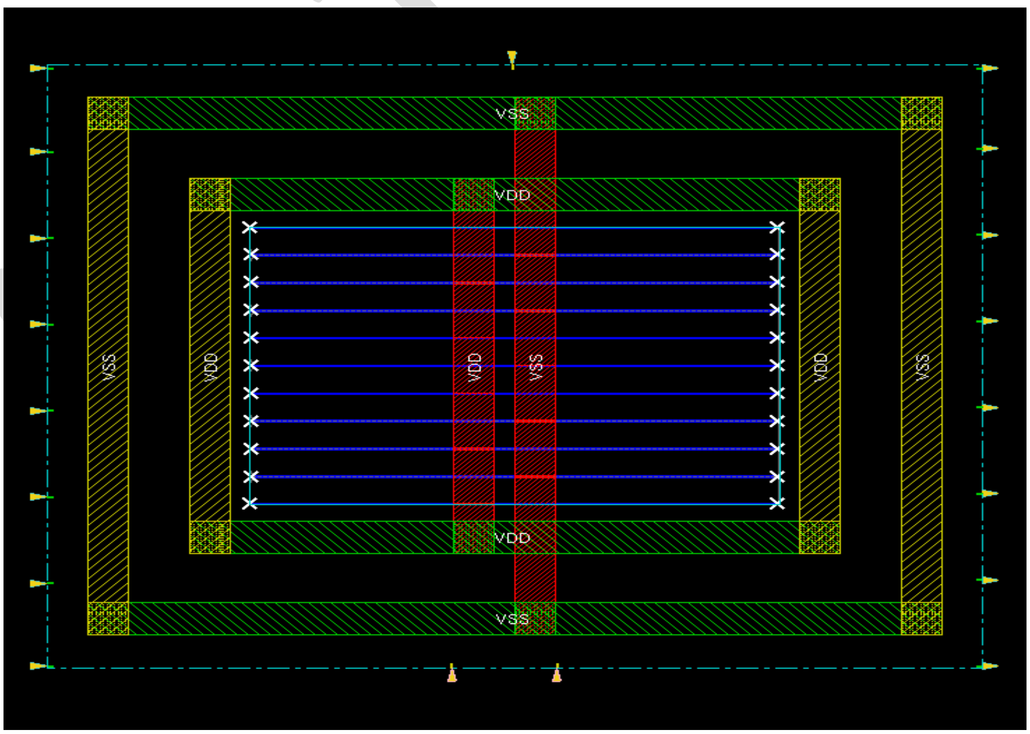


29. Now following step 28 add the rest of the pins according to Table-a.

Table-a

Pin Name	Side/Edge	Spread Type	Layer
A[] B[]	Left	Along entire edge	M3
clk	Top	From Center *[for single pin]*	M4
Y[]	Right	Along entire edge	M3
Opcode	Bottom	Between Points Starting X→20 Ending X →25	M4

30. After adding all the pins click on the **OK** button of the **Pin Editor** window. Now the design will look like the below figure on your encounter window.



31. Now save the design using the following command. This is the end of the pre-placement stage.

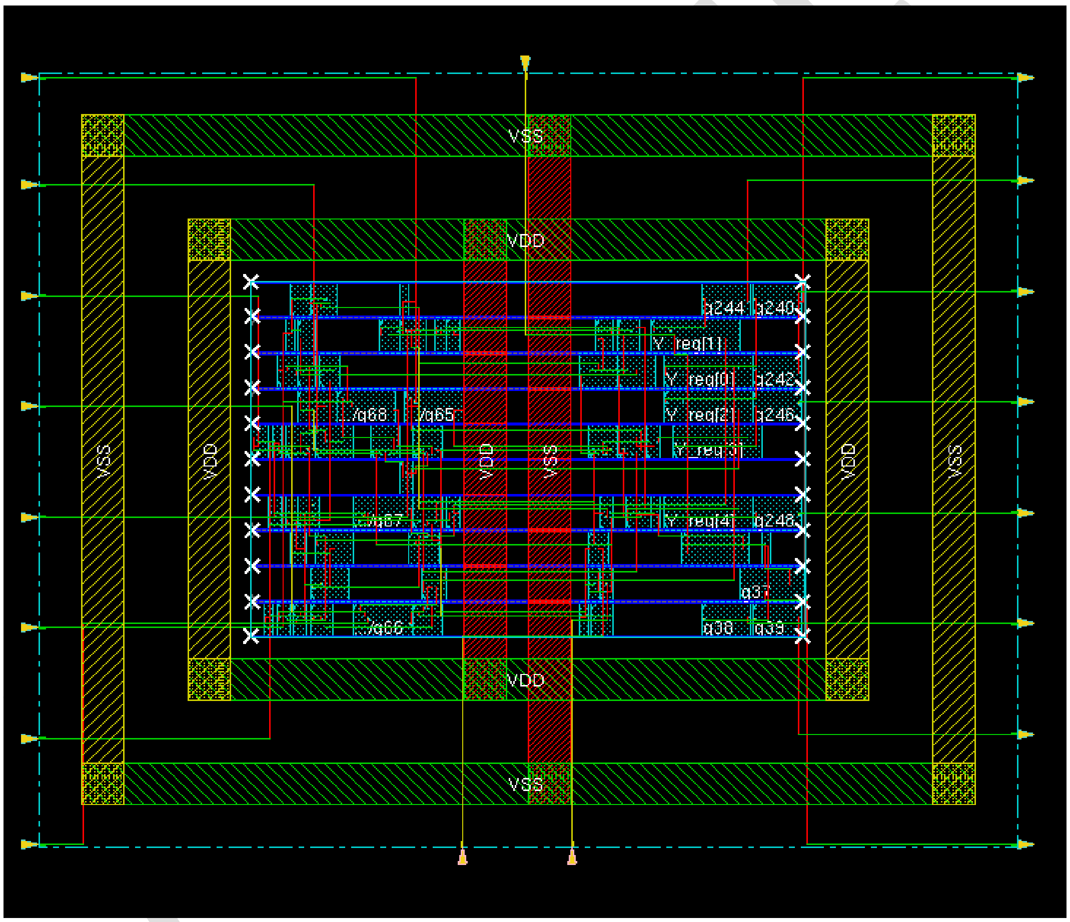
encounter 3> *saveDesign pin_placement.enc*

Placement

32. To place all the existing instances (standard cells and macros) in the design, use the following command

encounter 4> *placeDesign -noPrePlaceOpt*

After placement, click on the black screen of the encounter window and press the **F** key on your keyboard. It will clearly show the design with placed instances and the global routing between.



33. Now save the design to a different database name where all the instances are placed and connected with each other by global routing by the following command.

encounter 5> *saveDesign placement.enc*

Post Lab Task

1. Which metal should we use for power and ground rings, stripes, and sroute. why?
2. Check the difference between global routing and detail routing.
3. Check the manual of ***saveDesign, placeDesign*** using the man command.
4. What is No-Load violation?
5. Why can't we do hold optimization before building a clock tree?

Lab-7B: Static Timing Analysis Using Encounter Digital Implementation System

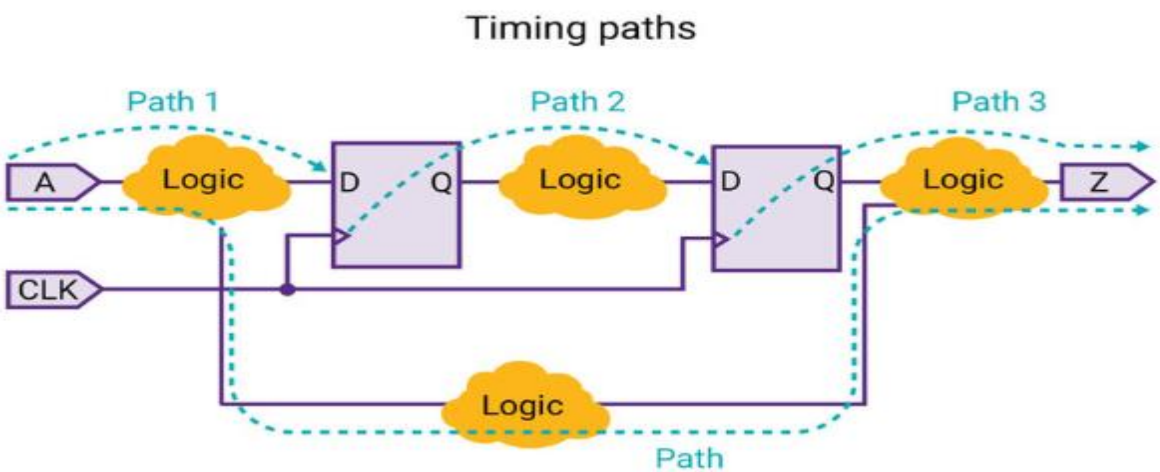
Objective

The main objectives of this lab are:

- Familiarization with Static Timing Analysis.
- Familiarization with clock tree synthesis, and detail routing.
- Familiarization with STA Optimization Techniques. (Pre-CTS and Post-Route)

Introduction

Static Timing Analysis (STA) is a method of validating the timing performance of an ASIC design by checking all possible paths for timing violations. STA breaks the design down into timing paths, calculates the signal propagation delay along each path, and checks for violations of timing constraints inside the design and at the input/output interface.



In the example, each logic cloud represents a combinational logic network. Each path starts at a data launch point, passes through some combinational logic, and ends at a data capture point.

Path	Startpoint	Endpoint
Path 1	Input port	Data input of a sequential element
Path 2	Clock pin of a sequential element	Data input of a sequential element
Path 3	Clock pin of a sequential element	Output port
Path 4	Input port	Output port

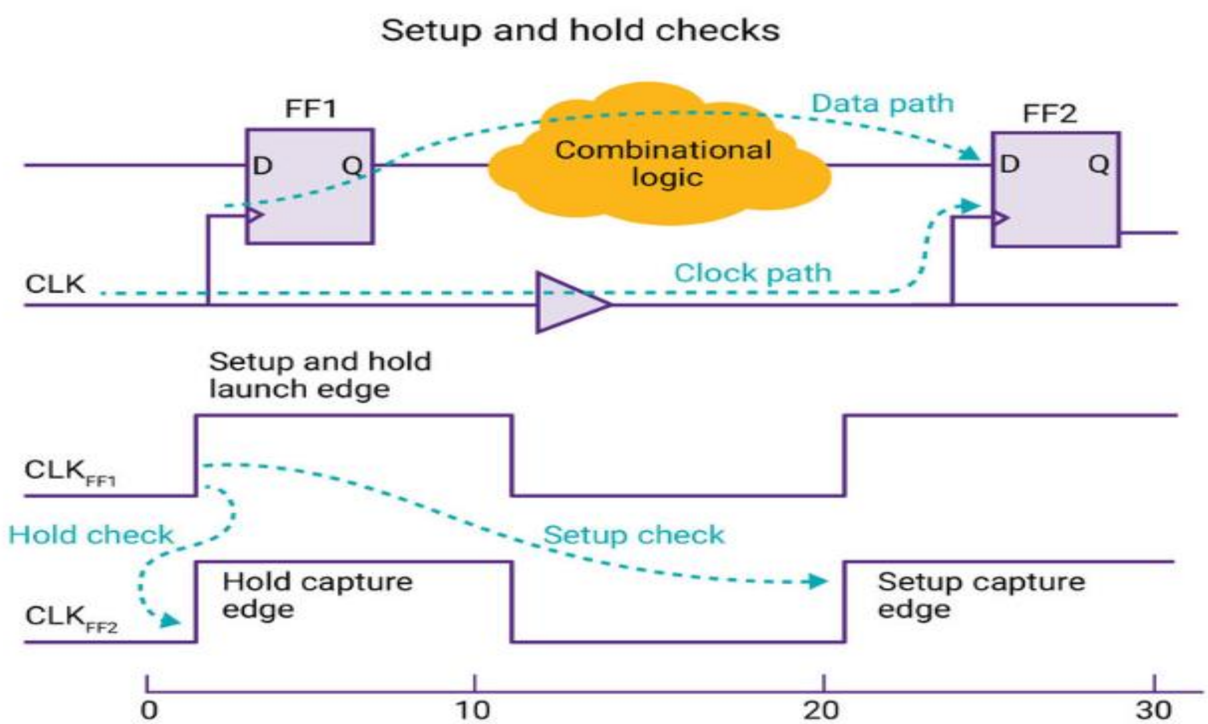
When performing timing analysis, STA first breaks down the design into timing paths. Each timing path consists of the following elements:

- **Start point**: The start of a timing path where data is launched by a clock edge or where the data must be available at a specific time. Every start point must be either an input port or a register clock pin.
- **Combinational logic network**: Elements that have no memory or internal state. Combinational logic can contain AND, OR, XOR, and inverter elements, but cannot contain flip-flops, latches, registers, or RAM.
- **Endpoint**: The end of a timing path where data is captured by a clock edge or where the data must be available at a specific time. Every endpoint must be either a register data input pin or an output port.

While performing STA, there are several types of violations that needs to be analyzed and must solved while debugging the violation paths. We are checking timing violations like setup and hold violations, and DRV (Design Rule Violations) like maximum transition, capacitance and fanout violations.

Setup: A setup constraint specifies how much time is necessary for data to be available at the input of a sequential device before the clock edge that captures the data in the device.

Hold: A hold constraint specifies how much time is necessary for data to be stable at the output of a sequential device after the clock edge that captures the data in the device.



For this example, assume that the flip-flops are defined in the logic library to have a minimum setup time of 1.0 time units and a minimum hold time of 0.0 time units. The clock period is defined in the tool to be 10 time units.

By default, the tool assumes that signals are propagated through each data path in one clock cycle. **Therefore, when the tool performs a setup check, it verifies that the data launched from FF1 reaches FF2 within one clock cycle, and arrives at least 1.0 time unit before the data gets captured by the next clock edge at FF2.** If the data path delay is too long, it is reported as a timing violation. For this setup check, the tool considers the longest possible delay along the data path and the shortest possible delay along the clock path between FF1 and FF2.

When the tool performs a hold check, it verifies that the data launched from FF1 reaches FF2 no sooner than the capture clock edge for the previous clock cycle. This check ensures that the data already existing at the input of FF2 remains stable long enough after the clock edge that captures data for the previous cycle. For this hold check, the tool considers the shortest possible delay along the data path and the longest possible delay along the clock path between FF1 and FF2. A hold violation can occur if the clock path has a long delay.

Max Transition: Transition delay or slew is defined as the time taken by signal to rise from logic low state to logic high state or fall from logic high state to logic low state. This check ensures that logic state is changing within a specific time, not taking longer time than that specific time.

Max Capacitance: The capacitance on a node is a combination of the fan-out of the output pin and capacitance of the net. This check ensures that the device does not drive more capacitance than the device is characterized for.

Max Fanout: Fanout is the number of CMOS logic inputs that can be driven by one CMOS logic output. It refers that how many inputs can be safely driven by a single output pin.

Lab Task

So far, we haven't done any sort of timing analysis or optimization. In this part, we will try to understand the pre-CTS timing reports and will try to optimize the violations that occurred during the pre-CTS stage. Then we will create CTS and will route the design. After that, we will analyze the post rout or post-CTS timing reports and will try to optimize the violations that occurred during the post-CTS stage.

1. Now from the encounter terminal, restore the ***placement.enc*** database using the following command.

encounter 1> *source placement.enc*

Pre-CTS Timing Optimization

2. To check the summary of existing setup and DRV violations in the placement stage (also known as the pre-CTS stage), use the following command

   encounter 2> *timeDesign -preCTS*

   A summary of timing violations will appear on the encounter terminal like the below figure.

```
-------------------------------------------------------------
          timeDesign Summary
-------------------------------------------------------------

+--------------------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  8.113  |   N/A   |  8.113  |
|           TNS (ns):|  0.000  |   N/A   |  0.000  |
|    Violating Paths:|    0    |   N/A   |    0    |
|          All Paths:|    6    |   N/A   |    6    |
+--------------------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |       0 (0)      |   0.000    |       0 (0)      |
|   max_tran     |       0 (0)      |   0.000    |       0 (0)      |
|   max_fanout   |       0 (0)      |     0      |       0 (0)      |
|   max_length   |       0 (0)      |     0      |       0 (0)      |
+----------------+------------------+------------+------------------+
```

3. After checking summary reports from the encounter terminal, we need to check the detailed reports of existing violations. A directory named **timingReports** will be created and detailed violation reports will be generated inside that directory every time when we use **timeDesign** command on the encounter. Check your ***pnr_lab*** directory whether **timingReports** directory and violations reports are created or not like the below table.

```
timingReports
├── alu_4bit_preCTS_all.tarpt
├── alu_4bit _preCTS.cap
├── alu_4bit_preCTS_default.tarpt
├── alu_4bit _preCTS.fanout
├── alu_4bit _preCTS.length
```

```
├── alu_4bit_preCTS_reg2reg.tarpt
├── alu_4bit _preCTS.summary
└── alu_4bit_preCTS.tran
0 directories, 8 files
```

4. In the report of step 2 if there is any negative value in **max_tran** and **max_cap**, it indicates that there is a violation in the design which must be optimized. From the report of step 2, we can say, there are no violations in the design. If we get any violations on the design, we have to use the following command for optimization.

   encounter 3> *optDesign -preCTS*

   Another optimized summary report will be generated on the encounter terminal where we can check how many violations still remain after optimization.

```
------------------------------------------------------------
     optDesign Final Summary
------------------------------------------------------------

+--------------------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  8.113  |   N/A   |  8.113  |
|           TNS (ns):|  0.000  |   N/A   |  0.000  |
|    Violating Paths:|    0    |   N/A   |    0    |
|          All Paths:|    6    |   N/A   |    6    |
+--------------------+---------+---------+---------+

+----------------+--------------------------------+------------------+
|                |              Real              |       Total      |
|    DRVs        +------------------+-------------+------------------|
|                |  Nr nets(terms)  | Worst Vio   |  Nr nets(terms)  |
+----------------+------------------+-------------+------------------+
|   max_cap      |      0 (0)       |    0.000    |       0 (0)      |
|   max_tran     |      0 (0)       |    0.000    |       0 (0)      |
|   max_fanout   |      0 (0)       |      0      |       0 (0)      |
|   max_length   |      0 (0)       |      0      |       0 (0)      |
+----------------+------------------+-------------+------------------+

Density: 52.720%
Routing Overflow: 0.00% H and 0.00% V
```

5. As we run the pre-CTS optimized command on encounter, many changes happened to the design like changes in the placement of cells and global routing. For that reason, we need to save the design again using the following command

   encounter 4> *saveDesign placement_optimized.enc*

Clock Tree Synthesis

A clock tree is needed to be built in the design for balancing clock skew and latency after optimizing the design in the placement stage (pre-CTS stage). It is built using a clock buffer or inverter cells.

6. At first, we have to mention the clock name and its port name using the following command

Encounter 5> *create_ccopt_clock_tree -name clk -source clk*

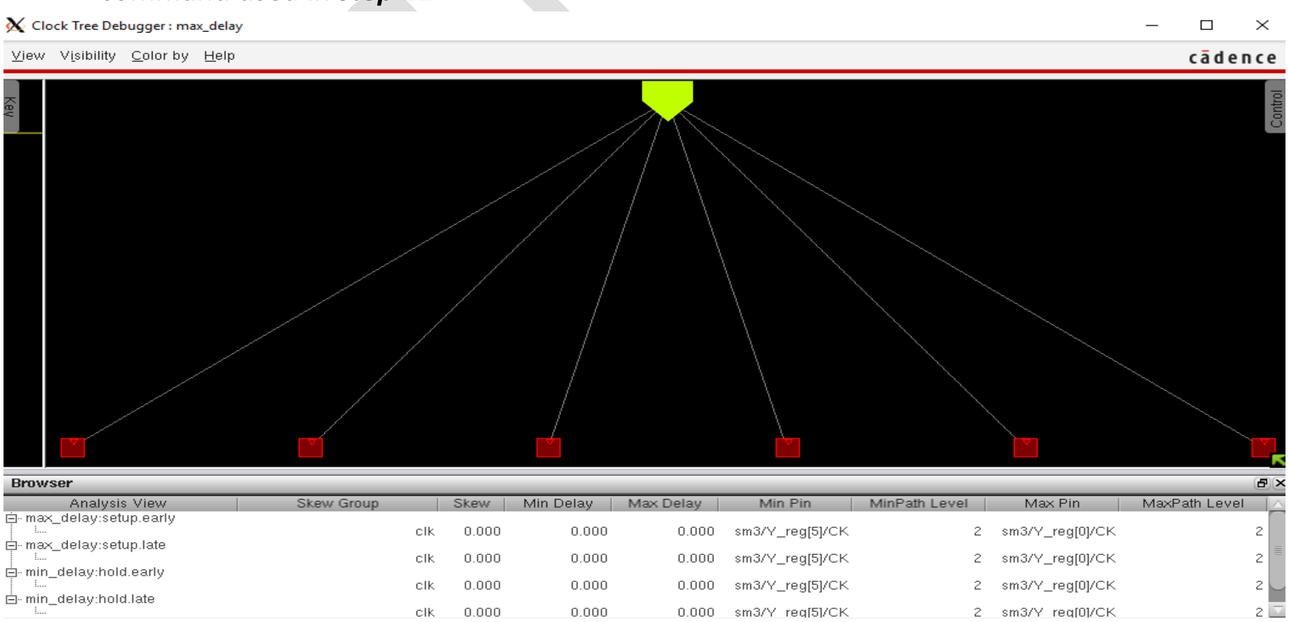
7. Now enter the following command which will give instructions to the tool to build a clock tree.

encounter 6> *ccopt_design -cts*

8. To check the clock tree from the encounter, use the following command

encounter 7> *ctd_win*

A ***Clock Tree Debugger*** window will appear which shows the clock created by the command used in ***step 11***



9. After successfully building the clock tree, save the design to a different database name using the following command.

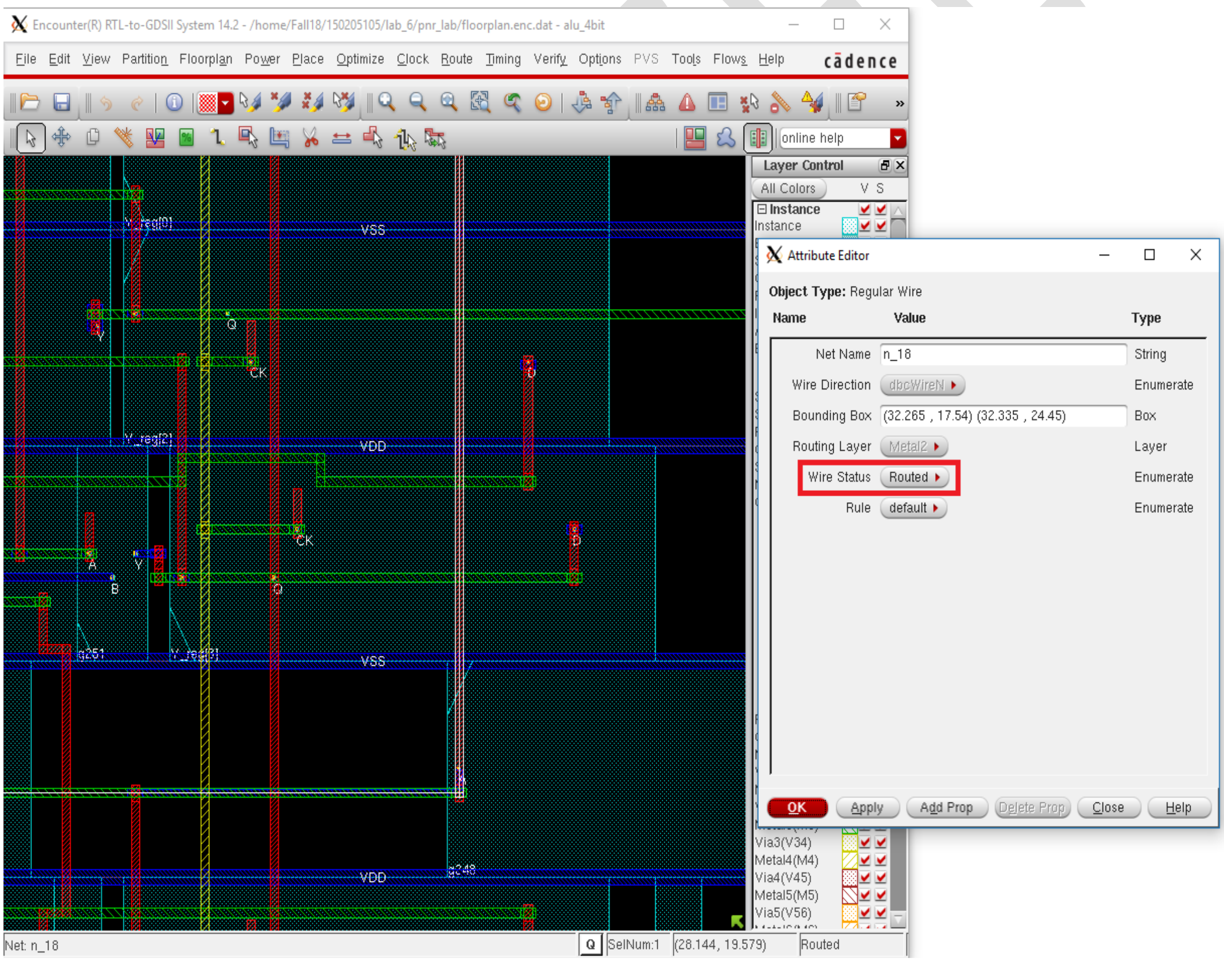
encounter 8> *saveDesign clock_tree_synthesis_optimized.enc*

Detail Routing

10. As a pre-CTS optimization is done in the placement stage and after that, we built the clock tree again in the CTS stage, we need to perform detail routing again. To perform again detail routing, use the following command again to the encounter terminal.

encounter 9> *routeDesign*

11. To check whether the detailed routing has been done or not, you can check the wiring status of the signal routing by selecting a wire and then pressing **Q**. If the Wire status is either **Routed** or **fixed,** detail routing is done successfully. If all the task has been performed successfully, your encounter window will be like the following window.



12. After routing save the design using the following command.

encounter 10> *saveDesign routeDesign.enc*

Post-Route Timing Optimization

13. To check the summary report of existing setup and DRV violations on the routing stage (post-route stage), use the following commands.

> **encounter 11>** *setAnalysisMode -analysisType onChipVariation*
>
> **encounter 12>** *timeDesign -postRoute*

```
-----------------------------------------------------------------
          timeDesign Summary
-----------------------------------------------------------------

+--------------------+----------+----------+----------+
|     Setup mode     |   all    | reg2reg  | default  |
+--------------------+----------+----------+----------+
|           WNS (ns):|  7.991   |   N/A    |  7.991   |
|           TNS (ns):|  0.000   |   N/A    |  0.000   |
|    Violating Paths:|    0     |   N/A    |    0     |
|          All Paths:|    6     |   N/A    |    6     |
+--------------------+----------+----------+----------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |     0 (0)        |   0.000    |     0 (0)        |
|   max_tran     |     0 (0)        |   0.000    |     0 (0)        |
|   max_fanout   |     0 (0)        |     0      |     0 (0)        |
|   max_length   |     0 (0)        |     0      |     0 (0)        |
+----------------+------------------+------------+------------------+

Density: 52.720%
Total number of glitch violations: 0
```

14. To check the summary report of hold violation from the post-route stage, use the following command.

> **encounter 13>** *timeDesign -postRoute -hold*

```
-----------------------------------------------------------------
          timeDesign Summary
-----------------------------------------------------------------

+--------------------+----------+----------+----------+
|     Hold mode      |   all    | reg2reg  | default  |
+--------------------+----------+----------+----------+
|           WNS (ns):|  0.000   |   N/A    |  0.000   |
|           TNS (ns):|  0.000   |   N/A    |  0.000   |
|    Violating Paths:|    0     |   N/A    |    0     |
|          All Paths:|    0     |   N/A    |    0     |
+--------------------+----------+----------+----------+

Density: 52.720%
-----------------------------------------------------------------
```

15. After using the above commands, a summary report will be shown on the encounter terminal and detailed reports of violations will be generated inside the **timingReports**

directory. Check the directory whether detail reports are generated or not like the below figure.

```
timingReports
├── alu_4bit_postRoute_all.tarpt
├── alu_4bit _postRoute.cap
├── alu_4bit_postRoute_default.tarpt
├── alu_4bit _postRoute.fanout
├── alu_4bit _postRoute.length
├── alu_4bit_postRoute_reg2reg.tarpt
├── alu_4bit _postRoute.SI_Glitches.rpt
├── alu_4bit_postRoute.summary
└── alu_4bit _postRoute.tran

0 directories, 9 files
```

16. To clean the existing setup and DRV violations at the post route stage, use the following command.

encounter 14> *optDesign -postRoute*

After automatic optimization, updated reports will be generated inside the **timingReports** directory.

```
------------------------------------------------------------
     optDesign Final SI Timing Summary
------------------------------------------------------------

+--------------------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  7.991  |   N/A   |  7.991  |
|           TNS (ns):|  0.000  |   N/A   |  0.000  |
|    Violating Paths:|    0    |   N/A   |    0    |
|          All Paths:|    6    |   N/A   |    6    |
+--------------------+---------+---------+---------+

+----------------+------------------------------+------------------+
|                |             Real             |       Total      |
|    DRVs        +------------------+-----------+------------------+
|                |  Nr nets(terms)  | Worst Vio |  Nr nets(terms)  |
+----------------+------------------+-----------+------------------+
|   max_cap      |     0 (0)        |   0.000   |     0 (0)        |
|   max_tran     |     0 (0)        |   0.000   |     0 (0)        |
|   max_fanout   |     0 (0)        |     0     |     0 (0)        |
|   max_length   |     0 (0)        |     0     |     0 (0)        |
+----------------+------------------+-----------+------------------+

Density: 52.720%
Total number of glitch violations: 0
```

17. To clean existing hold violations, use the following command.

encounter 15> *optDesign -postRoute -hold*

```
------------------------------------------------------------
     optDesign Final SI Timing Summary
------------------------------------------------------------

+--------------------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  7.991  |   N/A   |  7.991  |
|           TNS (ns):|  0.000  |   N/A   |  0.000  |
|    Violating Paths:|    0    |   N/A   |    0    |
|          All Paths:|    6    |   N/A   |    6    |
+--------------------+---------+---------+---------+

+--------------------+---------+---------+---------+
|     Hold mode      |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  0.000  |   N/A   |   N/A   |
|           TNS (ns):|  0.000  |   N/A   |   N/A   |
|    Violating Paths:|    0    |   N/A   |   N/A   |
|          All Paths:|    0    |   N/A   |   N/A   |
+--------------------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |     0 (0)        |   0.000    |     0 (0)        |
|   max_tran     |     0 (0)        |   0.000    |     0 (0)        |
|   max_fanout   |     0 (0)        |     0      |     0 (0)        |
|   max_length   |     0 (0)        |     0      |     0 (0)        |
+----------------+------------------+------------+------------------+

Density: 52.720%
Total number of glitch violations: 0
```

18. After optimization, save the design using the following command.

encounter 16> *saveDesign routeDesign_optimized.enc*

Post Lab Task

1. What are the goals of **CTS**?
2. Why are buffers used in the clock tree?
3. How many routings are done in PnR?
4. Compare **Setup** and **Hold** time.
5. Find out the advantage of using inverter over buffer while building a clock tree.
6. What is clock skew and latency? How does skew affect both setup and hold violations?
7. Check the manual of ***report_clocks, selectPin, ccopt_design, routeDesign*** using the man command.

Lab-8: Physical Verification and Power Analysis Using Encounter Digital Implementation System

The main objectives of this lab are:

- Familiarization with Physical Verification (DRC, Geometry and Connectivity Check)
- Familiarization with Power Analysis (IR Drops and Electromigration)

Introduction

This section will perform physical verifications to check whether the design layout is equivalent to its schematic and checks the layout against process manufacturing guidelines provided by the semiconductor fabrication labs to ensure it can be manufactured correctly. Some common verification techniques are listed below. This lab will check the DRC, LVS, and ARC under Physical Verification Steps.

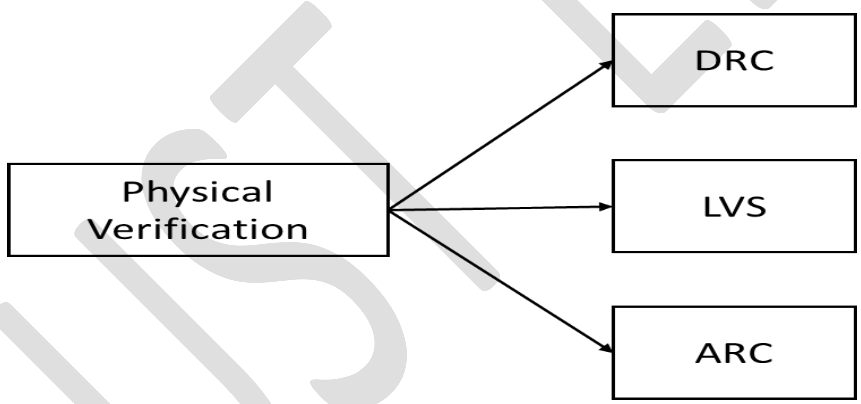


Fig: Physical Verification flow

Design Rule Check (DRC)

Design Rules define shapes/size/spacing and many other complex rules of each metal layer. It starts from the substrate to Newell to the op metal layers. DRC doesn't ensure that the device will work properly, it ensures it will get manufactured properly.

Layout versus schematic (LVS)

It checks for correct connectivity between the devices in the circuit. It is a method of verifying that the layout of the design is functionally equivalent to the schematic of the design.

ARC (Antenna Rule Check)

Checks for a large area of metals that might affect manufacturing. Ensure that the transistors of the chip are not destroyed during fabrication. Using metal jogging or inserting a diode at the gate can fix this.

Power Analysis

The power supply (VDD and VSS) in a chip is uniformly distributed through the metal rails and stripes which is called Power Delivery Network (PDN) or power grid. Each metal layer used in PDN has finite resistivity. When current flow through the power delivery network, a part of the applied voltage will be dropped in PDN as per Ohm's law. The amount of voltage drop will be V = I.R, which is called the IR drop. We will check in this lab whether special nets are shorted or not, and whether power via**s** are created properly, which will connect all the special nets.

Electromigration is the movement of atoms based on the flow of current through a material. If the current density is high enough, the heat dissipated within the material will repeatedly break atoms from the structure and move them. This will create both 'vacancies' and 'deposits'. The vacancies can grow and eventually break circuit connections resulting in open-circuits, while the deposits can grow and eventually close circuit connections resulting in short-circuit. In this lab, we will check the signal net's AC current limit violations.

Lab Task

1. Log in to the server in the GUI mode and source the Cadence license file.
   [**Xlaunch (enable SSH)→putty (load server IP) → login → csh→ source ~/cshrc_q→ nautilus]**

2. Open a terminal and make sure you are at the home directory of your account using the command pwd.

   [150205105@aust ~]$ pwd

3. Go to the directory ***lab_6/pnr_lab*** executing the command ***cd lab_6/pnr_lab***

   [150205105@aust ~]$ *cd lab_6/pnr_lab*

4. Make sure that the ***placement.enc*** database is present in the ***pnr_lab*** directory. Then launch the Encounter tool from the same directory using the command encounter

   [150205105@aust pnr_lab]$ *encounter*

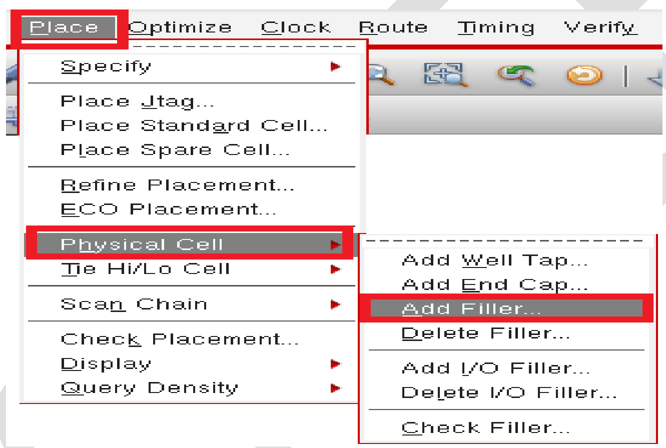
5. Now from the encounter terminal, restore the ***routeDesign_optimized.enc*** database using the following command.

```
encounter 1> source   routeDesign_optimized.enc
```

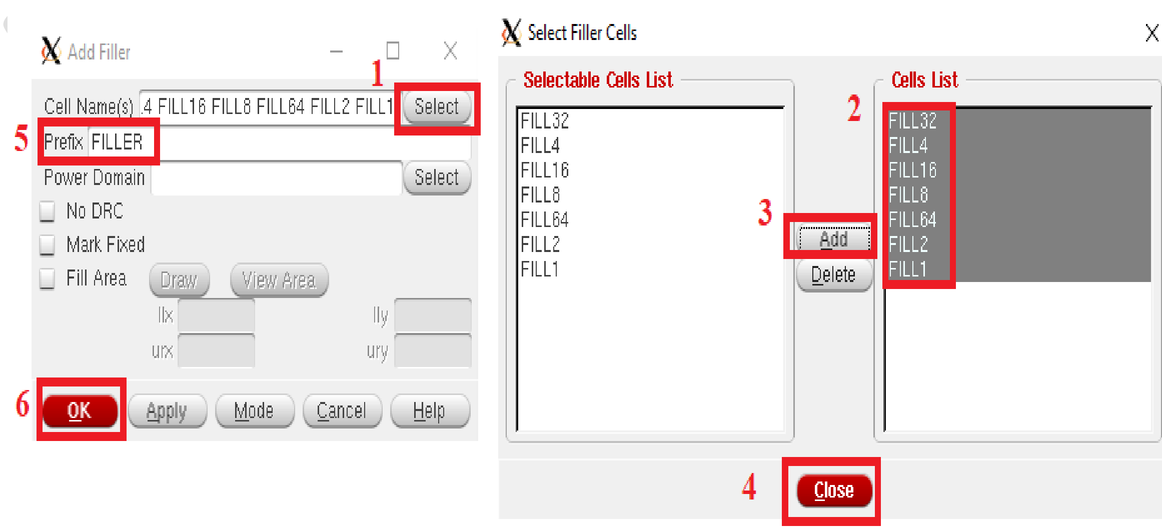
Filler Cell and Metal Filler

Filler cells are used to fill any spaces between regular library cells. They are needed when the density of the required metal or layer has not met the foundry or fabrication requirement.

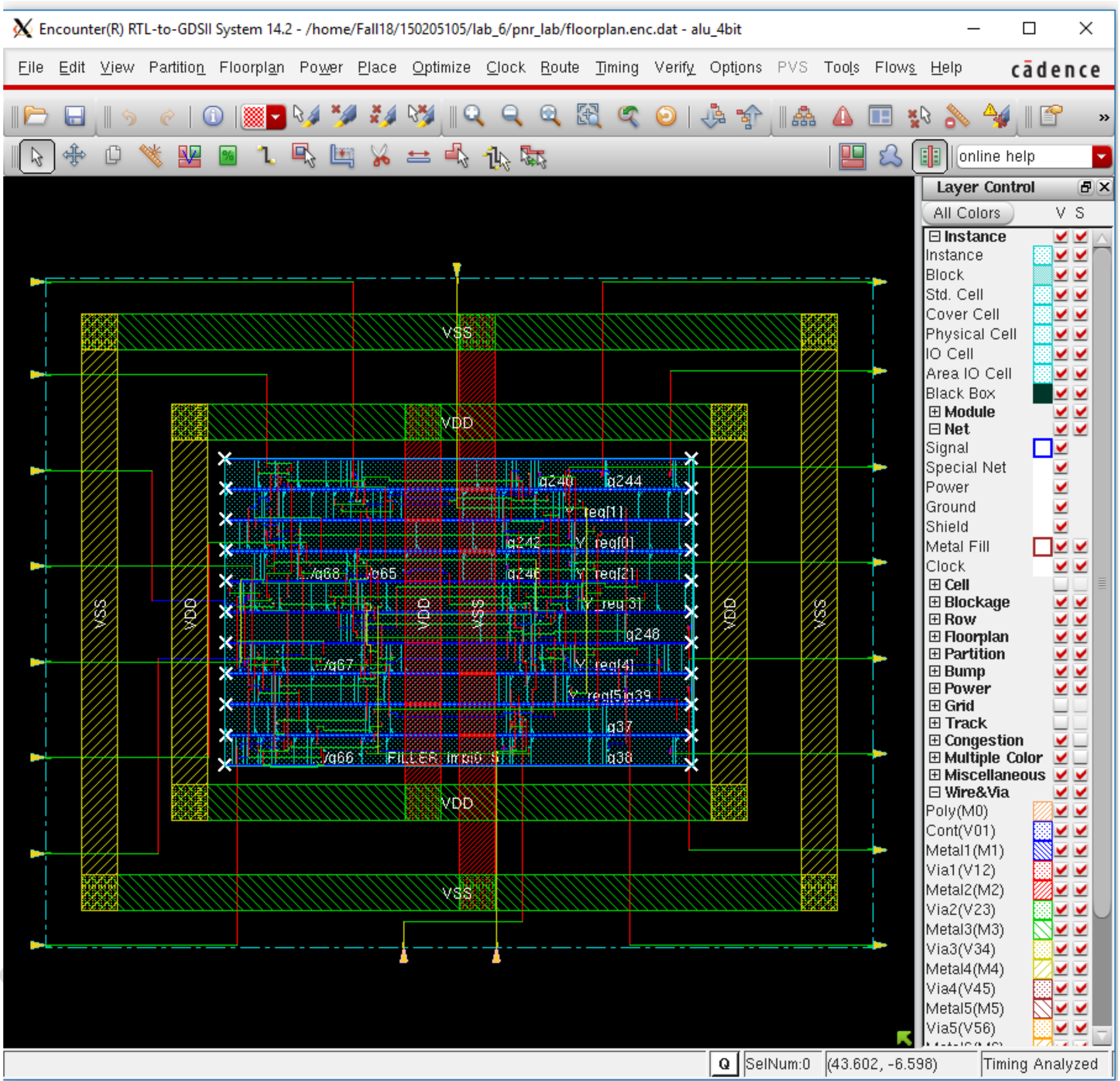
6. To add filler cells, execute **Place → Physical Cell → Add Filler.**



7. Then the **Add Filler** window will appear. Select all the filler cells from the **Cell Lists** of the **Select Filler Cells** window and give the **Prefix** *FILLER* as shown in the following figure. Then click **OK.**



After adding filler cells, the design will be like the following figure.



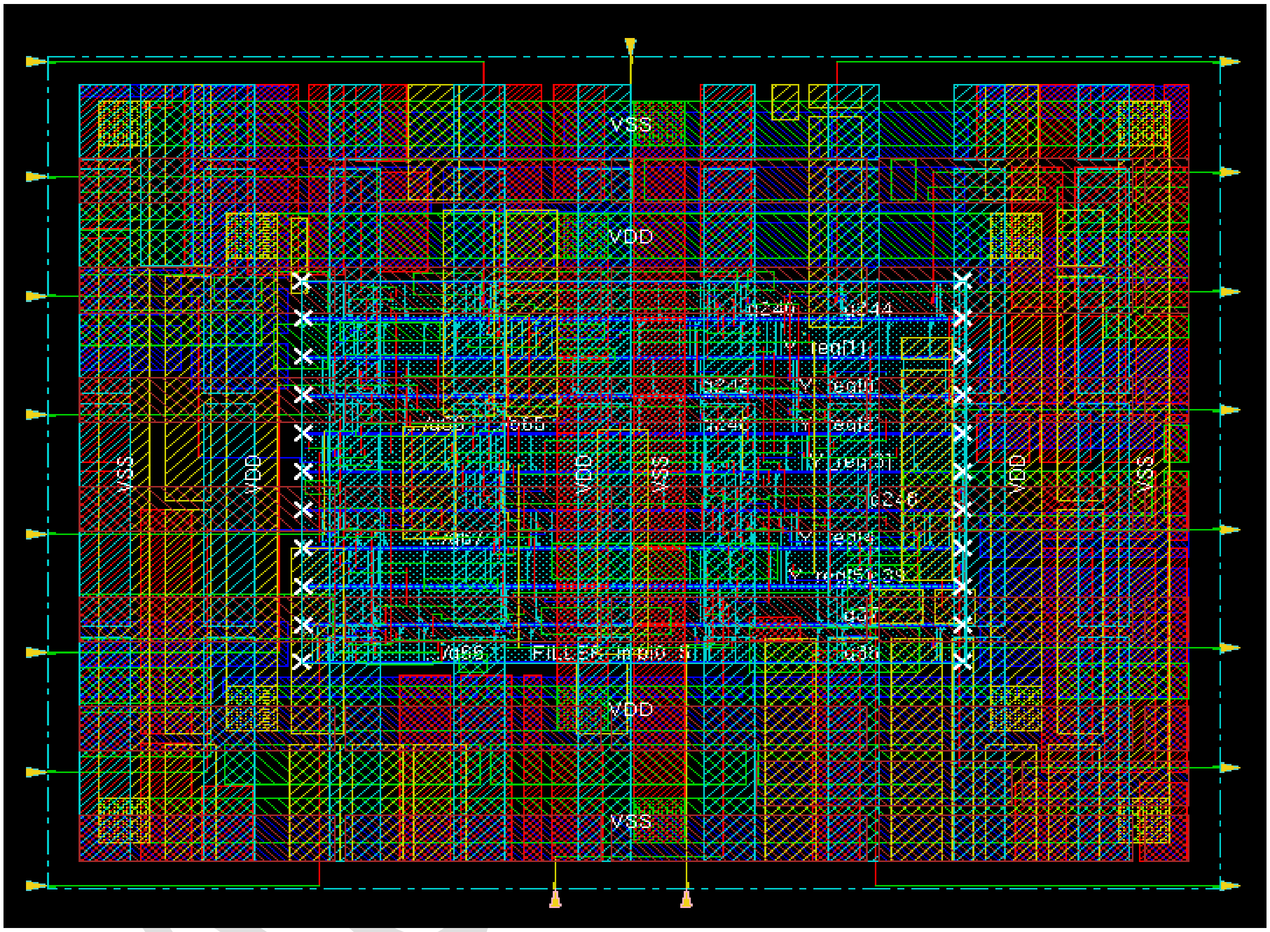
8. After adding filler cells, we have to re-route the modified design using the following command.

encounter 2> *ecoRoute*

9. Now to add metal filler use the following command.

encounter 3> *addMetalFill*

After adding metal filler, the design will be like the following figure.



10. Now, to check the placement density and number of placed cells use the following command.

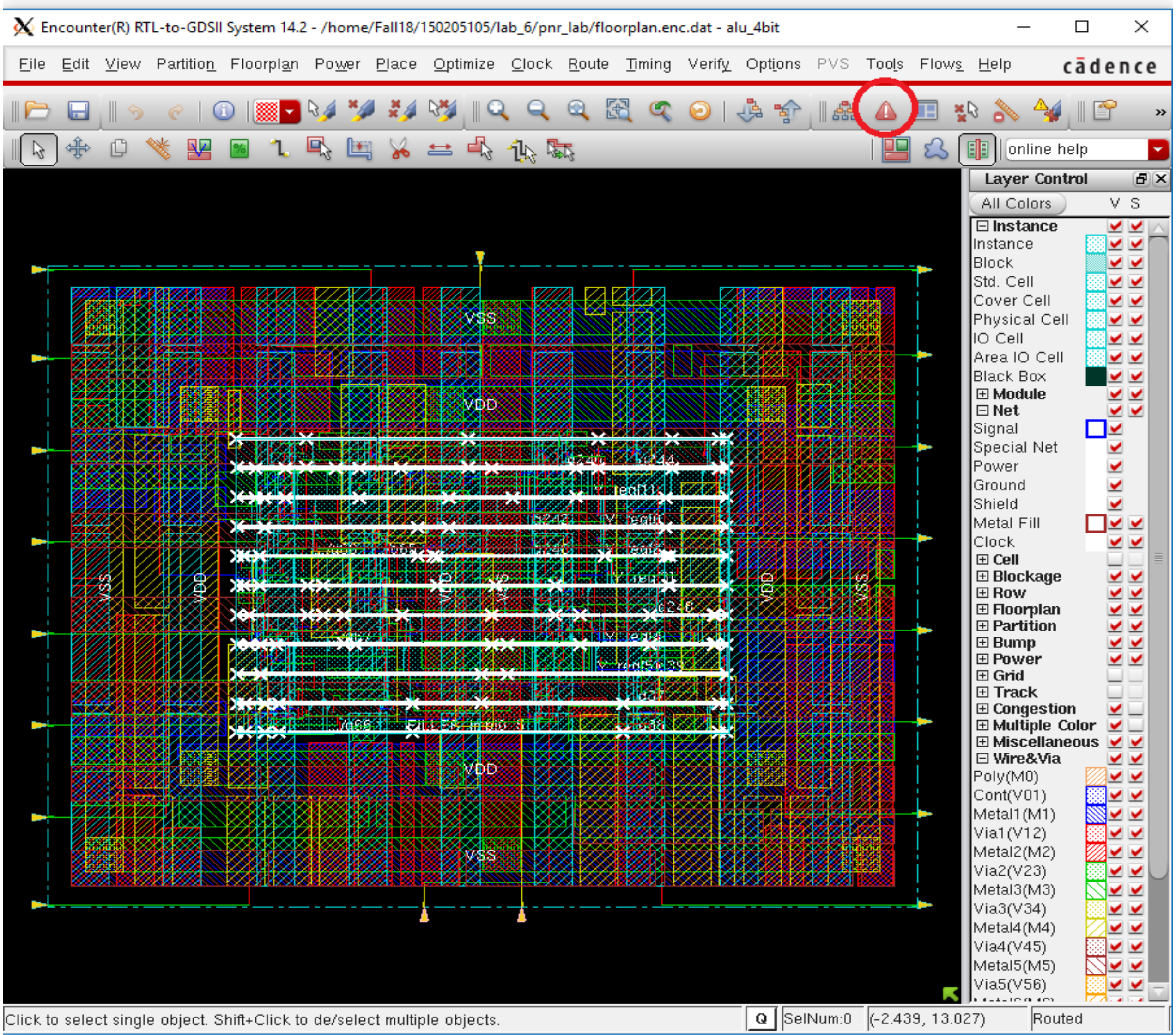
encounter 4> *checkPlace*

Physical Verification

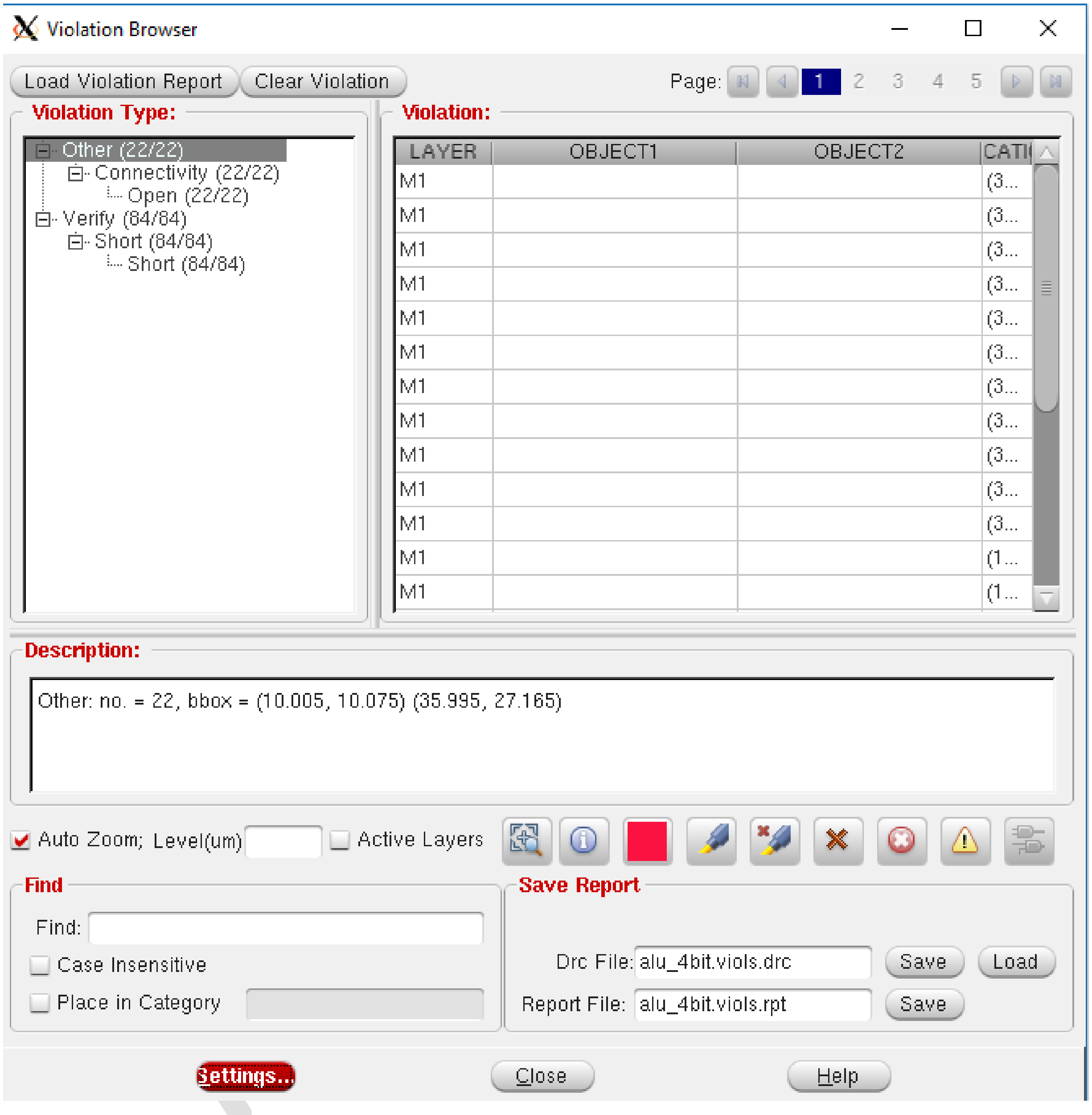
11. After routing, the design must pass all physical verification stages. At first, we will check all DRC (Design Rule Check) rules using encounter. Write the following command on the encounter terminal.

encounter 5> *verify_drc*

If the design has a DRC violation, you can see the DRC markers (white cross) from the encounter window. To check all the DRC violations, click on the ***Violation Browser*** icon marked on below the figure.



12. The following Violation **Browser** window will appear. In that window, all the DRC type and their detail violation can be checked. Click on any of the violation it will take you to that violation area.



13. To solve power net (VDD) and ground net (VSS) related violations, use the following commands on the encounter terminal.

encounter 6> *globalNetConnect VDD -pin VDD -instanceBasename * -verbose*

encounter 7> *globalNetConnect VSS -pin VSS -instanceBasename * -verbose*

14. Now to solve violations that occurred due to the shape of via, zoom into the violation area and change the via type by clicking the "**Shift+N**" key.

15. Now clear all DRC markers from the ***encounter*** and ***Violation Browser*** window and again check the DRC, using the following commands.

encounter 8> *clearDrc*

encounter 9> *verify_drc*

16. To check all violations related to the connectivity of the design, use the following command.

encounter 10> *verify_connectivity*

17. To check geometry violations from the encounter, write the following command on the encounter terminal.

encounter 11> *verifyGeometry*

18. To check ARC (Antenna Rule Check) using encounter, write the following command on the encounter terminal.

encounter 12> *verifyProcessAntenna*

Power Analysis

19. To check whether the Power/Ground net is short or not use the following command on the encounter terminal. The command checks short between
    a. PG and PG nets
    b. PG and signal nets
    c. PG and other special net

encounter 13> *verify_PG_short*

20. To check all the single power via are generated correctly to connect each of the PG net together.

encounter 14> *verify_power_via*

21. The following command will check only the generated stacked power via on the design and reports unconnected or weakly connected special nets.

encounter 15> v*erify_power_via   -stacked_via*

22. To prevent wire from self-heating or AC signal electromigration, signal interconnects should be analyzed for their AC current carrying capacity and measured against the AC current limits specified by the foundry. Use the following command to check AC current violations on signal nets

encounter 16> v*erifyACLimit*

23. Now if you optimized all the violations save the final design using the following command on the encounter terminal.

encounter 17> *saveDesign   finalDesign.enc*

Post Lab Task

1. Discuss the importance of filler cell and metal filler?
2. How the ARC problem can be solved?
3. What is IR drop? Define is Static and Dynamic power dissipation?
4. How LVS comparison is dine in digital design?
5. Check the manual of ***verify_drc, verify_connectivity***, ***verifyGeometry, verifyProcessAntenna, verify_PG_short, verify_power_via, verifyACLimit*** using the man command.

References and Acknowledgment

The following resources have been consulted while preparing the manual.

- Stephen Brown and Zvonko Vranesic , “**Fundamentals of Digital Logic with Verilog Design**”.
- Erik Brunvand , “**Digital VLSI Chip Design with Cadence and Synopsys CAD tools**”
- M. L. Bushnell and V. D. Agrawal, “**Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits**”, Kluwer Academic Publishers, , ISBN: 0-7923-7991-8.
- A. B. Kahng, J. Lienig, I. L. Markov, J. Hu, “**VLSI Physical Design: From Graph Partitioning to Timing Closure. Springer Publishers**”, ISBN 978-90-481-9590-9.
- https://linuxhint.com/
- https://www.synopsys.com/glossary/what-is-static-timing-analysis.html?fbclid=IwAR1Rs0F3NMyxNs-7y4FacKjfbu5M08XzhOTps_eZaTvqueUS4DMNgRzenhw

Prepared by:

I. Adnan Amin Siddiquee
Lecturer,
Department of EEE,
Ahsanullah University of Science and Technology,
Dhaka, Bangladesh

II. Partha Sanjoy Dev
Engineer,
Ulkasemi Limited, Dhaka, Bangladesh

Special Thanks to:

I. Dr. Satyendra Nath Biswas
Professor,
Department of EEE,
Ahsanullah University of Science and Technology,
Dhaka, Bangladesh

II. Mahmudul Hasan Shuvo
Assistant Engineer (former),
Ulkasemi Limited, Dhaka, Bangladesh