Experiment 1

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

Objective:

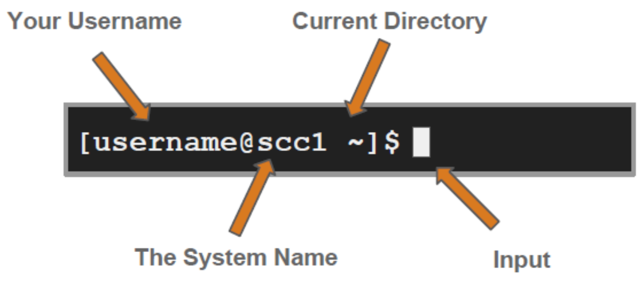
To understand and learn the basic commands required for Linux and Cadence environment for simulation of logic gates.

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

Why Linux

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

Linux: "prompt"



```
[username@scc1 ~]$
```

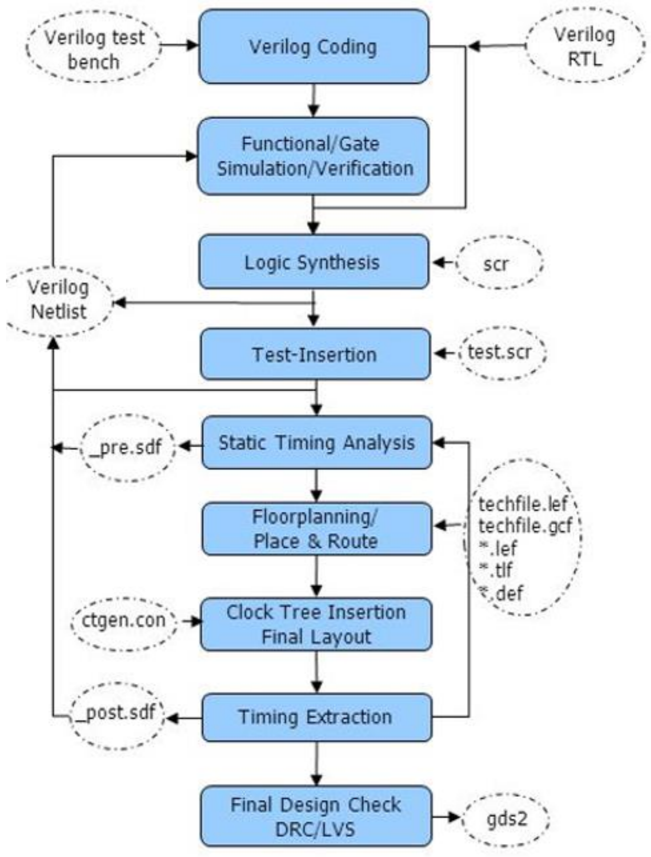
Some basic commands are:

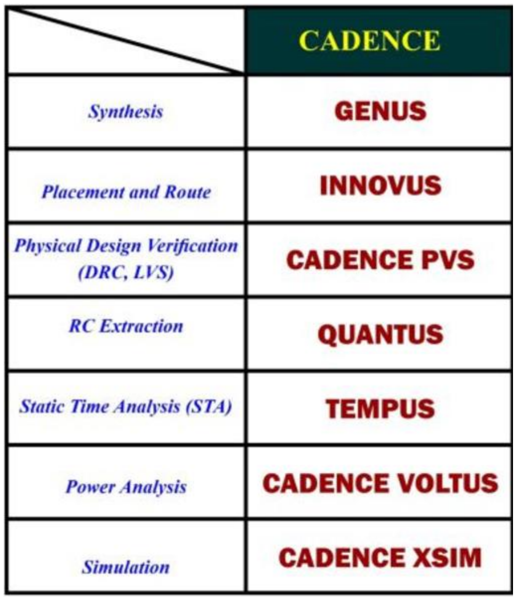
- **ls**: List directory contents.
- **cd**: Change the current directory.
- **pwd**: Print the working directory.
- **mkdir**: Create directories.
- **rm, rmdir**: Remove files and directories.
- **cp, mv**: Copy and move files or directories.

Cadence tools:

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

The design flow is as follows,



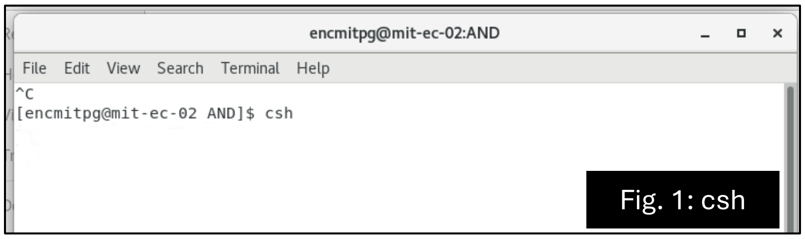


	CADENCE
Synthesis	GENUS
Placement and Route	INNOVUS
Physical Design Verification (DRC, LVS)	CADENCE PVS
RC Extraction	QUANTUS
Static Time Analysis (STA)	TEMPUS
Power Analysis	CADENCE VOLTUS
Simulation	CADENCE XSIM

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

Steps to start Cadence Tool:

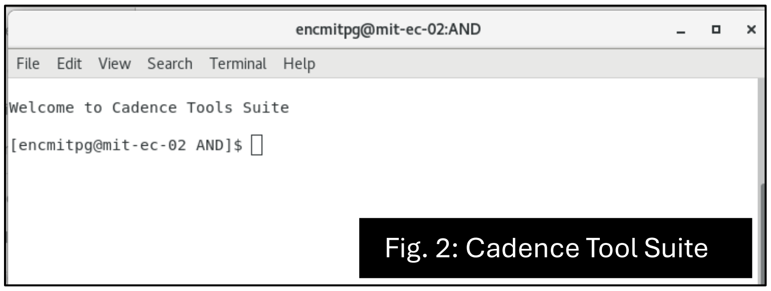
1. Creating a Workspace:
   a. In Documents create a folder and name it.
   b. Create a sub-folder and open terminal from the sub-folder.



```
encmitpg@mit-ec-02:AND
File  Edit  View  Search  Terminal  Help
^C
[encmitpg@mit-ec-02 AND]$ csh
```

Fig. 1: csh

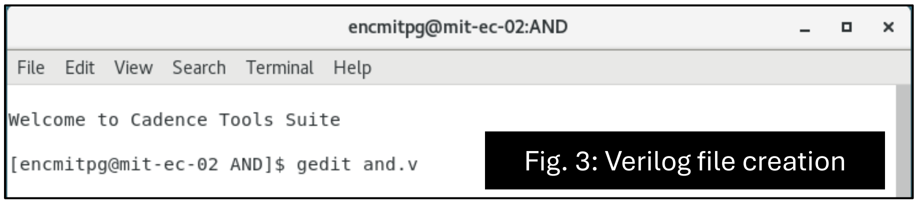
2. Functional Simulation:
   a. Enter the following command to open the cadence environment.
      csh (Invoke C-Shell)



```
Welcome to Cadence Tools Suite

[encmitpg@mit-ec-02 AND]$ 
```

Fig. 2: Cadence Tool Suite

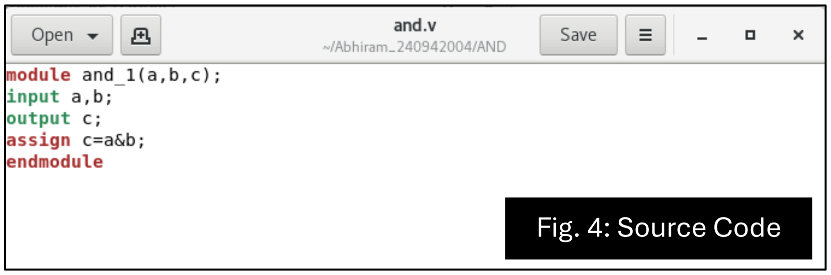
3. Creating Source Codes:
   a. In the Terminal, enter gedit <filename>.v
   b. A blank document is opened where the source code can be typed.



```
Welcome to Cadence Tools Suite

[encmitpg@mit-ec-02 AND]$ gedit and.v
```

Fig. 3: Verilog file creation

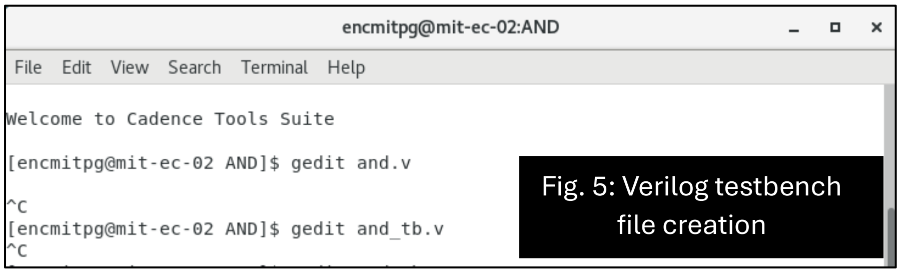
4. Source Code:
   a. Save the file and close the text file.



```
module and_1(a,b,c);
input a,b;
output c;
assign c=a&b;
endmodule
```

Fig. 4: Source Code

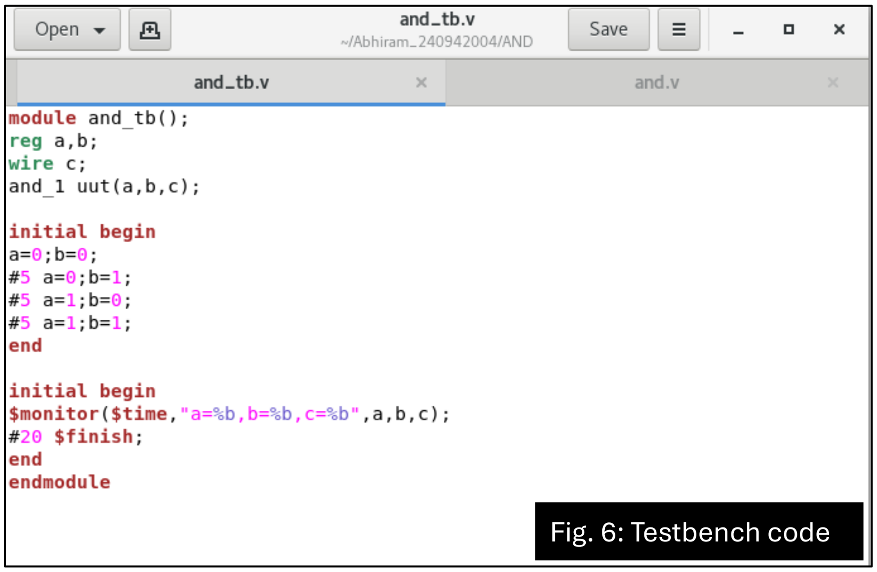
5. Creating Test bench:
   a. Similarly, create the test bench using ***gedit <filename_tb>.v*** to open a new blank document.



```
Welcome to Cadence Tools Suite

[encmitpg@mit-ec-02 AND]$ gedit and.v

^C
[encmitpg@mit-ec-02 AND]$ gedit and_tb.v
^C
```

Fig. 5: Verilog testbench file creation

b. Testbench code:



```
module and_tb();
reg a,b;
wire c;
and_1 uut(a,b,c);

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

initial begin
$monitor($time,"a=%b,b=%b,c=%b",a,b,c);
#20 $finish;
end
endmodule
```

Fig. 6: Testbench code

c. Save and close the file.

NC launch environment and incisive simulator:

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

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

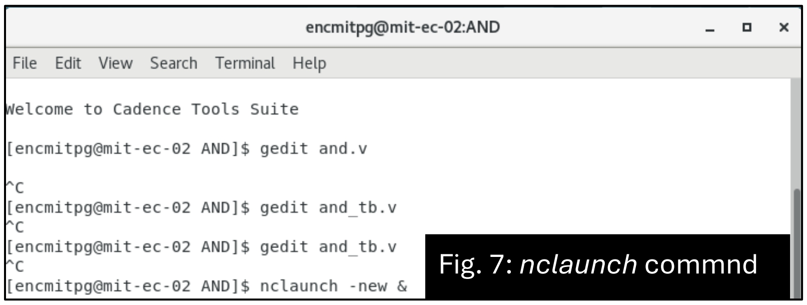
6. To Launch Simulation tool

   - write command: source /home/install/cshrc
   - *linux:/> nclaunch -new &*

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

   - *linux:/> nclaunch &*

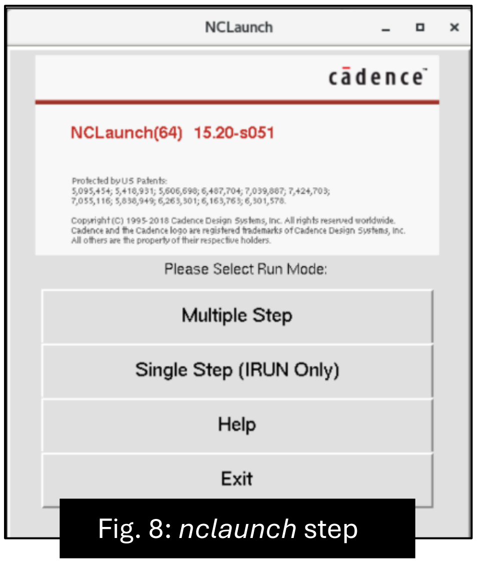
   // On subsequent calls to NCVERILOG



```
encmitpg@mit-ec-02:AND
File  Edit  View  Search  Terminal  Help

Welcome to Cadence Tools Suite

[encmitpg@mit-ec-02 AND]$ gedit and.v

^C
[encmitpg@mit-ec-02 AND]$ gedit and_tb.v
^C
[encmitpg@mit-ec-02 AND]$ gedit and_tb.v
^C
[encmitpg@mit-ec-02 AND]$ nclaunch -new &
```

Fig. 7: *nclaunch* commnd

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



NCLaunch

cadence

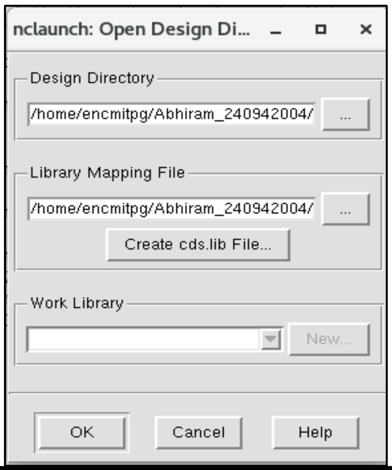
NCLaunch(64) 15.20-s051

Protected by US Patents: 5,095,454; 5,418,931; 5,606,698; 6,487,704; 7,039,887; 7,424,703; 7,055,116; 5,838,949; 6,263,301; 6,163,763; 6,301,578.

Copyright (C) 1995-2018 Cadence Design Systems, Inc. All rights reserved worldwide. Cadence and the Cadence logo are registered trademarks of Cadence Design Systems, Inc. All others are the property of their respective holders.

Please Select Run Mode:

Fig. 8: *nclaunch* step



nclaunch: Open Design Di...

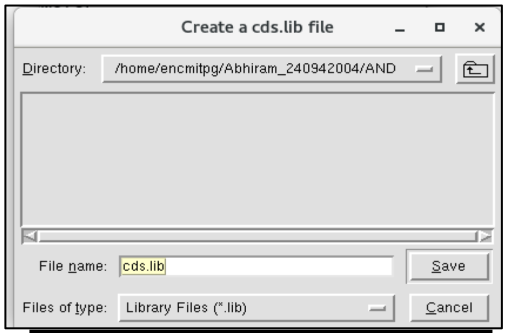
Design Directory: /home/encmitpg/Abhiram_240942004/

Library Mapping File: /home/encmitpg/Abhiram_240942004/

Work Library:

Fig. 9: *nclaunch* after selecting multiple steps

- Click the cds.lib file and save the file by clicking on Save option.
- Save cds.lib file and select the correct option for cds.lib file format based on the HDL Language and Libraries used.
- Select "Don't include any libraries (Verilog design)" from "New cds.lib file" and click On "OK" as in below figure.
- We are simulating Verilog design without using any libraries.



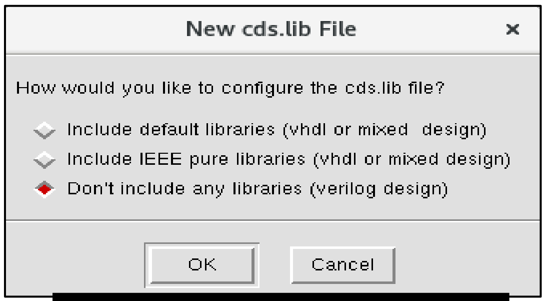
Create a cds.lib file

Directory: /home/encmitpg/Abhiram_240942004/AND

File name: cds.lib

Files of type: Library Files (*.lib)

Fig. 10: Create sdc file and save



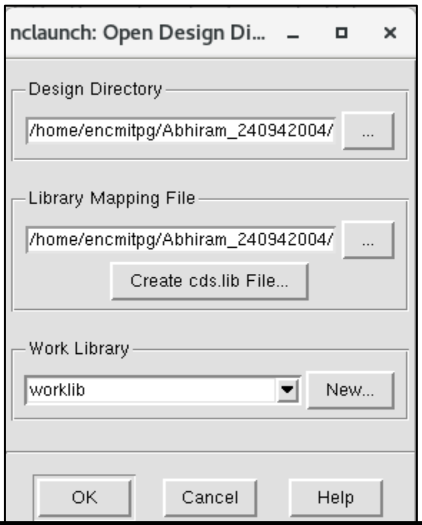
New cds.lib File

How would you like to configure the cds.lib file?

- Include default libraries (vhdl or mixed design)
- Include IEEE pure libraries (vhdl or mixed design)
- [✓ Don't include any libraries (verilog design)]

Fig. 11: Don't include libraries

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

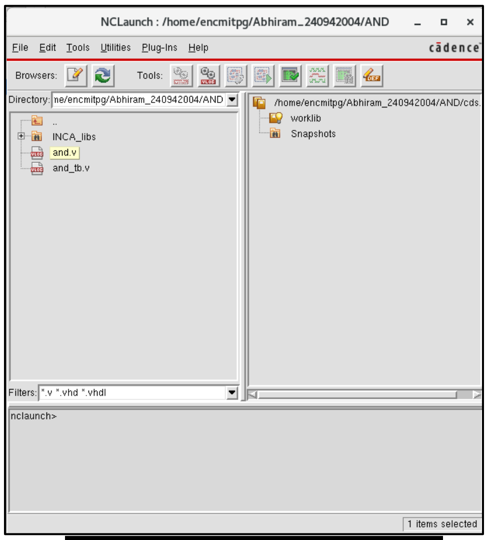


Design Directory: /home/encmitpg/Abhiram_240942004/

Library Mapping File: /home/encmitpg/Abhiram_240942004/

Work Library: worklib

Fig. 12: After saving cds file



Directory: ne/encmitpg/Abhiram_240942004/AND

```
..
INCA_libs
and.v
and_tb.v
```

```
/home/encmitpg/Abhiram_240942004/AND/cds.
  worklib
  Snapshots
```

Filters: *.v *.vhd *.vhdl

```
nclaunch>
```

1 items selected

Fig. 13: nclaunch window

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

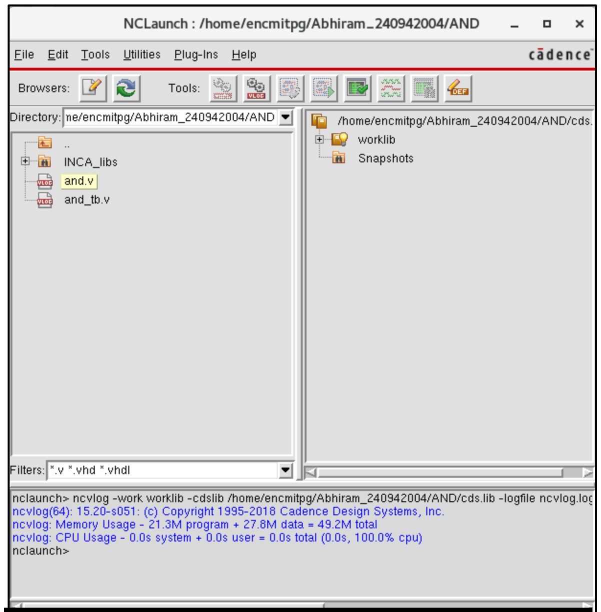
✓ To perform the function simulation, the following three steps are involved,
  - Compilation,
  - Elaboration, and
  - Simulation.

<u>Step 1:</u> <u>Compilation</u>: Process to check the correct Verilog language syntax and usage.
  - <u>Inputs</u>: Supplied are Verilog design and test bench codes.
  - <u>Outputs</u>: Compiled database created in mapped library if successful, generates report else error reported in log file.

<u>Steps for compilation</u>:
1. Create work/library directory (most of the latest simulation tools creates automatically).
2. Map the work to library created (most of the latest simulation tools creates automatically).
3. Run the compile command with compile options.

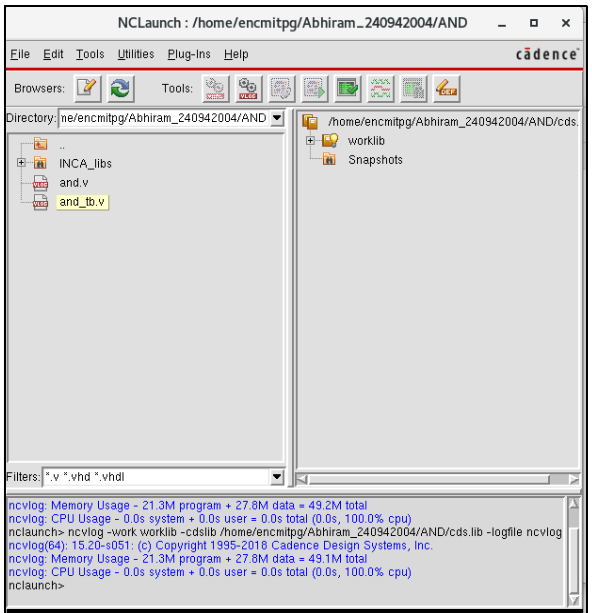
- Left side select the file and in Tools : launch Verilog compiler with current selection will get enable. Click it to compile the code.

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



```
nclaunch> ncvlog -work worklib -cdslib /home/encmitpg/Abhiram_240942004/AND/cds.lib -logfile ncvlog.log
ncvlog(64): 15.20-s051: (c) Copyright 1995-2018 Cadence Design Systems, Inc.
ncvlog: Memory Usage - 21.3M program + 27.8M data = 49.2M total
ncvlog: CPU Usage - 0.0s system + 0.0s user = 0.0s total (0.0s, 100.0% cpu)
nclaunch>
```

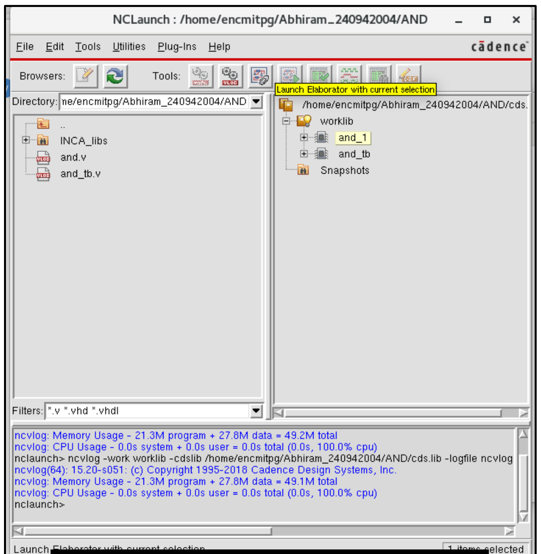
Fig. 14: Select Verilog code and launch Verilog compiler



```
ncvlog: Memory Usage - 21.3M program + 27.8M data = 49.2M total
ncvlog: CPU Usage - 0.0s system + 0.0s user = 0.0s total (0.0s, 100.0% cpu)
nclaunch> ncvlog -work worklib -cdslib /home/encmitpg/Abhiram_240942004/AND/cds.lib -logfile ncvlog
ncvlog(64): 15.20-s051: (c) Copyright 1995-2018 Cadence Design Systems, Inc.
ncvlog: Memory Usage - 21.3M program + 27.8M data = 49.1M total
ncvlog: CPU Usage - 0.0s system + 0.0s user = 0.0s total (0.0s, 100.0% cpu)
nclaunch>
```

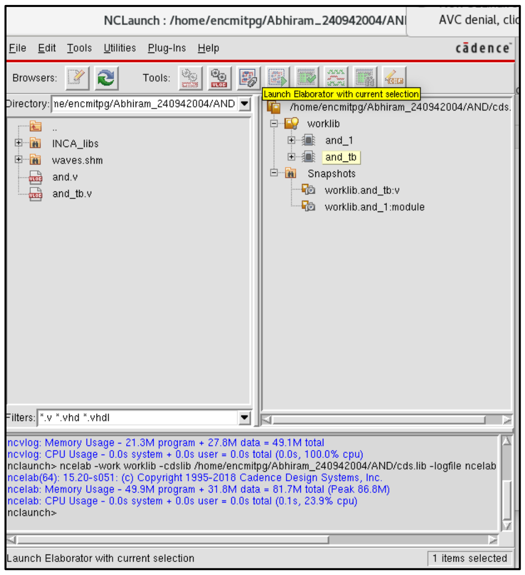
Fig. 15: Select Verilog code and launch Verilog compiler

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



```
ncvlog: Memory Usage - 21.3M program + 27.8M data = 49.2M total
ncvlog: CPU Usage - 0.0s system + 0.0s user = 0.0s total (0.0s, 100.0% cpu)
nclaunch> ncvlog -work worklib -cdslib /home/encmitpg/Abhiram_240942004/AND/cds.lib -logfile ncvlog
ncvlog(64): 15.20-s051: (c) Copyright 1995-2018 Cadence Design Systems, Inc.
ncvlog: Memory Usage - 21.3M program + 27.8M data = 49.1M total
ncvlog: CPU Usage - 0.0s system + 0.0s user = 0.0s total (0.0s, 100.0% cpu)
nclaunch>
```

Fig. 16 Compiled database in worklib



```
ncvlog: Memory Usage - 21.3M program + 27.8M data = 49.1M total
ncvlog: CPU Usage - 0.0s system + 0.0s user = 0.0s total (0.0s, 100.0% cpu)
nclaunch> ncelab -work worklib -cdslib /home/encmitpg/Abhiram_240942004/AND/cds.lib -logfile ncelab
ncelab(64): 15.20-s051: (c) Copyright 1995-2018 Cadence Design Systems, Inc.
ncelab: Memory Usage - 49.9M program + 31.8M data = 81.7M total (Peak 86.8M)
ncelab: CPU Usage - 0.0s system + 0.0s user = 0.0s total (0.1s, 23.9% cpu)
nclaunch>
```

Fig. 17: Compiled Test-bench

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

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

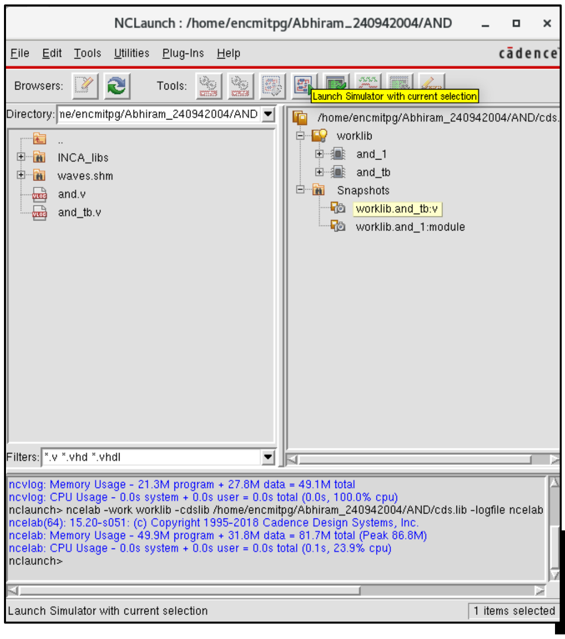
Step 2: Elaboration: To check the port connections in hierarchical design

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

Steps for elaboration – Run the elaboration command with elaborate options

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

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


NCLaunch : /home/encmitpg/Abhiram_240942004/AND
File Edit Tools Utilities Plug-Ins Help
cādence
Browsers:
Tools:
Launch Simulator with current selection
Directory: ne/encmitpg/Abhiram_240942004/AND
..
INCA_libs
waves.shm
and.v
and_tb.v
/home/encmitpg/Abhiram_240942004/AND/cds.
worklib
and_1
and_tb
Snapshots
worklib.and_tb:v
worklib.and_1:module
Filters: *.v *.vhd *.vhdl
Launch Simulator with current selection
1 items selected

```
ncvlog: Memory Usage - 21.3M program + 27.8M data = 49.1M total
ncvlog: CPU Usage - 0.0s system + 0.0s user = 0.0s total (0.0s, 100.0% cpu)
nclaunch> ncelab -work worklib -cdslib /home/encmitpg/Abhiram_240942004/AND/cds.lib -logfile ncelab
ncelab(64): 15.20-s051: (c) Copyright 1995-2018 Cadence Design Systems, Inc.
ncelab: Memory Usage - 49.9M program + 31.8M data = 81.7M total (Peak 86.8M)
ncelab: CPU Usage - 0.0s system + 0.0s user = 0.0s total (0.1s, 23.9% cpu)
nclaunch>
```

Fig. 18: Elaboration Launch Operation

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

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

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

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

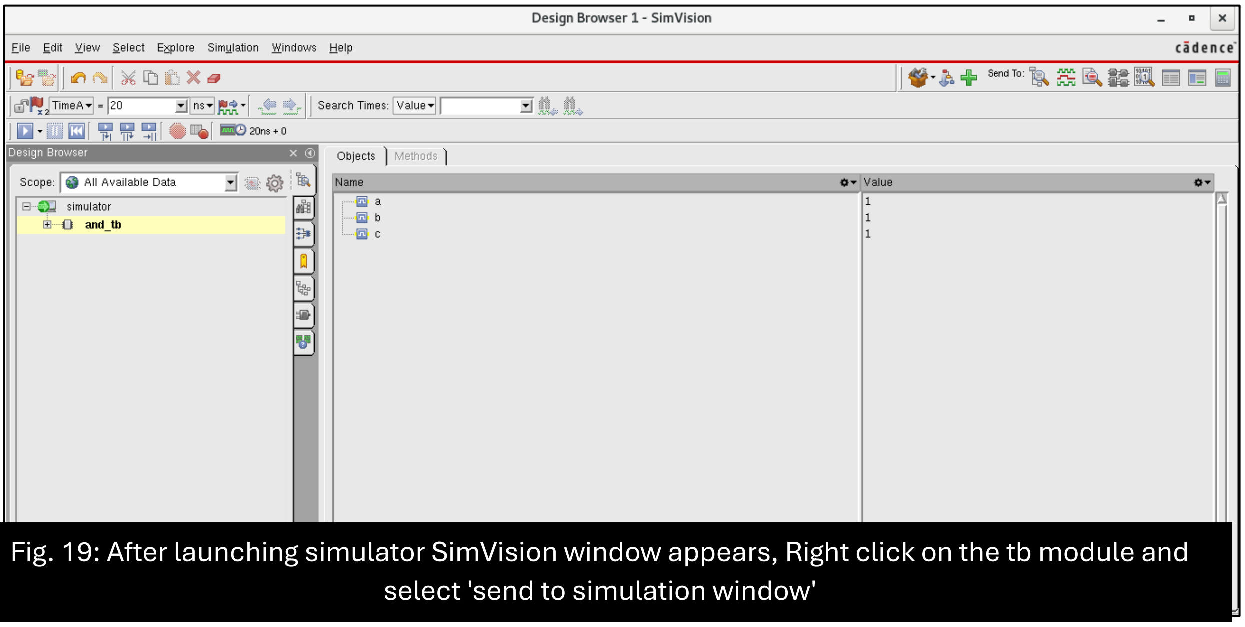

Design Browser 1 - SimVision
File Edit View Select Explore Simulation Windows Help
Scope: All Available Data
simulator
and_tb
Objects
Methods
Name
Value
a 1
b 1
c 1

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

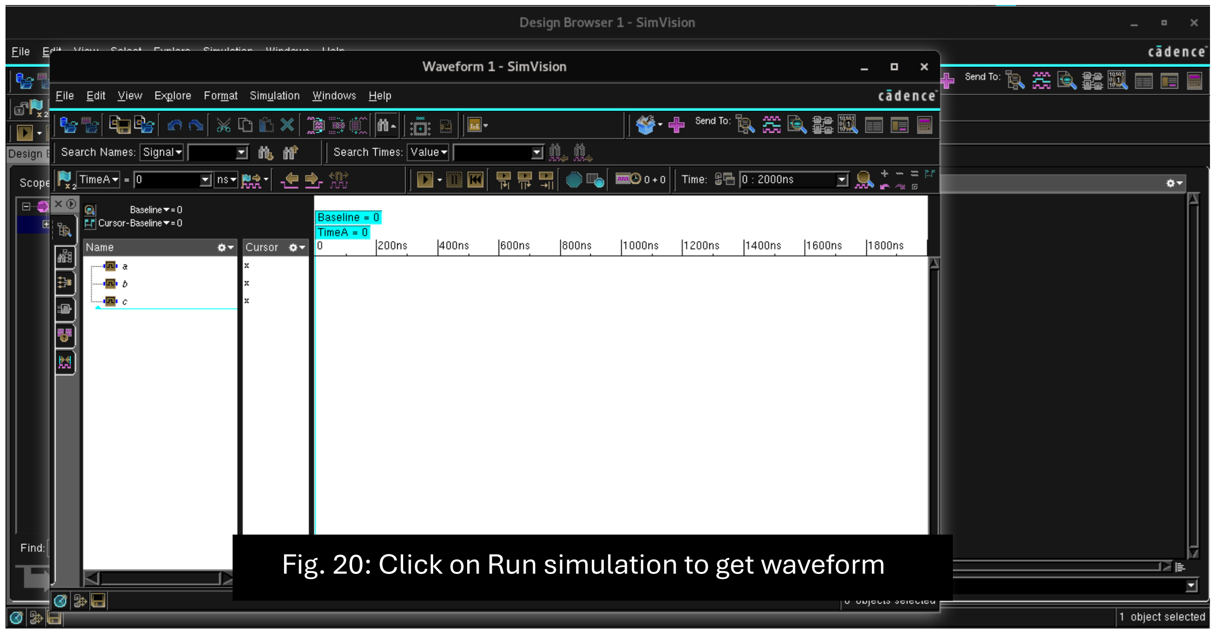


Fig. 20: Click on Run simulation to get waveform

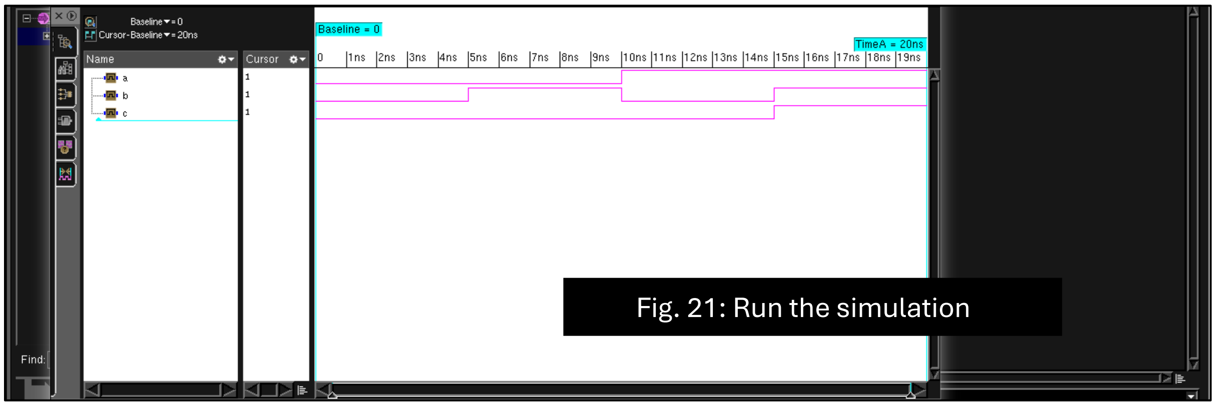
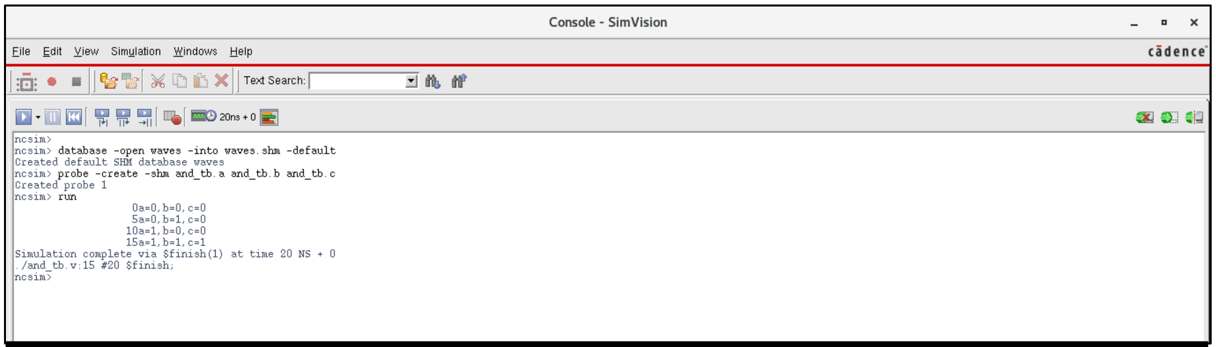
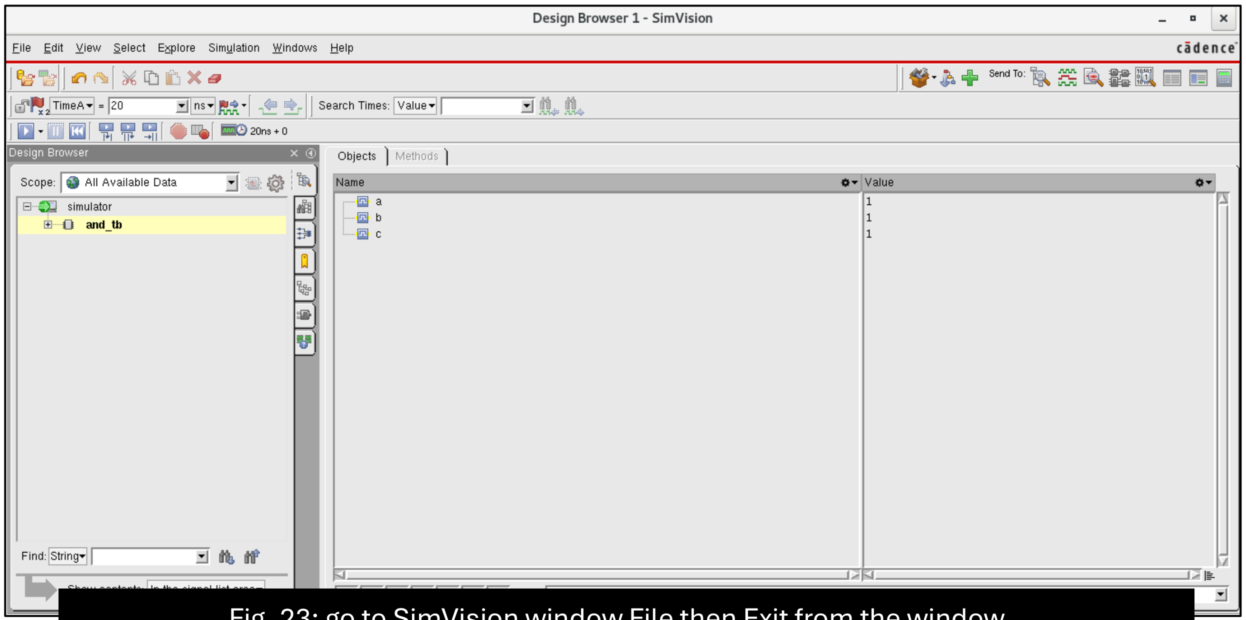

Baseline = 0
Cursor-Baseline = 20ns
Name
Cursor
a 1
b 1
c 1
Baseline = 0
TimeA = 20ns
0 1ns 2ns 3ns 4ns 5ns 6ns 7ns 8ns 9ns 10ns 11ns 12ns 13ns 14ns 15ns 16ns 17ns 18ns 19ns
Find:

Fig. 21: Run the simulation



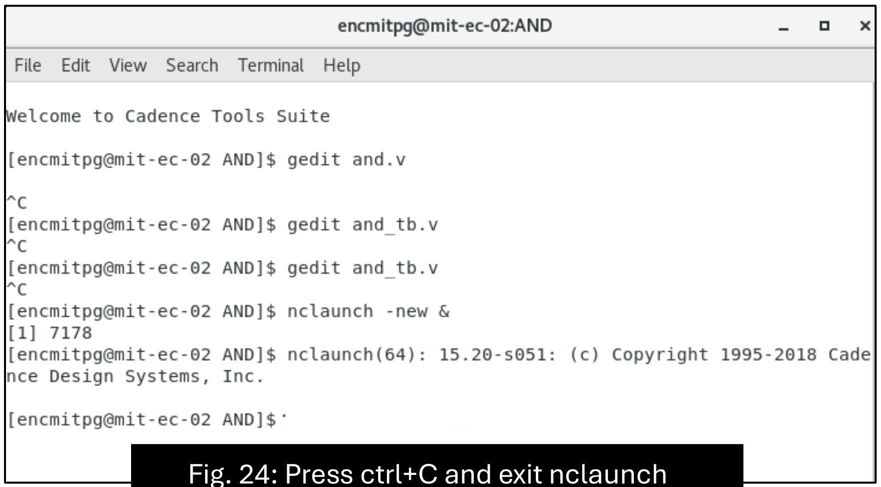
```
ncsim>
ncsim> database -open waves -into waves.shm -default
Created default SHM database waves
ncsim> probe -create -shm and_tb.a and_tb.b and_tb.c
Created probe 1
ncsim> run
                 0a=0,b=0,c=0
                 5a=0,b=1,c=0
                10a=1,b=0,c=0
                15a=1,b=1,c=1
Simulation complete via $finish(1) at time 20 NS + 0
./and_tb.v:15 #20 $finish;
ncsim>
```

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



```
Name    Value
a       1
b       1
c       1
```

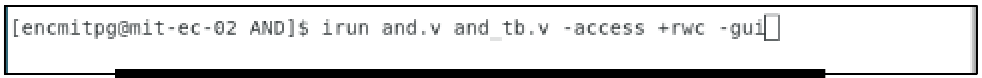
Fig. 23: go to SimVision window File then Exit from the window



```
Welcome to Cadence Tools Suite

[encmitpg@mit-ec-02 AND]$ gedit and.v

^C
[encmitpg@mit-ec-02 AND]$ gedit and_tb.v
^C
[encmitpg@mit-ec-02 AND]$ gedit and_tb.v
^C
[encmitpg@mit-ec-02 AND]$ nclaunch -new &
[1] 7178
[encmitpg@mit-ec-02 AND]$ nclaunch(64): 15.20-s051: (c) Copyright 1995-2018 Cade
nce Design Systems, Inc.

[encmitpg@mit-ec-02 AND]$
```

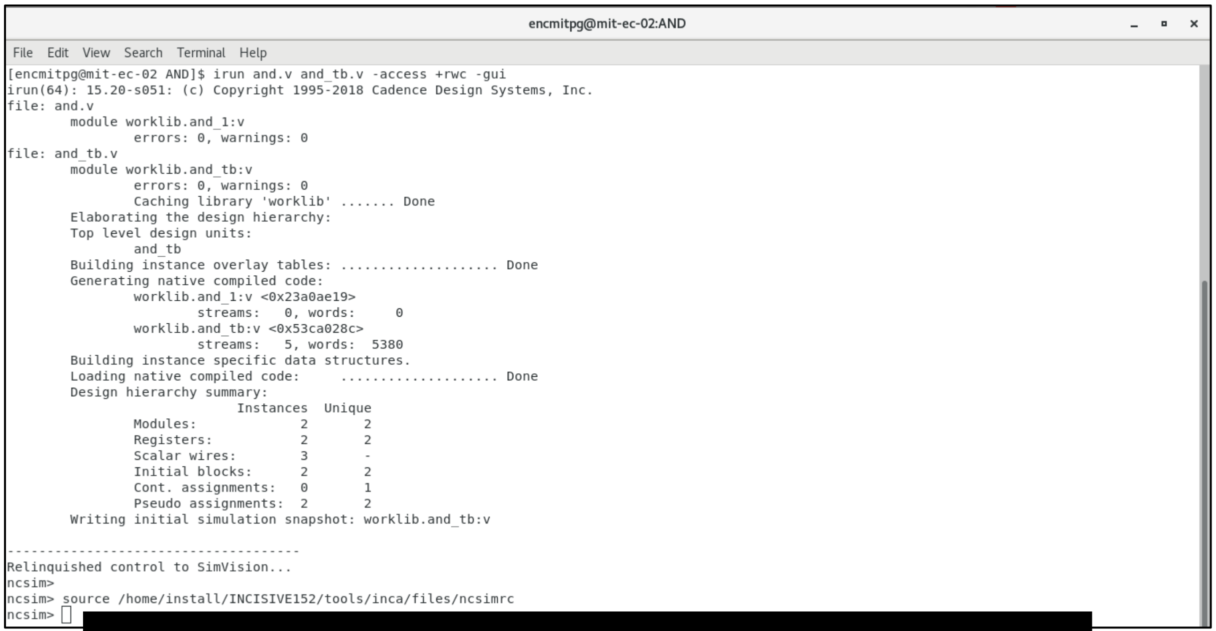
Fig. 24: Press ctrl+C and exit nclaunch

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



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

Fig. 25: irun single command to open SimVision



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

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

Fig. 26: Terminal window after successful compilation and elaboration

Exercise Problems

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

Experiment 2

Implementation of various logic circuits and waveform verifications using NCLAUNCH.

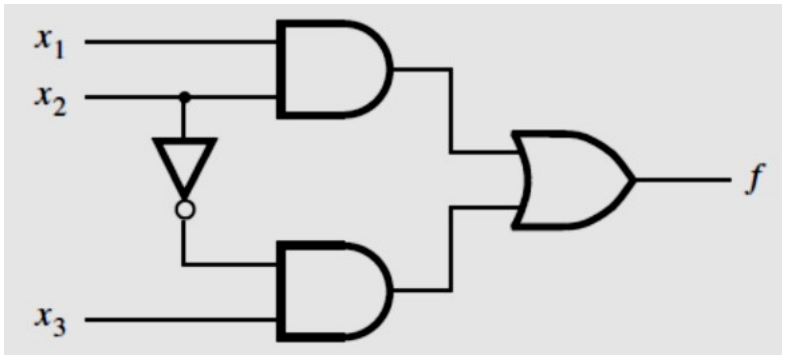
Objective:

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

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

NC Launch serves as a user-friendly interface for setting up simulation tasks, managing test benches, and visualizing results. It is designed to enhance productivity by abstracting much of the complexity associated with command-line simulation workflows. The details on how to launch NCLAUNCH are described in experiment 1 and are given on page 7.

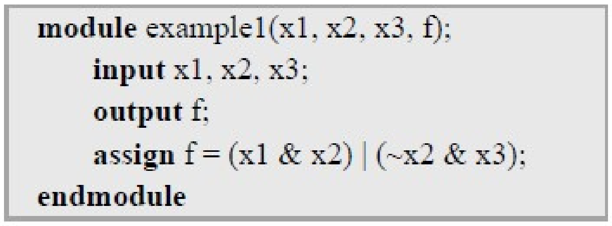
Example 1: Write Verilog code to implement the following circuit using the continuous assignment.



Truth table:

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

Verilog code:

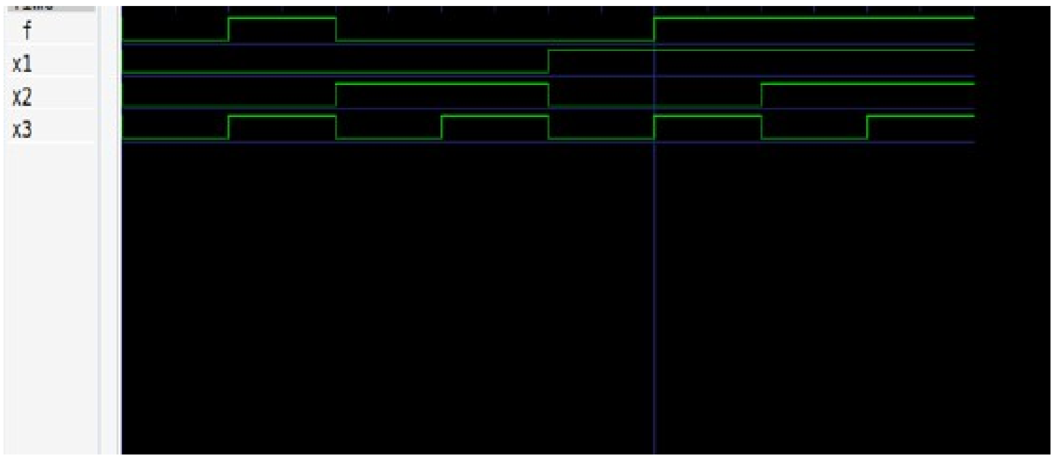


```
module example1(x1, x2, x3, f);
      input x1, x2, x3;
      output f;
      assign f = (x1 & x2) | (~x2 & x3);
endmodule
```

TestBench Code:

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

#20; x1=1'b0; x2=1'b1; x3=1'b1;
#20; x1=1'b1; x2=1'b0; x3=1'b0;
#20; x1=1'b1; x2=1'b0; x3=1'b1;
#20; x1=1'b1; x2=1'b1; x3=1'b0;
#20; x1=1'b1; x2=1'b1; x3=1'b1;
#20;
$display("Test complete");
end
endmodule
```

Waveform:



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

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

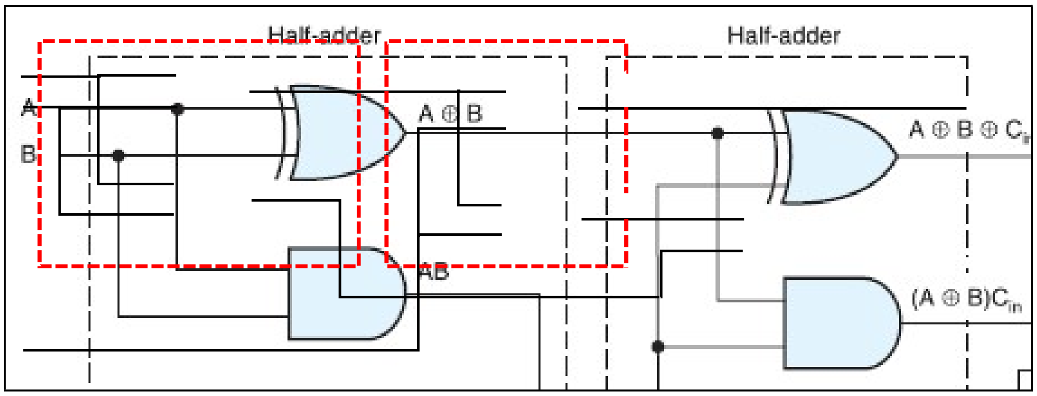
Sum (S): This is the result of adding the three input bits. The sum is given by the equation:

$$\mathbf{S = A \oplus B \oplus Cin}$$

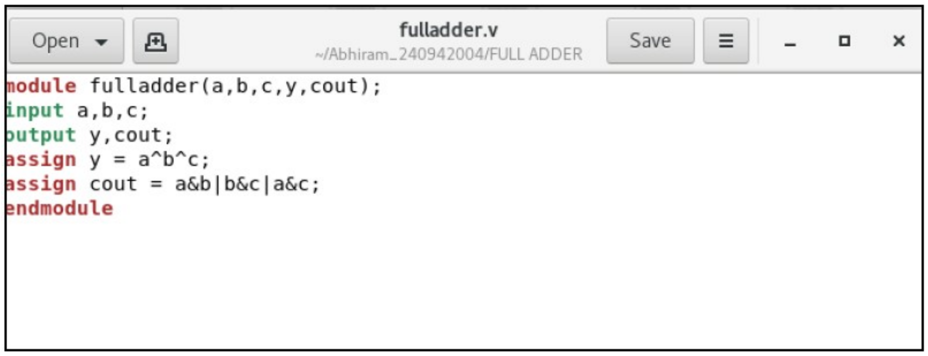
where ⊕ represents the XOR operation.

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

$$\mathbf{Cout = (A \cdot B) + (B \cdot Cin) + (Cin \cdot A)}$$

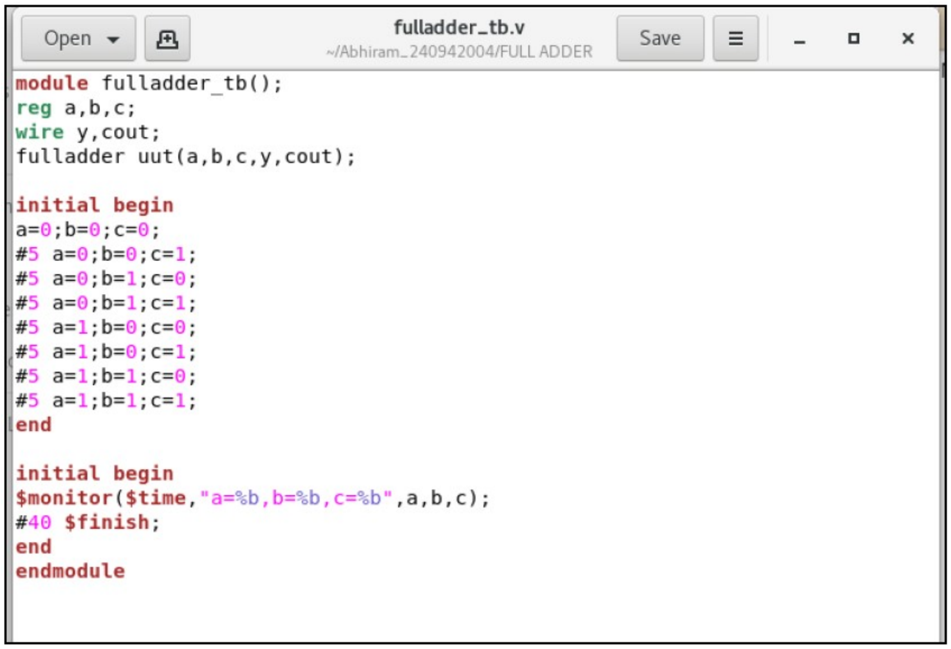


Source Code:



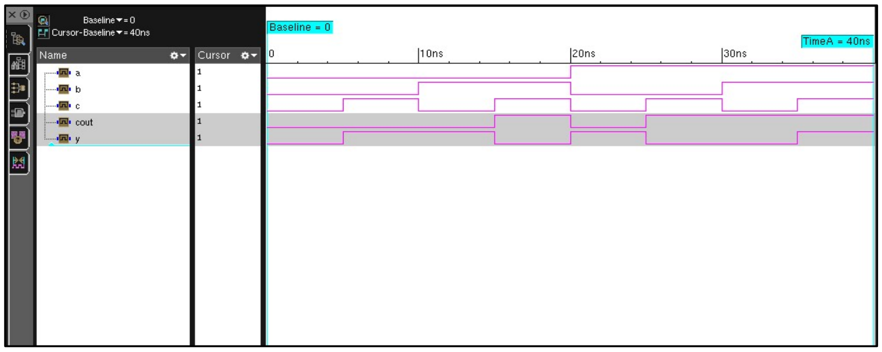
```
module fulladder(a,b,c,y,cout);
input a,b,c;
output y,cout;
assign y = a^b^c;
assign cout = a&b|b&c|a&c;
endmodule
```

Test Bench



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

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

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

Waveform:



Exercise Problems

1. Write a Verilog code for full subtractor and verify the design by simulation.
2. Implement a 16:4 Decoder using Verilog and verify its output.
3. Write the source and testbench code 4:1 MUX and 1:4 DEMUX.
4. Write the source and testbench code 4-bit priority encoder.
5. Write the source and testbench code to simulate gray to binary code converters

EXPERIMENT- 3

Introduction of various abstraction levels and simulation of logic circuits

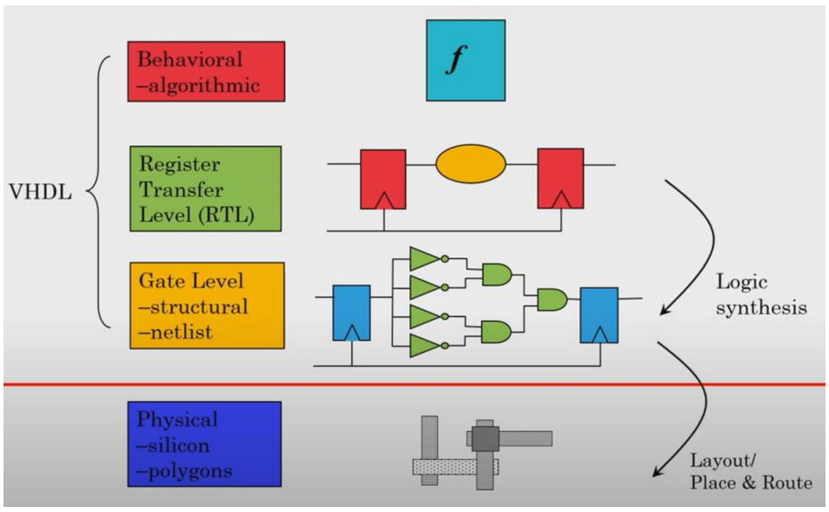
Objective:

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

Theory:

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

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



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

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

Syntax:

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

```
inout [msb:lsb] inout_port_list;
... statements...
endmodule
```

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

assign Y = Boolean Expression using variables and operators.

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

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

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

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

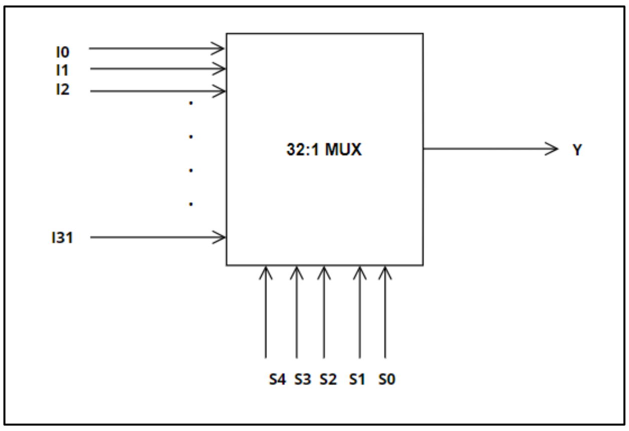
<u>32:1 mux</u>:

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

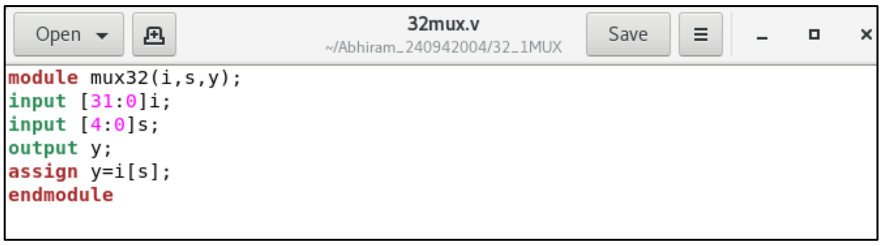
Components of a 32:1 MUX:

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

Circuit:

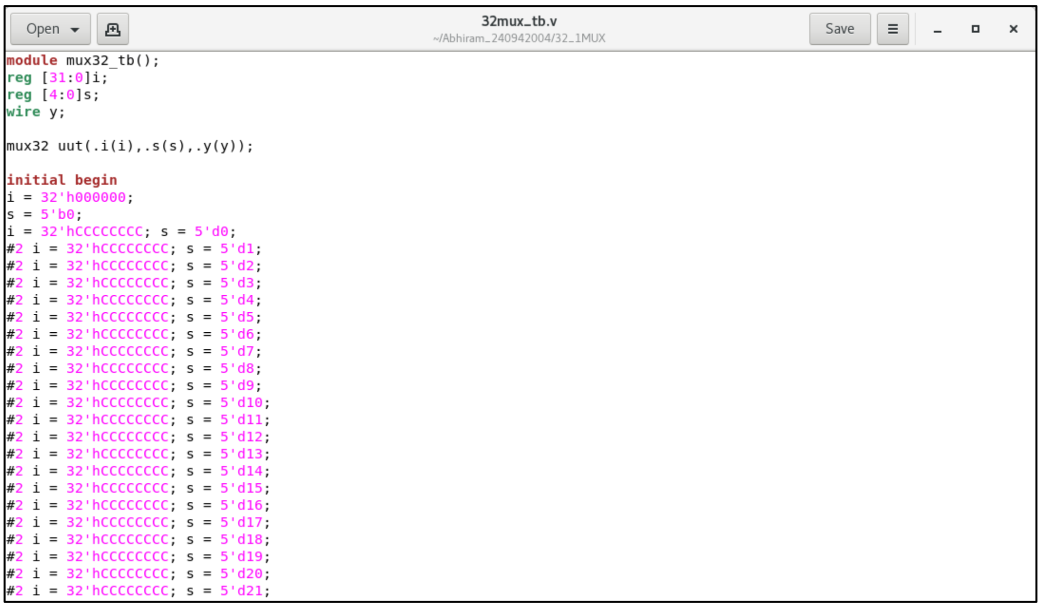


Source Code :

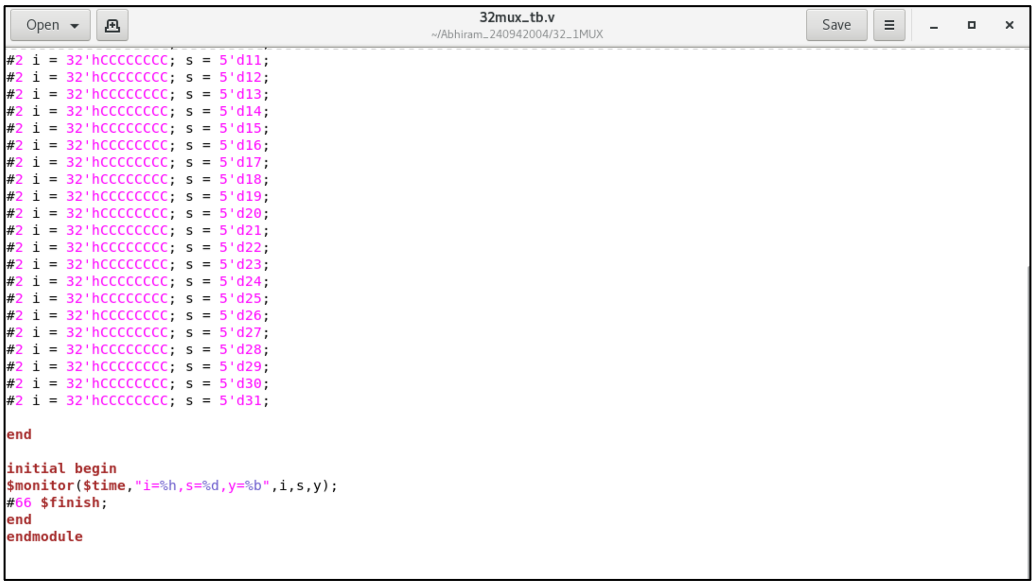


```
module mux32(i,s,y);
input [31:0]i;
input [4:0]s;
output y;
assign y=i[s];
endmodule
```

Testbench:

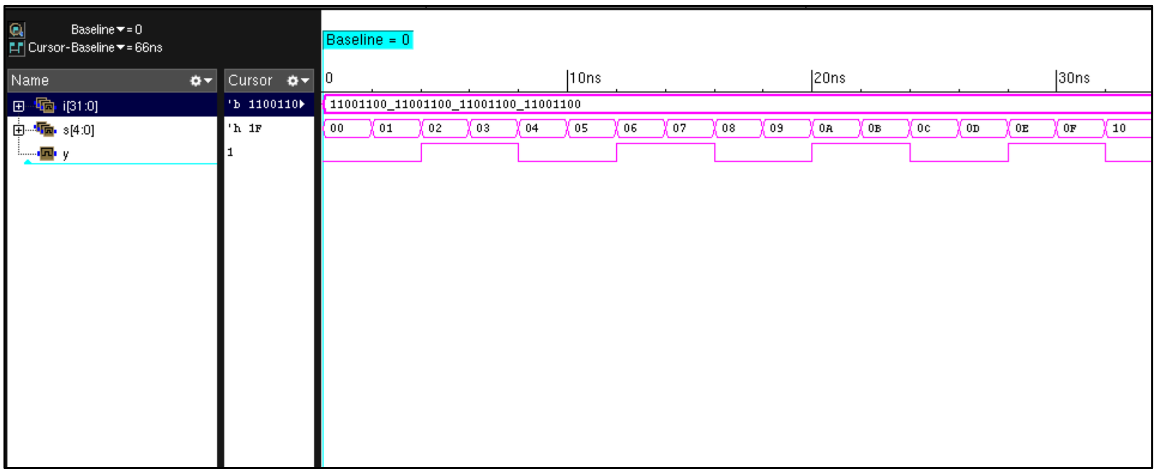


```
module mux32_tb();
reg [31:0]i;
reg [4:0]s;
wire y;

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

initial begin
i = 32'h000000;
s = 5'b0;
i = 32'hCCCCCCCC; s = 5'd0;
#2 i = 32'hCCCCCCCC; s = 5'd1;
#2 i = 32'hCCCCCCCC; s = 5'd2;
#2 i = 32'hCCCCCCCC; s = 5'd3;
#2 i = 32'hCCCCCCCC; s = 5'd4;
#2 i = 32'hCCCCCCCC; s = 5'd5;
#2 i = 32'hCCCCCCCC; s = 5'd6;
#2 i = 32'hCCCCCCCC; s = 5'd7;
#2 i = 32'hCCCCCCCC; s = 5'd8;
#2 i = 32'hCCCCCCCC; s = 5'd9;
#2 i = 32'hCCCCCCCC; s = 5'd10;
#2 i = 32'hCCCCCCCC; s = 5'd11;
#2 i = 32'hCCCCCCCC; s = 5'd12;
#2 i = 32'hCCCCCCCC; s = 5'd13;
#2 i = 32'hCCCCCCCC; s = 5'd14;
#2 i = 32'hCCCCCCCC; s = 5'd15;
#2 i = 32'hCCCCCCCC; s = 5'd16;
#2 i = 32'hCCCCCCCC; s = 5'd17;
#2 i = 32'hCCCCCCCC; s = 5'd18;
#2 i = 32'hCCCCCCCC; s = 5'd19;
#2 i = 32'hCCCCCCCC; s = 5'd20;
#2 i = 32'hCCCCCCCC; s = 5'd21;
```



```
#2 i = 32'hCCCCCCCC; s = 5'd11;
#2 i = 32'hCCCCCCCC; s = 5'd12;
#2 i = 32'hCCCCCCCC; s = 5'd13;
#2 i = 32'hCCCCCCCC; s = 5'd14;
#2 i = 32'hCCCCCCCC; s = 5'd15;
#2 i = 32'hCCCCCCCC; s = 5'd16;
#2 i = 32'hCCCCCCCC; s = 5'd17;
#2 i = 32'hCCCCCCCC; s = 5'd18;
#2 i = 32'hCCCCCCCC; s = 5'd19;
#2 i = 32'hCCCCCCCC; s = 5'd20;
#2 i = 32'hCCCCCCCC; s = 5'd21;
#2 i = 32'hCCCCCCCC; s = 5'd22;
#2 i = 32'hCCCCCCCC; s = 5'd23;
#2 i = 32'hCCCCCCCC; s = 5'd24;
#2 i = 32'hCCCCCCCC; s = 5'd25;
#2 i = 32'hCCCCCCCC; s = 5'd26;
#2 i = 32'hCCCCCCCC; s = 5'd27;
#2 i = 32'hCCCCCCCC; s = 5'd28;
#2 i = 32'hCCCCCCCC; s = 5'd29;
#2 i = 32'hCCCCCCCC; s = 5'd30;
#2 i = 32'hCCCCCCCC; s = 5'd31;

end

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

Waveform:

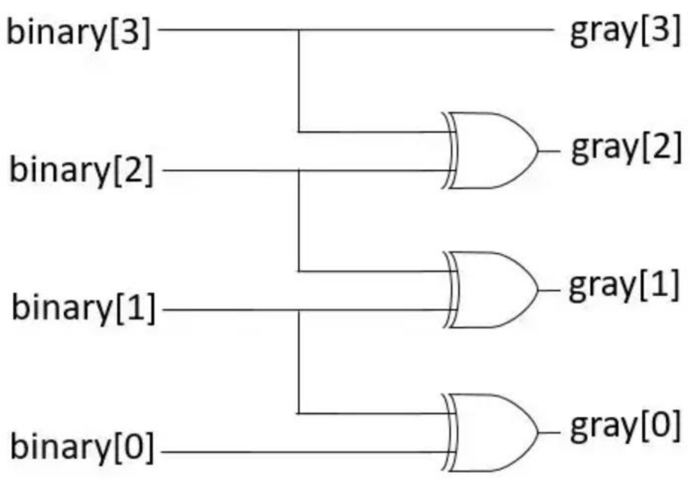


2. **Write a dataflow Verilog code for 4-bit binary to gray code converter and verify the design by simulation.**

A 4-bit Binary to Gray code converter is a digital circuit that converts a 4-bit Binary code input into its equivalent 4-bit gray code output.
Components of a 4-bit Binary to Gray Converter:
1. The most significant bit (MSB) of the Gray code is the same as the MSB of the binary number.
2. Each subsequent bit of the Gray code is obtained by XORing the current binary bit with the previous binary bit.

Conversion Formula:
- gray[3]=binary[3]
- gray[2]= binary [3]⊕ binary [2]
- gray[1]= binary [2]⊕ binary [1]
- gray[0]= binary [1]⊕ binary [0]

Circuit:



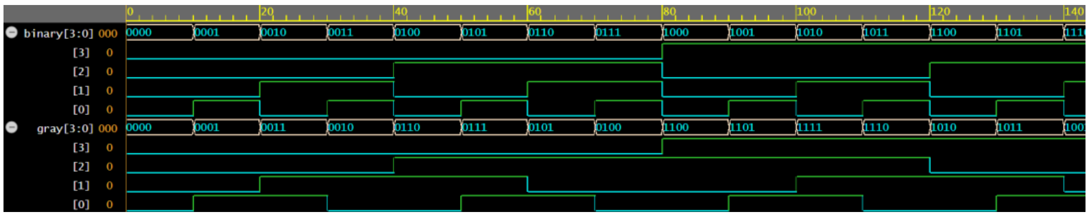
Source Code :

```
module binary_to_gray ( input  [3:0] binary,  output [3:0] gray );

   assign gray[3] = binary[3];                  // MSB remains the same
   assign gray[2] = binary[3] ^ binary[2];   // XOR of bit 3 and bit 2
   assign gray[1] = binary[2] ^ binary[1];   // XOR of bit 2 and bit 1
   assign gray[0] = binary[1] ^ binary[0];   // XOR of bit 1 and bit 0

endmodule
```

Testbench:

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

   // Instantiate the binary_to_gray module
   binary_to_gray uut ( .binary(binary), .gray(gray));

   initial begin
      $monitor("Binary = %b, Gray = %b", binary, gray);

      // Test cases
      binary = 4'b0000; #10;
      binary = 4'b0001; #10;
      binary = 4'b0010; #10;
      binary = 4'b0011; #10;
      binary = 4'b0100; #10;
      binary = 4'b0101; #10;
      binary = 4'b0110; #10;
      binary = 4'b0111; #10;
      binary = 4'b1000; #10;
      binary = 4'b1001; #10;
      binary = 4'b1010; #10;
      binary = 4'b1011; #10;
      binary = 4'b1100; #10;
      binary = 4'b1101; #10;
      binary = 4'b1110; #10;
      binary = 4'b1111; #10;

      $finish;
   end

endmodule
```

Waveform:



Exercise Problems

1. Write a dataflow Verilog code for following digital building blocks and verify the design by simulation: 5:32 Decoder
2. Write a dataflow Verilog code for 4-bit binary multiplier and verify the design by simulation.
3. Write a dataflow Verilog code for a BCD-to-seven-segment decoder and verify the design by simulation.
4. Write a dataflow Verilog code for binary to Excess-3 code and verify the design by simulation.
5. Write a dataflow Verilog code for 8:1 MUX with enable input and verify the design by simulation.

EXPERIMENT- 4

Simulation of logic circuits using Behavioural Modeling

Objective:

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

Theory:

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

i) Initial statement
ii) Always statement

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

```
initial
[timing_control] procedural_statement
```

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

```
always
[timing_control] procedural_statement
```

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

Exercise Problems

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

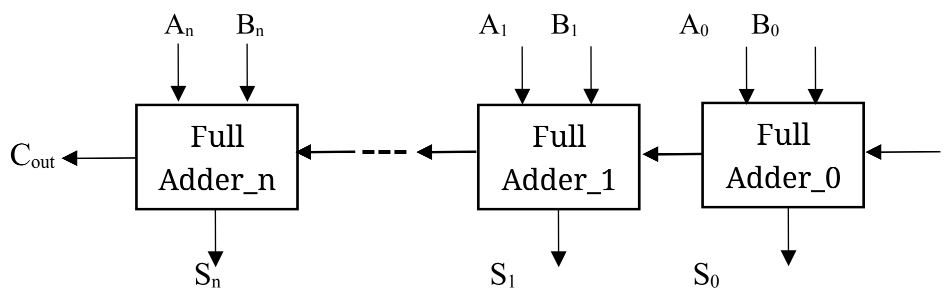
   Inputs:

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

   Outputs:

     - n-bit sum: S = S(n-1), S(n-2), ..., S0

- Carry-out: The carry-out from the most significant bit (Cout) is the final carry result after adding the two numbers.



Source Code:

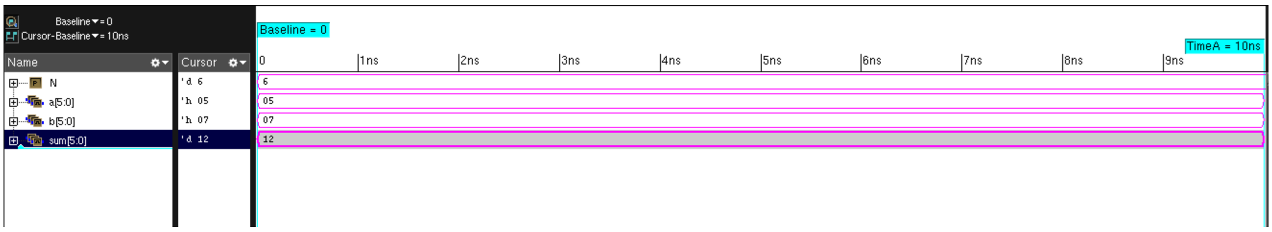
```
module nbitadder(a,b,sum);
parameter N=6;
input [N-1:0]a;
input [N-1:0]b;
output [N-1:0]sum;
wire [N-1:0]cout;
wire carry;
genvar i;
for(i=0;i<N;i=i+1)
begin
if(i==0)
fulladder fulladder_0(a[0],b[0],0,sum[0],cout[0]);
else
fulladder fulladder_0(a[i],b[i],cout[i-1],sum[i],cout[i]);
end
assign carry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

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

Testbench:

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

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

initial begin
a = 6'd5;
```

```
b = 6'd7;
end

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

Waveform:



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

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

Input/Output:

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

Behavior:

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

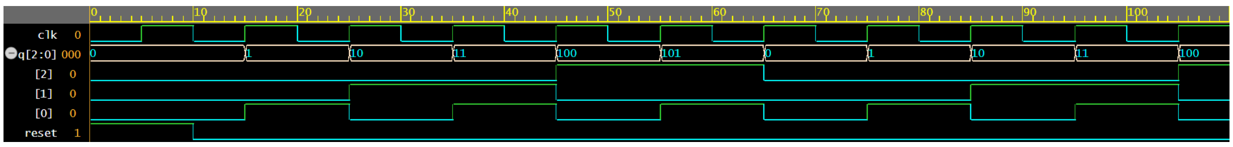
Source Code:

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

Testbench:

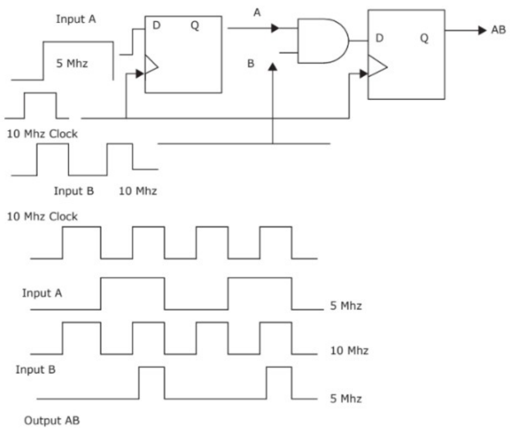
```
 module tb_mod6_counter;
    reg clk;              // Clock signal
    reg reset;            // Reset signal
    wire [2:0] q;         // Counter output

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

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

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

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

        $finish;
    end
endmodule
```

Waveform:



Exercise Problems

1. Write the behavioural Verilog code for synchronous mod 8 counter and verify the design by simulation.
2. Write a behavioural Verilog code and test bench to model and simulate 4 bit synchronous up/ down counter using Cadence tool.
3. Write sequential Verilog code for 4-bit universal shift register and verify the design by simulation.
4. Write sequential Verilog code for Master Slave JK flipflop and verify the design by simulation.
5. Write the behavioural code given circuit with input signals A and B and output AB.



Input A
D
Q
A
D
Q
AB
5 Mhz
B
10 Mhz Clock
Input B
10 Mhz
10 Mhz Clock
Input A
5 Mhz
10 Mhz
Input B
5 Mhz
Output AB