A

Laboratory Manual for

VLSI Design
(3151105)

B. E. Electronics and Communication
Semester 5

Directorate of Technical Education, Gandhinagar, Gujarat

VGEC, Chandkheda, Ahmedabad

Department of Electronics & Communication Engineering

Certificate

This is to certify that Mr./Ms. _____
Enrollment No. _____ of B. E. Semester-V Electronics and Communication Engineering of this Institute (GTU Code: 11) has satisfactorily completed the Practical / Tutorial work for the subject **VLSI Design (3151105)** for the academic year 2025-26.

Place: _____
Date: _____

Name and Sign of Faculty member

Head of the Department

Practical – Course Outcome Matrix

Course Outcomes (COs):

1. **To understand fabrication and operation of MOSFET device.**
2. **To obtain and verify the mathematical model of MOSFET device.**
3. **Analyze and design Static and Dynamic CMOS Circuits.**
4. **To understand techniques of DFT.**
5. **To write programs in Verilog/VHDL for combinational and sequential circuits and realize them on FPGA.**

Sr. No.	Objective(s) of Experiment	CO 1	CO2	CO 3	CO 4	CO 5
1.	Introduction to FPGA and CPLD.					√
2.	Introduction to Hardware Description Languages (HDLs).					√
3.	Introduction to Xilinx Tools.					√
4.	Implement all the basic Logic Gates and Boolean functions using different modeling styles in Verilog/ VHDL: a. Structural modeling b. Dataflow modeling c. Behavioural modeling			√		√
5.	Design Adder & Subtractor using Verilog / VHDL. a. Half Adder (structural and dataflow modeling) b. Full Adder using Half Adder (structural modeling) c. Full Adder (dataflow and behavioural modeling) d. Half Subtractor e. Full Subtractor			√		√
6.	Design Binary-to-Gray & Gray-to-Binary encoder using Verilog/VHDL			√		√
7.	Design Multiplexer and Demultiplexer using Verilog/ VHDL. a. 2:1 Mux (dataflow and behavioural modeling) b. 4:1 Mux (structural and dataflow modeling) c. 8:1 Mux (using 4:1 and 2:1 Mux: structural modeling) d. 16:1 Mux (using behavioural modeling & 4:1 Mux: structural modeling) e.1:8 Demux			√		√
8.	Design 2:4, 3:8, 4:16 Decoders using Verilog/VHDL			√		√
9.	Design 4*2, 8*3 Priority Encoder using Verilog / VHDL.			√		√
10.	Design 4-bit Comparator using Verilog / VHDL.			√		√
11.	Design BCD and Ripple Carry Adder using Verilog / VHDL.			√		√
12.	Design of S-R and D latches using structural / behavioral modeling usingVerilog/VHDL.			√		√
13.	Design positive edge triggered D-FF with asynchronous /synchronous active high reset using Verilog / VHDL.			√		√

14.	Design serial in serial out and serial in parallel out shift registers using Verilog/VHDL.			√		√
15.	Design Counter using Verilog/VHDL. a. BCD Counter. b. Up-Down Counter			√		√
16.	Introduction to LTSpice tool.	√		√		
17.	Implementation of Resistive load and CMOS inverters using LTSpice .	√		√		
18.	Implementation of CMOS NAND and NOR gate using LTSpice .	√		√		

Index
(Progressive Assessment Sheet)

Sr. No.	Title of Experiment	Page No.	Date of performance	Date of submission	Assessment Marks	Sign. of Teacher with date	Remarks
1	Introduction to FPGA and CPLD.						
2	Introduction to Hardware Description Languages (HDLs).						
3	Introduction to Xilinx Tools.						
4	Implement all the basic Logic Gates and Boolean functions using different modeling styles in Verilog/ VHDL: a. Structural modeling b. Dataflow modeling c. Behavioural modeling						
5	Design Adder & Subtractor using Verilog / VHDL. a. Half Adder (structural and dataflow modeling) b. Full Adder using Half Adder (structural modeling) c. Full Adder (dataflow and behavioural modeling) d. Half Subtractor e. Full Subtractor						
6	Design Binary-to-Gray & Gray-to-Binary encoder using Verilog/VHDL						
7	Design Multiplexer and Demultiplexer using Verilog/ VHDL. f. 2:1 Mux (dataflow and behavioural modeling) g. 4:1 Mux (structural and dataflow modeling) h. 8:1 Mux (using 4:1 and 2:1 Mux: structural modeling) i. 16:1 Mux (using behavioural modeling & 4:1 Mux: structural modeling) 1:8 Demux						
8	Design 2:4, 3:8, 4:16 Decoders using Verilog/VHDL						
9	Design 4*2, 8*3 Priority Encoder using Verilog / VHDL.						
10	Design 4-bit Comparator using Verilog / VHDL.						
11	Design BCD and Ripple Carry Adder using Verilog / VHDL.						
12	Design of S-R and D latches using structural / behavioral modeling usingVerilog/VHDL.						

13	Design positive edge triggered D-FF with asynchronous /synchronous active high reset using Verilog / VHDL.						
14	Design serial in serial out and serial in parallel out shift registers using Verilog/VHDL.						
15	Design Counter using Verilog/VHDL. a. BCD Counter. b. Up-Down Counter						
16	Introduction to LTSpice tool.						
17	Implementation of Resistive load and CMOS inverters using LTSpice.						
18	Implementation of CMOS NAND and NOR gate using LTSpice.						
	Total						

Experiment No: 1

Date: _____

Aim: Introduction to FPGA & CPLD.

Competency and Practical Skills:

Relevant CO:

Objectives:

Equipment / Instruments: Laptop or Computer with Xilinx Vivado.

Basic Theory:

What is an FPGA?

Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing. This feature distinguishes FPGAs from Application Specific Integrated Circuits (ASICs), which are custom manufactured for specific design tasks. Although one-time programmable (OTP) FPGAs are available, the dominant types are SRAM based which can be reprogrammed as the design evolves.

What are the differences between an ASIC and an FPGA?

_____

_____

_____

_____

_____

_____

_____

_____

_____

FPGA Architecture:

A basic FPGA architecture shown below consists of thousands of fundamental elements called configurable logic blocks (CLBs) surrounded by a system of programmable interconnects, called a fabric, that routes signals between CLBs. Input/output (I/O) blocks interface between the FPGA and external devices. Depending on the manufacturer, the CLB may also be referred to as a logic block (LB), a logic element (LE) or a logic cell (LC).

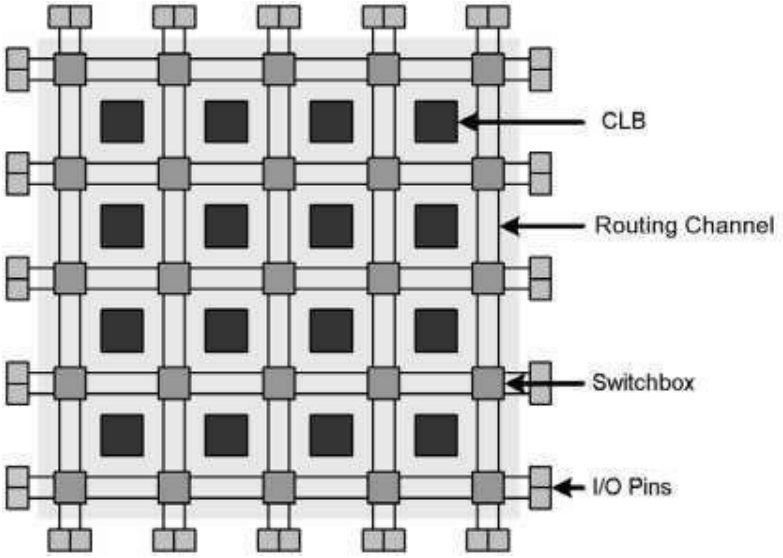


Fig 1.1 General Architecture of FPGA

Explain the structure of Switch matrix:

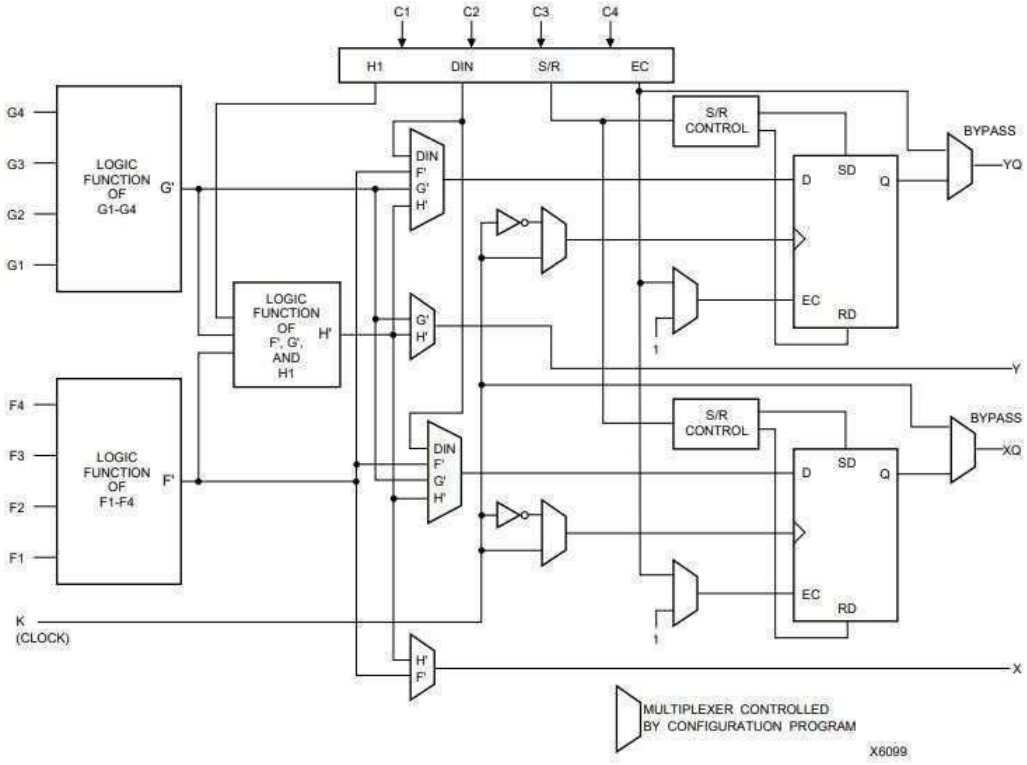


Fig 1.2 Simplified Block Diagram of XC4000-Families CLB

The XC4000 families achieve high speed through advanced semiconductor technology and through improved architecture, and supports system clock rates of up to 50 MHz. Compared to older Xilinx FPGA families, the XC4000 families are more powerful, offering on-chip RAM and wide-input decoders. They are more versatile in their applications, and design cycles are faster due to a combination of increased routing resources and more sophisticated software. And last, but not least,

they more than double the available complexity, up to the 20,000-gate level. The XC4000 families have 16 members, ranging in complexity from 2,000 to 25,000 gates.

The CLBs provide the functional elements for constructing the user's logic. The most important one is a more powerful and flexible CLB surrounded by a versatile set of routing resources, resulting in more "effective gates per CLB." The principal CLB elements are shown in Figure 1.2. Each new CLB also packs a pair of flip-flops and two independent 4-input function generators. The two function generators offer designers plenty of flexibility because most combinatorial logic functions need less than four inputs. Consequently, the design-software tools can deal with each function generator independently, thus improving cell usage.

FPGA Applications:

- **Aerospace & Defense -** Radiation-tolerant FPGAs along with intellectual property for image processing, waveform generation, and partial reconfiguration for SDRs.
- **ASIC Prototyping -** ASIC prototyping with FPGAs enables fast and accurate SoC system modeling and verification of embedded software
- **Automotive -** Automotive silicon and IP solutions for gateway and driver assistance systems, comfort, convenience, and in-vehicle infotainment. - Learn how AMD FPGA's enable Automotive Systems
- **Broadcast & Pro AV** - Adapt to changing requirements faster and lengthen product life cycles with Broadcast Targeted Design Platforms and solutions for high-end professional broadcast systems.
- **Consumer Electronics -** Cost-effective solutions enabling next generation, full-featured consumer applications, such as converged handsets, digital flat panel displays, information appliances, home networking, and residential set top boxes.
- **Data Center -** Designed for high-bandwidth, low-latency servers, networking, and storage applications to bring higher value into cloud deployments.
- **High Performance Computing and Data Storage** - Solutions for Network Attached Storage (NAS), Storage Area Network (SAN), servers, and storage appliances.
- **Industrial -** AMD FPGAs and targeted design platforms for Industrial, Scientific and Medical (ISM) enable higher degrees of flexibility, faster time-to-market, and lower overall non-recurring engineering costs (NRE) for a wide range of applications such as industrial imaging and surveillance, industrial automation, and medical imaging equipment.
- **Medical** - For diagnostic, monitoring, and therapy applications, the Virtex FPGA and Spartan™ FPGA families can be used to meet a range of processing, display, and I/O interface requirements.
- **Video & Image Processing -** FPGAs and targeted design platforms enable higher degrees of flexibility, faster time-to-market, and lower overall non-recurring engineering costs (NRE) for a wide range of video and imaging applications.
- **Wired Communications** - End-to-end solutions for the Reprogrammable Networking Linecard Packet Processing, Framer/MAC, serial backplanes, and more
- **Wireless Communications** - RF, base band, connectivity, transport and networking solutions for wireless equipment, addressing standards such as WCDMA, HSDPA, WiMAX and others.

CPLD

A Complex Programmable Logic Device (CPLD) is a combination of a fully programmable AND/OR array and a bank of microcells. The AND/OR array is reprogrammable and can perform a multitude of logic functions. Microcells are functional blocks that perform combinatorial or sequential logic, and also have the added flexibility for true or complement, along with varied feedback paths.

Traditionally, CPLDs have used analog sense amplifiers to boost the performance of their architectures. This performance boost came at the cost of very high current requirements.

Advantages of CPLD:

CPLDs perform a variety of useful functions in systems design due to their unique capabilities and as the market leader in programmable logic solutions, AMD provides a total solution to a designer's CPLD needs. Understanding the features and benefits of using CPLDs can help enable ease of design, lower development costs, and speed products to market.

Comparison of CPLD and FPGA

Sr. No	CPLD	FPGA
1	Few logic blocks (macrocells)	Large array of configurable logic blocks
2	Low (simple designs)	High (complex systems)
3	Faster (predictable timing)	Slower due to routing overhead
4	Cheaper for simple designs	Expensive, but powerful
5	Control logic, decoders, I/O expansion	DSP, AI, comms, prototyping ASICs

Conclusion: In this experiment, we learned the basic concepts of FPGA and CPLD devices and understood their role in digital system design. CPLDs are suitable for implementing small and simple control-oriented logic with non-volatile configuration, while FPGAs are better for large, complex, and high-performance applications due to their rich logic resources and reconfigurability. The experiment highlighted the importance of programmable logic devices as flexible alternatives to fixed hardware, making them essential tools for prototyping, testing, and modern VLSI design.

Suggested Reference:

https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html
https://www.xilinx.com/products/silicon-devices/cpld/cpld.html
https://media.digikey.com/pdf/data%20sheets/xilinx%20pdfs/xc4000,a,h.pdf

References used by the students:

Rubric wise marks obtained:

Rubrics	1	2	3	4	5	Total
Marks						

Experiment No: 2

Date: _____

Aim: Introduction to Hardware Description Languages (HDLs).

Competency and Practical Skills:

Relevant CO:

Objectives:

Equipment / Instruments: Laptop or Computer with Xilinx vivado .

Basic Theory:

What Are Hardware Description Languages (HDLs)?

HDLs are indeed similar to programming languages but not exactly the same. We utilize a programming language to build or create software, whereas we use a hardware description language to describe or express the behavioral characteristics of digital logic circuits.

We utilize HDLs for designing processors, motherboards, CPUs (i.e., computer chips), as well as various other digital circuitry.

What Is VHDL?

Very High-Speed Integrated Circuit Hardware Description Language (VHDL) is a description language used to describe hardware. It is utilized in electronic design automation to express mixed-signal and digital systems, such as ICs (integrated circuits) and FPGA (field-programmable gate arrays). We can also use VHDL as a general-purpose parallel programming language.

We utilize VHDL to write text models that describe or express logic circuits. If the text model is part of the logic design, the model is processed by a synthesis program. The next step in the process incorporates a simulation program to test the logic design. During this step, we utilize the simulation models to characterize the logic circuits that interface to the design. We refer to this collection of simulation models as a testbench.

Typically, a VHDL simulator is an event-driven simulator which means that we add each transaction to an event queue for a particular scheduled time. For example, if a signal assignment occurs after one nanosecond, we add the event to the queue as time + 1 ns. Although a zero delay is allowed, it still must be scheduled, and for these scenarios we utilize a Delta delay.

VHDL Functionality

These simulations alternate between two modes:

- Statement Execution: In this mode, the triggered statements are evaluated.
- Event Processing: During this mode, the events in the queue are processed.

Though there is an inherent similarity in hardware designs, VHDL has processes that can make the necessary accommodations. However, these processes differ in syntax from the parallel processes in tasks (Ada).

Similarly to Ada, VHDL is a predefined part of the programming language, plus, it is not case sensitive. However, VHDL provides various features that are unavailable in Ada, e.g., an extensive set of Boolean operators which include nor and nand. These additional features enable VHDL to precisely represent operations that are customary in hardware.

Another feature of VHDL is it has file output and input capabilities that you can utilize as a general-purpose language for text processing. Although, we typically see them in use by a simulation testbench for data verification or stimulus. Specific VHDL compilers build executable binaries, which afford the option to use VHDL to write a testbench for functionality verification designs utilizing files on the host computer to compare expected results, user interaction, and define stimuli.

Note: **Ada** is a statically typed, structured, object-oriented, and imperative high-level programming language; it is an extension that derives from Pascal and other programming languages.

What Is Verilog?

As I am sure you are aware, Verilog is also a Hardware Description Language. It employs a textual format to describe electronic systems and circuits. In the area of electronic design, we apply Verilog for verification via simulation for testability analysis, fault grading, logic synthesis, and timing analysis.

Verilog is also more compact since the language is more of an actual hardware modeling language. As a result, you typically write fewer lines of code, and it elicits a comparison to the C language. However, Verilog has a superior grasp on hardware modeling as well as a lower level of programming constructs. Verilog is not as wordy as VHDL, which accounts for its compact nature. Although VHDL and Verilog are similar, their differences tend to outweigh their similarities.

Verilog HDL is an IEEE standard (IEEE 1364). It received its first publication in 1995, with a subsequent revision in 2001. SystemVerilog, which is the 2005 revision of Verilog, is the latest publication of the standard. We call the IEEE Verilog standard document the LRM (Language Reference Manual). Currently, the IEEE 1364 standard defines the PLI (Programming Language Interface).

Note: The PLI is a collective of software routines that allows a bidirectional interface between other languages such as C and Verilog.

VHDL vs Verilog

Sr. No.	VHDL	Verilog
1	Strongly typed	Weakly typed
2	Easier to understand	Less code to write
3	More natural in use	More of a hardware modeling language
4	Non-C-like syntax	Similarities to the C language
5	Variables must be described by data type	A lower level of programming constructs
6	Widely used for FPGAs and military	A better grasp on hardware modeling
7	More difficult to learn	Simpler to learn

The most important thing to remember when you are writing HDL code is that you are describing real hardware, not writing a computer program. The most common beginner's mistake is to write HDL code without thinking about the hardware you intend to produce. If you don't know what hardware you are implying, you are almost certain not going to get what you want. Instead, begin by sketching a block diagram of your system, identifying which portions are combinational logic, which portions are sequential circuits or finite state machines, and so forth. Then, write HDL code for each portion, using the correct idioms to imply the kind of hardware you need.

Conclusion: In this experiment, we gained an understanding of Hardware Description Languages such as VHDL/Verilog and their importance in digital design. HDLs allow designers to describe hardware behavior and structure at various abstraction levels, making the design process faster, more flexible, and less error-prone compared to manual circuit implementation. Through this introduction, we observed how HDLs serve as a bridge between algorithmic concepts and physical hardware, enabling simulation, verification, and synthesis of complex digital systems effectively.

Suggested Reference:

https://resources.pcb.cadence.com/blog/2020-hardware-description-languages-vhdl-vs-verilog-and-their-functional-uses
https://www.sciencedirect.com/topics/computer-science/hardware-description-languages

References used by the students:

Rubric wise marks obtained:

Rubrics	1	2	3	4	5	Total
Marks						

Experiment No: 3

Date: _____

Aim: Introduction to Xilinx Tools.

Competency and Practical Skills:

Relevant CO:

Objectives:

Equipment / Instruments: Laptop or Computer with Xilinx Tools.

Basic:

The Xilinx Vivado Design Suite controls all aspects of the design flow. Through the Vivado IDE interface, one can access all of the design entry, simulation, and implementation tools. One can also access the source files, constraint files, and reports associated with their project.

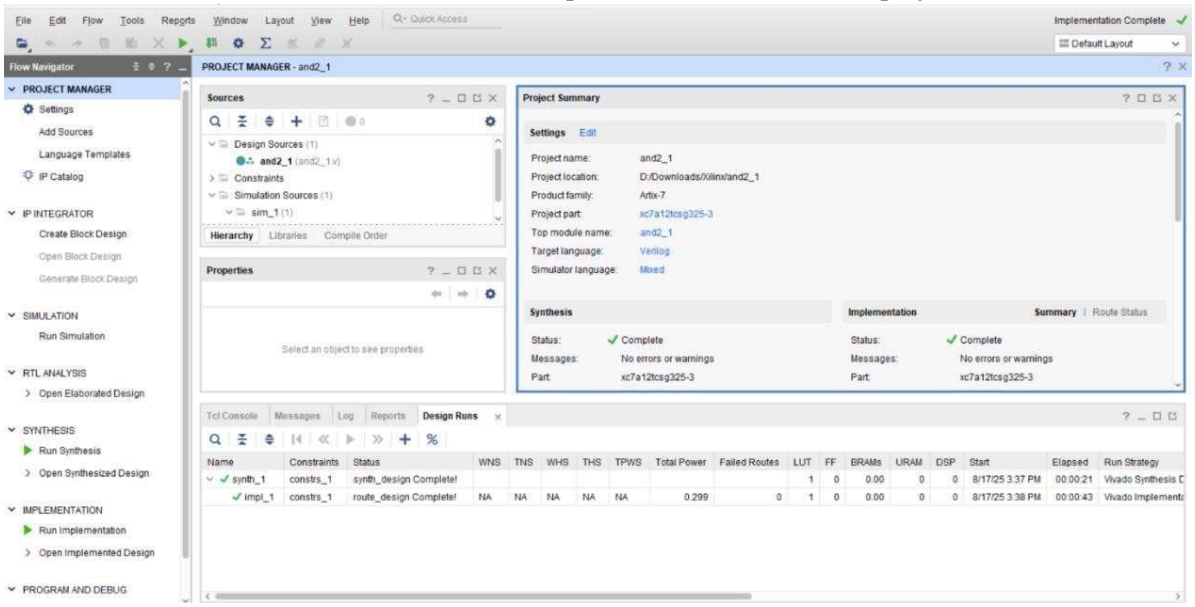


Fig. 3.1 Vivado Interface

<u>Vivado IDE Interface:</u>

- The Vivado IDE interface is divided into several main panels and windows, as seen in Figure 2.1.
- On the left side is the Flow Navigator, which provides access to all steps of the design flow such as Project Manager, IP Integrator, Simulation, Synthesis, Implementation, and Program and Debug. It also includes quick access to open projects, reference material, documentation, and tutorials.
- At the bottom of the Vivado IDE are the Tcl Console, Messages, and Log panels, which display status messages, errors, warnings, and process details.

- The central area is the Workspace, where design sources, reports, schematics, waveforms, and analysis tools are displayed.
- Each window in Vivado can be resized, docked, undocked, or arranged using the Window menu, and the default layout can be restored using Window > Reset Layout.
- The Sources window shows the hierarchy of the design, including Simulation and Implementation views. You can switch between views using the drop-down menu.
- The Hierarchy view displays the project name, target device, constraints, simulation sources, and design source files. Expandable nodes (+) indicate lower levels of hierarchy, and files can be opened for editing with a double-click.
- Each file in the Sources window has an associated icon that indicates its type (HDL file, block design, IP core, or constraints file, for example).
- The Tcl Console provides all standard output from processes run in Vivado. It displays messages, errors, and warnings, as well as executed Tcl commands.
- The Workspace hosts Vivado's built-in editors and viewers, such as the HDL Text Editor, IP Integrator Block Design Canvas, Constraints Editor, Report Viewer, RTL/Technology Schematic Viewer, and Timing Reports.
- Other tools such as the Vivado Logic Analyzer (ILA), Hardware Manager, Power Analyzer, and third-party editors may open in separate windows or tabs depending on their function.
- The Design Summary (Reports tab) provides an overview of key project data, including utilization summary, timing and performance results, constraints information, and direct links to detailed synthesis and implementation reports.

Tutorial:

Open vivado software and follow the steps as follows:

1. **We click on create project.**

Fig 3.1: create project

2. **Click on next.**

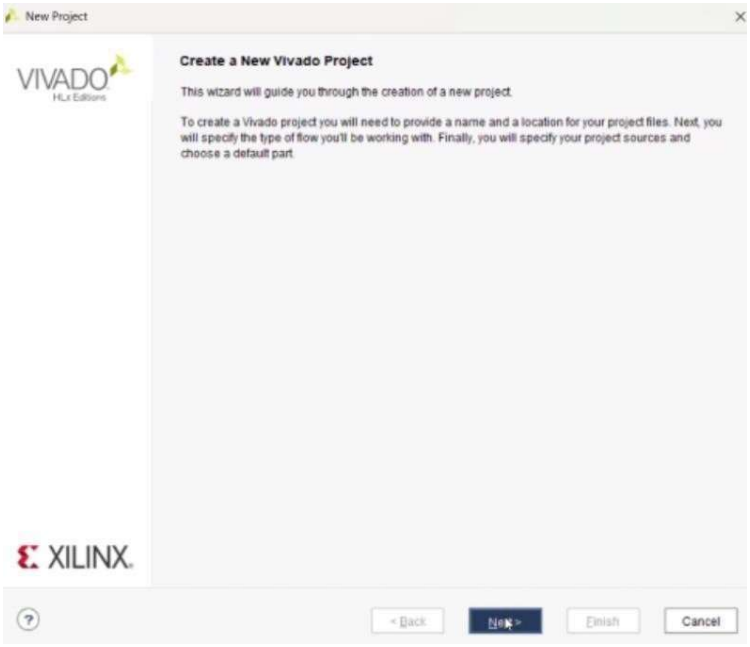


Fig 3.2 : click next

3. **We select the type of project as RTL project. Then click next.**

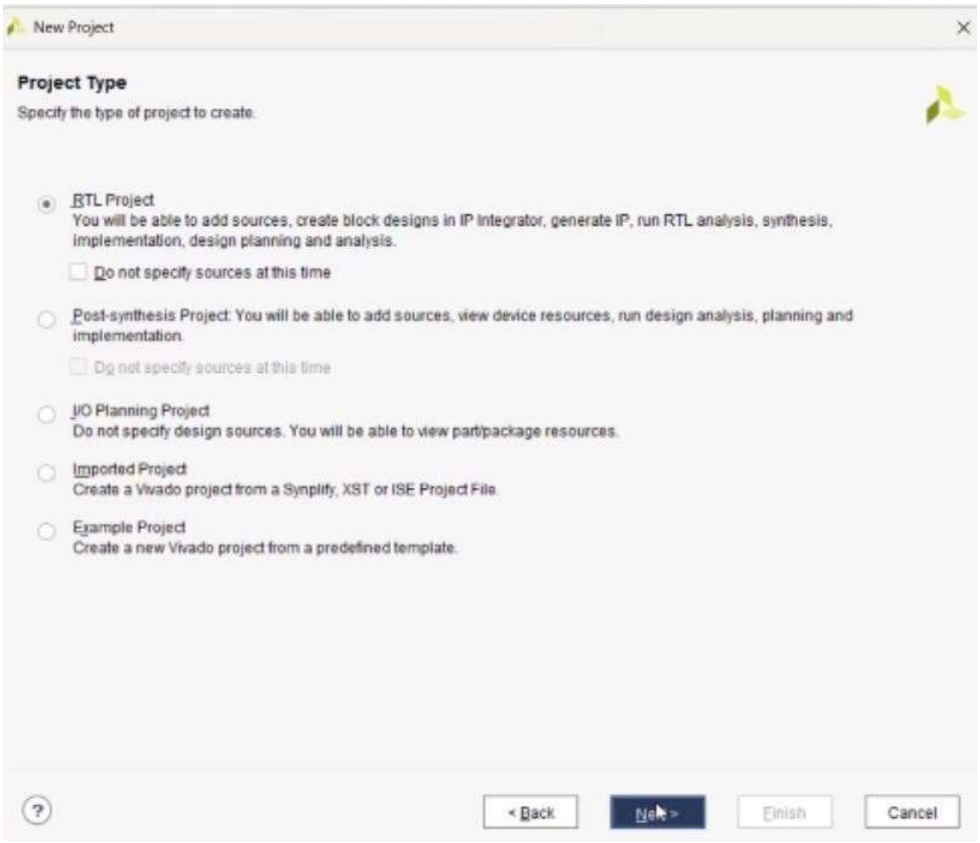


Fig 3.3 : select RTL project

4. **Now we add source files where we create a file for our module.**

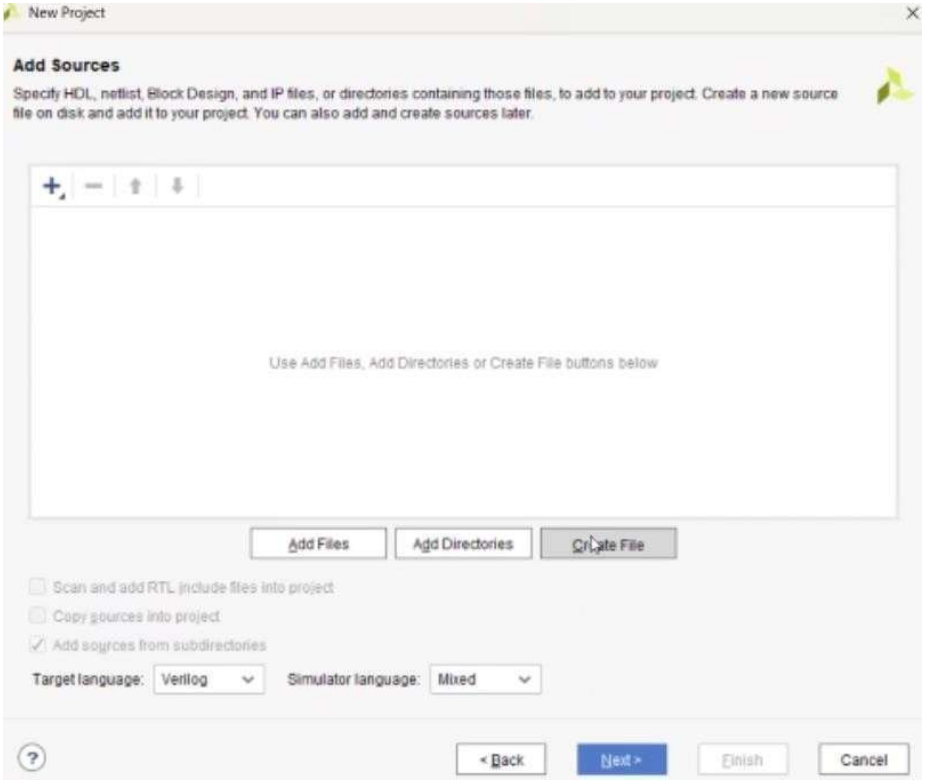


Fig 3.4 : create file

5. **We name the file as the name of our module. Then click next**

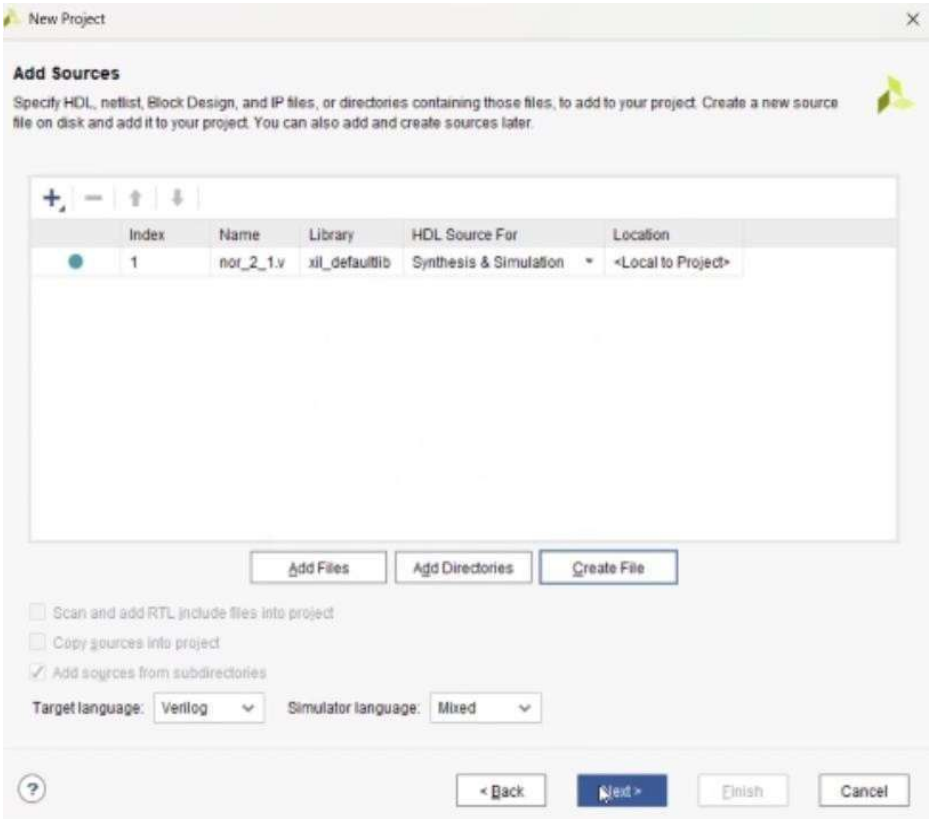


Fig 3.5 : select next

6. We select the family as Artix-7 and choose a 150 available IOBs option for simple operations.

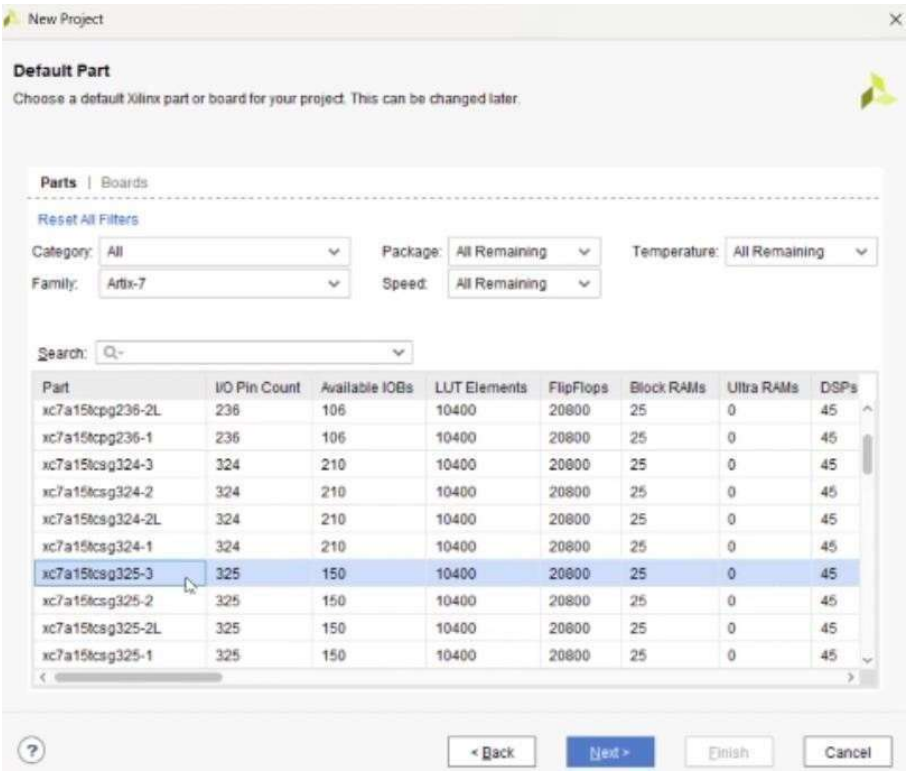


Fig 3.6 : select family

7. When we click next it gives a project summary and we can verify if our selections are proper or not.

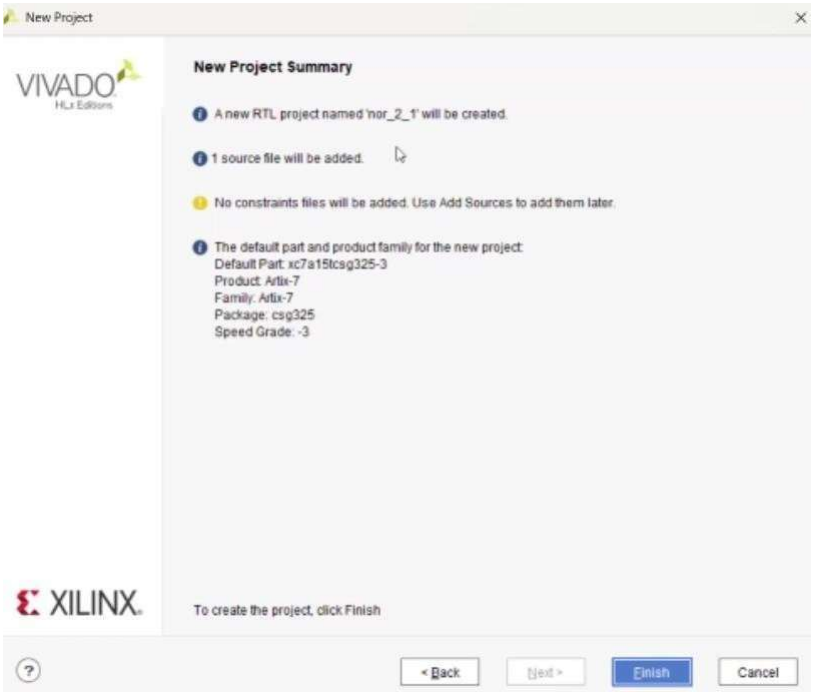


Fig 3.7 : click finish

8. Now we define the inputs and outputs of our module.

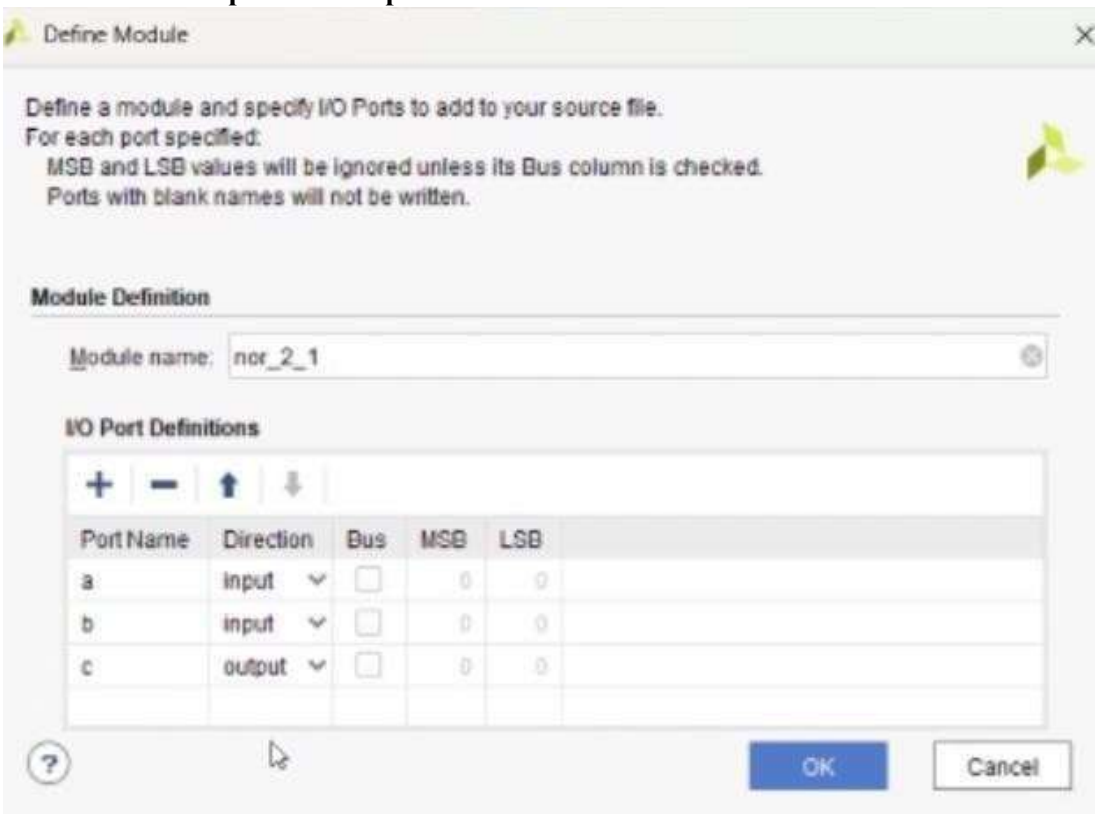


Fig 3.8 : select i/p and o/p

9. Once we do that, we gt this type of window where our module is created with the initial inputs and outputs set , now we write the logic for our module.

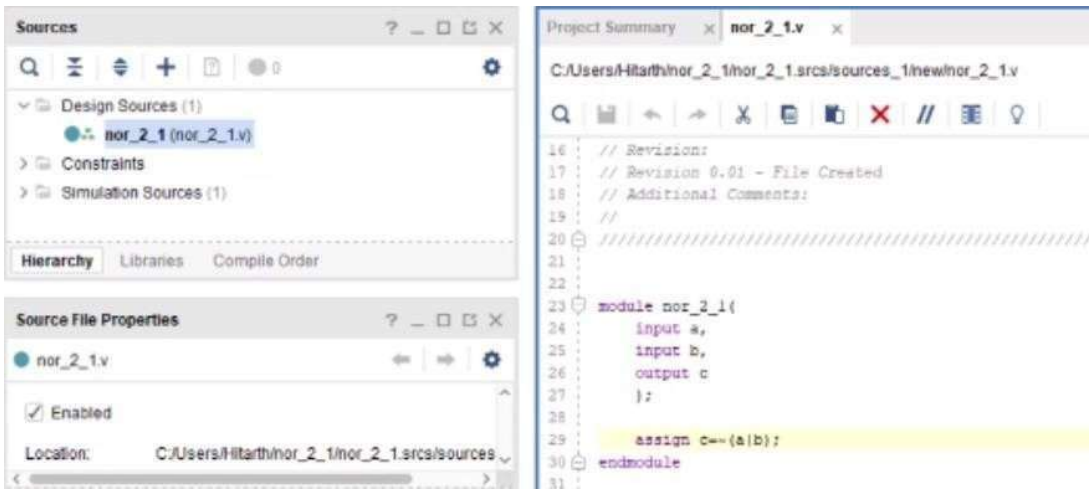


Fig 3.9 : enter code

10. In RTL option we click on schematic to obtain the design schematic.

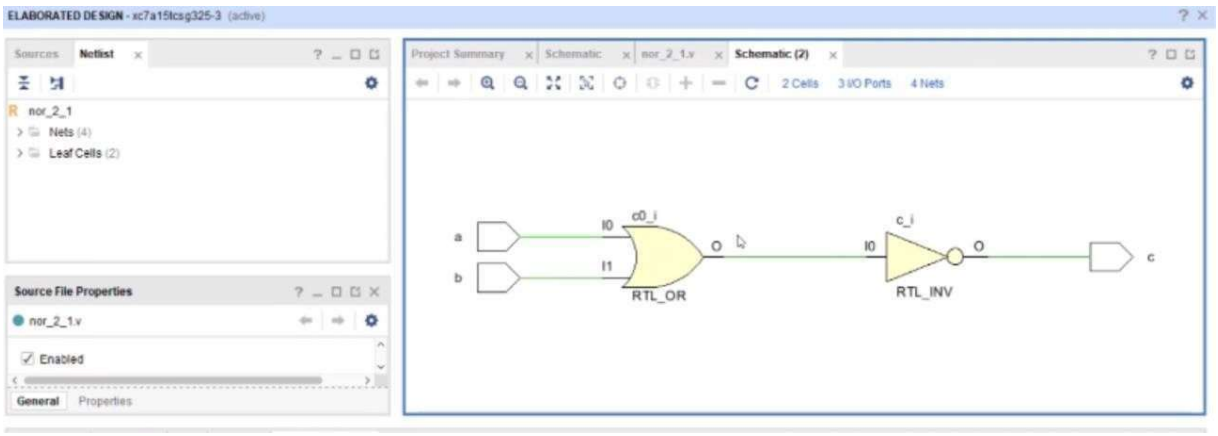


Fig 3.10 : create RTL schematic

11. Now we go to synthesis and do run synthesis in order to get an option csalled Schematic in Open Synthesised Design then we we click schematic to obtain as follows.

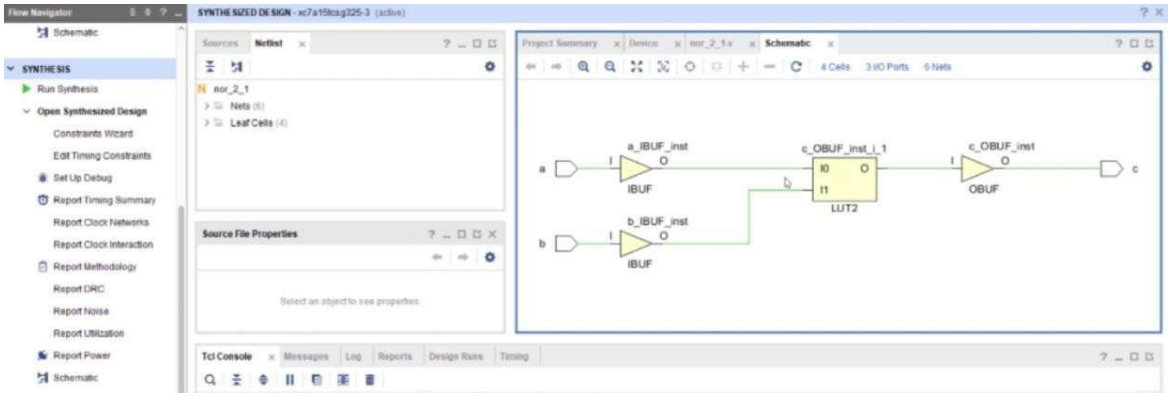


Fig 3.11 : create synthesis schematic

12. Now we add simulation file to our module i.e. testbench module as follows.

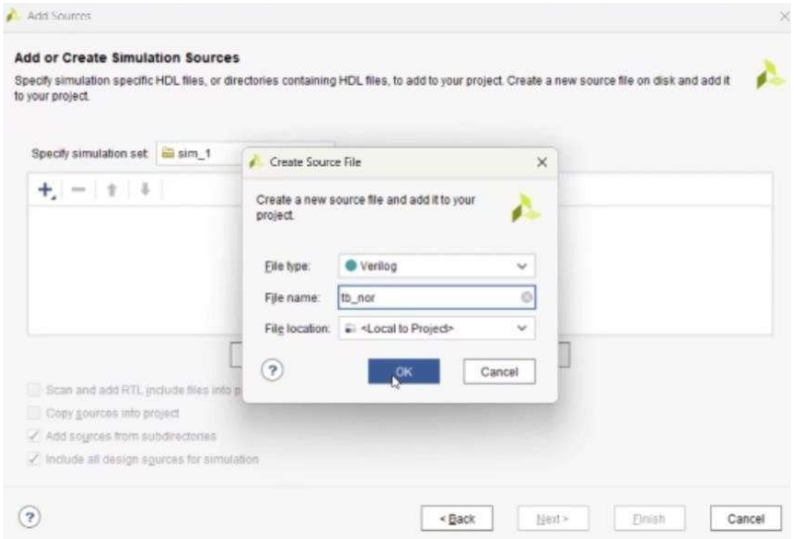


Fig 3.12 : create testbench

13. Now we write the testbench code as our simulation requirement.

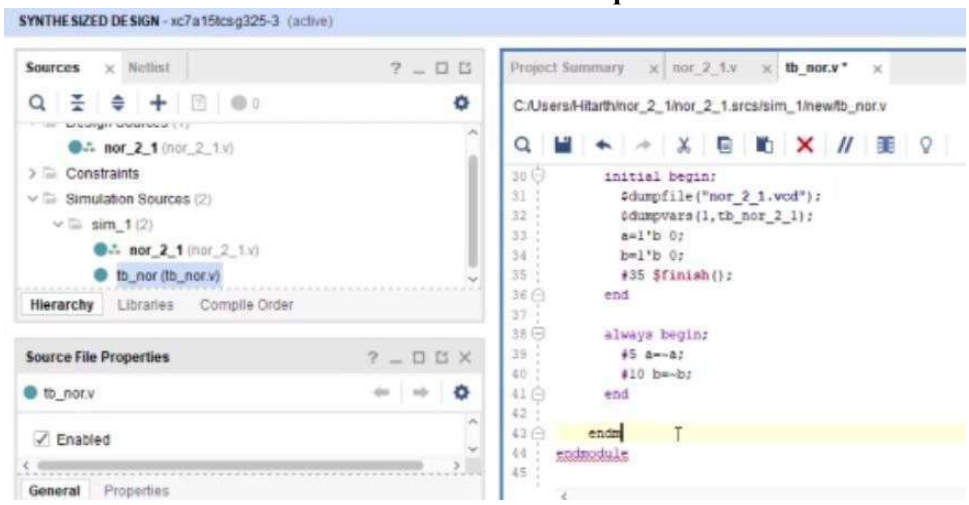


Fig 3.13 : Enter testbench code

14. Once it is done we go to simulation and click on run simulation , we wait for some time and obtain such simulation waveform.

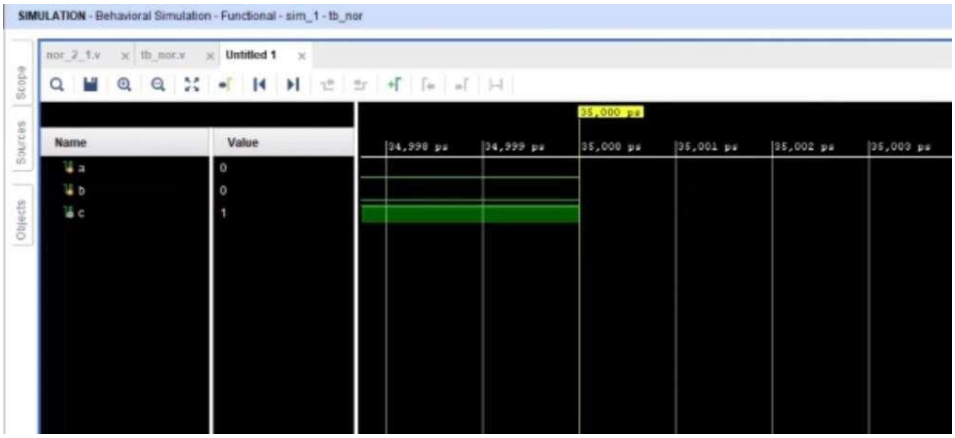


FIG 3.14 : Simulation

Conclusion: In this experiment, we learned to use Xilinx Vivado tools for digital circuit design and simulation. We studied the Project Navigator interface, its different panels, and how to manage source files, hierarchy, errors, and simulation results. By simulating a sample program, we understood the stepby-step design flow from code entry to implementation and verification. Thus, Xilinx tools provide an integrated environment for efficient design, synthesis, and simulation of digital systems.

Suggested Reference:

https://www.xilinx.com/htmldocs/xilinx13_3/ise_tutorial_ug695.pdf

References used by the students:

Rubric wise marks obtained:

Rubrics	1	2	3	4	5	Total
Marks						

Experiment No: 4

Date: _____

Aim: Implement all the basic Logic Gates and Boolean functions using different modeling styles in Verilog/ VHDL:

(a) Structural modeling
(b) Dataflow modeling
(c) Behavioural modeling

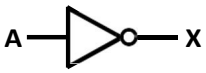
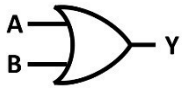
Competency and Practical Skills: Basic Digital Design

Relevant CO: CO5

Objectives: Understanding of different modeling styles in Verilog HDL.

Equipment / Instruments: Laptop or Computer with Xilinx / . **Basic Theory:**

Part – 1 Logic Gate Implementation

<table>
<tr><th>Logic Gate</th><th>Truth Table</th></tr>
<tr><td>Inverter
 
</td><td>
<table>
<tr><th>A</th><th>X</th></tr>
<tr><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td></tr>
</table>
</td></tr>
<tr><td>OR Gate
 
</td><td>
<table>
<tr><th>A</th><th>B</th><th>Y</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>
</td></tr>
<tr><td>AND Gate</td><td></td></tr>
<tr><td>NAND Gate</td><td></td></tr>
<tr><td>NOR Gate</td><td></td></tr>
</table>

XOR Gate	
XNOR Gate	

RTL Codes:

(a) Structural Modeling

```
module Not2_1(A,Y);
        input A;
        output Y;

        not a1(Y,A);
endmodule
```

(b) Dataflow Modeling

```
module Not2_1(A,Y);
        input A;
        output Y;

        assign Y=~A;
endmodule
```

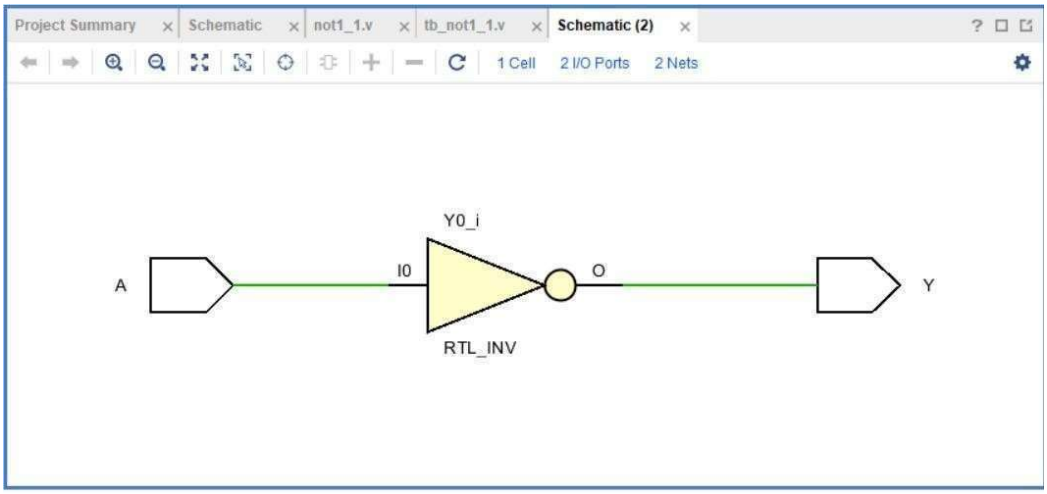
(c) Behavioural Modeling

```
module Not2_1(A,Y);
        input A;
        output Y;
        always @(A)
        Y=~A;

endmodule
```

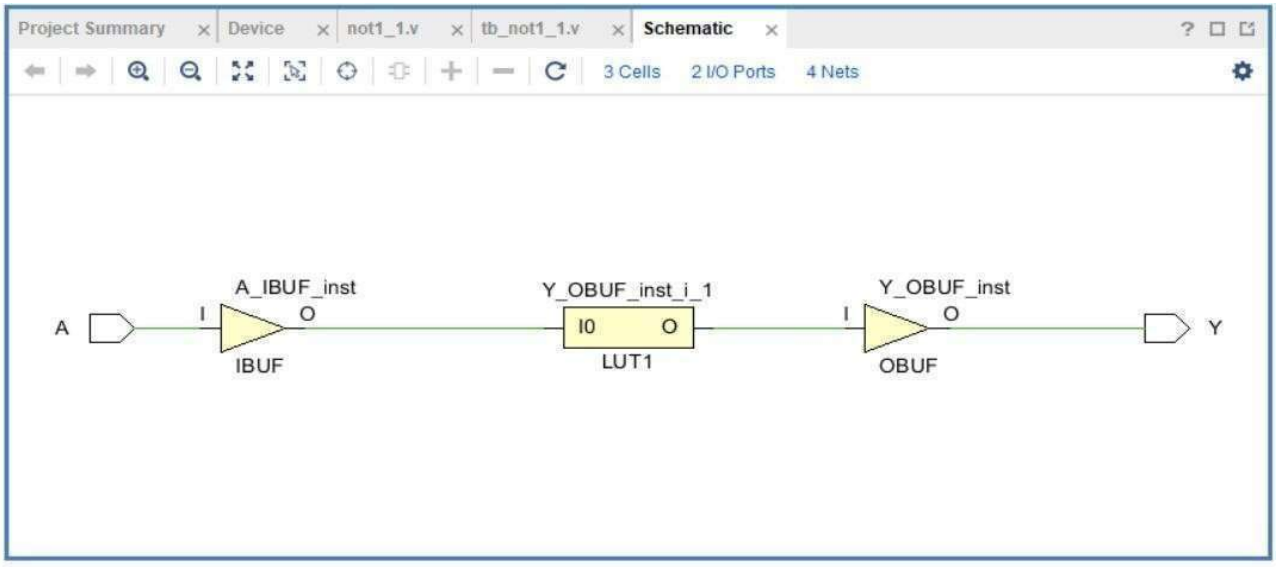
Test Bench:

```
module tb_Not2_1();
  reg A;
  wire Y;

  Not2_1 x1(A,Y);
  initial begin
    A=1'b0;
   // B=1'b0;
  #30 $finish();
  end

  always begin
    //#5 B=~B;
    #1 A=~A;
  end

  initial begin
    $dumpfile("Not2_1.vcd");
    $dumpvars();
  end
Endmodule
```

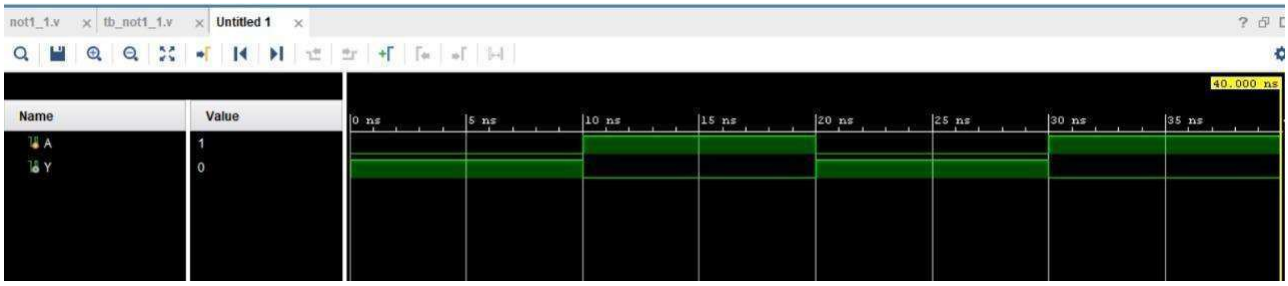
RTL Schematic:



Synthesis Schematic:



Simulation waveforms:



RTL Codes:

(a) Structural Modeling

```
module OR2_1(A.B.Y);
        input A,B;
        output Y;

      OR2_1 a1(Y,A,B);
endmodule
```

(b) Dataflow Modeling

```
module OR2_1(A,B,Y);
        input A,B;
        output Y;

      assign Y=A|B;

endmodule
```

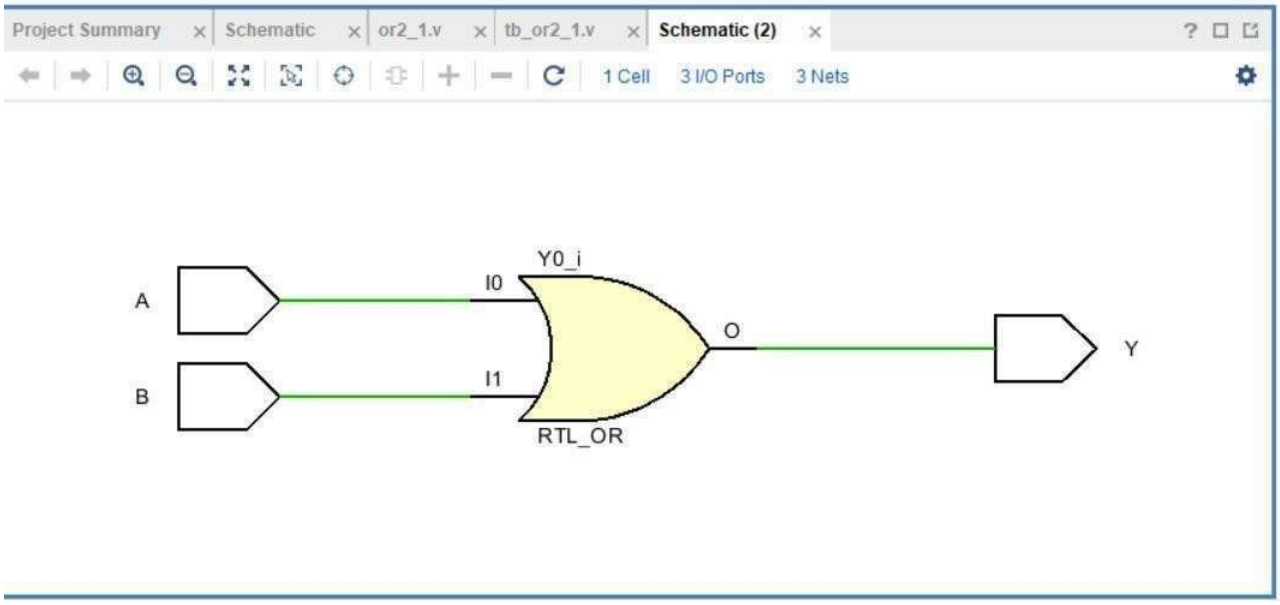
(c) Behavioural Modeling

```
module OR2_1(A,B,Y);
                input A,B;
                output Y;
                always @ (A,B)
                Y=A|B;

  endmodule
```

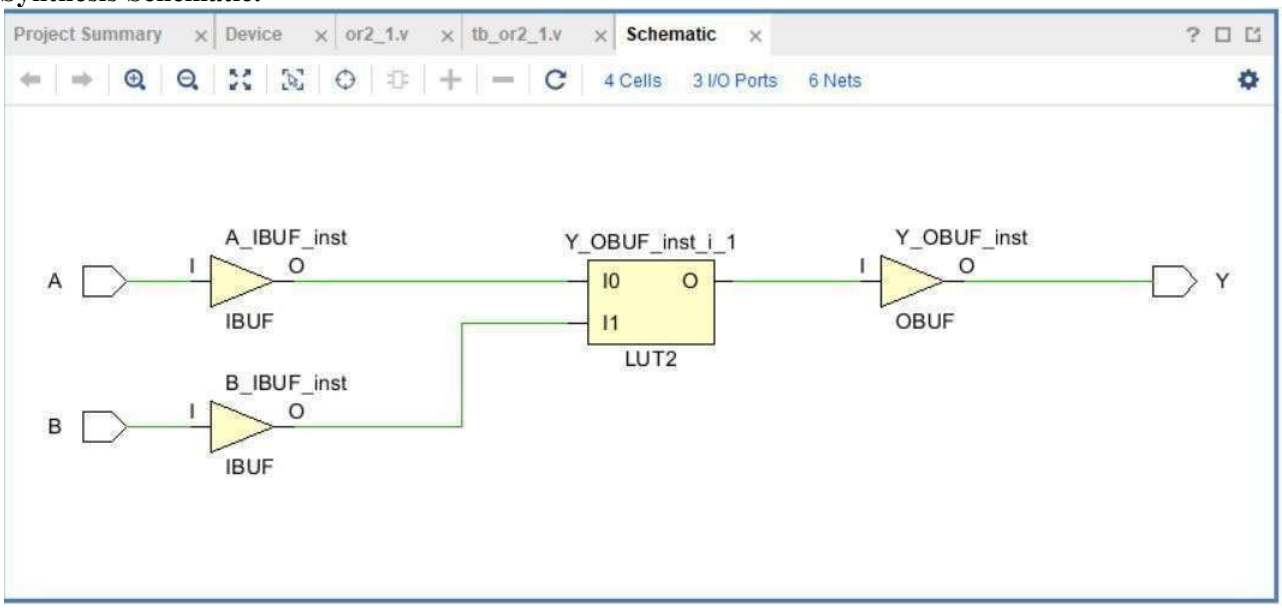
Test Bench:

```
module tb_OR2_1();
  reg A,B;
  wire Y;


OR2_1 x1(A,B,Y);
  initial begin
    a=1'b0;
    b=1'b0;
    #30 $finish();
  end
  always begin
    #5 b=~b;
    #10 a=~a;
  end
  initial begin
    $dumpfile("OR2_1.vcd");
    $dumpvars();
  end
endmodule
```

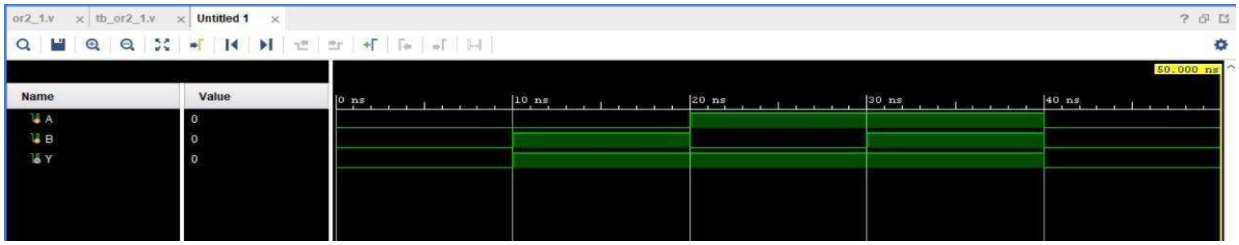
RTL Schematic:



Synthesis Schematic:



Simulation waveforms:



RTL Codes:

(a) Structural Modeling

```
module AND2_1(a,b,c);
        input a,b;
        output c;

      AND2_1 a1(c,a,b);
endmodule
```

(b) Dataflow Modeling

```
module AND2_1(a,b,c);
        input a,b;
       output c;
       assign c=a&b;
endmodule
```

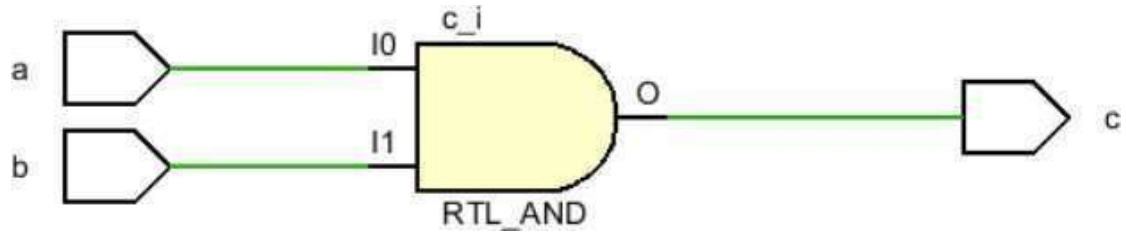
(c) Behavioural Modeling

```
module AND2_1(a,b,c);
        input a,b;
        output c;
        always @ (a,b)
        c= a&b;

endmodule
```

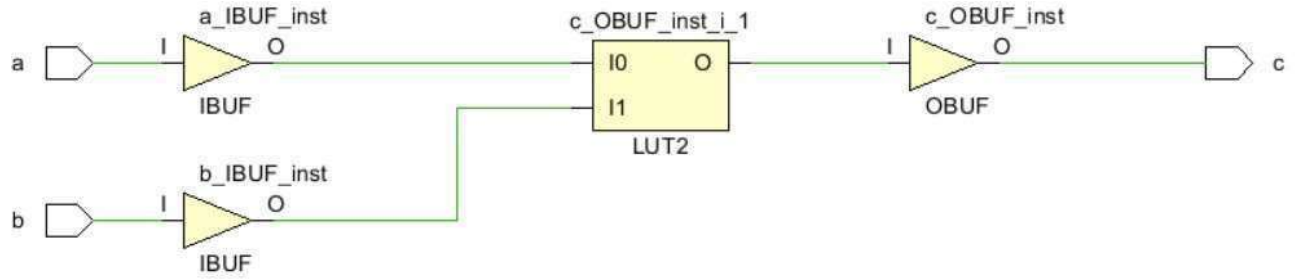
Test Bench:

```
module tb_AND2_1();
 reg a,b;
 wire c;

AND2_1 x1(a,b,c);
 initial begin
a=1'b0;
b=1'b0;
  #30 $finish();
 end
 always begin
  #5 b=~b;
  #10 a=~a;
 end
 initial begin
  $dumpfile("AND2_1.vcd");
  $dumpvars();
 end
endmodule
```

RTL Schematic:



Synthesis Schematic:



Simulation waveforms:

RTL Codes:

(a) Structural Modeling

```
module nand2_1 (A,B,Y);
        input A,B;
        output Y;

      nand2_1a1(Y,A,B);
endmodule
```

(b) Dataflow Modeling

```
module nand2_1 (a,b,c);
        input a,b;
        output c;
       assign c=~(a&b);
endmodule
```

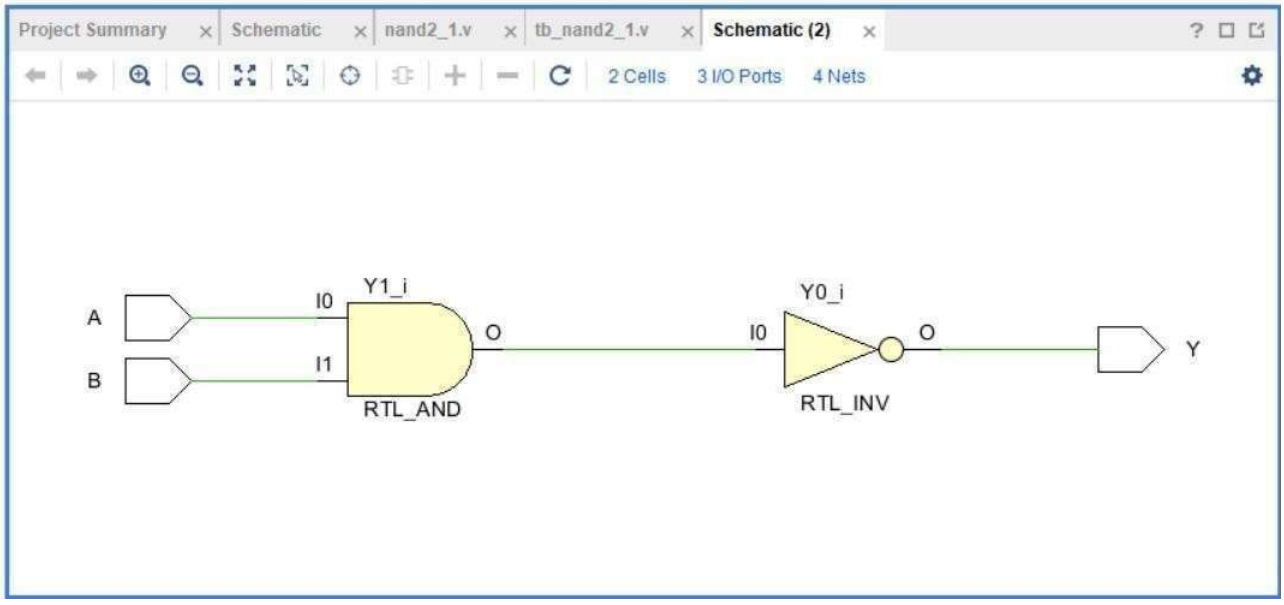
(c) Behavioural Modeling

```
module nand2_1 (A,B,Y);
        input A,B;
        output Y;
        always @ (A,B)
        Y= ~(A&B);

endmodule
```

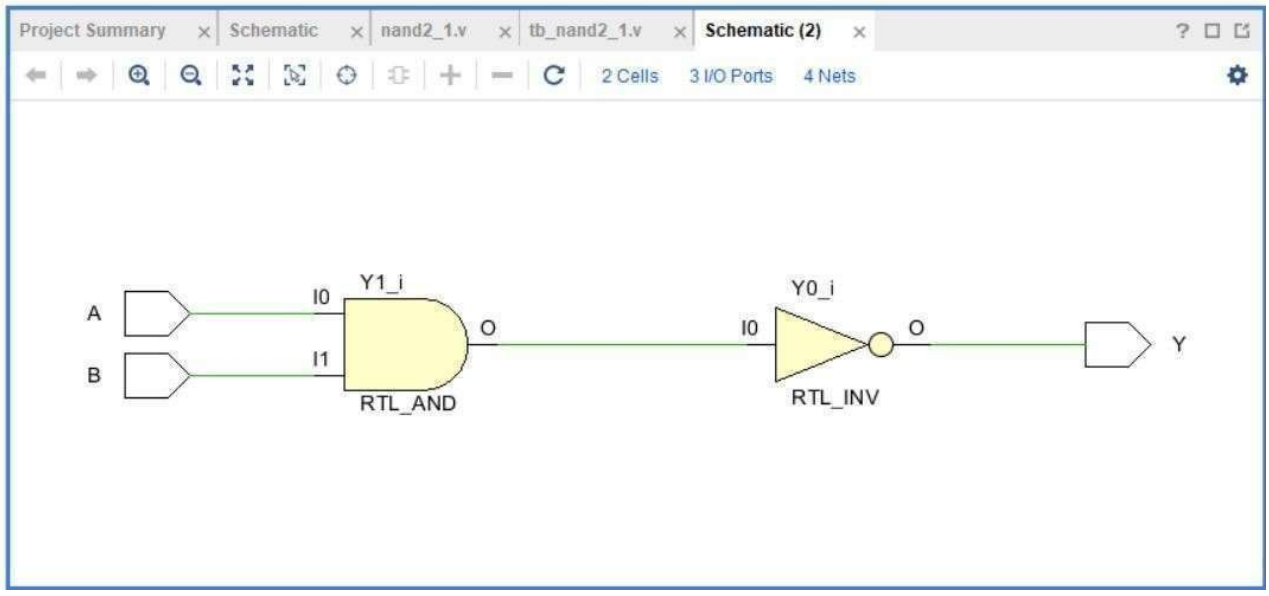
Test Bench:

```
module tb_ nand2_1 ();
 reg A,B;
 wire Y;

nand2_1x1(A,B,Y);
 initial begin
 A=1'b0;
 B=1'b0;
  #30 $finish();
 end
 always begin
  #5 B=~B;
  #10 A=~A;
 end
 initial begin
  $dumpfile("nand2_1.vcd");
  $dumpvars();
 end
endmodule
```

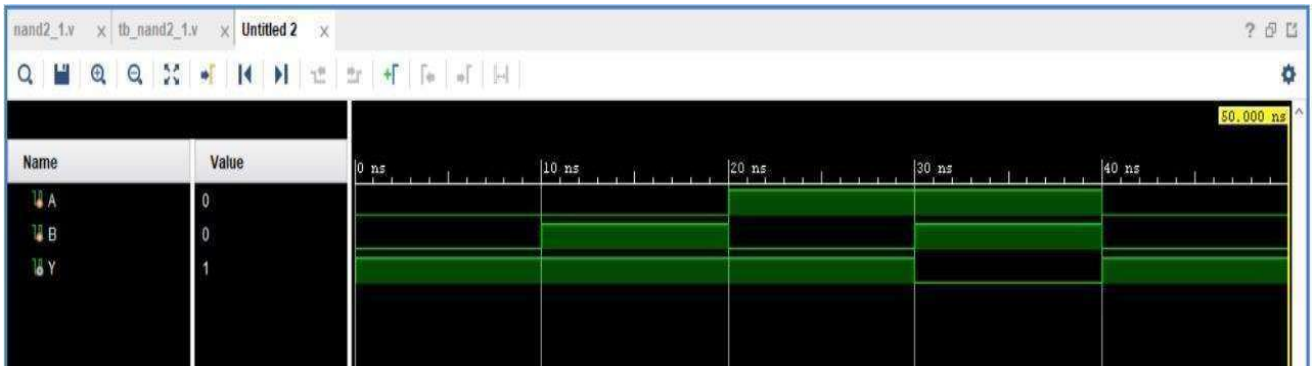
RTL Schematic:



Synthesis Schematic



Simulation waveforms:



RTL Codes:

(a) Structural Modeling

```
module nor2_1 (A,B,Y);
        input A,B;
       output Y;
       nor2_1 (A,B,Y);
endmodule
```

(b) Dataflow Modeling

```
module nor2_1 (A,B,Y);
        input A,B;
        output Y;

       assign Y=A^B
endmodule
```

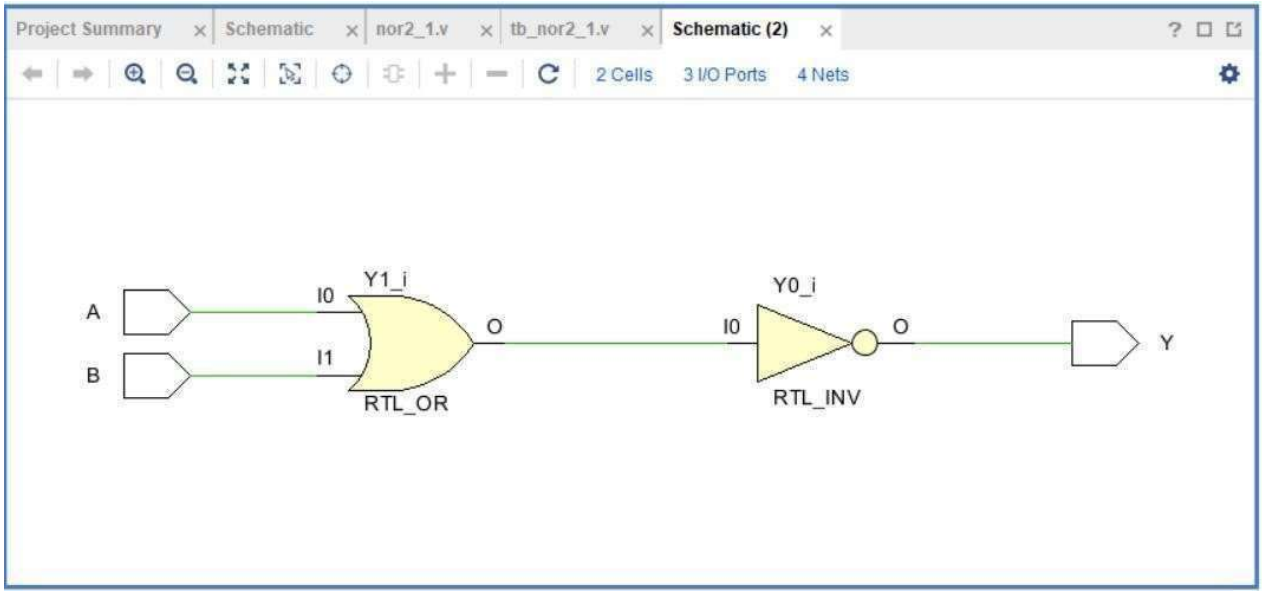
(c) Behavioural Modeling

```
module nor2_1 (A,B,Y);
        input A,B;
        output Y;
        always @ (A,B)
        Y= A^B;

endmodule
```

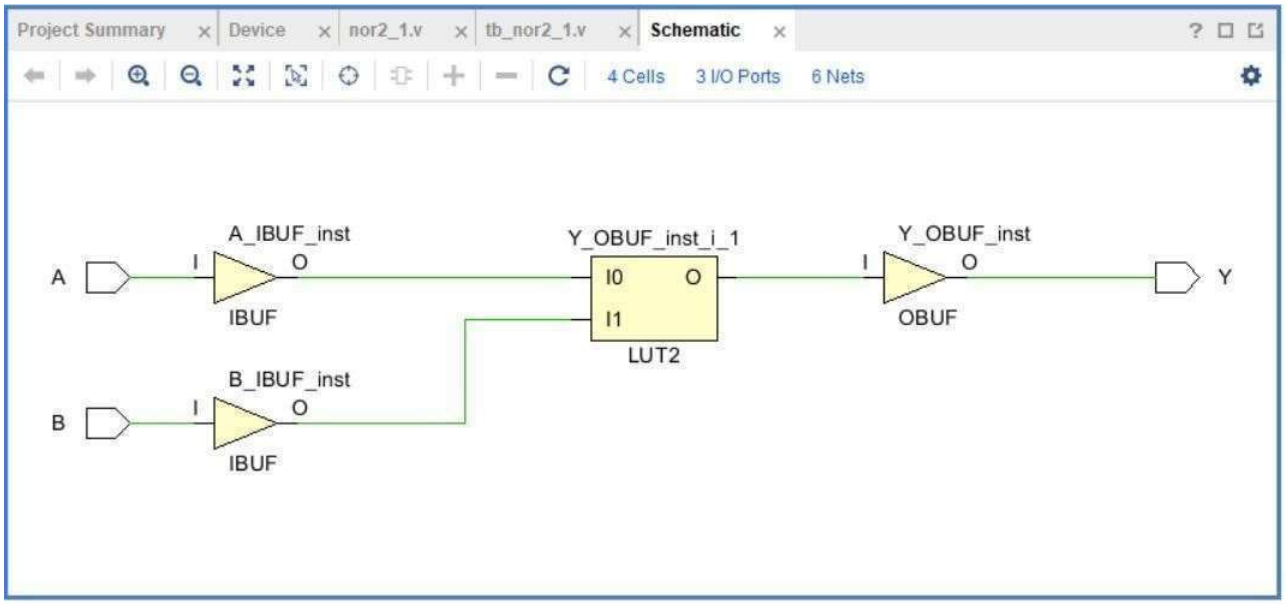
Test Bench:

```
module tb_ nor2_1 ();
 reg A,B;
 wire Y;
 nor2_1 x1(A,B,Y);
 initial begin
 A=1'B0;
  B=1'B0;
  #30 $finish();
 end
 always begin
  #5 B=~B;
  #10 A=~A;
 end
 initial begin
  $dumpfile("nor2_1.vcd");
  $dumpvars();
 end
endmodule
```

RTL Schematic:



Synthesis Schematic:



Simulation waveforms:

RTL Codes:

(a) Structural Modeling

```
module xor2_1 (A,B,Y);
        input A,B;
       output Y;
       xor2_1 (A,B,Y);
endmodule
```

(b) Dataflow Modeling

```
module xor2_1 (A,B,Y);
        input A,B;
        output Y;

       assign C=A^B
endmodule
```

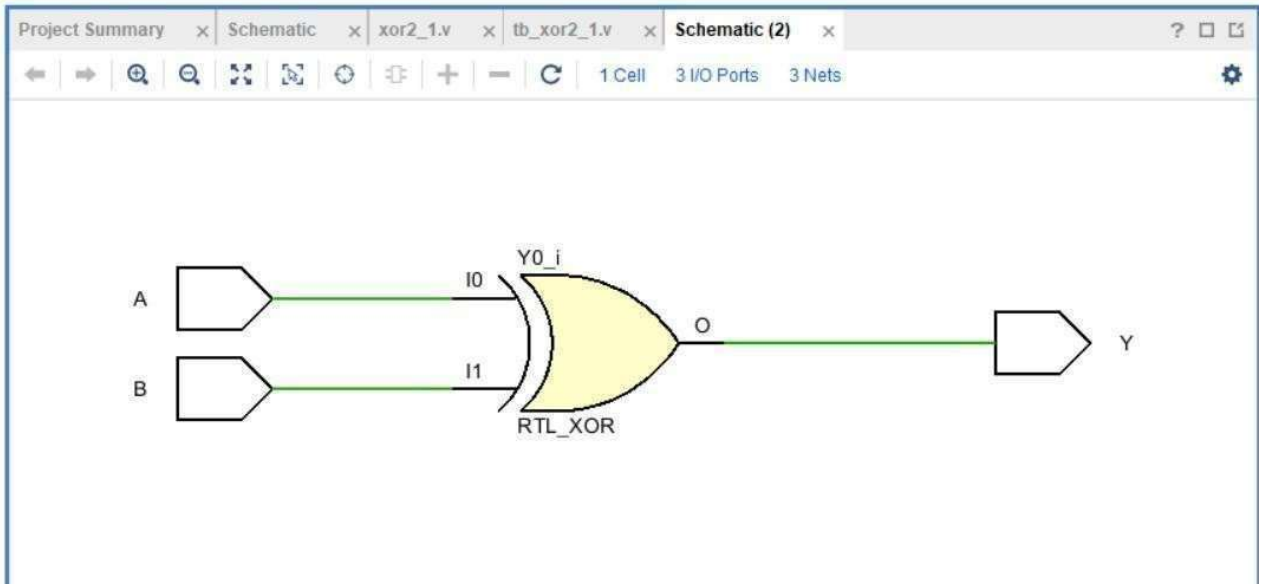
Behavioural Modeling

```
module xor2_1 (A,B,Y);
        input A,B;
        output Y;
        always @ (A,B)
        Y= A^B;

endmodule
```

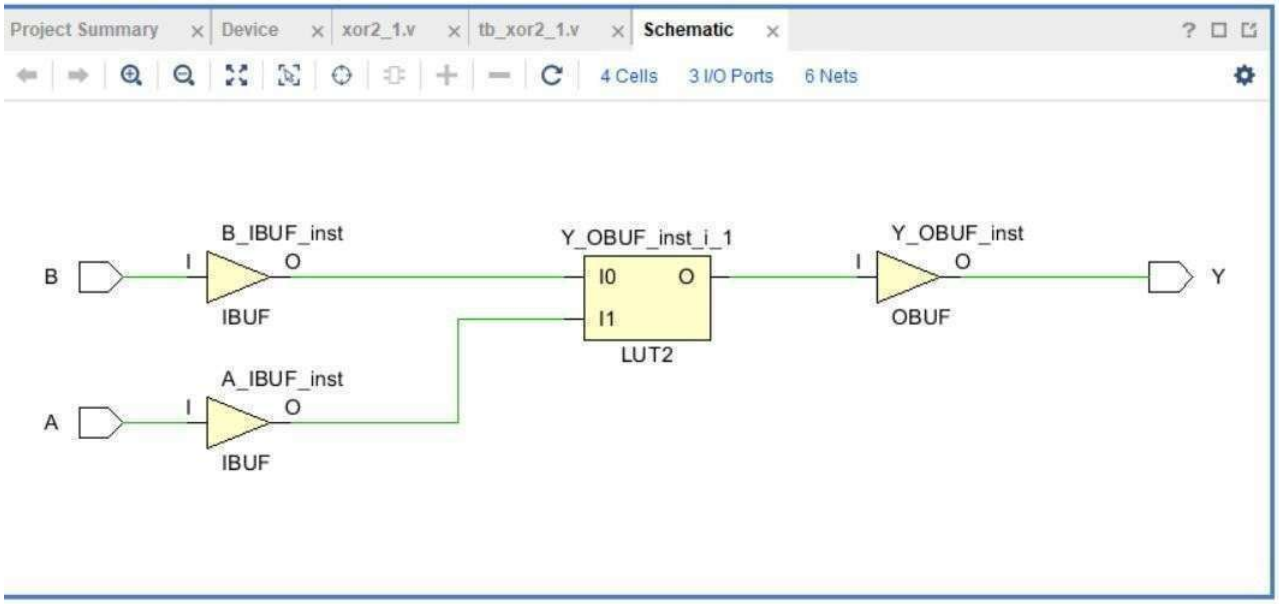
Test Bench:

```
module tb_ xor2_1 ();
 reg A,B;
 wire Y;
xor2_1 x1(A,B,Y);
 initial begin
 A=1'B0;
  B=1'B0;
  #30 $finish();
 end
 always begin
  #5 B=~B;
  #10 A=~A;
 end
 initial begin
  $dumpfile("xor2_1.vcd");
  $dumpvars();
 end
endmodule
```

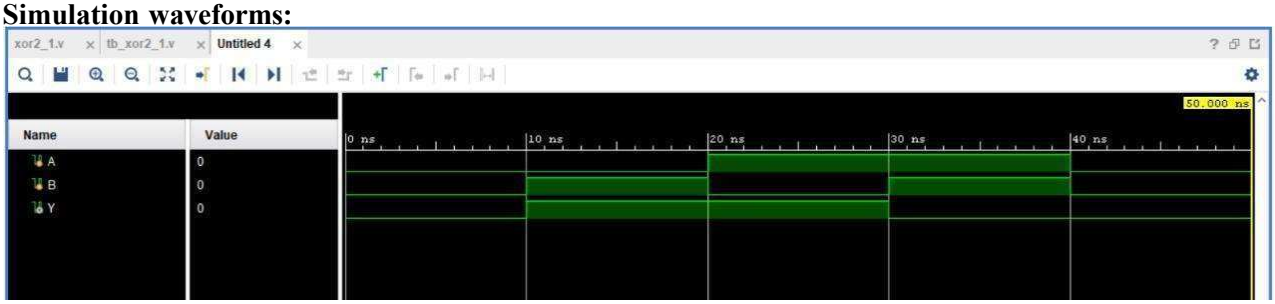
RTL Schematic:



Synthesis Schematic:



Simulation waveforms:



RTL Codes:

(a) Structural Modeling

```
module xnor2_1 (a,b,c);
        input a,b;
        output c;

        xnor2_1 a1(c,a,b);
endmodule
```

(b) Dataflow Modeling

```
module xnor2_1 (a,b,c);
        input a,b;
        output c;
       assign c=~(a^b);
endmodule
```

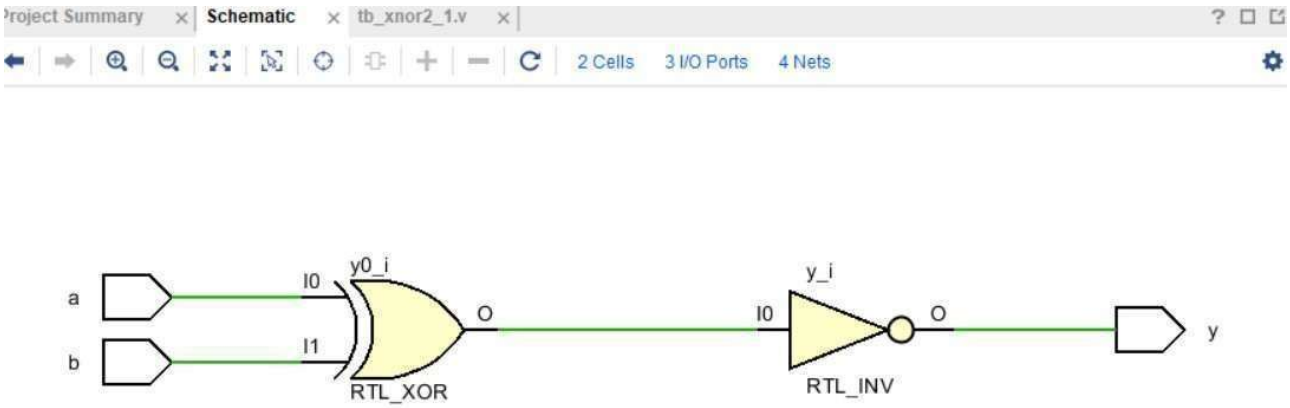
(c) Behavioural Modeling

```
module xnor2_1 (a,b,c);
        input a,b;
        output c;
        always @ (a,b)
        c= ~(a^b);

endmodule
```

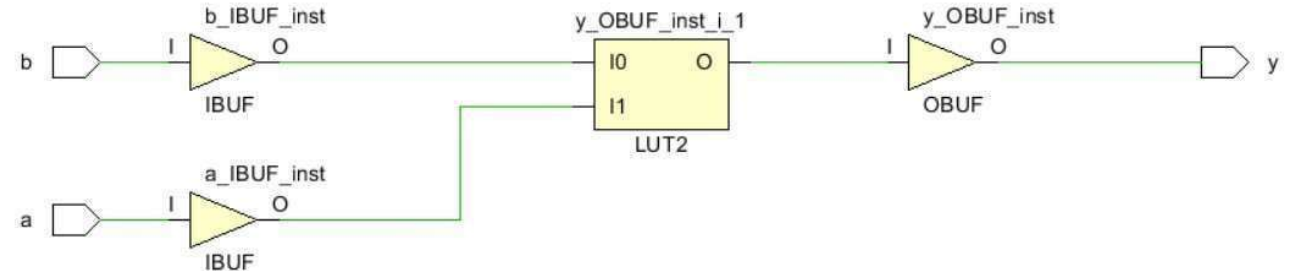
Test Bench:

```
module tb_ xnor2_1 ();
 reg a,b;
 wire c;

xnor2_1 x1(a,b,c);
 initial begin
 a=1'b0;
  b=1'b0;
  #30 $finish();
 end
 always begin
  #5 b=~b;
  #10 a=~a;
 end
 initial begin
  $dumpfile("xnor2_1.vcd");
  $dumpvars();
 end
endmodule
```

RTL Schematic:



Synthesis Schematic



Simulation waveforms:

Part – 2 Boolean Function Implementation

1. $f = x'y' + xy$
2. $g = x'yz + xyz' + x'y'z' + xyz$

RTL Code:

(a) Structural Modeling

```
module boolean_fun(x, y, z, f, g);
        input x, y, z;
        output f, g;
        wire a,b,c,d,e;                 //internal signal definitions
        wire p,q,r,s;                   //internal signal definitions

        not (a, x);                     // a = x'
        not (b, y);                     // b = y'
        not (c, z);                     // c = z'

        and (d, a, b);                  // d = x'y'
        and (e, x, y);                  // e = xy
        or (f, d, e);                   // f = x'y' + xy

        and (p, a, y, z);               // p = x'yz
        and (q, x, y, c);               // q = xyz'
        and (r, a, b, c);               // r = x'y'z'
        and (s, x, y, z);               // s = xyz
        or (g,p,q,r,s);                 //g = x'yz + xyz' + x'y'z' + xyz

    endmodule
```

(b) Dataflow Modeling

```
module boolean_fun(x, y, z, f, g);
    input x, y, z;
    output f, g;

    assign f = (~x & ~y) | (x & y);          //  f = x'y' + xy
    assign g = (~x & y & z) | (x & y & ~z) | (~x & ~y & ~z) | (x & y & z);
                                             //g = x'yz + xyz' + x'y'z' + xyz
endmodule
```

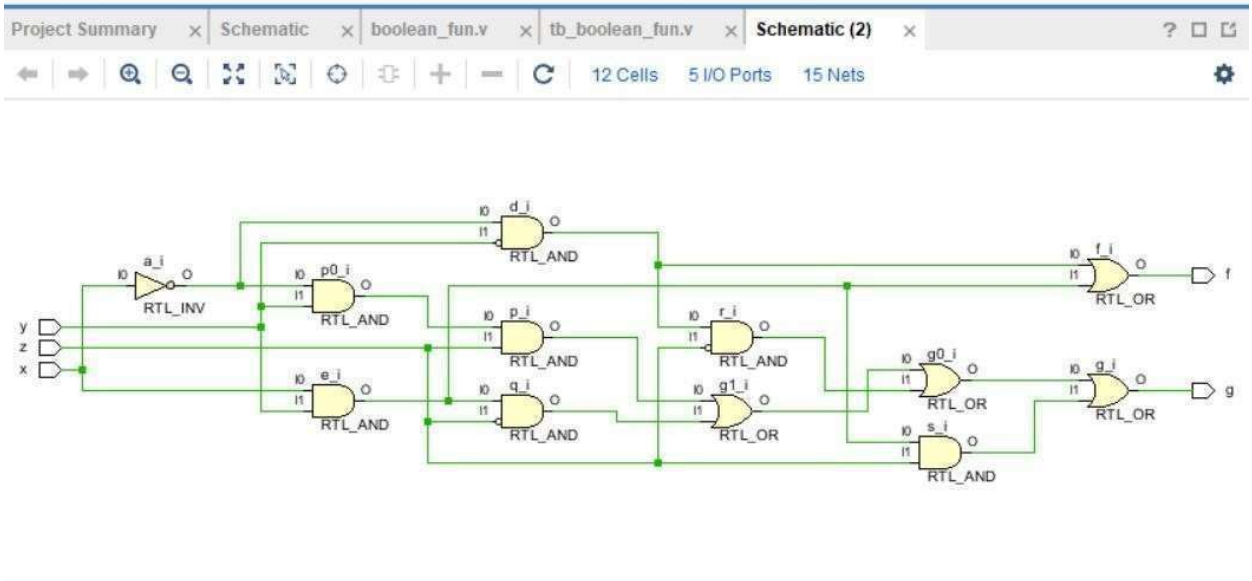
(c) Behavioural Modeling

```
module boolean_fun(x, y, z, f, g);
        input x, y, z;
        output f, g;

      always @(x, y, z)
        begin
          f = (~x & ~y) | (x & y);          //  f = x'y' + xy
          g = (~x & y & z) | (x & y & ~z) | (~x & ~y & ~z) | (x & y & z);
                                            //g = x'yz + xyz' + x'y'z' + xyz
        end
    endmodule
```

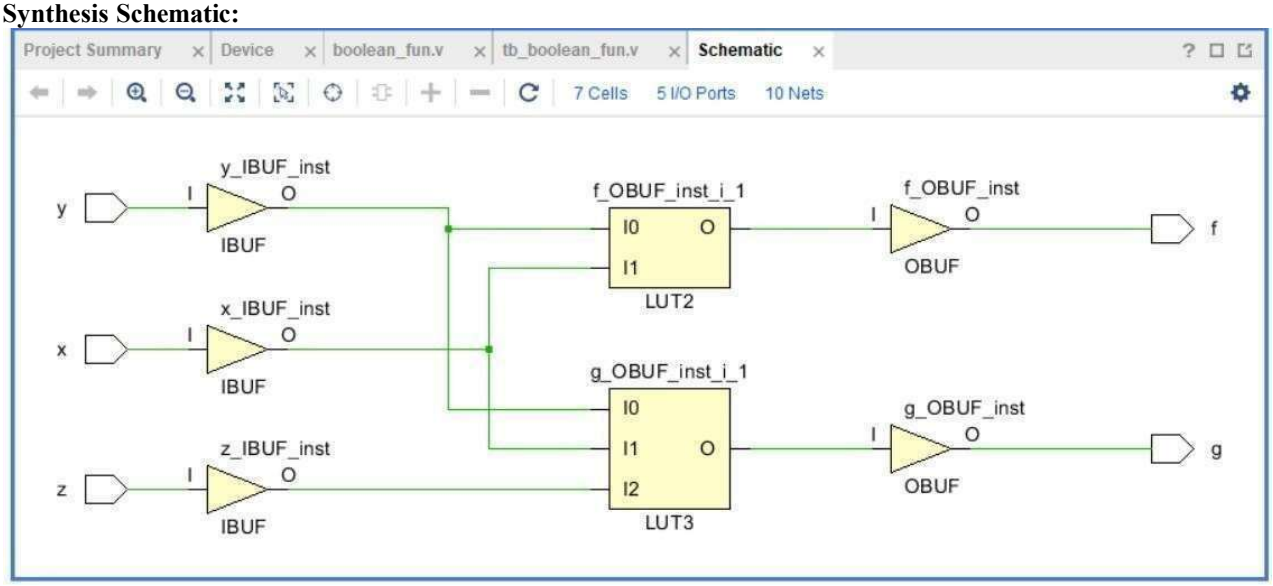
Testbench:

```
module tb_boolean_fun;
    // Inputs are reg (we drive them)
    reg x, y, z;

    // Outputs are wire (driven by DUT)
    wire f, g;

    // Instantiate DUT
    boolean_fun uut (
        .x(x),
        .y(y),
        .z(z),
        .f(f),
        .g(g)
    );

    initial begin
        // Initialize inputs
        x = 0; y = 0; z = 0;

        // Apply test patterns
        #10 x = 0; y = 0; z = 1;
        #10 x = 0; y = 1; z = 0;
        #10 x = 0; y = 1; z = 1;
        #10 x = 1; y = 0; z = 0;
        #10 x = 1; y = 0; z = 1;
        #10 x = 1; y = 1; z = 0;
        #10 x = 1; y = 1; z = 1;

        #10 $finish;  // End simulation
    end
endmodule
```

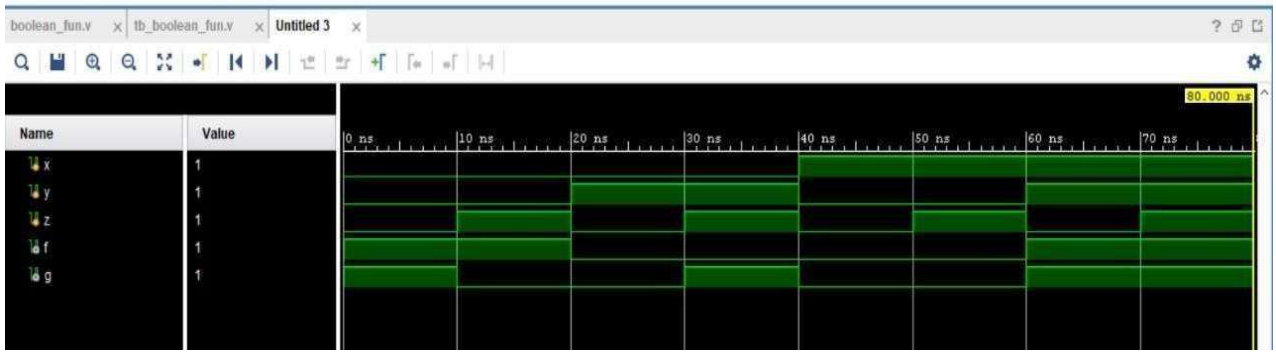
RTL Schematic:



Synthesis Schematic:



Simulation waveforms:



Conclusion: The basic logic gates and Boolean functions were successfully implemented in Verilog using structural, dataflow, and behavioral modeling. The simulation results matched the expected truth tables, helping to understand different modeling styles and their applications in digital design.

Suggested Reference: ---

References used by the students:

Rubric wise marks obtained:

Rubrics	**1**	**2**	**3**	**4**	**5**	**Total**
Marks						

Experiment No: 5

Date: _____

Aim: Design Adder & Subtractor using Verilog / VHDL.

(a) Half Adder (Using structural and dataflow modeling)
(b) Full Adder using Half Adder (Structure Method).
(c) Full Adder (Using dataflow and behavioural modeling)
(d) Half Subtractor
(e) Full Subtractor

Competency and Practical Skills: Basic Digital Design

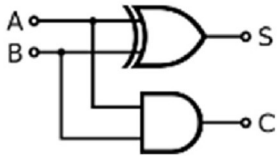
Relevant CO: CO5

Objectives: Designing of adders and subtractor using different modeling styles in Verilog HDL.

Equipment / Instruments: Laptop or Computer with Xilinx.

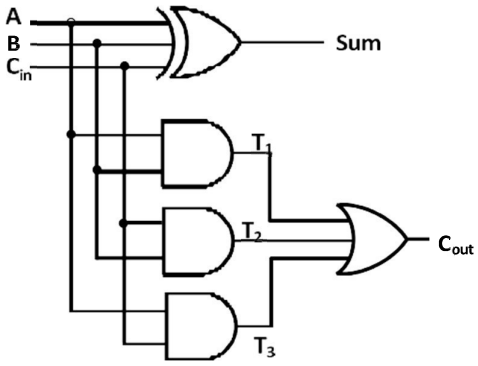
Basic Theory:

Half Adder



A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Full Adder



A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Half Subtractor

A	B	D	Br
0	0		
0	1		
1	0		
1	1		

Full Subtractor

A	B	C_{in}	Diff	B_{out}
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

RTL Code:

(a) Half Adder (Structural modeling): $S = A \oplus B$
$C = AB$

```
module HA (A, B, S, C);              // HA - Half Adder
        input A, B;
        output S, C;                 // S- Sum , C - Carry
        xor x1(S, A, B);             // sum = A xor B
        and n1(C, A, B);             // carry = A and B
endmodule
```

Half Adder (Dataflow modeling):

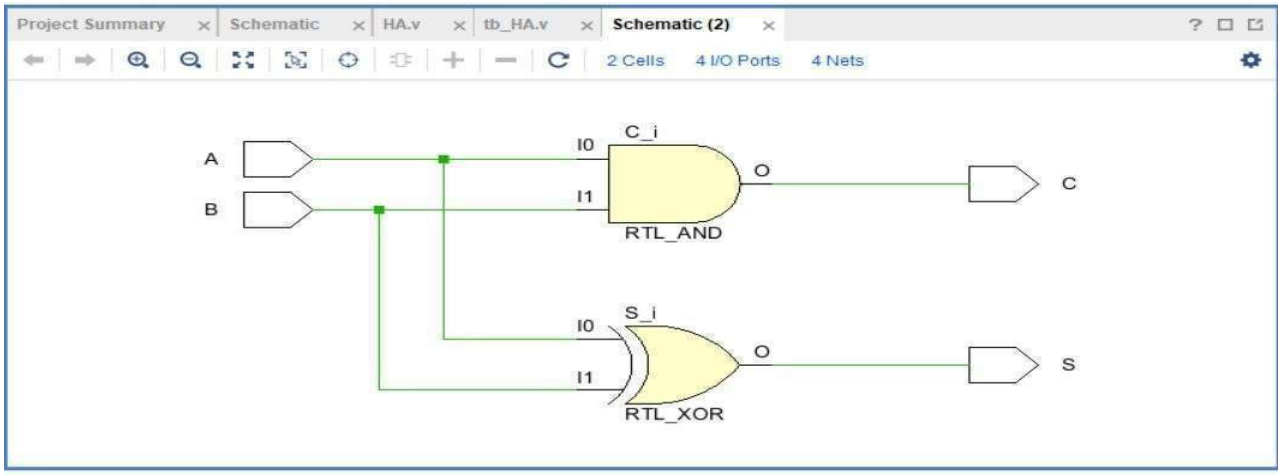
```
module HA (A, B, S, C);              // HA - Half Adder
        input A, B;
        output S, C;                 // S- Sum , C - Carry
        assign  S = A ^ B;           // sum = A xor B
        assign  C =A & B;            // carry = A and B
endmodule
```

Testbench:

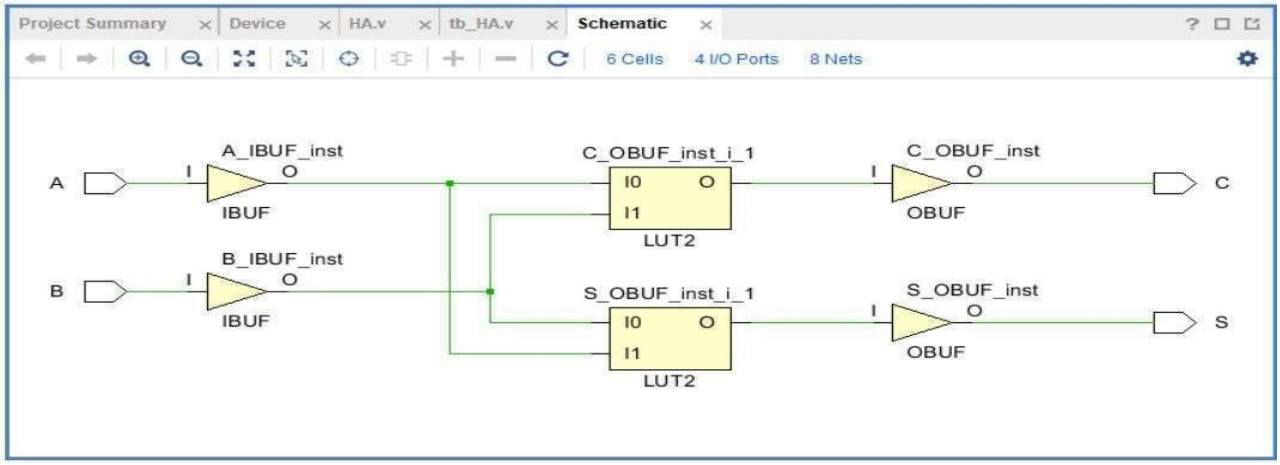
```
`timescale 1ns/1ps   // Simulation time unit precision

module tb_HA;        // Testbench name
reg A, B;        // Testbench inputs
wire S, C;       // Testbench outputs

// Instantiate the Half Adder (UUT = Unit UnderTest)
HA uut (
.A(A),
.B(B),
.S(S),
.C(C)
);

// Apply stimulus
initial begin
$monitor("Time=%0t | A=%b B=%bSum=%bCarry=%b", $time, A, B, S, C);

// Test cases
A = 0; B = 0; #10;
A = 0; B = 1; #10;
A = 1; B = 0; #10;
```

```
A = 1; B = 1; #10;

$stop;  // End simulation
end
endmodule
```

RTL Schematic:

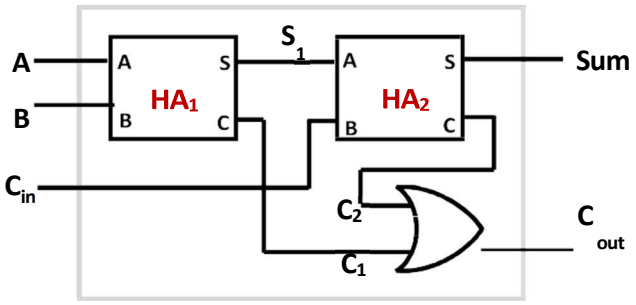


Synthesis Schematic:



Simulation waveforms:

(b) 1-bit Full Adder Implementation using Half Adders (structural modeling):



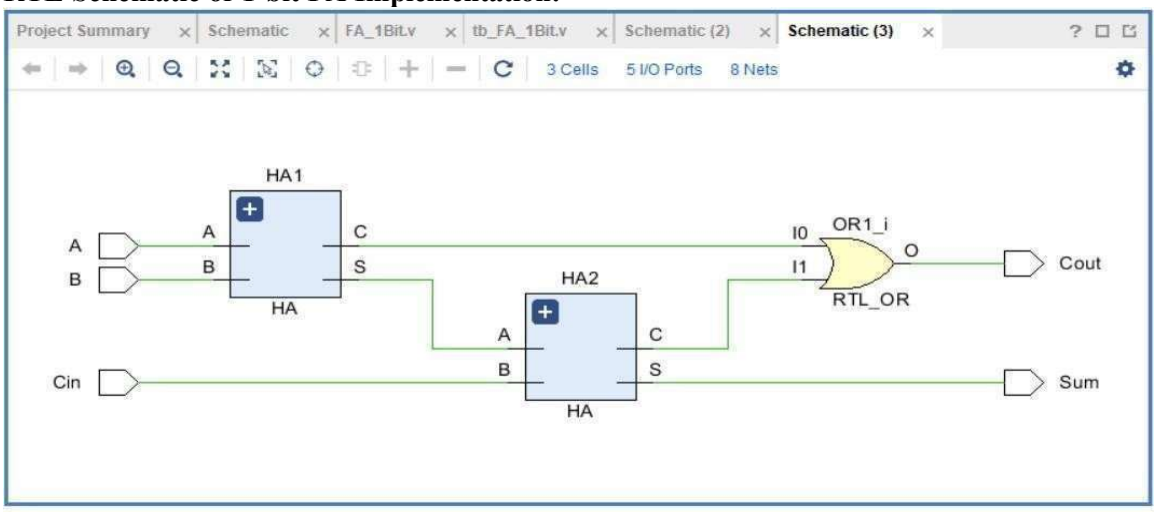

```
//Half Adder Module
module HA (A, B, S, C);                 // HA - Half Adder
        input A, B;
        output S, C;                    // S- Sum , C - Carry
        xor x1(S, A, B);                // sum = A xor B
        and n1(C, A, B);                // carry = A and B
endmodule

//Full Adder Module
module FA_1Bit( A, B, Cin, Sum, Cout);
        input A, B, Cin;
        output Sum, Cout;
        wire S1, C1, C2;
        // Module instantiation
        HA HA1 (A, B, S1, C1);
        HA HA2 (S1, Cin, Sum, C2);
        OR OR1 (Cout, C1, C2);
endmodule
```

Testbench:

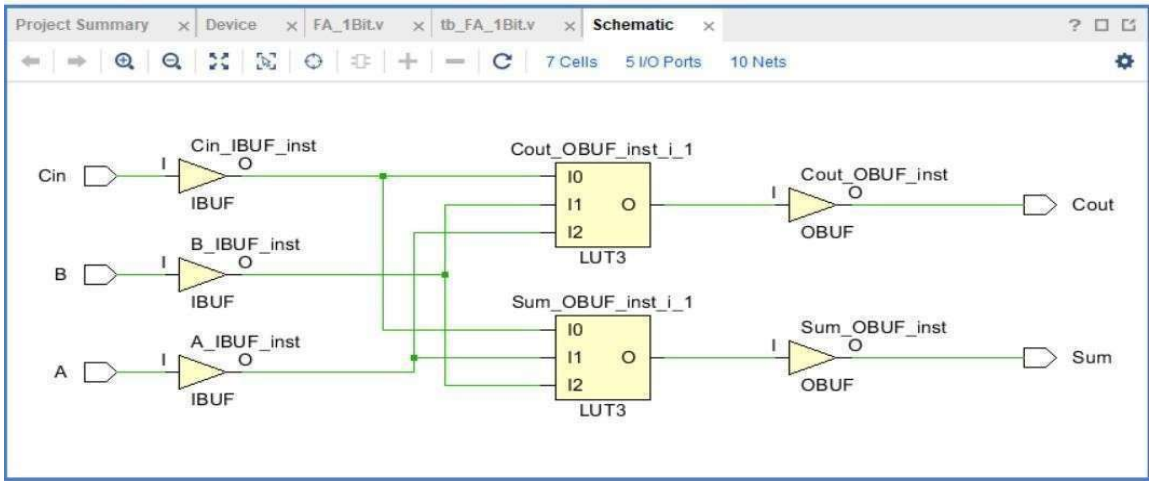
```
`timescale 1ns/1ps  // Simulation time unit /

precision module tb_FA_1Bit;     // Testbench for

Full Adder
reg A, B, Cin;       // Testbench inputs
wire Sum, Cout;      // Testbench outputs

// Instantiate the Full Adder (Unit Under Test)
FA_1Bit uut (
.A(A),
.B(B),
.Cin(Cin),
.Sum(Sum),
.Cout(Cout)
);
```

```
// Stimulus block
initial begin
    $monitor("Time=%0t | A=%b B=%b Cin=%b | Sum=%b Cout=%b",
            $time, A, B, Cin, Sum, Cout);

                // Apply all 8 input combinations
                A = 0; B = 0; Cin = 0; #10;   // Case 1
                A = 0; B = 0; Cin = 1; #10;   // Case 2
                A = 0; B = 1; Cin = 0; #10;   // Case 3
                A = 0; B = 1; Cin = 1; #10;   // Case 4
                A = 1; B = 0; Cin = 0; #10;   // Case 5
                A = 1; B = 0; Cin = 1; #10;   // Case 6
                A = 1; B = 1; Cin = 0; #10;   // Case 7
                A = 1; B = 1; Cin = 1; #10;   // Case 8

        $stop; // End simulation
      end
endmodule
```

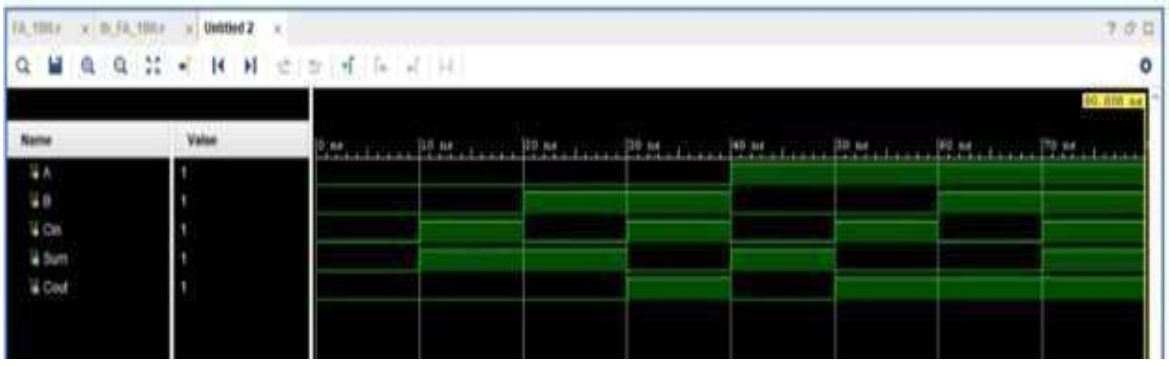
RTL Schematic of 1-bit FA Implementation:



Synthesis Schematic:



Simulation waveforms:



(c) 1-bit Full Adder (Dataflow and behavioural modeling):

$$Sum = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + BC_{in} + AC_{in}$$

Full Adder (Dataflow modeling):

```
module FA_1_Bit_DF(A,B,Cin,Sum,Cout);
        input A,B,Cin;
        output Sum,Cout;
        assign {Cout,Sum} = A + B + Cin;
endmodule
```

Full Adder (Behavioural modeling):

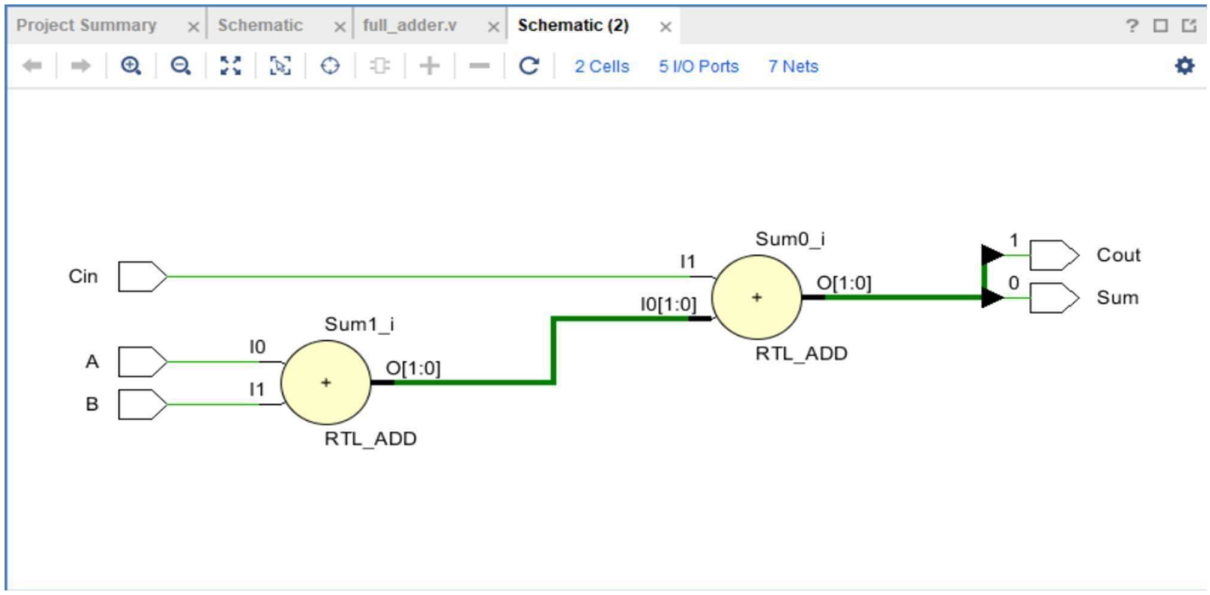
```
module FA_1_Bit_BH(A,B,Cin,Sum,Cout);
        input A,B,Cin;
        output reg Sum,Cout;
        always @( A,B,Cin)
            begin
                {Cout,Sum} = A + B + Cin;
             end
endmodule
```

Full Adder (Test Bench):

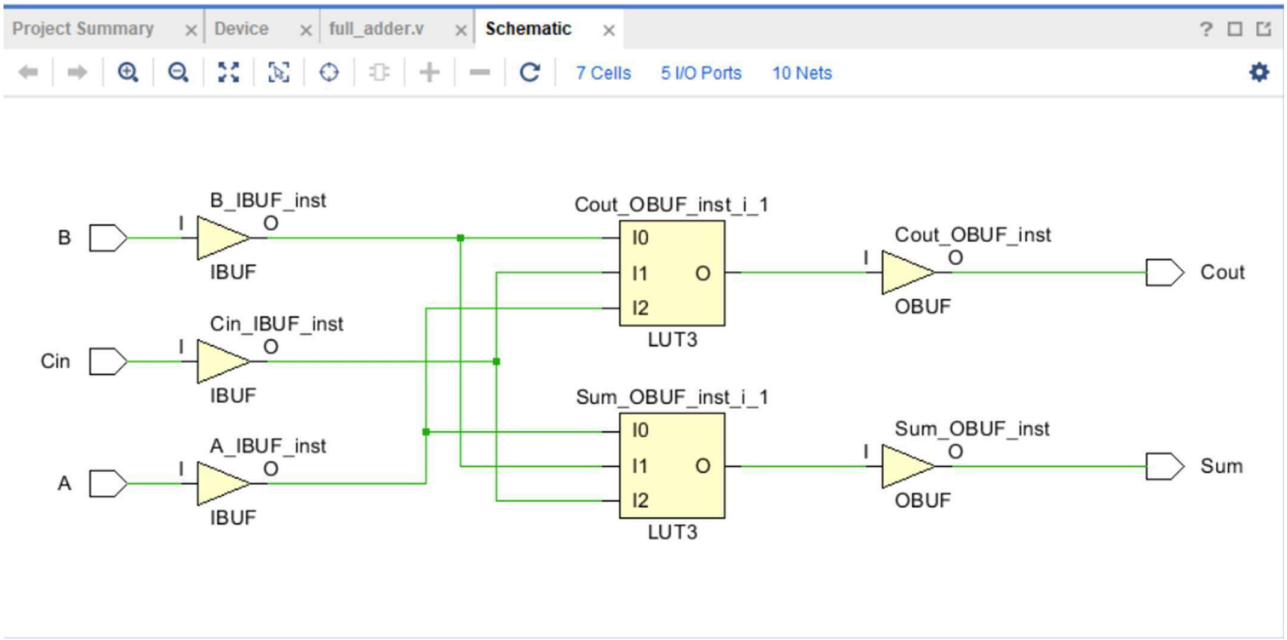
```
`timescale 1ns/1ps

module tb_full_adder;
    reg A, B, Cin;
    wire Sum, Cout;

    // Instantiate the DUT
    full_adder uut (
        .A(A),
        .B(B),
        .Cin(Cin),
        .Sum(Sum),
        .Cout(Cout)
    );

    initial begin
        $monitor("Time=%0d | A=%b B=%b Cin=%b | Sum=%b Cout=%b",
                 $time, A, B, Cin, Sum, Cout);
```

```
        // Test all input combinations
        A=0; B=0; Cin=0; #10;
        A=0; B=0; Cin=1; #10;
        A=0; B=1; Cin=0; #10;
        A=0; B=1; Cin=1; #10;
        A=1; B=0; Cin=0; #10;
        A=1; B=0; Cin=1; #10;
        A=1; B=1; Cin=0; #10;
        A=1; B=1; Cin=1; #10;

        $stop;
    end
endmodule
```

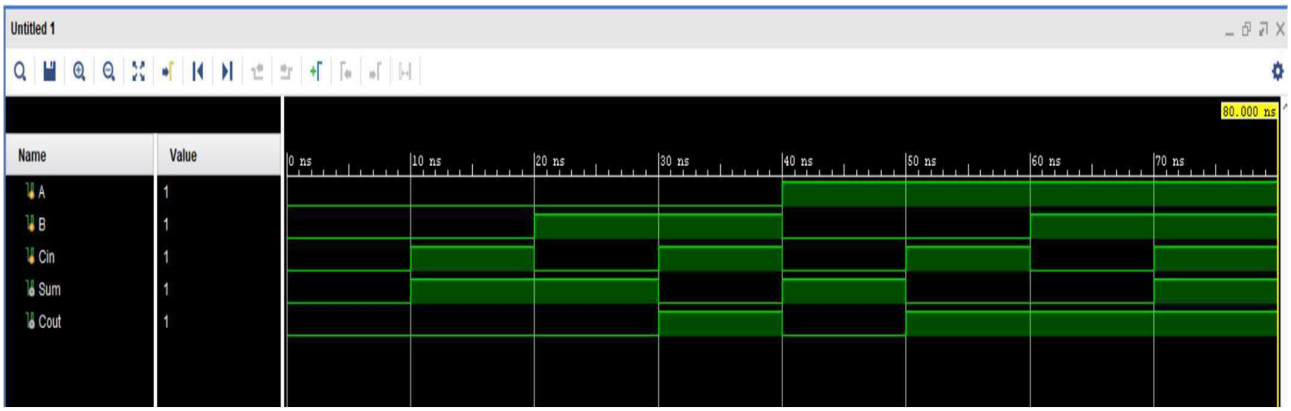
RTL Schematic



Synthesis Schematic:



Simulation waveforms:



(d) Half Subtractor:

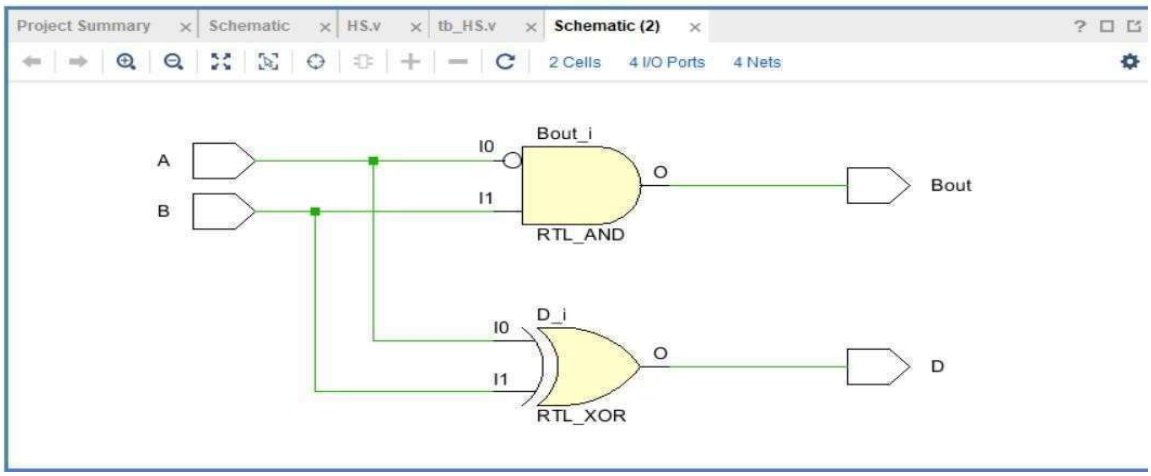
RTL CODE:

```
module HS(A, B, D, Bout);
   input A, B;              // Inputs
   output D, Bout;          // Outputs: Difference, Borrow

   assign D = A ^ B;        // Difference = A XOR B
   assign Bout = (~A) & B;  // Borrow = NOT(A) AND B
endmodule
```

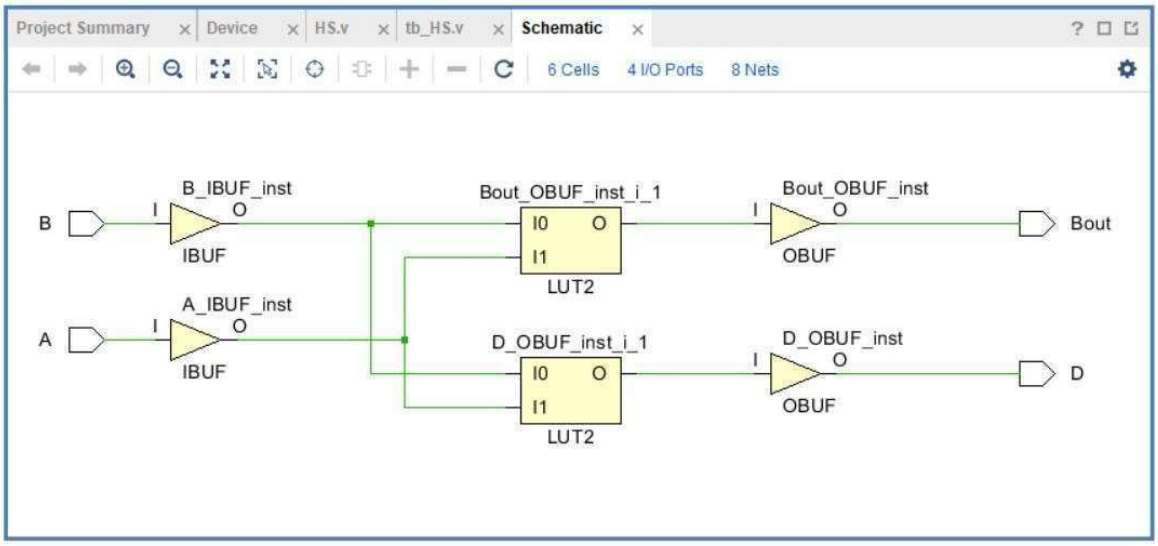
Testbench:

```
`timescale 1ns/1ps

module tb_HS;
   reg A, B;          // Inputs
   wire D, Bout;      // Outputs

   // Instantiate HS
   HS uut (
      .A(A),
      .B(B),
      .D(D),
      .Bout(Bout)
   );

   initial begin
      $monitor("Time=%0t | A=%b B=%b | Diff=%b Borrow=%b",
            $time, A, B, D, Bout);

      // Test all 4 cases
      A=0; B=0; #10;
      A=0; B=1; #10;
      A=1; B=0; #10;
      A=1; B=1; #10;

      $stop;

   end
endmodule
```

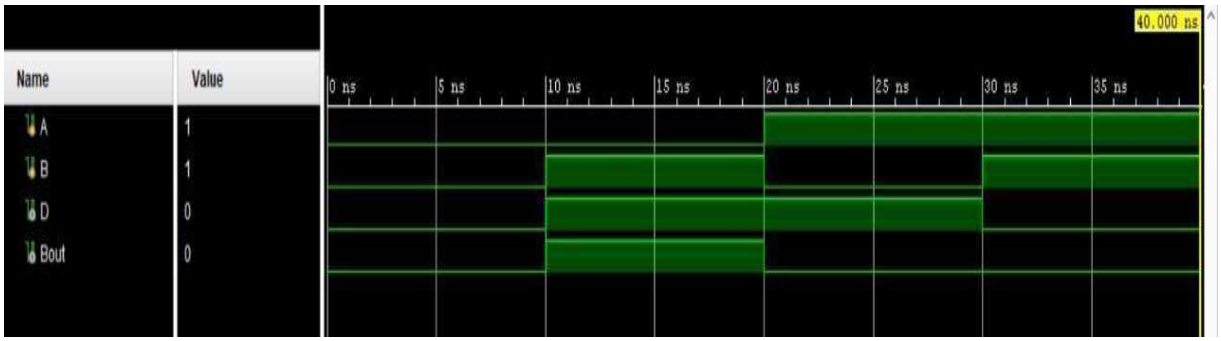
RTL Schematic:



Synthesis Schematic:



Simulation waveforms:



(e) Full Subtractor:

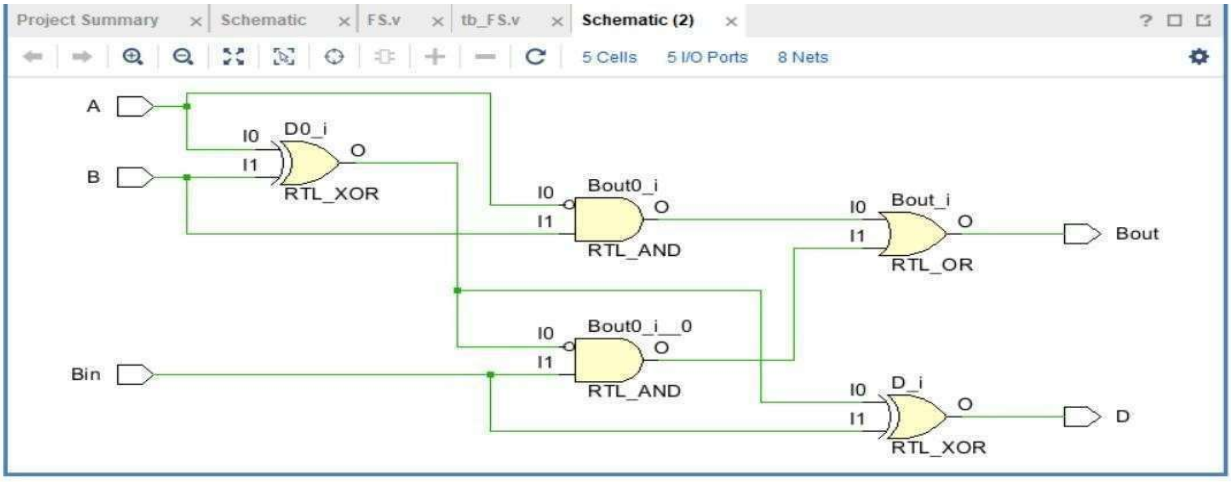
RTL code:

```
// Full Subtractor Module
module FS(A, B, Bin, D, Bout);
   input A, B, Bin;        // Inputs
   output D, Bout;         // Outputs: Difference, Borrow

   assign D = A ^ B ^ Bin;                      // Difference
   assign Bout = ((A) & B) | (((A ^ B)) & Bin);  // Borrow logic
endmodule
```

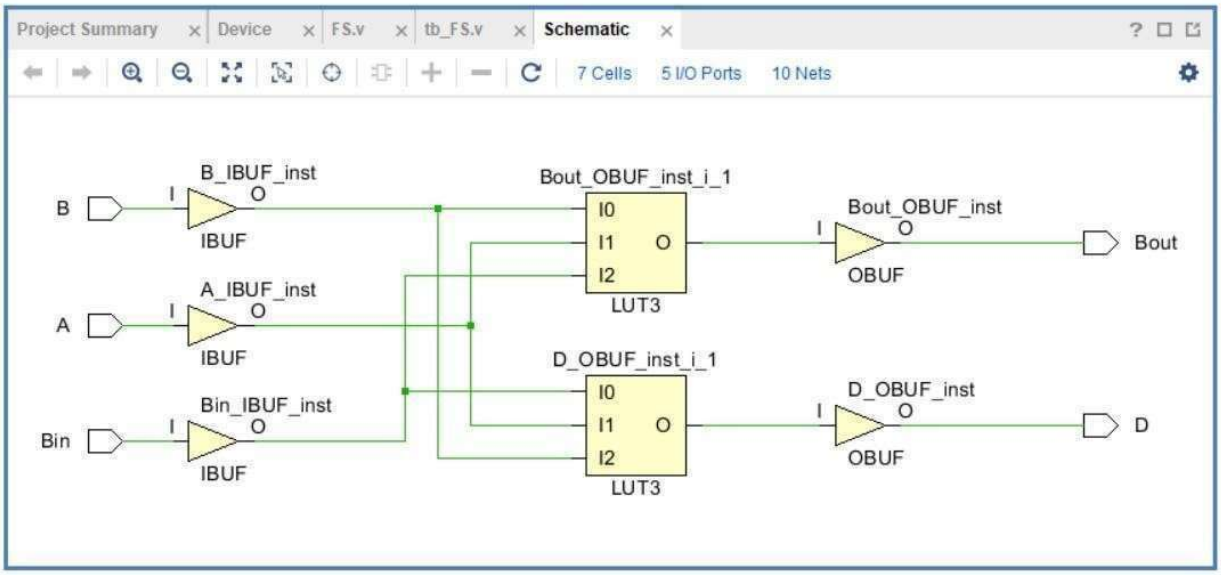
Testbench:

```
`timescale 1ns/1ps

module tb_FS;
   reg A, B, Bin;        // Inputs
   wire D, Bout;         // Outputs

   // Instantiate FS
   FS uut (
      .A(A),
      .B(B),
      .Bin(Bin),
      .D(D),
      .Bout(Bout)
   );

   initial begin
      $monitor("Time=%0t | A=%b B=%b Bin=%b | Diff=%b Borrow=%b",
               $time, A, B, Bin, D, Bout);

      // Test all 8 input cases
      A=0; B=0; Bin=0; #10;
      A=0; B=0; Bin=1; #10;
      A=0; B=1; Bin=0; #10;
      A=0; B=1; Bin=1; #10;
      A=1; B=0; Bin=0; #10;
      A=1; B=0; Bin=1; #10;
      A=1; B=1; Bin=0; #10;
      A=1; B=1; Bin=1; #10;

      $stop;
   end
endmodule
```

RTL Schematic:



Synthesis Schematic:



Simulation waveforms:

Conclusion: In this experiment, various arithmetic circuits like Half Adder, Full Adder, Half Subtractor, and Full Subtractor were successfully designed and simulated using Verilog in structural, dataflow, and behavioral modeling styles. The obtained RTL schematics and waveforms verified the correctness of the designs, confirming that the modules performed the expected logical operations. This helped in understanding the implementation of combinational logic using HDL and the importance of different modeling approaches in digital design.

Suggested Reference:

References used by the students:

Rubric wise marks obtained:

Rubrics	**1**	**2**	**3**	**4**	**5**	**Total**
Marks						

Experiment No: 6

Date: _____

Aim: Design Binary to Gray & Gray to Binary encoder using Verilog/VHDL.

Competency and Practical Skills: Basic Digital Design

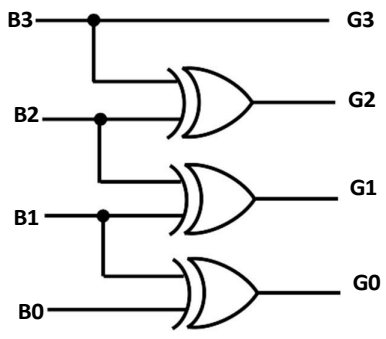
Relevant CO: CO5

Objectives: Designing of code converters

Equipment / Instruments: Laptop or Computer with Xilinx / .

Basic Theory:

Decimal Equivalent	Binary Code				Gray code			
	B3	B2	B1	B0	G3	G2	G1	G0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1
10	1	0	1	0	1	1	1	1
11	1	0	1	1	1	1	1	0
12	1	1	0	0	1	0	1	0
13	1	1	0	1	1	0	1	1
14	1	1	1	0	1	0	0	1
15	1	1	1	1	1	0	0	0

Binary to Gray Converter	Gray to Binary Converter
	

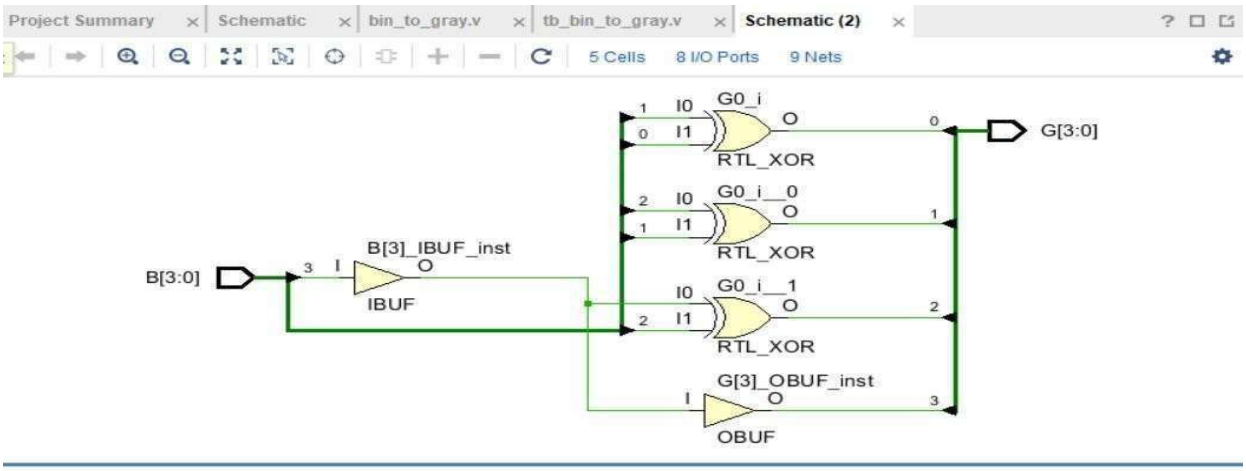
RTL Code:

```
 module bin_to_gray(B, G);
     input [3:0] B;
     output [3:0] G;
     assign G[3] = B[3];
     assign G[2] = B[3] ^ B[2];
     assign G[1] = B[2] ^ B[1];
     assign G[0] = B[1] ^ B[0];
endmodule
```

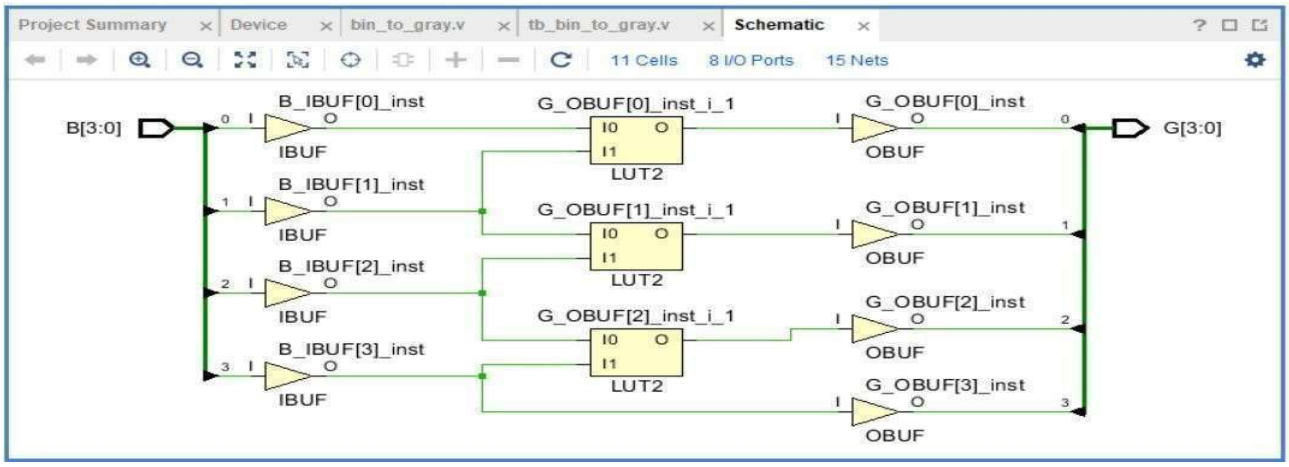
Testbench:

```
`timescale 1ns/1ps

module tb_bin_to_gray;
   reg [3:0] B;       // 4-bit Binary input
   wire [3:0] G;      // 4-bit Gray output

   // Instantiate the Binary to Gray module
   bin_to_gray uut (
      .B(B),
      .G(G)
   );

   initial begin
      $monitor("Time=%0t | Binary=%b | Gray=%b",
               $time, B, G);

      // Apply all 16 possible 4-bit binary inputs
      B = 4'b0000; #10;
      B = 4'b0001; #10;
      B = 4'b0010; #10;
      B = 4'b0011; #10;
      B = 4'b0100; #10;
      B = 4'b0101; #10;
      B = 4'b0110; #10;
      B = 4'b0111; #10;
      B = 4'b1000; #10;
      B = 4'b1001; #10;
      B = 4'b1010; #1B = 4'b1011; #10;
      B = 4'b1100; #10;
      B = 4'b1101; #10;
      B = 4'b1110; #10;
      B = 4'b1111; #10;

      $stop; // End simulation
   end
endmodule
```

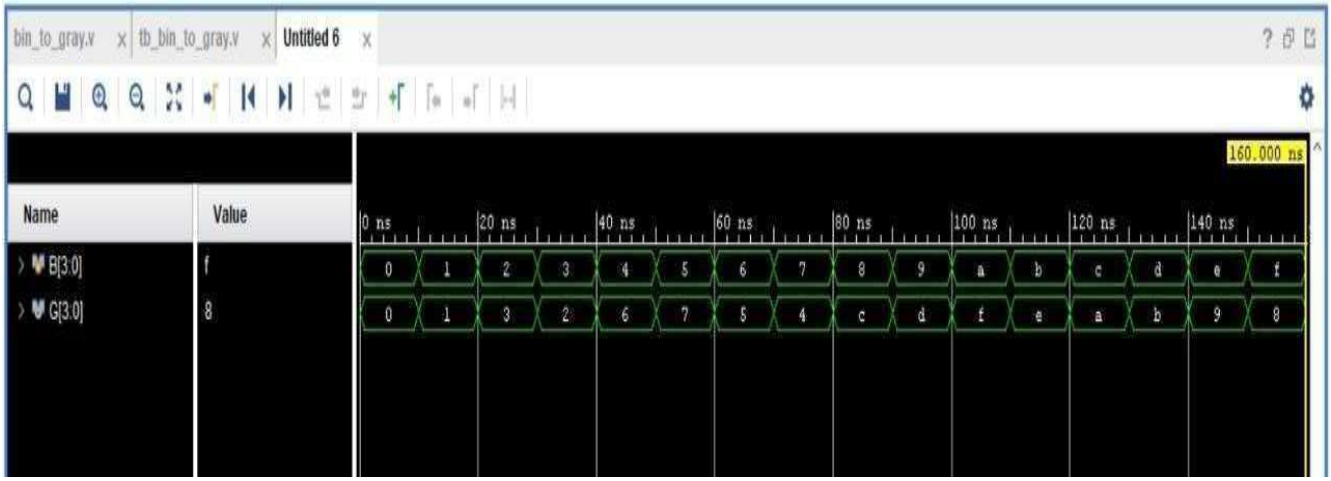
RTL Schematic:



Synthesis Schematic:



Simulation waveforms:



Gray to Binary Encoder :

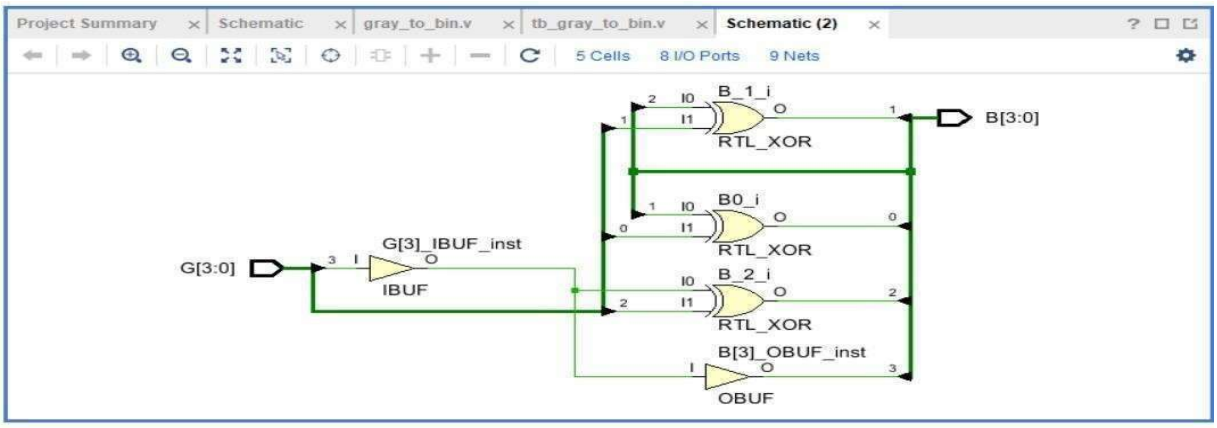
RTL code:

```
// Gray to Binary Module
module gray_to_bin(G, B);
   input [3:0] G;  // 4-bit Gray input output
   [3:0] B;    // 4-bit Binary output

   // Conversion logic
   assign B[3] = G[3];            // MSB same
   assign  B[2]  =  B[3]  ^  G[2];  // Next  bit
   assign  B[1]  =  B[2]  ^  G[1];  // Next  bit
   assign B[0] = B[1] ^ G[0];    // LSB
endmodule
```

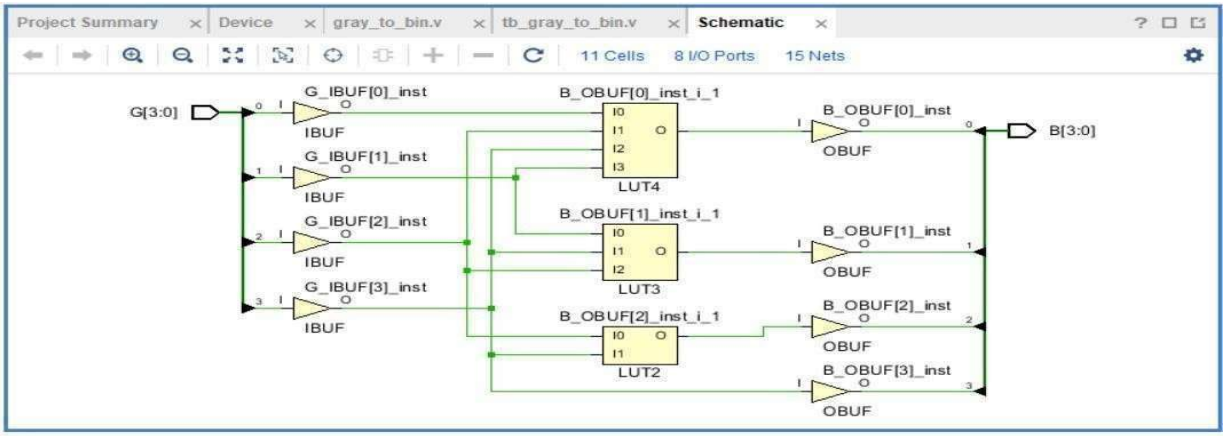
Testbench:

```
`timescale 1ns/1ps

module tb_gray_to_bin;
   reg  [3:0] G;      // 4-bit Gray input
   wire [3:0] B;      // 4-bit Binary output

   // Instantiate the Gray to Binary module
   gray_to_bin uut ( .G(G) .B(B) );

   initial begin
      $monitor("Time=%0t | Gray=%b | Binary=%b",
            $time, G, B);

      // Apply all 16 possible 4-bit Gray inputs
      G = 4'b0000; #10;
      G = 4'b0001; #10;
      G = 4'b0011; #10;
      G = 4'b0010; #10;
      G = 4'b0110; #10;
      G = 4'b0111; #10;
      G = 4'b0101; #10;
      G = 4'b0100; #10;
      G = 4'b1100; #10;
      G = 4'b1101; #10;
      G = 4'b1111; #10;
      G = 4'b1110; #10;
      G = 4'b1010; #10;
      G = 4'b1011; #10;
      G = 4'b1001; #10;
      G = 4'b1000; #10;

      $stop;
   end
endmodule
```

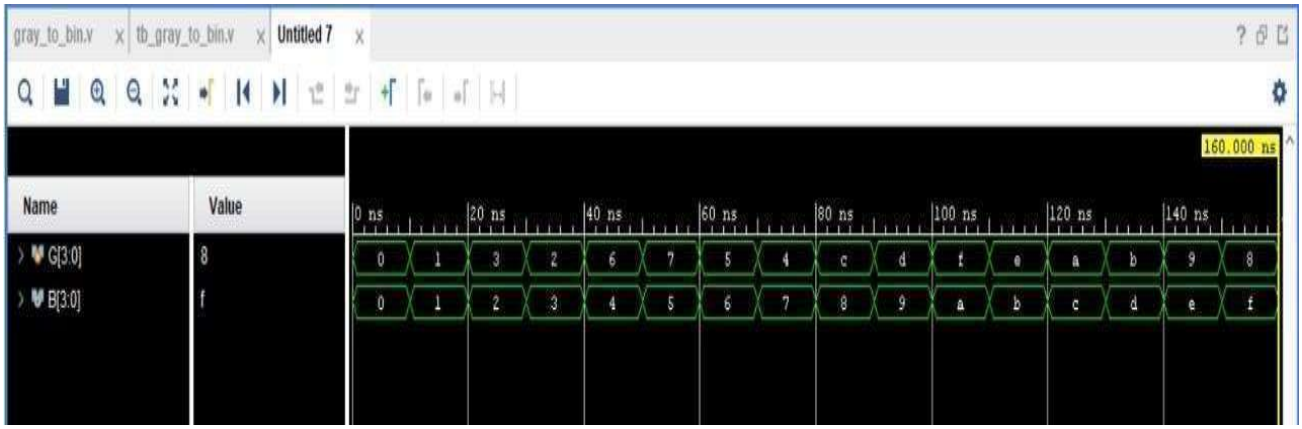
RTL Schematic:



Synthesis Schematic:



Simulation waveforms:



Conclusion: Binary-to-Gray and Gray-to-Binary converters were implemented and verified using Verilog. The simulation results matched the truth table, confirming proper conversion. The experiment demonstrated the importance of code converters in minimizing errors during digital communication and sequential logic design.

Suggested Reference:

References used by the students:

Rubric wise marks obtained:

Rubrics	1	2	3	4	5	Total
Marks						

Experiment No: 7

Date: _____

Aim: Design Multiplexer and Demultiplexer using Verilog / VHDL.

a. **2:1 Mux (Dataflow and Behavioural modeling)**
b. **4:1 Mux (Structural and Dataflow modeling)**
c. **8:1 Mux (Using 4:1 and 2:1 Mux : Structural modeling)**
d. **16:1 Mux (Using Behavioural Modeling & 4:1 Mux : Structural modeling)**
e. **1:8 Demux**

Competency and Practical Skills: Basic Digital Design

Relevant CO: CO5

Objectives: Designing of Multiplexers and Demultiplexers

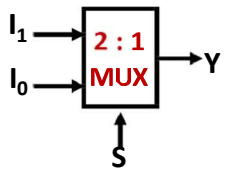
Equipment / Instruments: Laptop or Computer with Xilinx Vivado.p .

Basic Theory:

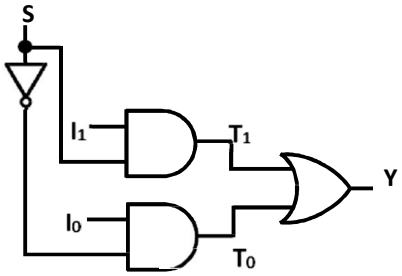
MUX:

A digital logic circuit which is capable of accepting several inputs and generating a single output is known as multiplexer or MUX.

(a) 2:1 Mux (Dataflow and Behavioural modeling)



$Y = S I_1 + \overline{S} I_0$



RTL Code (Dataflow Modeling):

```
module mux2_to_1(I, S, Y);
    input [1 : 0] I;
    input S;
    output Y;
    assign Y = S ? I[1] : I[0];          // Data flow modeling
endmodule
```

RTL Code (Behavioural Modeling):

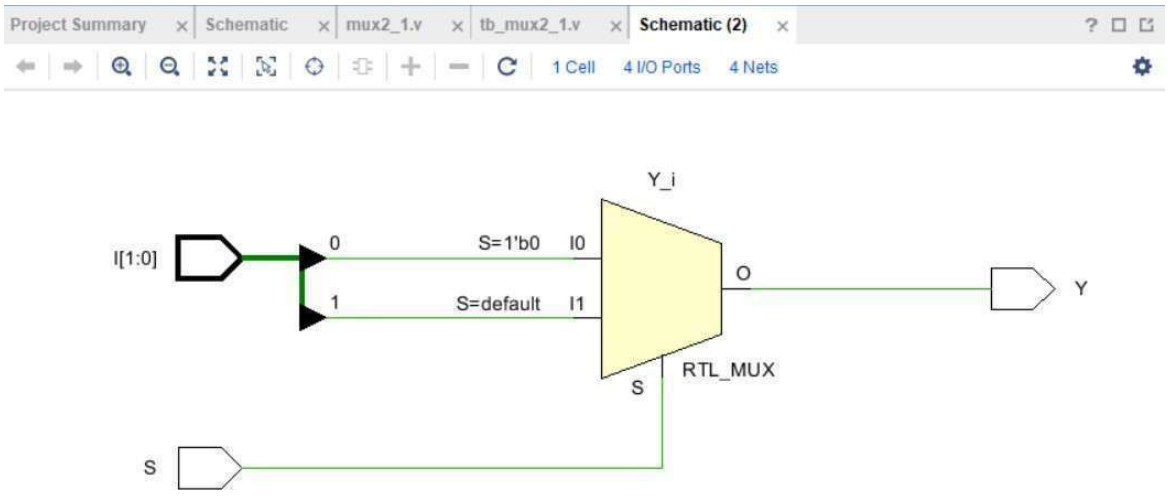
```
module mux2_to_1(I, S, Y);
    input [1 : 0] I;
    input S;
    output reg Y;
```

```
always @(I, S)
  begin
      if (S == 0)
            Y <= I[0];
      else
            Y <= I[1];
endmodule
```

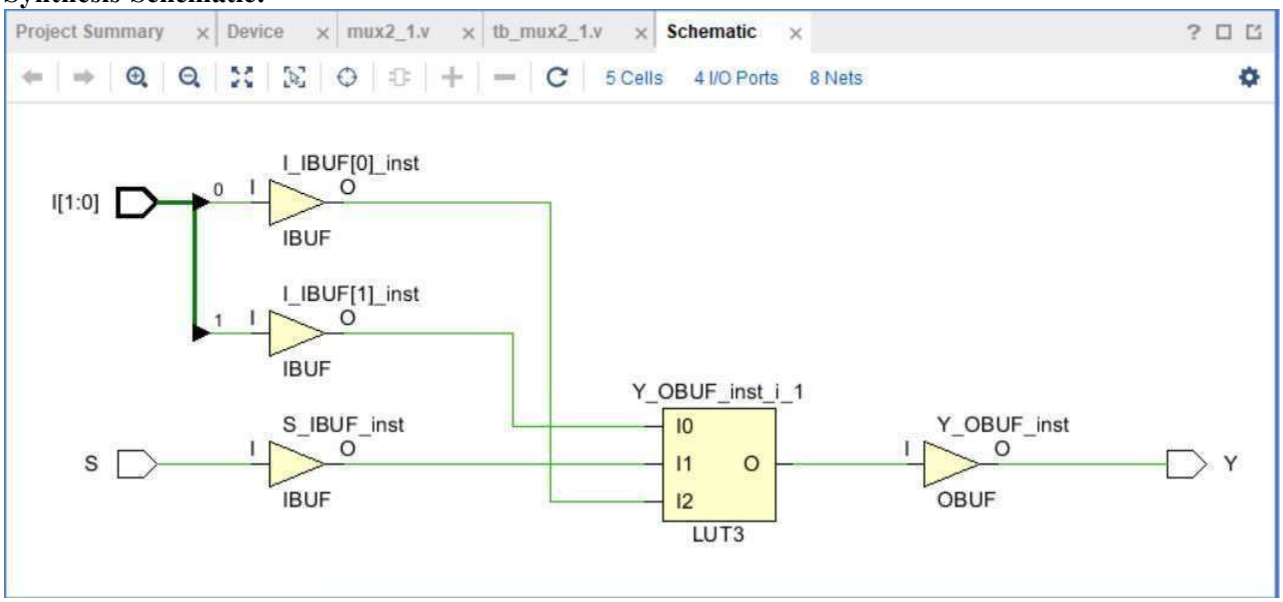
Test Bench:

```
`timescale 1ns / 1ps
module tb_mux2to1;
        reg [1:0] I;       // Inputs
        reg S;
        wire Y;            // Outputs

// Instantiate the Unit Under Test (UUT)
mux2to1 uut (.I(I), .S(S), .Y(Y));
initial
   begin
        // Initialize Inputs
        I[0] = 0; I[1] = 0;  S = 0;
        // Add stimulus here
        #10  I[0] = 0;   I[1] = 1; S = 0;
        #10  I[0] = 1;   I[1] = 1; S = 0;
        #10  I[0] = 0;   I[1] = 0; S = 1;
        #10  I[0] = 0;   I[1] = 1; S = 1;
        #10 $finish;
    end

endmodule
```

RTL Schematic:



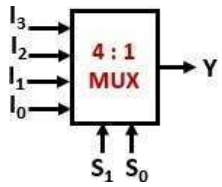
Synthesis Schematic:



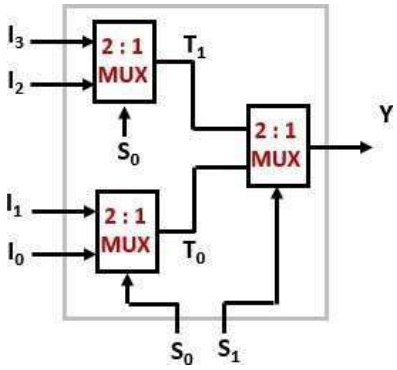
Simulation Waveforms:

(b) 4:1 Mux (Structural and Dataflow modeling)

$$Y = S_1S_0\,I_3 + S_1\bar{S}_0\,I_2 + \bar{S}_1S_0\,I_1 + \bar{S}_1\bar{S}_0\,I_0$$



(4 : 1 MUX using 2 : 1 MUX)



RTL Code (structural modeling):

```
// 2-to-1 MUX module
module mux2_to_1(I, S, Y);
    input [1 : 0] I;
    input S;
    output Y;

    assign Y = S ? I[1] : I[0];          // Data flow modeling

endmodule

// 4-to-1 MUX module
module mux4_to_1(I, S, Y);
    input [3 : 0] I;
    input [1: 0] S;
    output Y;

    wire [1: 0] t;

     mux2_to_1  mux1(I[1:0], S[0], t[0]);
     mux2_to_1 mux1(I[3:2], S[0], t[1]);
     mux2_to_1  mux1(t, S[1], Y);

endmodule
```

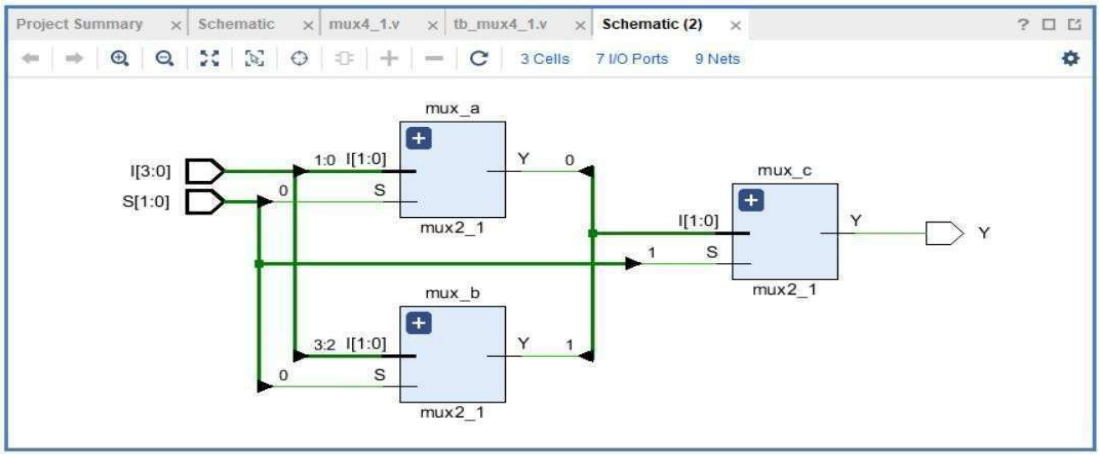
// 4-to-1 MUX (DataFlow Modeling):

```
module mux4_to_1(I, S, Y);
    input [3 : 0] I;
    input [1: 0] S;
    output Y;

        assign Y = I[S];

endmodule
```

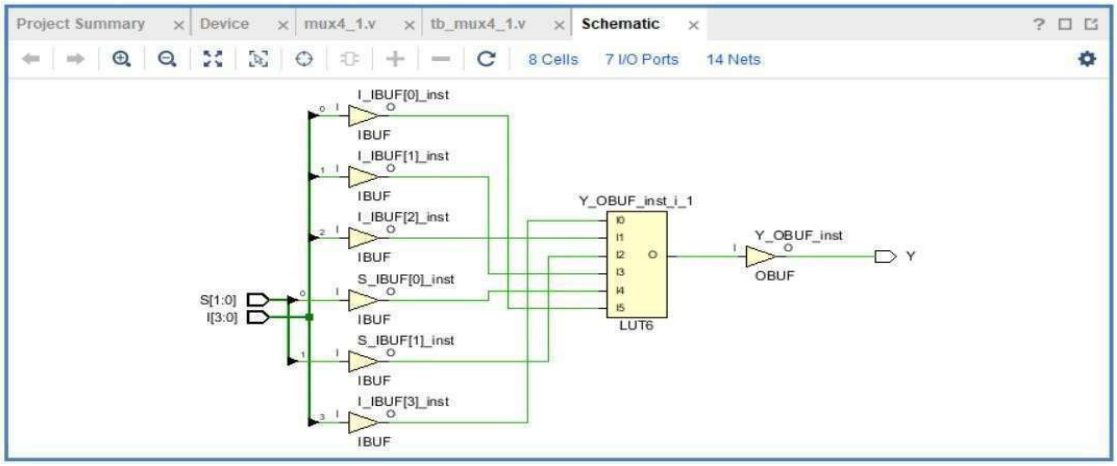
Test Bench

```
`timescale 1ns / 1ps
 module tb_mux_4_to_1;

      reg [3:0] I;              // Inputs
      reg [1:0] S;

      wire Y;                   // Outputs

  // Instantiate the Unit Under Test (UUT)
      mux_4_to_1_st uut (.I(I), .S(S), .Y(Y));

      initial
            begin
        // Initialize Inputs
        I = 0;  S = 0;

        // Add stimulus here
        #10 I = 4'b0001; S = 2'b00;
        #10 I = 4'b1011; S = 2'b00;
        #10 I = 4'b0101; S = 2'b01;
        #10 I = 4'b1111; S = 2'b01;
        #10 I = 4'b1001; S = 2'b10;
        #10 I = 4'b1101; S = 2'b10;
        #10 I = 4'b1011; S = 2'b11;
        #10  I = 4'b1101; S = 2'b11;
        #10  $finish;
        end
       Endmodule
```

RTL Schematic:

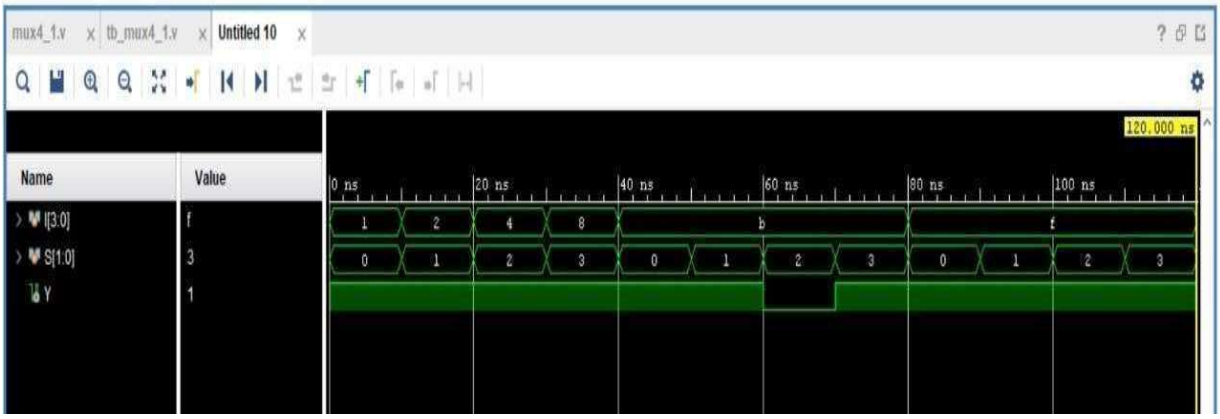


Synthesis Schematic:

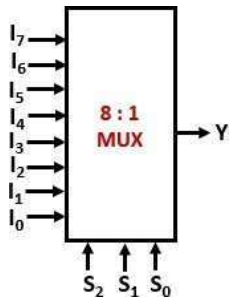


Simulation waveforms:

:



(c) 8:1 Mux (Using 4:1 and 2:1 Mux : Structural modeling)



RTL code:

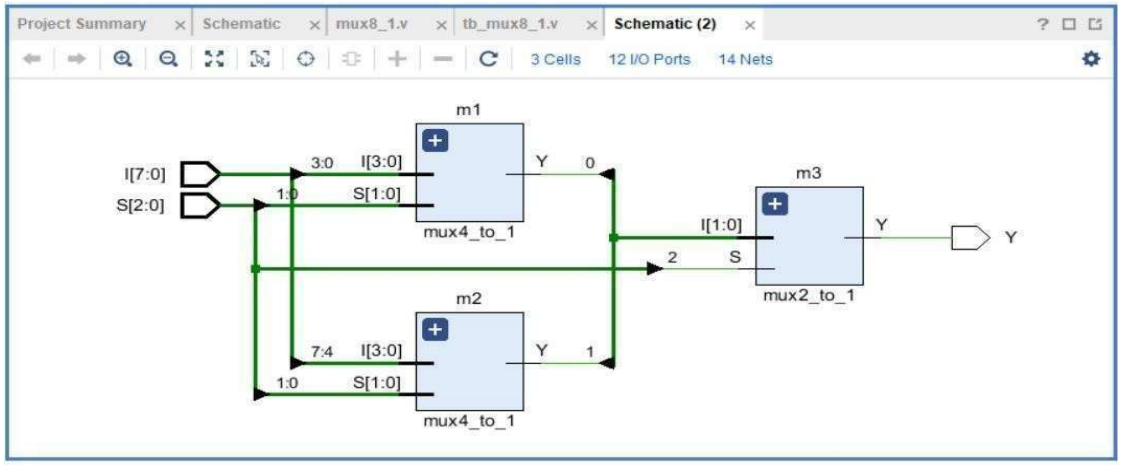
```
// 2-to-1 MUX
module mux2_to_1(I, S, Y);
    input [1:0] I;
    input S;
    output Y;

    assign Y = S ? I[1] : I[0];  // Dataflow
endmodule

// 4-to-1 MUX
module mux4_to_1(I, S, Y);
    input [3:0] I;
    input [1:0] S;
    output Y;

    assign Y = I[S];  // Dataflow (array indexing)
endmodule

// 8-to-1 MUX (Using two 4:1 and one 2:1)
module mux8_to_1(I, S, Y);
    input [7:0] I;
    input [2:0] S;
    output Y;

    wire [1:0] t;

    // First 4:1 handles I[3:0]
    mux4_to_1 m1 (.I(I[3:0]), .S(S[1:0]), .Y(t[0]));

    // Second 4:1 handles I[7:4]
    mux4_to_1 m2 (.I(I[7:4]), .S(S[1:0]), .Y(t[1]));

    // Final 2:1 selects between them using MSB of select
mux2_to_1 m3 (.I(t), .S(S[2]), .Y(Y));
endmodule
```

Testbench:

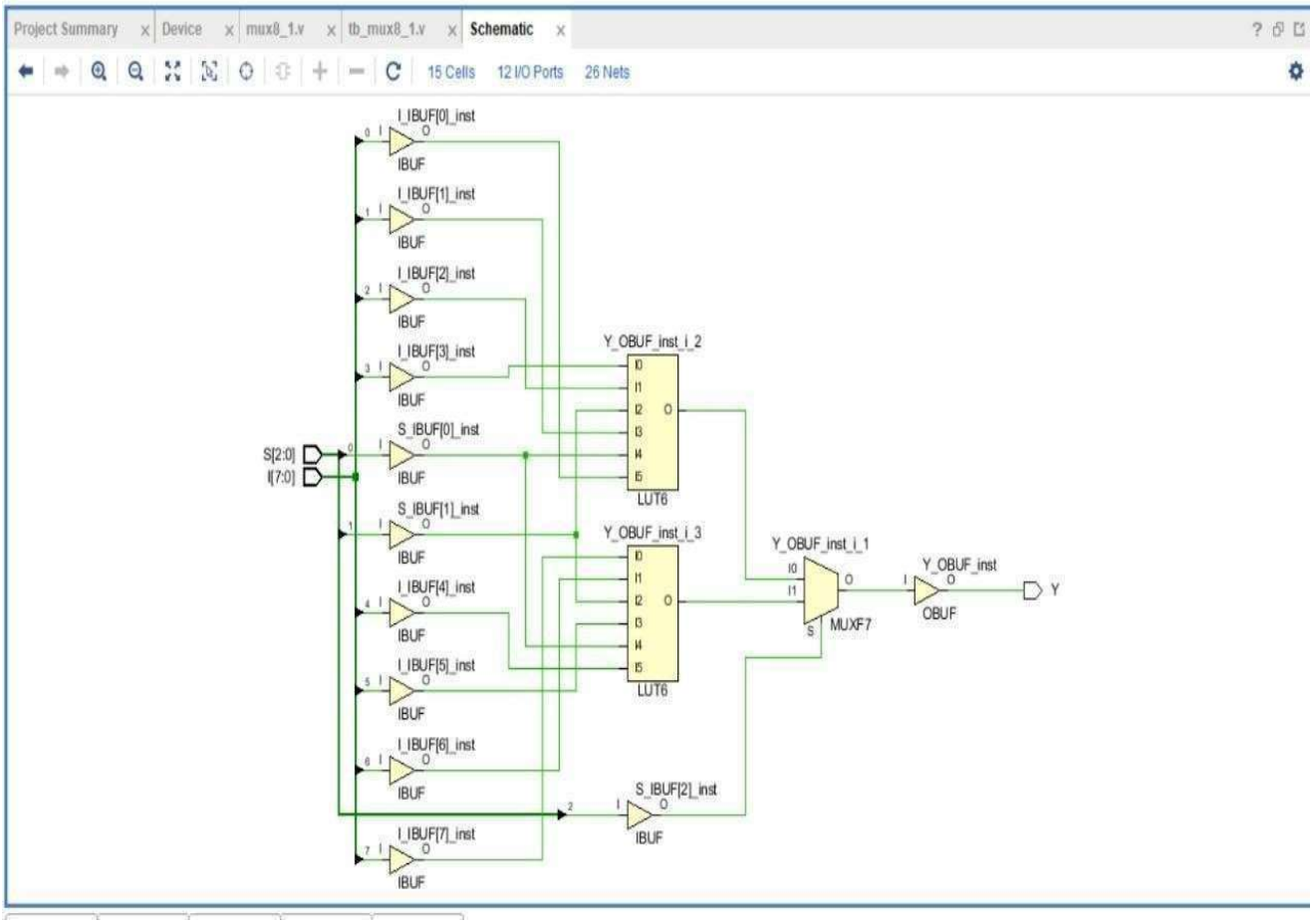
```
`timescale 1ns / 1ps

module tb_mux8_to_1;

    reg [7:0] I;       // 8-bit inputs
    reg [2:0] S;       // Select lines
```

```
    wire Y;            // Output

    // Instantiate the Unit Under Test (UUT)
    mux8_to_1 uut (.I(I), .S(S), .Y(Y));

    integer i, sel;

    initial begin
        $monitor("Time=%0t | I=%b | S=%b | Y=%b", $time, I, S, Y);

        // Loop over all 256 input combinations
        for (i = 0; i < 256; i = i + 1) begin
            I = i[7:0];   // Assign 8-bit input

            // Loop over all 8 select lines
            for (sel = 0; sel < 8; sel = sel + 1) begin
                S = sel[2:0];
                #5; // small delay for waveform
            end
        end

        $finish;
    end

endmodule
```

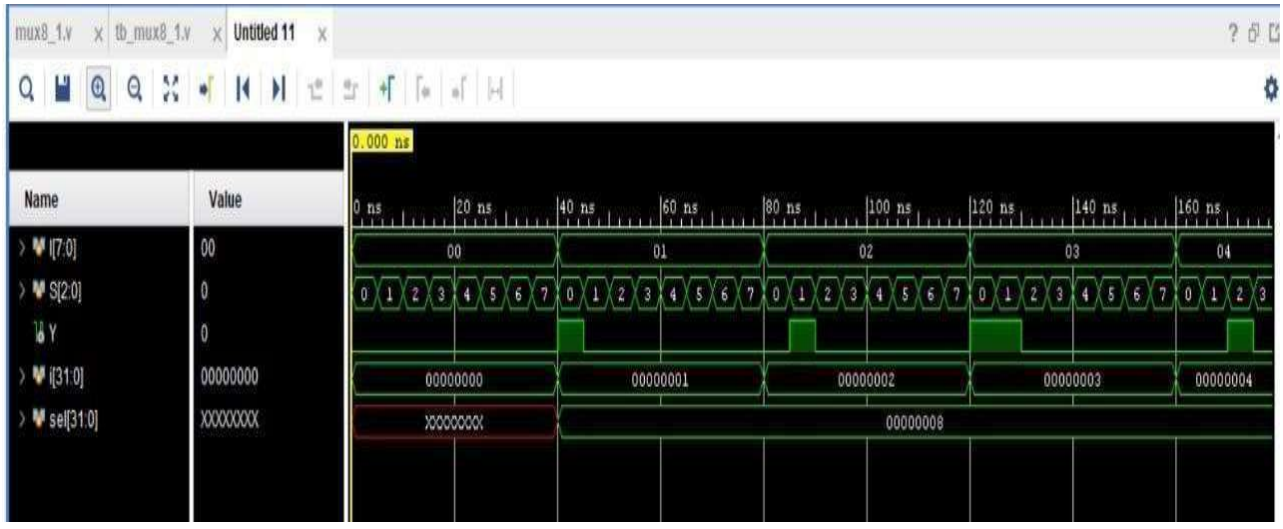
RTL Schematic:



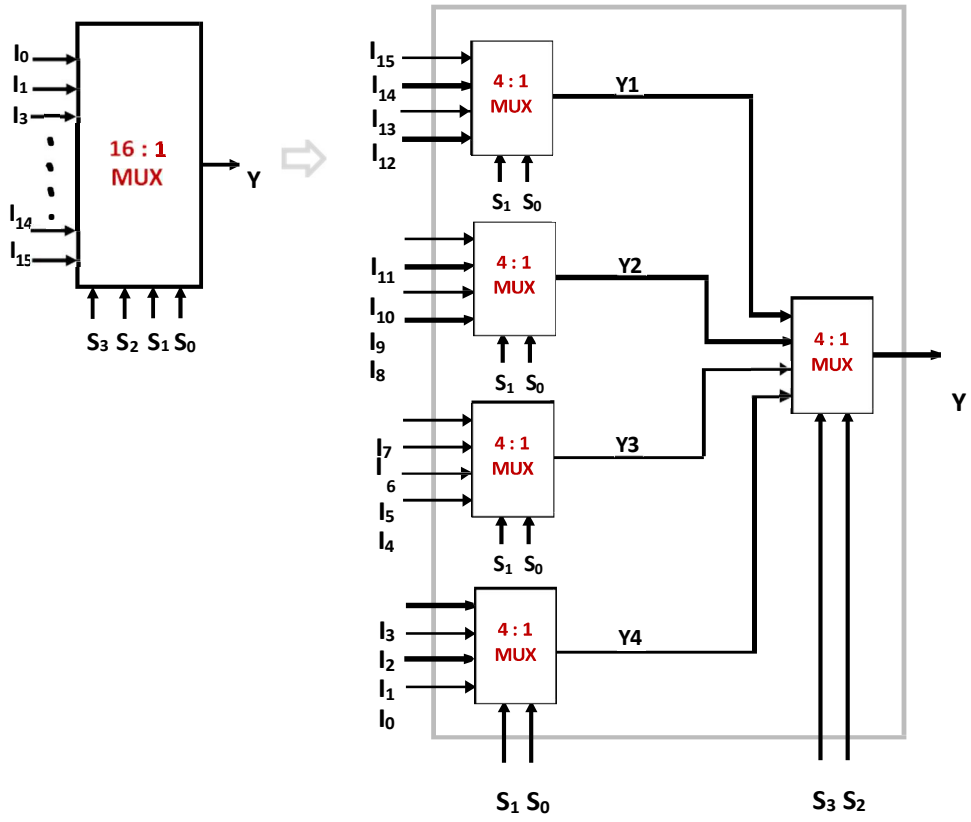
Synthesis Schematic:



Simulation waveforms:



(d) 16:1 Mux (Using Behavioural Modeling & 4:1 Mux : Structural modeling)




```
// 16-to-1 MUX (Behavioral)
module mux16_1(I, S, Y);
   input [15:0] I;      // 16-bit inputs
   input [3:0] S;       // 4-bit select
   output Y;

   assign Y = I[S];      // Direct array indexing
endmodule


// Reusable 2:1 MUX
module mux2_1(I, S, Y);
   input [1:0] I;
   input S;
   output Y;
   assign Y = S ? I[1] : I[0];
endmodule

// Reusable 4:1 MUX
module mux4_1(I, S, Y);
```

```
    input [3:0] I;
    input [1:0] S;
    output Y;
    assign Y = I[S];
endmodule
```

// 16-to-1 MUX (Structural)

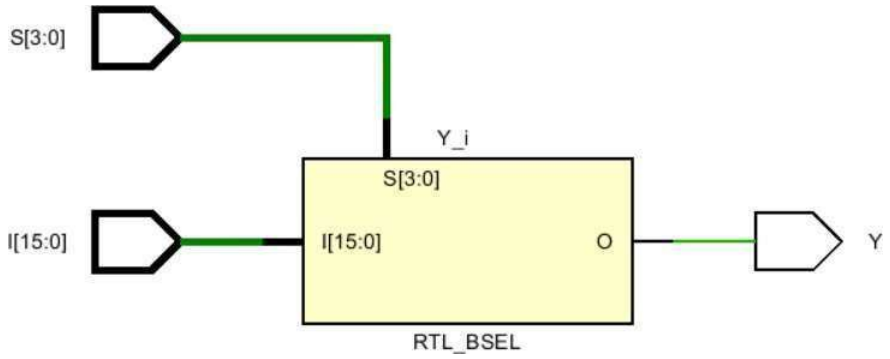
```
module mux16_1(I, S, Y);
    input [15:0] I;
    input [3:0] S;
    output Y;

    wire [3:0] t;   // Intermediate outputs from 4:1 MUXes

    // Four 4:1 MUXes
    mux4_1 m0(.I(I[3:0]),   .S(S[1:0]), .Y(t[0]));
    mux4_1 m1(.I(I[7:4]),   .S(S[1:0]), .Y(t[1]));
    mux4_1 m2(.I(I[11:8]),  .S(S[1:0]), .Y(t[2]));
    mux4_1 m3(.I(I[15:12]), .S(S[1:0]), .Y(t[3]));

    // Final 4:1 MUX to select one of the four
    mux4_1 m4(.I(t), .S(S[3:2]), .Y(Y));
endmodule
```

Testbench:

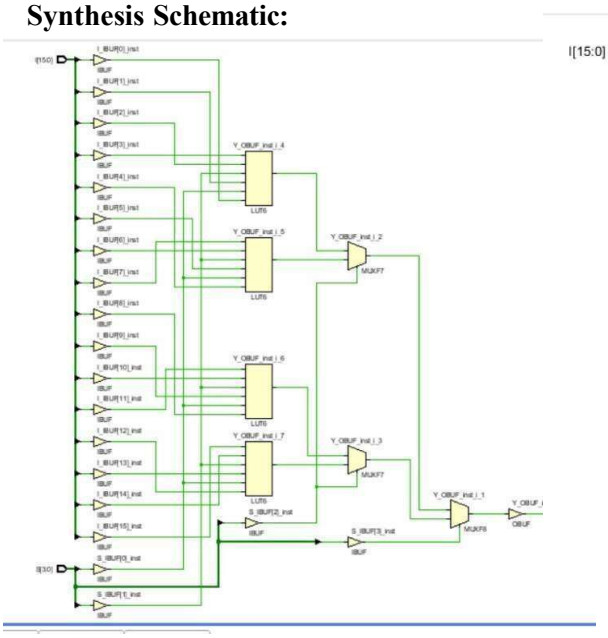
```
`timescale 1ns / 1ps

module tb_mux16_1;
    reg [15:0] I;
    reg [3:0] S;
    wire Y;

    // Instantiate the version you want to test:
    // Uncomment the one you need
    // Behavioral version
    //mux16_1 uut (.I(I), .S(S), .Y(Y));

    // Structural version
    mux16_1 uut (.I(I), .S(S), .Y(Y));

    integer i, sel;
    initial begin
        $monitor("Time=%0t | I=%b | S=%b | Y=%b", $time, I, S, Y);
```

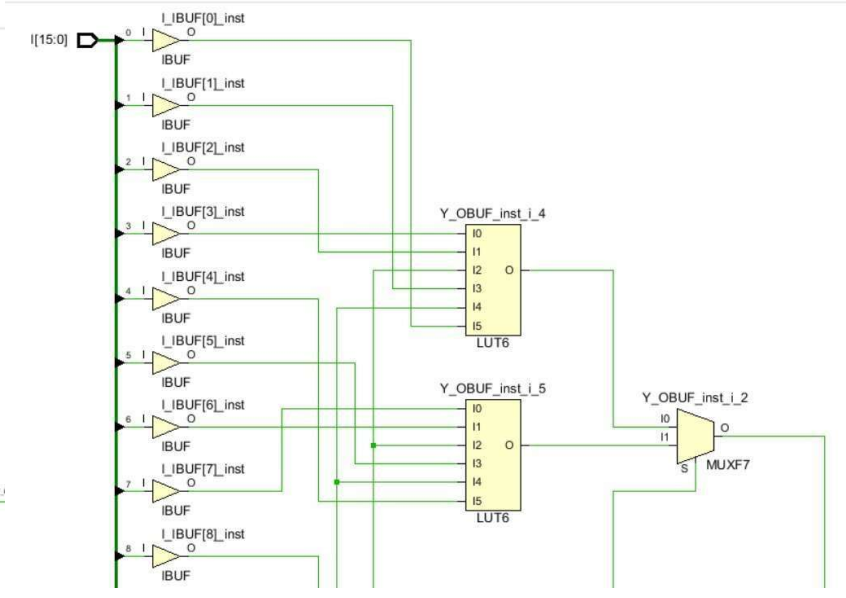
```
        // Test all single-bit-hot input patterns
        for (i = 0; i < 16; i = i + 1) begin
            I = 16'b0;
            I[i] = 1'b1;

            for (sel = 0; sel < 16; sel = sel + 1) begin
                S = sel[3:0];
                #5;
            end
        end
        $finish;
    end
endmodule
```

RTL Schematic:

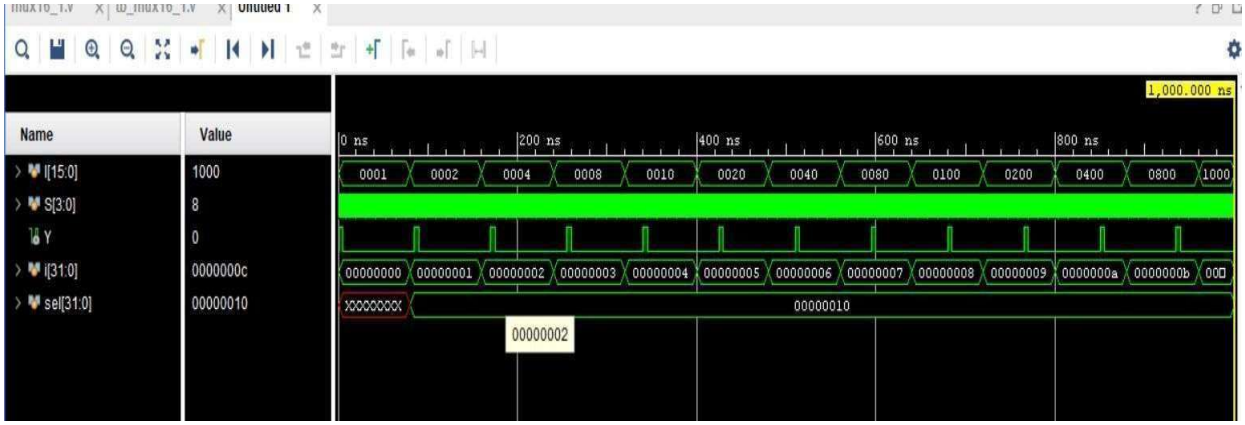


Synthesis Schematic:



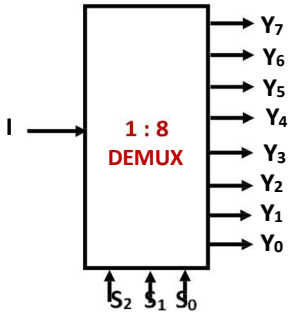


Simulation waveforms:



(e) 1:8 DEMUX

DEMUX: A digital combinational circuit which takes one input line and routs it to one of the multiple output lines depending on the status of the select lines is known as demultiplexer or DEMUX.



RTL Code:

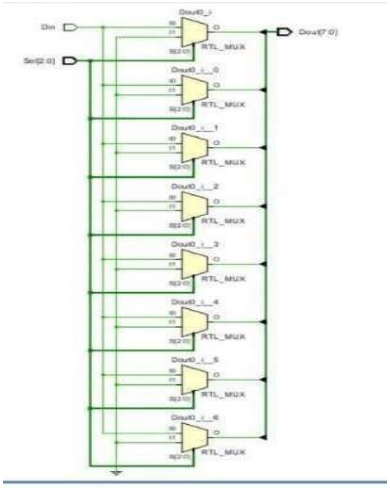
```
// 1-to-8 Demux (Dataflow)
module demux1_8(Din, Sel, Dout);
   input Din;          // Single input
   input [2:0] Sel;     // 3-bit select
   output [7:0] Dout;  // 8-bit output

   assign Dout[0] = (Sel == 3'b000) ? Din : 0;
   assign Dout[1] = (Sel == 3'b001) ? Din : 0;
   assign Dout[2] = (Sel == 3'b010) ? Din : 0;
   assign Dout[3] = (Sel == 3'b011) ? Din : 0;
   assign Dout[4] = (Sel == 3'b100) ? Din : 0;
   assign Dout[5] = (Sel == 3'b101) ? Din : 0;
```

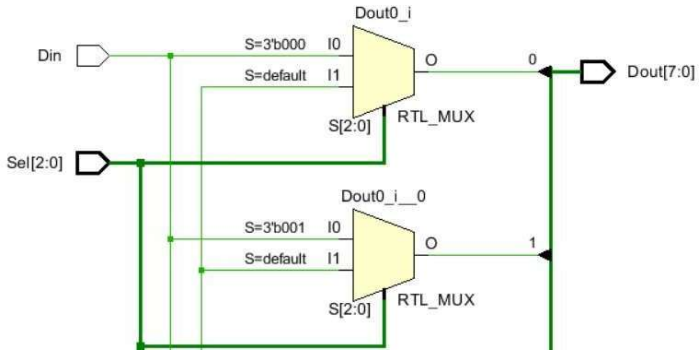
```
    assign Dout[6] = (Sel == 3'b110) ? Din : 0;
    assign Dout[7] = (Sel == 3'b111) ? Din : 0;
endmodule
```

Testbench Code:

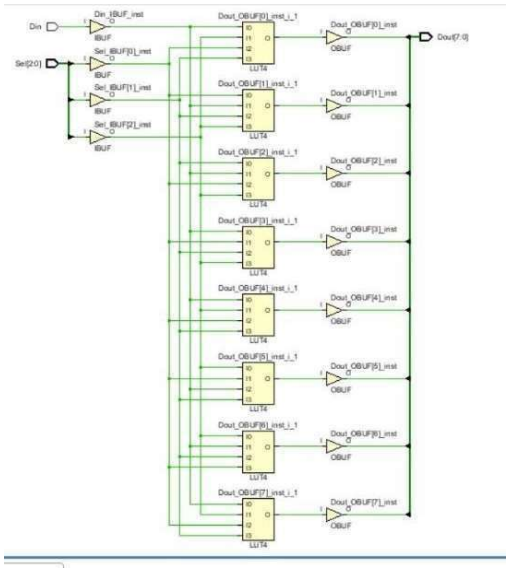
```
`timescale 1ns / 1ps

module tb_demux1_8;
    reg Din;            // Input
    reg [2:0] Sel;      // Select
    wire [7:0] Dout;    // Output

    // Instantiate the Demux
    demux1_8 uut (.Din(Din), .Sel(Sel), .Dout(Dout));

    integer sel;

    initial begin
        $monitor("Time=%0t | Din=%b | Sel=%b | Dout=%b", $time, Din, Sel, Dout);

        // Test Din=0, all selects
        Din = 0;
        for (sel = 0; sel < 8; sel = sel + 1) begin
            Sel = sel[2:0];
            #5;
        end

        // Test Din=1, all selects
        Din = 1;
        for (sel = 0; sel < 8; sel = sel + 1) begin
            Sel = sel[2:0];
            #5;
        end

        $finish;
    end
endmodule
```

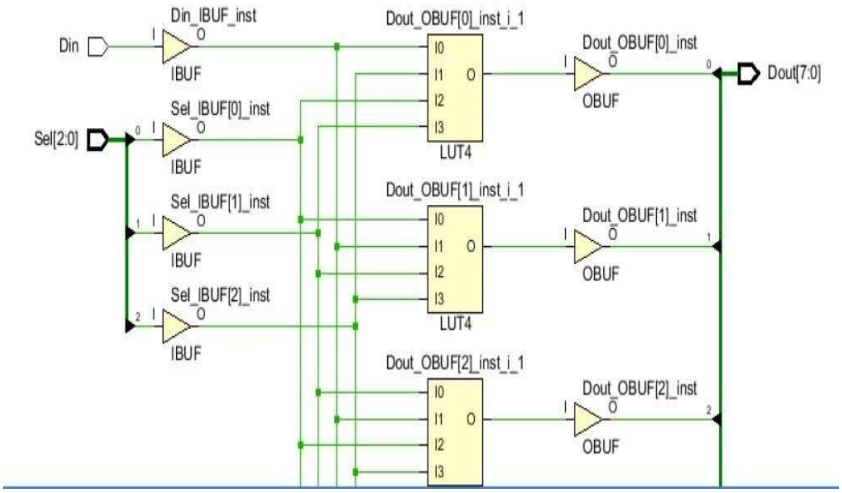
RTL Schematic:





Synthesis Schematic:





Simulation waveforms:

Conclusion: Different types of multiplexers (2:1, 4:1, 8:1, 16:1) and a 1:8 demultiplexer were successfully designed and simulated. RTL and waveform outputs verified correct selection and distribution of data lines. The experiment highlighted how multiplexers simplify logic implementation and demultiplexers distribute data signals based on control inputs.

Suggested Reference:

References used by the students:

Rubric wise marks obtained:

Rubrics	1	2	3	4	5	Total
Marks						

Experiment No: 8

Date: _____

Aim: Design 2:4, 3:8, 4:16 Decoders using Verilog/VHDL.

Competency and Practical Skills:

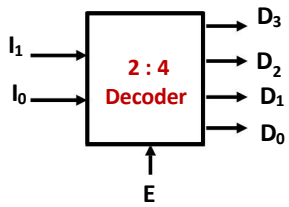
Relevant CO:

Objectives:

Equipment / Instruments: Laptop or Computer with Xilinx / .

Basic Theory: Decoder is a combinational circuit that has '*n*' input lines and maximum of 2^n output lines. Depending on the code presented by input lines, one of the output lines will be active high, when the decoder is enabled. It reveals a decoder detects a particular code presented at input.

(a) 2-to-4 Decoder



Truth Table of 2-to-4 Decoder

Enable	Inputs		Outputs			
E	I_1	I_0	D_3	D_2	D_1	D_0
0	X	X	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

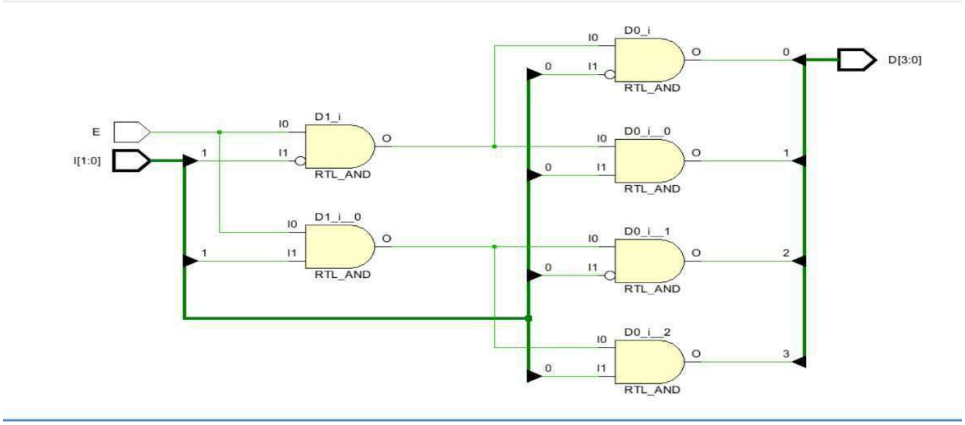
RTL Code (DataFlow Modeling):

```
module Decoder2to4_df(I,E,D);
        input [1:0] I;
        input E;
        output [3:0] D;
        assign D = {E&I[1]&I[0],E&I[1]&~I[0],E&~I[1]&I[0],E&~I[1]&~I[0]};

endmodule
```

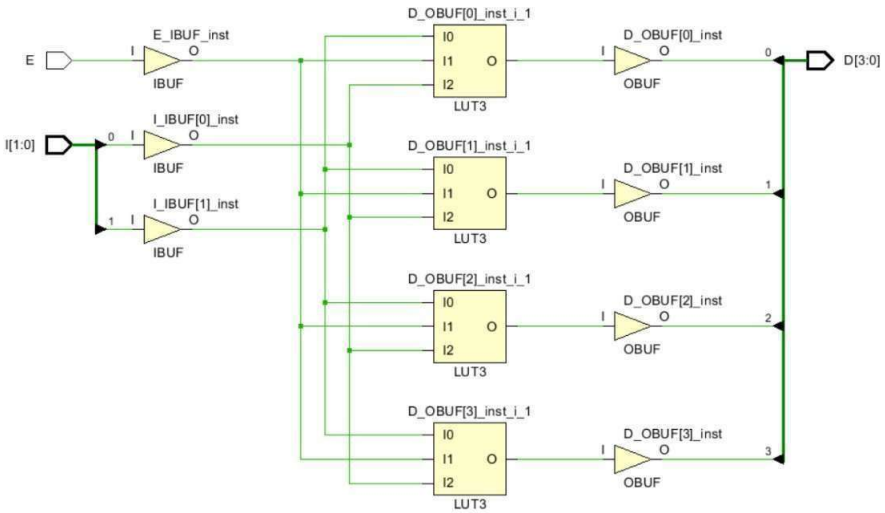
Test Bench:

```
`timescale 1ns / 1ps
module tb_Deco2to4;
        reg [1:0] I;   reg E;          // Inputs
        wire [3:0] D;                  // Outputs
// Instantiate the Unit Under Test (UUT)
Decoder2to4_df uut (.I(I), .E(E), .D(D));
        initial begin
            // Initialize Inputs
                E = 0; I = 2'b00;
           // Add stimulus here
              #10 E = 0; I = 2'b10;
              #10 E = 0; I = 2'b11;
              #10 E = 0; I = 2'b00;
              #10 E = 1; I = 2'b00;
              #10 E = 1; I = 2'b01;
              #10 E = 1; I = 2'b10;
              #10 E = 1; I = 2'b11;
              #10 E = 0; I = 2'b00;
              #10 $finish;
           end
    endmodule
```

RTL Schematic:



Synthesis Schematic:



Simulation waveforms:

(b) 3-to-8 Decoder (Behavioural Modeling):

RTL code :

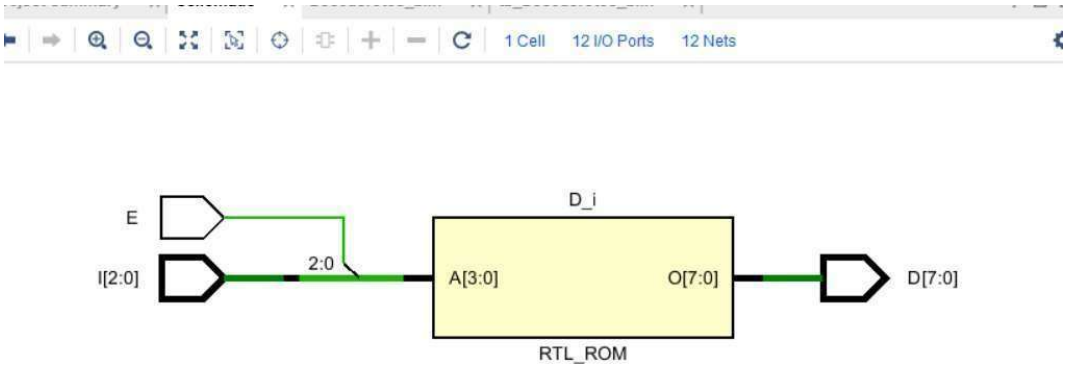
```
module Decoder3to8_bh(I,E,D);
        input [2:0] I;    input E;
        output [7:0] D;

        always @(I, E)
          begin
            case({E,I})
                4'b1000: D = 8'b00000001;
                4'b1001: D = 8'b00000010;
                4'b1010: D = 8'b00000100;
                4'b1011: D = 8'b00001000;
                4'b1100: D = 8'b00010000;
                4'b1101: D = 8'b00100000;
                4'b1110: D = 8'b01000000;
                4'b1111: D = 8'b10000000;
                default: D = 8'b00000000;
            endcase
          end
    endmodule
```

Testbench:

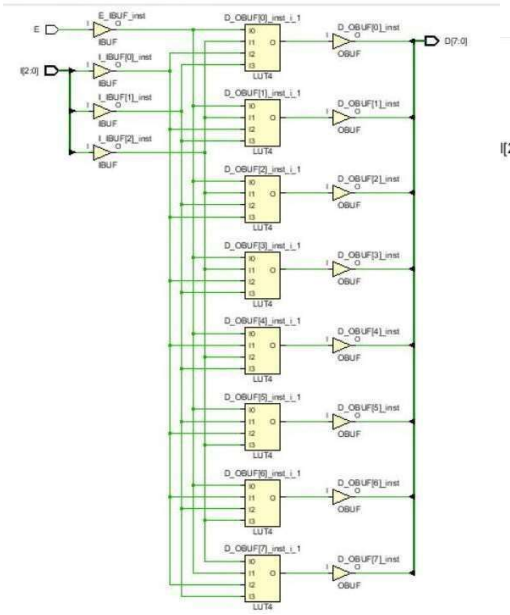
```
`timescale 1ns/1ps  // sets simulation time units

module tb_Decoder3to8;

  // Declare inputs as reg
  reg [2:0] I;
  reg E;

  // Declare outputs as wire
  wire [7:0] D;
```

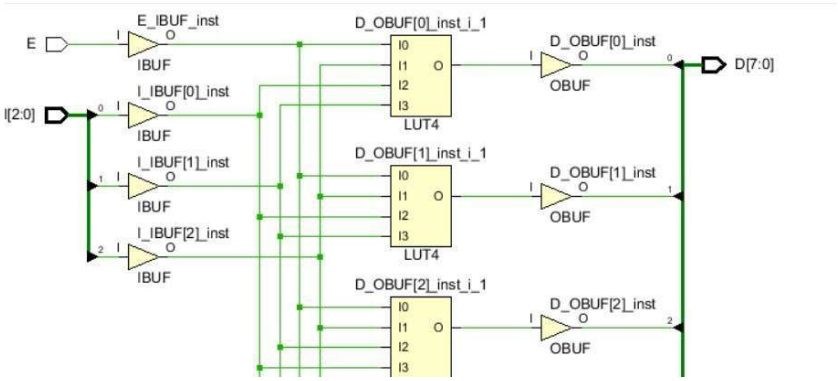
```
  // Instantiate the Unit Under Test (UUT)
  Decoder3to8_bh uut (
    .I(I),
    .E(E),
    .D(D)
  );

  initial begin
    // Monitor outputs
    $monitor("Time=%0t | E=%b | I=%b | D=%b", $time, E, I, D);

    // Test cases
    E = 0; I = 3'b000; #10;  // Output should be 00000000
    E = 1; I = 3'b000; #10;  // Output should be 00000001
    E = 1; I = 3'b001; #10;  // Output should be 00000010
    E = 1; I = 3'b010; #10;  // Output should be 00000100
    E = 1; I = 3'b011; #10;  // Output should be 00001000
    E = 1; I = 3'b100; #10;  // Output should be 00010000
    E = 1; I = 3'b101; #10;  // Output should be 00100000
    E = 1; I = 3'b110; #10;  // Output should be 01000000
    E = 1; I = 3'b111; #10;  // Output should be 10000000
    E = 0; I = 3'b111; #10;   // Output should be 00000000

    // Finish simulation
    $finish;
  end

endmodule
```

RTL Schematic:

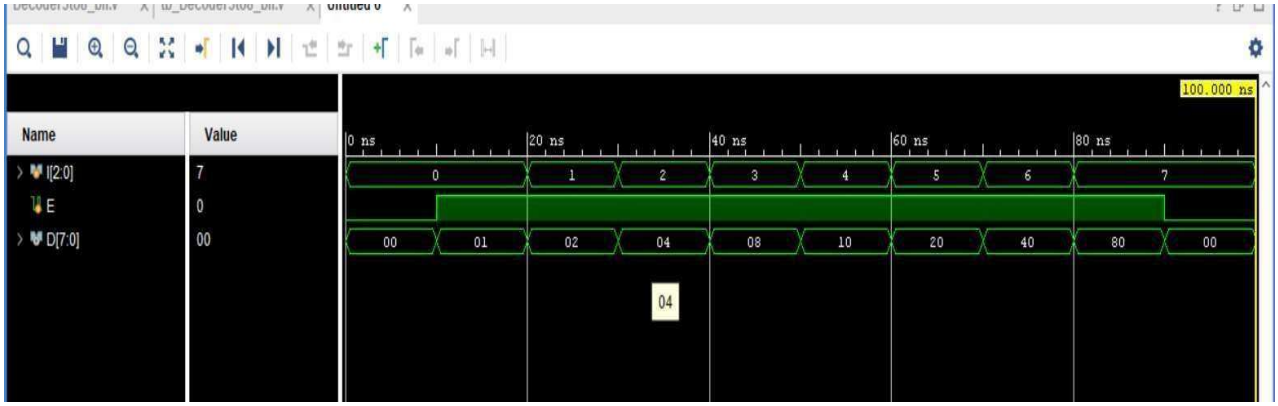


Synthesis Schematic:

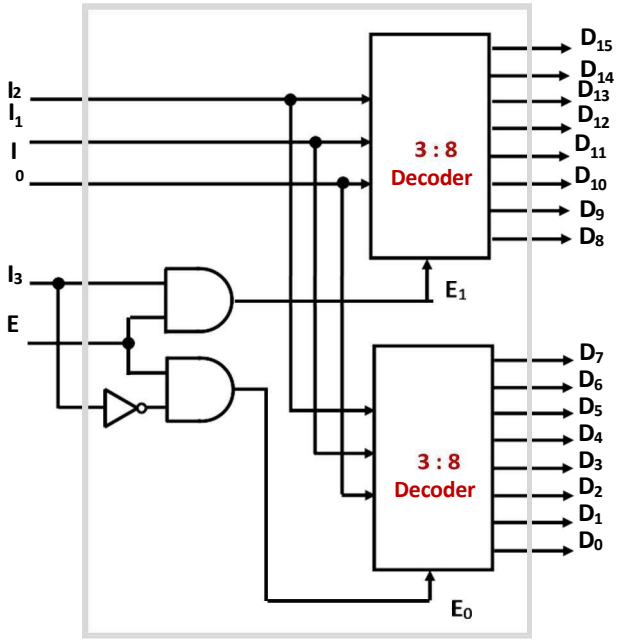




Simulation waveforms:



(c) 4-to-16 Decoder (Structural Modeling using 3-to-8 Decoder): ()



RTL CODE:

Decoder3to8

```
module decoder3to8 (
    input [2:0] a,  // 3-bit input
    input en,       // Enable input
    output [7:0] y  // 8 outputs
);
    assign y = (en) ? (1 << a) : 8'b00000000;
endmodule
```

Decoder4to16

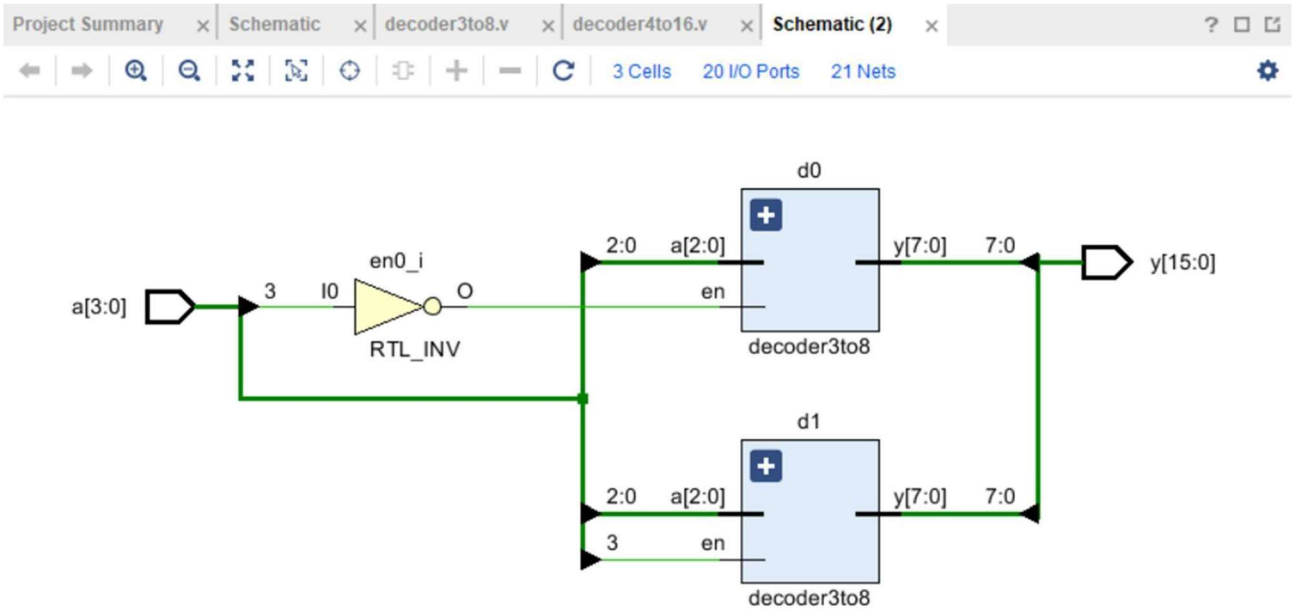
```
module decoder4to16 (
    input [3:0] a,      // 4-bit input
    output [15:0] y     // 16 outputs
);
    wire [7:0] y0, y1;

    // Lower 3-to-8 decoder active when MSB (a[3]) = 0
    decoder3to8 d0 (.a(a[2:0]), .en(~a[3]), .y(y0));

    // Upper 3-to-8 decoder active when MSB (a[3]) = 1
    decoder3to8 d1 (.a(a[2:0]), .en(a[3]), .y(y1));

    // Combine outputs
    assign y = {y1, y0};
endmodule
```

TestBench:

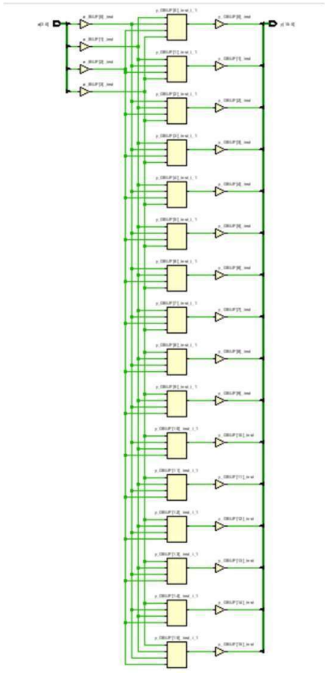
```
`timescale 1ns/1ps

module tb_decoder4to16;
    reg [3:0] a;
    wire [15:0] y;
```

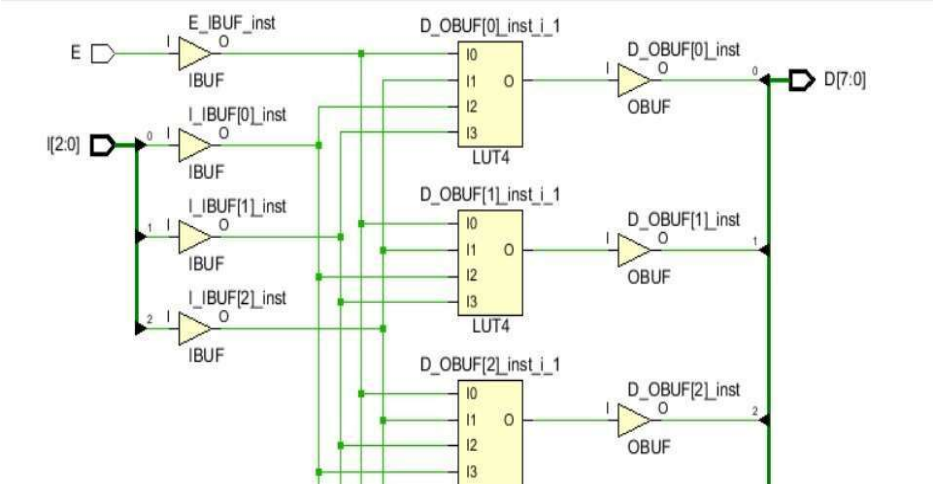
```
    // Instantiate the Unit Under Test (UUT)
    decoder4to16 uut (.a(a), .y(y));

    initial begin
        $monitor("Time=%0d | Input=%b | Output=%b", $time, a, y);

        // Apply all possible inputs
        a = 4'b0000; #10;
        a = 4'b0001; #10;
        a = 4'b0010; #10;
        a = 4'b0011; #10;
        a = 4'b0100; #10;
        a = 4'b0101; #10;
        a = 4'b0110; #10;
        a = 4'b0111; #10;
        a = 4'b1000; #10;
        a = 4'b1001; #10;
        a = 4'b1010; #10;
        a = 4'b1011; #10;
        a = 4'b1100; #10;
        a = 4'b1101; #10;
        a = 4'b1110; #10;
        a = 4'b1111; #10;

        $stop;
    end
endmodule
```

RTL Schematic:

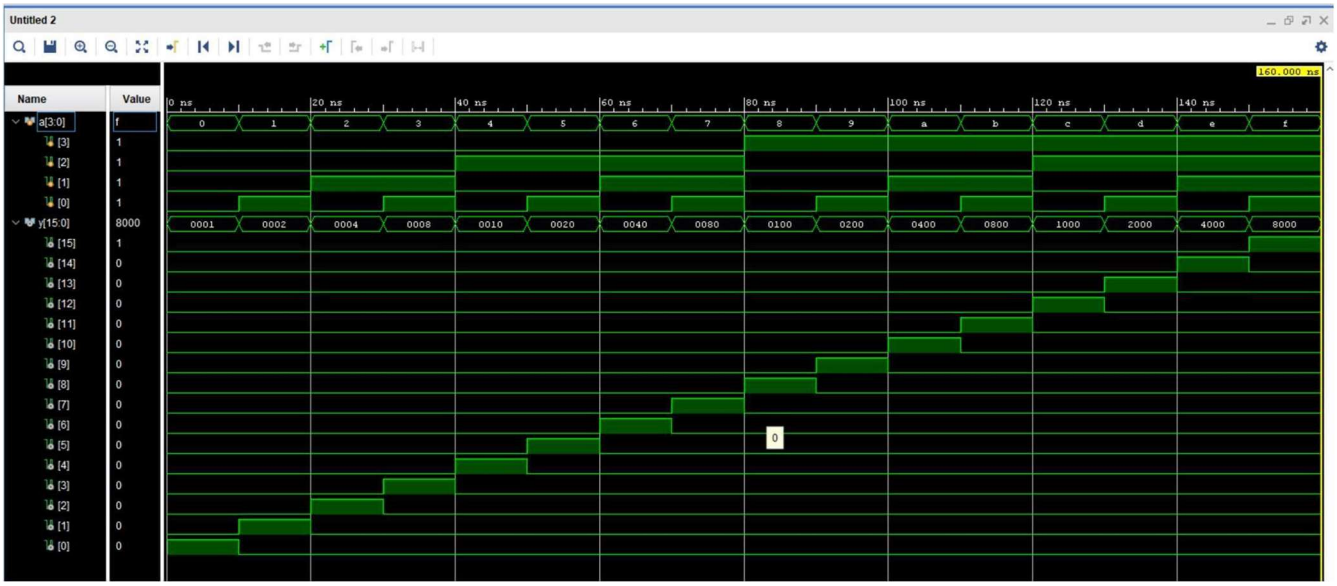


Synthesis Schematic:





Simulation waveforms:



Conclusion: Various decoders were designed using dataflow, behavioral, and structural modeling. The simulated results matched theoretical truth tables, validating the designs. This experiment demonstrated how decoders activate specific outputs for given binary inputs and are essential in memory addressing and instruction decoding.

Suggested Reference:

References used by the students:

Rubric wise marks obtained:

Rubrics	1	2	3	4	5	Total
Marks						

Experiment No: 9

Date: _____

Aim: Design 4:2, 8:3 Priority Encoder using Verilog / VHDL.

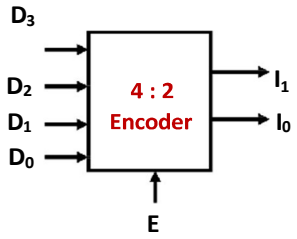
Competency and Practical Skills:

Relevant CO:

Objectives:

Equipment / Instruments: Laptop or Computer with Xilinx / .

Basic Theory: An Encoder is a combinational circuit that performs the reverse operation of Decoder. It has maximum of 2^n input lines and 'n' output lines. It will produce a binary code depending on the input line activated.



Truth Table of 4-to-2 Encoder

Enable	Inputs				Outputs	
E	D_3	D_2	D_1	D_0	I_1	I_0
0	X	X	X	X	X	X
1	1	0	0	0	1	1
1	0	1	0	0	1	0
1	0	0	1	0	0	1
1	0	0	0	1	0	0

Priority Encoder: A priority encoder produces correct code at the output even when multiple lines at the inputs are simultaneously active high (logic '1'). As shown above, the 4-to-2 priority encoder has four inputs (D_3, D_2, D_1, D_0) and two outputs (I_1 and I_0). Here, the input, D_3 has the highest priority; whereas, the input, D_0 has the lowest priority. In this case, even if more than one input lines are at logic '1' at the same time, the output will be the binary code corresponding to the input, which is having higher priority.

(a) 4 : 2 Priority Encoder

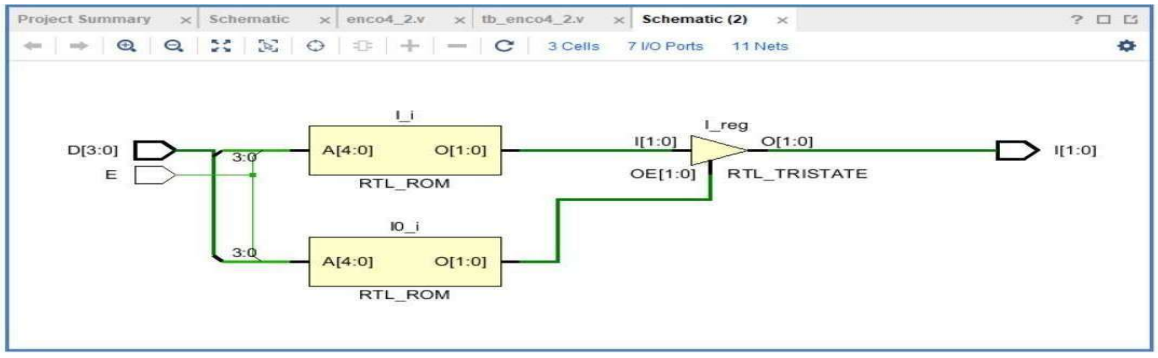
RTL Code:

```
module Encoder4to2_bh(D,E,I);
        input [3:0] D;
        input E;
        output reg [1:0] I;

        always @(D,E)
          begin
                casex({E,D})
                      5'b11xxx: I = 2'b11;
                      5'b101xx: I = 2'b10;
                      5'b1001x: I = 2'b01;
                      5'b10001: I = 2'b00;
                      default: I = 2'bzz;
                endcase
          end
endmodule
```

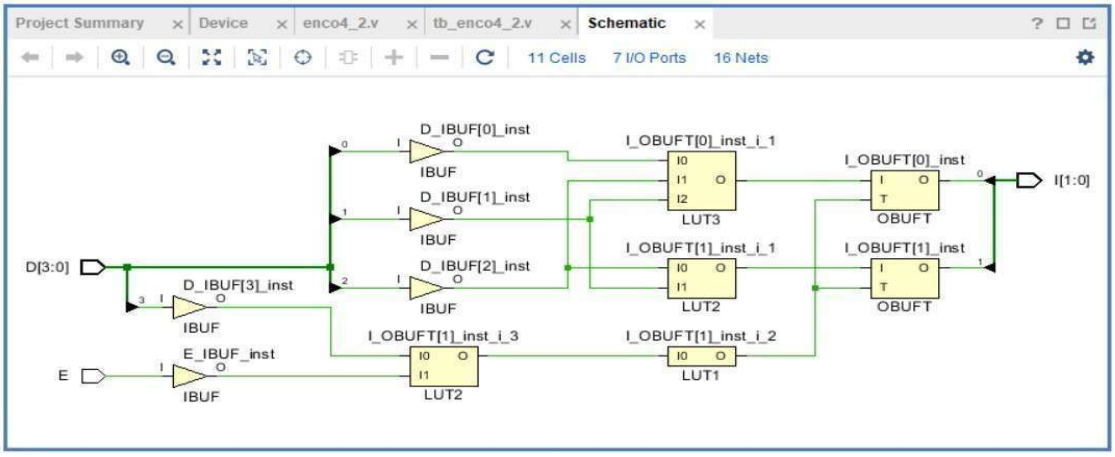
Testbench Code:

```
`timescale 1ns / 1ps
module tb_enco4_2;
    reg [3:0] D;          // 4-bit input
    reg E;                // Enable
    wire [1:0] I;         // 2-bit output
    // Instantiate the Encoder
    enco4_2 uut (.D(D), .E(E), .I(I));
    initial begin
        $monitor("Time=%0t | E=%b | D=%b | I=%b", $time, E, D, I);
        // Test with E = 0 (disabled)
        E = 0; D = 4'b0000; #10;
        D = 4'b1000; #10;
        // Test with E = 1 (enabled)
        E = 1; D = 4'b0000; #10;  // No input active → I = zz
        D = 4'b0001; #10;         // D[0] → I=00
        D = 4'b0010; #10;         // D[1] → I=01
        D = 4'b0100; #10;         // D[2] → I=10
        D = 4'b1000; #10;         // D[3] → I=11
        // Test multiple inputs active (priority)
        D = 4'b1100; #10;         // D[3] & D[2] → I=11 (highest priority)
        D = 4'b1010; #10;        // D[3] & D[1] → I=11
        D = 4'b0110; #10;         // D[2] & D[1] → I=10 (D[2] priority)
        $finish;
    end
endmodule
```

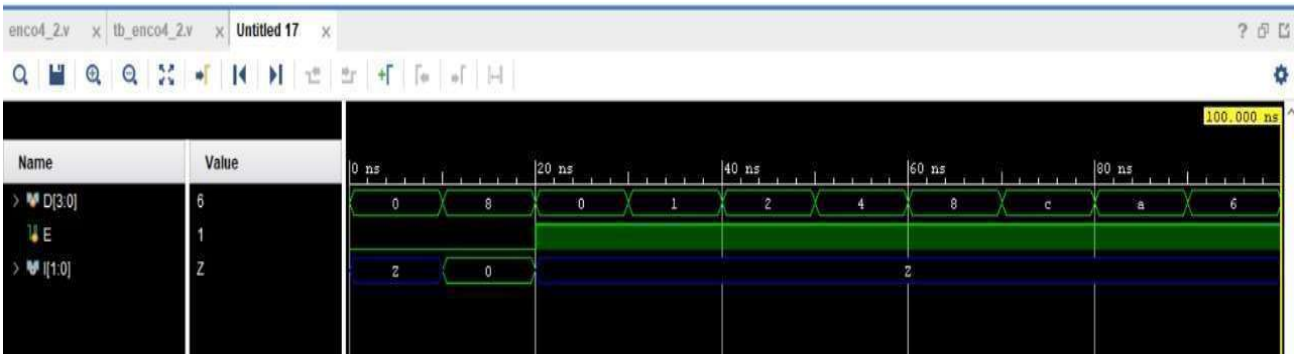
RTL Schematic:



Synthesis Schematic:



Simulation waveforms:



(b) 8:3 Priority Encoder

RTL Code

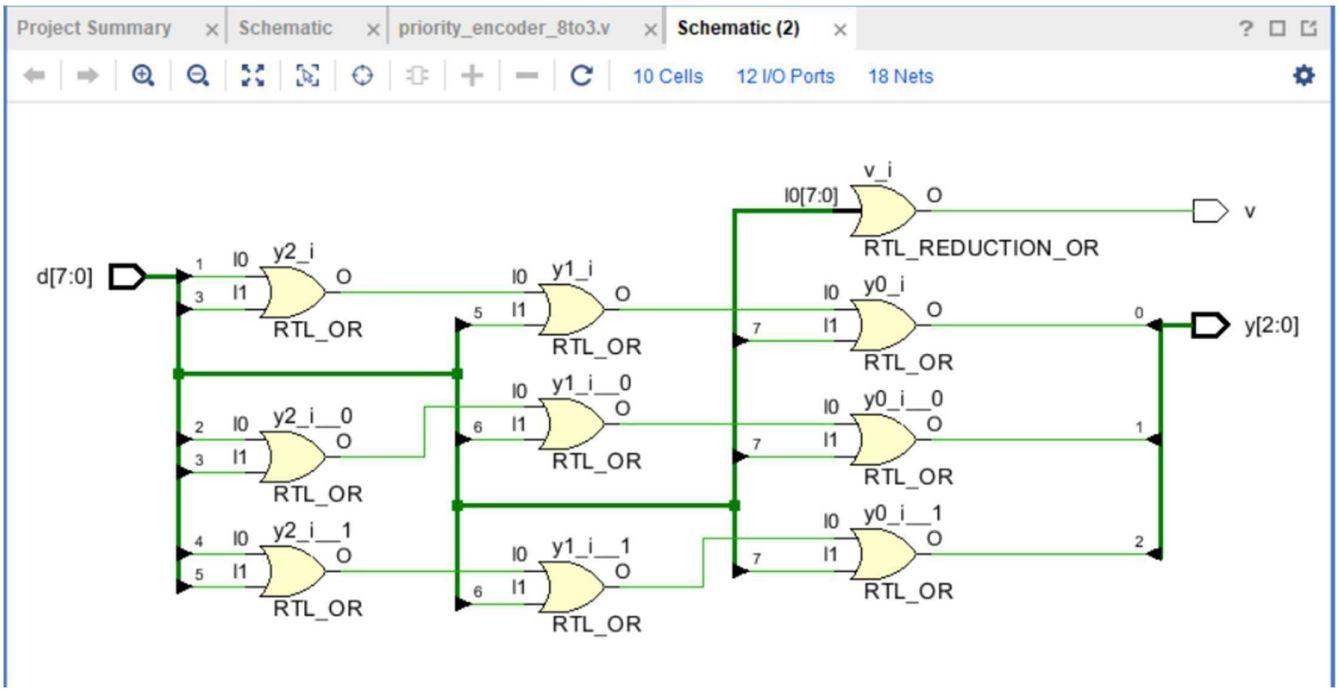
```
module priority_encoder_8to3 (
    input  [7:0] d,      // Inputs D7..D0
    output [2:0] y,      // Encoded output
    output v             // Valid output
);
    // Valid = OR of all inputs
    assign v = |d;
    // Encoded outputs (priority: D7 highest, D0 lowest)
    assign y[2] = d[4] | d[5] | d[6] | d[7];
    assign y[1] = d[2] | d[3] | d[6] | d[7];
    assign y[0] = d[1] | d[3] | d[5] | d[7];

endmodule
```

Test Bench

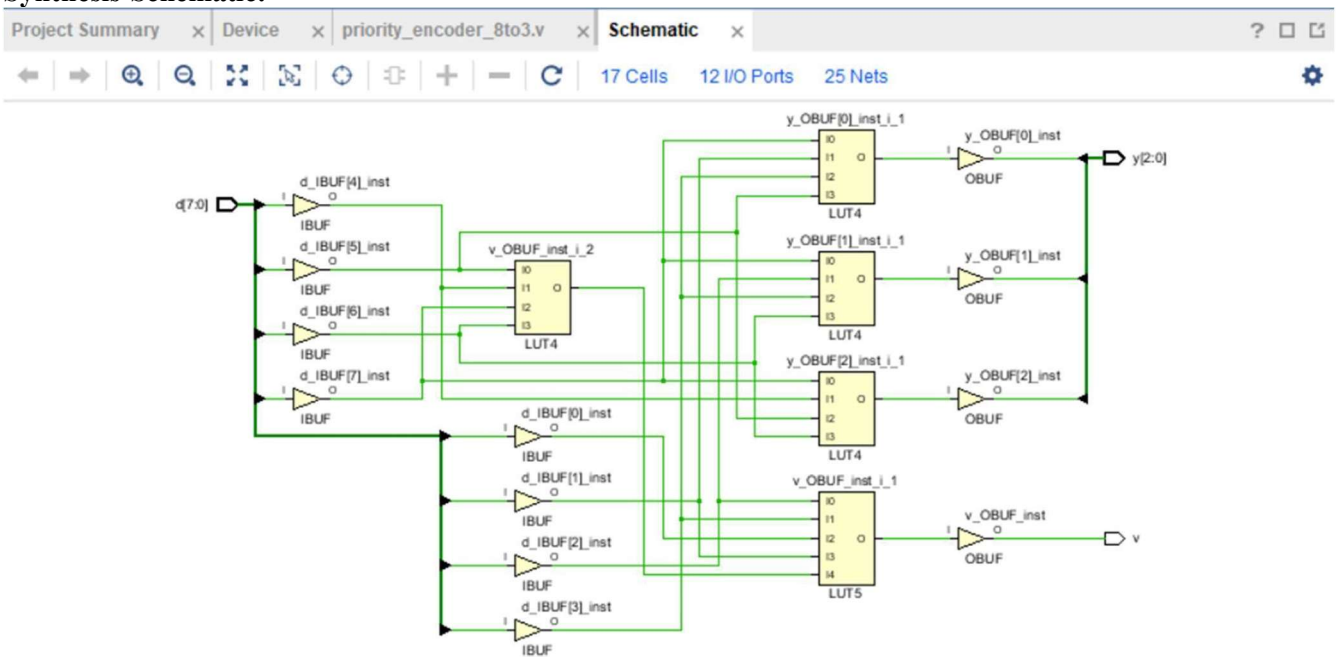
```
`timescale 1ns/1ps

module tb_priority_encoder_8to3;

    // Testbench signals
    reg [7:0] d;
    wire [2:0] y;
    wire v;

    // Instantiate DUT
    priority_encoder_8to3 uut (
        .d(d),
        .y(y),
        .v(v)
    );

    initial begin
        $monitor("Time=%0t | d=%b | y=%b | v=%b", $time, d, y, v);

        d = 8'b00000000; #10;  // No input
        d = 8'b00000001; #10;  // D0 active
```

```
        d = 8'b00000100; #10;  // D2 active
        d = 8'b00010000; #10;  // D4 active
        d = 8'b01000000; #10;  // D6 active
        d = 8'b10000000; #10;  // D7 active
        d = 8'b10101010; #10;  // Multiple inputs, D7 priority
        $stop;
    end

endmodule
```

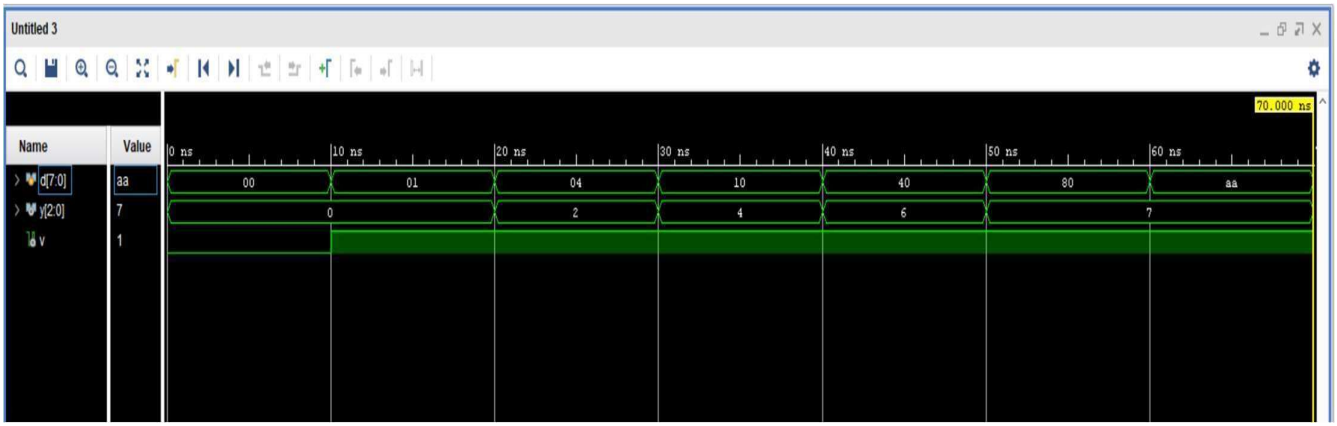
RTL Schematic:



Synthesis Schematic:



Simulation waveforms:



Conclusion: Priority encoders were designed and tested successfully. Simulation verified correct output even with multiple active inputs, giving priority to higher-order signals. This experiment showed how priority encoders resolve conflicts in digital systems and are vital in interrupt handling and resource allocation.

Suggested Reference:

References used by the students:

Rubric wise marks obtained:

Rubrics	1	2	3	4	5	Total
Marks						

Experiment No: 10

Date: _____

Aim: Design 4-bit Comparator using Verilog / VHDL.

Competency and Practical Skills:

Relevant CO:

Objectives:

Equipment / Instruments: Laptop or Computer with Xilinx / Altera (Intel)Tools.

Basic Theory:

Program Code:

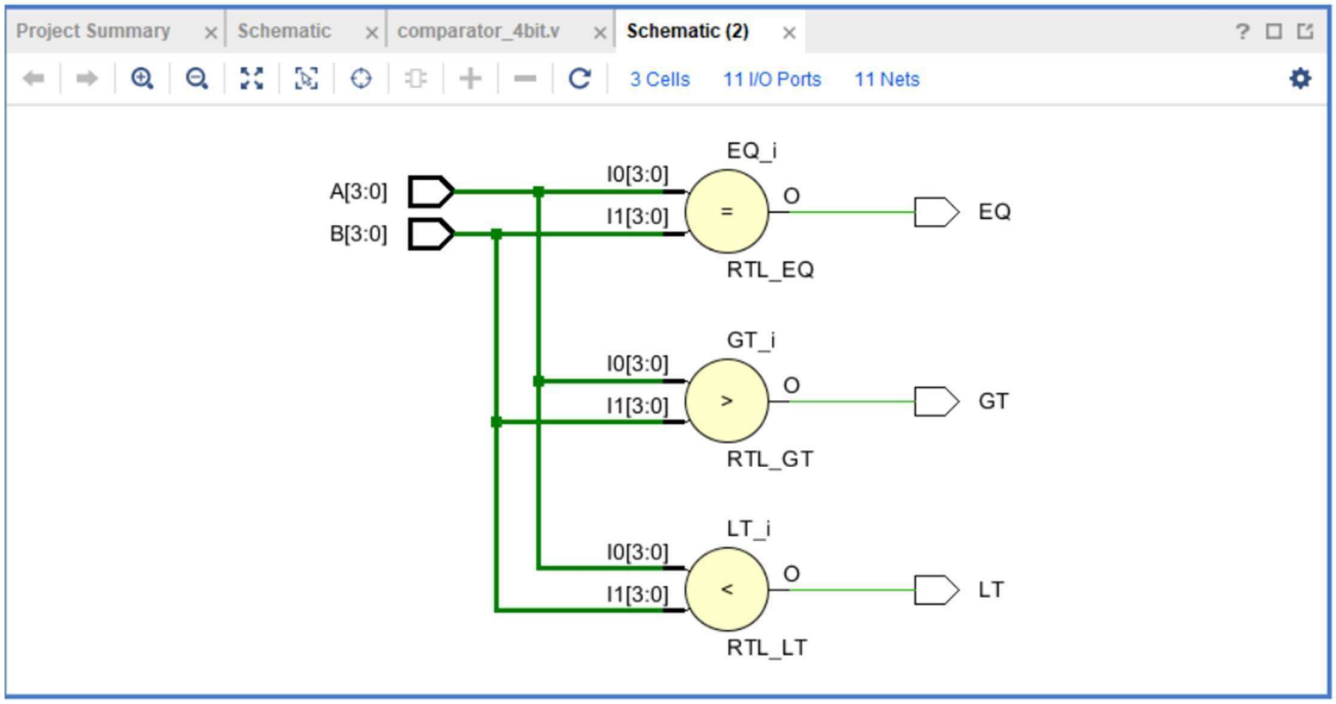
```
module comparator_4bit (
   input  [3:0] A, B,
   output EQ, GT, LT
);

   // Equal if all bits match
   assign EQ = (A == B);

   // Greater if A > B
   assign GT = (A > B);

   // Less if A < B
   assign LT = (A < B);

endmodule
```

Test Bench:

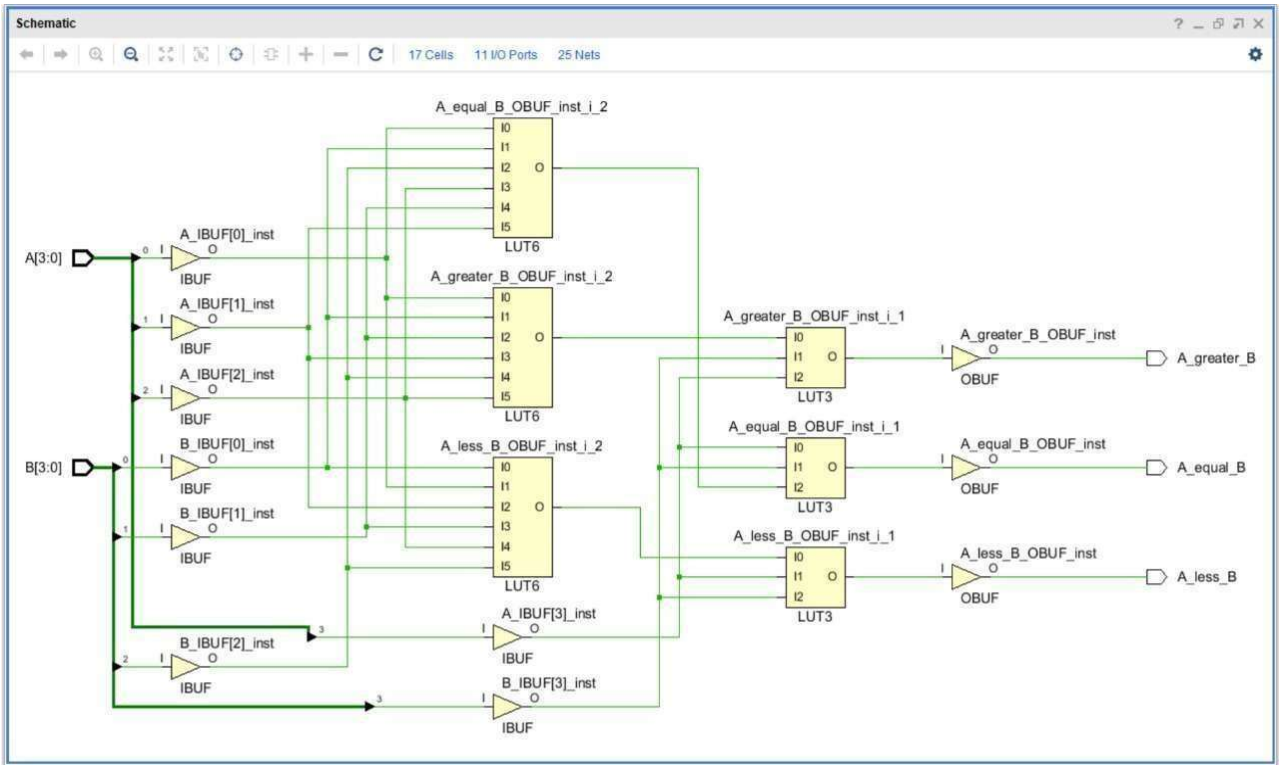
```
`timescale 1ns/1ps

module tb_comparator_4bit;
   reg [3:0] A, B;
   wire EQ, GT, LT;

   // Instantiate DUT
   comparator_4bit uut (.A(A), .B(B), .EQ(EQ), .GT(GT), .LT(LT));

   initial begin
      $monitor("Time=%0t | A=%b | B=%b | EQ=%b | GT=%b | LT=%b",
            $time, A, B, EQ, GT, LT);

      // Test cases
      A = 4'b0000; B = 4'b0000; #10;  // Equal
      A = 4'b1010; B = 4'b0110; #10;  // Greater
      A = 4'b0101; B = 4'b1110; #10;  // Less
```

```
        A = 4'b1111; B = 4'b1111; #10;  // Equal
        A = 4'b0011; B = 4'b0010; #10;  // Greater
        A = 4'b1000; B = 4'b1111; #10;  // Less
        $stop;
    end
endmodule
```

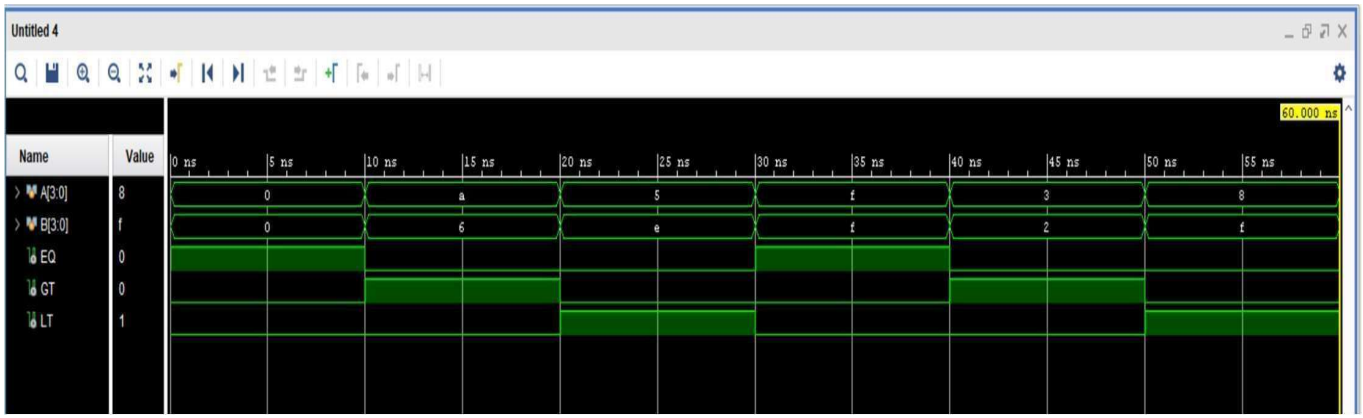
RTL Schematic:



Synthesis Schematic:



Output waveform:



Conclusion: A 4-bit comparator was designed and simulated using Verilog. The outputs correctly indicated greater-than, less-than, and equality conditions. The experiment verified the role of comparators in digital systems where magnitude comparison is needed, such as sorting circuits and arithmetic units.

Quiz:

Suggested Reference:

References used by the students:

Rubric wise marks obtained:

Rubrics	1	2	3	4	5	Total
Marks						

Experiment : 16

Date: _____

Aim : Introduction to LT Spice Tool

Competency and Practical Skills:

Relevant CO: CO 3 , CO 5

Objectives:

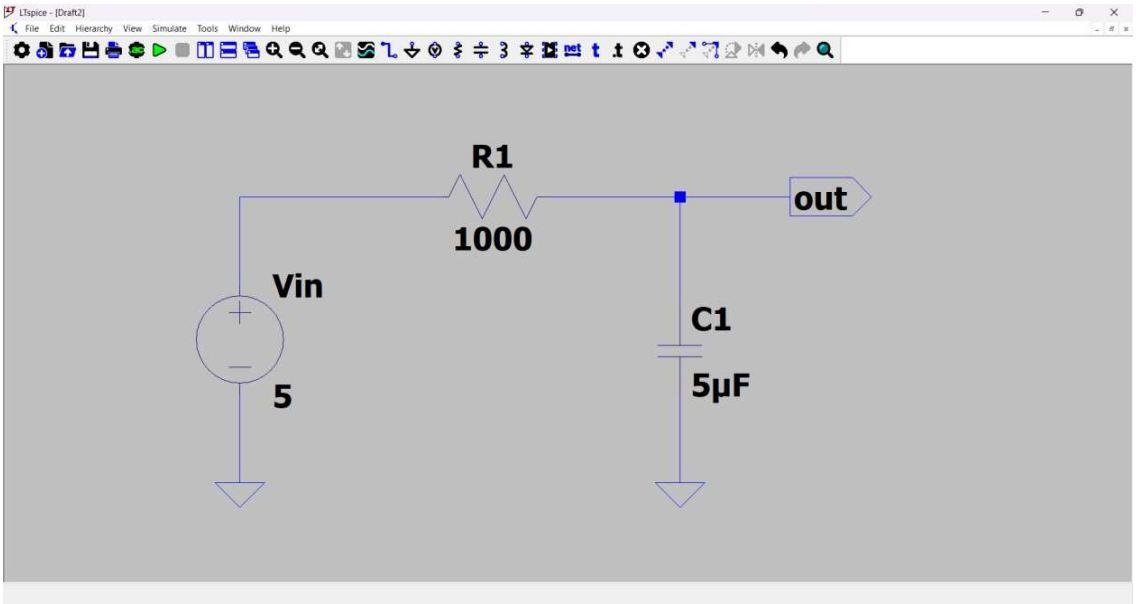
Equipment / Instruments: Laptop or Computer with LT Spice tool installed .

Introduction:

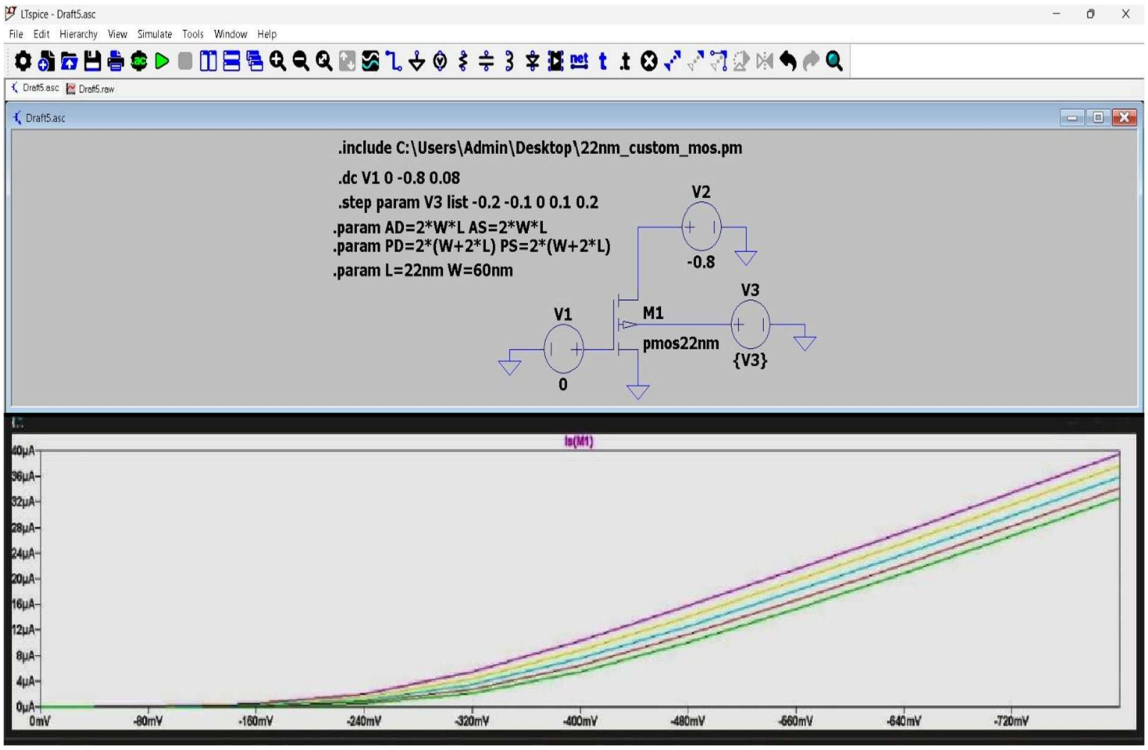
LT SPICE Tool

LTspice is a high-performance SPICE-based analog and digital circuit simulator developed by Analog Devices. It provides a **comprehensive environment for designing, simulating, and analyzing circuits**, including CMOS logic gates, resistive-load circuits, amplifiers, and mixed-signal designs. LTspice is widely used in both academia and industry to validate designs before fabrication, helping reduce errors and improve efficiency.

LTspice allows designers to simulate **circuit behavior at the electrical level**, perform various analyses, and generate graphical waveforms. Unlike Microwind, which emphasizes **layout and physical design**, LTspice focuses on **accurate electrical simulation and behavioral modeling**, but it also supports hierarchical circuit design through sub-circuits, akin to a logical schematic editor.



LT SPICE User Interface



Key Features of LTspice :

1. **Schematic Editor and Simulation Environment**

LTspice provides a **user-friendly schematic editor** comparable to Microwind's DSCH editor for logical design:

Rapid circuit design: Place components and connect nodes with wires.
Mouse-driven intuitive interface: Click-and-drag placement, zooming, and labeling for easy circuit editing.
Hierarchical design support: Create sub-circuits and reuse modules for inverters, NAND/NOR gates, and complex logic networks.
Netlist extraction: Automatically generates a SPICE netlist compatible with PSpice, WinSpice, and other SPICE-based simulators.

Analysis capabilities:

- DC, AC, and transient analysis.
- Measurement of voltage, current, and power at any node.
- Parametric sweeps to study circuit response over variable component values.

Simulation of CMOS devices: Using built-in NMOS/PMOS models with adjustable W/L ratios.

Immediate access to properties: Observe node voltages, probe currents, and analyze timing delays of logic gates.

Symbol library: Large library of standard components and user-defined symbols.

Parasitic modeling: Include capacitance, resistance, and crosstalk in device models for accurate simulations.

2. **CMOS and Semiconductor Device Libraries**

Built-in models for **NMOS and PMOS transistors**, BJTs, diodes, voltage/current sources, and passive components.

Device parameters include channel length and width, threshold voltage, and parasitic capacitances, allowing **accurate CMOS circuit modeling**.

Users can define **custom MOSFET models** to simulate different fabrication technologies, analogous to Microwind's CMOS libraries (Cmos12.rul, Cmos08.rul, etc.).

Supports **parameterized device definitions**, making it easy to simulate scaling, sizing, and technology variations

.

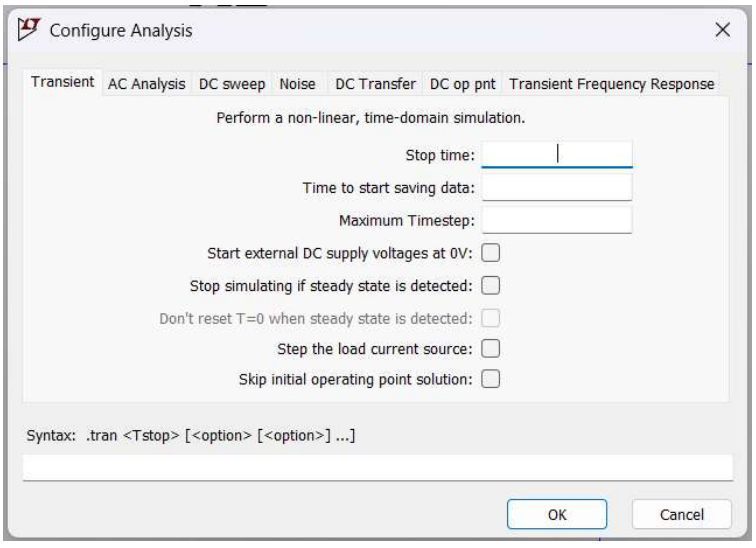
3. **Simulation Types**

DC Analysis: Determines voltage and current distribution at steady state.

Transient Analysis: Observes time-domain switching behavior of logic gates, inverters, and other circuits.

AC Analysis: Frequency response for small-signal circuits.

Parametric and Noise Analysis: Evaluate performance over varying component values and quantify noise effects.

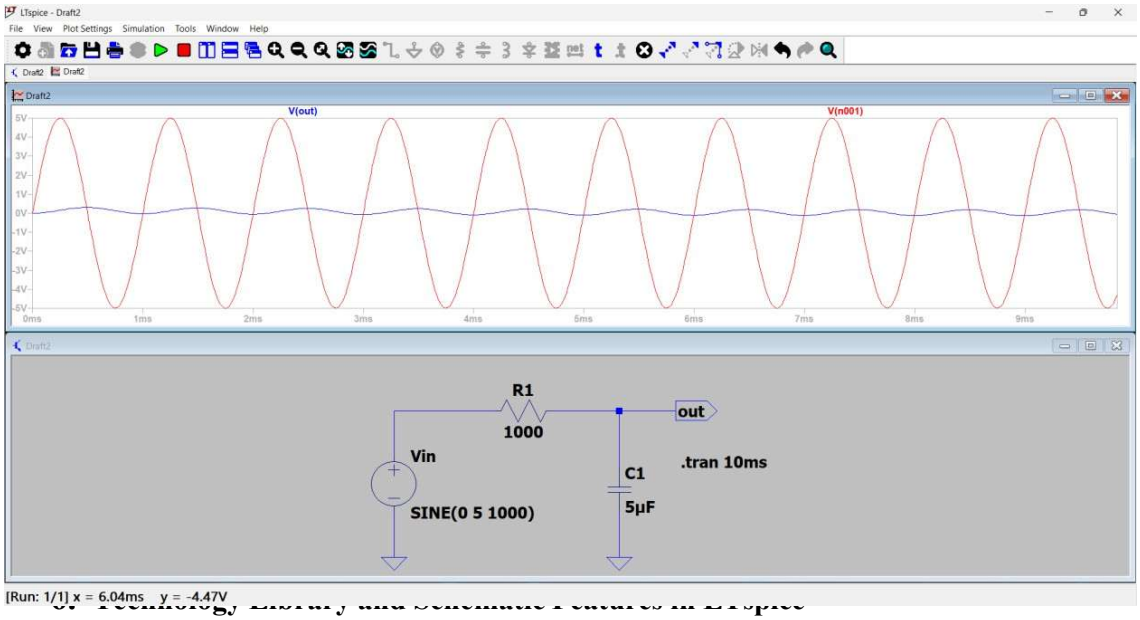


4. Sub-circuit and Hierarchical Design

Allows creation of **reusable modules** for CMOS inverters, NAND, NOR, and more. Facilitates **modular circuit design** similar to Microwind's hierarchical layouts and symbol-based designs.

5. Waveform Viewer

Built-in plotting tool for **voltage, current, and power** at any node. Multiple signals can be overlaid and measured, enabling detailed analysis of logic behavior. Supports labeling nodes and exporting plots for reports, making documentation easier.



6. Technology Library and Schematic Features in LTspice

LTspice comes with an **extensive built-in library of semiconductor devices and passive components**, suitable for analog, digital, and mixed-signal circuits. While it doesn't provide layout-based CMOS rules like Microwind, it allows modeling of **sub-micron to nanoscale MOSFETs** via adjustable transistor parameters: channel length (L), width (W), threshold voltage (Vth), and parasitic capacitances.

Device Model	Approx. Feature Size (for simulation purpose)
NMOS / PMOS default	1.2 µm – 0.05 µm (adjustable W/L)
Custom MOSFET	User-defined W/L and model parameters
Diodes / BJTs	Device-level parameters for analog/digital simulation
Passive Components (R, C, L)	N/A

Sub-circuit support: LTspice allows creating **modular sub-circuits**, similar to hierarchical CMOS cells in Microwind.

Custom models: Users can import third-party SPICE models to simulate advanced technologies, including deep-submicron CMOS devices.

Parameter scaling: Channel lengths and widths can be scaled to emulate **different fabrication technologies**, down to ~50 nm feature sizes.

7. Advantages of LTspice

1. Lightweight, fast, and freely available.
2. Supports **large circuits with thousands of components**.
3. Immediate feedback through waveform plots and measurements.
4. Facilitates learning and verification of **CMOS and logic circuits**, bridging conceptual design and practical simulation.
5. Can generate **netlists compatible with other SPICE tools** for advanced analysis.

➢ **Conclusion :** This experiment provided an introduction to LTspice, a SPICE based circuit simulation tool, highlighting its schematic editor, hierarchical design capabilities, component libraries, and simulation environment. By exploring device models, sub-circuits, and waveform analysis, the experiment familiarized us with designing, simulating, and analyzing analog and digital circuits effectively. LTspice enables observation of voltage, current, and logic behavior in circuits before implementation, supporting sub-micron to nanoscale CMOS simulation through adjustable device parameters. Overall, it establishes the foundation for subsequent CMOS inverter and logic gate experiments, emphasizing accurate modeling and efficient circuit verification.

Suggested Reference :

1. LTspice user guide and manuals.
2. https://web.mit.edu/6.101/www/s2020/handouts/LTSpiceIntro.pdf
3. https://www.analog.com/en/resources/media-center/videos/series/ltspice-essentials-tutorial.html
4. *The LTspice IV Simulator: Users Guide and Reference* – by Kenneth Kundert
5. *Design of Analog CMOS Integrated Circuits* – by Behzad Razavi
6. *CMOS Digital Integrated Circuits: Analysis and Design* – by Sung-Mo Kang and Yusuf Leblebici

References Used by Students :

1. *Design of Analog CMOS Integrated Circuits* – by Behzad Razavi
2. https://www.analog.com/en/resources/media-center/videos/series/ltspice-essentials-tutorial.html

Rubric wise marks obtained:

Rubrics	1	2	3	4	5	Total
Marks						

Experiment : 17

Date: _____

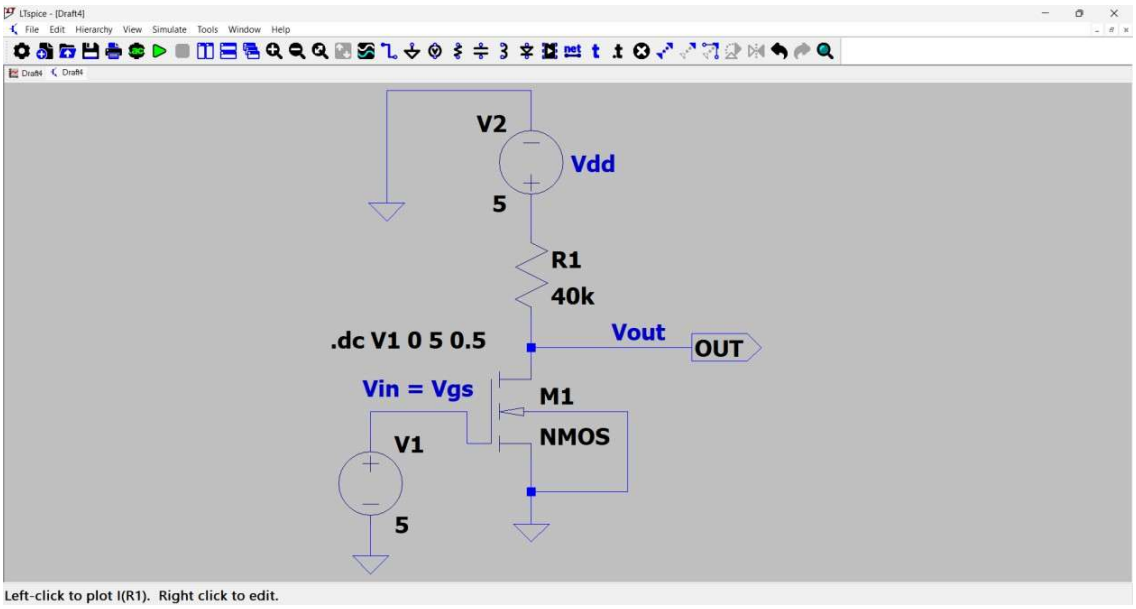
Aim: Implementation of Resistive load and CMOS inverters using LT SPICE

Competency and Practical Skills:

Relevant CO: CO 1 , CO 3

Equipment / Instruments: Laptop or Computer with LT SPICE Tool.

- *Resistive Load Inverter Layout (Using LT SPICE) :*

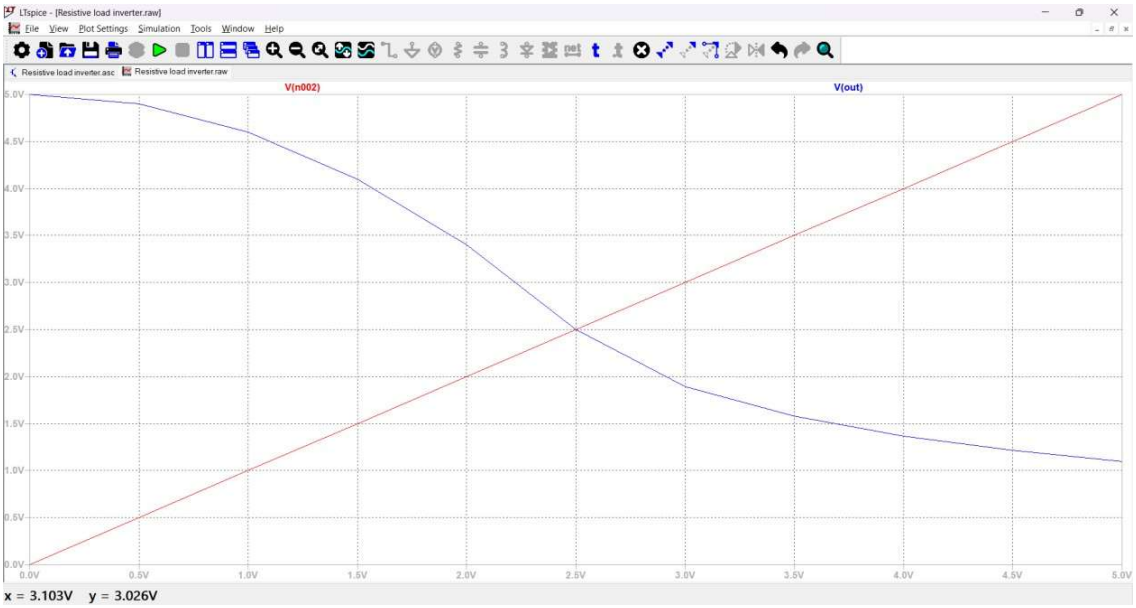


Resistive Load Inverter => Circuit Schematic

```
* D:\Work_Study\230170111140 - VLSI\P_17_1\Resistive load inverter.asc
* Generated by LTspice 24.1.9 for Windows.
M1 OUT N002 0 0 NMOS
V1 N002 0 5
V2 N001 0 5
R1 N001 OUT 40k
.model NMOS NMOS
.model PMOS PMOS
.lib C:\Users\Admin\AppData\Local\LTspice\lib\cmp\standard.mos
* Vout
* Vdd
* Vin = Vgs
.dc V1 0 5 0.5
.backanno
.end
```

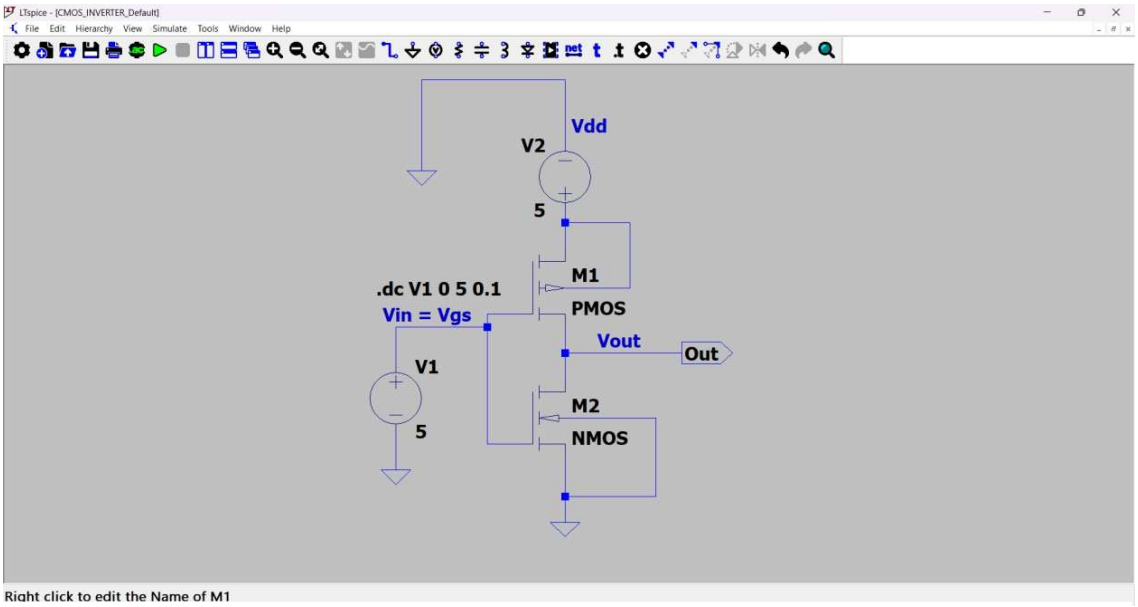
Resistive Load Inverter => SPICE NETLIST

- Simulation Waveforms :



Resistive Load Inverter => Voltage Graph (Simulation)

- *<u>CMOS Inverter Layout (Using LT SPICE) :</u>*

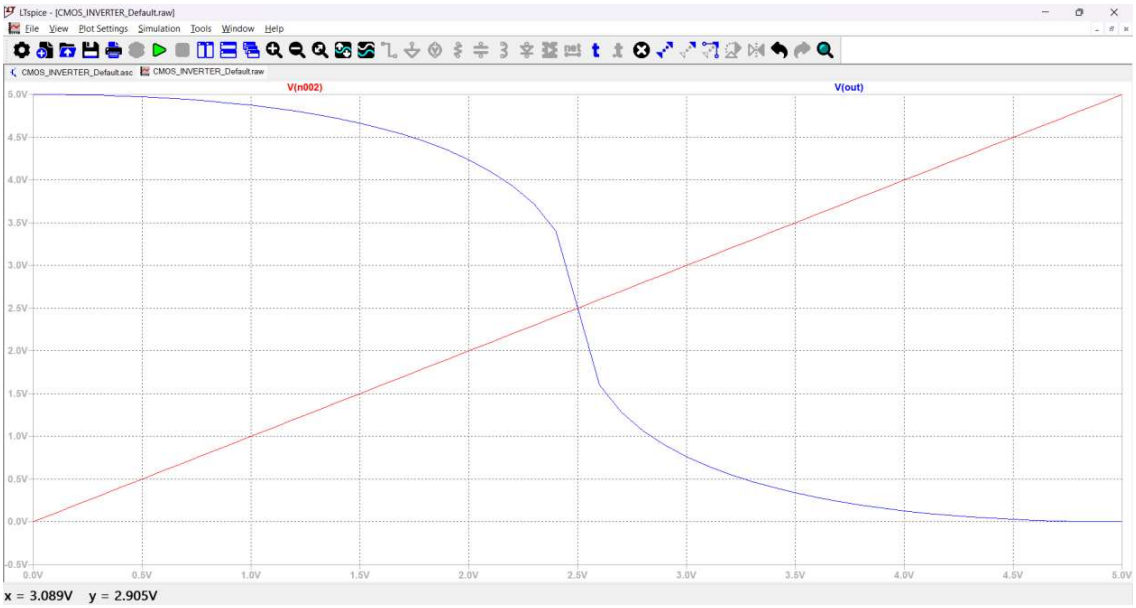


CMOS INVERTER (Deafult Lib) => Circuit Schematic

```
* D:\Work_Study\230170111140 - VLSI\P_17_1\CMOS_INVERTER_Default.asc
* Generated by LTspice 24.1.9 for Windows.
M1 N001 N002 Out N001 PMOS
M2 Out N002 0 0 NMOS
V1 N002 0 5
V2 N001 0 5
.model NMOS NMOS
.model PMOS PMOS
.lib C:\Users\Admin\AppData\Local\LTspice\lib\cmp\standard.mos
* Vout
* Vdd
* Vin = Vgs
.dc V1 0 5 0.1
.backanno
.end
```

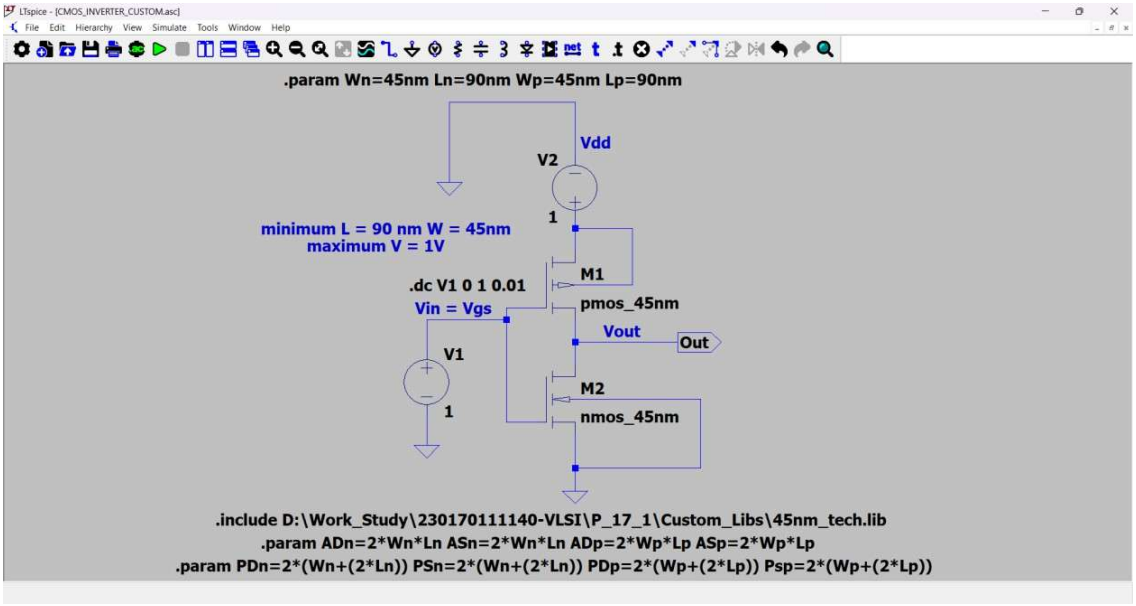
CMOS INVERTER (Deafult Lib) => SPICE NETLIST

- Simulation Waveforms :



CMOS INVERTER (Deafult Lib) => Voltage Graph (Simulation)

➢ *CMOS INVETER Layout (LT SPICE) (Custom Lib) :*



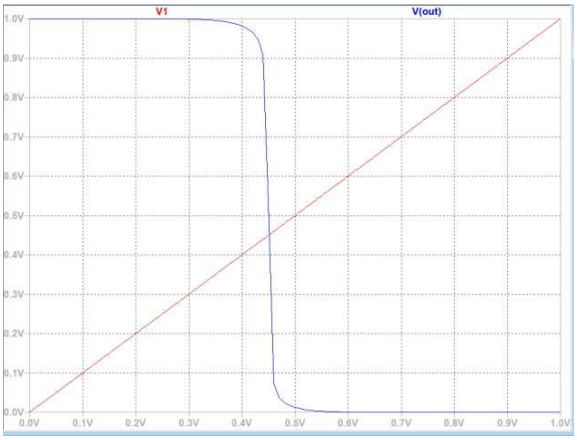
CMOS INVERTER (45nm_tech.lib) => Circuit Schematic

```
* D:\Work_Study\230170111140-VLSI\P_17_1\CMOS_INVERTER_CUSTOM.asc
* Generated by LTspice 24.1.9 for Windows.
M1 N001 N002 Out N001 pmos_45nm l={Lp} w={Wp} ad={ADp} as={ASp} pd={PDp} ps={PSp}
M2 Out N002 0 0 nmos_45nm l={Ln} w={Wn} ad={ADn} as={ASn} pd={PDn} ps={PSn}
V1 N002 0 1
V2 N001 0 1
.model NMOS NMOS
.model PMOS PMOS
.lib C:\Users\Admin\AppData\Local\LTspice\lib\cmp\standard.mos
* Vout
* Vdd
* Vin = Vgs
.dc V1 0 1 0.01
.include  D:\Work_Study\230170111140-VLSI\P_17_1\Custom_Libs\45nm_tech.lib
.param ADn=2*Wn*Ln ASn=2*Wn*Ln ADp=2*Wp*Lp ASp=2*Wp*Lp
.param PDn=2*(Wn+(2*Ln)) PSn=2*(Wn+(2*Ln)) PDp=2*(Wp+(2*Lp)) Psp=2*(Wp+(2*Lp))
.param Wn=45nm Ln=90nm Wp=45nm Lp=90nm
* minimum L = 90 nm W = 45nm
* maximum V = 1V
.backanno
.end
```

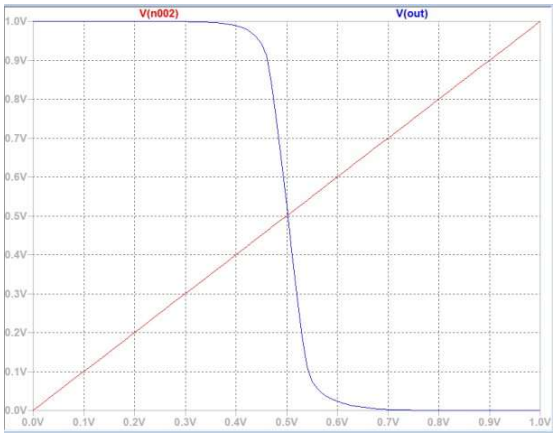
CMOS INVERTER (45nm_tech.lib) => SPICE NETLIST

- Simulation Graphs :

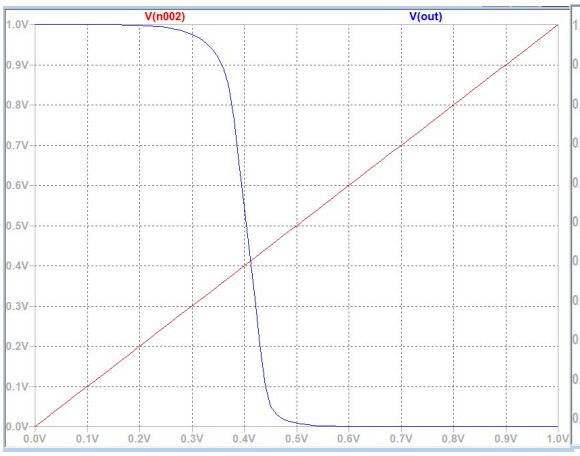
CMOS INVERTER (45nm_tech.lib) => Voltage Graph (Simulation)



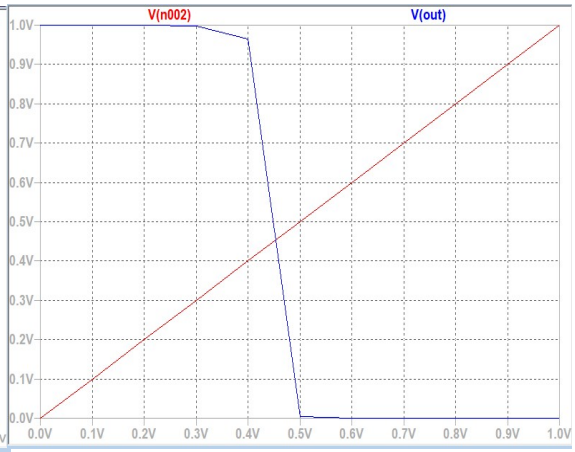
→ Wp = Wn = 45nm | Ln= Ln = 90nm



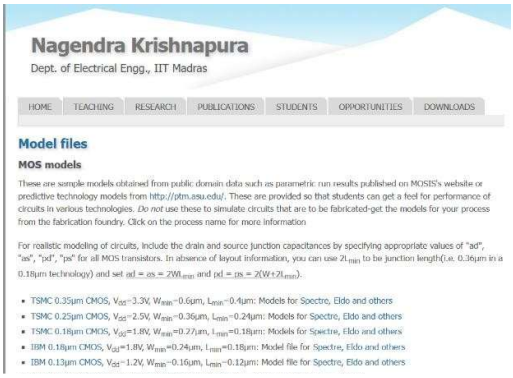
→ Wp = 22.5nm , Lp = 45nm
→ Wn = 45nm , Ln = 90nm



→ Wp = 45nm , Lp = 90nm
→ Wn = 22.5nm , Ln = 45nm



→ Wp = 45nm , Lp = 180nm
→ Wn = 22.5nm , Ln = 180nm



For using Custom Libraries : Find a library data on internet with all parameters like on this https://www.ee.iitm.ac.in/~nagendra/cadinfo.html

Rename the model name in lib to avoid conflict

Conclusion : The experiment on resistive load inverter and CMOS inverter using LT Spice verified the basic principle of logic inversion. The resistive load inverter successfully inverted the input signal but showed static power dissipation and limited voltage swing. On the other hand, the CMOS inverter produced a full rail-to-rail output with very low static power consumption. The transfer characteristics highlighted better switching behavior and noise margin in CMOS design. Thus, CMOS inverters are more power-efficient, reliable, and widely used in modern digital circuits compared to resistive load inverters.

Suggested Reference :

1. LTspice user guide and manuals.
2. https://web.mit.edu/6.101/www/s2020/handouts/LTSpiceIntro.pdf
3. https://www.analog.com/en/resources/media-center/videos/series/ltspice-essentials-tutorial.html
4. *The LTspice IV Simulator: Users Guide and Reference* – by Kenneth Kundert
5. *Design of Analog CMOS Integrated Circuits* – by Behzad Razavi
6. *CMOS Digital Integrated Circuits: Analysis and Design* – by Sung-Mo Kang and Yusuf Leblebici
7. https://www.ee.iitm.ac.in/~nagendra/cadinfo.html for Custom MOS Models

References Used by Students :

1. *Design of Analog CMOS Integrated Circuits* – by Behzad Razavi
2. https://www.analog.com/en/resources/media-center/videos/series/ltspice-essentials-tutorial.html

Rubric wise marks obtained:

Rubrics	**1**	**2**	**3**	**4**	**5**	**Total**
Marks						