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. Van Vishal Manji** Enrollment No. **230170111140** 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: **VGEC - Ahmedabad**
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	C O2	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 LT-SPICE.	√		√		
17.	Implementation of Resistive load and CMOS inverters using LT-SPICE.	√		√		
18.	Implementation of CMOS NAND and NOR gate using LT-SPICE.	√		√		

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 LT-SPICE tool.						
17	Implementation of Resistive load and CMOS inverters using LT-SPICE.						
18	Implementation of CMOS NAND and NOR gate using LT-SPICE.						
	Total						

Experiment No: 1

Date: _____

Aim: Introduction to FPGA & CPLD.

Competency and Practical Skills:

Relevant CO: CO5

Equipment / Instruments: Laptop or Computer with Xilinx Tools.

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?

Factors	**ASIC (Application Specific Integrated Circuit)**	**FPGA (Field Programmable Gate Array)**
1. Customization	Fixed hardware, designed for one specific application only (cannot be reprogrammed after manufacturing).	Reconfigurable hardware — logic can be programmed and changed multiple times after manufacturing.
2. Performance & Efficiency.	Faster speed and lower power consumption because the circuit is optimized for a single task.	Slower and less power-efficient compared to ASIC since it has programmable logic blocks and routing overhead.
3. Cost & Flexibility.	Very high initial development cost (design + fabrication), but cheaper per unit when mass-produced	Low upfront cost, suitable for prototyping and small production runs, but expensive per unit for large volumes.

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).

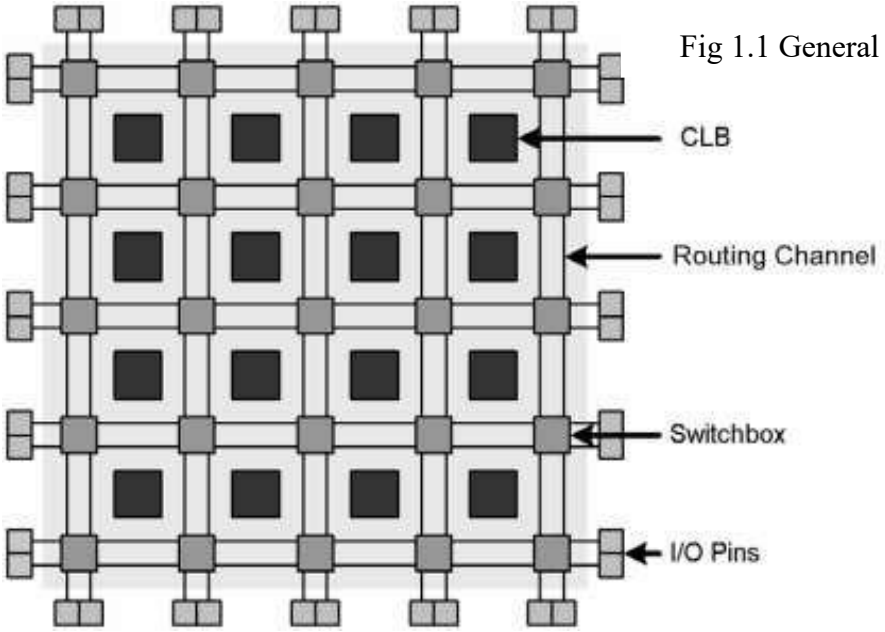


Fig 1.1 General Architecture of FPGA

Explain the structure of switch matrix :

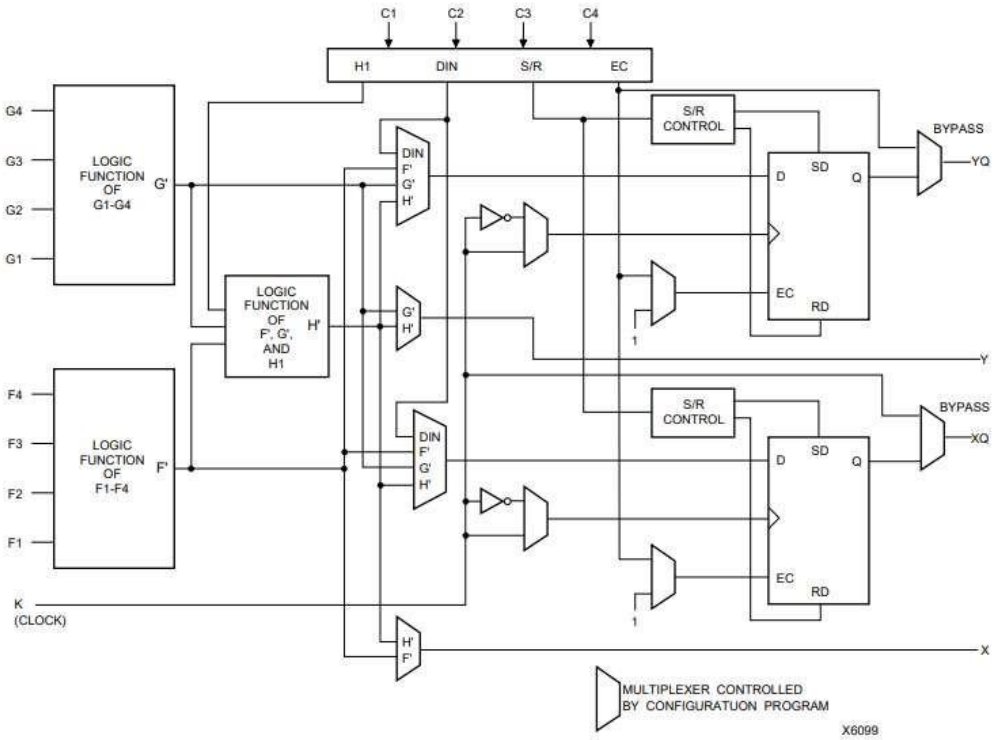


Fig 1.2 Simplified Block Diagram of XC4000- Families CLB

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. Architecture	Consists of a few large logic blocks with limited interconnects	Contains thousands of small configurable logic blocks (CLBs) with rich interconnects.
2. Complexity	Suitable for simpler, smaller designs (glue logic, control circuits).	Suitable for complex, large designs (processors, DSP, SoCs).
3. Speed	Faster and more predictable timing for small logic functions.	Can be slower for simple logic due to routing overhead, but handles parallel operations well.
4. Power Consumption	Generally lower power for small designs.	Higher power consumption, especially for large/complex designs.
5. Memory	Usually non-volatile (retains configuration after power off).	Usually volatile (needs configuration at power-up, e.g., from external memory).
6. Cost	Lower cost for small-scale, simple applications.	More expensive but cost-effective for complex designs and prototyping.

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:

Fundamentals of Digital Logic with VHDL Design. McGraw-Hill. Brown, S., & Vranesic, Z. (2009).
https://media.digikey.com/pdf/data%20sheets/xilinx%20pdfs/xc4000,a,h.pdf

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: CO 5

Objectives:

Equipment / Instruments: Laptop or Computer with Xilinx Tools.

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 mixedsignal 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 generalpurpose 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-verilogand-their-functional-uses
https://www.sciencedirect.com/topics/computer-science/hardware-description-languages

References used by the students:

Fundamentals of Digital Logic with VHDL Design. McGraw-Hill. Brown, S., & Vranesic, Z. (2009).

https://resources.pcb.cadence.com/blog/2020-hardware-description-languages-vhdl-vs-verilogand-their-functional-uses

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: CO 5

Objectives:

Equipment / Instruments: Laptop or Computer with Xilinx Tools.

Basic:

The Xilinx ISE software controls all aspects of the design flow. Through the Project Navigator interface, one can access all of the design entry and design implementation tools. One can also access the files and documents associated with their project.

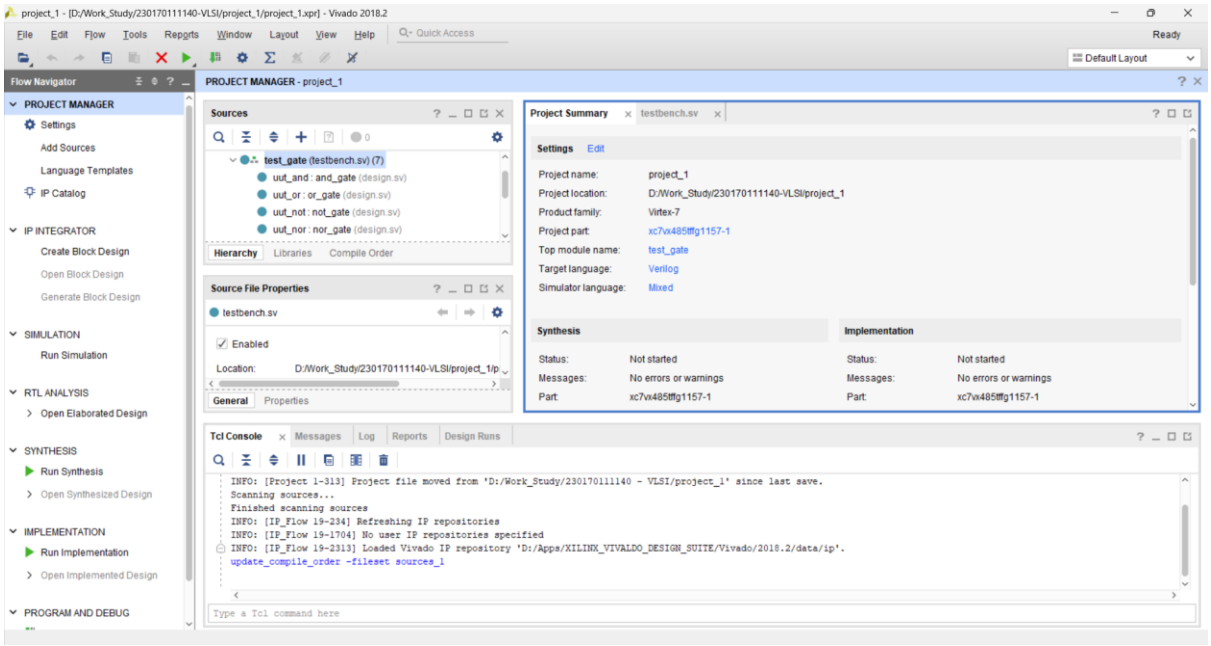


Fig. 3.1 Project Navigator

The Project Navigator interface is divided into four panel sub-windows, as seen in Figure 3.1.

- On the top left are the Start, Design, Files, and Libraries panels, which include display and access to the source files in the project as well as access to running processes for the currently selected source. The Start panel provides quick

access to opening projects as well as frequently access reference material, documentation and tutorials.

- At the bottom of the Project Navigator are the Console, Errors, and Warnings panels, which display status messages, errors, and warnings.

- To the right is a multi-document interface (MDI) window referred to as the Workspace. The Workspace enables person to view design reports, text files, schematics, and simulation waveforms.

- Each window can be resized, undocked from Project Navigator, moved to a new location within the main Project Navigator window, tiled, layered, or closed.

- One can use the View > Panels menu commands to open or close panels. You can use the Layout > Load Default Layout to restore the default window layout.

- The Design panel provides access to the View, Hierarchy, and Processes panes.

- The View pane radio buttons enable you to view the source modules associated with the Implementation or Simulation Design View in the Hierarchy pane. If you select Simulation, you must select a simulation phase from the drop-down list.

- The Hierarchy pane displays the project name, the target device, user documents, and design source files associated with the selected Design View.

- The View pane at the top of the Design panel allows you to view only those source files associated with the selected Design View, such as Implementation or Simulation.

- Each file in the Hierarchy pane has an associated icon. The icon indicates the file type (HDL file, schematic, core, or text file, for example).

- For a complete list of possible source types and their associated icons, see the “Source File Types” topic in the ISE Help.

- From Project Navigator, select Help > Help Topics to view the ISE Help. If a file contains lower levels of hierarchy, the icon has a plus symbol (+) to the left of the name. One can expand the hierarchy by clicking the plus symbol (+). One can open a file for editing by double-clicking on the filename.

- The Console provides all standard output from processes run from Project Navigator. It displays errors, warnings, and information messages.

- The Workspace is where design editors, viewers, and analysis tools open. These include ISE Text Editor, Schematic Editor, Constraint Editor, Design Summary/Report Viewer, RTL and Technology Viewers, and Timing Analyzer.

- Other tools such as the PlanAhead™ software for I/O planning and floorplanning, ISim, third-party text editors, XPower Analyzer, and iMPACT open in separate windows outside the main Project Navigator environment when invoked.

- The Design Summary provides a summary of key design data as well as access to all of the messages and detailed reports from the synthesis and implementation tools.

- The summary lists high-level information about your project, including overview information, a device utilization summary, performance data gathered from the Place and Route (PAR) report, constraints information, and summary information from all reports with links to the individual reports.

➔ **Sample Program on Xilinx Vivado with steps for simulation.**

- Design Code :

```
module mux2to1
        (
        input wire a,      // Input 1
        input wire b,      // Input 2
        input wire sel,    // Select Line
        output wire y      // Output
        );

        assign y = (sel) ? b : a;
endmodule
```

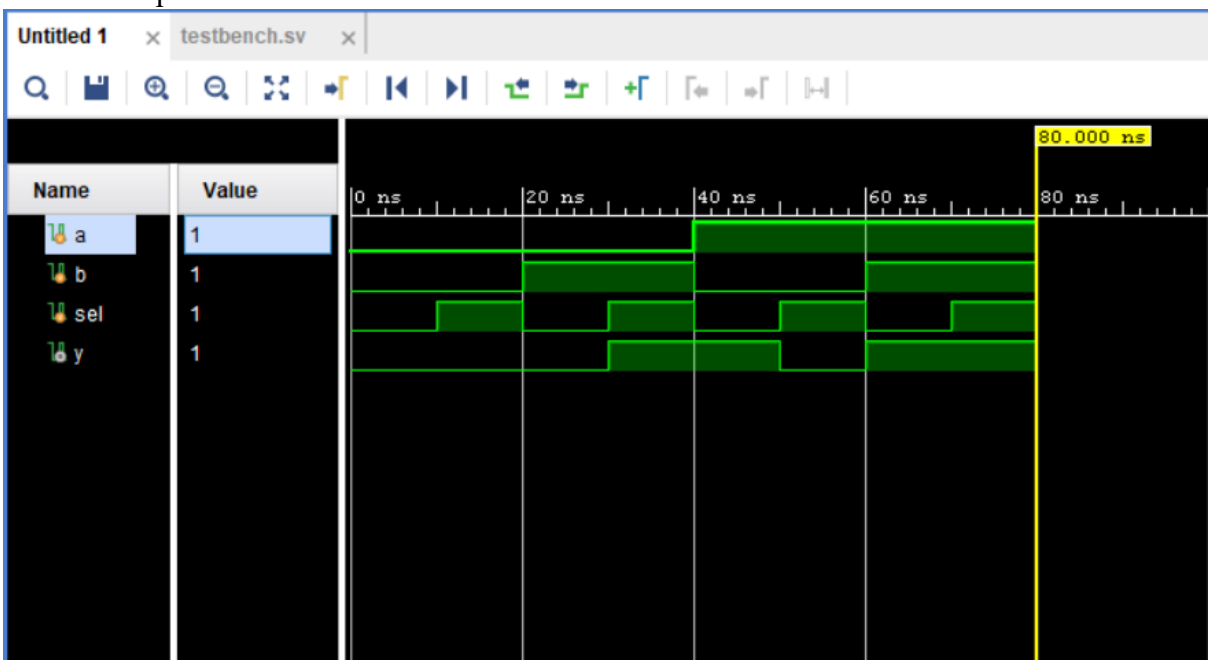
- Testbench Code :

```
`timescale 1ns / 1ps

module tb_mux2to1;

    // Testbench signals
    reg a;
    reg b;
    reg sel;
    wire y;

    // Instantiate the DUT (Device Under Test)
    mux2to1 uut (
        .a(a),
        .b(b),
        .sel(sel),
        .y(y)
    );

    // Stimulus
    initial begin
        // Print header
        $display("Time\tA B SEL | Y");
        $display("------------------");

        // Test all combinations
        a = 0; b = 0; sel = 0; #10; $display("%0t\t%b %b  %b  | %b", $time, a, b, sel, y);
        a = 0; b = 0; sel = 1; #10; $display("%0t\t%b %b  %b  | %b", $time, a, b, sel, y);
        a = 0; b = 1; sel = 0; #10; $display("%0t\t%b %b  %b  | %b", $time, a, b, sel, y);
        a = 0; b = 1; sel = 1; #10; $display("%0t\t%b %b  %b  | %b", $time, a, b, sel, y);
        a = 1; b = 0; sel = 0; #10; $display("%0t\t%b %b  %b  | %b", $time, a, b, sel, y);
        a = 1; b = 0; sel = 1; #10; $display("%0t\t%b %b  %b  | %b", $time, a, b, sel, y);
        a = 1; b = 1; sel = 0; #10; $display("%0t\t%b %b  %b  | %b", $time, a, b, sel, y);
        a = 1; b = 1; sel = 1; #10; $display("%0t\t%b %b  %b  | %b", $time, a, b, sel, y);

        $finish;
    end

endmodule
```

- Output Log (Display Truth table) for simulation :

```
Time        A B SEL | Y
------------------
10000       0 0  0  | 0
20000       0 0  1  | 0
30000       0 1  0  | 0
40000       0 1  1  | 1
50000       1 0  0  | 1
60000       1 0  1  | 0
70000       1 1  0  | 1
80000       1 1  1  | 1
```

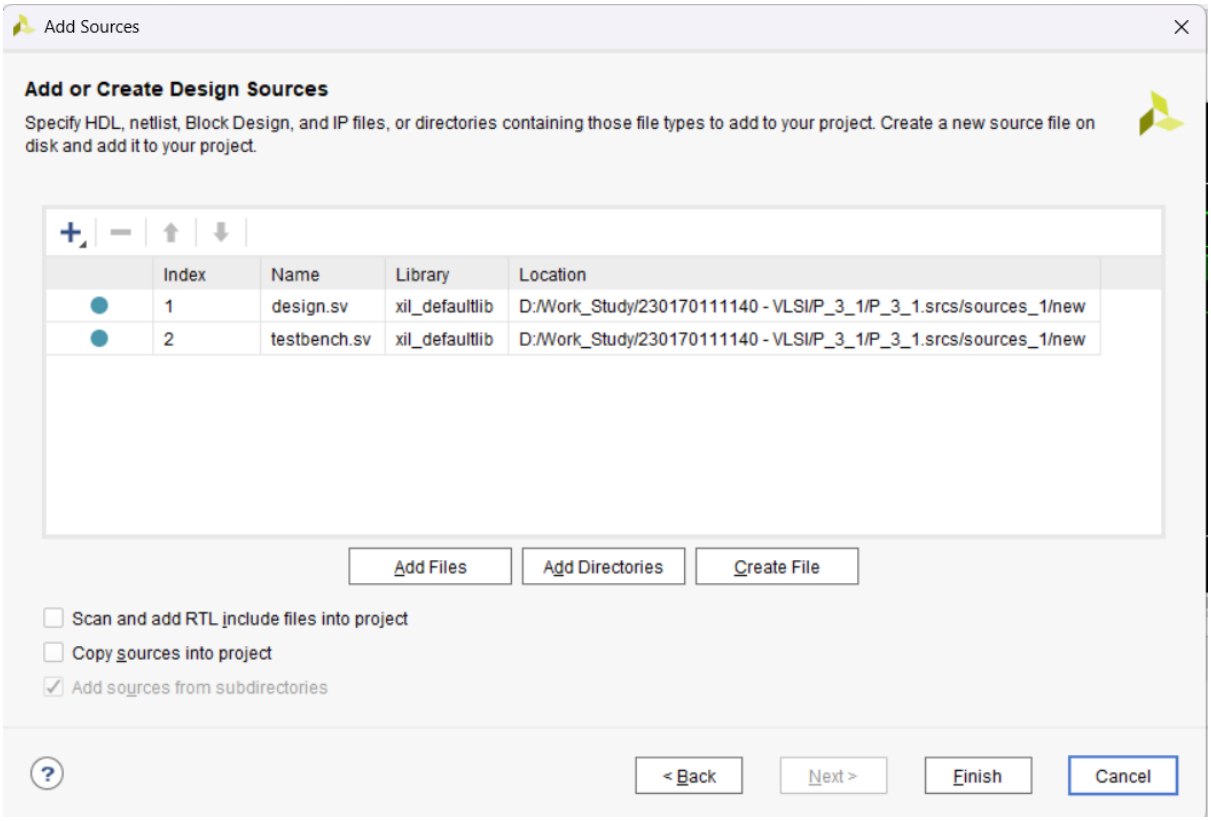
➢ Graph for the truth table :

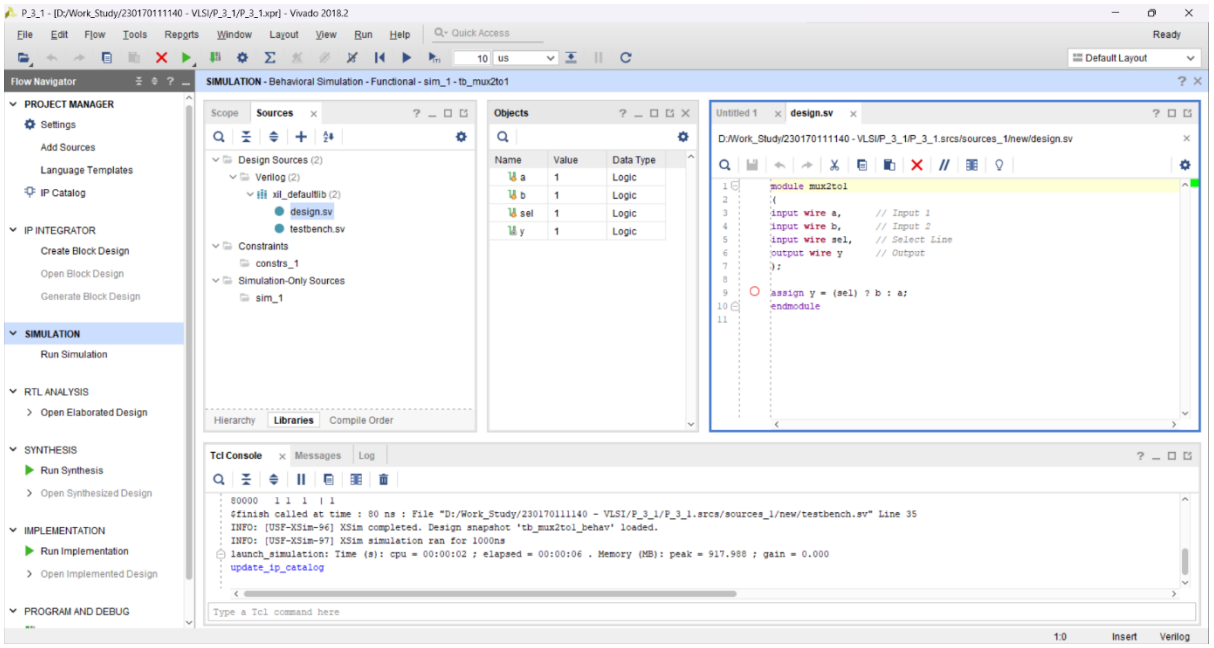


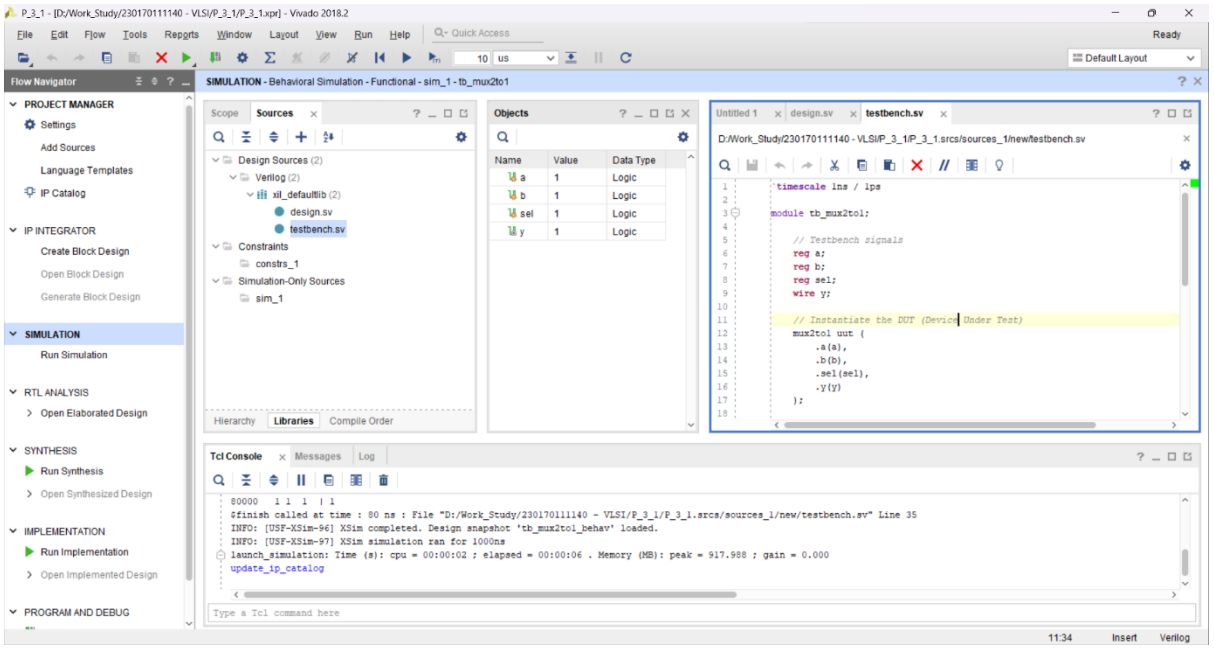
From the 2x1 MUX Waveform , it is evident that y = b if sel = 1 or y = a if sel = 0

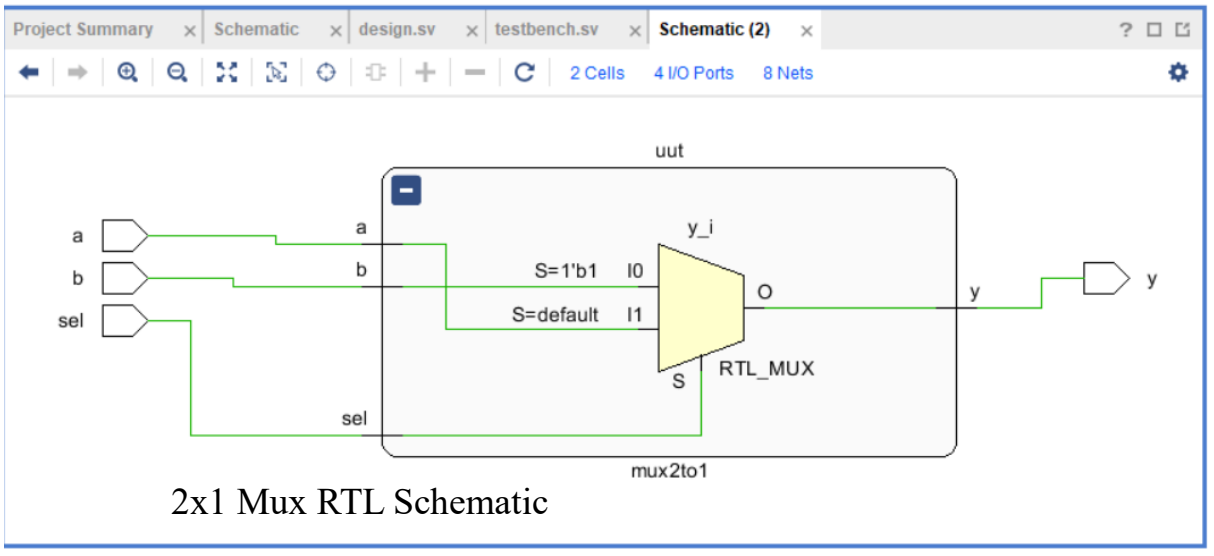
➢ Steps for simulation :

Step 1 : Copy the design code in the design.sv file and testbench code in the testbench.sv after creating 2 verilog files “design.sv” , “testbench.sv”the add or create the design sources tab



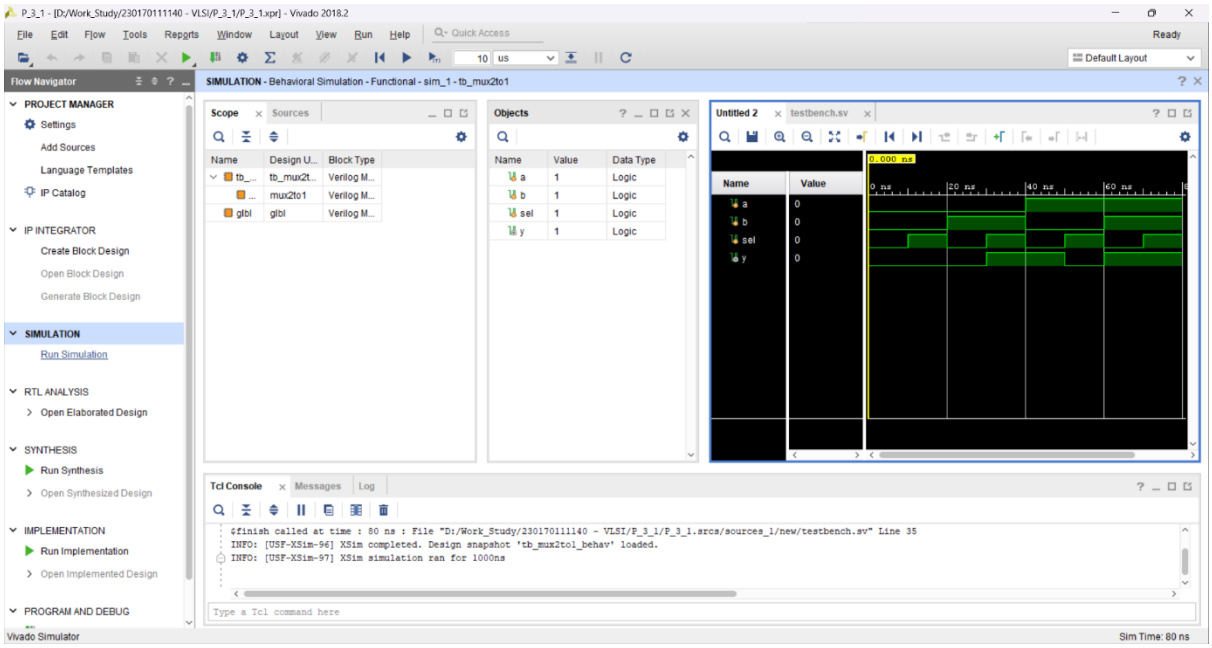


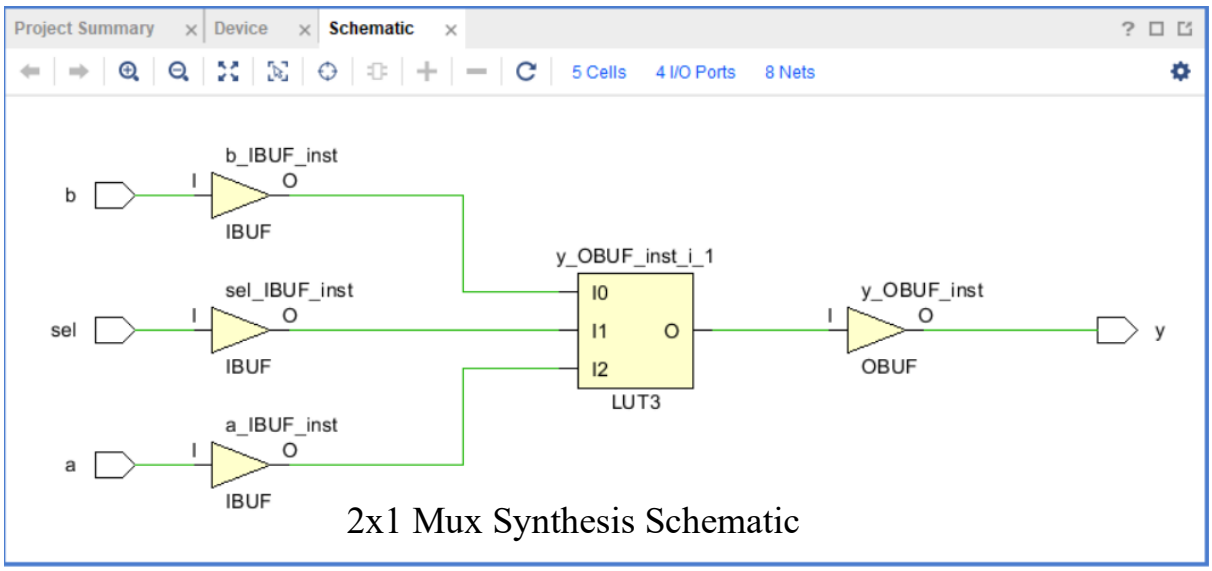




2x1 Mux RTL Schematic

Step 2 : Click on run simulation to generate the Waveform graph (UntitledX) and truth table (TCL Console) for output , also on schematic on RTL analysis & schematic after generating synthesis.





2x1 Mux Synthesis Schematic

Conclusion: In this experiment, we were introduced to Xilinx design tools, which provide an environment for creating, simulating, and implementing digital circuits using HDLs. The tools help in writing code, verifying functionality through simulation, and generating bitstreams for FPGA/CPLD programming. This practical exposure highlighted the role of Xilinx tools in simplifying the design flow, from code entry to hardware implementation, making them essential for modern digital system design and prototyping.

Suggested Reference:

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

References used by the students:

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

Circuit Design with VHDL. MIT Press. Pedroni, V. A. (2008).

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

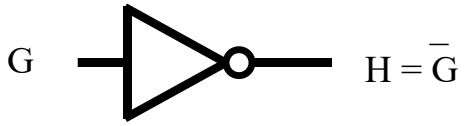
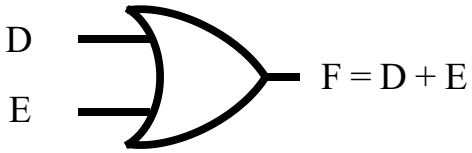
Relevant CO: CO 5, CO 3

Objectives: Understanding of different modeling styles in Verilog HDL.

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

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>G</th><th>H</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>D</th><th>E</th><th>F</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>
<table>
<tr><th>A</th><th>B</th><th>C</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>
</td></tr>
</table>

NAND Gate

$N = \overline{L \cdot M}$

L	M	N
0	0	1
0	1	1
1	0	1
1	1	0

NOR Gate

$K = \overline{I + J}$

I	J	K
0	0	1
0	1	0
1	0	0
1	1	0

XOR Gate

$Q = O \oplus P$

O	P	Q
0	0	0
0	1	1
1	0	1
1	1	0

XNOR Gate

$T = \overline{R \oplus S}$

R	S	T
0	0	1
0	1	0
1	0	0
1	1	1

1.Logic Gates.

- NOT GATE:
  - Design code :

```
module not_gate
        (
        input G,
        output H,
        );
        assign H = ~ G;
endmodule

// Data Flow Modelling
```

a.Data Flow Modelling

```
module not_gate
        (
        input G,
        output reg H
        );
        always@(G)
        begin
        H = ~ G;
        end
endmodule
// Behavioural Modelling
```

b. Behavioural Modelling

```
module not_gate
    (
    input G,
    output H
    );
    not n1(H,G);
endmodule

//Structural Modelling
```

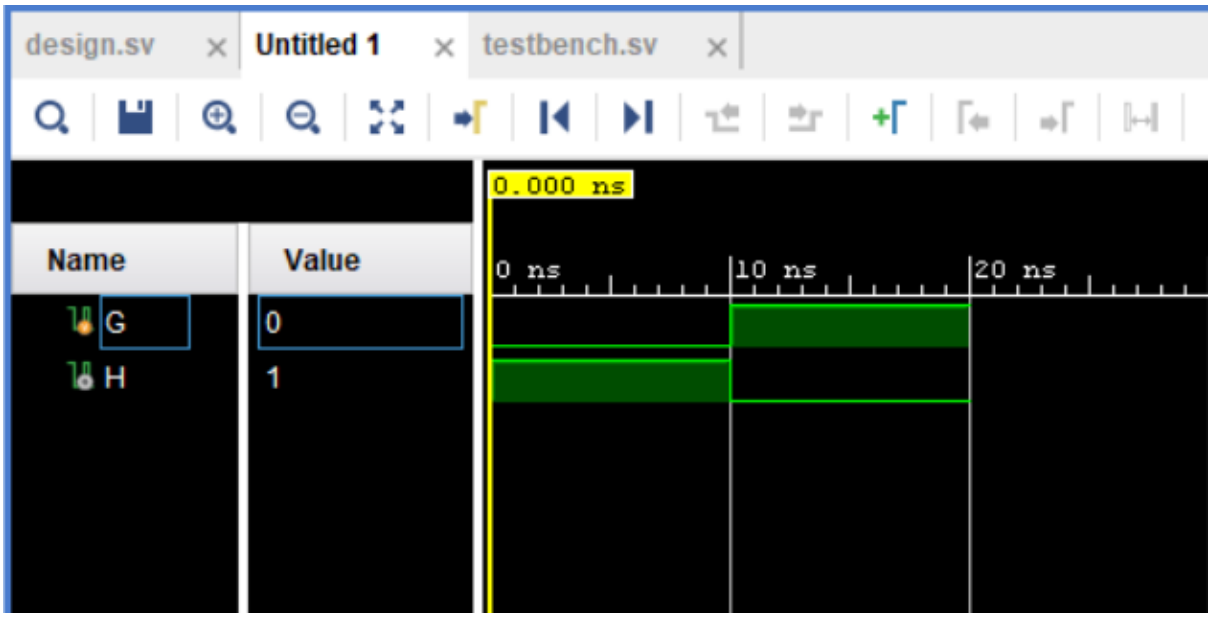
c. Structural Modelling

- Testbench Code :

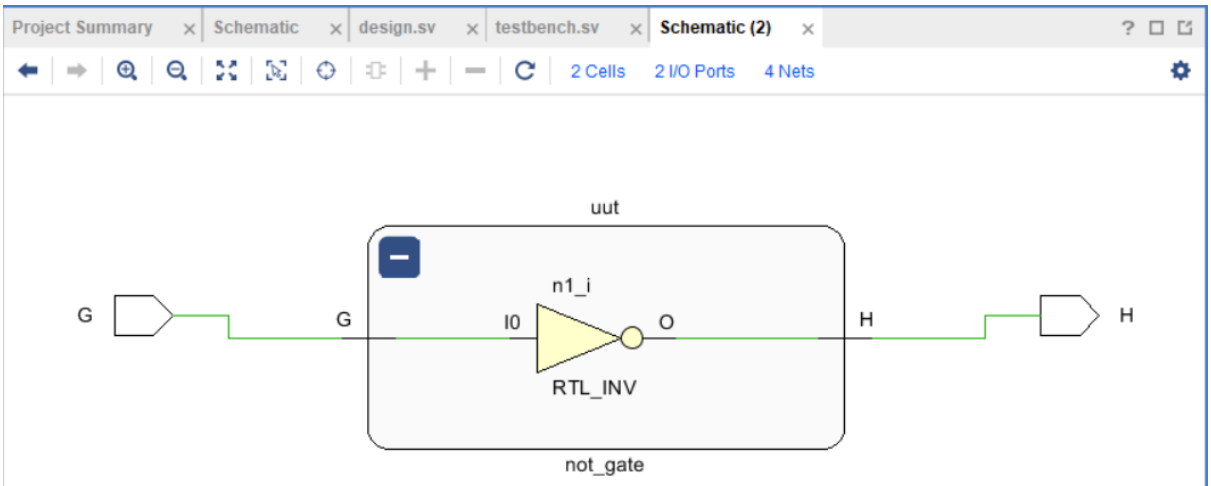
```
module test_gate;
    reg G;
    wire H;

    // Instantiate not_gate
    not_gate uut_not
        (
        .G(G),
        .H(H)
        );
    initial begin
    $dumpfile("dump4_1.vcd");
    $dumpvars();

    // NOT gate test
    $display("");
    $display("NOT Gate Test");
    $display("G | H");
    $display("-----");
    G = 0; #10; $display("%b | %b", G, H);
    G = 1; #10; $display("%b | %b", G, H);
    $finish;
    end
endmodule
```

- Output Truth table :

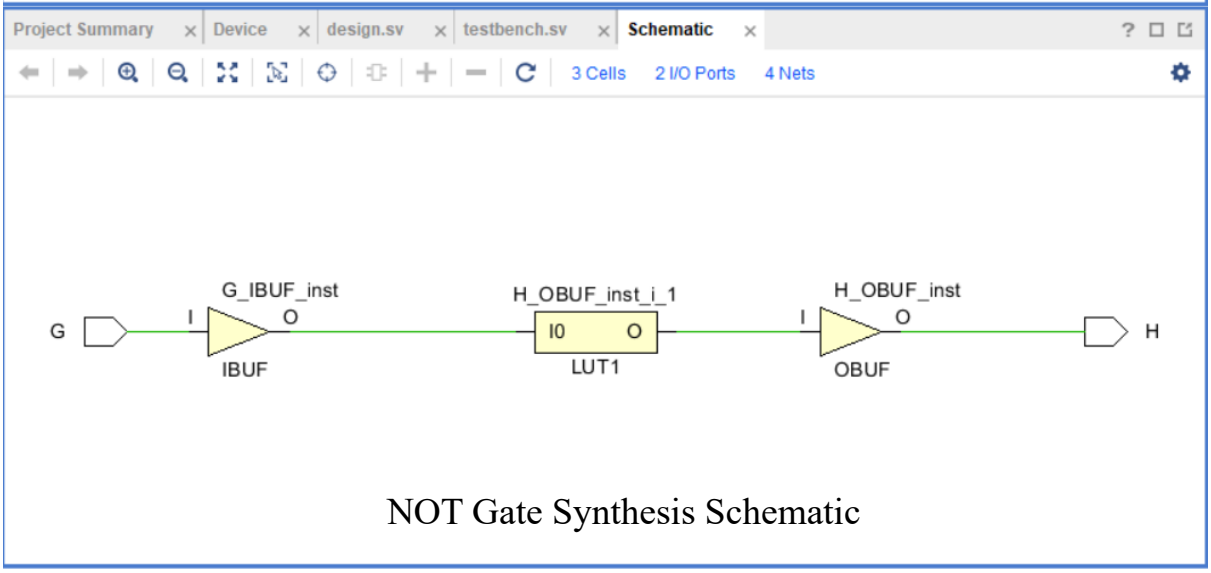
```
NOT Gate Test
G | H
-----
0 | 1
1 | 0
```



NOT Gate Waveform Graph



NOT Gate RTL Schematic



NOT Gate Synthesis Schematic

- OR GATE :
  - Design Code :

```
module or_gate
        (
        input D,
        input E,
        output F
        );
        assign F = D | E ;
endmodule

// Data Flow Modelling
```

a.Data Flow Modelling

```
module or_gate
        (
        input D,
        input E,
        output reg F
        );
        always@(D,E)
        begin
        F = D | E ;
        end
endmodule
// Behavioural Modelling
```

b. Behavioural Modelling

```
module or_gate
        (
        input D,
        input E,
        output F
        );
        or o1(F,D,E);
endmodule

//Structural Modelling
```

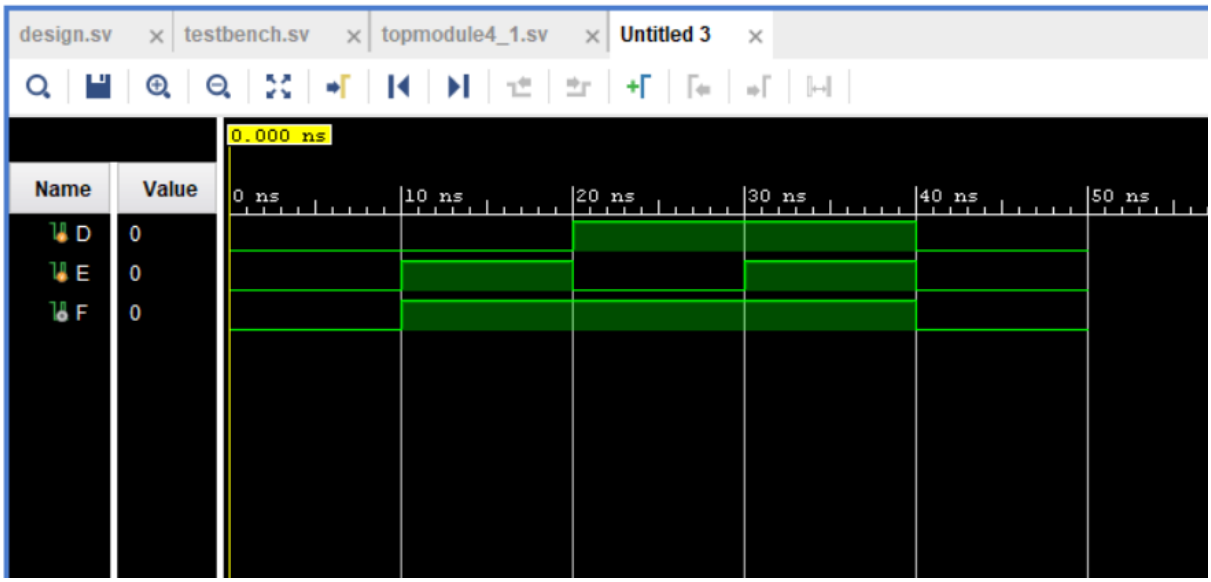
c. Structural Modelling

- Testbench Code :

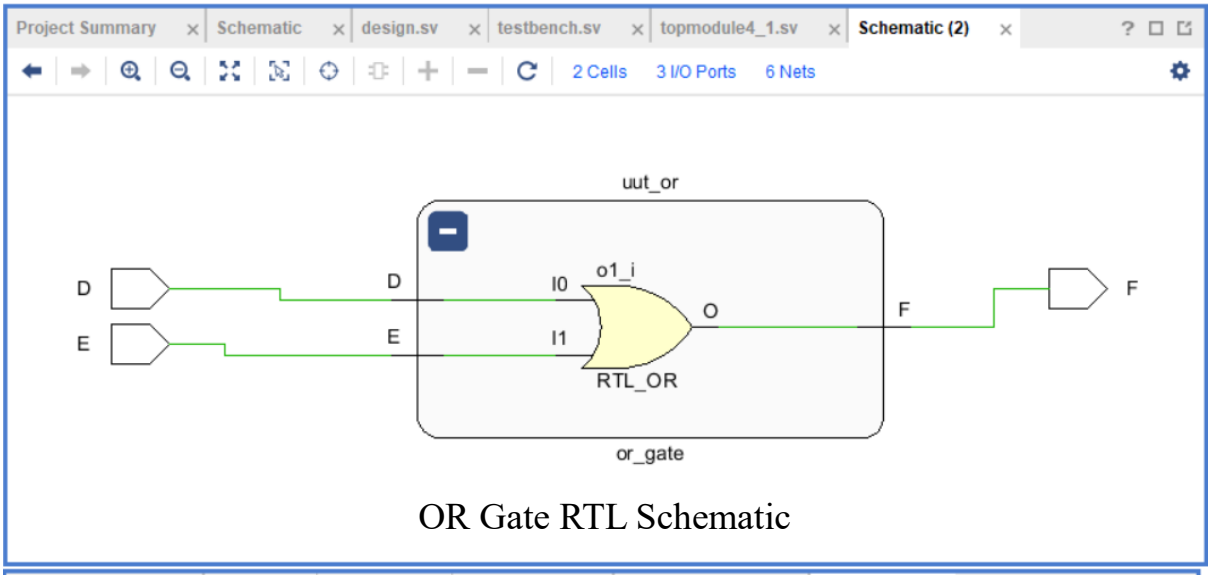
```
module test_gate;
        reg D,E;
        wire F;

        // Instantiate or_gate
        or_gate uut_or
        (
        .D(D),
        .E(E),
        .F(F)
        );
        initial begin
        $dumpfile("dump4_1.vcd");
        $dumpvars();

        // OR gate test
        $display("");
        $display("Or Gate Test");
        $display("D E | F");
        $display("---------");
        D = 0; E = 0; #10; $display("%b %b | %b", D, E, F);
        D = 0; E = 1; #10; $display("%b %b | %b", D, E, F);
        D = 1; E = 0; #10; $display("%b %b | %b", D, E, F);
        D = 1; E = 1; #10; $display("%b %b | %b", D, E, F);
        D = 0; E = 0; #10; $display("%b %b | %b", D, E, F);
        $finish;
    end
endmodule
```

- Output Truth Table :

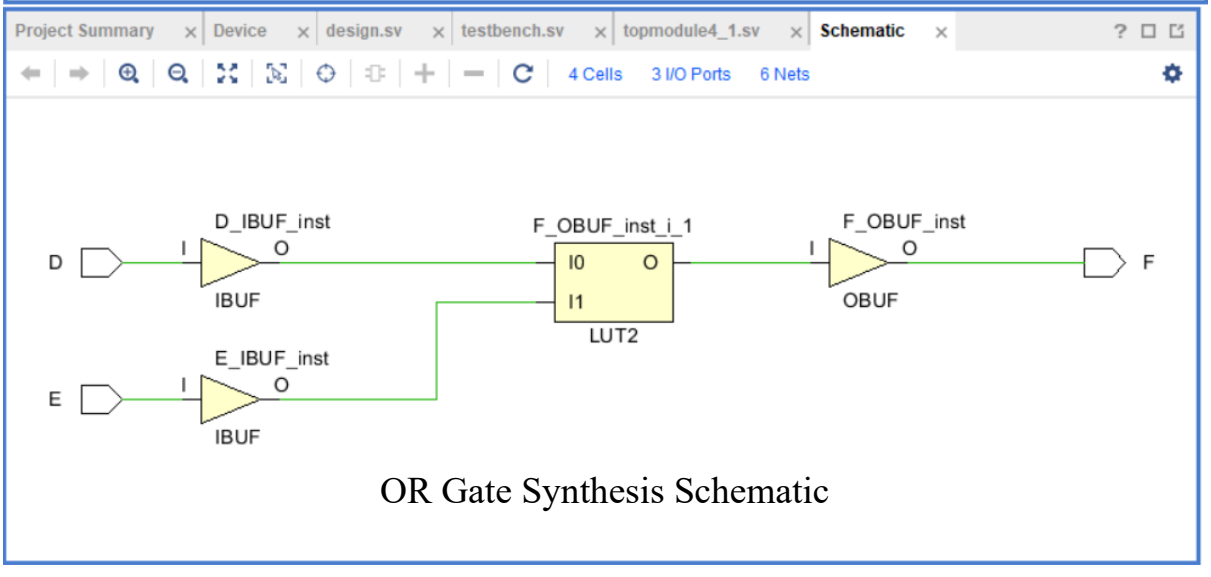
```
Or Gate Test
D E | F
---------
0 0 | 0
0 1 | 1
1 0 | 1
1 1 | 1
0 0 | 0
```



OR Gate Waveform Graph



OR Gate RTL Schematic



OR Gate Synthesis Schematic

- AND GATE :
  - Design Code :

```
module and_gate
        (
        input A,
        input B,
        output C
        );
        assign C = A & B;
endmodule

// Data Flow Modelling
```

a.Data Flow Modelling

```
module and_gate
        (
        input A,
        input B,
        output reg C
        );
        always@(A,B)
        begin
        C = A & B;
        end
endmodule
// Behavioural Modelling
```

b. Behavioural Modelling

```
module and_gate
        (
        input A,
        input B,
        output C
        );
        and a1(C,A,B);
endmodule

//Structural Modelling
```

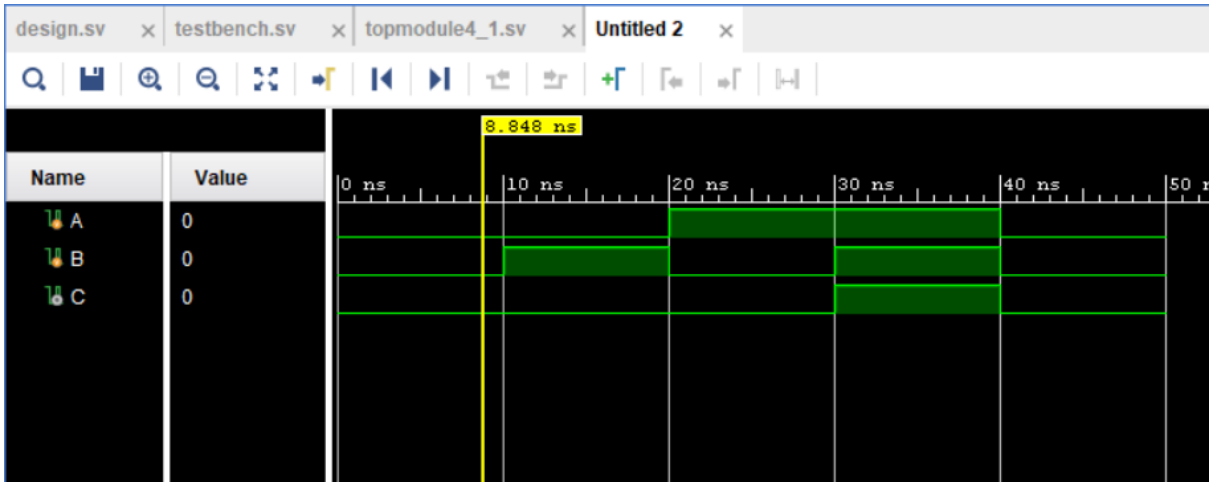
c. Structural Modelling

- Testbench Code :

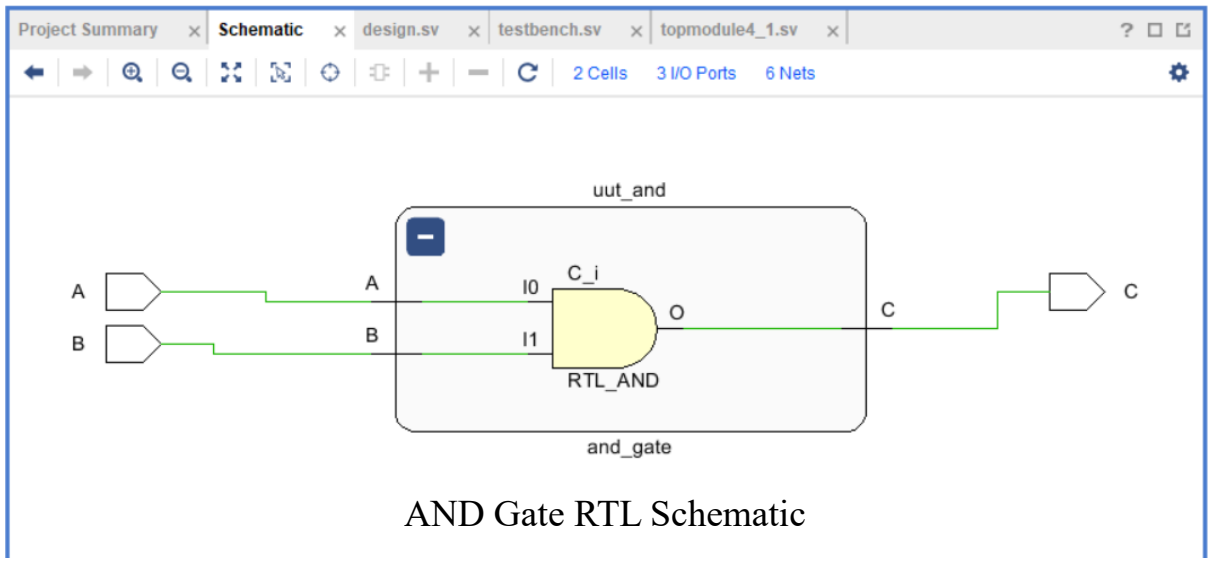
```
module test_gate;
        reg A,B;
        wire C;

        // Instantiate and_gate
        and_gate uut_and
        (
        .A(A),
        .B(B),
        .C(C)
        );
        initial begin
        $dumpfile("dump4_1.vcd");
        $dumpvars();

        // AND gate test
        $display("And Gate Test");
        $display("A B | C");
        $display("---------");
        A = 0; B = 0; #10; $display("%b %b | %b", A, B, C);
        A = 0; B = 1; #10; $display("%b %b | %b", A, B, C);
        A = 1; B = 0; #10; $display("%b %b | %b", A, B, C);
        A = 1; B = 1; #10; $display("%b %b | %b", A, B, C);
        A = 0; B = 0; #10; $display("%b %b | %b", A, B, C);
        $finish;
    end
endmodule
```

- Output Truth Table :

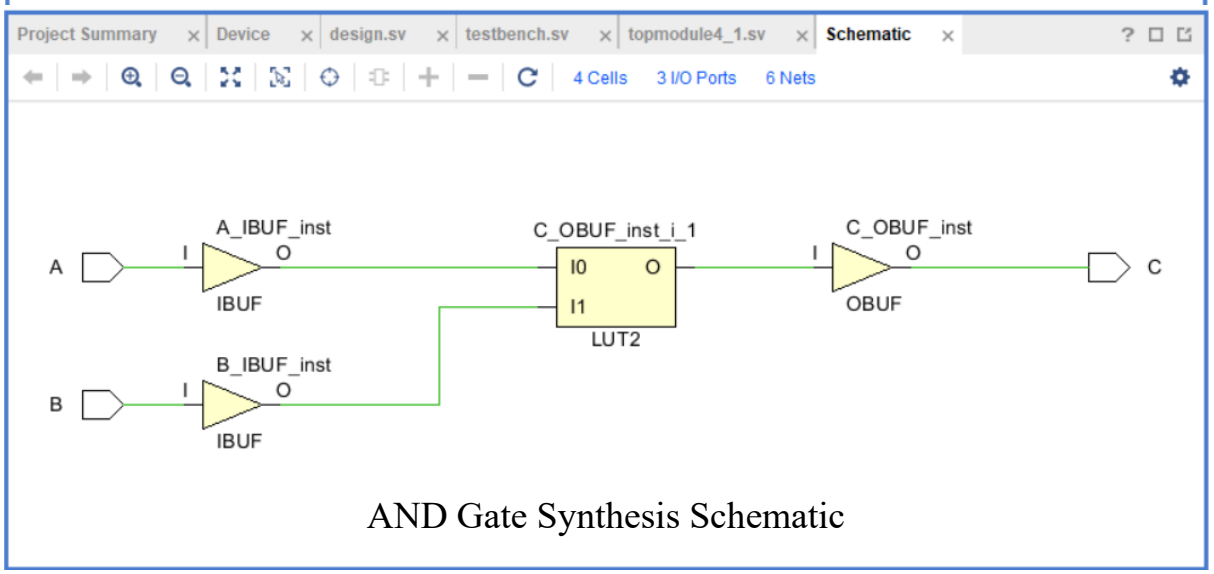
```
And Gate Test
A B | C
---------
0 0 | 0
0 1 | 0
1 0 | 0
1 1 | 1
0 0 | 0
```



AND Gate Waveform Graph



AND Gate RTL Schematic



AND Gate Synthesis Schematic

- NAND GATE :
  - Design Code :

```
module nand_gate
        (
        input L,
        input M,
        output N
        );
    assign N = L & M ;
endmodule

// Data Flow Modelling
```

a.Data Flow Modelling

```
module nand_gate
        (
        input L,
        input M,
        output reg N
        );
        always@(L,M)
        begin
        N = L & M;
        end
endmodule
// Behavioural Modelling
```

b. Behavioural Modelling

```
module nand_gate
        (
        input L,
        input M,
        output N
        );
    nand na1(N,L,M);
endmodule

//Structural Modelling
```

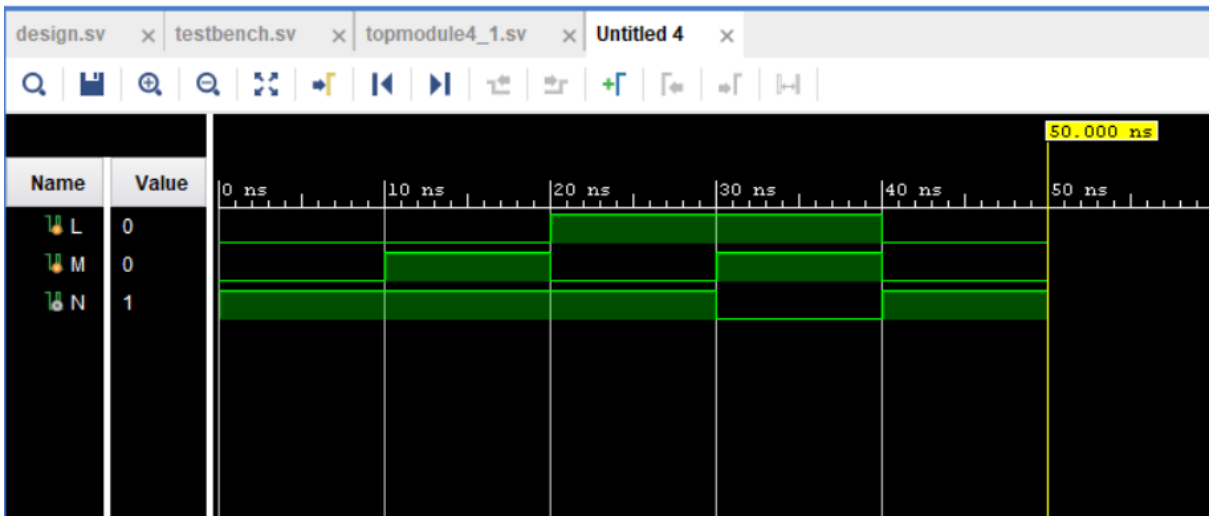
c. Structural Modelling

- Testbench Code :

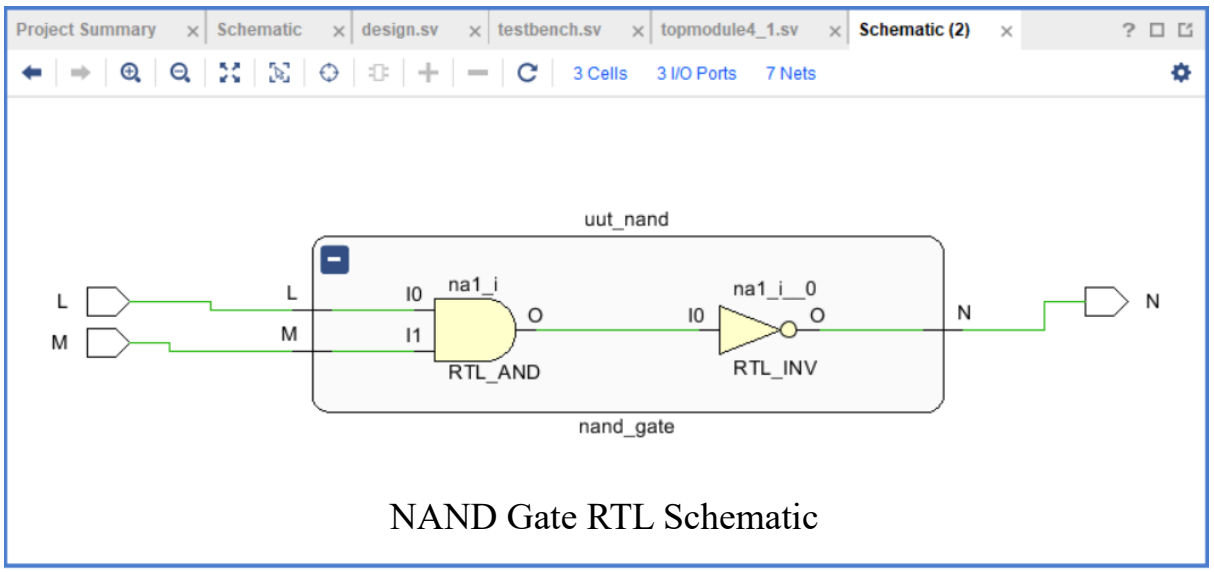
```
module test_gate;
        reg L,M;
        wire N;

        // Instantiate nand_gate
        nand_gate uut_nand
        (
        .L(L),
        .M(M),
        .N(N)
        );
        initial begin
        $dumpfile("dump4_1.vcd");
        $dumpvars();

        // Nand gate test
        $display("");
        $display("Nand Gate Test");
        $display("L M | N");
        $display("---------");
        L = 0; M = 0; #10; $display("%b %b | %b", L, M, N);
        L = 0; M = 1; #10; $display("%b %b | %b", L, M, N);
        L = 1; M = 0; #10; $display("%b %b | %b", L, M, N);
        L = 1; M = 1; #10; $display("%b %b | %b", L, M, N);
        L = 0; M = 0; #10; $display("%b %b | %b", L, M, N);
    $finish;
    end
endmodule
```

- Output Truth Table :

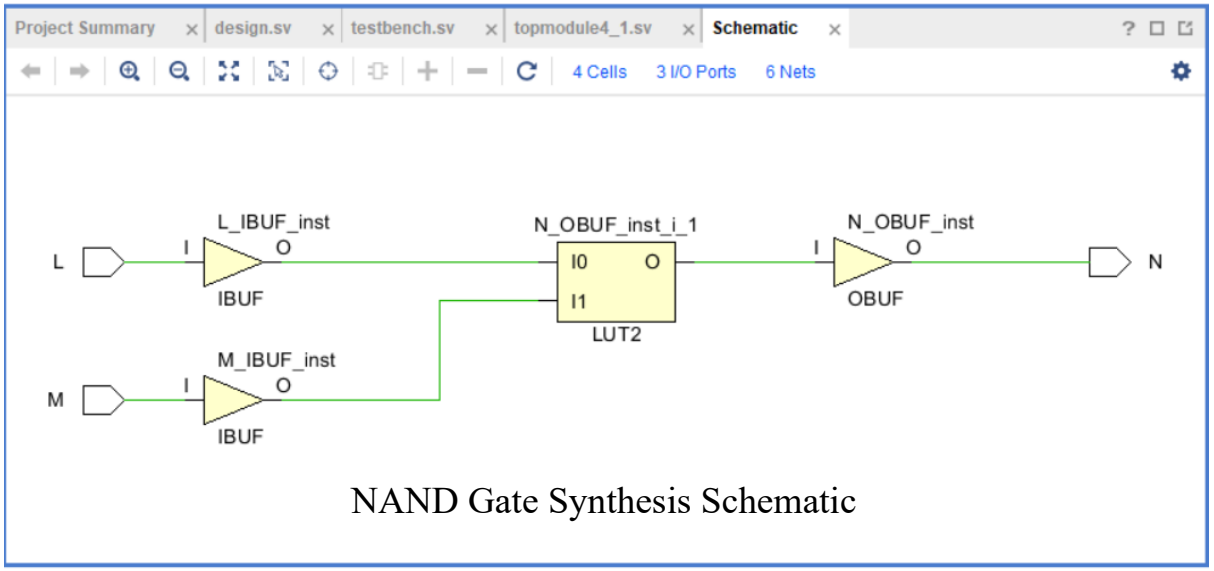
```
Nand Gate Test
L M | N
---------
0 0 | 1
0 1 | 1
1 0 | 1
1 1 | 0
0 0 | 1
```



NAND Gate Waveform Graph



NAND Gate RTL Schematic



NAND Gate Synthesis Schematic

- NOR GATE :
  - Design Code :

```
module nor_gate
        (
        input I,
        input J,
        output K
        );
        assign K = ( I | J );
endmodule

// Data Flow Modelling
```

a.Data Flow Modelling

```
module nor_gate
        (
        input I,
        input J,
        output reg K
        );
        always@(I,J)
        begin
        K = ~ ( I | J );
        end
endmodule
// Behavioural Modelling
```

b. Behavioural Modelling

```
module nor_gate
        (
        input I,
        input J,
        output K
        );
        nor no1(K,I,J);
endmodule

//Structural Modelling
```

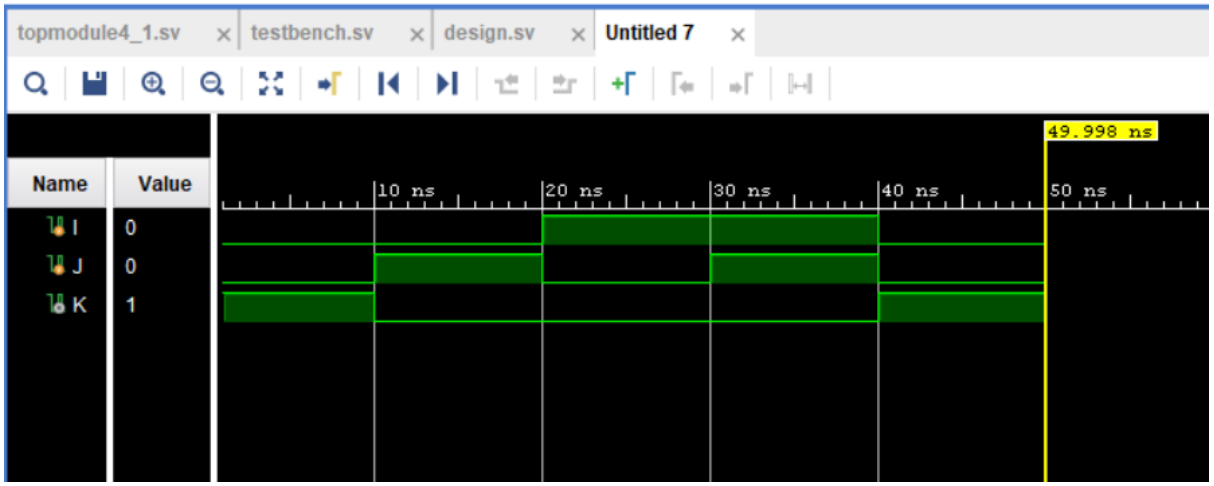
c. Structural Modelling

- Testbench Code :

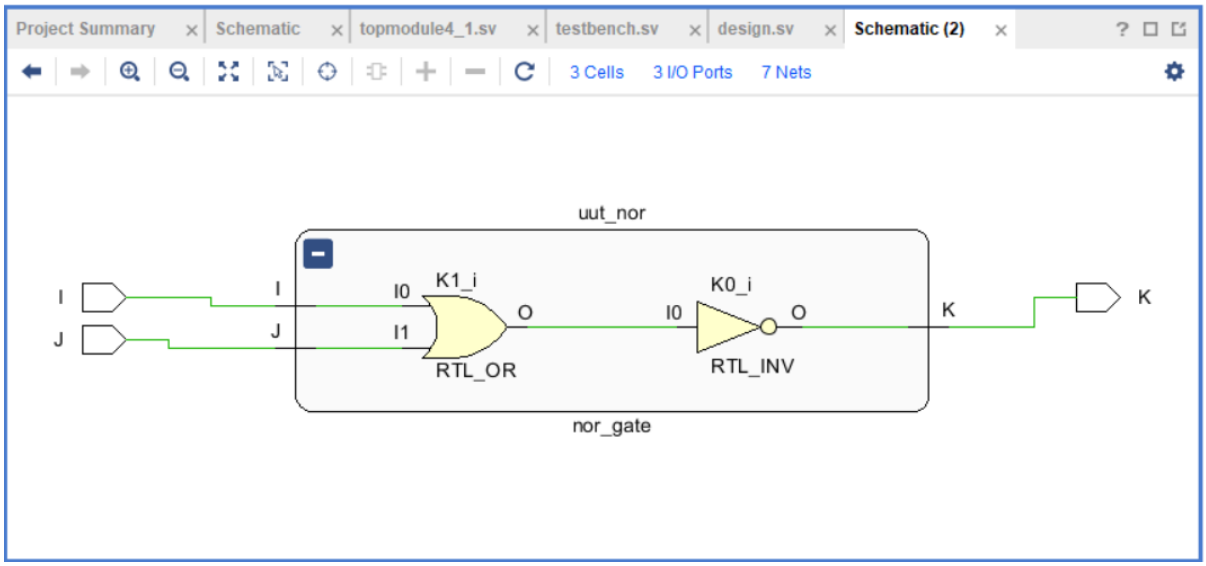
```
module test_gate;
        reg I,J;
        wire K;

        // Instantiate nor_gate
        nor_gate uut_nor
        (
        .I(I),
        .J(J),
        .K(K)
        );
        initial begin
        $dumpfile("dump4_1.vcd");
        $dumpvars();

        // NOR gate test
        $display("");
        $display("NOR Gate Test");
        $display("I J | K");
        $display("---------");
        I = 0; J = 0; #10; $display("%b %b | %b", I, J, K);
        I = 0; J = 1; #10; $display("%b %b | %b", I, J, K);
        I = 1; J = 0; #10; $display("%b %b | %b", I, J, K);
        I = 1; J = 1; #10; $display("%b %b | %b", I, J, K);
        I = 0; J = 0; #10; $display("%b %b | %b", I, J, K);
    $finish;
    end
endmodule
```

- Output Truth Table :

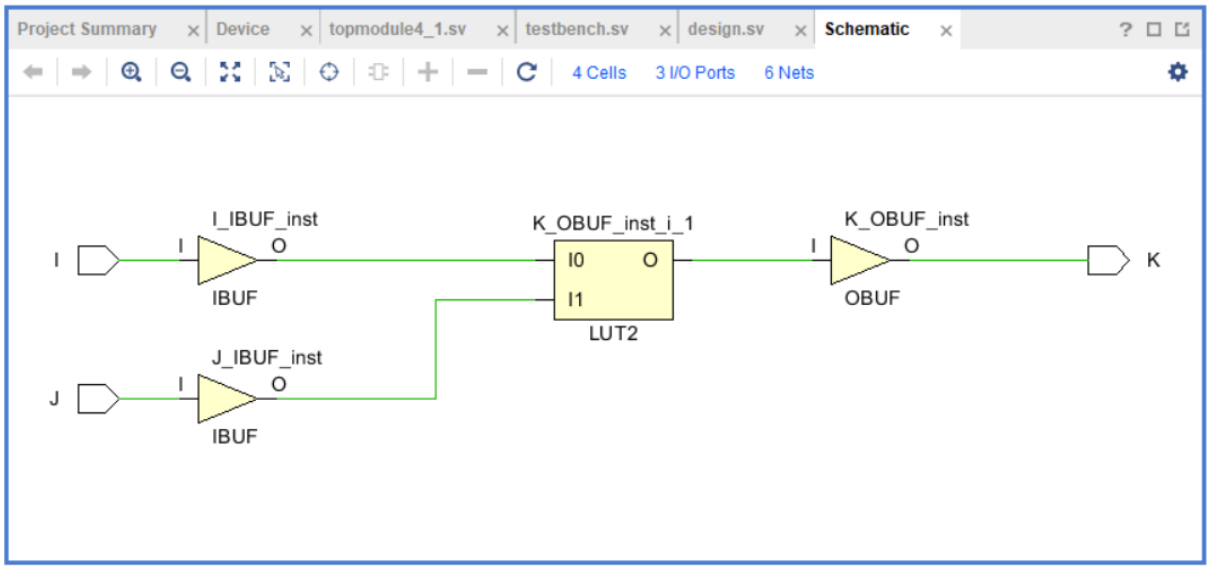
```
NOR Gate Test
I J | K
---------
0 0 | 1
0 1 | 0
1 0 | 0
1 1 | 0
0 0 | 1
```



NOR Gate Waveform Graph



NOR Gate RTL Schematic



NOR Gate Synthesis Schematic

- XOR GATE :
  - Design Code :

```
module xor_gate
        (
        input O,
        input P,
        output Q
        );
        assign Q = O ^ P;
endmodule

// Data Flow Modelling
```

a.Data Flow Modelling

```
module xor_gate
        (
        input O,
        input P,
        output reg Q
        );
        always@(O,P)
        begin
        Q =  O ^ P;
        end
endmodule
// Behavioural Modelling
```

b. Behavioural Modelling

```
module xor_gate
        (
        input O,
        input P,
        output Q
        );
        xor xo1(Q,O,P);
endmodule

//Structural Modelling
```

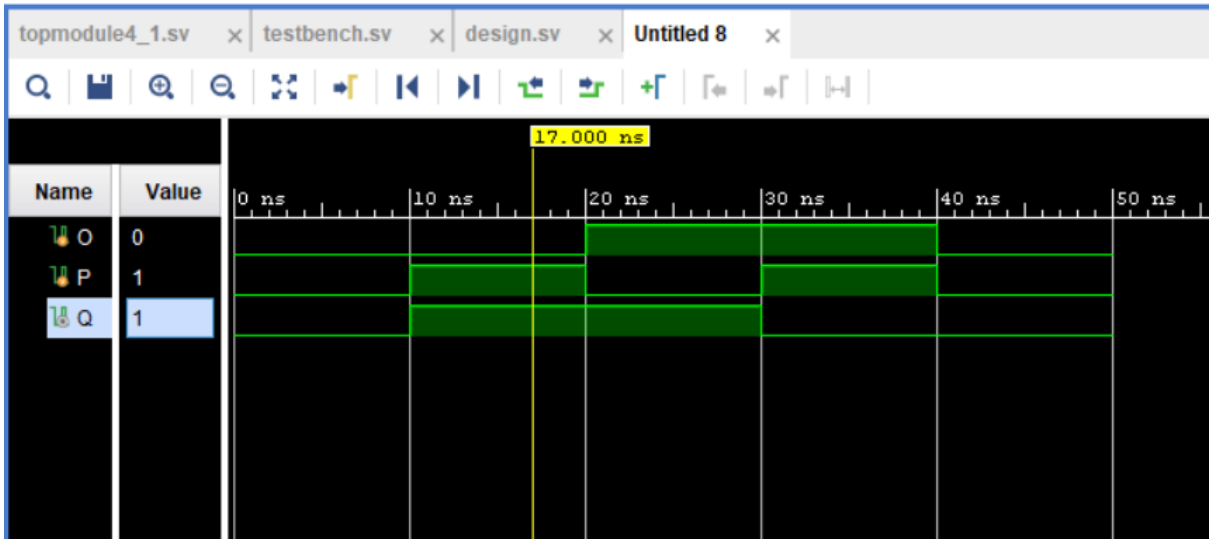
c. Structural Modelling

  - Testbench Code :

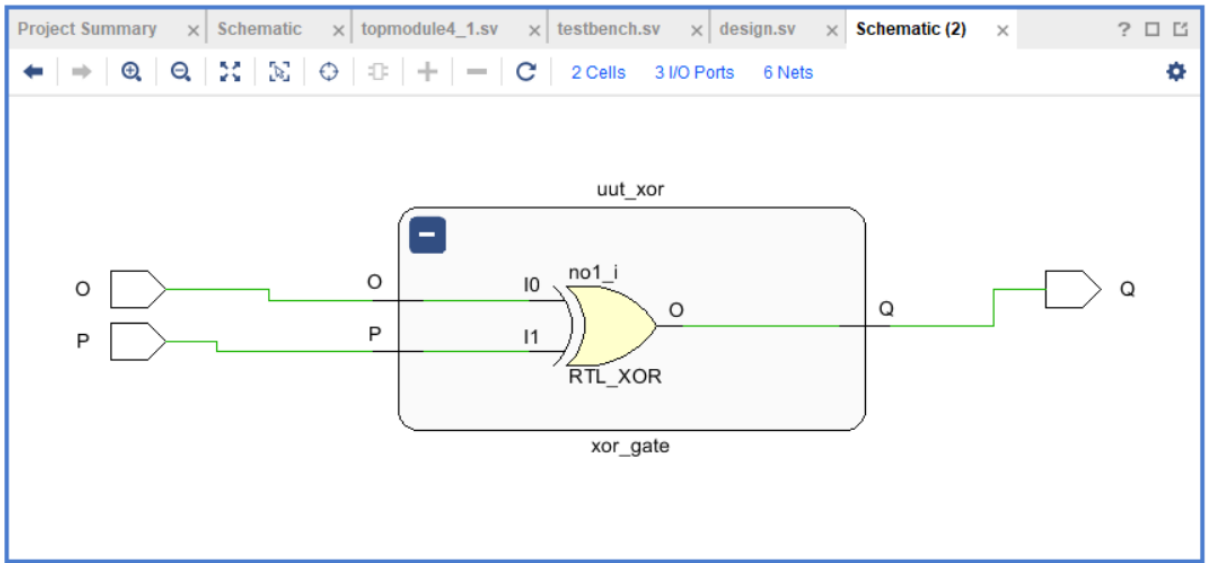
```
module test_gate;
        reg O,P;
        wire Q;

        // Instantiate xor_gate
        xor_gate uut_xor
        (
        .O(O),
        .P(P),
        .Q(Q)
        );
        initial begin
        $dumpfile("dump4_1.vcd");
        $dumpvars();

        // XOR gate test
        $display("");
        $display("XOR Gate Test");
        $display("O P | Q");
        $display("---------");
        O = 0; P = 0; #10; $display("%b %b | %b", O, P, Q);
        O = 0; P = 1; #10; $display("%b %b | %b", O, P, Q);
        O = 1; P = 0; #10; $display("%b %b | %b", O, P, Q);
        O = 1; P = 1; #10; $display("%b %b | %b", O, P, Q);
        O = 0; P = 0; #10; $display("%b %b | %b", O, P, Q);
    $finish;
    end
endmodule
```

- Output Truth Table :

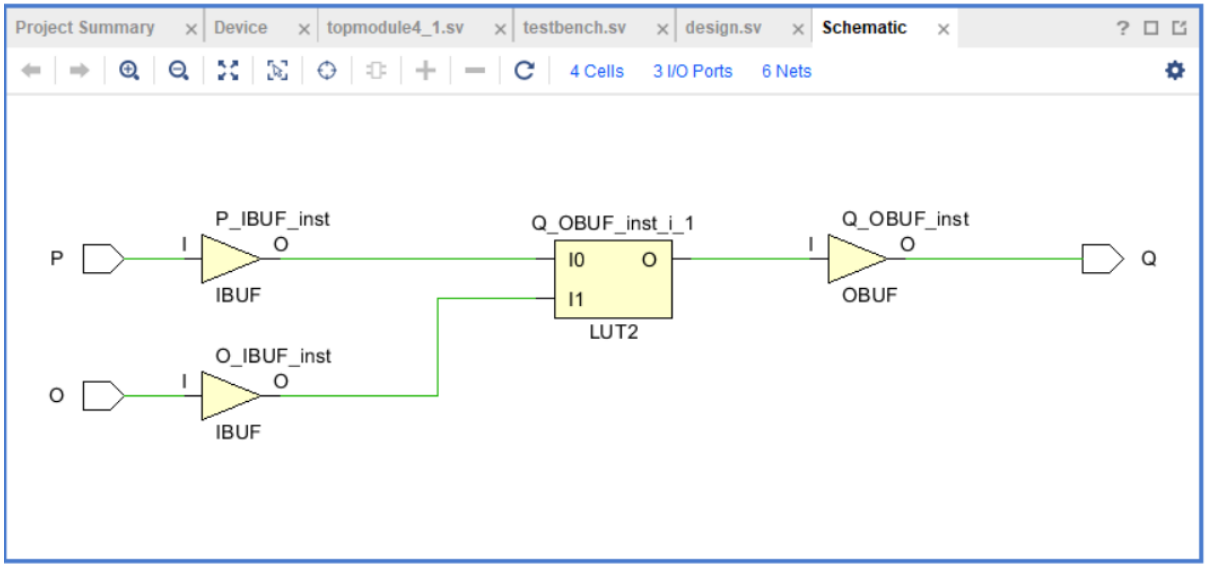
```
XOR Gate Test
O P | Q
---------
0 0 | 0
0 1 | 1
1 0 | 1
1 1 | 0
0 0 | 0
```



XOR Gate Waveform Graph



XOR Gate RTL Schematic



XOR Gate Synthesis Schematic

- XNOR GATE :
  - Design Code :

```
module xnor_gate
        (
        input R,
        input S,
        output T
        );
    assign T = ~ ( R ^ S );
endmodule

// Data Flow Modelling
```

a.Data Flow Modelling

```
module xnor_gate
        (
        input R,
        input S,
        output reg T
        );
        always@(R,S)
        begin
        T =  ~ ( R ^ S );
        end
endmodule
// Behavioural Modelling
```

b. Behavioural Modelling

```
module xnor_gate
        (
        input R,
        input S,
        output T
        );
    xnor xno1(T,R,S);
endmodule

//Structural Modelling
```

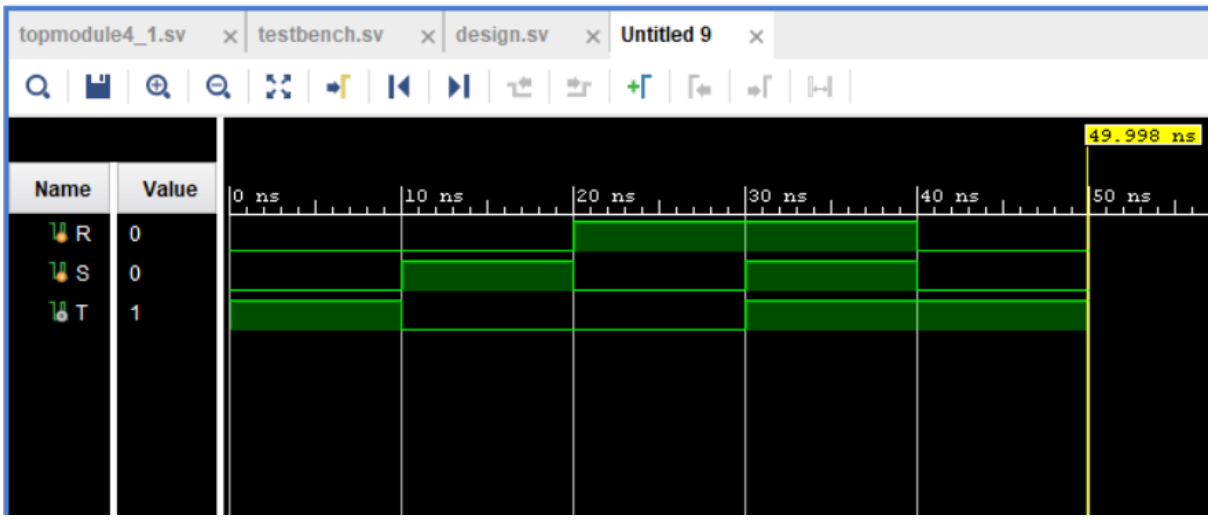
c. Structural Modelling

- Testbench Code :

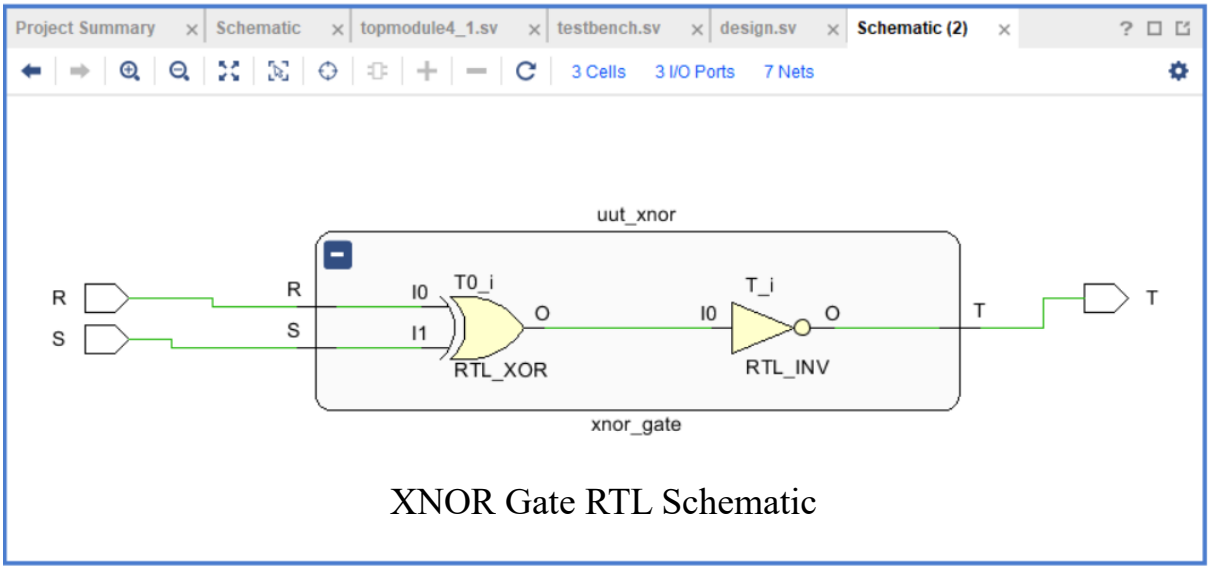
```
module test_gate;
        reg R,S;
        wire T;

        // Instantiate xnor_gate
        xnor_gate uut_xnor
        (
        .R(R),
        .S(S),
        .T(T)
        );
        initial begin
        $dumpfile("dump4_1.vcd");
        $dumpvars();

        // XNOR gate test
        $display("");
        $display("XNOR Gate Test");
        $display("R S | T");
        $display("---------");
        R = 0; S = 0; #10; $display("%b %b | %b", R, S, T);
        R = 0; S = 1; #10; $display("%b %b | %b", R, S, T);
        R = 1; S = 0; #10; $display("%b %b | %b", R, S, T);
        R = 1; S = 1; #10; $display("%b %b | %b", R, S, T);
        R = 0; S = 0; #10; $display("%b %b | %b", R, S, T);
    $finish;
    end
endmodule
```

- Output Truth Table :

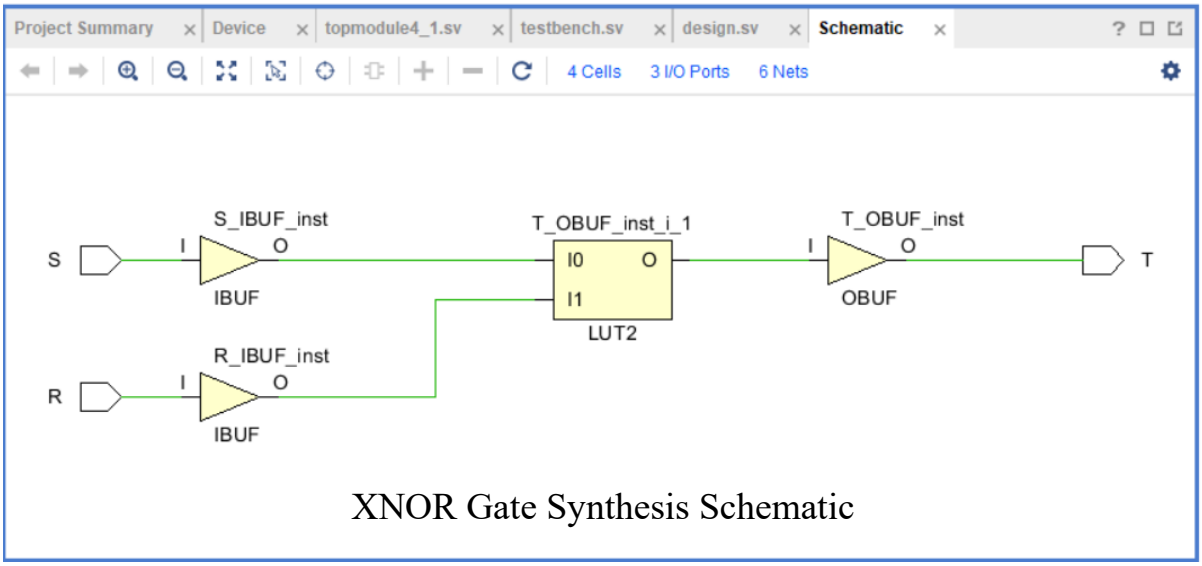
```
XNOR Gate Test
R S | T
---------
0 0 | 1
0 1 | 0
1 0 | 0
1 1 | 1
0 0 | 1
```



XNOR Gate Waveform Graph



XNOR Gate RTL Schematic



XNOR Gate Synthesis Schematic

Part – 2 Boolean Function Implementation

- Function 1 => $f = x'y' + xy$
  - Design Code :

```
module bool_fun_1(x,y,f);
        input x,y;
        output f;

        assign f = ( ~ x & ~ y ) | ( x & y ) ;
endmodule

// Data Flow Modelling
```

a. Data Flow Modelling

```
module bool_fun_1(x,y,f);
        input x,y;
        output reg f;

        always@(x,y)
                begin
                f = ( ~ x & ~ y ) | ( x & y ) ;
                end
endmodule

// Behavioural Modelling
```

b. Behavioural Modelling

```
module bool_fun_1(x,y,f);
        input x,y;
        output f;
        wire a,b,c,d,e;

        not(a,x) ; not(b,y) ; not(c,z);       // a = x' , b = y' ,  c = z'
        and(d,a,b) ; and(e,x,y) ; or(f,d,e);
        // d = x'y , e = x,y , f = x'y + xy

endmodule

//Structural Modelling
```

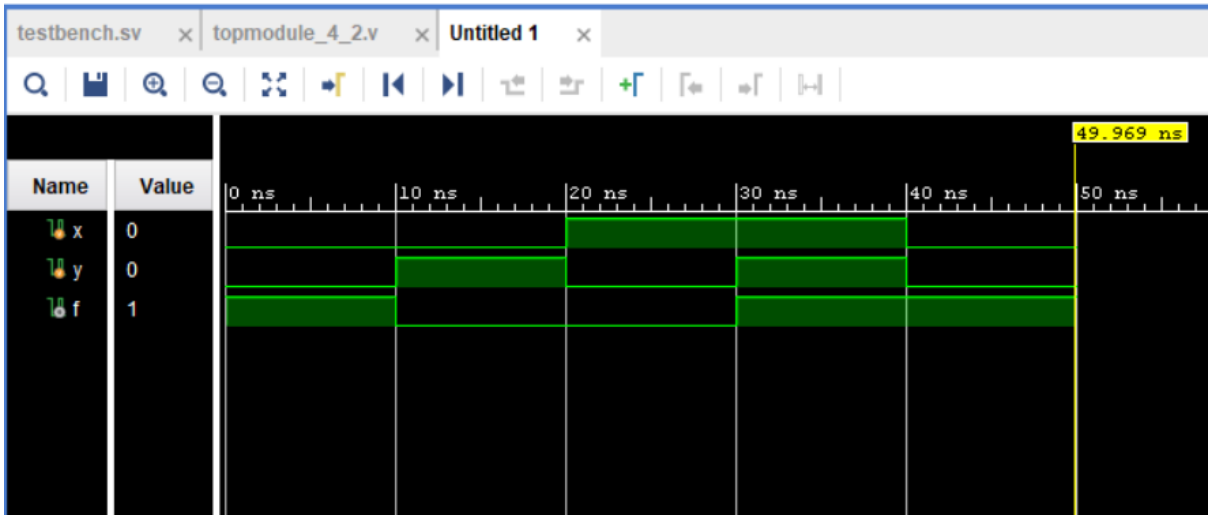
c. Structural Modelling

- Testbench Code :

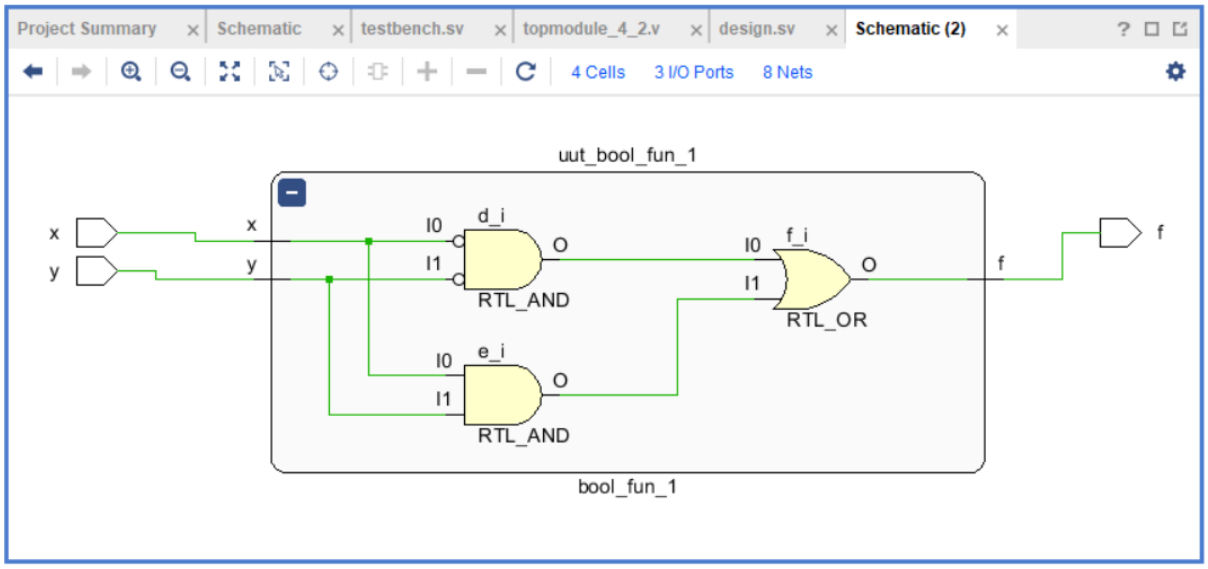
```
module test_fun_1;
        reg x,y;
        wire f;

        // Instantiate bool_fun_1
        bool_fun_1 uut_bool_fun_1
        (
        .x(x),
        .y(y),
        .f(f)
        );
        initial begin
        $dumpfile("dump4_2.vcd");
        $dumpvars();

        // bool_fun_1 Test
        $display("");
        $display("Boolean Function 1 Test");
        $display("x y | f");
        $display("---------");
        x = 0; y = 0; #10; $display("%b %b | %b", x, y, f);
        x = 0; y = 1; #10; $display("%b %b | %b", x, y, f);
        x = 1; y = 0; #10; $display("%b %b | %b", x, y, f);
        x = 1; y = 1; #10; $display("%b %b | %b", x, y, f);
        x = 0; y = 0; #10; $display("%b %b | %b", x, y, f);
    $finish;
    end
endmodule
```

- Output Truth Table :

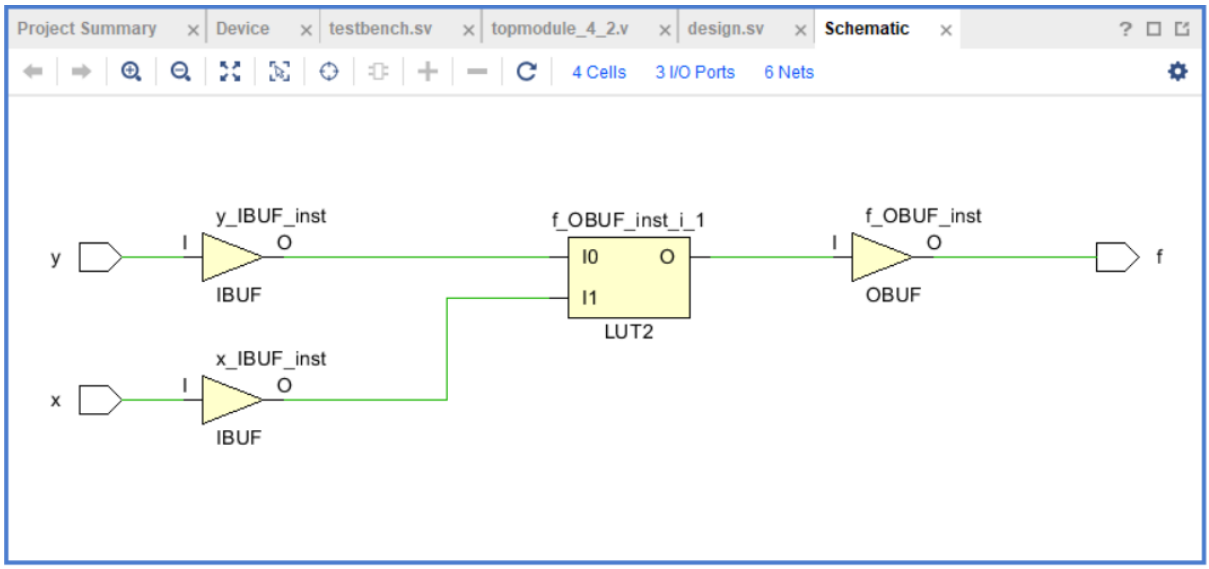
```
Boolean Function 1 Test
x y | f
---------
0 0 | 1
0 1 | 0
1 0 | 0
1 1 | 1
0 0 | 1
```



Boolean Expression 1 => f = x'y' + xy => Waveform Graph



Boolean Expression 1 => f = x'y' + xy => RTL Schematic



Boolean Expression 1 => f = x'y' + xy => Synthesis Schematic

- Function 2 => $g = x'yz + xyz' + x'y'z' + xyz$
  - Design Code :

```
module bool_fun_2(x,y,z,g);
        input x,y,z;
        output g;

        assign g = ( ~ x & y & z ) | ( x & y & ~ z ) | ( ~ x & ~ y & ~ z ) | ( x & y & z );
endmodule

// Data Flow Modelling
```

a. Data Flow Modelling

```
module bool_fun_2(x,y,z,g);
        input x,y,z;
        output reg g;

        always@(x,y,z)
                begin
                g = ( ~ x & y & z ) | ( x & y & ~ z ) | ( ~ x & ~ y & ~ z ) | ( x & y & z ) ;
                end
endmodule

// Behavioural Modelling
```

b. Behavioural Modelling

```
module bool_fun_2(x,y,z,g);
        input x,y,z;
        output g;
        wire a,b,c;
        wire p,q,r,s;

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

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

endmodule

//Structural Modelling
```

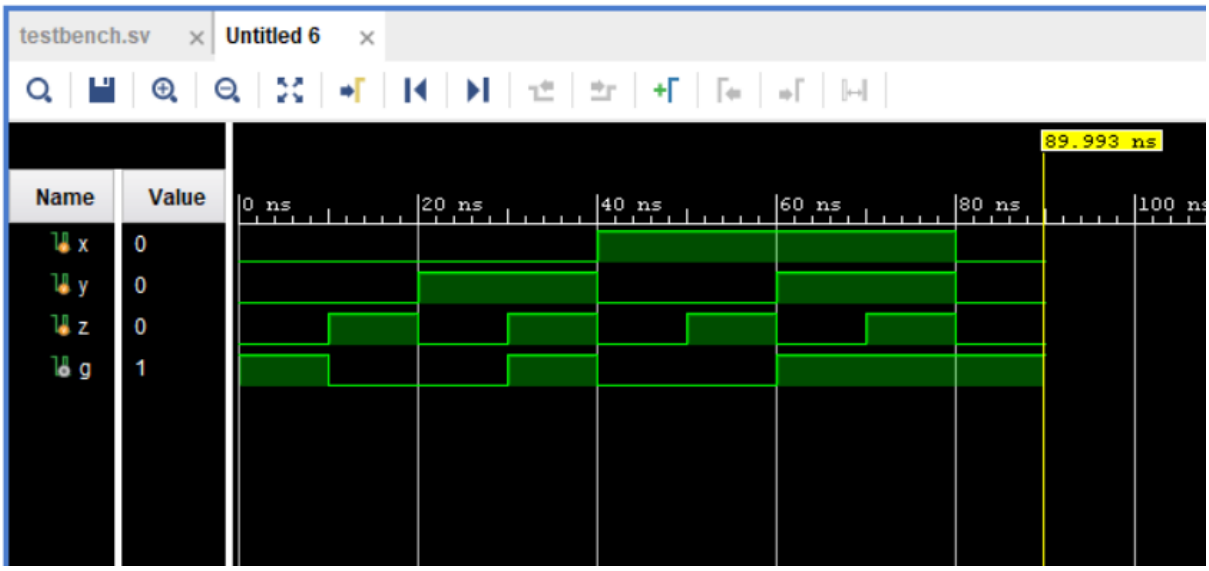
c. Structural Modelling

- Testbench Code :

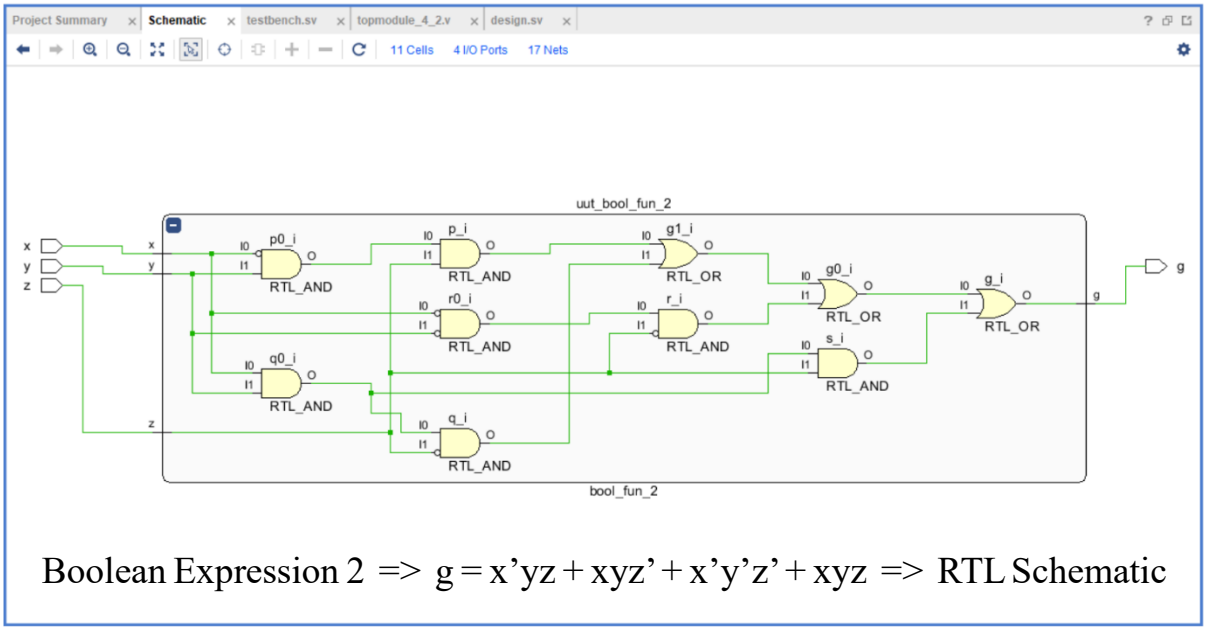
```
module test_fun_2;
        reg x,y,z;
        wire g;

        // Instantiate bool_fun_1
        bool_fun_2 uut_bool_fun_2
        (
        .x(x),
        .y(y),
        .z(z),
        .g(g)
        );
        initial begin
        $dumpfile("dump4_2.vcd");
        $dumpvars();

        // bool_fun_2 Test
        $display("");
        $display("Boolean Function 2 Test");
        $display("x y z | g");
        $display("------------");
        x = 0; y = 0; z = 0; #10; $display("%b %b %b | %b", x, y, z, g);
        x = 0; y = 0; z = 1; #10; $display("%b %b %b | %b", x, y, z, g);
        x = 0; y = 1; z = 0; #10; $display("%b %b %b | %b", x, y, z, g);
        x = 0; y = 1; z = 1; #10; $display("%b %b %b | %b", x, y, z, g);
        x = 1; y = 0; z = 0; #10; $display("%b %b %b | %b", x, y, z, g);
        x = 1; y = 0; z = 1; #10; $display("%b %b %b | %b", x, y, z, g);
        x = 1; y = 1; z = 0; #10; $display("%b %b %b | %b", x, y, z, g);
        x = 1; y = 1; z = 1; #10; $display("%b %b %b | %b", x, y, z, g);
        x = 0; y = 0; z = 0; #10; $display("%b %b %b | %b", x, y, z, g);
    $finish;
    end
endmodule
```

- Output Truth Table :

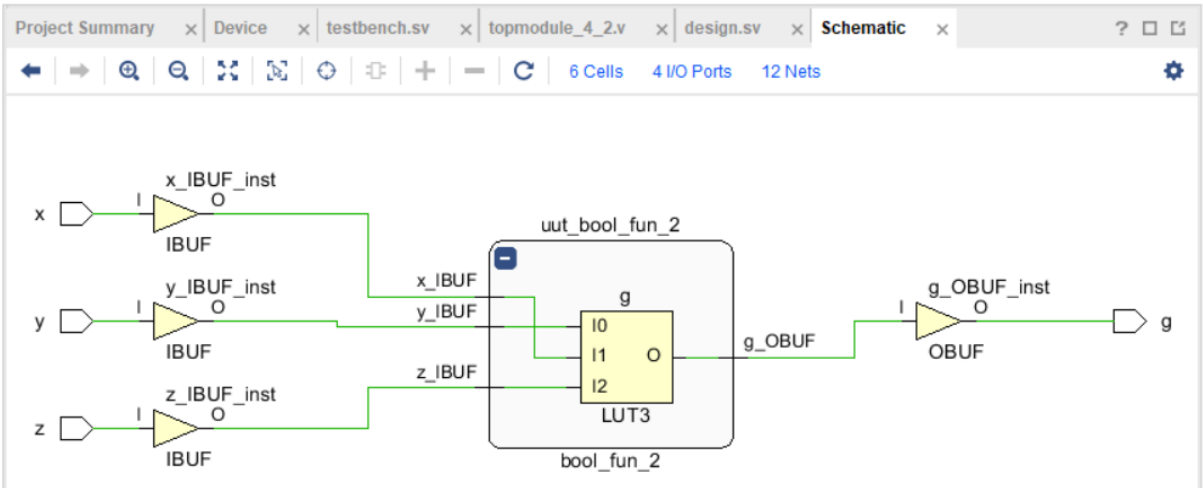
```
Boolean Function 2 Test
x y z | g
------------
0 0 0 | 1
0 0 1 | 0
0 1 0 | 0
0 1 1 | 1
1 0 0 | 0
1 0 1 | 0
1 1 0 | 1
1 1 1 | 1
0 0 0 | 1
```



Boolean Expression 2 => g = x'yz + xyz' + x'y'z' + xyz => Waveform Graph



Boolean Expression 2 => g = x'yz + xyz' + x'y'z' + xyz => RTL Schematic



Boolean Expression 2 => g = x'yz + xyz' + x'y'z' + xyz
=> Synthesis Schematic

Conclusion : In this experiment, all basic logic gates and Boolean functions were implemented using dataflow, behavioral, and structural modeling. Each approach demonstrated a different abstraction level, helping us understand how the same logic can be described and realized in multiple ways for digital circuit design.

Suggested Reference:

1. *Verilog HDL: A Guide to Digital Design and Synthesis*, Pearson - Samir Palnitkar
2. *Fundamentals of Digital Logic with Verilog Design*, McGraw-Hill - Stephen Brown & Zvonko Vranesic
3. *Vivado Design Suite User Guide* – HDL Coding Techniques (UG901) - Xilinx Inc.
4. IEEE Standard for Verilog Hardware Description Language (IEEE Std 1364).
5. TutorialsPoint / NPTEL lectures on Verilog and Digital Logic Design (for supplementary learning).

References used by the students:

1. *Fundamentals of Digital Logic with Verilog Design, McGraw-Hill* - Stephen Brown & Zvonko Vranesic
2. *Vivado Design Suite User Guide* – HDL Coding Techniques (UG901) - Xilinx Inc.

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 & Data Flow Modelling).**
b) **Full Adder using Half adder (Structure Method).**
c) **Full Adder (Using Data Flow & Behavioural Modelling).**
d) **Half Subtractor**
e) **Full Subtractor**

Competency and Practical Skills: Basic Digital Design

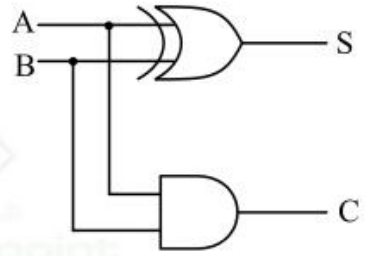
Relevant CO: CO 5 **,** CO 3

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

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

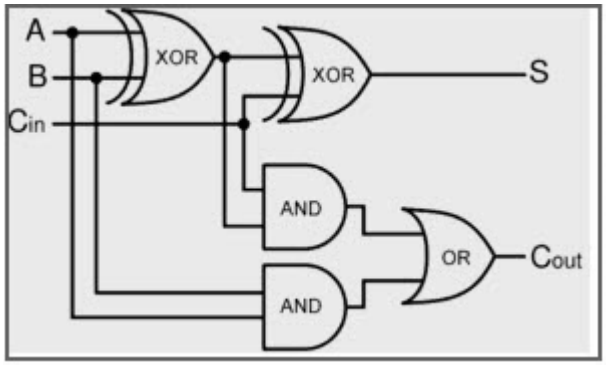
Basic Theory:

Half Adder



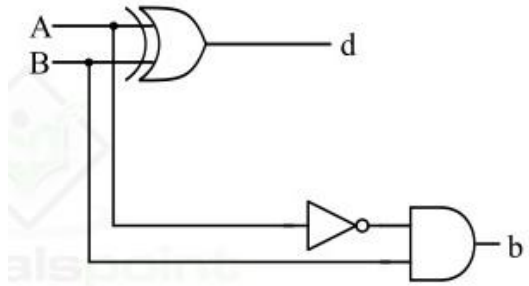
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Full Adder



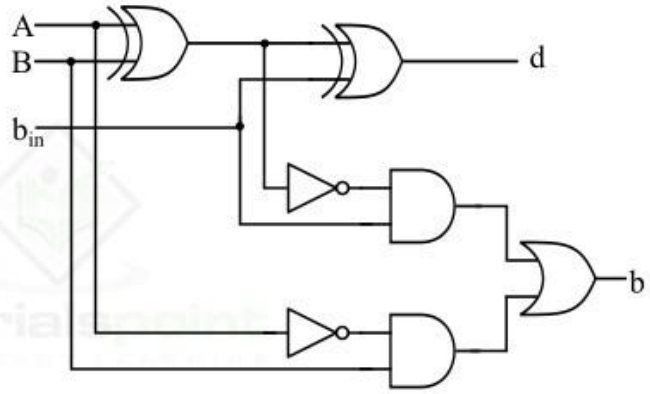
A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Half Subtractor



A	B	D	Borr
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Full Subtractor



A	B	Bin	D	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

1. Adders & Subtractors.

- HALF ADDER :
  - Design Code :

```
module HA_1(A, B, S, C);
        input A, B;
        output S, C;

        xor x1(S, A, B); // sum = A ⊕ B
        and n1(C, A, B); // carry = A.B

endmodule
// Structural Modelling
```

a. Structural Modelling

```
module HA_1(A, B, S, C);
        input A, B;
        output S, C;

        assign S = A ^ B ; // sum = A ⊕ B
        assign C = A & B ; // carry = A.B

endmodule
// Data Flow Modelling
```

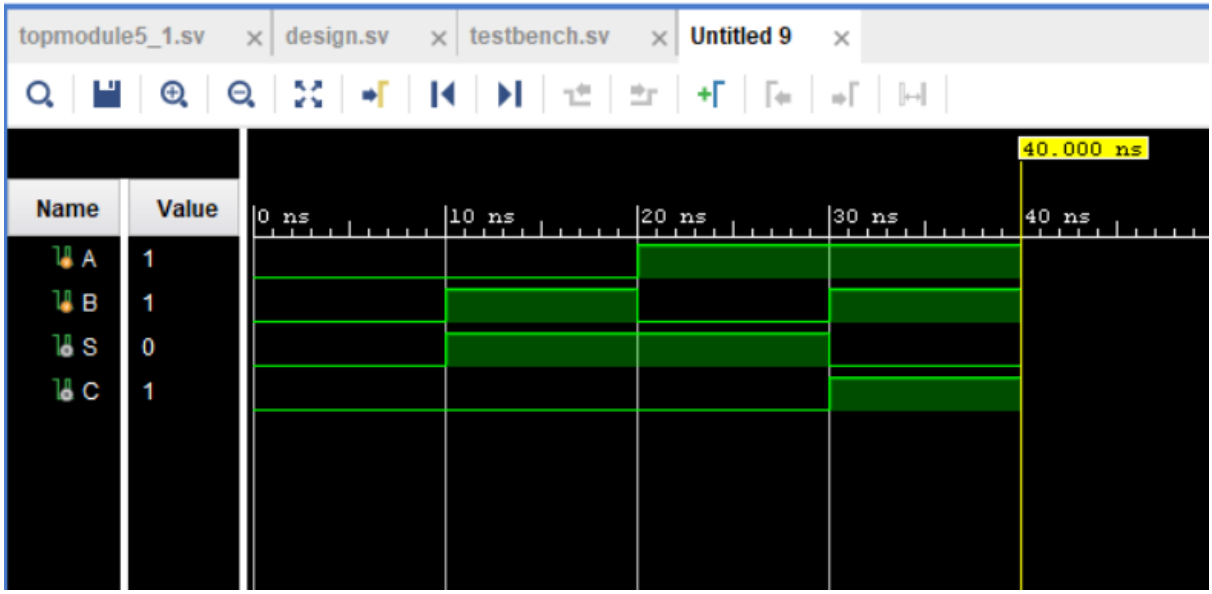
b. Data Flow Modelling

- Testbench Code :

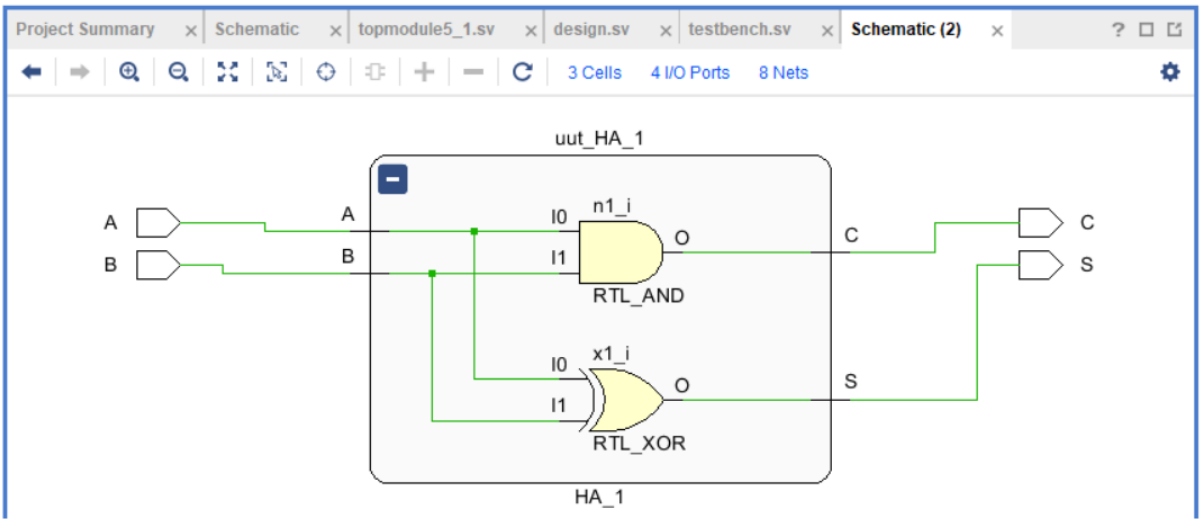
```
module tb_HA_1;
    reg A, B;
    wire S, C;

    // Instantiate HA_1
    HA_1 uut_HA_1
    (
      .A(A),
      .B(B),
      .S(S),
      .C(C)
    );

    initial begin
      $dumpfile("dump5_1.vcd");
      $dumpvars();

      // Half adder test
      $display("");
      $display("Half Adder Test");
      $display("A B | S C");
      $display("-----------");

      // Test Cases
      A = 0; B = 0; #10; $display("%b %b | %b %b", A, B, S, C);
      A = 0; B = 1; #10; $display("%b %b | %b %b", A, B, S, C);
      A = 1; B = 0; #10; $display("%b %b | %b %b", A, B, S, C);
      A = 1; B = 1; #10; $display("%b %b | %b %b", A, B, S, C);

      $finish;
    end
endmodule
```

- Output Truth Table :

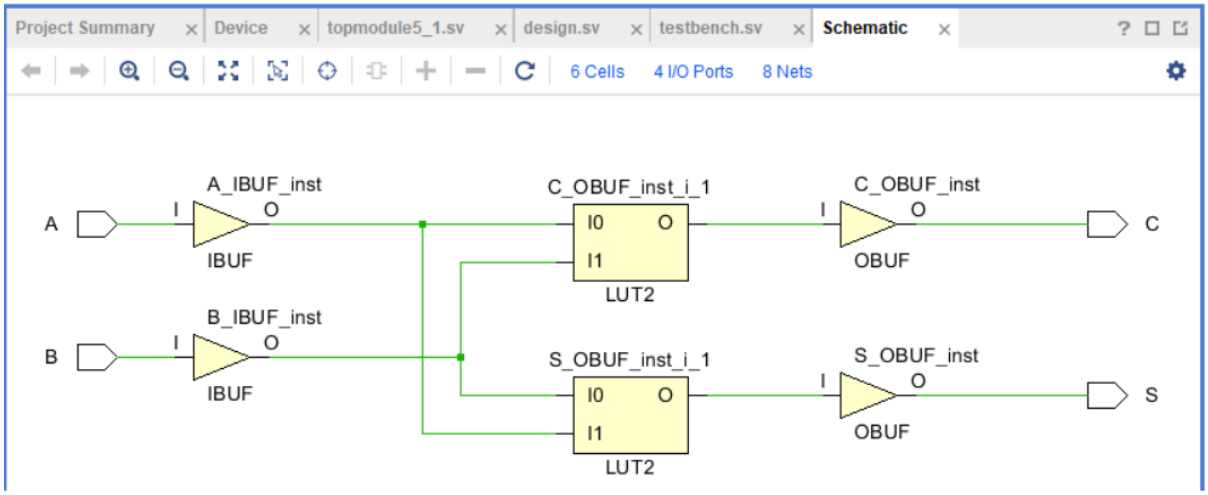
```
Half Adder Test
A B | S C
-----------
0 0 | 0 0
0 1 | 1 0
1 0 | 1 0
1 1 | 0 1
```



Half Adder => S = A ⊕ B |||| C = A.B => Waveform Graph



Half Adder => S = A ⊕ B |||| C = A.B => RTL Schematic



Half Adder => S = A ⊕ B |||| C = A.B => Synthesis Schematic

- FULL ADDER :
  - Design Code :

```
module HA_1(A, B, S, C);
        input A, B;
        output S, C;

        xor x1(S, A, B); // sum = A ⊕ B
        and n1(C, A, B); // carry = A.B

endmodule
// Structural Modelling

module FA_1(A, B, Cin, S, Cout);
        input A, B, Cin;
        output S, Cout;
        wire S1, C1, C2;

        HA_1 HA1 (A, B, S1, C1);
        HA_1 HA2 (S1, Cin, S, C2);
        or or1(Cout,C1,C2);

endmodule
//Structural modelling
```

Full Adder using Half Adder by Structural Modelling

```
module FA_1(A, B, Cin, S, Cout);
    input A, B ,Cin ;
    output S, Cout;

    assign S = A ^ B ^ Cin ; // sum = A ⊕ B ⊕ Cin
    assign Cout = ( A & B ) | ( B & Cin ) | ( A & Cin ) ;
    // carry = A.B + B.Cin + A.Cin

endmodule
// Data Flow Modelling
```

Full Adder (Direct) Data Flow Modelling

```
module FA_1(A, B, Cin, S, Cout);
    input A, B ,Cin ;
    output reg S, Cout;

    always@(A,B,Cin)
        begin
        S = A ^ B ^ Cin ;
        Cout = ( A & B ) | ( B & Cin ) | ( A & Cin ) ;
        end
endmodule
// Behavioral  Modelling
```

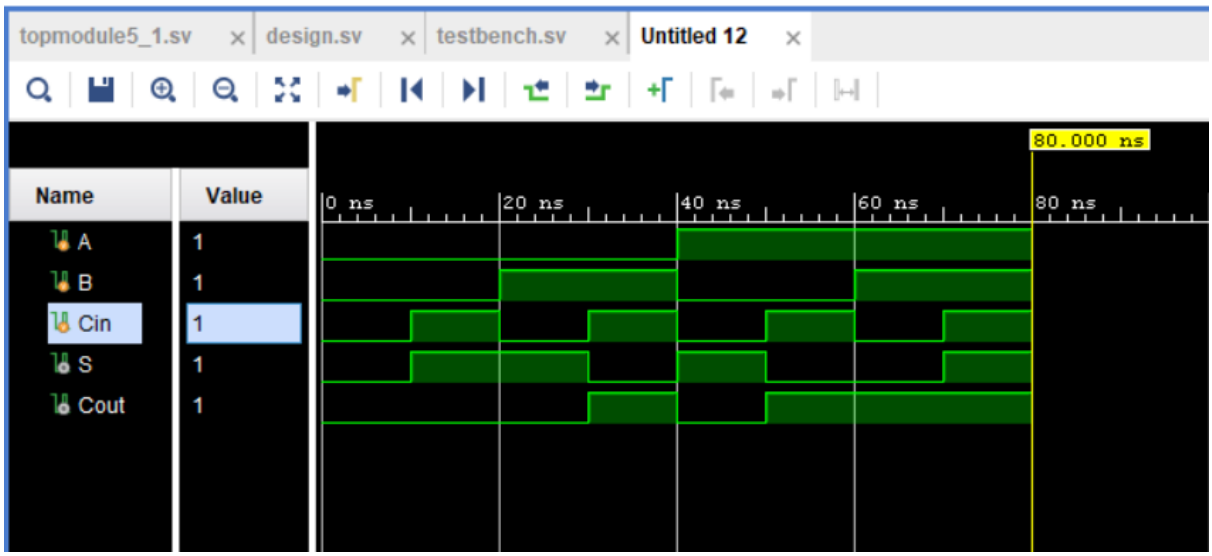
Full Adder (Direct) Behavioral Modelling

- Testbench Code :

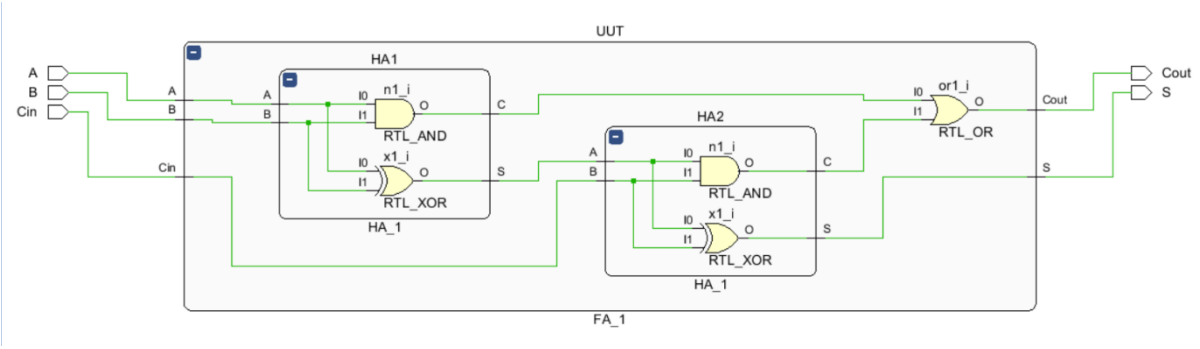
```
module tb_FA_1;
        reg A, B, Cin;
        wire S, Cout;

        // Instantiate Full Adder
    FA_1 uut (
        .A(A),
        .B(B),
        .Cin(Cin),
        .S(S),
        .Cout(Cout)
    );

   initial begin
        $dumpfile("dump5_1.vcd");
        $dumpvars();

        // Full adder test
        $display("Full Adder Test");
        $display("A B Cin | S Cout");
        $display("----------------");

        // Test all 8 combinations
        A=0; B=0; Cin=0; #10; $display("%b %b  %b  | %b  %b", A,B,Cin,S,Cout);
        A=0; B=0; Cin=1; #10; $display("%b %b  %b  | %b  %b", A,B,Cin,S,Cout);
        A=0; B=1; Cin=0; #10; $display("%b %b  %b  | %b  %b", A,B,Cin,S,Cout);
        A=0; B=1; Cin=1; #10; $display("%b %b  %b  | %b  %b", A,B,Cin,S,Cout);
        A=1; B=0; Cin=0; #10; $display("%b %b  %b  | %b  %b", A,B,Cin,S,Cout);
        A=1; B=0; Cin=1; #10; $display("%b %b  %b  | %b  %b", A,B,Cin,S,Cout);
        A=1; B=1; Cin=0; #10; $display("%b %b  %b  | %b  %b", A,B,Cin,S,Cout);
        A=1; B=1; Cin=1; #10; $display("%b %b  %b  | %b  %b", A,B,Cin,S,Cout);

      $finish;
    end
endmodule
```

- Output Truth Table :

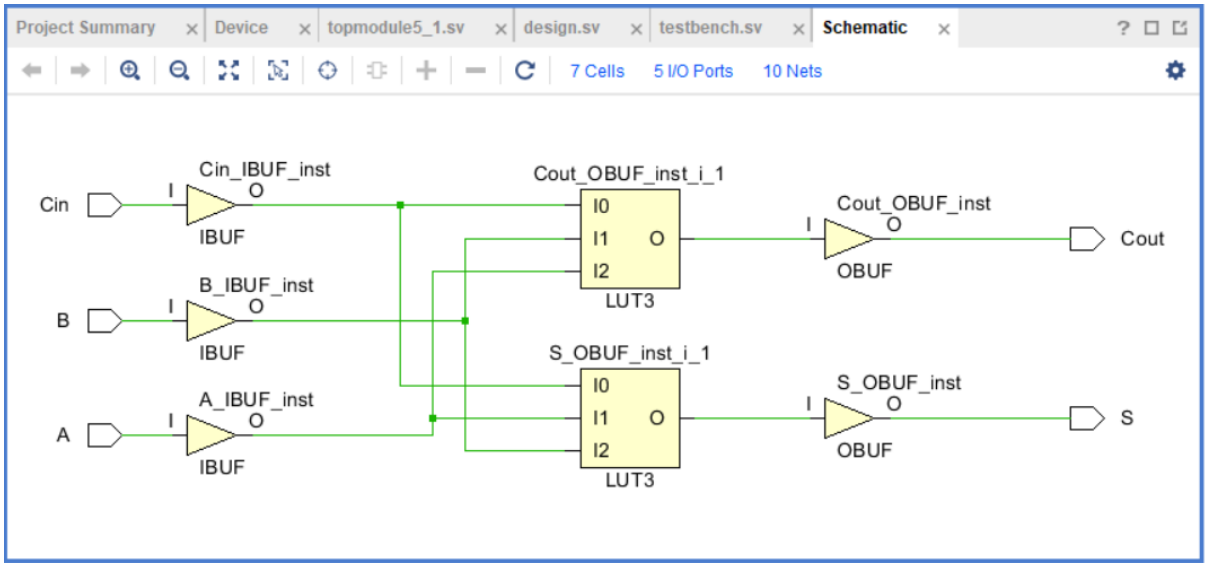
```
Full Adder Test
A B Cin | S Cout
----------------
0 0  0  | 0  0
0 0  1  | 1  0
0 1  0  | 1  0
0 1  1  | 0  1
1 0  0  | 1  0
1 0  1  | 0  1
1 1  0  | 0  1
1 1  1  | 1  1
```



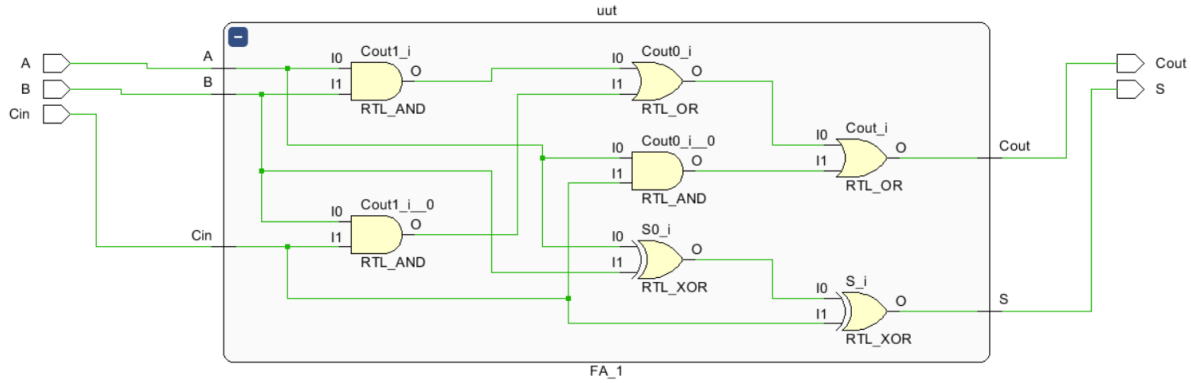
Full Adder => S = A ⊕ B ⊕ Cin |||| Cout = A.B + B.Cin + A.Cin => Waveform Graph



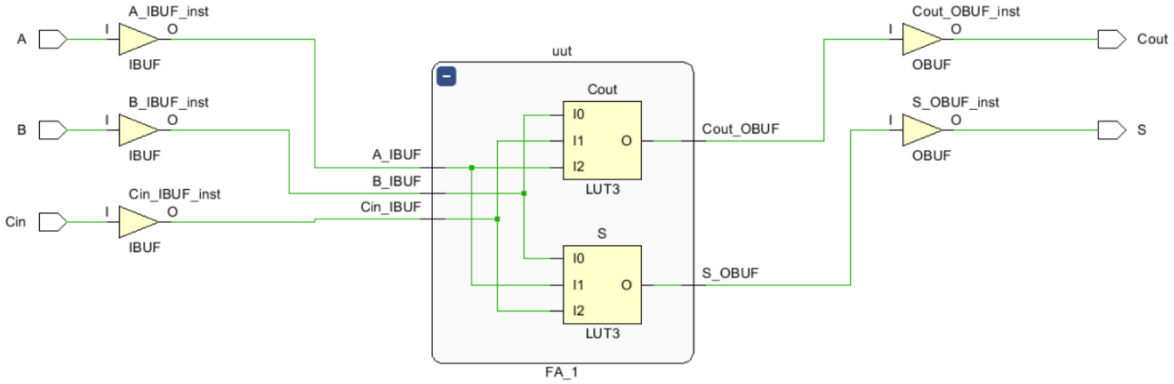
Full Adder => S = A ⊕ B ⊕ Cin |||| Cout = A.B + B.Cin + A.Cin => RTL Schematic (Using Half Adders)



Full Adder => S = A ⊕ B ⊕ Cin |||| Cout = A.B + B.Cin + A.Cin => Synthesis Schematic (Using Half Adders)



Full Adder => S = A ⊕ B ⊕ Cin |||| Cout = A.B + B.Cin + A.Cin => RTL Schematic (Direct)



Full Adder => S = A ⊕ B ⊕ Cin |||| Cout = A.B + B.Cin + A.Cin => Synthesis Schematic (Direct)

- HALF SUBTRACTOR :
  - Design Code :

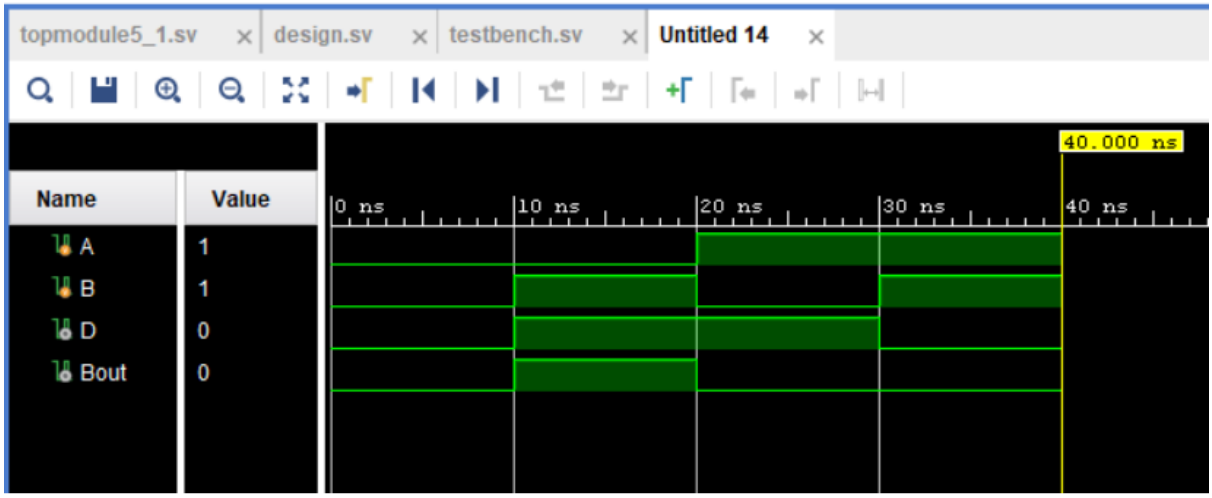
```
module HS_1(A, B, D, Bout);
        input A, B;
        output D, Bout;

        assign D = A ^ B ;
        // Difference = A ⊕ B
        assign Bout = ~ A & B;
        // Borrow = B > A

endmodule
```

- Testbench Code :

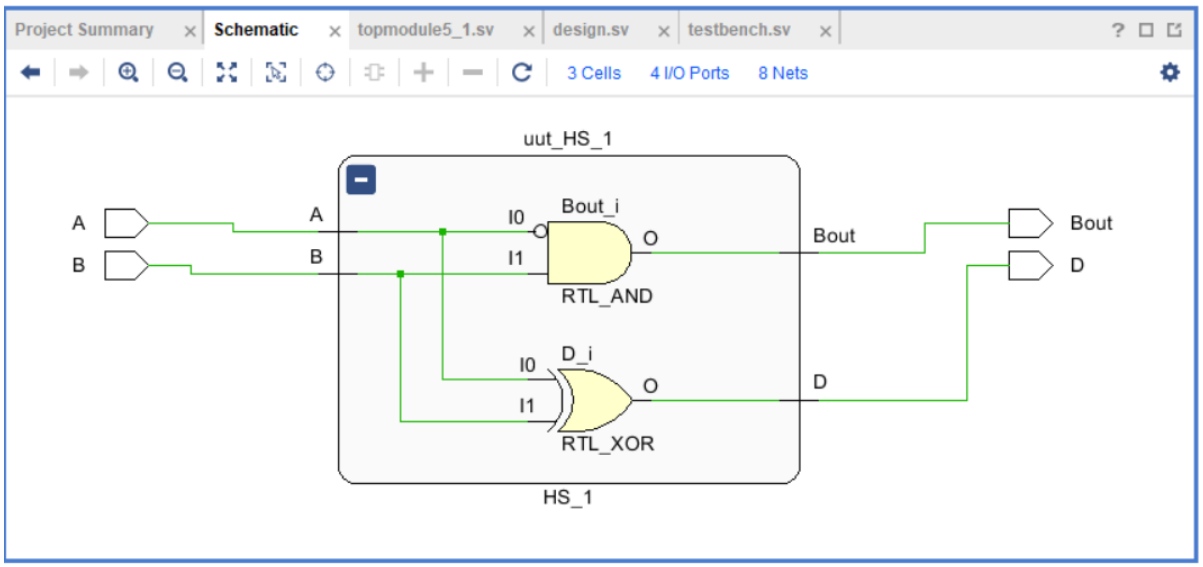
```
module tb_HS_1;
        reg A, B;
        wire D, Bout;

        // Instantiate HS_1
    HS_1 uut_HS_1
        (
        .A(A),
        .B(B),
        .D(D),
        .Bout(Bout)
        );

    initial begin
        $dumpfile("dump5_1.vcd");
        $dumpvars();

        // Half Subtractor test
        $display("");
        $display("Half Subtractor Test");
        $display("A B | D Bout");
        $display("-------------");

        // Test Cases
        A = 0; B = 0; #10; $display("%b %b | %b %b", A, B, D, Bout);
        A = 0; B = 1; #10; $display("%b %b | %b %b", A, B, D, Bout);
        A = 1; B = 0; #10; $display("%b %b | %b %b", A, B, D, Bout);
        A = 1; B = 1; #10; $display("%b %b | %b %b", A, B, D, Bout);
        $finish;
    end
endmodule
```

- Output Truth Table :

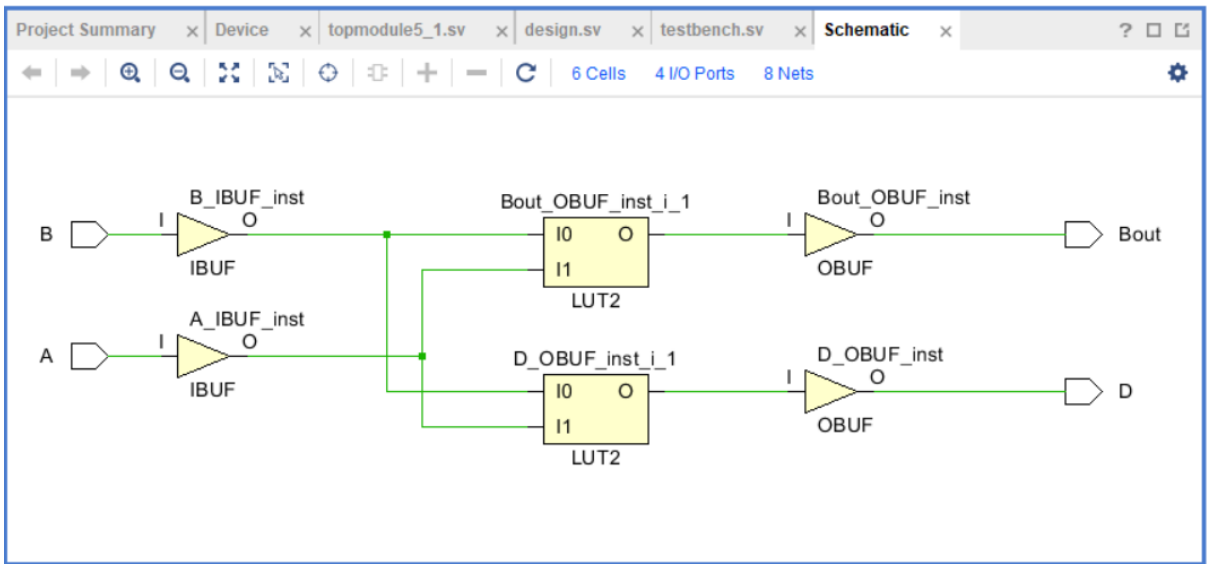
```
Half Subtractor Test
A B | D Bout
-------------
0 0 | 0 0
0 1 | 1 1
1 0 | 1 0
1 1 | 0 0
```



Half Subtractor => D = A ⊕ B |||| Bout = A'.B => Waveform Graph



Half Subtractor => D = A ⊕ B |||| Bout = A'.B => RTL Schematic



Half Subtractor => D = A ⊕ B |||| Bout = A'.B => Synthesis Schematic

- FULL SUBTRACTOR :
  - Design Code :

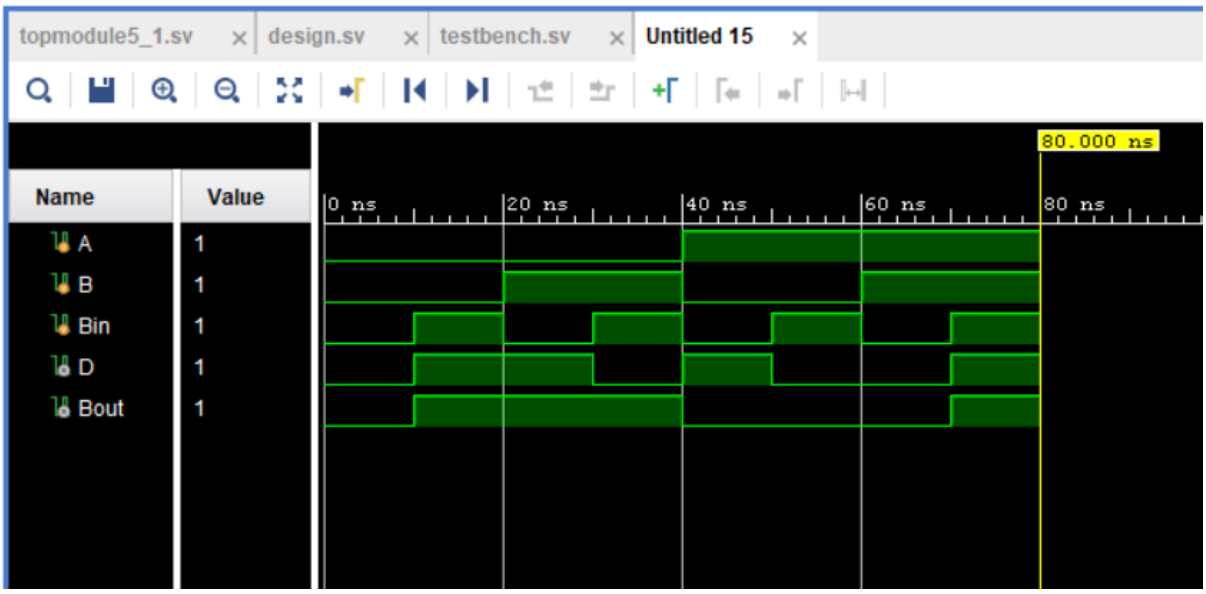
```
module FS_1(A, B, Bin, D, Bout);
        input A, B, Bin;
        output D, Bout;

        assign D = A ^ B ^ Bin; // Difference = A ⊕ B ⊕ Bin
        assign Bout = (~A & B) | (B & Bin) | (~A & Bin);
        // Borrow = A'B + B.Bin + A'Bin

endmodule
```

- Testbench Code :

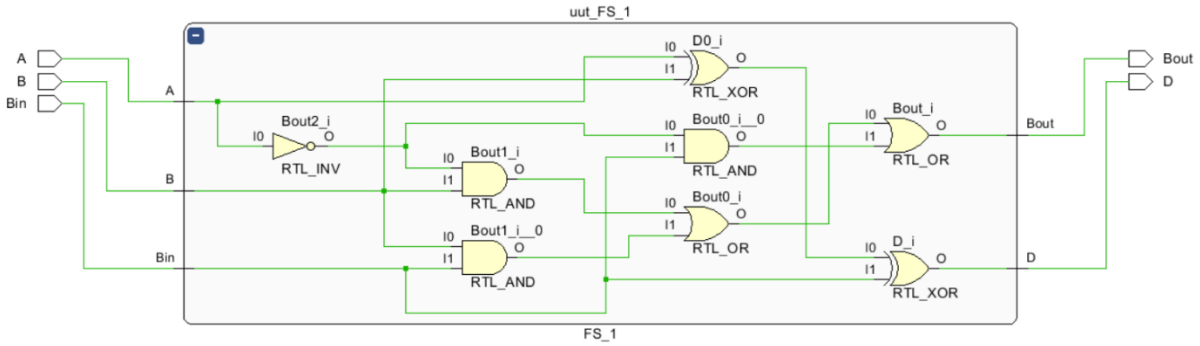
```
module tb_FS_1;
        reg A, B, Bin;
        wire D, Bout;

        // Instantiate Full Subtractor
    FS_1 uut
        (
        .A(A),
        .B(B),
        .Bin(Bin),
        .D(D),
        .Bout(Bout)
        );

    initial begin
        $dumpfile("dump5_1.vcd");
        $dumpvars();

        // Full subtractor test
        $display("Full Subtractor Test");
        $display("A B Bin | D Bout");
        $display("----------------");

        // Test all 8 combinations
        A=0; B=0; Bin=0; #10; $display("%b %b  %b  | %b  %b", A,B,Bin,D,Bout);
        A=0; B=0; Bin=1; #10; $display("%b %b  %b  | %b  %b", A,B,Bin,D,Bout);
        A=0; B=1; Bin=0; #10; $display("%b %b  %b  | %b  %b", A,B,Bin,D,Bout);
        A=0; B=1; Bin=1; #10; $display("%b %b  %b  | %b  %b", A,B,Bin,D,Bout);
        A=1; B=0; Bin=0; #10; $display("%b %b  %b  | %b  %b", A,B,Bin,D,Bout);
        A=1; B=0; Bin=1; #10; $display("%b %b  %b  | %b  %b", A,B,Bin,D,Bout);
        A=1; B=1; Bin=0; #10; $display("%b %b  %b  | %b  %b", A,B,Bin,D,Bout);
        A=1; B=1; Bin=1; #10; $display("%b %b  %b  | %b  %b", A,B,Bin,D,Bout);

        $finish;
    end
endmodule
```

- Output Truth Table :

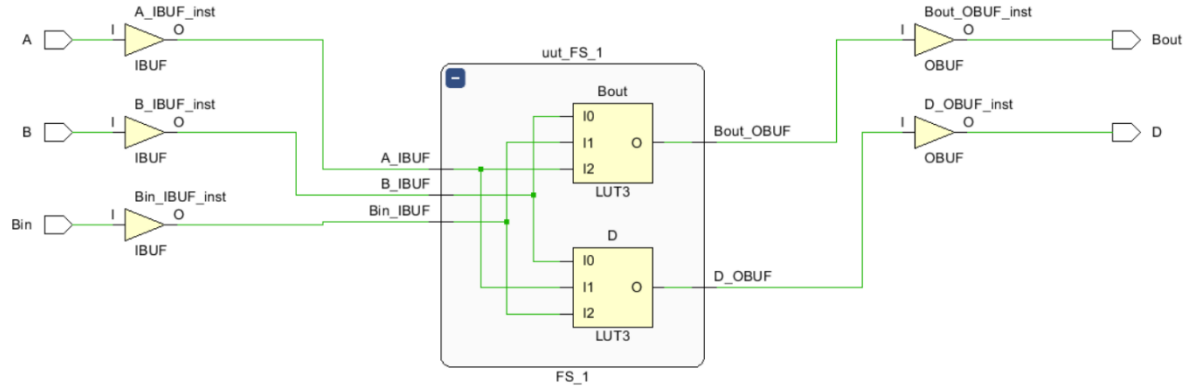
```
Full Subtractor Test
A B Bin | D Bout
----------------
0 0  0  | 0  0
0 0  1  | 1  1
0 1  0  | 1  1
0 1  1  | 0  1
1 0  0  | 1  0
1 0  1  | 0  0
1 1  0  | 0  0
1 1  1  | 1  1
```



Full Subtractor => D = A ⊕ B ⊕ Bin |||| Bout = A'.B + B.Bin + A'.Bin
=> Waveform Graph



Full Subtractor => D = A ⊕ B ⊕ Bin |||| Bout = A'.B + B.Bin + A'.Bin
=> RTL Schematic



Full Subtractor => D = A ⊕ B ⊕ Bin |||| Bout = A'.B + B.Bin + A'.Bin
=> Synthesis Schematic

Conclusion: In this experiment, half and full adders, as well as half and full subtractors, were implemented and tested using Verilog. The circuits correctly produced sum/difference and carry/borrow outputs for all input combinations. Half circuits handle two inputs, while full circuits include carry-in or borrow-in for sequential operations. Using dataflow modeling, the Boolean logic of these arithmetic circuits was successfully represented. Simulation verified the functionality and correctness of each design. This experiment strengthened understanding of basic combinational circuits and their role in digital arithmetic operations.

Suggested Reference:

1. *Verilog HDL: A Guide to Digital Design and Synthesis, Pearson* - Samir Palnitkar
2. *Fundamentals of Digital Logic with Verilog Design, McGraw-Hill* - Stephen Brown & Zvonko Vranesic
3. *Vivado Design Suite User Guide – HDL Coding Techniques (UG901)* - Xilinx Inc.
4. IEEE Standard for Verilog Hardware Description Language (IEEE Std 1364).
5. TutorialsPoint / NPTEL lectures on Verilog and Digital Logic Design (for supplementary learning).

References used by the students:

1. *Fundamentals of Digital Logic with Verilog Design, McGraw-Hill* - Stephen Brown & Zvonko Vranesic
2. *Vivado Design Suite User Guide – HDL Coding Techniques (UG901*) - Xilinx Inc.

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

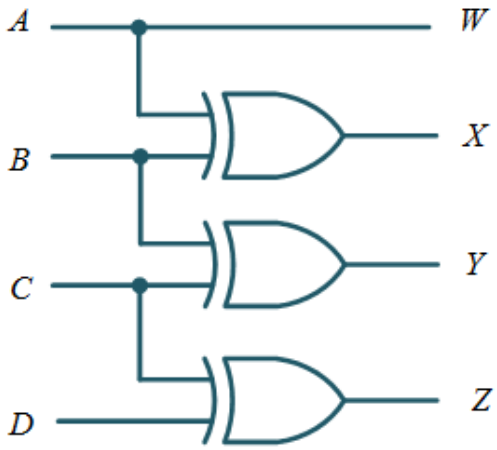
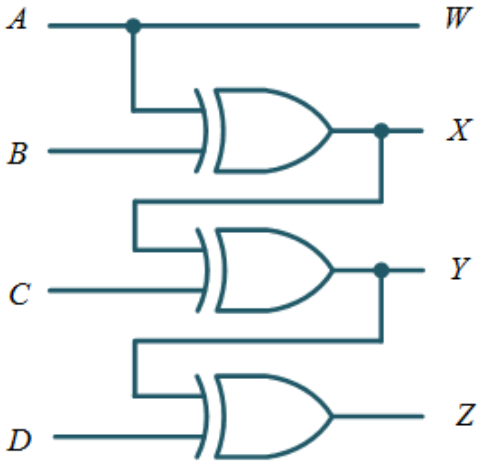
Relevant CO: CO 5 , CO 3

Objectives: Designing of code converters

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

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
	

➢ BINARY TO GRAY ENCODER :

- Design Code :

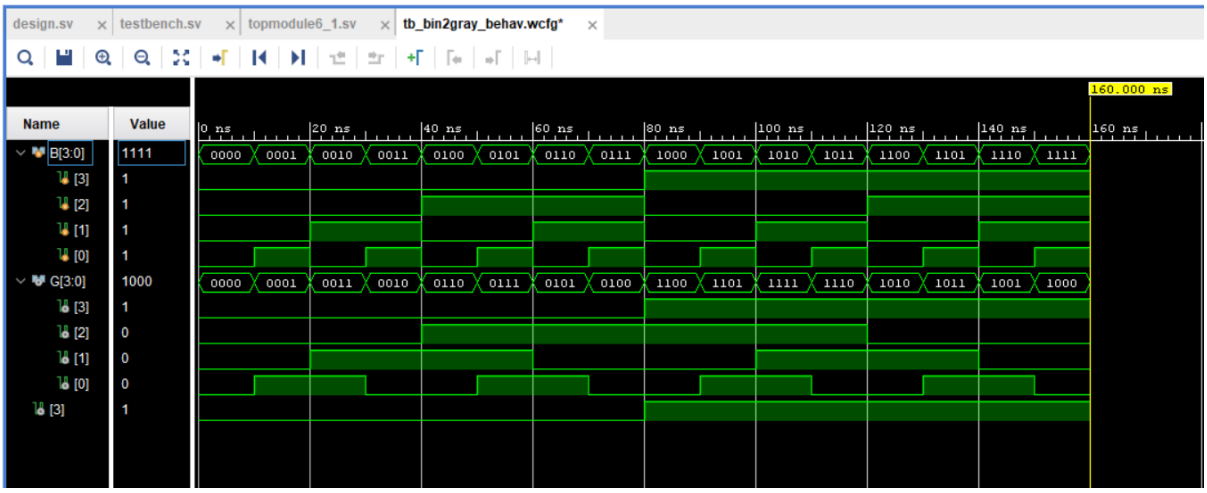
```
module bin2gray (B,G);
        input [3:0] B;
        output [3:0] G;

        // Gray code conversion
        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
```

- Output Truth Table :

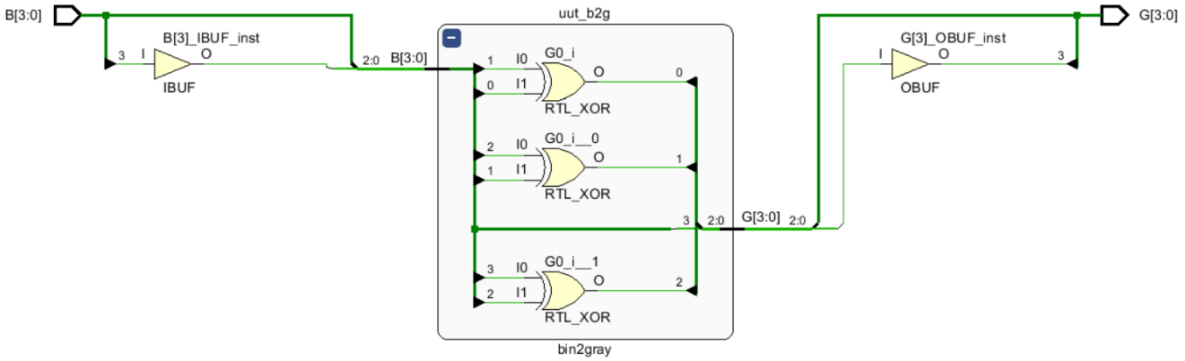
Binary				Gray			
B3	B2	B1	B0	G3	G2	G1	G0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

- Testbench Code :

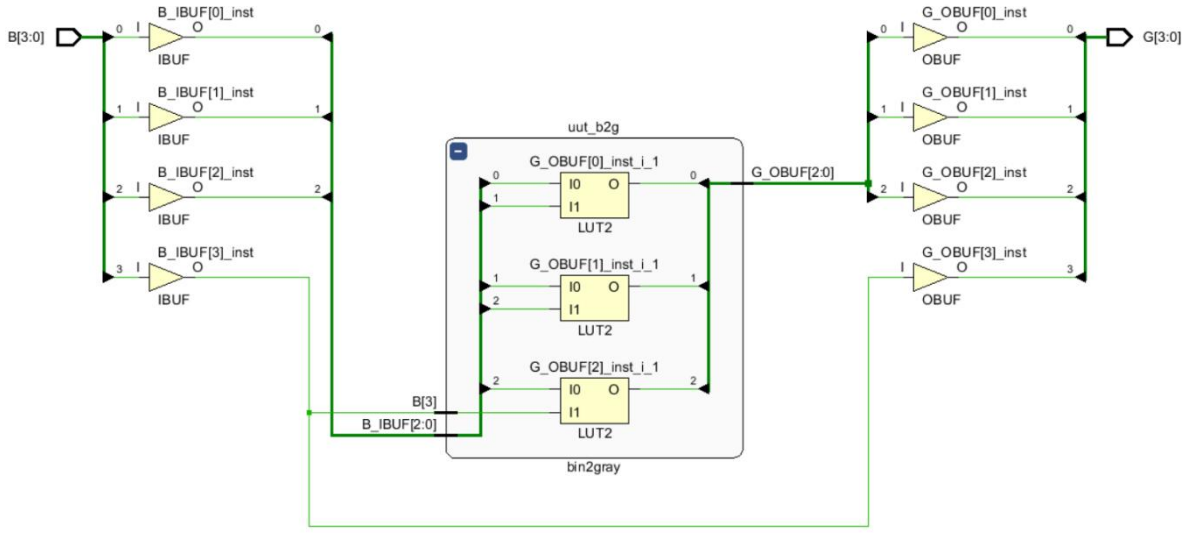
```
module tb_bin2gray;
        reg  [3:0] B;
        wire [3:0] G;
        integer i;

        // Instantiate Binary to Gray Encoder
        bin2gray uut_b2g
                (
                .B(B),
                .G(G)
                );
    initial begin
        $dumpfile("dump6_1.vcd");
        $dumpvars();

        // Binary to Gray encoder test
        $display("   Binary   |    Gray");
        $display("B3 B2 B1 B0 | G3 G2 G1 G0");
        $display("-----------------------------");
        // Apply all 16 combinations, one every 10 ns
        for (i = 0; i < 16; i = i+1)
   begin
     B = i ; #10 ; $display(" %b  %b  %b  %b |  %b  %b  %b %b ", B[3], B[2], B[1], B[0],
G[3], G[2], G[1], G[0]);
   end

   $finish;
  end
endmodule
```



Binary to Gray Encoder => G3 = B3 || G2 = B3 ⊕ B2 || G1 = B2 ⊕ B1 || G0 = B1 ⊕ B0 || => Waveform Graph



Binary to Gray Encoder => G3 = B3 || G2 = B3 ⊕ B2 || G1 = B2 ⊕ B1 || G0 = B1 ⊕ B0 || => RTL Schematic



Binary to Gray Encoder => G3 = B3 || G2 = B3 ⊕ B2 || G1 = B2 ⊕ B1 || G0 = B1 ⊕ B0 || => Synthesis Schematic

- GRAY TO BINARY ENCODER :
  - Design Code :

```
module gray2bin (G, B);
        input [3:0] G;
        output [3:0] B;

        // Binary conversion
        assign B[3] = G[3];
        assign B[2] = B[3] ^ G[2];
        assign B[1] = B[2] ^ G[1];
        assign B[0] = B[1] ^ G[0];
endmodule
```

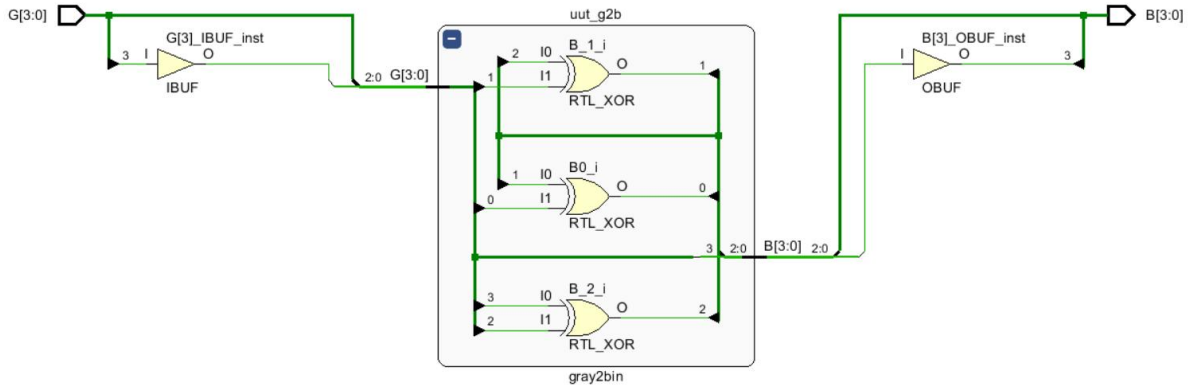
- Testbench Code :

```
module tb_gray2bin;
        reg  [3:0] G;
        wire [3:0] B;
        integer i;

        // Instantiate Gray to Binary Encoder
        gray2bin uut_g2b
                (
                .G(G),
                .B(B)
                );
    initial begin
        $dumpfile("dump6_2.vcd");
        $dumpvars();

        // Gray to Binary encoder test
        $display("   Gray    |  Binary");
        $display("G3 G2 G1 G0 | B3 B2 B1 B0");
        $display("----------------------------");

        // Apply all 16 combinations, one every 10 ns
        for (i = 0; i < 16; i = i+1)
    begin
    G = i; #10;
    $display(" %b  %b  %b  %b |  %b  %b  %b  %b ", G[3], G[2], G[1], G[0], B[3], B[2],
    B[1], B[0]);
    end
        $finish;
  end
endmodule
```

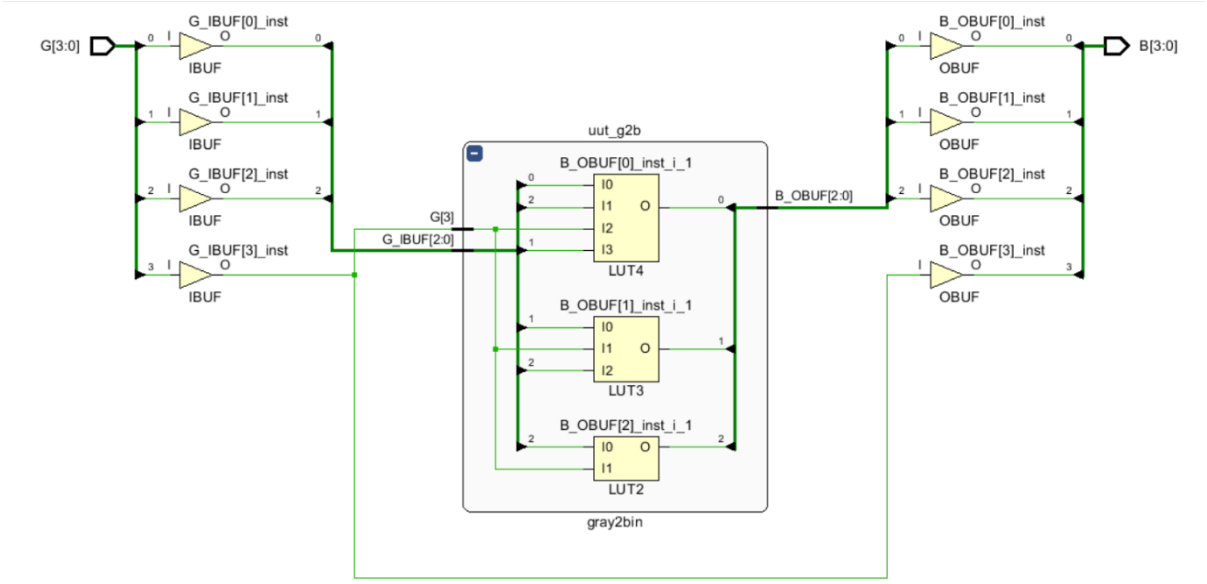
- Output Truth Table :

Gray				Binary			
G3	G2	G1	G0	B3	B2	B1	B0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	1
1	0	0	0	1	1	1	1
1	0	0	1	1	1	1	0
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	1
1	1	1	0	1	0	1	1
1	1	1	1	1	0	1	0

Gray to Binary Encoder => B3 = G3 || B2 = B3 ⊕ G2 || B1 = B2 ⊕ G1 || B0 = B1 ⊕ G0 || => Waveform Graph



Gray to Binary Encoder => B3 = G3 || B2 = B3 ⊕ G2 || B1 = B2 ⊕ G1 || B0 = B1 ⊕ G0 || => RTL Schematic



Gray to Binary Encoder => B3 = G3 || B2 = B3 ⊕ G2 || B1 = B2 ⊕ G1 || B0 = B1 ⊕ G0 || => Synthesis Schematic

Conclusion: The experiment successfully implemented 4-bit **Binary-to-Gray** and **Gray-to-Binary encoders**. The Binary-to-Gray encoder converted binary numbers to Gray code, ensuring that only one bit changes between successive values, which helps reduce errors in digital systems. The Gray-to-Binary encoder correctly reconstructed the original binary numbers using XOR operations. Testbench simulations verified all possible input combinations and confirmed accurate conversions. This experiment highlights the practical use of Gray codes in digital circuits and demonstrates efficient bidirectional conversion between binary and Gray code.

Suggested Reference:

1. *Verilog HDL: A Guide to Digital Design and Synthesis, Pearson* - Samir Palnitkar
2. *Fundamentals of Digital Logic with Verilog Design, McGraw-Hill* - Stephen Brown & Zvonko Vranesic
3. *Vivado Design Suite User Guide – HDL Coding Techniques (UG901)* - Xilinx Inc.
4. IEEE Standard for Verilog Hardware Description Language (IEEE Std 1364).
5. TutorialsPoint / NPTEL lectures on Verilog and Digital Logic Design (for supplementary learning).

References used by the students:

1. *Fundamentals of Digital Logic with Verilog Design, McGraw-Hill* - Stephen Brown & Zvonko Vranesic
2. *Vivado Design Suite User Guide – HDL Coding Techniques (UG901*) - Xilinx Inc.

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 / Altera (Intel) Tools.

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.

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.

- 2:1 MULTIPLEXER
  - Design Code :

```
module mux2to1(a,b,sel,y);
        input a,b,sel;
        output y;

        assign y = (sel) ? b : a;
endmodule

// Data Flow Modelling
```

Data Flow Modelling

```
module mux2to1(a,b,sel,y);
        input a,b,sel;
        output reg y;

    always@(a,b,sel)
        begin
        y = (sel) ? b : a;
        end
endmodule

// Behavioral Modelling
```

Behavioral Modelling

- Testbench Code :

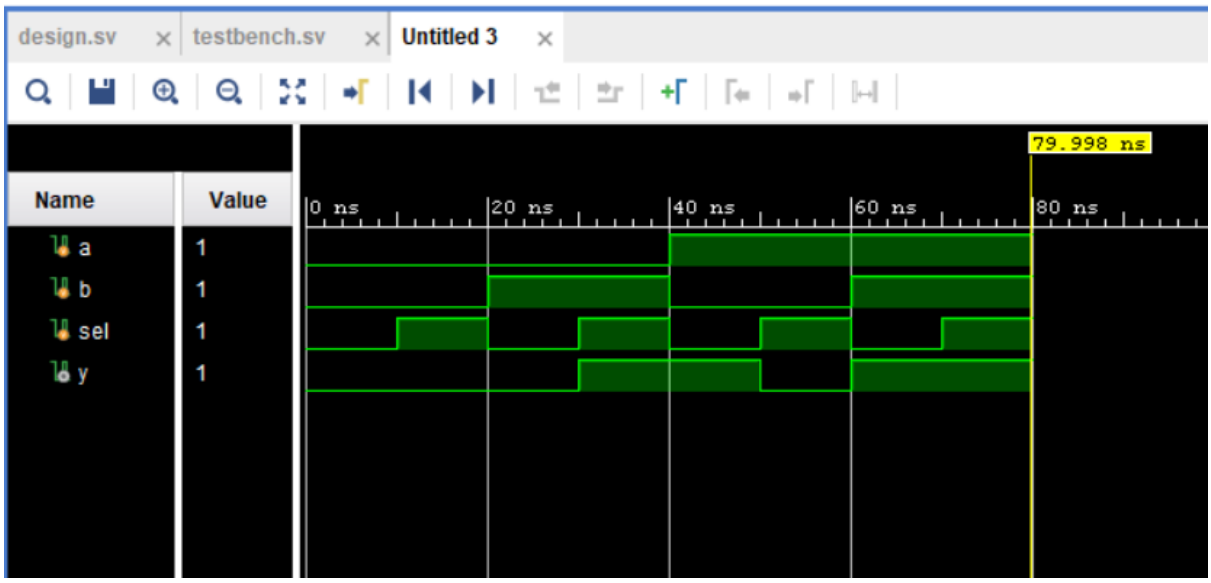
```
module tb_mux2to1;
        // Testbench signals
        reg a;
        reg b;
        reg sel;
        wire y;

        // Instantiate the DUT (Device Under Test)
    mux2to1 uut
        (
        .a(a),
        .b(b),
        .sel(sel),
        y(y)
        );
    initial begin
        // Print header
        $display("A B SEL | Y");
        $display("--------------");

        // Test all combinations
        a = 0; b = 0; sel = 0; #10; $display("%b %b  %b  | %b", a, b, sel, y);
        a = 0; b = 0; sel = 1; #10; $display("%b %b  %b  | %b", a, b, sel, y);
        a = 0; b = 1; sel = 0; #10; $display("%b %b  %b  | %b", a, b, sel, y);
        a = 0; b = 1; sel = 1; #10; $display("%b %b  %b  | %b", a, b, sel, y);
        a = 1; b = 0; sel = 0; #10; $display("%b %b  %b  | %b", a, b, sel, y);
        a = 1; b = 0; sel = 1; #10; $display("%b %b  %b  | %b", a, b, sel, y);
        a = 1; b = 1; sel = 0; #10; $display("%b %b  %b  | %b", a, b, sel, y);
        a = 1; b = 1; sel = 1; #10; $display("%b %b  %b  | %b", a, b, sel, y);
        $finish;
    end
endmodule
```

- Output Truth Table :

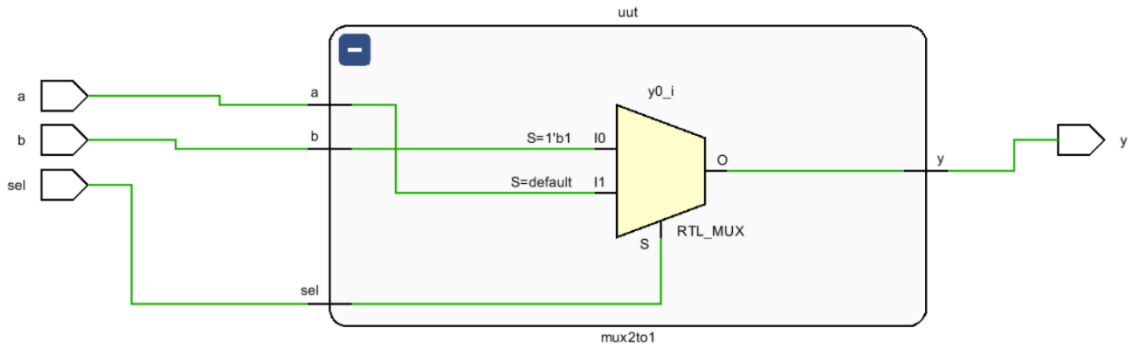
```
A B SEL | Y
------------------
0 0  0  | 0
0 0  1  | 0
0 1  0  | 0
0 1  1  | 1
1 0  0  | 1
1 0  1  | 0
1 1  0  | 1
1 1  1  | 1
```

- Boolean Table :

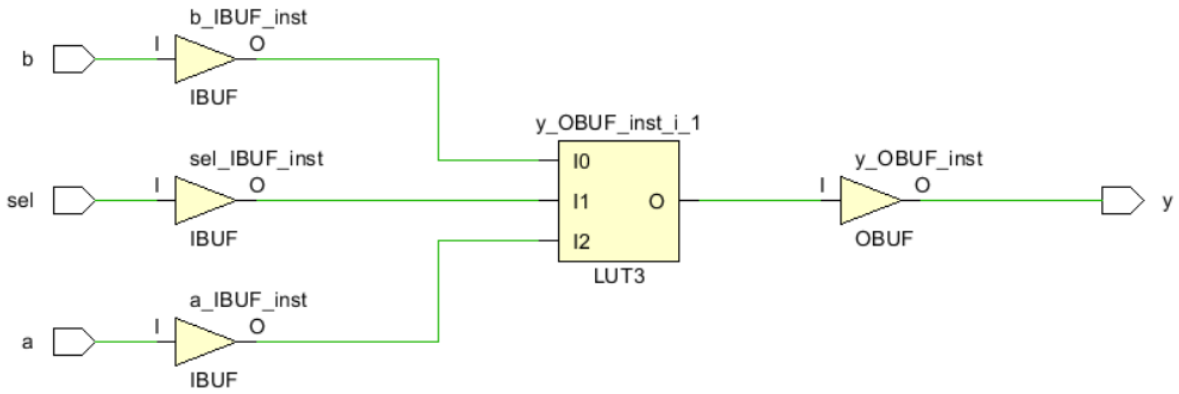
```
SEL  | Y
---------------
 0 | A
 1 | B
```



2x1 MUX => y = a if sel = 0 ||| y = b if sel = 1 ||| y = sel'.a + sel.b |||
=> Waveform Graph



2x1 MUX => y = a if sel = 0 ||| y = b if sel = 1 ||| y = sel'.a + sel.b |||
=> RTL Schematic



2x1 MUX => y = a if sel = 0 ||| y = b if sel = 1 ||| y = sel'.a + sel.b |||
=> Synthesis Schematic

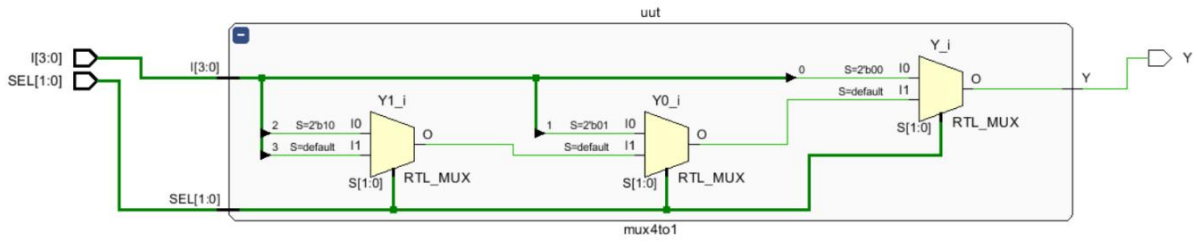
- 4x1 MULTIPLEXER :
  - Design Code :

```
module mux4to1(input [3:0]I , input [1:0]SEL , output Y);

        assign Y = (SEL == 2'b00) ? I[0] : (SEL == 2'b01) ? I[1] : (SEL == 2'b10) ? I[2] :
I[3];
endmodule

// Data Flow Modelling
```

a. Data Flow Modelling

```
module mux4to1(input [3:0]I , input [1:0]SEL , output Y);
      wire nS0, nS1;      // inverted select lines
      wire t0, t1, t2, t3; // product terms

      // Inverters
      not (nS0, SEL[0]);
      not (nS1, SEL[1]);

      // AND terms for each input
      and (t0, nS1, nS0, I[0]);   // when SEL=00 → I0
      and (t1, nS1, SEL[0], I[1]); // when SEL=01 → I1
      and (t2, SEL[1], nS0, I[2]); // when SEL=10 → I2
      and (t3, SEL[1], SEL[0], I[3]); // when SEL=11 → I3

      // OR together the terms
      or  (Y, t0, t1, t2, t3);
   endmodule
// Structural Modelling
```

b. Structural Modelling



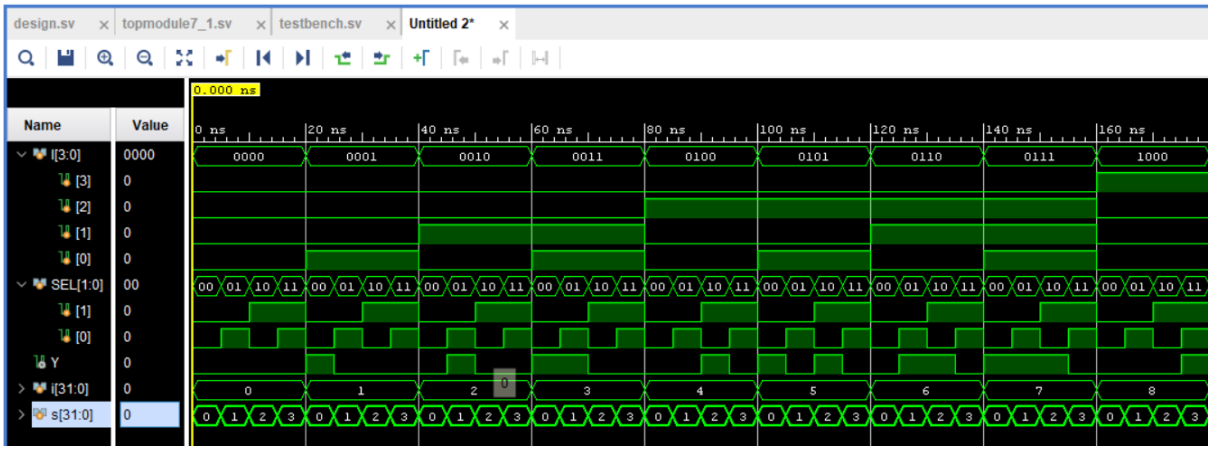
4x1 MUX => Y = S1'.S0'.I0 + S1'.S0.I1 + S1.S0'.I2 + S1.S0.I3 ||| Y = I[0] if sel = 00 ||| Y = I[1] if sel = 01 ||| Y = I[2] if sel = 10 Y = I[3] if sel = 11 ||| => RTL Schematic (Data Flow Model)

- Testbench Code :

```
module tb_mux4to1;
        reg [3:0] I;
        reg [1:0] SEL;
        wire Y;

        // Instantiate the DUT
   mux4to1 uut (.I(I), .SEL(SEL), .Y(Y));

        integer i;  // loop variable for input combinations
        integer s;  // loop variable for select lines

    initial begin
        $display("SEL  I3 I2 I1 I0 | Y");
        $display("---------------------");

        // Loop through all input combinations
    for (i = 0; i < 16; i = i + 1) begin
        I = i;  // assign input

        // Loop through all select combinations
        for (s = 0; s < 4; s = s + 1) begin
                SEL = s; #5; // small delay to allow signal propagation
                $display("%b   %b  %b  %b  %b | %b", SEL, I[3], I[2], I[1], I[0], Y);
        end
    end
        $finish;
    end
endmodule
```

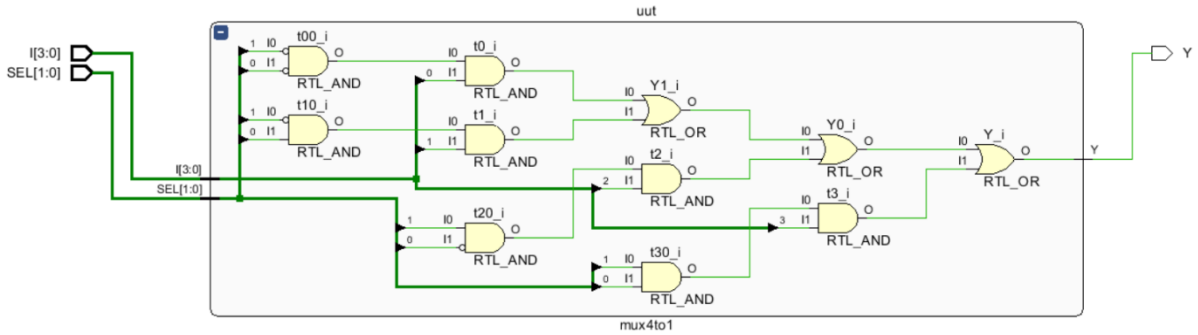
- Boolean Table :

SEL	Y
00	I[0]
01	I[1]
10	I[2]
11	I[3]

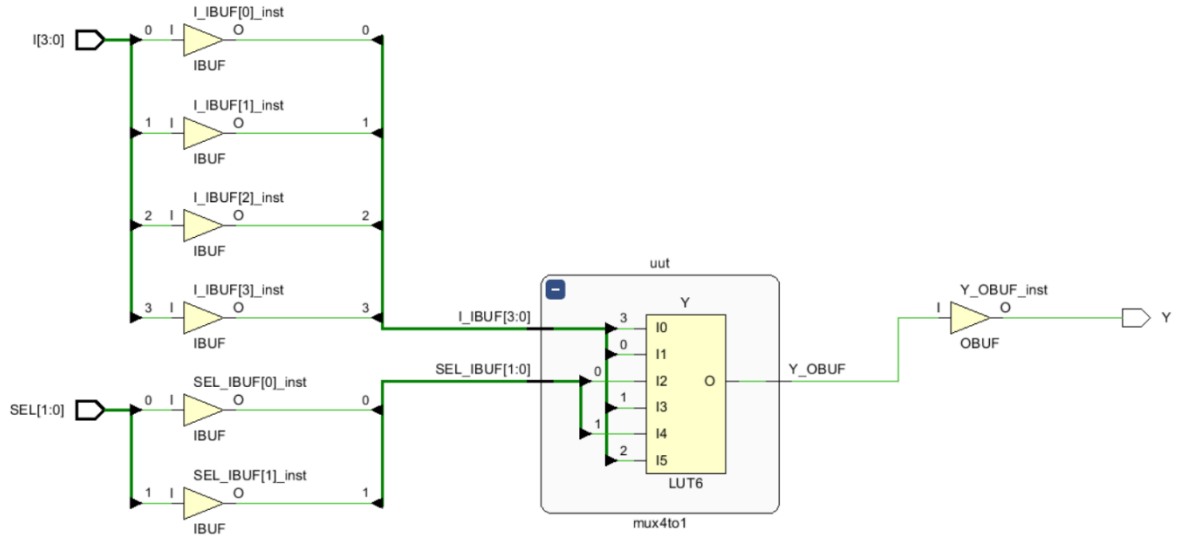


4x1 MUX => Y = S1'.S0'.I0 + S1'.S0.I1 + S1.S0'.I2 + S1.S0.I3 ||| Y = I[0] if sel = 00 ||| Y = I[1] if sel = 01 ||| Y = I[2] if sel = 10 Y = I[3] if sel = 11 ||| => Waveform Graph

4x1 MUX => Y = S1'.S0'.I0 + S1'.S0.I1 + S1.S0'.I2 + S1.S0.I3 ||| Y = I[0] if sel = 00 ||| Y = I[1] if sel = 01 ||| Y = I[2] if sel = 10 Y = I[3] if sel = 11 ||| => Waveform Graph



4x1 MUX => Y = S1'.S0'.I0 + S1'.S0.I1 + S1.S0'.I2 + S1.S0.I3 ||| Y = I[0] if sel = 00 ||| Y = I[1] if sel = 01 ||| Y = I[2] if sel = 10 Y = I[3] if sel = 11 ||| => RTL Schematic (Structural Model)



4x1 MUX => Y = S1'.S0'.I0 + S1'.S0.I1 + S1.S0'.I2 + S1.S0.I3 ||| Y = I[0] if sel = 00 ||| Y = I[1] if sel = 01 ||| Y = I[2] if sel = 10 Y = I[3] if sel = 11 ||| => Synthesis Schematic

- **Output Truth Table :**

SEL	I3	I2	I1	I0	Y
00	0	0	0	0	0
01	0	0	0	0	0
10	0	0	0	0	0
11	0	0	0	0	0
00	0	0	0	1	1
01	0	0	0	1	0
10	0	0	0	1	0
11	0	0	0	1	0
00	0	0	1	0	0
01	0	0	1	0	1
10	0	0	1	0	0
11	0	0	1	0	0
00	0	0	1	1	1
01	0	0	1	1	1
10	0	0	1	1	0
11	0	0	1	1	0
00	0	1	0	0	0
01	0	1	0	0	0
10	0	1	0	0	1
11	0	1	0	0	0
00	0	1	0	1	1
01	0	1	0	1	0
10	0	1	0	1	1
11	0	1	0	1	0
00	0	1	1	0	0
01	0	1	1	0	1
10	0	1	1	0	1
11	0	1	1	0	0
00	0	1	1	1	1
01	0	1	1	1	1
10	0	1	1	1	1
11	0	1	1	1	0
00	1	0	0	0	0
01	1	0	0	0	0
10	1	0	0	0	0
11	1	0	0	0	1
00	1	0	0	1	1
01	1	0	0	1	0
10	1	0	0	1	0
11	1	0	0	1	1
00	1	0	1	0	0
01	1	0	1	0	1
10	1	0	1	0	0
11	1	0	1	0	1
00	1	0	1	1	1
01	1	0	1	1	1
10	1	0	1	1	0
11	1	0	1	1	1
00	1	1	0	0	0
01	1	1	0	0	0
10	1	1	0	0	1
11	1	1	0	0	1
00	1	1	0	1	1
01	1	1	0	1	0
10	1	1	0	1	1
11	1	1	0	1	1
00	1	1	1	0	0
01	1	1	1	0	1
10	1	1	1	0	1
11	1	1	1	0	1
00	1	1	1	1	1
01	1	1	1	1	1
10	1	1	1	1	1
11	1	1	1	1	1

- 8x1 MULTIPLEXER :
  - Design Code :

```
module mux4to1(input [3:0]I , input [1:0]SEL , output Y);
        wire nS0, nS1;      // inverted select lines
        wire t0, t1, t2, t3; // product terms

        // Inverters
        not (nS0, SEL[0]);
        not (nS1, SEL[1]);

        // AND terms for each input
        and (t0, nS1, nS0, I[0]);   // when SEL=00 → I0
        and (t1, nS1, SEL[0], I[1]); // when SEL=01 → I1
        and (t2, SEL[1], nS0, I[2]); // when SEL=10 → I2
        and (t3, SEL[1], SEL[0], I[3]); // when SEL=11 → I3

        // OR together the terms
        or  (Y, t0, t1, t2, t3);
    endmodule

module mux2to1(input a, input b, input sel, output y);
          wire nsel, t0, t1;
          not (nsel, sel);
          and (t0, a, nsel);
          and (t1, b, sel);
          or  (y, t0, t1);
endmodule

module mux8to1(input [7:0] I, input [2:0] SEL, output Y);
          wire y0, y1;

          // Lower 4 inputs → First 4x1 mux
          mux4to1 m1 (.I(I[3:0]), .SEL(SEL[1:0]), .Y(y0));
          // Upper 4 inputs → Second 4x1 mux
          mux4to1 m2 (.I(I[7:4]), .SEL(SEL[1:0]), .Y(y1));

          // Select between y0 and y1 using MSB (SEL[2])
          mux2to1 m3 (.a(y0), .b(y1), .sel(SEL[2]), .y(Y));
endmodule
// 8x1 MUX Using 4x1 , 2x1 , Structural modelling
```

- Testbench Code :

```
module tb_mux8to1;
        reg [7:0] I;
        reg [2:0] SEL;
        wire Y;

        // Instantiate the DUT
        mux8to1 uut (.I(I), .SEL(SEL), .Y(Y));
        integer s; // loop variable for select lines
        integer i; // loop for test inputs

        // Representative input patterns
        reg [7:0] test_inputs [0:3];

    initial begin
        // Define 4 representative test vectors
        test_inputs[0] = 8'b00000000; // Case 1: all zeros
        test_inputs[1] = 8'b11111111; // Case 2: all ones
        test_inputs[2] = 8'b10101010; // Case 3: alternating bits
        test_inputs[3] = 8'b11001100; // Case 4: grouped bits

        $display("Output Truth Table for 8x1 MUX\n");
        // Loop through each test vector
    for (i = 0; i < 4; i = i + 1) begin
        I = test_inputs[i];
        $display("Case %0d: I = %b", i+1, I);
        $display("SEL   I7 I6 I5 I4 I3 I2 I1 I0 | Y");
        $display("-----------------------------------");

        // Sweep select lines
                for (s = 0; s < 8; s = s + 1) begin
                SEL = s; #5;
                $display("%03b   %b  %b  %b  %b  %b  %b  %b  %b | %b", SEL , I[7] ,
        I[6] , I[5] , I[4] , I[3] , I[2] , I[1] , I[0] , Y);
                end
        // Separation line between blocks
        $display("-----------------------------------\n");
    end
        $finish;
    end
endmodule
```

- Boolean Table :

SEL	Y
000	I[0]
001	I[1]
010	I[2]
011	I[3]
100	I[4]
101	I[5]
110	I[6]
111	I[7]

- **Output Truth Table :**

Case 1: I = 00000000

SEL	I7	I6	I5	I4	I3	I2	I1	I0	Y
000	0	0	0	0	0	0	0	0	0
001	0	0	0	0	0	0	0	0	0
010	0	0	0	0	0	0	0	0	0
011	0	0	0	0	0	0	0	0	0
100	0	0	0	0	0	0	0	0	0
101	0	0	0	0	0	0	0	0	0
110	0	0	0	0	0	0	0	0	0
111	0	0	0	0	0	0	0	0	0

Case 2: I = 11111111

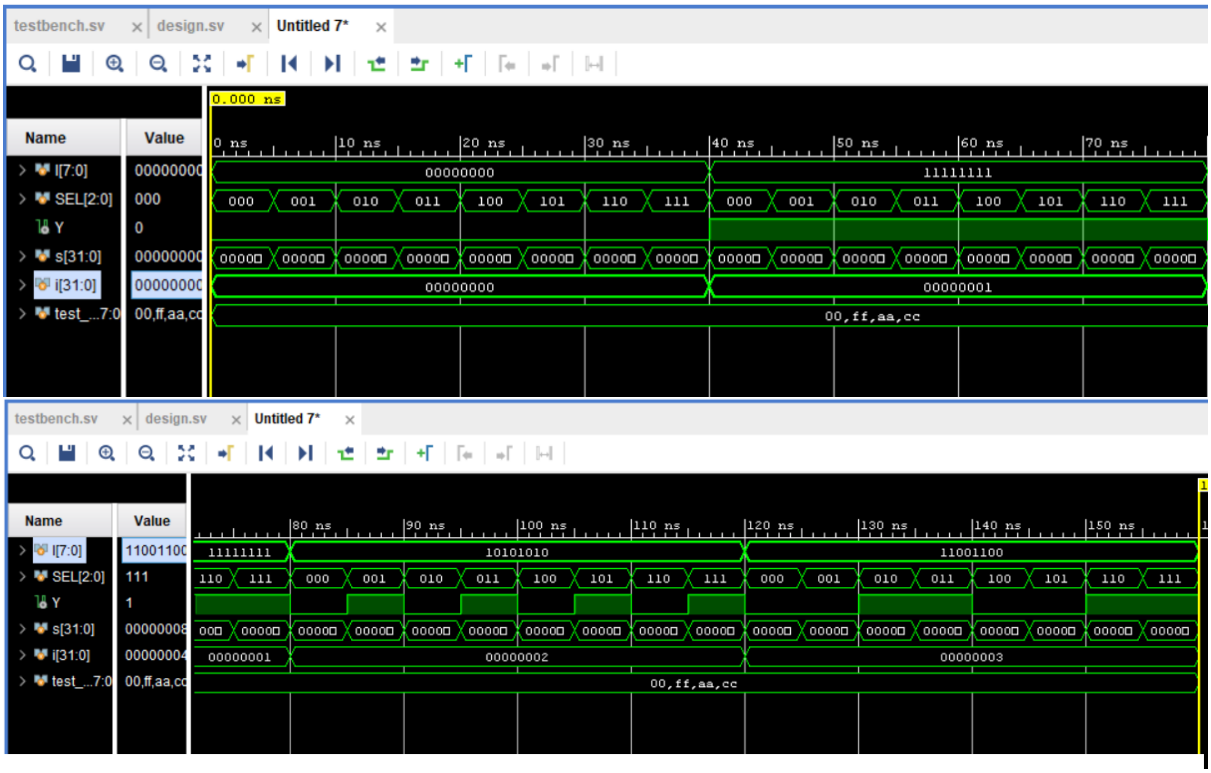
SEL	I7	I6	I5	I4	I3	I2	I1	I0	Y
000	1	1	1	1	1	1	1	1	1
001	1	1	1	1	1	1	1	1	1
010	1	1	1	1	1	1	1	1	1
011	1	1	1	1	1	1	1	1	1
100	1	1	1	1	1	1	1	1	1
101	1	1	1	1	1	1	1	1	1
110	1	1	1	1	1	1	1	1	1
111	1	1	1	1	1	1	1	1	1

Case 3: I = 10101010

SEL	I7	I6	I5	I4	I3	I2	I1	I0	Y
000	1	0	1	0	1	0	1	0	0
001	1	0	1	0	1	0	1	0	1
010	1	0	1	0	1	0	1	0	0
011	1	0	1	0	1	0	1	0	1
100	1	0	1	0	1	0	1	0	0
101	1	0	1	0	1	0	1	0	1
110	1	0	1	0	1	0	1	0	0
111	1	0	1	0	1	0	1	0	1

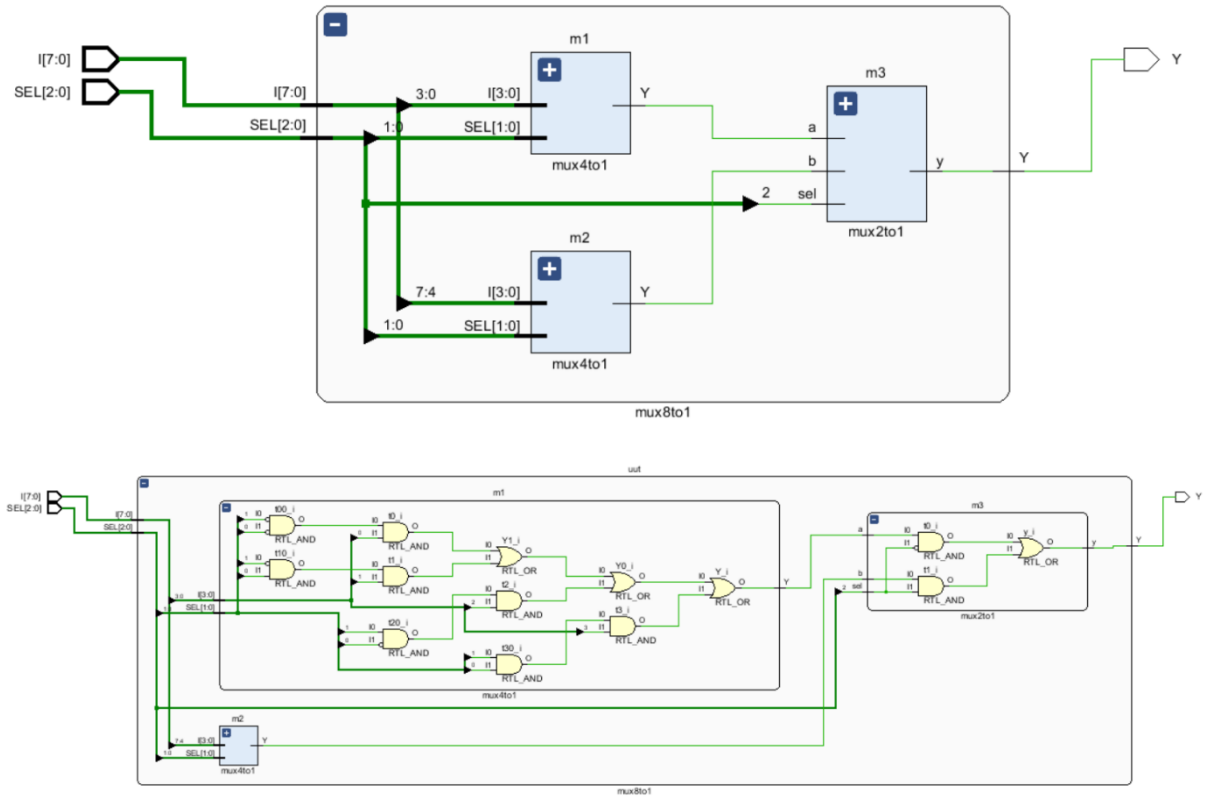
Case 4: I = 11001100

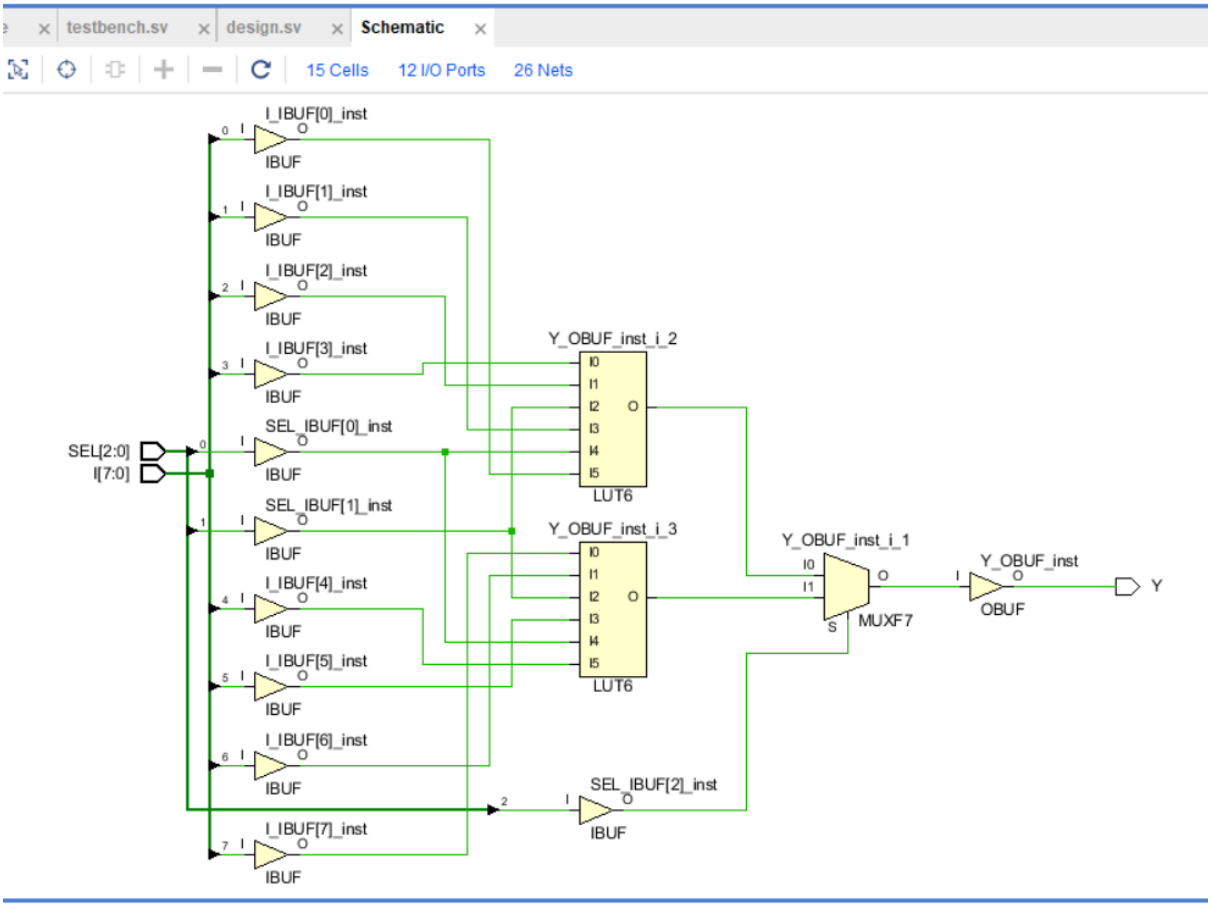
SEL	I7	I6	I5	I4	I3	I2	I1	I0	Y
000	1	1	0	0	1	1	0	0	0
001	1	1	0	0	1	1	0	0	0
010	1	1	0	0	1	1	0	0	1
011	1	1	0	0	1	1	0	0	1
100	1	1	0	0	1	1	0	0	0
101	1	1	0	0	1	1	0	0	0
110	1	1	0	0	1	1	0	0	1
111	1	1	0	0	1	1	0	0	1



8x1 MUX => Waveform Graph

8x1 MUX => RTL Schematic Graph





8x1 MUX => Synthesis Schematic

- 16x1 MULTIPLEXER :
  - Design Code :

```
module mux16to1 ( input  wire [15:0] I , input  wire [3:0]  SEL , output reg );

        always @(*) begin
        case (SEL)
                4'b0000: Y = I[0];
                4'b0001: Y = I[1];
                4'b0010: Y = I[2];
                4'b0011: Y = I[3];
                4'b0100: Y = I[4];
                4'b0101: Y = I[5];
                4'b0110: Y = I[6];
                4'b0111: Y = I[7];
                4'b1000: Y = I[8];
                4'b1001: Y = I[9];
                4'b1010: Y = I[10];
                4'b1011: Y = I[11];
                4'b1100: Y = I[12];
                4'b1101: Y = I[13];
                4'b1110: Y = I[14];
                4'b1111: Y = I[15];
                default: Y = 1'b0; // default safety
        endcase
   end
endmodule
// Behacioral Modelling ( Direct )
```

  - Boolean Table :

SEL	Y
0000	I[0]
0001	I[1]
0010	I[2]
0011	I[3]
0100	I[4]
0101	I[5]
0110	I[6]
0111	I[7]

SEL	Y
1000	I[8]
1001	I[9]
1010	I[10]
1011	I[11]
1100	I[12]
1101	I[13]
1110	I[14]
1111	I[15]

- Design Code : (Using Structural and 4x1 Mux)

```
module mux4to1(input [3:0] I, input [1:0] SEL, output Y);
        wire nS0, nS1;
        wire t0, t1, t2, t3;
        not (nS0, SEL[0]);
        not (nS1, SEL[1]);

        and (t0, nS1, nS0, I[0]);
        and (t1, nS1, SEL[0], I[1]);
        and (t2, SEL[1], nS0, I[2]);
        and (t3, SEL[1], SEL[0], I[3]);
        or  (Y, t0, t1, t2, t3);
endmodule

module mux16to1(input [15:0] I, input [3:0] SEL, output Y);
        wire y0, y1, y2, y3;  // outputs of first stage

        // First stage: four 4x1 muxes for 16 inputs
        mux4to1 m1 (.I(I[3:0]),   .SEL(SEL[1:0]), .Y(y0));
        mux4to1 m2 (.I(I[7:4]),   .SEL(SEL[1:0]), .Y(y1));
        mux4to1 m3 (.I(I[11:8]),  .SEL(SEL[1:0]), .Y(y2));
        mux4to1 m4 (.I(I[15:12]), .SEL(SEL[1:0]), .Y(y3));

        // Second stage: one 4x1 mux for the outputs of first stage
        mux4to1 m5 (.I({y3, y2, y1, y0}), .SEL(SEL[3:2]), .Y(Y));
endmodule
```

- Testbench Code :

```
module tb_mux16to1;
        reg [15:0] I ; reg [3:0] SEL ; wire Y;

        // Instantiate the DUT
    mux16to1 uut (.I(I), .SEL(SEL), .Y(Y));
        integer s; // loop variable for select lines
        integer i; // loop for test inputs

        // Representative input patterns
        reg [15:0] test_inputs [0:3];
    initial begin
        // Define 4 representative test vectors
        test_inputs[0] = 16'b0000000000000000; // Case 1: all zeros
        test_inputs[1] = 16'b1111111111111111; // Case 2: all ones
        test_inputs[2] = 16'b1010101010101010; // Case 3: alternating bits
        test_inputs[3] = 16'b1100110011001100; // Case 4: grouped bits
        $display("Output Truth Table for 16x1 MUX\n");

        // Loop through each test vector
    for (i = 0; i < 4; i = i + 1) begin
        I = test_inputs[i];
        $display("Case %0d: I = %b", i+1, I);
        $display("SEL   I15 I14 I13 I12 I11 I10 I9 I8 I7 I6 I5 I4 I3 I2 I1 I0 | Y");
        $display("-----------------------------------------------------------");

        // Sweep select lines
        for (s = 0; s < 16; s = s + 1) begin
                SEL = s; #5;
                $display("%04b  %b %b %b %b %b %b %b %b %b %b %b %b %b %b %b | %b", SEL , I[15] , I[14] , I[13] , I[12] , I[11] , I[10] , I[9] , I[8], I[7] , I[6] , I[5] , I[4] , I[3] , I[2] , I[1] , I[0] , Y);
        end
        // Separation line between block
        $display("-----------------------------------------------------------\n");
    end
        $finish;
    end
endmodule
```

- Output Truth Table :

Case 1: I = 0000000000000000

SEL	I15	I14	I13	I12	I11	I10	I9	I8	I7	I6	I5	I4	I3	I2	I1	I0	Y
0000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0010	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1010	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Case 2: I = 1111111111111111

SEL	I15	I14	I13	I12	I11	I10	I9	I8	I7	I6	I5	I4	I3	I2	I1	I0	Y
0000	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0001	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0010	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0011	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0100	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0101	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0110	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0111	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1000	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1001	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1010	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1011	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1100	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1101	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1110	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1111	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Case 3: I = 1010101010101010

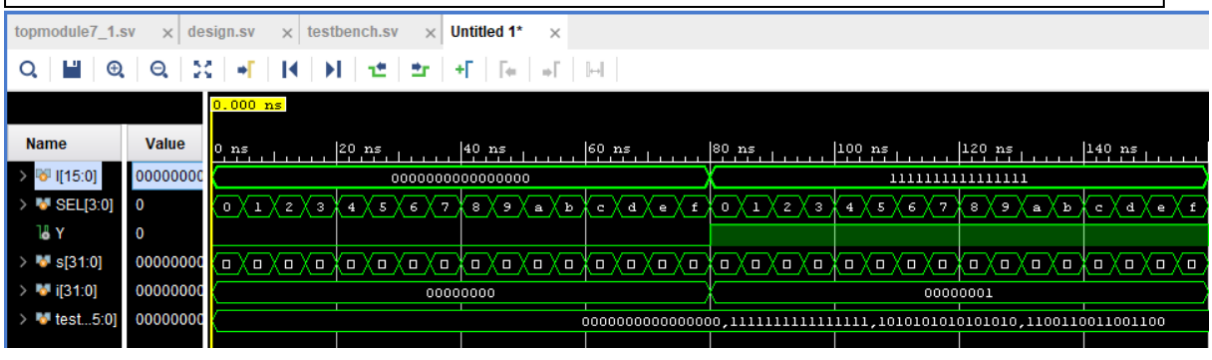
SEL I15 I14 I13 I12 I11 I10 I9 I8 I7 I6 I5 I4 I3 I2 I1 I0 | Y

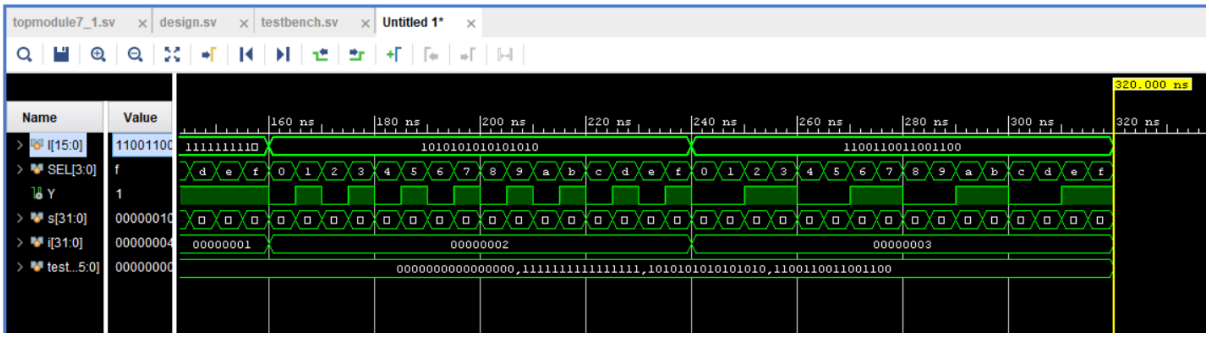
0000 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 | 0
0001 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 | 1
0010 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 | 0
0011 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 | 1
0100 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 | 0
0101 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 | 1
0110 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 | 0
0111 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 | 1
1000 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 | 0
1001 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 | 1
1010 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 | 0
1011 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 | 1
1100 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 | 0
1101 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 | 1
1110 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 | 0
1111 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 | 1

Case 4: I = 1100110011001100

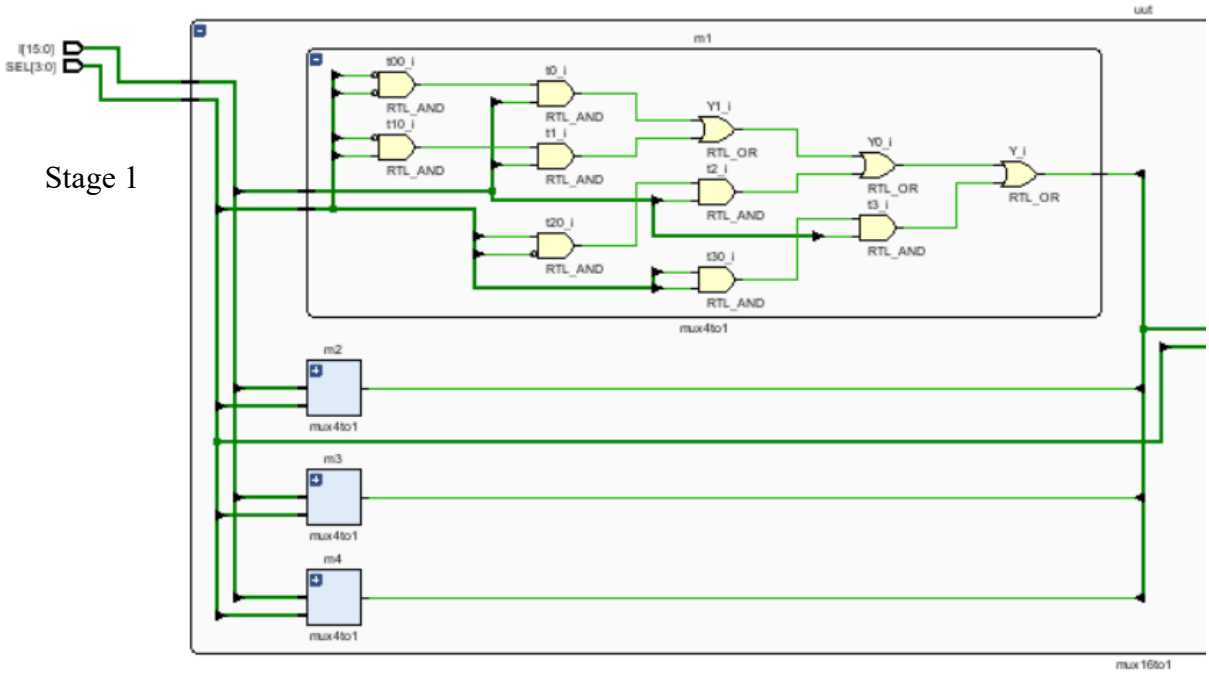
SEL I15 I14 I13 I12 I11 I10 I9 I8 I7 I6 I5 I4 I3 I2 I1 I0 | Y

0000 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 | 0
0001 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 | 0
0010 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 | 1
0011 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 | 1
0100 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 | 0
0101 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 | 0
0110 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 | 1
0111 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 | 1
1000 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 | 0
1001 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 | 0
1010 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 | 1
1011 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 | 1
1100 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 | 0
1101 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 | 0
1110 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 | 1
1111 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 | 1

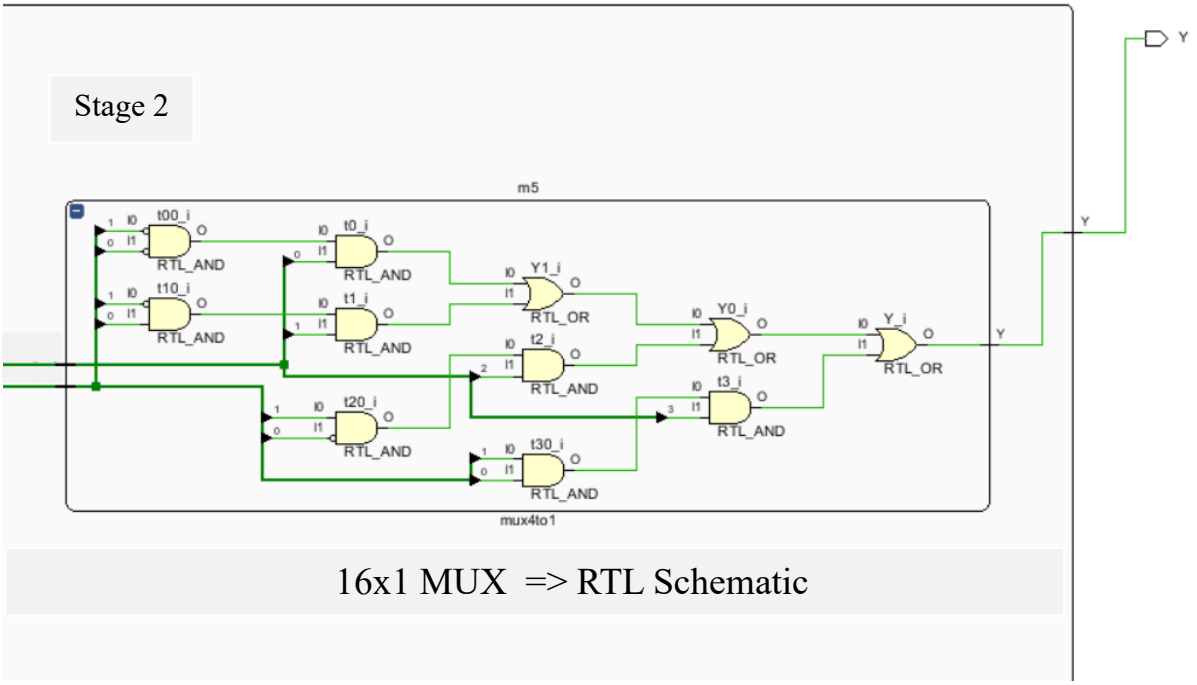




16x1 MUX => Waveform Graph

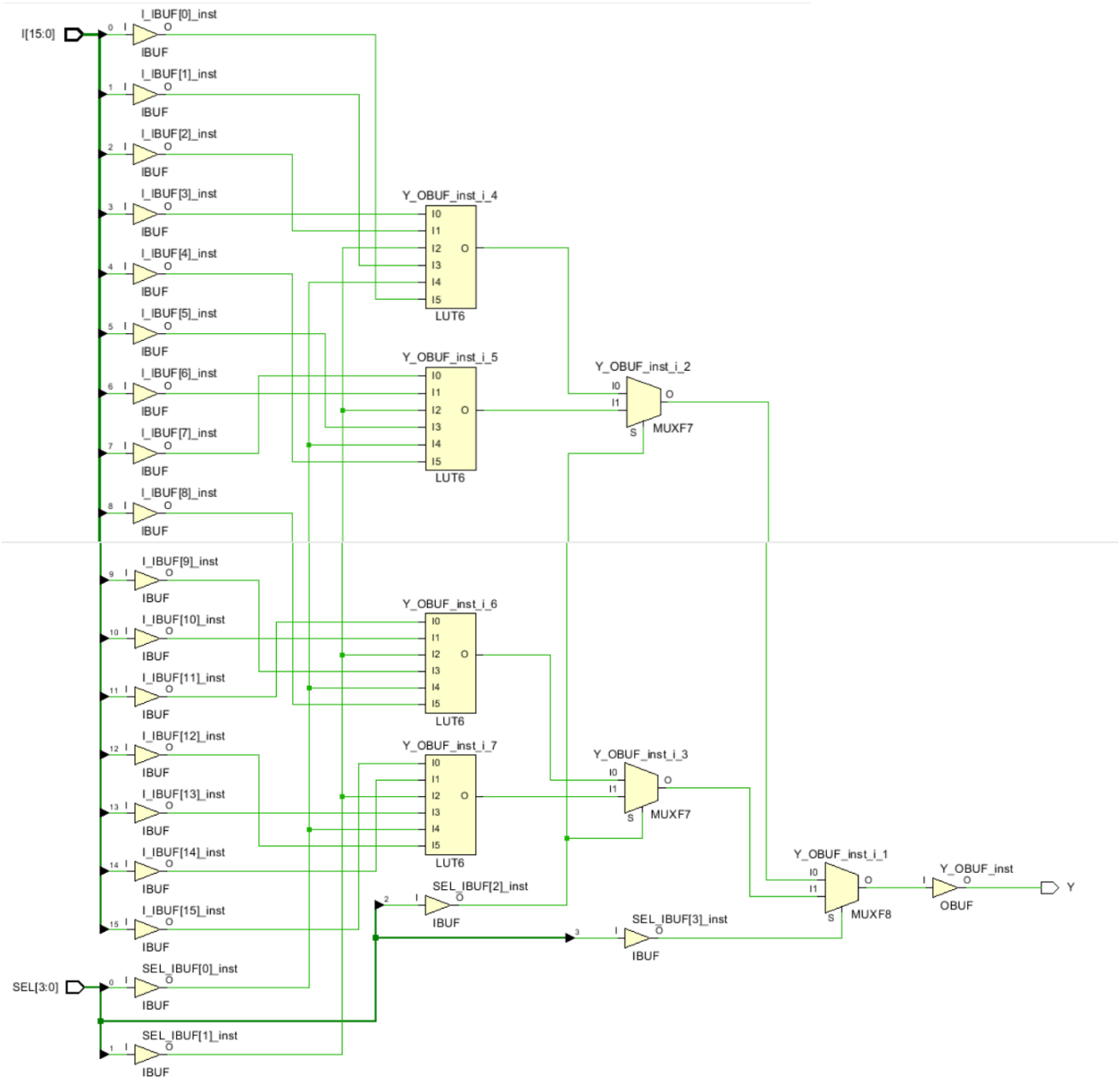


Stage 1



Stage 2

16x1 MUX => RTL Schematic



16x1 MUX => Synthesis Schematic

- 1x8 DEMULTIPLEXER :
  - Design Code :

```
module demux1to8 ( input D , input [2:0] SEL , output [7:0] Y );
        assign Y[0] = (SEL == 3'b000) ? D : 1'b0;
        assign Y[1] = (SEL == 3'b001) ? D : 1'b0;
        assign Y[2] = (SEL == 3'b010) ? D : 1'b0;
        assign Y[3] = (SEL == 3'b011) ? D : 1'b0;
        assign Y[4] = (SEL == 3'b100) ? D : 1'b0;
        assign Y[5] = (SEL == 3'b101) ? D : 1'b0;
        assign Y[6] = (SEL == 3'b110) ? D : 1'b0;
        assign Y[7] = (SEL == 3'b111) ? D : 1'b0;
endmodule
```

- Testbench Code :

```
module tb_demux1to8;
        reg D ; reg [2:0] SEL ; wire [7:0] Y;

        // Instantiate the DUT
    demux1to8 uut (.D(D), .SEL(SEL), .Y(Y));
        integer s; // loop variable for select lines
        integer i; // loop for test inputs
        // Representative input patterns
        reg test_inputs [0:1];

    initial begin
        // Define 2 representative test vectors
        test_inputs[0] = 1'b0; // Case 1: input 0
        test_inputs[1] = 1'b1; // Case 2: input 1
        $display("Output Truth Table for 1x8 DEMUX\n");

        // Loop through each test vector
    for (i = 0; i < 2; i = i + 1) begin
        D = test_inputs[i];
        $display("Case %0d: D = %b", i+1, D);
        $display("SEL   Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0");
        $display("---------------------------------");

        // Sweep select lines
        for (s = 0; s < 8; s = s + 1) begin
                SEL = s; #5;
                $display("%03b   %b  %b  %b  %b  %b  %b  %b  %b", SEL , Y[7] , Y[6]
        , Y[5] , Y[4] , Y[3] , Y[2] , Y[1] , Y[0]);
        end
        // Separation line between blocks
        $display("---------------------------------\n");
    end
    $finish;
    end
endmodule
```

- Boolean Table :

SEL	Y[7]	Y[6]	Y[5]	Y[4]	Y[3]	Y[2]	Y[1]	Y[0]
000	0	0	0	0	0	0	0	D
001	0	0	0	0	0	0	D	0
010	0	0	0	0	0	D	0	0
011	0	0	0	0	D	0	0	0
100	0	0	0	D	0	0	0	0
101	0	0	D	0	0	0	0	0
110	0	D	0	0	0	0	0	0
111	D	0	0	0	0	0	0	0

- Output Truth Table :

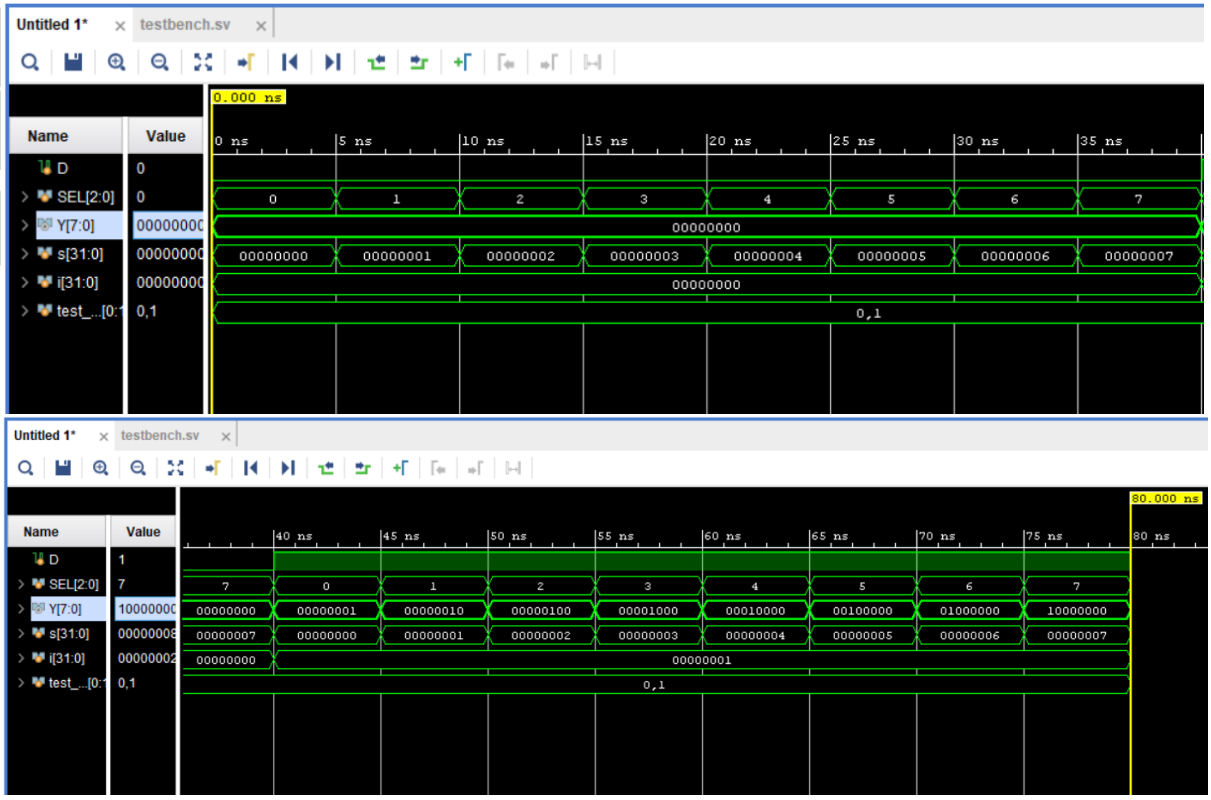
Output Truth Table for 1x8 DEMUX

Case 1: D = 0

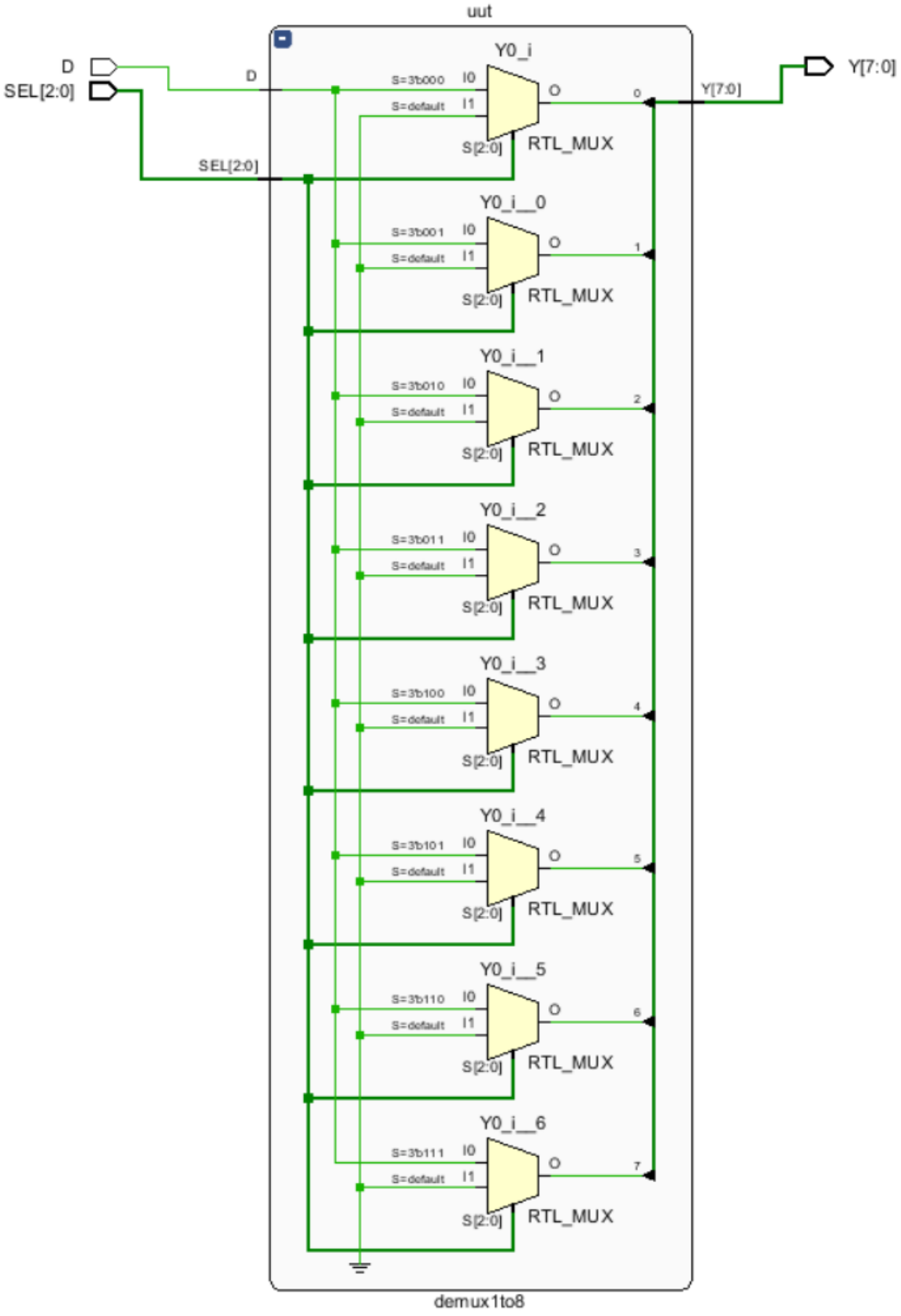
SEL	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
000	0	0	0	0	0	0	0	0
001	0	0	0	0	0	0	0	0
010	0	0	0	0	0	0	0	0
011	0	0	0	0	0	0	0	0
100	0	0	0	0	0	0	0	0
101	0	0	0	0	0	0	0	0
110	0	0	0	0	0	0	0	0
111	0	0	0	0	0	0	0	0

Case 2: D = 1

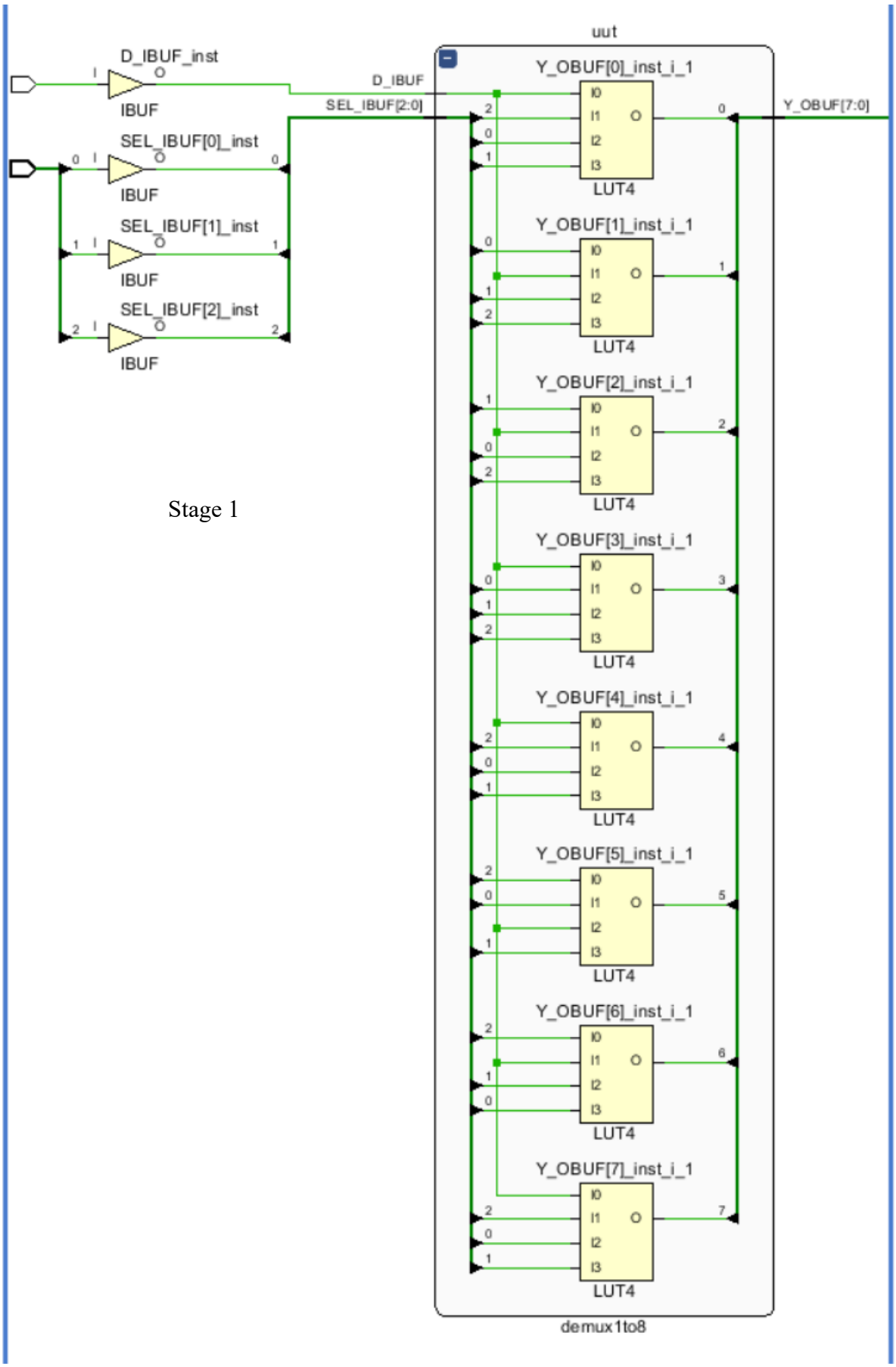
SEL	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
000	0	0	0	0	0	0	0	1
001	0	0	0	0	0	0	1	0
010	0	0	0	0	0	1	0	0
011	0	0	0	0	1	0	0	0
100	0	0	0	1	0	0	0	0
101	0	0	1	0	0	0	0	0
110	0	1	0	0	0	0	0	0
111	1	0	0	0	0	0	0	0



1x8 DEMUX => Waveform Graph

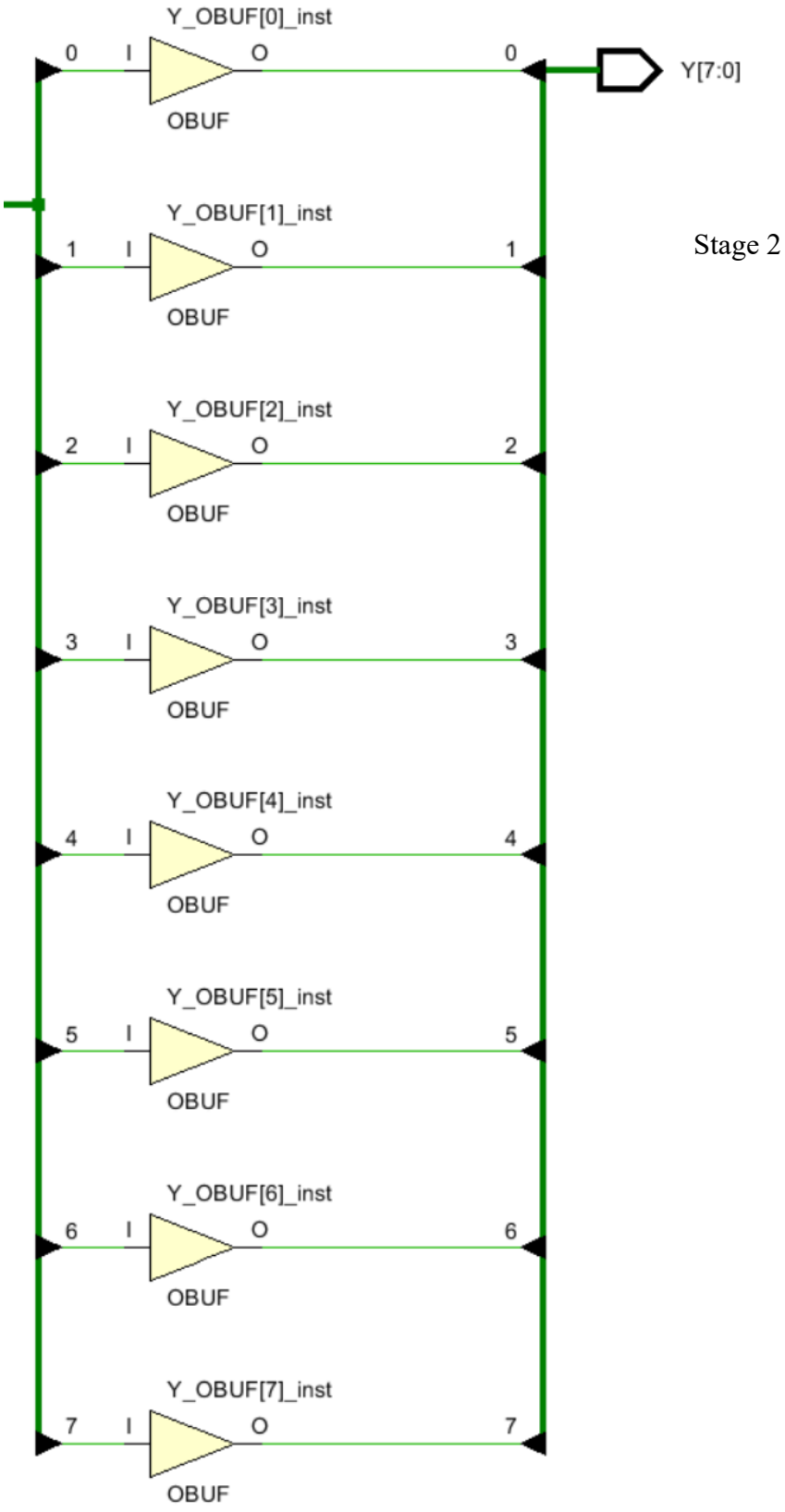


1x8 DEMUX => RTL Schematic



Stage 1

1x8 DEMUX => Synthesis Schematic



Stage 2

1x8 DEMUX => Synthesis Schematic

Conclusion: In this experiment, multiplexers and demultiplexers were designed and simulated using Verilog HDL. The MUX was observed to select one input from many based on the select lines, while the DEMUX distributed a single input to one of many outputs. The truth tables and simulation results matched the expected theoretical behavior. This verified the working principle of MUX as a **data selector** and DEMUX as a **data distributor**. The experiment also showed how larger circuits like 8×1 or 16×1 MUX can be built hierarchically. Overall, the experiment demonstrated the importance of MUX and DEMUX in digital systems and their applications in communication and data routing.

Suggested Reference:

1. *Verilog HDL: A Guide to Digital Design and Synthesis, Pearson* - Samir Palnitkar
2. *Fundamentals of Digital Logic with Verilog Design, McGraw-Hill* - Stephen Brown & Zvonko Vranesic
3. *Vivado Design Suite User Guide – HDL Coding Techniques (UG901)* - Xilinx Inc.
4. IEEE Standard for Verilog Hardware Description Language (IEEE Std 1364).
5. TutorialsPoint / NPTEL lectures on Verilog and Digital Logic Design (for supplementary learning).

References used by the students:

1. *Fundamentals of Digital Logic with Verilog Design, McGraw-Hill* - Stephen Brown & Zvonko Vranesic
2. *Vivado Design Suite User Guide – HDL Coding Techniques (UG901*) - Xilinx Inc.

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: CO 3 , CO 5

Objectives:

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

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.

- 2x4 DECODER :
  - Design Code :

```
module decoder2to4 ( input [1:0] A , input EN , output [3:0] Y );

        assign Y[0] = (EN & ~A[1] & ~A[0]); // 00 → Y0 = 1
        assign Y[1] = (EN & ~A[1] &  A[0]); // 01 → Y1 = 1
        assign Y[2] = (EN & A[1] & ~A[0]); // 10 → Y2 = 1
        assign Y[3] = (EN & A[1] &  A[0]); // 11 → Y3 = 1
endmodule
```

  - Output Truth Table :

Truth Table for 2x4 Decoder with Enable

EN	A1	A0	Y3	Y2	Y1	Y0
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	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

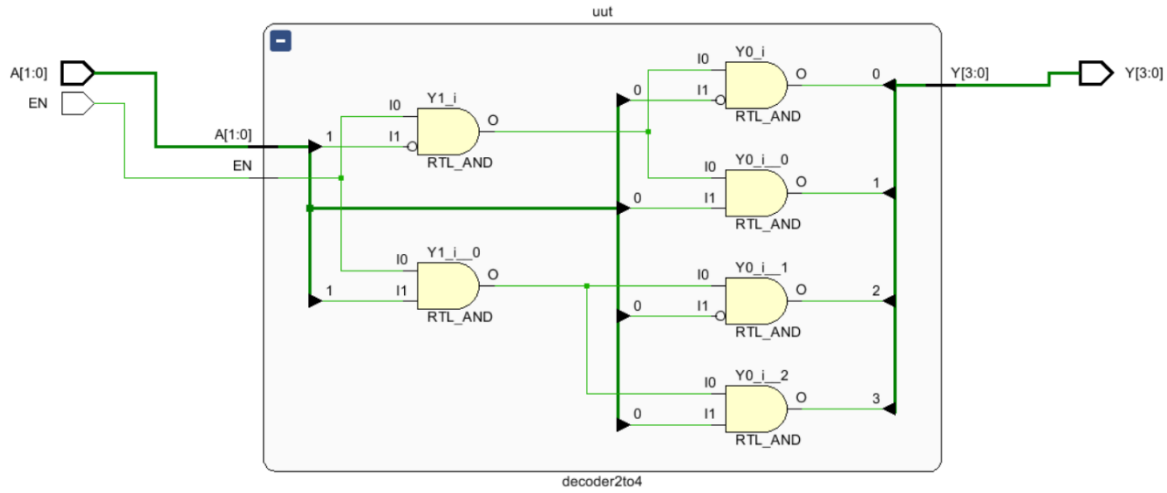
  - Boolean Table :

EN	A1 A0	Y3 Y2 Y1 Y0
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

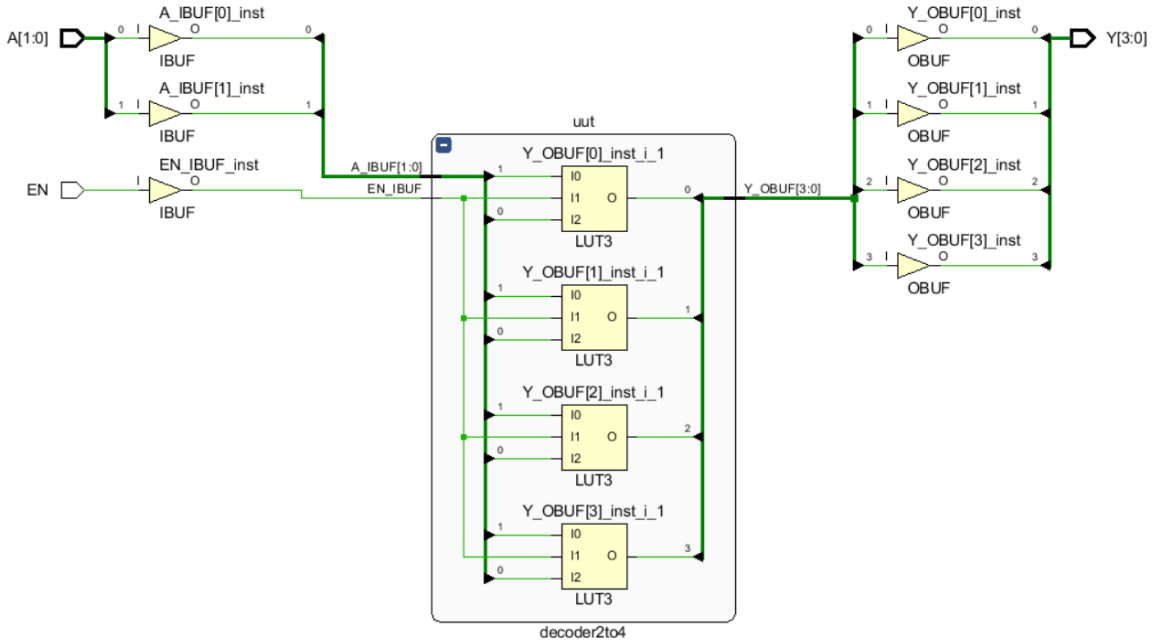
- Testbench Code :

```
module tb_decoder2to4;
        reg [1:0] A ; reg EN ; wire [3:0] Y;

        // Instantiate the DUT
    decoder2to4 uut (.A(A), .EN(EN), .Y(Y));
        integer i;

    initial begin
        $display("Truth Table for 2x4 Decoder with Enable\n");
        $display(" EN  A1 A0 | Y3 Y2 Y1 Y0");
        $display("---------------------------");

        // Case 1: EN=0 → All outputs should be 0
        EN = 0;
    for (i = 0; i < 4; i = i + 1) begin
        A = i; #5; $display("  %b   %b  %b |  %b  %b  %b  %b", EN , A[1] ,  A[0] , Y[3],
    Y[2] , Y[1] , Y[0]);
    end
        $display("---------------------------");

        // Case 2: EN=1 → Normal decoding
        EN = 1;
    for (i = 0; i < 4; i = i + 1) begin
        A = i; #5; $display("  %b   %b  %b |  %b  %b  %b  %b", EN , A[1] , A[0] , Y[3] ,
    Y[2] , Y[1] , Y[0]);
        end
        $display("---------------------------");
    $finish;
    end
endmodule
```

2x4 DECODER => Waveform Graph



2x4 DECODER => RTL Schematic



2x4 DECODER => Synthesis Schematic

- 3x8 DECODER :
  - Design Code :

```
module decoder3to8 ( input [2:0] A , input EN , output [7:0] Y );
        assign Y[0] = EN & ~A[2] & ~A[1] & ~A[0];
        assign Y[1] = EN & ~A[2] & ~A[1] &  A[0];
        assign Y[2] = EN & ~A[2] &  A[1] & ~A[0];
        assign Y[3] = EN & ~A[2] &  A[1] &  A[0];
        assign Y[4] = EN &  A[2] & ~A[1] & ~A[0];
        assign Y[5] = EN &  A[2] & ~A[1] &  A[0];
        assign Y[6] = EN &  A[2] &  A[1] & ~A[0];
        assign Y[7] = EN &  A[2] &  A[1] &  A[0];
endmodule
```

- Testbench Code :

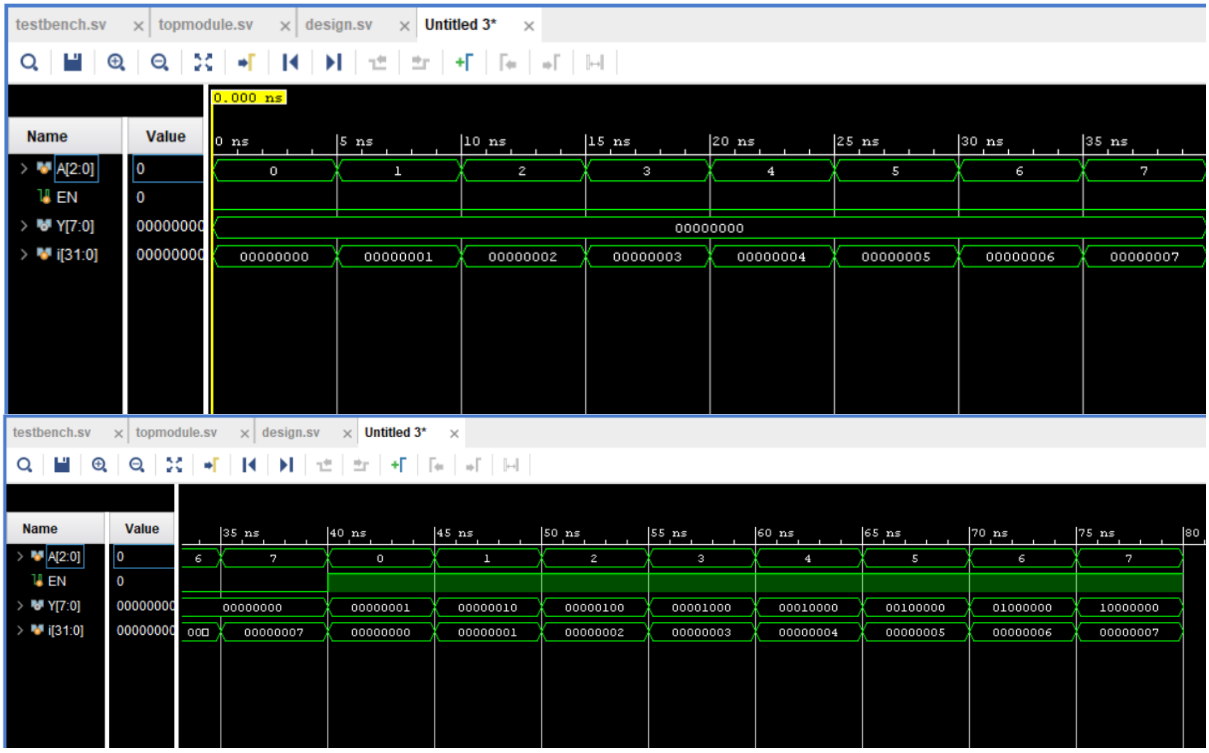
```
module tb_decoder3to8;
        reg [2:0] A ; reg EN ; wire [7:0] Y;

        // Instantiate the DUT
    decoder3to8 uut (.A(A), .EN(EN), .Y(Y));
        integer i;

    initial begin
        $display("Truth Table for 3x8 Decoder with Enable\n");
        $display(" EN  A2 A1 A0 | Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0");
        $display("-----------------------------------------");

        // Case 1: EN=0 → All outputs should be 0
        EN = 0;
    for (i = 0; i < 8; i = i + 1) begin
        A = i; #5; $display(" %b   %b %b %b | %b %b %b %b %b %b %b %b", EN
    , A[2] , A[1] , A[0] , Y[7] , Y[6] , Y[5] , Y[4] , Y[3] , Y[2] , Y[1] , Y[0]);
    end
        $display("-----------------------------------------");

        // Case 2: EN=1 → Normal decoding
        EN = 1;
    for (i = 0; i < 8; i = i + 1) begin
        A = i; #5; $display(" %b   %b %b %b | %b %b %b %b %b %b %b %b", EN
    , A[2] , A[1] , A[0] , Y[7] , Y[6] , Y[5] , Y[4] , Y[3] , Y[2] , Y[1] , Y[0]);
    end
        $display("-----------------------------------------");
    $finish;
    end
endmodule
```

- Output Truth Table :

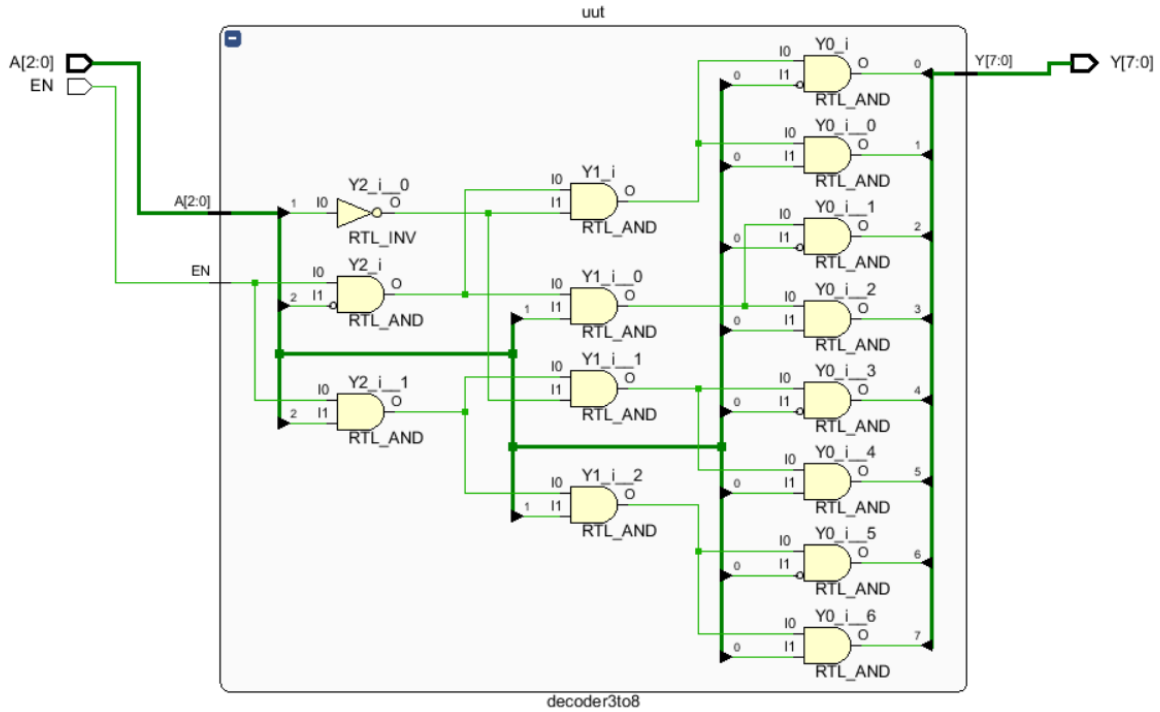
```
Truth Table for 3x8 Decoder with Enable

EN  A2 A1 A0 | Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0
-----------------------------------------
 0   0  0  0 | 0  0  0  0  0  0  0  0
 0   0  0  1 | 0  0  0  0  0  0  0  0
 0   0  1  0 | 0  0  0  0  0  0  0  0
 0   0  1  1 | 0  0  0  0  0  0  0  0
 0   1  0  0 | 0  0  0  0  0  0  0  0
 0   1  0  1 | 0  0  0  0  0  0  0  0
 0   1  1  0 | 0  0  0  0  0  0  0  0
 0   1  1  1 | 0  0  0  0  0  0  0  0
-----------------------------------------
 1   0  0  0 | 0  0  0  0  0  0  0  1
 1   0  0  1 | 0  0  0  0  0  0  1  0
 1   0  1  0 | 0  0  0  0  0  1  0  0
 1   0  1  1 | 0  0  0  0  1  0  0  0
 1   1  0  0 | 0  0  0  1  0  0  0  0
 1   1  0  1 | 0  0  1  0  0  0  0  0
 1   1  1  0 | 0  1  0  0  0  0  0  0
 1   1  1  1 | 1  0  0  0  0  0  0  0
-----------------------------------------
```

- Boolean Table :

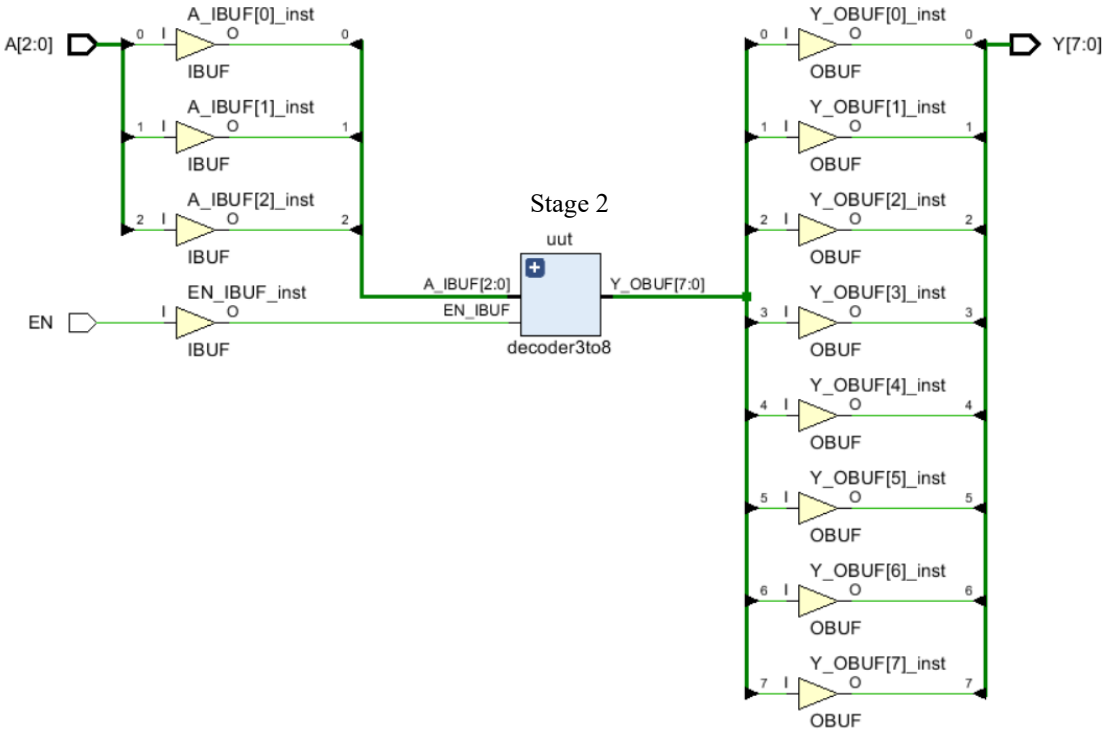
EN	A2	A1	A0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0	x	x	x	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0



3x8 DECODER => Waveform Graph

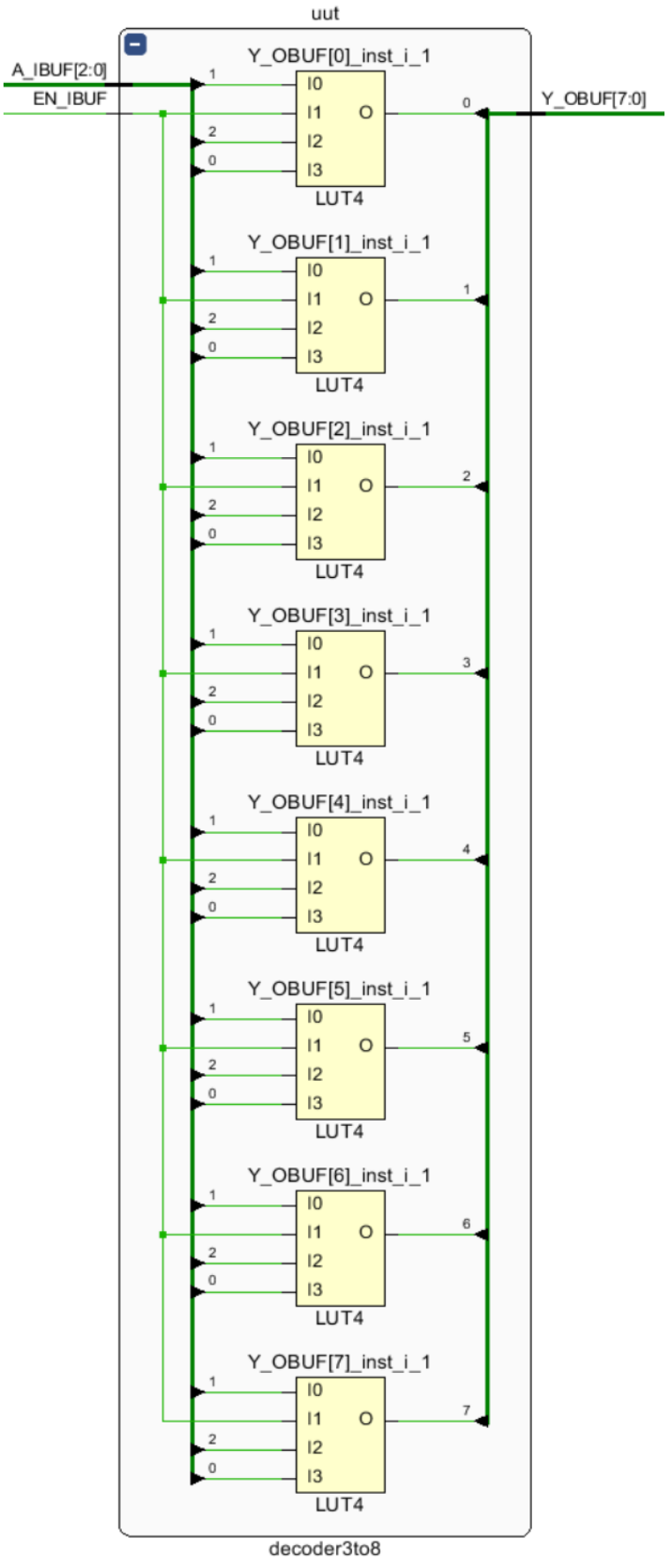


3x8 DECODER => RTL Schematic



3x8 DECODER => Sythesis Schematic

Stage 2


uut
A_IBUF[2:0]
EN_IBUF
Y_OBUF[7:0]
Y_OBUF[0]_inst_i_1
Y_OBUF[1]_inst_i_1
Y_OBUF[2]_inst_i_1
Y_OBUF[3]_inst_i_1
Y_OBUF[4]_inst_i_1
Y_OBUF[5]_inst_i_1
Y_OBUF[6]_inst_i_1
Y_OBUF[7]_inst_i_1
I0
I1
I2
I3
O
LUT4
decoder3to8

- 4x16 DECODER :
  - Design Code :

```
module decoder4to16 ( input [3:0] A , input EN , output [15:0] Y );

    assign Y[0]  = EN & ~A[3] & ~A[2] & ~A[1] & ~A[0];
    assign Y[1]  = EN & ~A[3] & ~A[2] & ~A[1] &  A[0];
    assign Y[2]  = EN & ~A[3] & ~A[2] &  A[1] & ~A[0];
    assign Y[3]  = EN & ~A[3] & ~A[2] &  A[1] &  A[0];
    assign Y[4]  = EN & ~A[3] &  A[2] & ~A[1] & ~A[0];
    assign Y[5]  = EN & ~A[3] &  A[2] & ~A[1] &  A[0];
    assign Y[6]  = EN & ~A[3] &  A[2] &  A[1] & ~A[0];
    assign Y[7]  = EN & ~A[3] &  A[2] &  A[1] &  A[0];
    assign Y[8]  = EN &  A[3] & ~A[2] & ~A[1] & ~A[0];
    assign Y[9]  = EN &  A[3] & ~A[2] & ~A[1] &  A[0];
    assign Y[10] = EN &  A[3] & ~A[2] &  A[1] & ~A[0];
    assign Y[11] = EN &  A[3] & ~A[2] &  A[1] &  A[0];
    assign Y[12] = EN &  A[3] &  A[2] & ~A[1] & ~A[0];
    assign Y[13] = EN &  A[3] &  A[2] & ~A[1] &  A[0];
    assign Y[14] = EN &  A[3] &  A[2] &  A[1] & ~A[0];
    assign Y[15] = EN &  A[3] &  A[2] &  A[1] &  A[0];

endmodule
```

  - Boolean Table :

EN	A3	A2	A1	A0	Y15	Y14	Y13	Y12	Y11	Y10	Y9	Y8	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0	x	x	x	x	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1	1	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- **Testbench Code :**

```
module tb_decoder4to16;
        reg [3:0] A; reg EN ; wire [15:0] Y ;

        // Instantiate the DUT
    decoder4to16 uut (.A(A), .EN(EN), .Y(Y));
        integer i;

    initial begin
        $display("Truth Table for 4x16 Decoder with Enable\n");
        $display(" EN  A3 A2 A1 A0 | Y15 Y14 Y13 Y12 Y11 Y10 Y9 Y8 Y7 Y6 Y5 Y4
Y3 Y2 Y1 Y0");
        $display("-------------------------------------------------------------------------------");

        // Case 1: EN=0 → All outputs should be 0
        EN = 0;
    for (i = 0; i < 16; i = i + 1) begin
        A = i; #5; $display(" %b  %b %b %b %b | %b %b %b %b %b %b %b %b
%b %b %b %b %b %b %b %b", EN , A[3] , A[2] , A[1] , A[0] , Y[15] , Y[14] , Y[13]
, Y[12] , Y[11] , Y[10] , Y[9] , Y[8] , Y[7] , Y[6] , Y[5] , Y[4] , Y[3] , Y[2] , Y[1] , Y[0] );
    end
        $display("-------------------------------------------------------------------------------");

        // Case 2: EN=1 → Normal decoding
        EN = 1;
    for (i = 0; i < 16; i = i + 1) begin
      A = i; #5; $display(" %b  %b %b %b %b | %b %b %b %b %b %b %b %b
%b %b %b %b %b %b %b %b", EN , A[3] , A[2] , A[1] , A[0], Y[15] , Y[14] , Y[13]
, Y[12] , Y[11] , Y[10] , Y[9] , Y[8] , Y[7] , Y[6] , Y[5] , Y[4] , Y[3] , Y[2], Y[1] , Y[0] );
       end
        $display("-------------------------------------------------------------------------------");
    $finish;
    end
endmodule
```

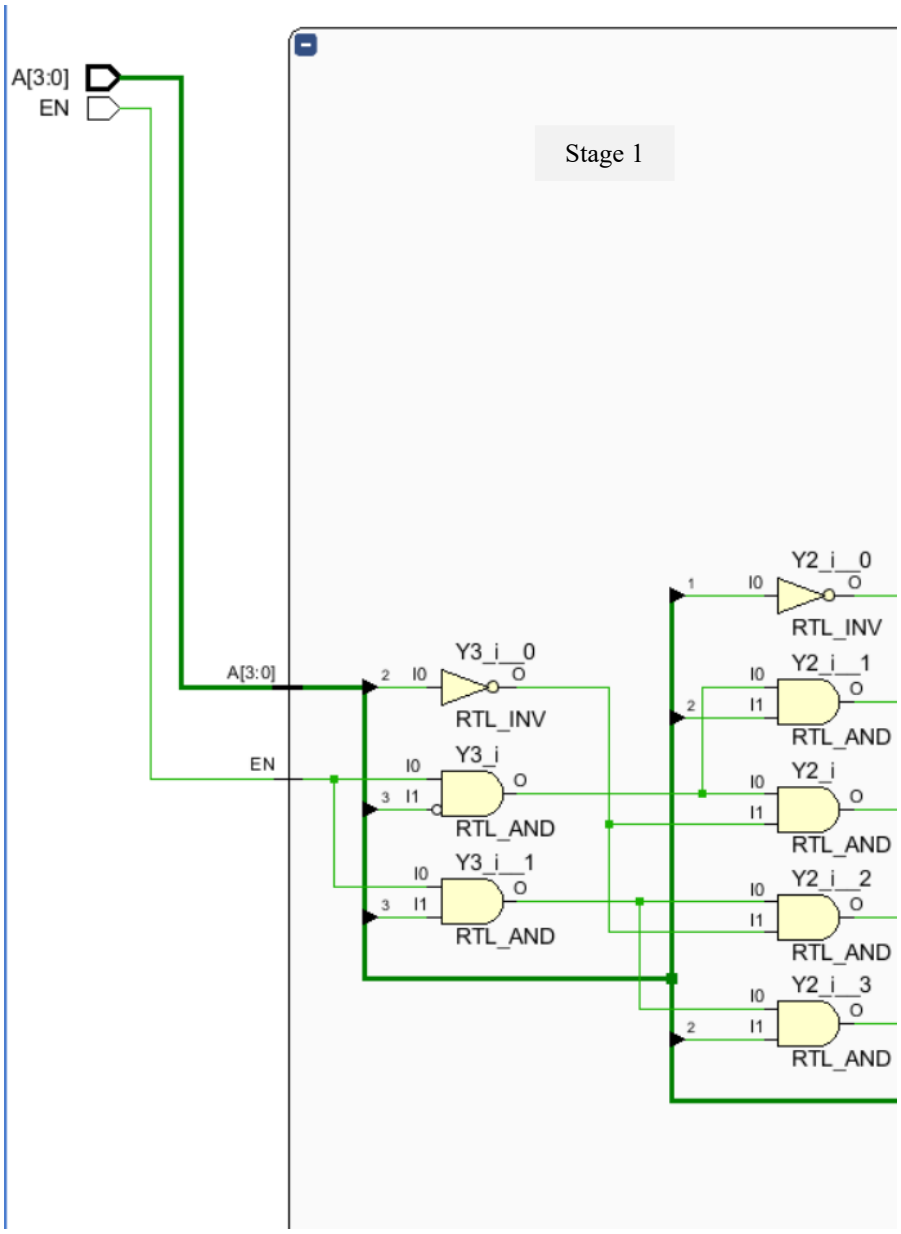
4x16 DECODER (EN = 0) => Waveform Graph

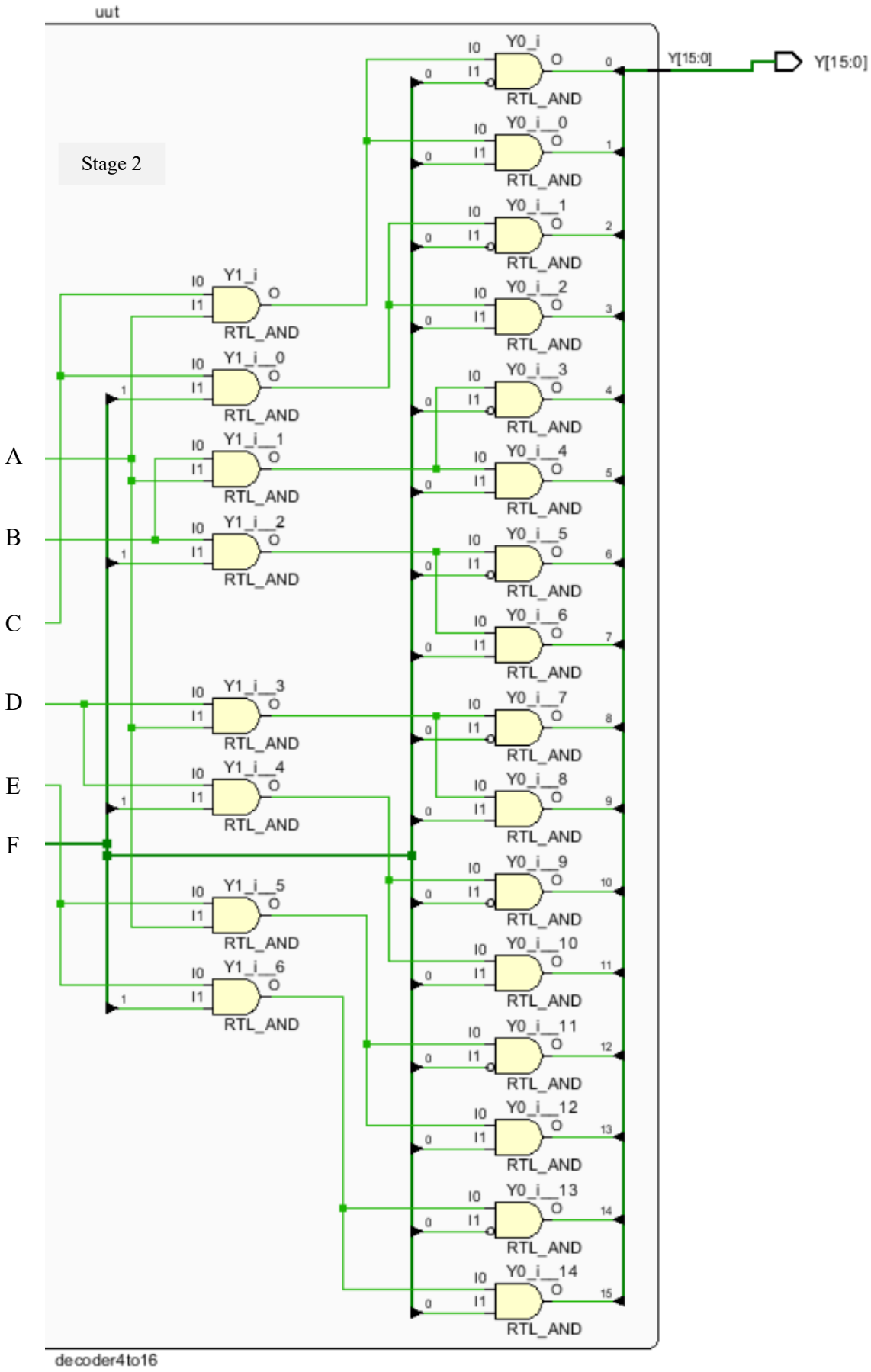
- Output Truth Table :

Truth Table for 4x16 Decoder with Enable

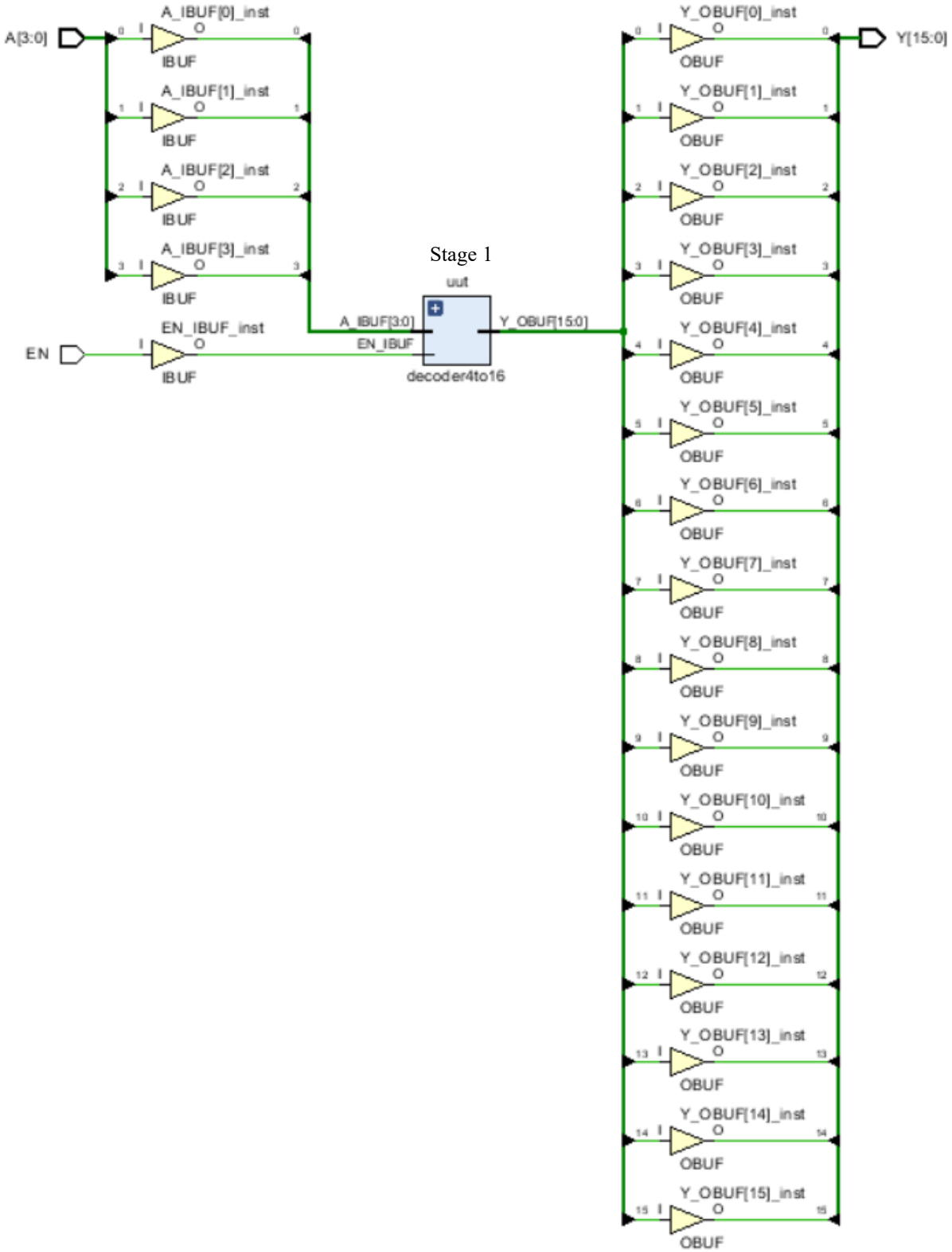
EN	A3	A2	A1	A0	Y15	Y14	Y13	Y12	Y11	Y10	Y9	Y8	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1	1	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

4x16 DECODER (EN = 1) => Waveform Graph

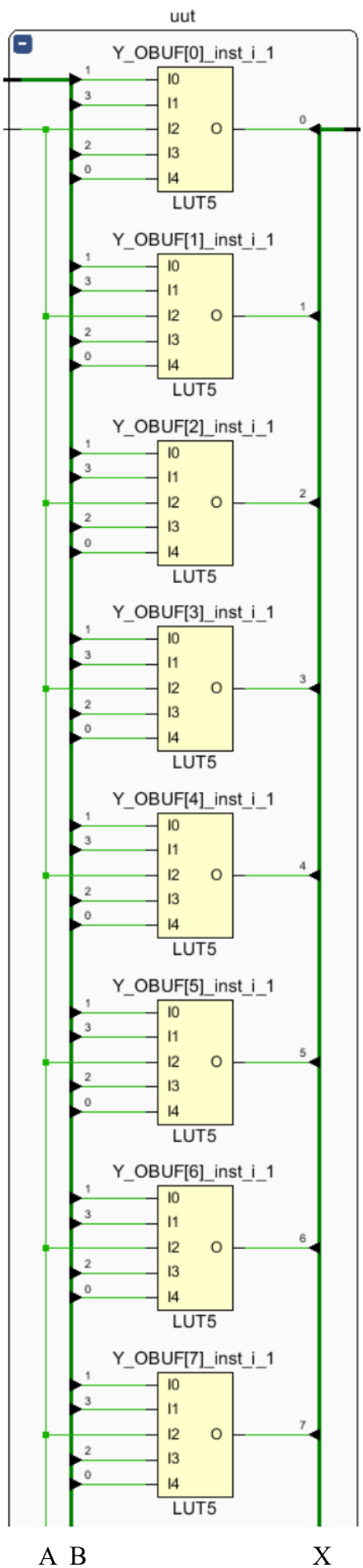

A[3:0]
EN
Stage 1
A[3:0]
EN
Y3_i__0
RTL_INV
Y3_i
RTL_AND
Y3_i__1
RTL_AND
Y2_i__0
RTL_INV
Y2_i__1
RTL_AND
Y2_i
RTL_AND
Y2_i__2
RTL_AND
Y2_i__3
RTL_AND
A
B
C
D
E
F



4x16 DECODER => RTL Schematic



4x16 DECODER => Synthesis Schematic



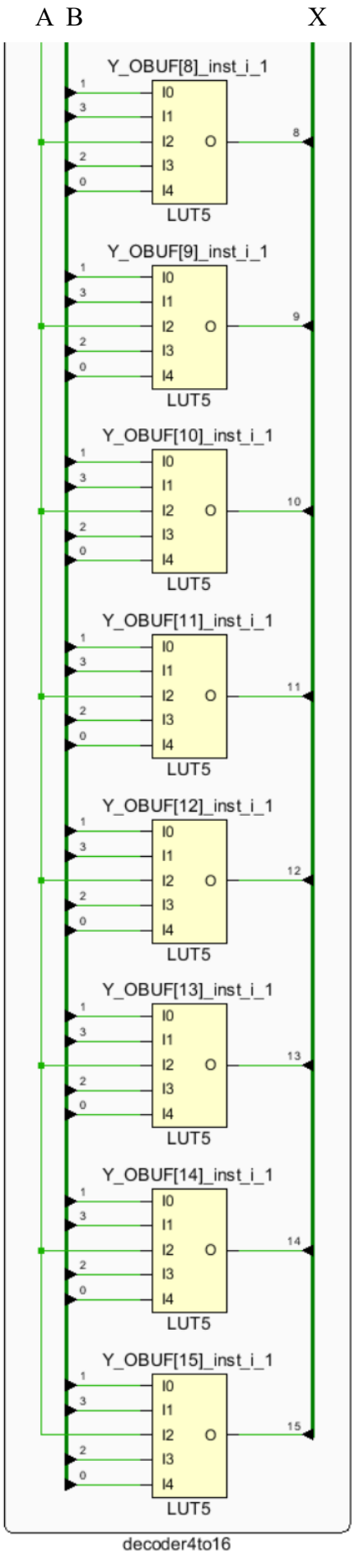
Stage 1

←

Top part of the Decoder

Bottom Part of Decoder

→



Conclusion: In this experiment, we studied the operation of **binary decoders** and their ability to convert binary inputs into **one-hot outputs**. The 2×4, 3×8, and 4×16 decoders were implemented using **data-flow modeling with enable control**. The enable pin allows the outputs to be **disabled when not required**, demonstrating control over the circuit. The truth tables verified that for **each input combination, only the corresponding output is high**, while all others remain low. The experiment highlighted the use of decoders in **digital systems for selection and addressing applications**. Testbenches were used to validate functionality across all input combinations. This reinforced understanding of **combinational logic design and verification techniques**.

Suggested Reference:

1. *Verilog HDL: A Guide to Digital Design and Synthesis, Pearson* - Samir Palnitkar
2. *Fundamentals of Digital Logic with Verilog Design, McGraw-Hill* - Stephen Brown & Zvonko Vranesic
3. *Vivado Design Suite User Guide – HDL Coding Techniques (UG901)* - Xilinx Inc.
4. IEEE Standard for Verilog Hardware Description Language (IEEE Std 1364).
5. TutorialsPoint / NPTEL lectures on Verilog and Digital Logic Design (for supplementary learning).

References used by the students:

1. *Fundamentals of Digital Logic with Verilog Design, McGraw-Hill* - Stephen Brown & Zvonko Vranesic
2. *Vivado Design Suite User Guide – HDL Coding Techniques (UG901*) - Xilinx Inc.

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:

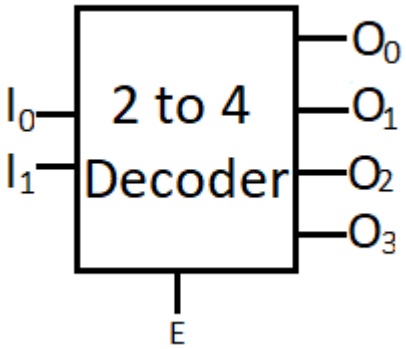
Relevant CO: CO 3 , CO 5

Objectives:

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

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.

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.



Truth Table for 2x4 Decoder

EN	Inputs		Outputs			
----	I1	I0	O3	O2	O1	O0
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

- 4x2 PRIORITY ENCODER :
  - Design Code :

```
module encoder4to2 ( input [3:0] A , input EN , output [1:0] Y ) ;
        assign Y[1] = EN & (A[3] | A[2]);
        assign Y[0] = EN & (A[3] | (~A[3] & A[1]));
endmodule
```

- Testbench Code :

```
module tb_encoder4to2;
        reg [3:0] A ; reg EN ; wire [1:0] Y ;

        // Instantiate the DUT
    encoder4to2 uut (.A(A), .EN(EN), .Y(Y));
        integer i;

    initial begin
        $display("Truth Table for 4x2 Encoder with Enable\n");
        $display(" EN  A3 A2 A1 A0 | Y1 Y0");
        $display("---------------------------");

        // Case 1: EN=0 → All outputs should be 0
        EN = 0;
    for (i = 0; i < 16; i = i + 1) begin
        A = i; #5; $display("  %b   %b  %b  %b  %b |  %b  %b", EN , A[3] , A[2] , A[1] ,
A[0] , Y[1] , Y[0]);
    end
        $display("---------------------------");

        // Case 2: EN=1 → Normal encoding
        EN = 1;
    for (i = 0; i < 16; i = i + 1) begin
        A = i; #5; $display("  %b   %b  %b  %b  %b |  %b  %b", EN , A[3] , A[2] , A[1] ,
A[0] , Y[1] , Y[0]);
    end
        $display("---------------------------");
    $finish;
    end
endmodule
```

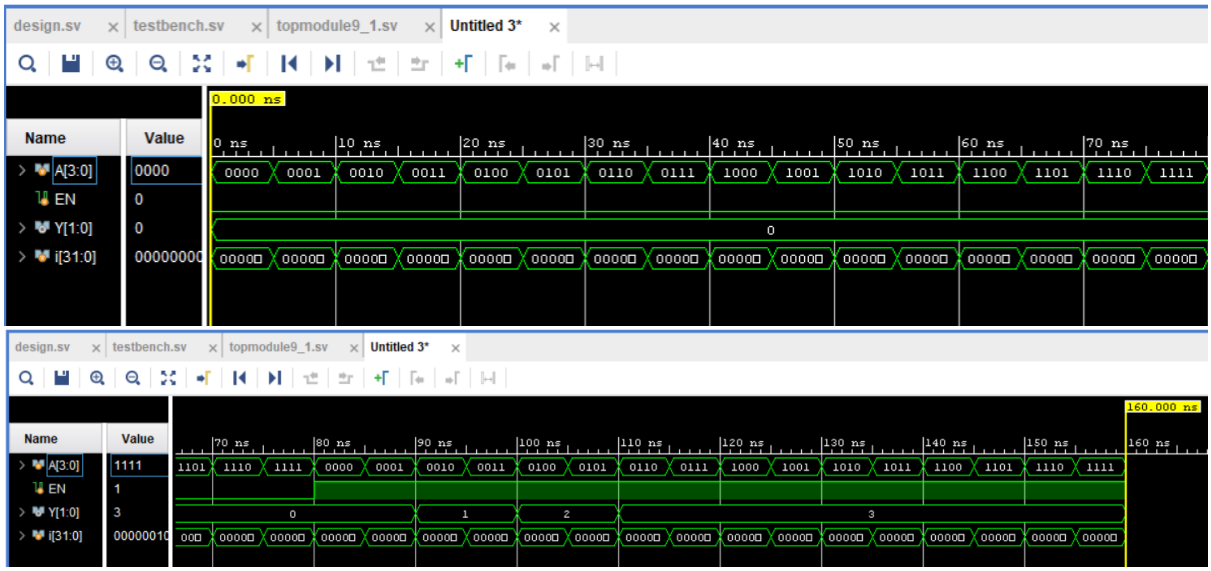
- Boolean Table :

EN	A3	A2	A1	A0	Y1	Y0
0	x	x	x	x	0	0
1	0	0	0	x	0	0
1	0	0	1	x	0	1
1	0	1	x	x	1	0
1	1	x	x	x	1	1

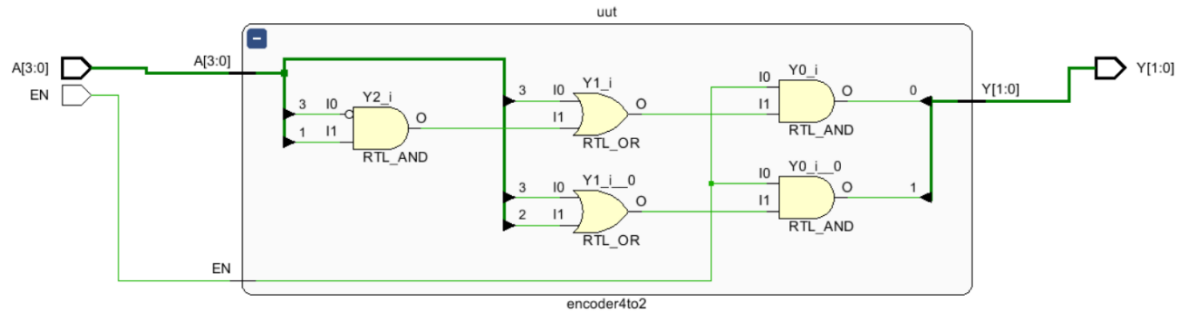
- Output Truth Table :

Truth Table for 4x2 Encoder with Enable

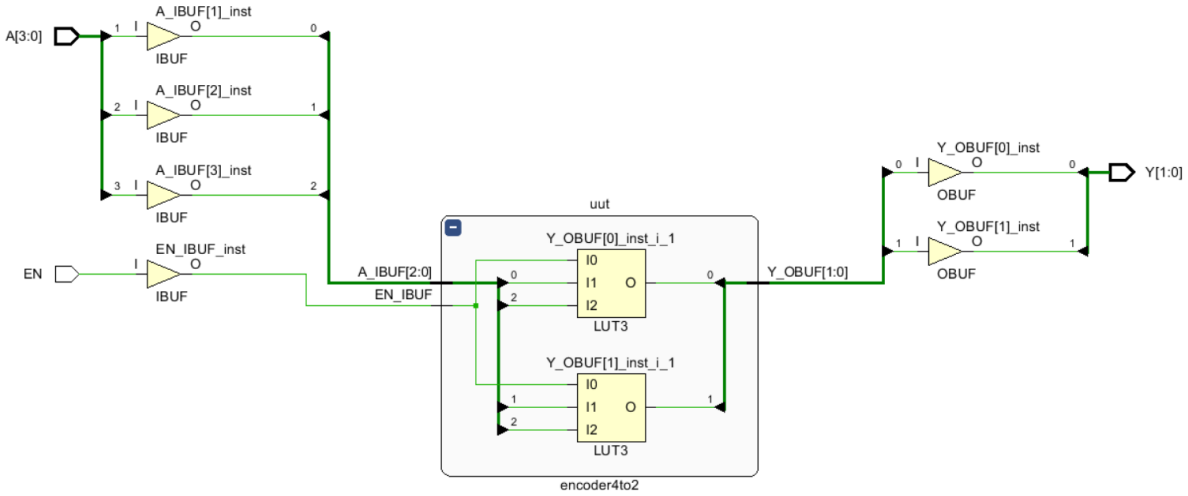
EN	A3	A2	A1	A0	Y1	Y0
0	0	0	0	0	0	0
0	0	0	0	1	0	0
0	0	0	1	0	0	0
0	0	0	1	1	0	0
0	0	1	0	0	0	0
0	0	1	0	1	0	0
0	0	1	1	0	0	0
0	0	1	1	1	0	0
0	1	0	0	0	0	0
0	1	0	0	1	0	0
0	1	0	1	0	0	0
0	1	0	1	1	0	0
0	1	1	0	0	0	0
0	1	1	0	1	0	0
0	1	1	1	0	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	0
1	0	0	0	1	0	0
1	0	0	1	0	0	1
1	0	0	1	1	0	1
1	0	1	0	0	1	0
1	0	1	0	1	1	0
1	0	1	1	0	1	1
1	0	1	1	1	1	1
1	1	0	0	0	1	1
1	1	0	0	1	1	1
1	1	0	1	0	1	1
1	1	0	1	1	1	1
1	1	1	0	0	1	1
1	1	1	0	1	1	1
1	1	1	1	0	1	1
1	1	1	1	1	1	1



4x2 PRIORITY CODER => Waveform Graph



4x2 PRIORITY CODER => RTL Schematic



4x2 PRIORITY CODER => Synthesis Schematic

- 8x3 PRIORITY ENCODER :
  - Design Code :

```
module encoder8to3 ( input [7:0] A , input EN , output [2:0] Y );

        assign Y[2] = EN & ( A[7] | A[6] | A[5] | A[4] );
        assign Y[1] = EN & ( A[7] | A[6] | (~A[5] & ~A[4] & (A[3] | A[2])) );
        assign Y[0] = EN & ( A[7] | (~A[6] & A[5]) | (~A[6] & ~A[4] & A[3]) | (~A[6] &
~A[4] & ~A[2] & A[1]) );
endmodule
```

- **Testbench Code :**

```
module tb_encoder8to3;
        reg [7:0] A ; reg EN; wire [2:0] Y;

        // Instantiate the DUT
        encoder8to3 uut (.A(A), .EN(EN), .Y(Y));
        integer i ; reg [7:0] test_vectors [0:5];

    initial begin
        $display("Truth Table for 8x3 Priority Encoder with Enable\n");
        $display(" EN  A7 A6 A5 A4 A3 A2 A1 A0 | Y2 Y1 Y0");
        $display("-----------------------------------------");

        // Case 1: EN=0 → All outputs should be 0
        EN = 0;
    for (i = 0; i < 8; i = i + 1) begin
        A = 8'b00000001 << i; #5; $display("  %b   %b %b %b %b %b %b %b %b |  %b
%b %b", EN , A[7] , A[6] , A[5] , A[4] , A[3] , A[2] , A[1] , A[0] , Y[2] , Y[1] , Y[0] );
    end
        $display("-----------------------------------------");

        // Case 2: EN=1 → Normal one-hot encoding
        EN = 1;
    for (i = 0; i < 8; i = i + 1) begin
        A = 8'b00000001 << i; #5; $display("  %b   %b %b %b %b %b %b %b %b |  %b
%b %b", EN ,  A[7] , A[6] , A[5] , A[4] , A[3] , A[2] , A[1] , A[0] , Y[2] , Y[1] , Y[0] );
    end
        $display("-----------------------------------------");

        // Case 3: EN=1 → Multiple-inputs to test priority
        test_vectors[0] = 8'b11000000; // A7 priority
        test_vectors[1] = 8'b01100000; // A6 priority
        test_vectors[2] = 8'b00011000; // A4 priority
        test_vectors[3] = 8'b00001100; // A3 priority
        test_vectors[4] = 8'b00000011; // A1 priority
        test_vectors[5] = 8'b10101010; // A7 priority

    for (i = 0; i < 6; i = i + 1) begin
        A = test_vectors[i]; #5; $display("  %b   %b %b %b %b %b %b %b %b |  %b  %b
%b", EN , A[7] , A[6] , A[5] , A[4] , A[3] , A[2] , A[1] , A[0] , Y[2] , Y[1] , Y[0] );
    end
        $display("-----------------------------------------");
    $finish;
    end
endmodule
```

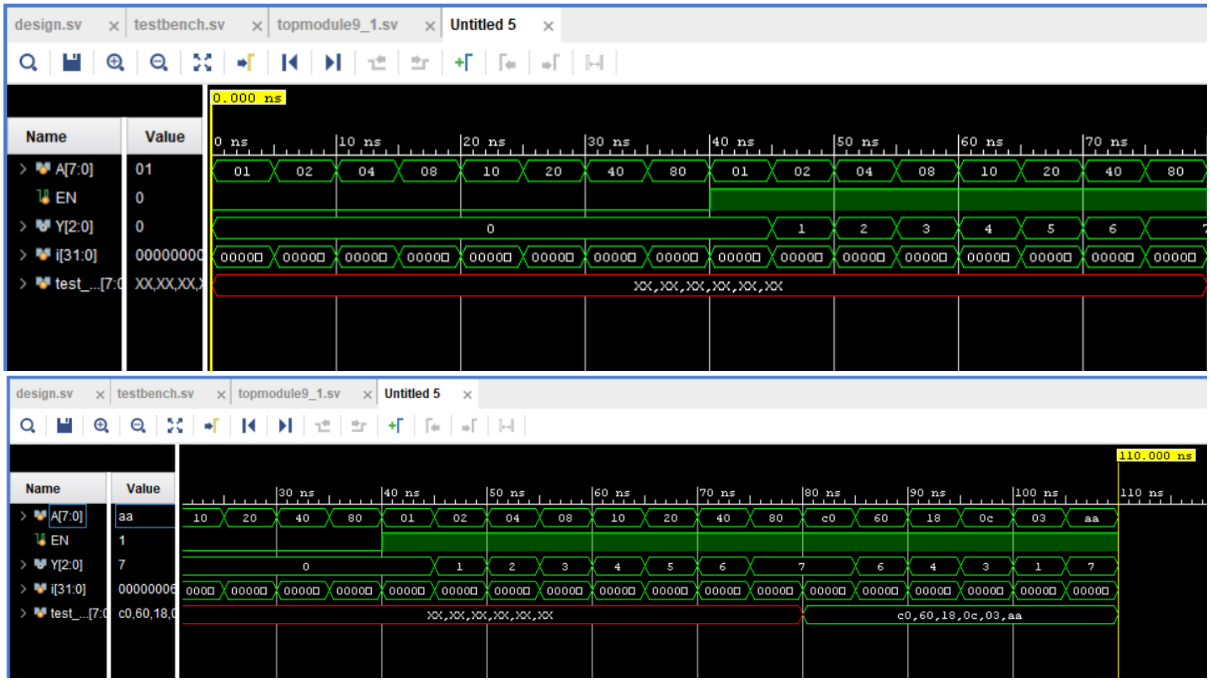
- Boolean Table :

EN	A7	A6	A5	A4	A3	A2	A1	A0	Y2	Y1	Y0
0	x	x	x	x	x	x	x	x	0	0	0
1	0	0	0	0	0	0	0	x	0	0	0
1	0	0	0	0	0	0	1	x	0	0	1
1	0	0	0	0	0	1	x	x	0	1	0
1	0	0	0	0	1	x	x	x	0	1	1
1	0	0	0	1	x	x	x	x	1	0	0
1	0	0	1	x	x	x	x	x	1	0	1
1	0	1	x	x	x	x	x	x	1	1	0
1	1	x	x	x	x	x	x	x	1	1	1

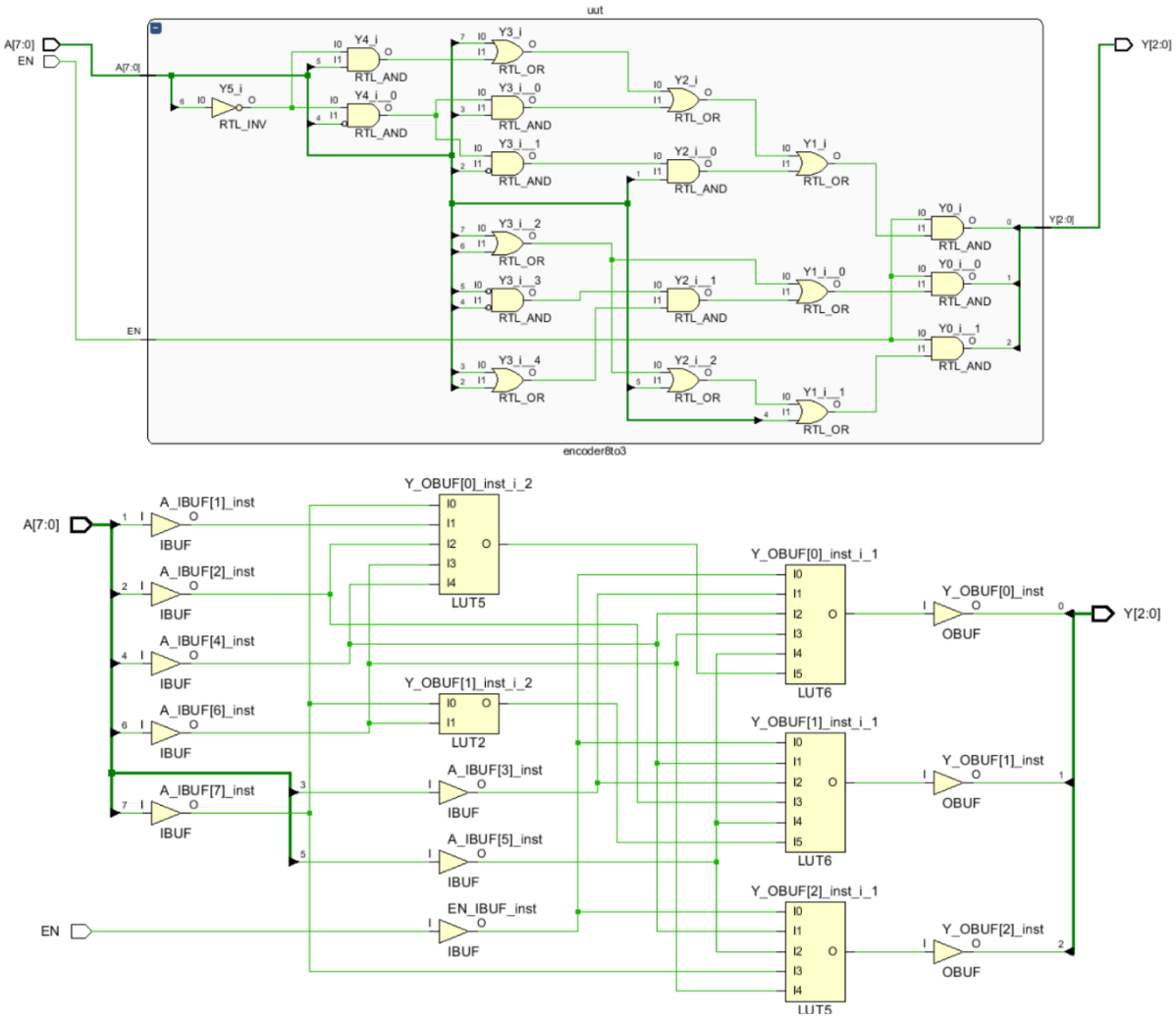
- Output Truth Table :

Truth Table for 8x3 Priority Encoder with Enable

EN	A7	A6	A5	A4	A3	A2	A1	A0	Y2	Y1	Y0
0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	1	0	0	0	1
1	0	0	0	0	0	1	0	0	0	1	0
1	0	0	0	0	1	0	0	0	0	1	1
1	0	0	0	1	0	0	0	0	1	0	0
1	0	0	1	0	0	0	0	0	1	0	1
1	0	1	0	0	0	0	0	0	1	1	0
1	1	0	0	0	0	0	0	0	1	1	1
1	1	1	0	0	0	0	0	0	1	1	1
1	0	1	1	0	0	0	0	0	1	1	0
1	0	0	0	1	1	0	0	0	1	0	0
1	0	0	0	0	1	1	0	0	0	1	1
1	0	0	0	0	0	0	1	1	0	0	1
1	1	0	1	0	1	0	1	0	1	1	1



8x3 PRIORITY CODER => Waveform Graph = ↑ ||| RTL Schematic ↓



8x3 PRIORITY CODER => RTL Schematic

Conclusion: In this experiment, the design and implementation of a priority encoder were successfully carried out. Unlike a normal encoder, the priority encoder resolves situations when multiple inputs are active by giving priority to the highest-order input. The truth table and simulation confirmed that only the most significant active input is encoded into the binary output. This ensures predictable and reliable behavior even with multiple input activations. The testbench verified all possible cases, and the results matched the theoretical expectations. Hence, the priority encoder design is validated for use in digital systems requiring conflict resolution in input signals.

Suggested Reference:

1. *Verilog HDL: A Guide to Digital Design and Synthesis, Pearson* - Samir Palnitkar
2. *Fundamentals of Digital Logic with Verilog Design, McGraw-Hill* - Stephen Brown & Zvonko Vranesic
3. *Vivado Design Suite User Guide – HDL Coding Techniques (UG901)* - Xilinx Inc.
4. IEEE Standard for Verilog Hardware Description Language (IEEE Std 1364).
5. TutorialsPoint / NPTEL lectures on Verilog and Digital Logic Design (for supplementary learning).

References used by the students:

1. *Fundamentals of Digital Logic with Verilog Design, McGraw-Hill* - Stephen Brown & Zvonko Vranesic
2. *Vivado Design Suite User Guide – HDL Coding Techniques (UG901*) - Xilinx Inc.

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:

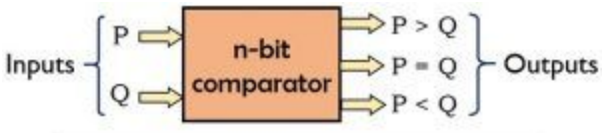
Relevant CO: CO 3 , CO 5

Objectives:

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

Basic Theory:

1. **Comparators :** A **comparator** is a combinational logic circuit that compares two binary numbers and determines their relative magnitude. It produces outputs indicating whether one number is **greater than**, **equal to**, or **less than** the other. For example, a 1-bit comparator compares single bits, while multi-bit comparators (like 4-bit or 8-bit) are used for larger binary numbers. Comparators are widely used in digital systems such as arithmetic units, sorting circuits, and control applications where decision-making based on size or equality of data is required.



Block Diagram of Comparator

- 4 – BIT COMPARATOR :
  - Design Code :

```
module comparator4bit
( input  [3:0] A , input  [3:0] B , output A_gt_B, output A_eq_B , output A_lt_B );

        assign A_eq_B = (A == B);   // Equality check
        assign A_gt_B = (A > B);    // Greater than check
        assign A_lt_B = (A < B);    // Less than check

endmodule
```

- Testbench Code :

```
module tb_comparator4bit;
        reg [3:0] A, B ; wire A_gt_B, A_lt_B, A_eq_B;

        // Instantiate the DUT (Device Under Test)
        comparator4bit uut ( .A(A) , .B(B), .A_gt_B(A_gt_B), .A_lt_B(A_lt_B),
.A_eq_B(A_eq_B) );

    initial begin
        $display("Sample Truth Table for 4-bit Comparator\n");
        $display("   A     B   | A>B  A<B  A=B");
        $display("---------------------------------");

        // Representative test cases
        A = 4'b0000; B = 4'b0000; #5; $display(" %4b  %4b  |   %b    %b    %b", A , B ,
A_gt_B, A_lt_B, A_eq_B);
        A = 4'b1010; B = 4'b0110; #5; $display(" %4b  %4b  |   %b    %b    %b", A , B ,
A_gt_B, A_lt_B, A_eq_B);
        A = 4'b0101; B = 4'b1100; #5; $display(" %4b  %4b  |   %b    %b    %b", A , B ,
A_gt_B, A_lt_B, A_eq_B);
        A = 4'b1111; B = 4'b1111; #5; $display(" %4b  %4b  |   %b    %b    %b", A , B ,
A_gt_B, A_lt_B, A_eq_B);
        A = 4'b0011; B = 4'b0100; #5; $display(" %4b  %4b  |   %b    %b    %b", A , B ,
A_gt_B, A_lt_B, A_eq_B);

        $display("---------------------------------");
    $finish;
    end
endmodule
```

- Output Truth Table :

```
        Sample Truth Table for 4-bit Comparator

          A     B   | A>B  A<B  A=B
        ---------------------------------
         0000  0000 |   0    0    1
         1010  0110 |   1    0    0
         0101  1100 |   0    1    0
         1111  1111  |   0    0    1
         0011  0100 |   0    1    0
        ---------------------------------
```

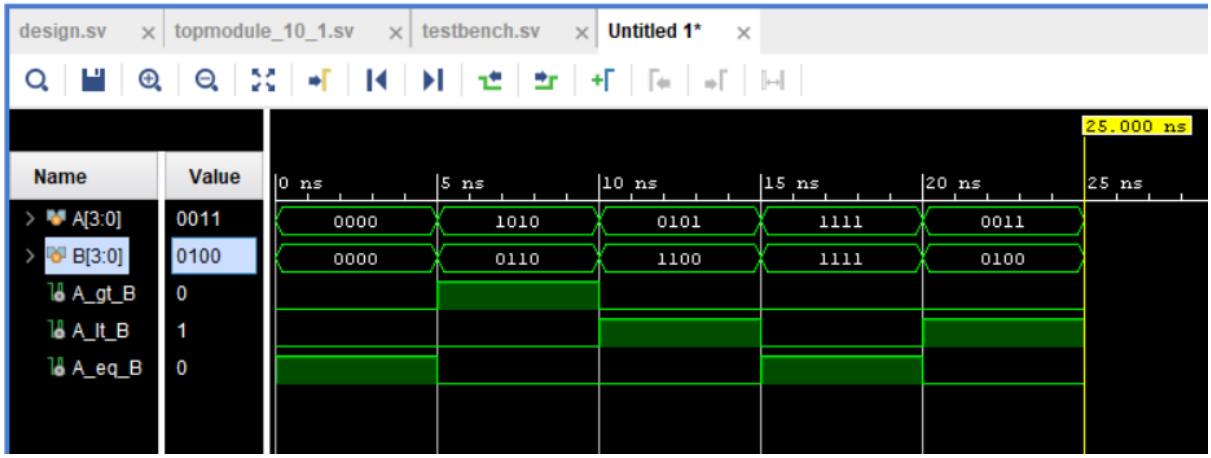
- Boolean table :

A [3:0]	B[3:0]	A > B	A < B	A = B
a3a2a1a0	b3b2b1b0	0	0	1
a3a2a1a0	b3b2b1b0	0	1	0
a3a2a1a0	b3b2b1b0	1	0	0

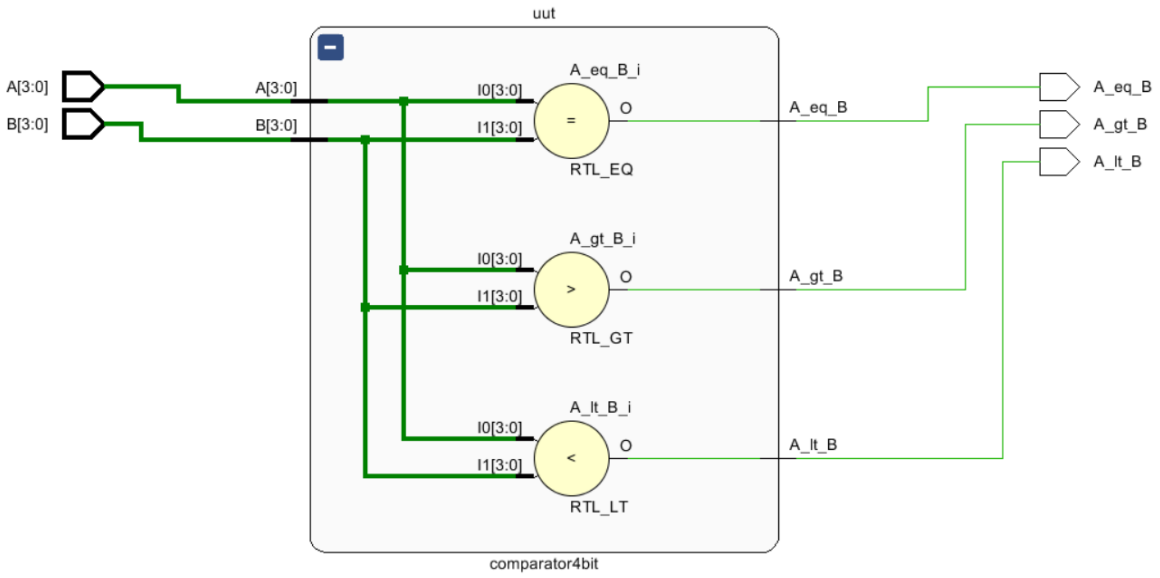
For A = B : [if (a0 = b0) and (a1 = b1) and (a2 = b2) and (a3 = b3)]

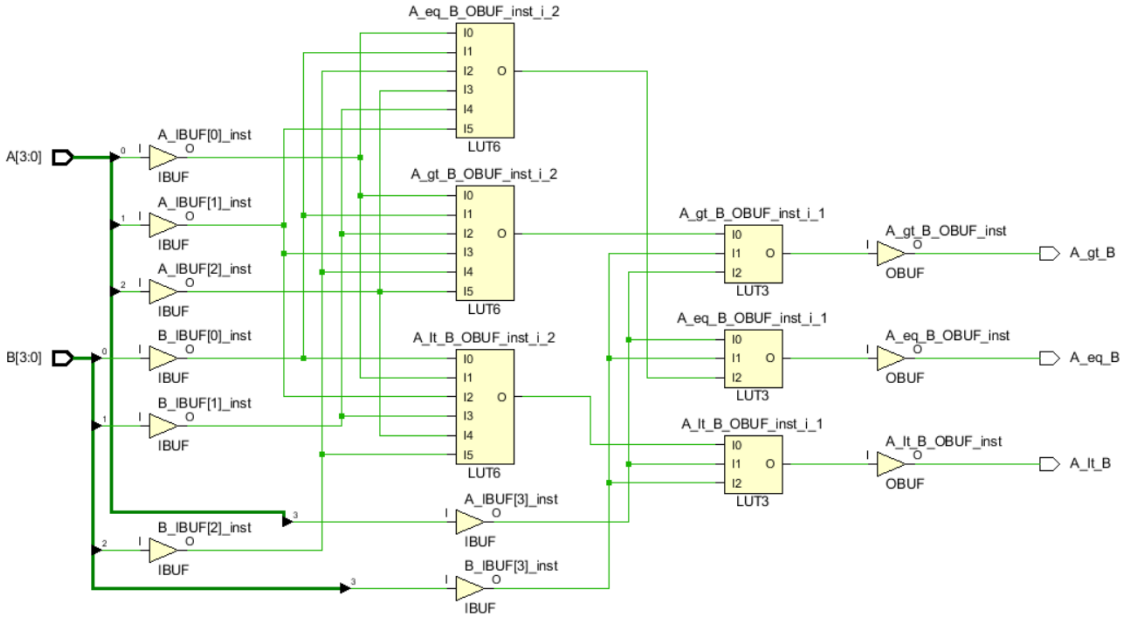
For A < B : [if (a0 < b0)
or ((a0 >= b0) and (a1 < b1))
or ((a1 >= b1) and (a2 < b2))
or ((a2 >= b2) and (a3 < b3))
]

For A > B : [if (a0 > b0)
or ((a0 <= b0) and (a1 > b1))
or ((a1 <= b1) and (a2 > b2))
or ((a2 <= b2) and (a3 > b3))
]



4 - BIT COMPARATOR => Waveform Graph ↑ || RTL Schematic ↓





4 - BIT COMPARATOR => Synthesis Schematic

Conclusion: The comparator experiment shows how digital circuits compare binary numbers, producing equality, greater-than, and less-than outputs, with only one active at a time. This verifies the implemented logic and demonstrates how comparison circuits can be efficiently designed in Verilog, reinforcing practical digital design concepts..

Suggested Reference:

1. *Verilog HDL: A Guide to Digital Design and Synthesis, Pearson* - Samir Palnitkar
2. *Fundamentals of Digital Logic with Verilog Design, McGraw-Hill* - Stephen Brown & Zvonko Vranesic
3. *Vivado Design Suite User Guide – HDL Coding Techniques (UG901)* - Xilinx Inc.
4. IEEE Standard for Verilog Hardware Description Language (IEEE Std 1364).
5. TutorialsPoint / NPTEL lectures on Verilog and Digital Logic Design (for supplementary learning).

References used by the students:

1. *Fundamentals of Digital Logic with Verilog Design, McGraw-Hill* - Stephen Brown & Zvonko Vranesic
2. *Vivado Design Suite User Guide – HDL Coding Techniques (UG901*) - Xilinx Inc.

Rubric wise marks obtained:

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

Experiment No: 11

Date: _____

Aim: Design BCD and Ripple Carry Adder using Verilog / VHDL.

Competency and Practical Skills:

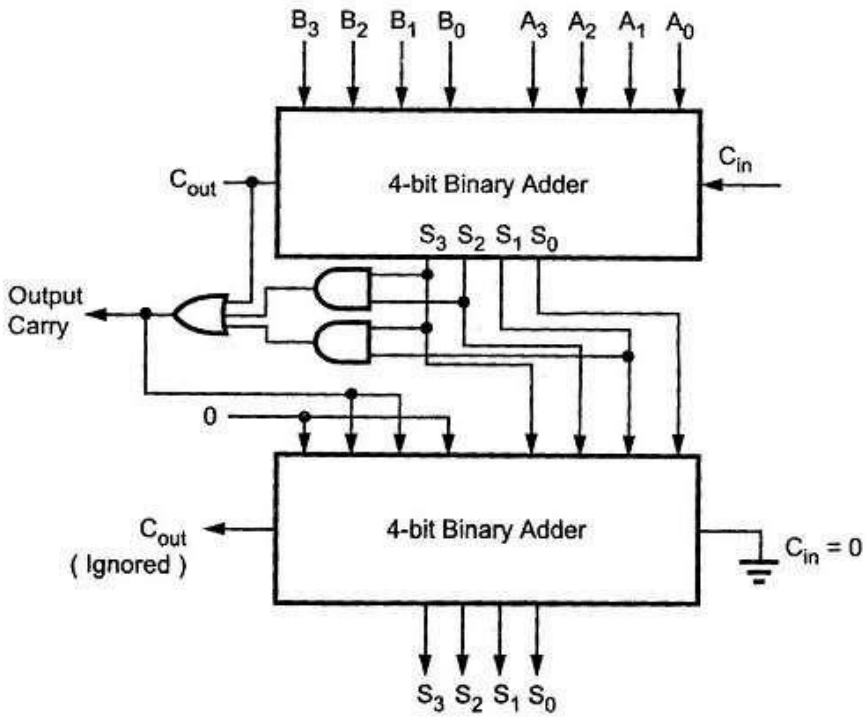
Relevant CO:

Objectives:

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

Basic Theory:

BCD Adder Block diagram:



Program Code:

RTL Schematic:

Output waveform:

Conclusion:

Suggested Reference:

References used by the students:

Rubric wise marks obtained:

Rubrics	1	2	3	4	5	Total
Marks						

Experiment No: 12

Date: _____

Aim: Design of S-R and D latches with structural / behavioral modeling using Verilog / VHDL.

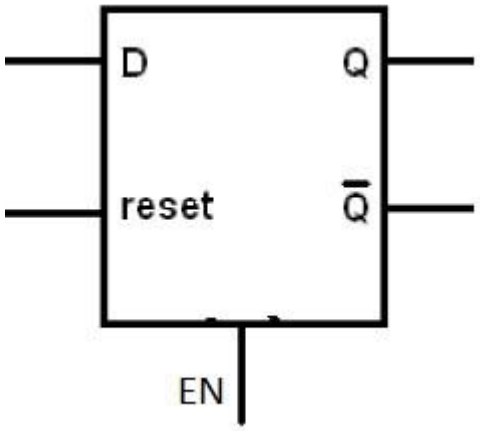
Competency and Practical Skills:

Relevant CO:

Objectives:

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

Basic Theory:

D – Latch	SR Latch
	
Truth Table of D - Latch	Truth Table SR Latch

Program Code:

Sample code:

```
module Dlatch1( D, EN, RSTN,
Q );

input D, EN, RSTN;  // reset - RSTN is Active low
output reg Q;


always @ (EN or RSTN or D)
        if (!RSTN)      // If reset is asserted then Q will be 0
                Q <= 0;
        else
                if (EN)
                Q <= D;   // Q will get D only when EN is high

endmodule
```

RTL Schematic:

Output waveform:

Conclusion:

Suggested Reference:

References used by the students:

Rubric wise marks obtained:

Rubrics	1	2	3	4	5	Total
Marks						

Experiment No:13

Date: _____

Aim : Design positive edge triggered D FF with asynchronous / synchronous active high reset using Verilog / VHDL.

Competency and Practical Skills:

Relevant CO:

Objectives:

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

Basic Theory:

Program Code:

RTL Schematic:

Output waveform:

Conclusion:

Suggested Reference:

References used by the students:

Rubric wise marks obtained:

Rubrics	1	2	3	4	5	Total
Marks						

Experiment No:14

Date: _____

Aim : Design serial in serial out and serial in parallel out shift registers using Verilog / VHDL.

Competency and Practical Skills:

Relevant CO:

Objectives:

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

Basic Theory:

Program Code:

RTL Schematic:

Output waveform:

Conclusion:

Suggested Reference:

References used by the students:

Rubric wise marks obtained:

Experiment No: 15

Date: _____

Aim : Design Counter using Verilog / VHDL.

1. **BCD Counter.**
2. **Up-Down Counter**

Competency and Practical Skills:

Relevant CO:

Objectives:

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

Basic Theory:

Program Code:

RTL Schematic:

Output waveform:

Conclusion:

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:

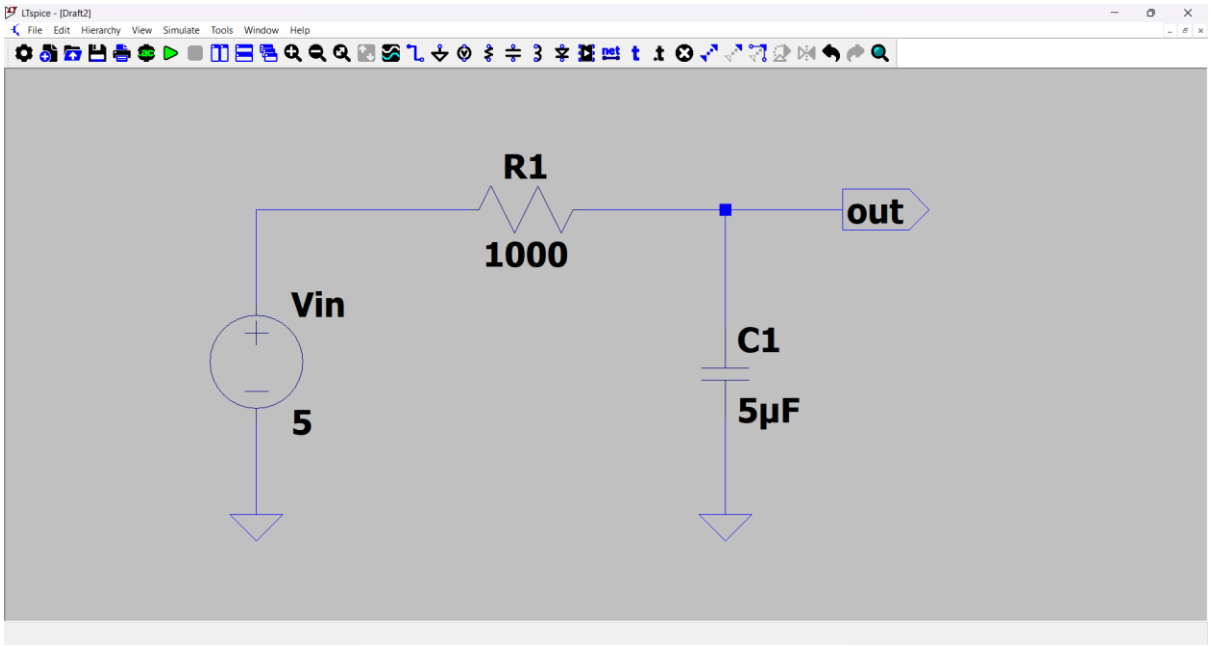
Equipment / Instruments: Laptop or Computer with LT Spice tool installed .

Introduction:

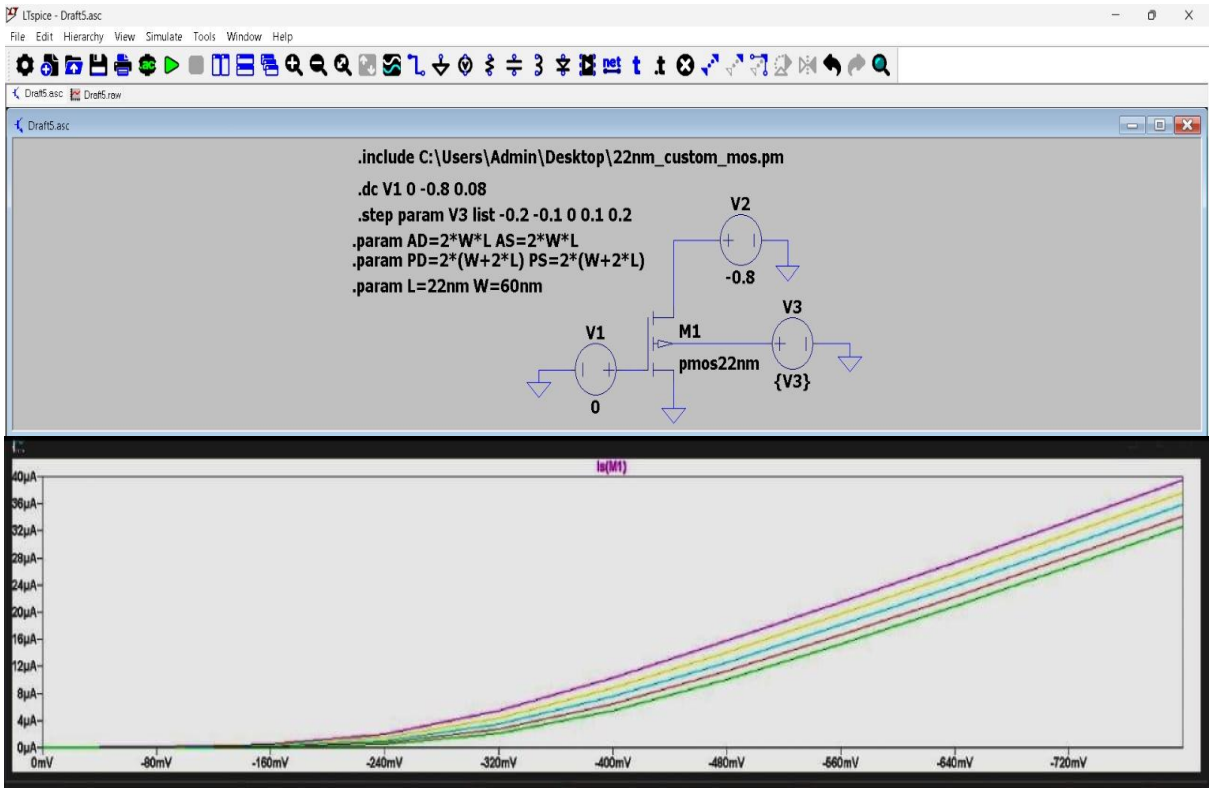
LT SPICE Tool

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

.

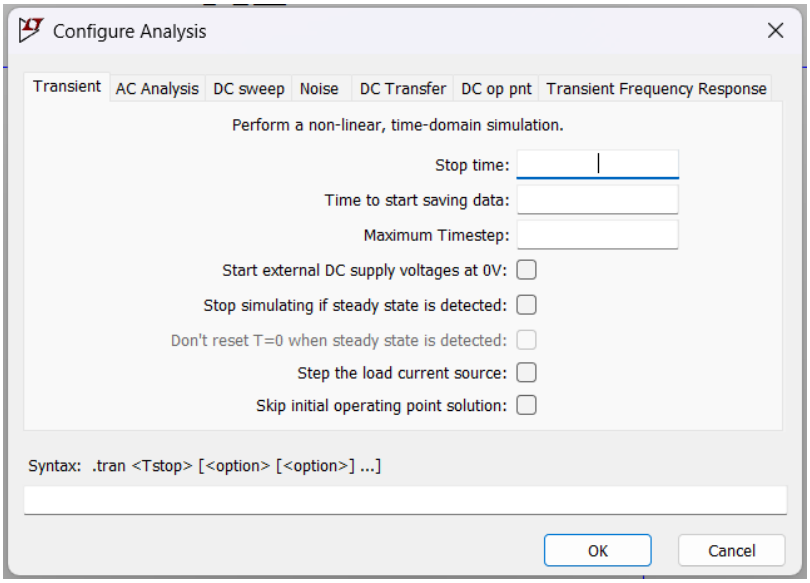
3. **Simulation Types**

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.

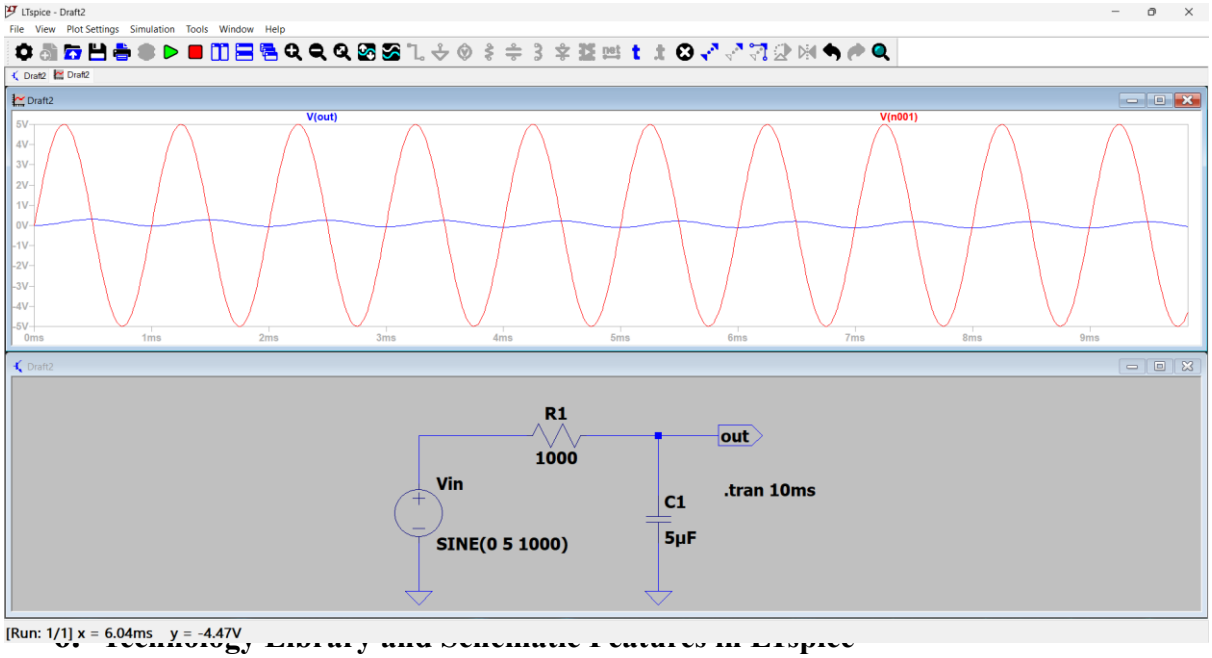


4. **Sub-circuit and Hierarchical Design**

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.



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: _____

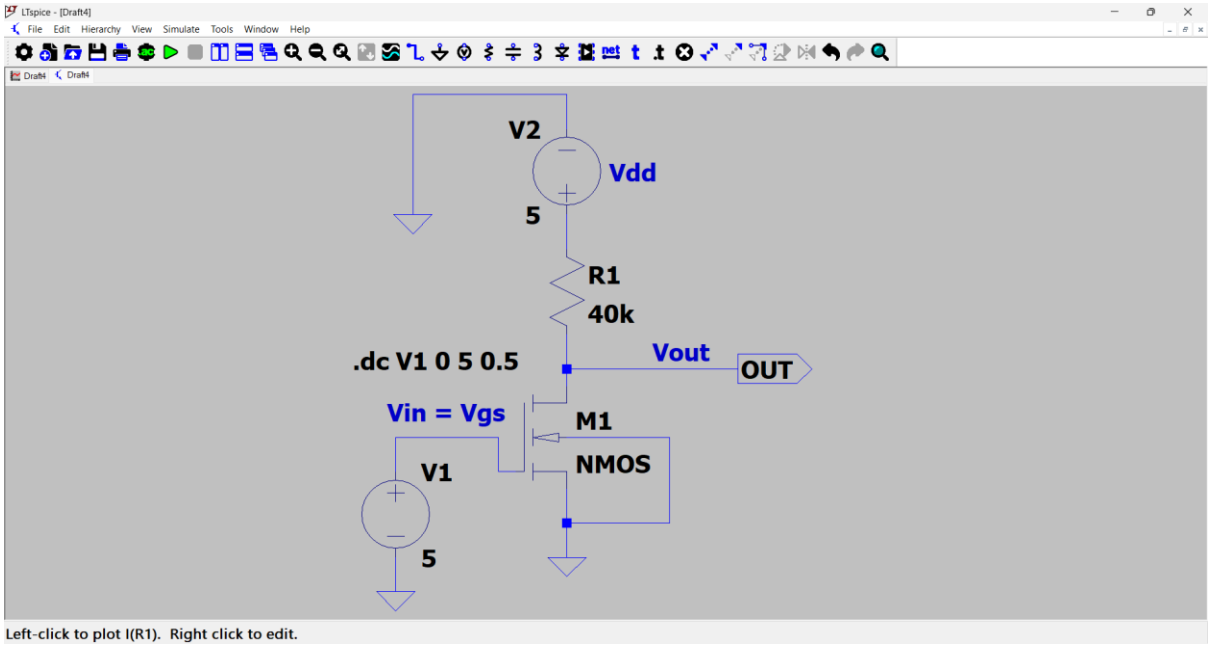
Aim: Implementation of Resistive load and CMOS inverters using LT SPICE

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) :*

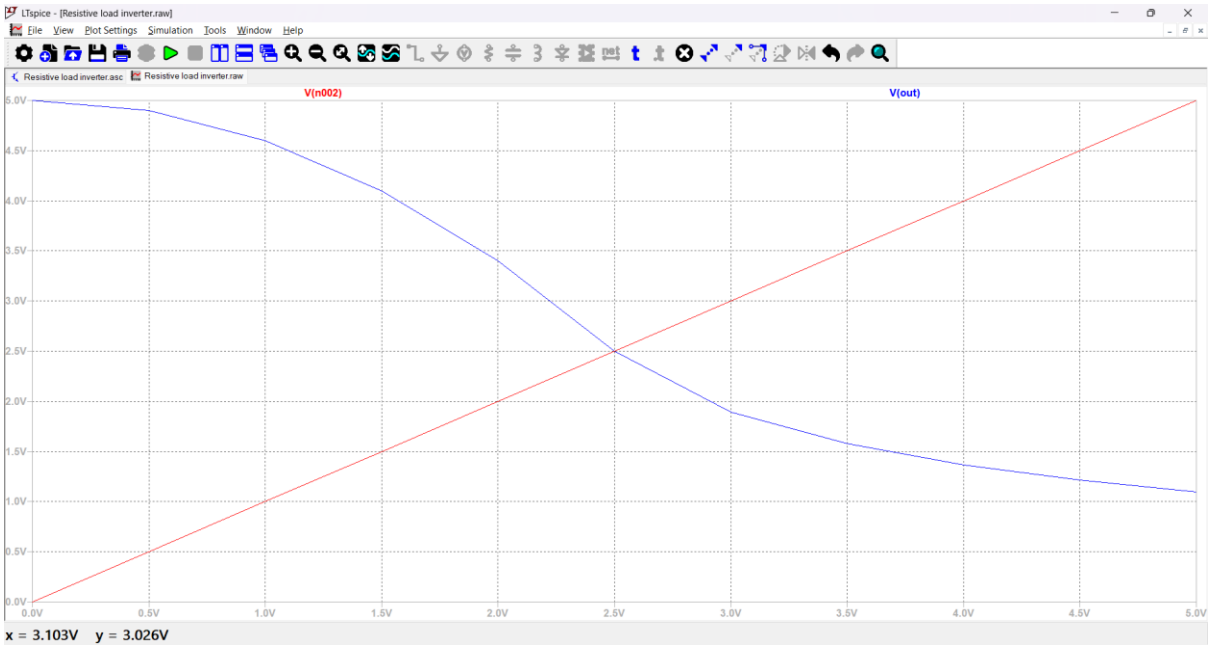


Resistive Load Inverter => Circuit Schematic

```
* 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
```

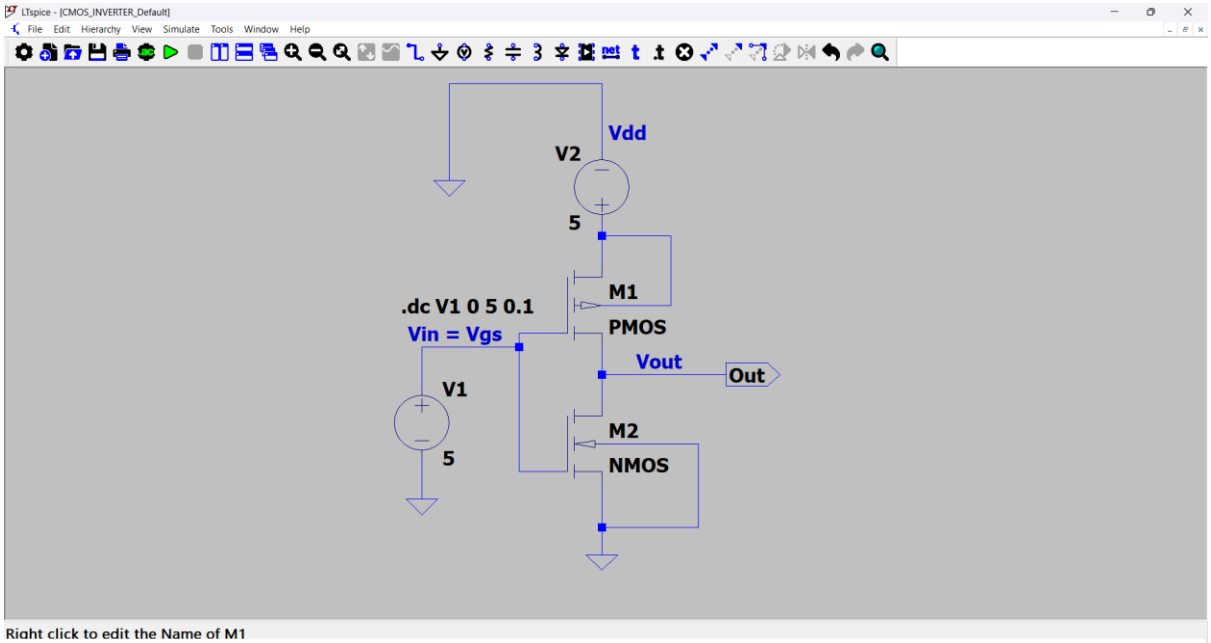
Resistive Load Inverter => SPICE NETLIST

- Simulation Waveforms :



Resistive Load Inverter => Voltage Graph (Simulation)

➢ *CMOS Inverter Layout (Using LT SPICE) :*

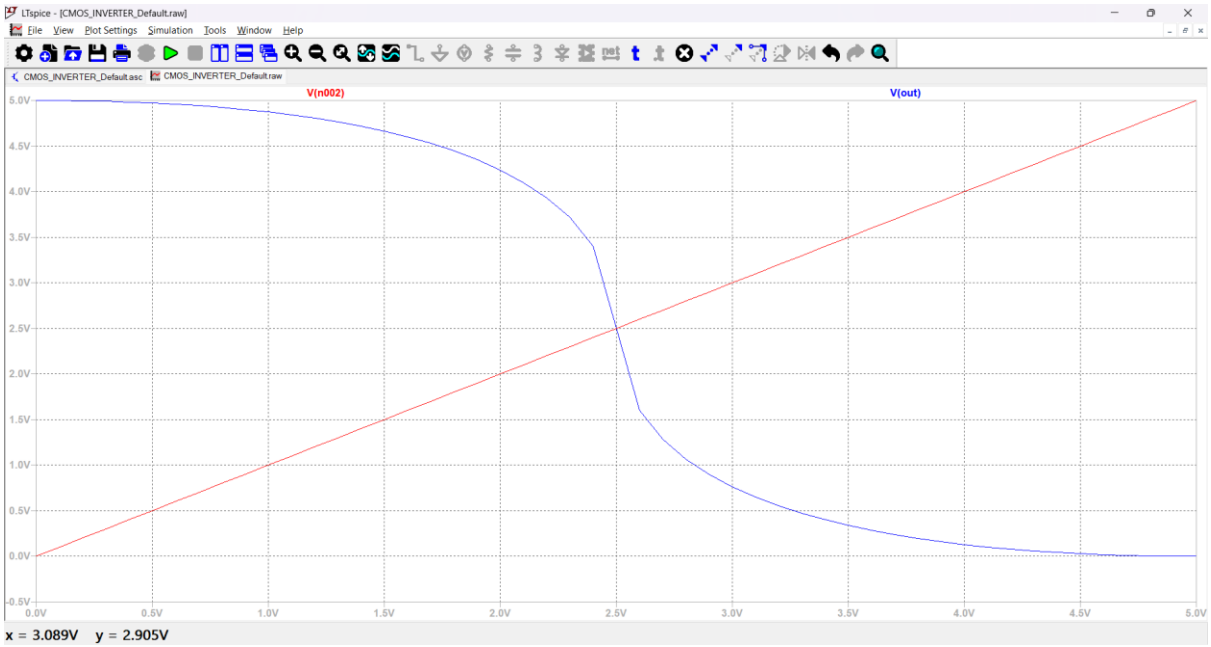


CMOS INVERTER (Deafult Lib) => Circuit Schematic

```
* 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
```

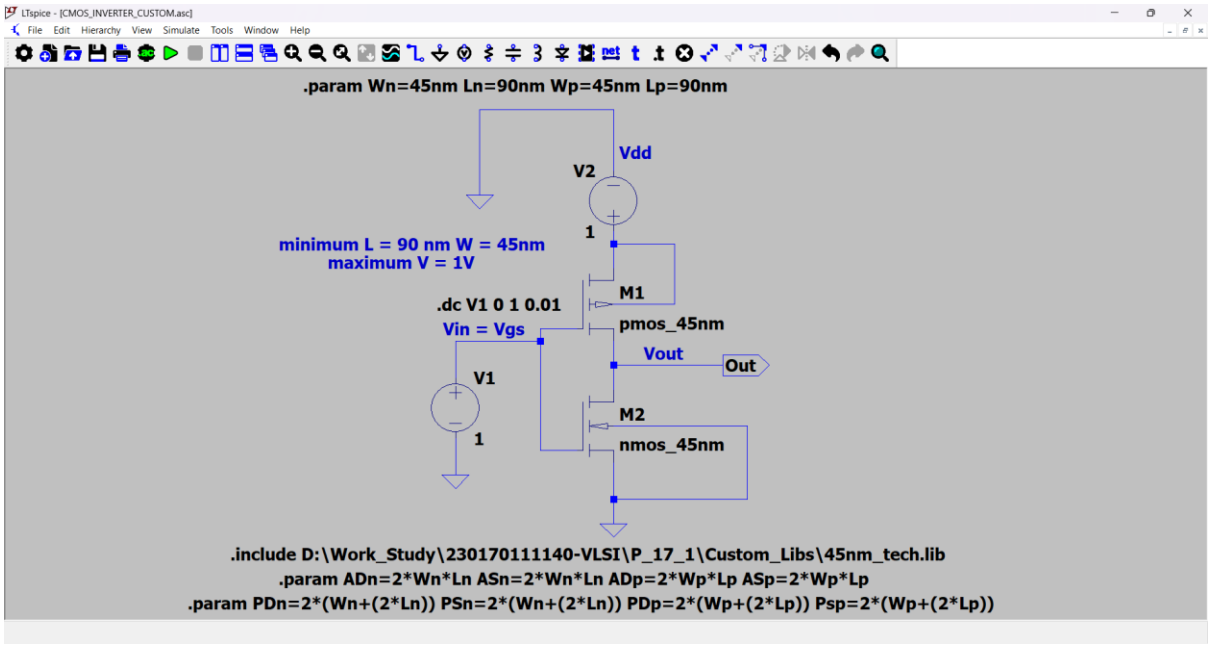
CMOS INVERTER (Deafult Lib) => SPICE NETLIST

- Simulation Waveforms :



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

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



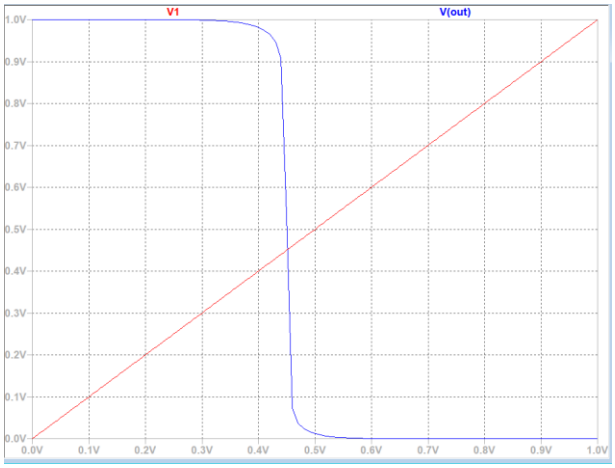
CMOS INVERTER (45nm_tech.lib) => Circuit Schematic

```
* 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
```

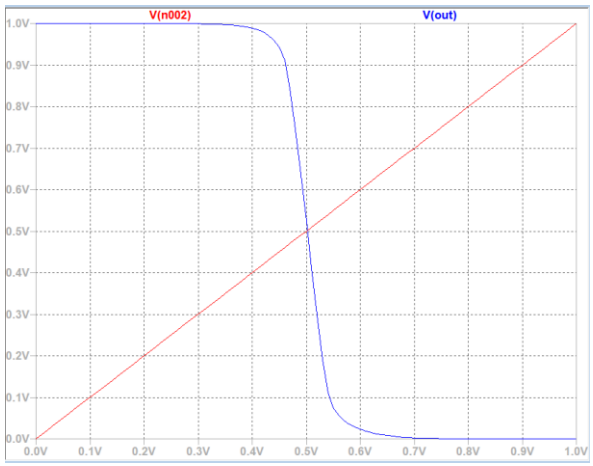
CMOS INVERTER (45nm_tech.lib) => SPICE NETLIST

- Simulation Graphs :

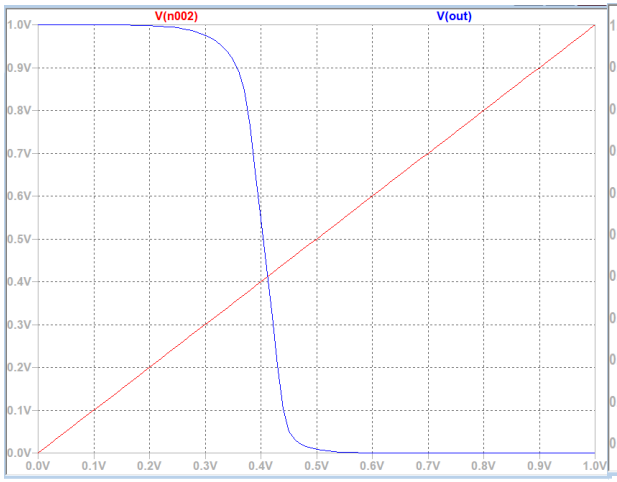
CMOS INVERTER (45nm_tech.lib) => Voltage Graph (Simulation)



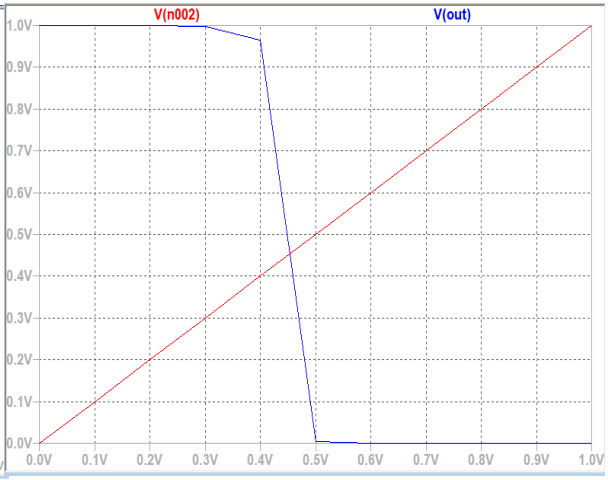
→ Wp = Wn = 45nm | Ln= Ln = 90nm



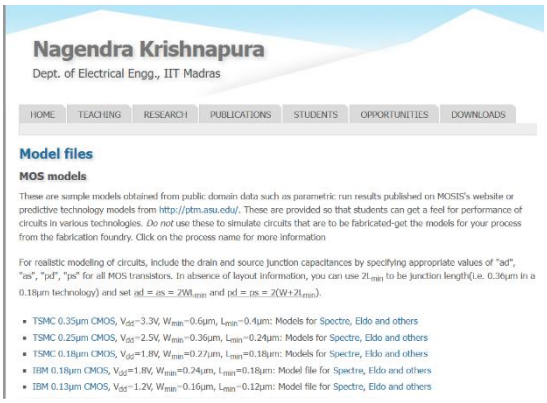
→ Wp = 22.5nm , Lp = 45nm
→ Wn = 45nm , Ln = 90nm



→ 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						

Experiment No: 18

Date: _____

Aim: Implementation of CMOS NAND and NOR gate using LT SPICE

Competency and Practical Skills:

Relevant CO: CO 1 , CO 5

Objectives:

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

- *2 – Input NAND Gate Layout (CMOS) (LT SPICE) :*

- Simulation Waveforms :

- *2 – Input NOR Gate Layout (CMOS) (LT SPICE) :*

- Simulation Waveforms :

Conclusion : The LT Spice simulation of CMOS NAND and NOR gates confirmed their correct logic functionality. Both gates provided rail-to-rail outputs with very low static power dissipation, showcasing the advantage of CMOS design. The NAND gate output was low only when all inputs were high, while the NOR gate output was high only when all inputs were low. The transfer characteristics showed sharp switching and good noise margins. Hence, CMOS NAND and NOR gates are reliable, power-efficient, and form the fundamental building blocks of digital logic circuits.

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						