A

Laboratory Manual for

VLSI Design
(3151105)

B. E. Electronics and Communication
Semester 5

Directorate of Technical Education, Gandhinagar, Gujarat

VGEC, Chandkheda, Ahmedabad

Department of Electronics & Communication Engineering

Certificate

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

Place: _____
Date: _____

Name and Sign of Faculty member

Head of the Department

Practical – Course Outcome Matrix

Course Outcomes (COs):

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

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

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

Index
(Progressive Assessment Sheet)

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

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

Experiment No: 1

Date: _____

Aim: Introduction to FPGA & CPLD.

Competency and Practical Skills:

Relevant CO:

Objectives:

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

Basic Theory:

What is an FPGA?

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

What are the differences between an ASIC and an FPGA?

An ASIC (Application Specific Integrated Circuit) is a chip designed for a fixed and dedicated function.

It provides very high performance, low power consumption, and compact area optimization.

Once fabricated, its functionality cannot be changed, making it suitable only for mass production.

The development of ASICs requires high cost and long design time but results in low cost per unit at large volumes.

An FPGA (Field Programmable Gate Array), on the other hand, is a reconfigurable device.

It consists of configurable logic blocks and programmable interconnects that can be reprogrammed.

FPGAs offer shorter time-to-market and flexibility for design modifications even after deployment.

They are more expensive per unit compared to ASICs but require less initial investment.

Thus, ASICs are ideal for stable, high-volume products, whereas FPGs are best for prototyping and evolving applications.

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).

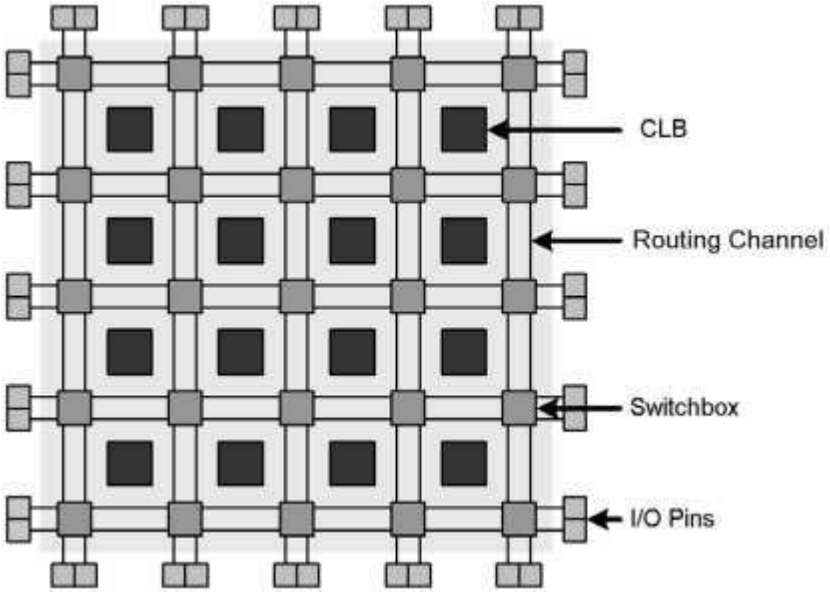


Fig 1.1 General Architecture of FPGA

Explain the structure of Switch matrix:

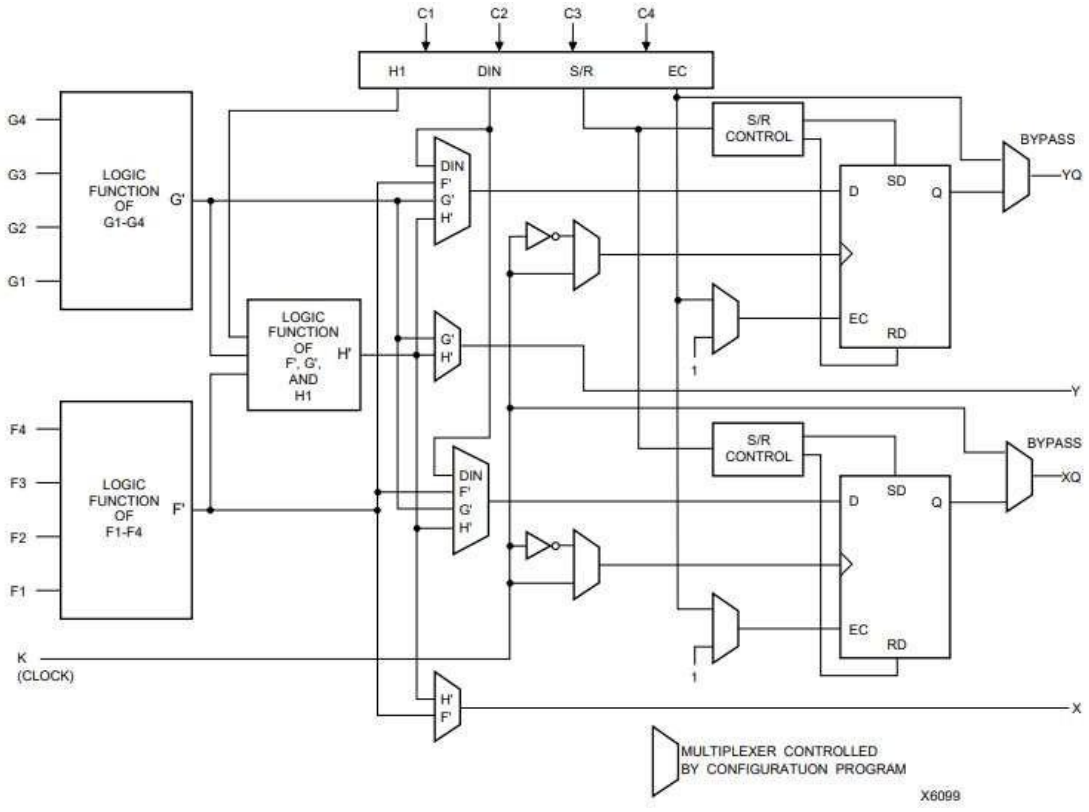


Fig 1.2 Simplified Block Diagram of XC4000-Families CLB

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	FPGA	CPLD
1	Made up of **Configurable Logic Blocks (CLBs)** interconnected by programmable routing.	Built using a **programmable AND/OR array** connected to microcells.
2	Mostly **SRAM-based**, volatile, requires reconfiguration after power ON.	Usually **EEPROM/Flash-based**, non-volatile, retains configuration after power OFF.
3	Suitable for **very large and complex designs** (tens of thousands to millions of gates).	Suitable for **medium-scale logic designs** (few thousand gates).
4	Offers **high flexibility and reconfigurability**, ideal for prototyping.	Offers **predictable timing** and fast performance for simple logic.
5	**Higher cost and power consumption** compared to CPLDs.	**Lower cost and power consumption**, economical for small systems.
6	Used in DSP, AI, image processing, data Centers, aerospace, and advanced compu Ting.	Used in glue logic, address decoding, I/O interfacing, and simple controllers.

Conclusion:

In this experiment, we studied the architecture and working of FPGA and CPLD.
We learned that FPGAs are highly flexible, reconfigurable devices consisting of configurable logic blocks and switch matrices, making them suitable for complex and high-performance applications.
On the other hand, CPLDs are based on programmable AND/OR arrays with microcells, suitable for medium-scale control logic with low power consumption and predictable timing.
We also understood the differences between ASIC, FPGA, and CPLD, and identified their advantages and applications in real-world systems.
Thus, FPGA and CPLD are important programmable devices that reduce development time, provide design flexibility, and are widely used in modern electronics and embedded systems.

Suggested Reference:

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

References used by the students:

Rubric wise marks obtained:

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

Experiment No: 2

Date: _____

Aim: Introduction to Hardware Description Languages (HDLs).

Competency and Practical Skills:

Relevant CO:

Objectives:

Equipment / Instruments: Laptop or Computer with Xilinx -Vivado

Basic Theory:

What Are Hardware Description Languages (HDLs)?

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

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

What Is VHDL?

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

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

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

VHDL Functionality

These simulations alternate between two modes:

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

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

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

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

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

What Is Verilog?

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

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

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

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

VHDL vs Verilog

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

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

Conclusion:

In this experiment, we learned about Hardware Description Languages (HDLs) and their importance in digital circuit design. We studied that VHDL is strongly typed and descriptive, while Verilog is compact, closer to C, and simpler to learn. HDLs facilitate efficient modelling, simulation, and verification of hardware. Thus, HDLs provide a powerful way to design complex digital systems with accuracy and flexibility.

Suggested Reference:

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

References used by the students:

Rubric wise marks obtained:

Rubrics	1	2	3	4	5	Total
Marks						

Experiment No: 3

Date: _____

Aim: Introduction to Xilinx -Vivado

Competency and Practical Skills:

Relevant CO:

Objectives:

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

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.

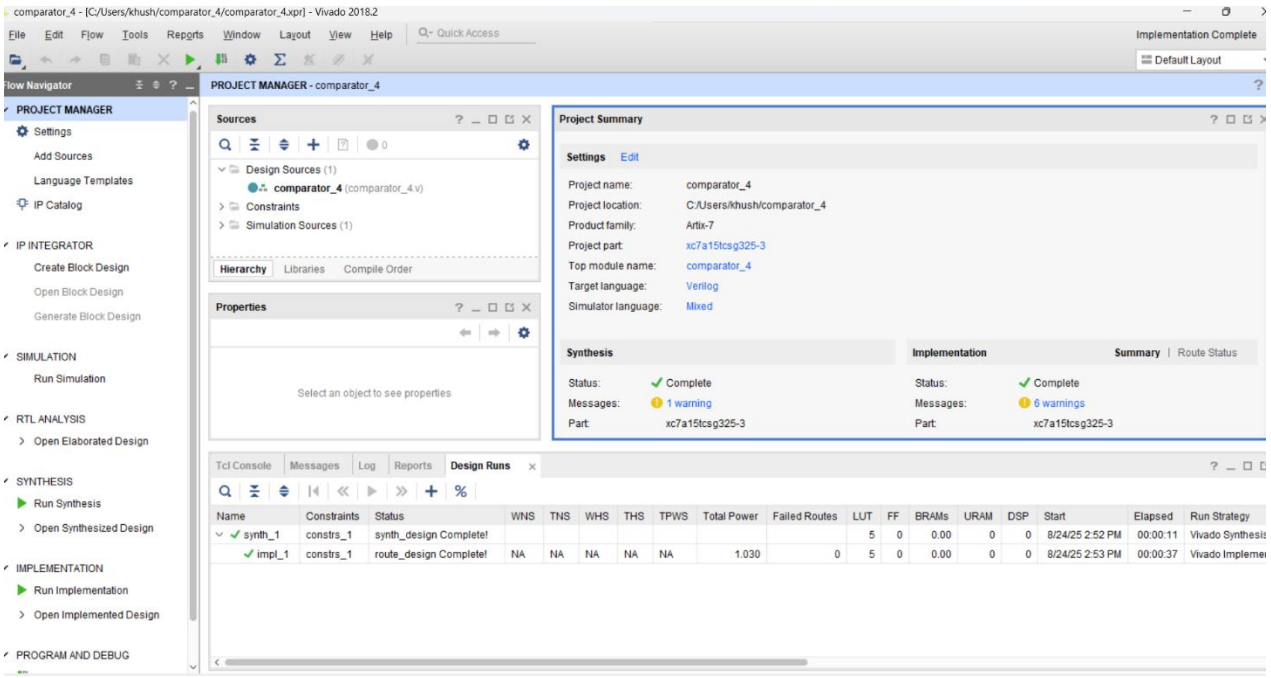


Fig. 2.1 Project Navigator

The Project Navigator interface is divided into four panel sub-windows, as seen in Figure 2.1.

- 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.

Steps for the whole process in Vivado:

1. Create Project

- Open Vivado → **Create Project** → Name: not_1 → Location → Next.
- Select **RTL Project** → Next.
- Skip adding sources for now → Next.
- Choose FPGA part (e.g., Artix-7 xc7a15tcsg325-3) → Next → Finish.

2. Add Design Code (not_1.v)

- Flow Navigator → **Add Sources → Add or Create Design Sources**.
- **Create File** → Name: not_1.v → OK → Finish.
- Paste design code:

```
module not_1(input A,output B);
assign B = ~A;
endmodule
```

- Save.

3. Add Testbench (tb_not_1.v)

- Flow Navigator → **Add Sources → Add or Create Simulation Sources**.
- **Create File** → Name: tb_not_1.v → OK → Finish.
- Paste Vivado-compatible testbench:
-

```
timescale 1ns / 1ps
module tb_not_1();
reg a;
wire b;
not_1 dut (.A(a), .B(b));
initial begin
a = 1'b0;
forever #5 a = ~a;
end
initial begin
#30 $stop;
end
endmodule
```

- Save.

4. Run Behavioral Simulation (Waveforms)

- Flow Navigator → **Simulation → Run Simulation → Run Behavioral Simulation**.
- Vivado waveform window opens.
- Observe: a toggles (0→1→0→1) every 5 ns, and b is always the inverted output.

5. Generate RTL Schematic

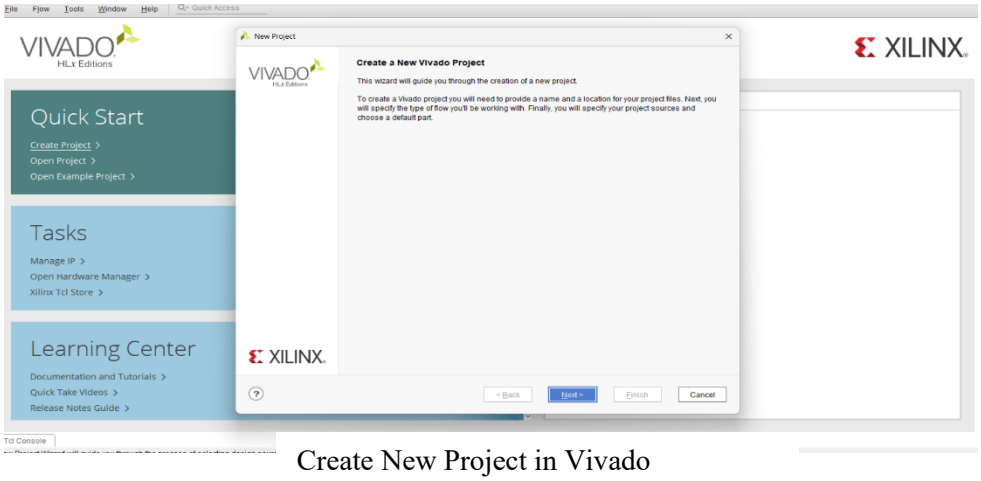
- Flow Navigator → **Open Elaborated Design**.
- View RTL schematic (NOT gate block).

6. Run Synthesis + View Synthesized Schematic

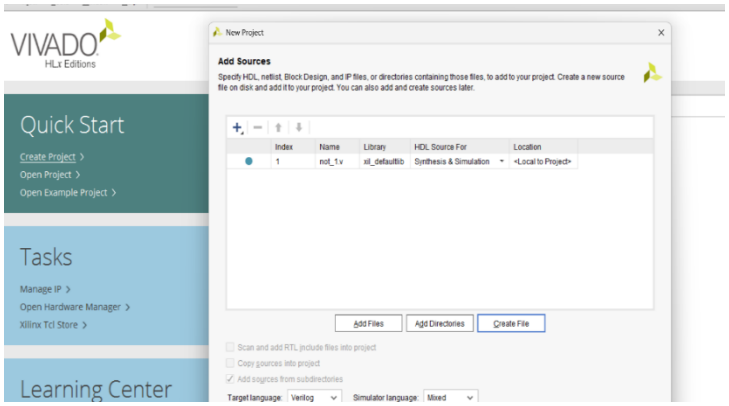
- Flow Navigator → **Synthesis → Run Synthesis**.
- After run → **Open Synthesized Design**.
- View FPGA-primitive mapped schematic (NOT implemented using FPGA LUTs).

7. (Optional) Implementation & Device View

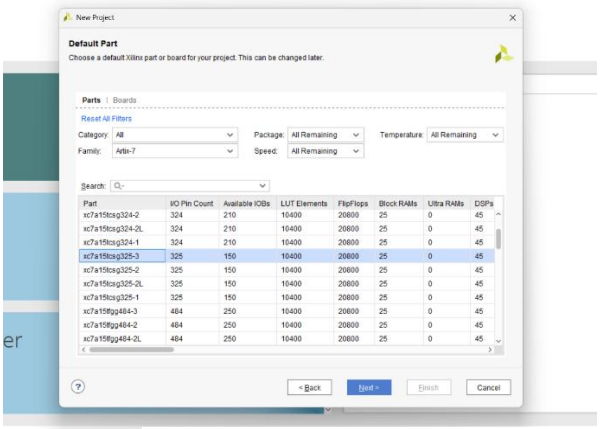
- Flow Navigator → **Run Implementation**.
- Device window shows placement inside FPGA (colored regions, CLBs, LUTs).



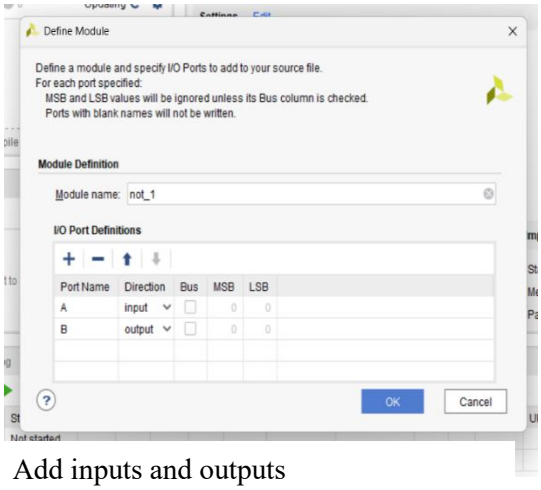
Create New Project in Vivado



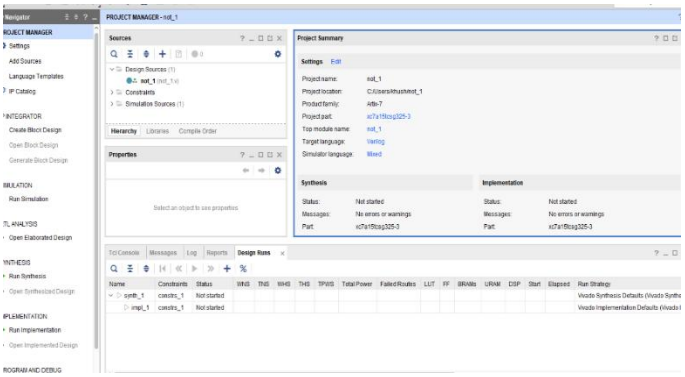
Enter Project Name and Location



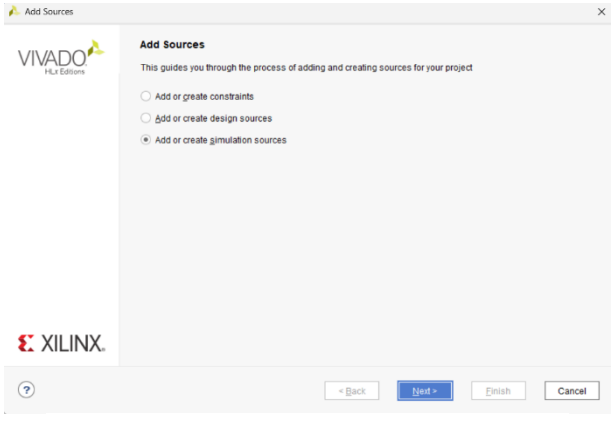
Select the family



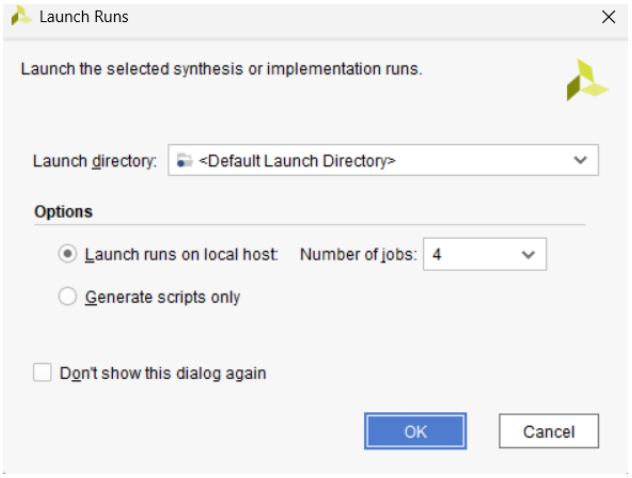
Add inputs and outputs



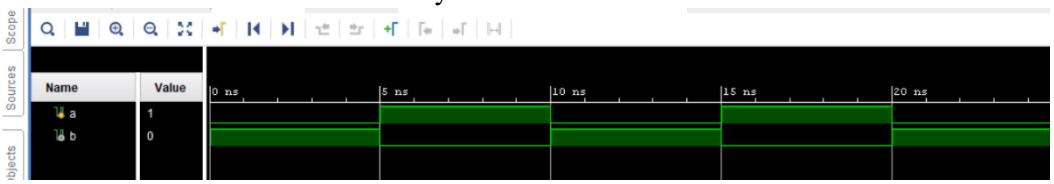
Window after creating a project



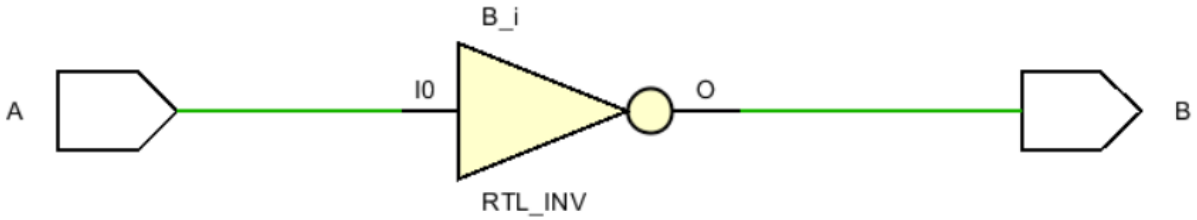
Add simulation source for testbench



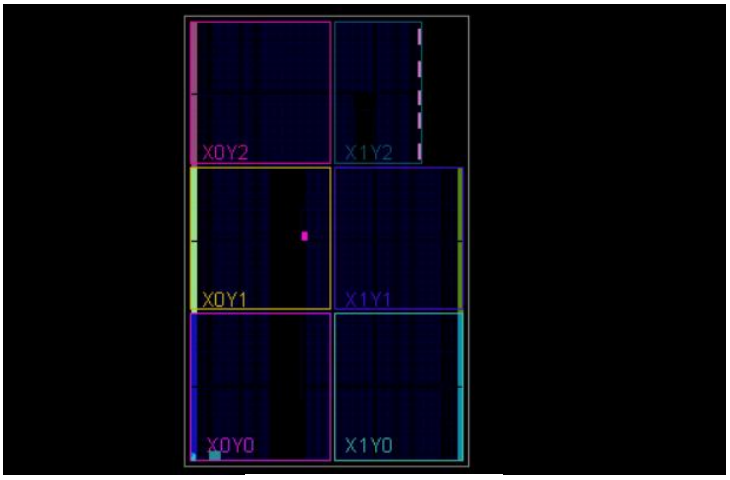
Synthesis simulation



Waveform



RTL Schematic



Synthesis design

Conclusion:

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

Suggested Reference:

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

References used by the

students: Rubric wise marks

obtained:

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

Experiment No: 4

Date: _____

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

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

Competency and Practical Skills: Basic Digital Design

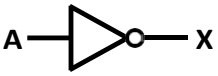
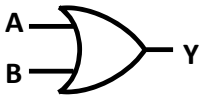
Relevant CO: CO5

Objectives: Understanding of different modeling styles in Verilog HDL.

Equipment / Instruments: Laptop or Computer with Xilinx Vivado.

Basic Theory:

Part – 1 Logic Gate Implementation

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

XOR Gate

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

XNOR Gate

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

1) NOT GATE

Design Code:

(a) Structural Modeling

```
module NOT_1(
input A,
output B
);
not n1(B,A);
endmodule
```

(b) Dataflow Modeling

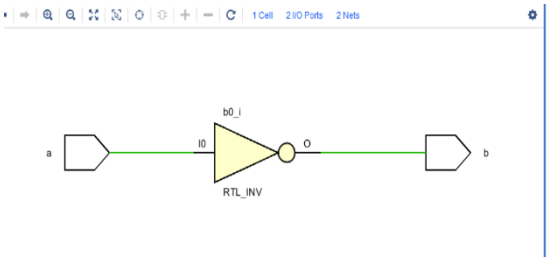
```
module NOT_2(
input A,
output B
);
assign B=~A;
endmodule
```

(c) Behavioural Modeling

```
module NOT_3(
 input A,
output reg B
);
always @(A,B)
begin
B = ~A;
end

endmodule
```

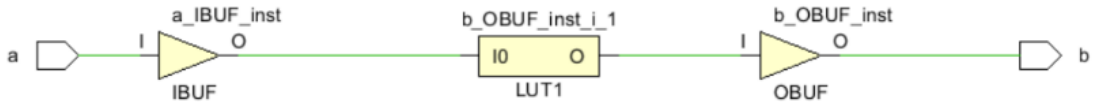
Testbench code:

```
module tb_NOT_1();
  reg a;
  wire b;
  NOT_1 dut(b,a);
  initial begin
    $dumpfile("NOT_1.vcd");
    $dumpvars(1,tb_NOT_1);
    #30 $finish();
  end
  initial begin
    a=1'b0;
  end
  always begin
    #5 a=~a;
  end
  endmodule
```



RTL Schematic

Waveform



Synthesis Schematic

2) OR GATE

Design Code:

(a) Structural Modeling

```
module OR_1(
input A,B
output C
);
or o1(C,A,B);
endmodule
```

(b) Dataflow Modeling

```
module OR_2(C,A,B);
  input A,B;
  output  C;
 assign C=A|B;
endmodule
```

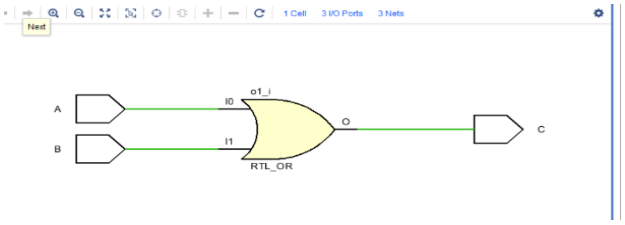
(c) Behavioural Modeling

```
 module OR_3(C,A,B);
  input A,B;
  output reg C;
);
always @(A,B) begin
      C = A|B;
end
endmodule
```

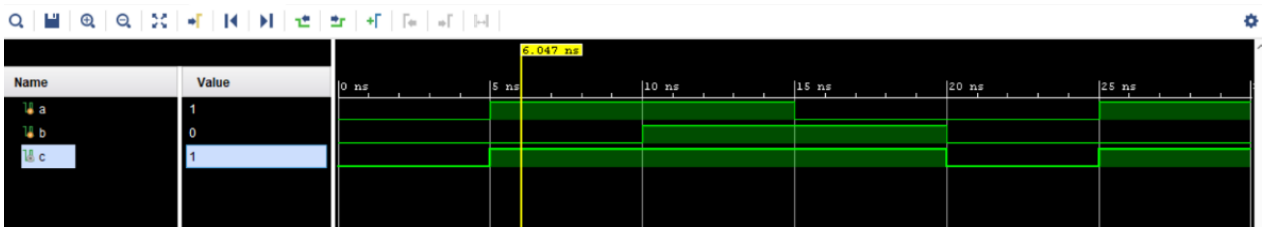
Testbench code:

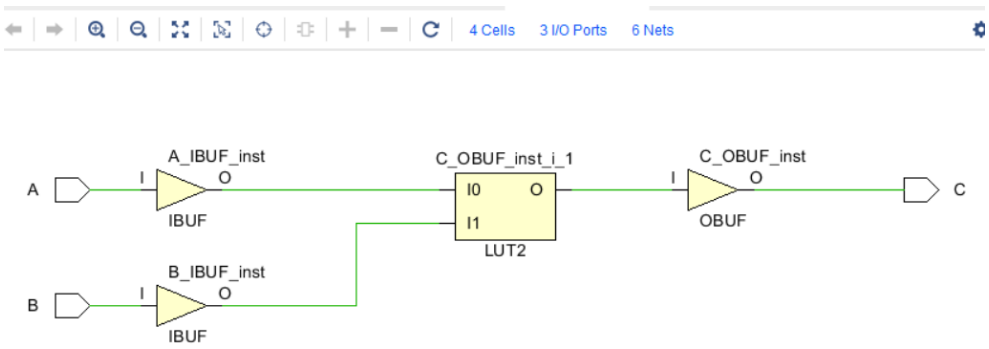
```
module tb_OR_1();
  reg a,b;
  wire c;
  OR_1 dut(c,a,b);
  initial begin
    $dumpfile("OR_1.vcd");
    $dumpvars(1,tb_OR_1);
    #30 $finish();
  end
  initial begin
    a=1'b0;
    b=1'b0;
  end
 always begin
    #5 a=~a;
    #5 b=~b;
  end
  endmodule
```

RTL Schematic



Waveform





Synthesis Schematic

3) AND GATE

Design Code:

(a) Structural Modeling

```
module AND_1(
input A,B
output C
);
and a1(C,A,B);
endmodule
```

(b) Dataflow Modeling

```
module AND_2(C,A,B);
  input A,B;
  output  C;
 assign C=A&B;
endmodule
```

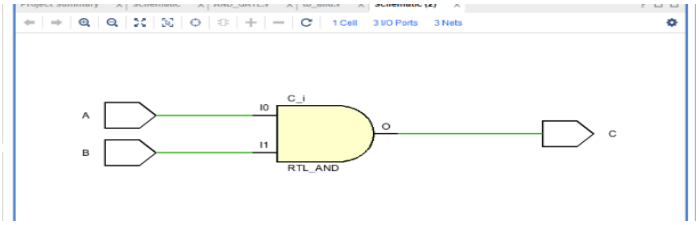
(c) Behavioural Modeling

```
 module AND_3(C,A,B);
  input A,B;
  output reg C;
);
always @(A,B) begin
      C = A&B;
end
endmodule
```

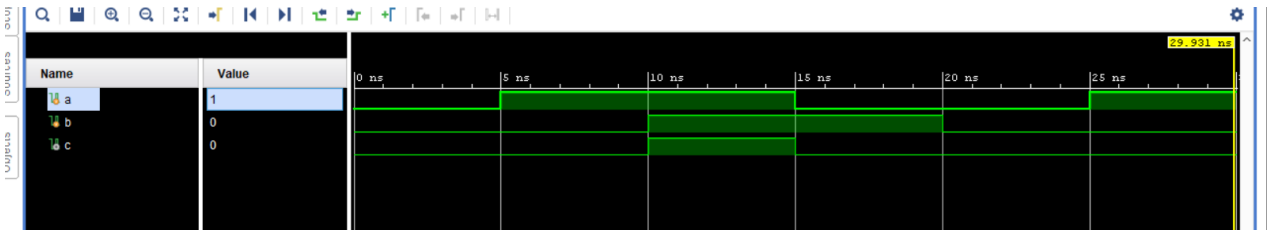
Testbench code:

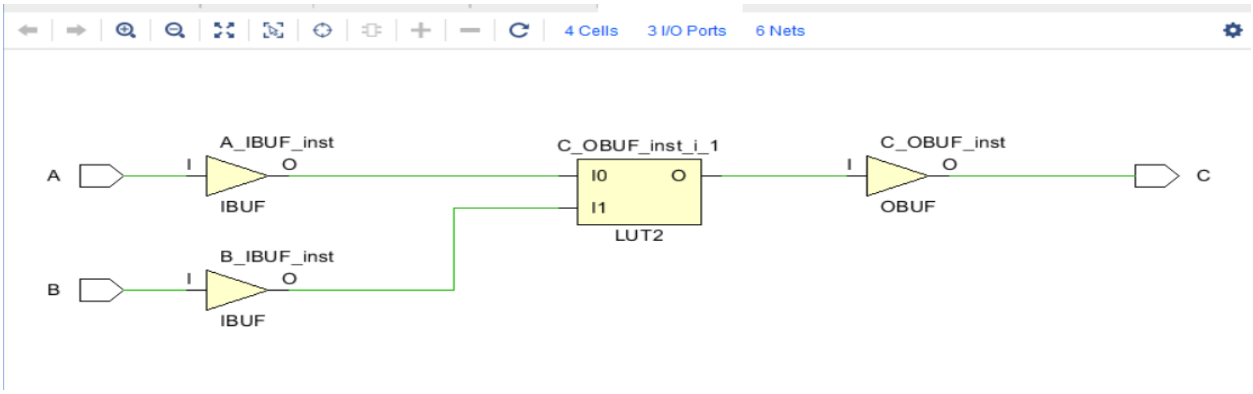
```
module tb_AND_1();
  reg a,b;
  wire c;
  AND_1 dut(c,a,b);
  initial begin
    $dumpfile("AND_1.vcd");
    $dumpvars(1,tb_AND_1);
    #30 $finish();
  end
  initial begin
    a=1'b0;
    b=1'b0;
  end
 always begin
    #5 a=~a;
    #5 b=~b;
  end
  endmodule
```

RTL Schematic



Waveform





Synthesis Schematic

4) NAND GATE

Design Code:

(a) Structural Modeling

```
module NAND_1(
input A,B
output C
);
nand na1(C,A,B);
endmodule
```

(b) Dataflow Modeling

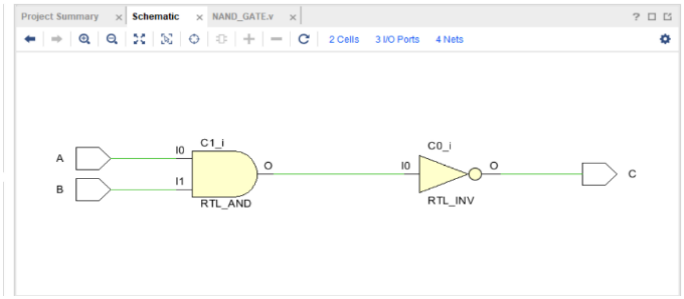
```
module NAND_2(C,A,B);
  input A,B;
  output  C;
 assign C=~(A&B);
endmodule
```

(c) Behavioural Modeling

```
 module NAND_3(C,A,B);
  input A,B;
  output reg C;
);
always @(A,B) begin
      C =~(A&B);
end
endmodule
```

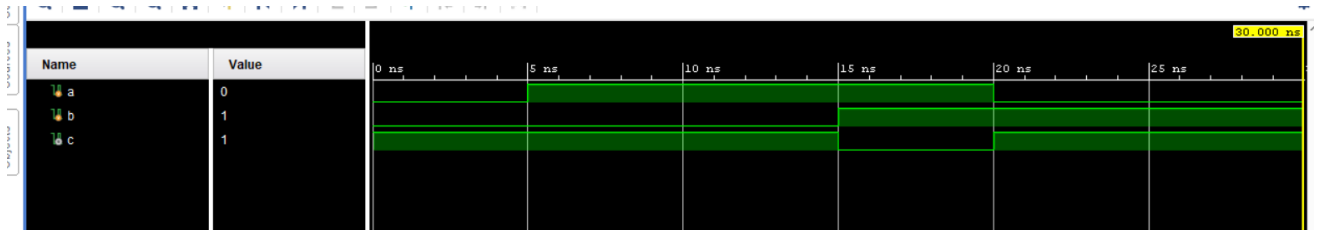
Testbench code:

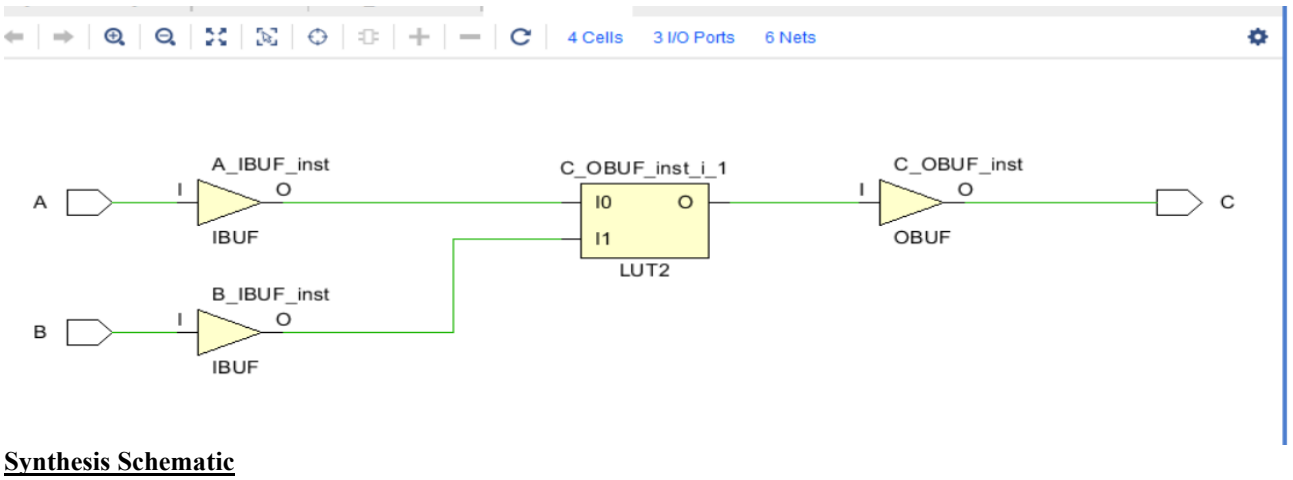
```
module tb_NAND_1();
  reg a,b;
  wire c;
  NAND_1 dut(c,a,b);
  initial begin
    $dumpfile("NAND_1.vcd");
    $dumpvars(1,tb_NAND_1);
    #30 $finish();
  end
  initial begin
    a=1'b0;
    b=1'b0;
  end
 always begin
    #5 a=~a;
    #5 b=~b;
  end
  endmodule
```



Waveform

RTL Schematic





Synthesis Schematic

5) NOR GATE

Design Code:

(a) Structural Modeling

```
module NOR_1(
input A,B
output C
);
nor no1(C,A,B);
endmodule
```

(b) Dataflow Modeling

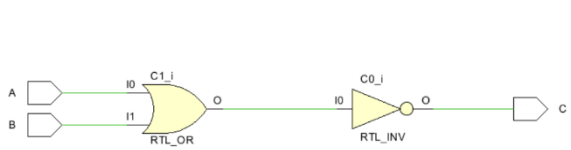
```
module NOR_2(C,A,B);
  input A,B;
  output  C;
 assign C=~(A|B);
endmodule
```

(c) Behavioural Modeling

```
 module NOR_3(C,A,B);
  input A,B;
  output reg C;
);
always @(A,B) begin
      C = ~(A|B);
end
endmodule
```

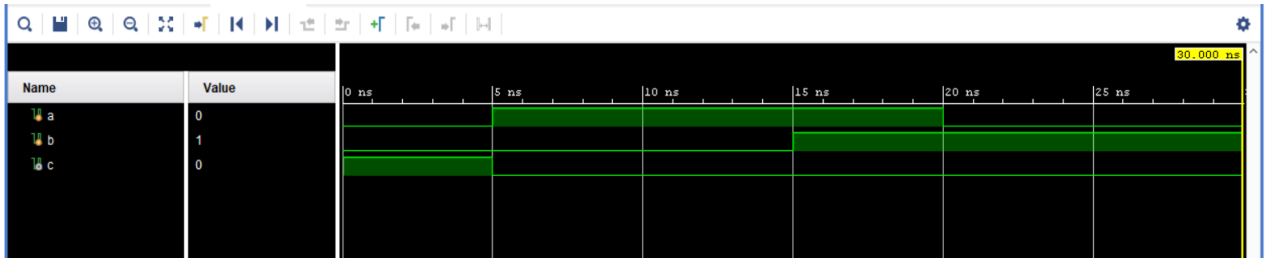
Testbench code:

```
module tb_NOR_1();
 reg a,b;
 wire c;
 NOR_1 dut(c,a,b);
 initial begin
   $dumpfile("NOR_1.vcd");
   $dumpvars(1,tb_NOR_1);
   #30 $finish();
 end
 initial begin
   a=1'b0;
   b=1'b0;
 end
 always begin
   #5 a=~a;
   #5 b=~b;
 end
 endmodule
```

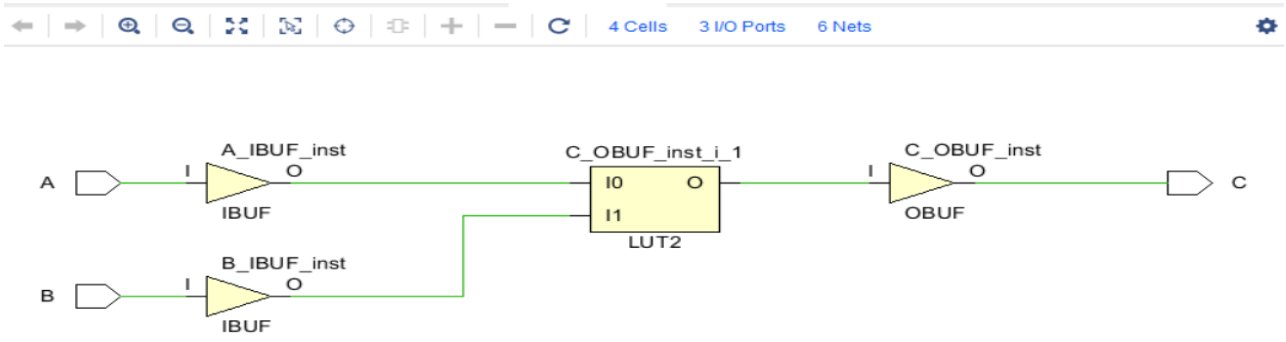


RTL Schematic

Waveform



Synthesis Schematic



6) XOR GATE

Design Code:

(a) Structural Modeling

```
module XOR_1(
input A,B
output C
);
xor xo1(C,A,B);
endmodule
```

(b) Dataflow Modeling

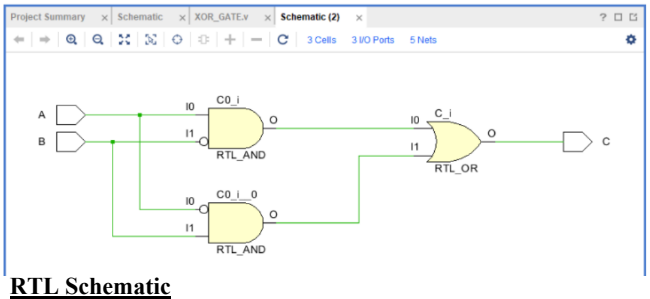
```
module XOR_2(C,A,B);
  input A,B;
  output  C;
 assign C=(((A)&(~B))|((~A)&(B)));
endmodule
```

(c) Behavioural Modeling

```
 module XOR_3(C,A,B);
  input A,B;
  output reg C;
);
always @(A,B) begin
    C=(((A)&(~B))|((~A)&(B)));
end
endmodule
```

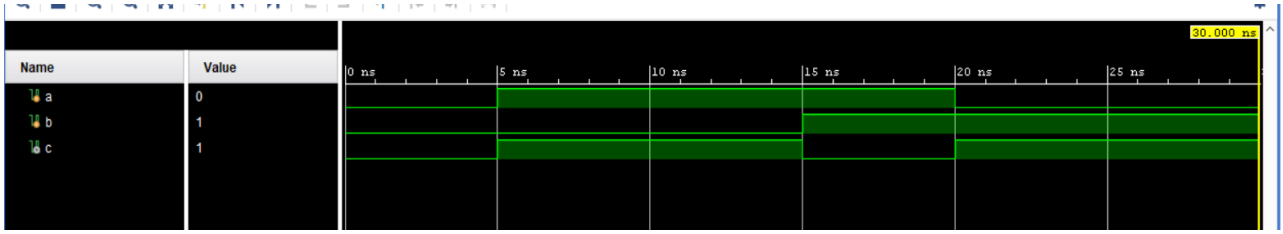
Testbench code:

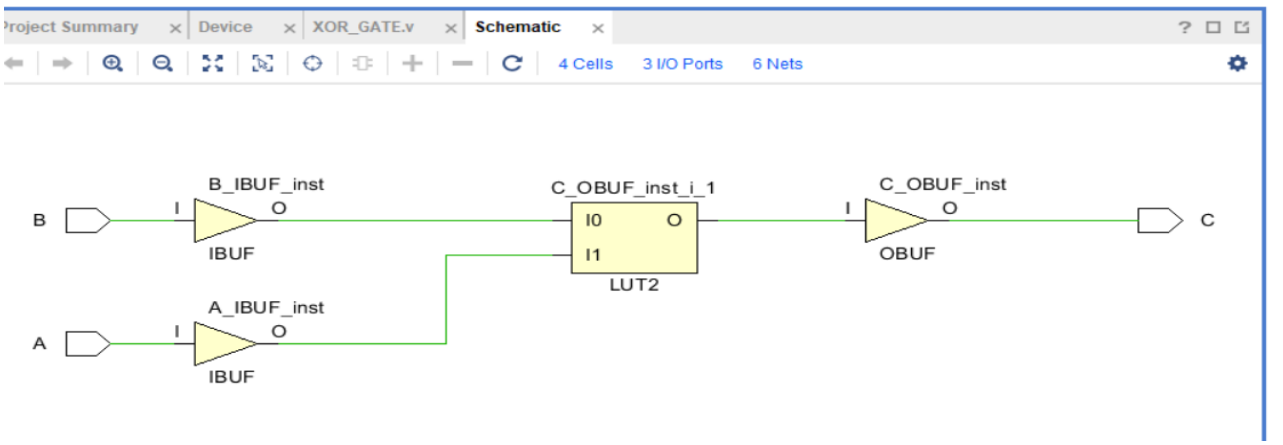
```
module tb_XOR();
  reg a,b;
  wire c;
  XOR_1 dut(c,a,b);
  initial begin
    $dumpfile("XOR_1.vcd");
    $dumpvars(1,tb_XOR);
    #30 $finish();
  end
  initial begin
    a=1'b0;
    b=1'b0;
  end
 always begin
    #5 a=~a;
    #5 b=~b;
  end
  endmodule
```



RTL Schematic

Waveform





Synthesis Schematic

7) XNOR GATE

Design Code:

(a) Structural Modeling

```
module XNOR_1(
input A,B
output C
);
xnor xno1(C,A,B);
endmodule
```

(b) Dataflow Modeling

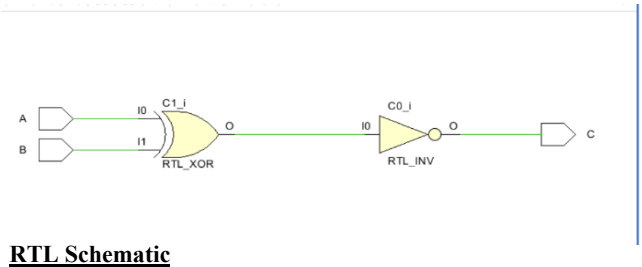
```
module XNOR_2(C,A,B);
  input A,B;
  output  C;
  assign C = ~(A^B);
endmodule
```

(c) Behavioural Modeling

```
 module XNOR_3(C,A,B);
  input A,B;
  output reg C;
);
always @(A,B) begin
    C = ~(A^B);
end
endmodule
```

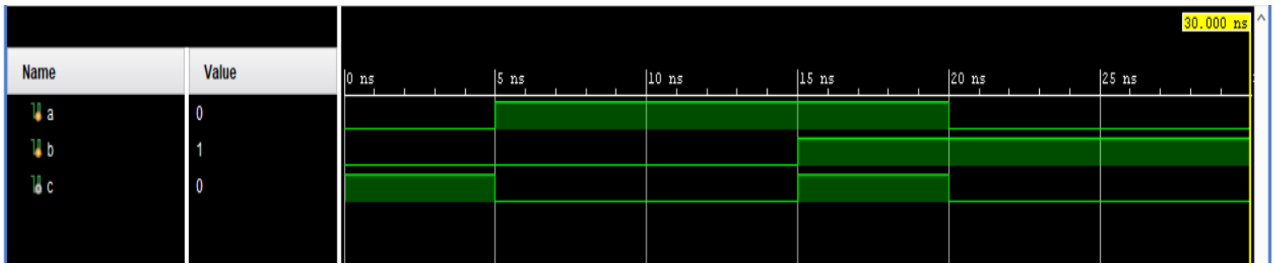
Testbench code:

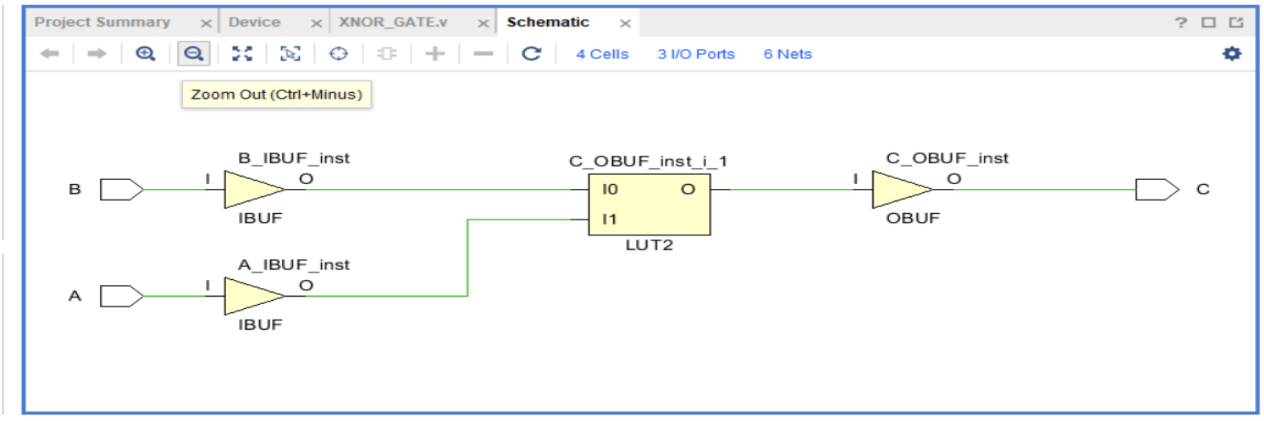
```
module tb_XNOR();
 reg a,b;
 wire c;
 XNOR_1 dut(c,a,b);
 initial begin
   $dumpfile("XNOR_1.vcd");
   $dumpvars(1,tb_XNOR);
   #30 $finish();
 end
 initial begin
   a=1'b0;
   b=1'b0;
 end
 always begin
   #5 a=~a;
   #5 b=~b;
 end
 endmodule
```



RTL Schematic

Waveform





Synthesis Schematic

Part – 2 Boolean Function Implementation

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

FUNCTION 1:

Design Code:

(a) Structural Modeling

```
module part2_1(input x, input y, output f);
wire a,b,c,d;
NOT_GATE no1(a,x);
NOT_GATE no2(b,y);
 AND_GATE an1(a,b,c);
AND_GATE an2(x,y,d);
OR_GATE o1(f,c,d);
endmodule
```

(b) Dataflow Modeling

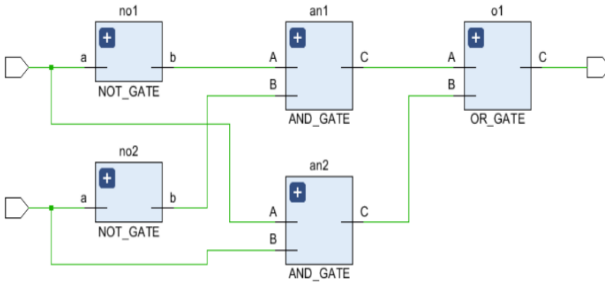
```
module part2_1(input x, input y, output f);
  assign f=(~x&~y)|(x&y);
endmodule
```

(c) Behavioural Modeling

```
module part2_1(input x, input y, output f);
wire a,b,c,d;
always @(x,y) begin
f=(~x&~y)|(x&y);
end
endmodule
```

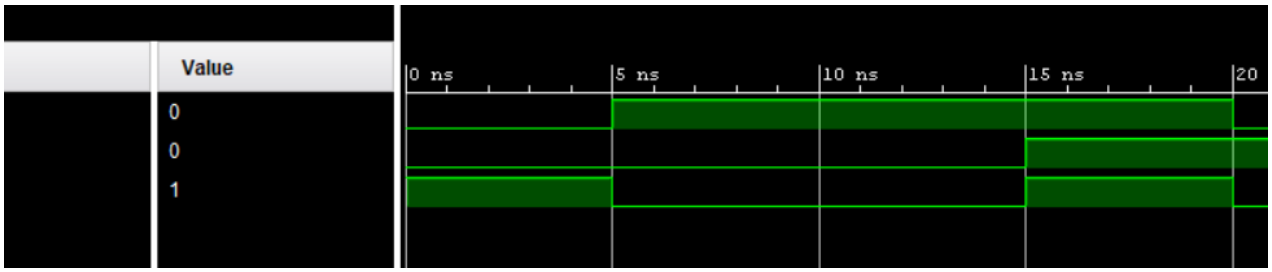
Testbench code:

```
module tb_part2_1(
 );
 reg x,y;
 wire f;
 part2_1 dut(x,y,f);
 initial begin;
  $dumpfile("part2_1.vcd");
  $dumpvars(1,tb_part2_1);
  x=1'b 0;
  y=1'b 0;
  #35 $finish();
 end
 always begin;
  #5 x=~x;
  #10 y=~y;
 end
```

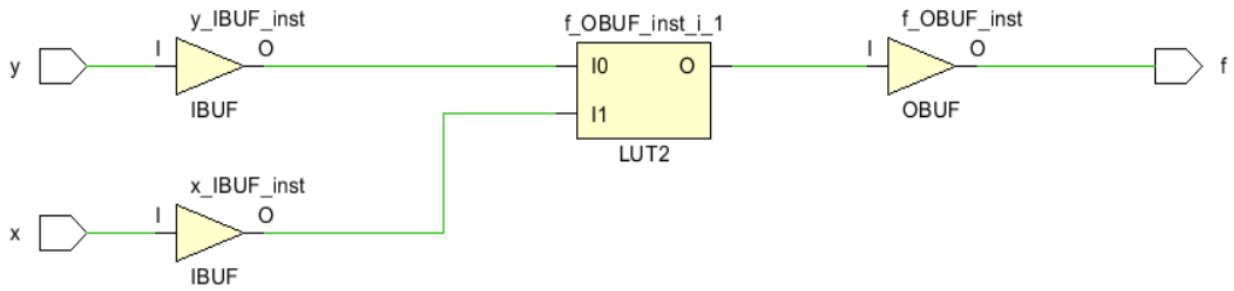


RTL Schematic

Waveform



Synthesis Schematic



FUNCTION 2:

Design Code:

(a) Structural Modeling

```
module part2_2(x,y,z,g);
input x,y,z;
output reg g;
wire a,b,c,d,e;
NOT_GATE n1(x,a);
NOT_GATE n2(y,b);
NOT_GATE n3(z,c);
AND_GATE a1(a,y,z,d);
AND_GATE a2(x,y,c,e);
AND_GATE a3(a,b,c,f);
AND_GATE a4(x,y,z,h);
OR_GATE o1(d,e,f,h,g);
endmodule
```

(b) Dataflow Modeling

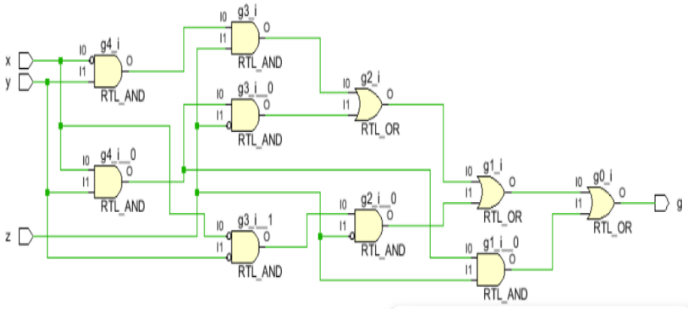
```
module part2_2(input x, input y, input z, output g);
assign g=(~x&y&z)|(x&y
&~z)|(~x&~y&~z)|
(x&y&z);
endmodule
```

(c) Behavioural Modeling

```
module part2_2(input x, input y,input z,
output reg g);
always @(x,y,z) begin
g=(~x&y&z)|(x&y&~z)
|(~x&~y&~z)|(x&y&z);
end
endmodule
```

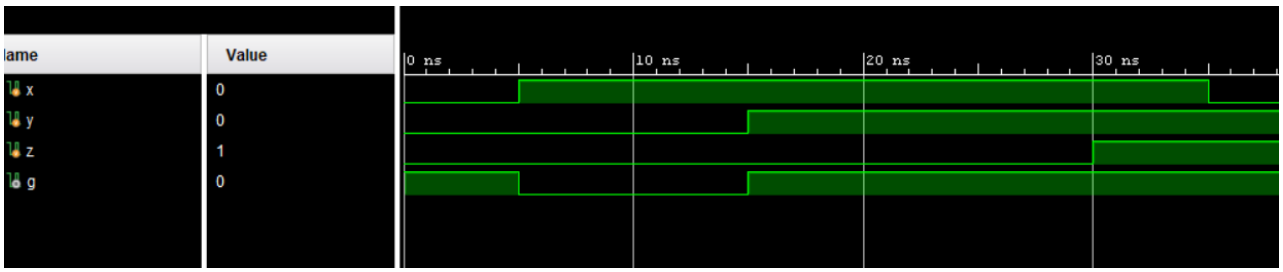
Testbench code:

```
module tb_part2_2(
  );
  reg x,y;
  wire f;
  part2_2 dut(x,y,f);
  initial begin;
    $dumpfile("part2_2.vcd");
    $dumpvars(1,tb_part2_2);
    x=1'b 0;
    y=1'b 0;
    #35 $finish();
  end
  always begin;
    #5 x=~x;
    #10 y=~y;
  end
endmodule
```

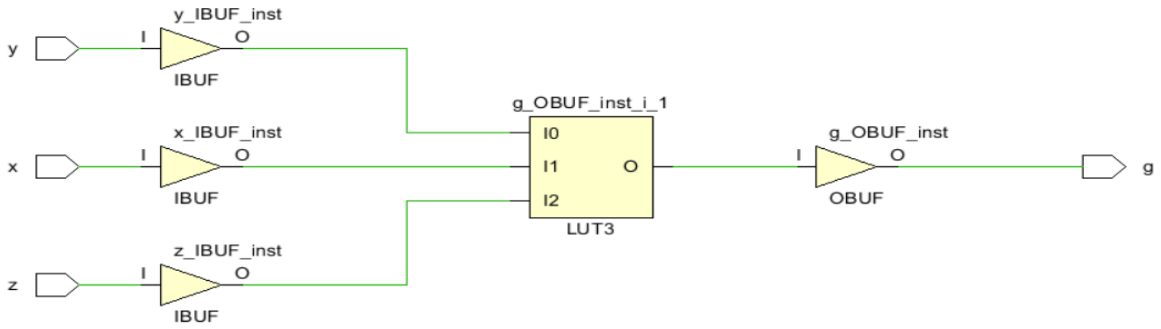


RTL Schematic

Waveform



Synthesis Schematic



Whereas modules used here are:

```
module NOT_1(a,b);
input a;
output b;
assign b=~a;
endmodule

module AND_1(A,B,C,D);
input A,B,C;
output D;
assign D=A&B&C;
endmodule

module OR_1(A,B,C,D);
input A,B,C;
output D;
assign D=A|B|C;
endmodule

module AND_1(A,B,C);
input A,B;
output reg C;
assign C=A&B;
endmodule

module OR_1(A,B,C);
input A,B;
output C;
assign C=A|B;
endmodule
```

Conclusion:

All three modeling styles (Structural, Dataflow, and Behavioral) were implemented and simulated successfully in Xilinx. The simulation results verified that all basic logic gates and Boolean functions worked correctly according to their truth tables. While structural modeling shows actual gate connections, dataflow is concise for combinational logic, and behavioral modeling is more flexible for complex/sequential designs. Thus, all methods produce the same logical outputs but differ in coding style and abstraction level.

Suggested Reference: ---

References used by the students:

Rubric wise marks obtained:

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

Experiment No: 5

Date: _____

Aim: Design Adder & Subtractor using Verilog / VHDL.

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

Competency and Practical Skills: Basic Digital Design

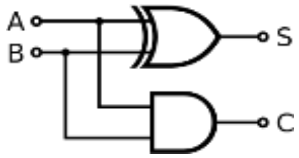
Relevant CO: CO5

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

Equipment / Instruments: Laptop or Computer with Xilinx Vivado.

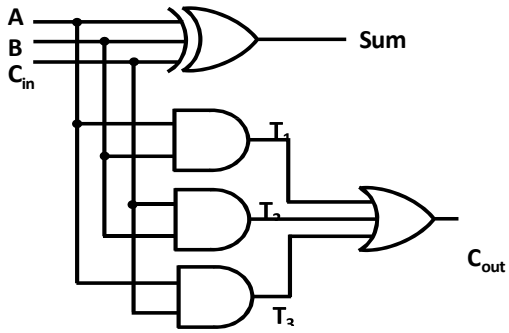
Basic Theory:

Half Adder



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

Full Adder



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

Half Subtractor

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

Full Subtractor

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

a) Half Adder (Using structural and dataflow modeling)

Design Code:

(a) Structural Modeling

```
module Halfadder(A,B,Sum,Carry);
    input A,B;
    output Sum, Carry;
    XOR_GATE x1(A,B,Sum);
    AND_GATE a1(A,B,Carry);
endmodule
```

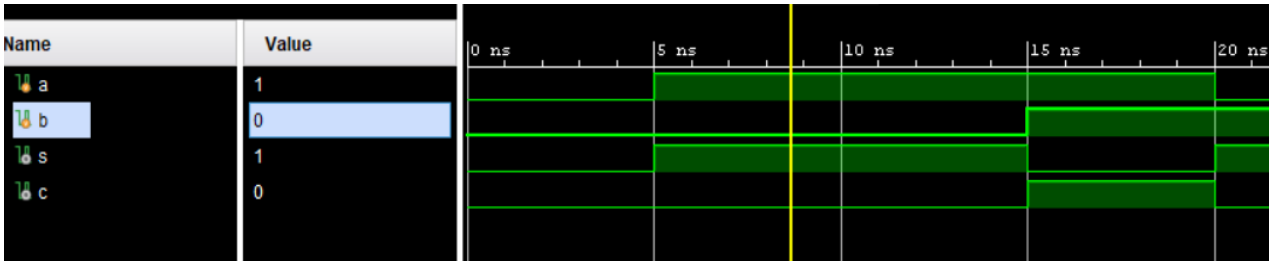
(b) Dataflow Modeling

```
module Halfadder_1(A,B,Sum,Carry);
   input A,B;
    output Sum, Carry;
      assign Sum=A^B;
      assign Carry=A&B;
endmodule
```

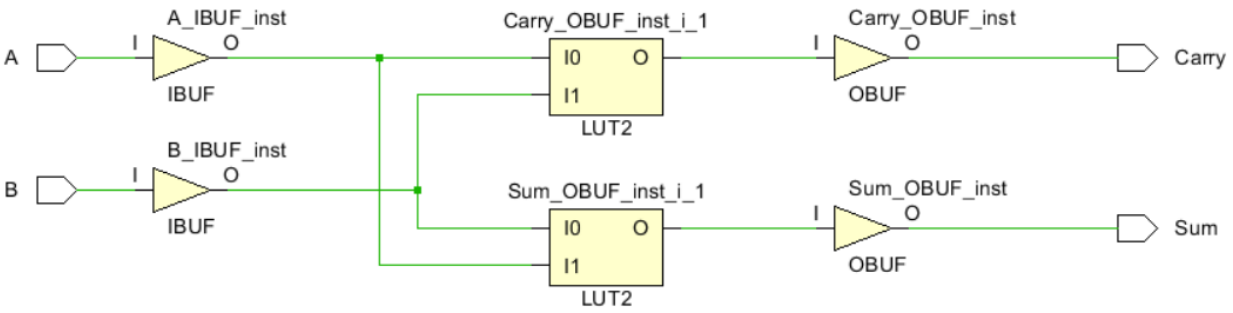
Testbench code:

```
module tb_Halfadder();
 reg a,b;
 wire s,c;
 Halfadder dut(a,b,s,c);
 initial begin
  $dumpfile("Halfadder.vcd");
  $dumpvars(1,tb_Halfadder);
  a=1'b0;
  b=1'b0;
  #35 $finish();
 end
 always begin
  #5 a=~a;
  #10 b=~b;
 end
endmodule
```

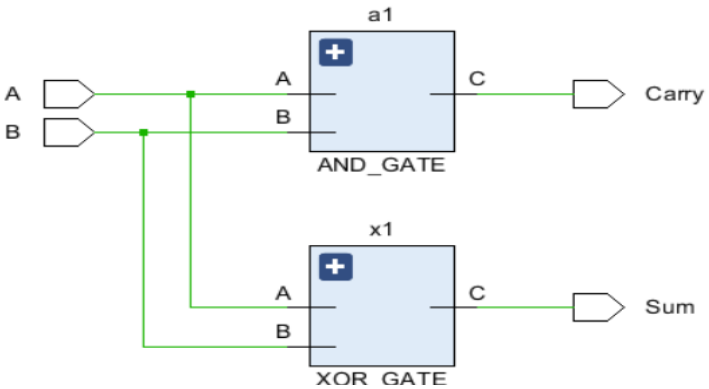
Waveform



Synthesis Schematic



RTL Schematic



b) Full Adder using Half Adders (Structural Modelling)

Design Code:

(a) Structural Modeling

```
module Fulladder(
    input A,
    input B,
    input Cin,
    output Sum,
    output Cout
    );
     wire S1,C1,C2;
        HA ha1(A,B,S1,C1);
        HA ha2(S1,Cin,Sum,C2);
        assign Cout= C1|C2;
    endmodule
```

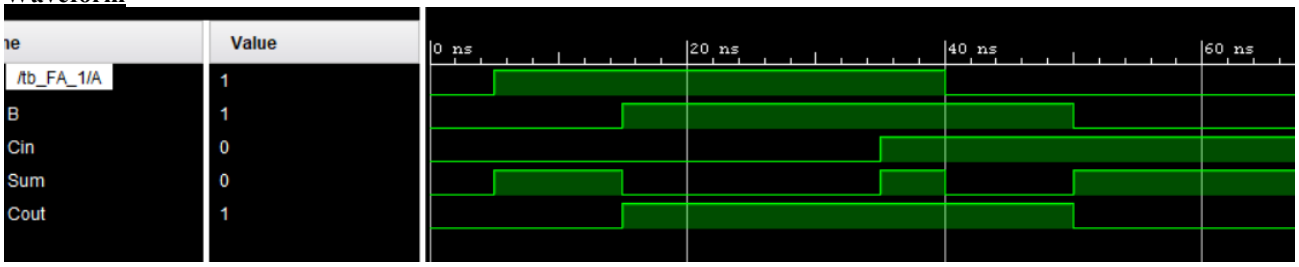
Half Adder Module as follows:

```
module Halfadder(A,B,Sum,Carry);
 input A,B;
 output Sum,Carry;
    assign Sum=A^B;
    assign Carry=A&B;
endmodule
```

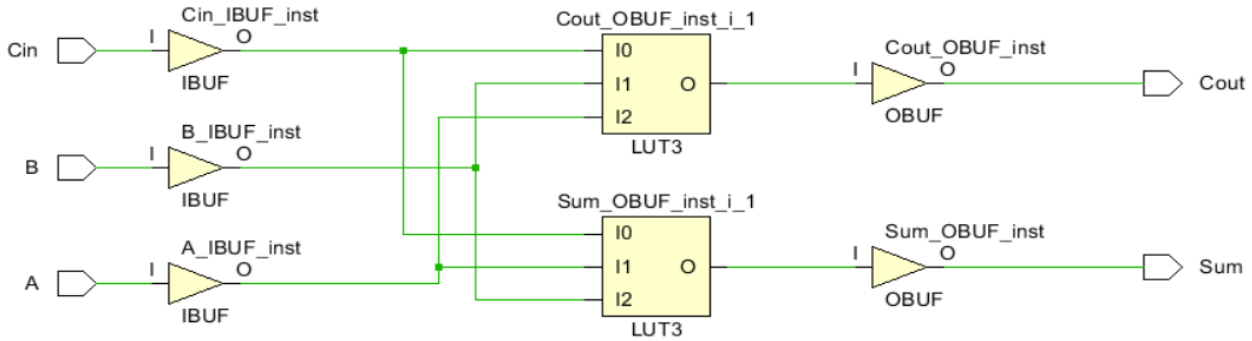
Testbench code:

```
module tb_Fulladder();
  reg A,B,Cin;
  wire Sum,Cout;
  FA_1 dut(A,B,Cin,Sum,Cout);
  initial begin
    A=0; B=0; Cin=0;
    $dumpfile("Fulladder.vcd");
    $dumpvars(1,tb_Fulladder);
    $display("time | A B Cin | Sum Cout");
    $monitor("%4t | %b  %b  %b |  %b  %b",
$time, A, B, Cin, Sum, Cout);
    #100 $finish;
    end
          always begin
                #5 A=~A;
             #10 B=~B;
             #20 Cin=~Cin;
             end
endmodule
```

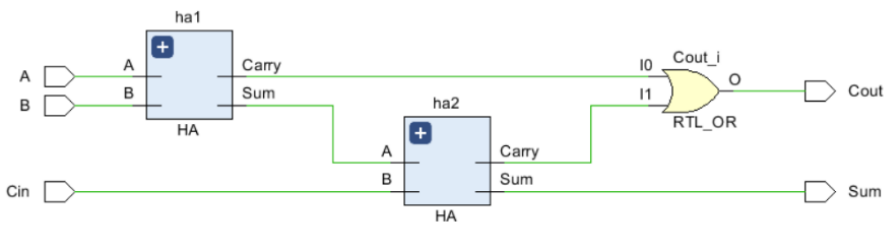
Waveform



Synthesis Schematic



RTL Schematic



c) Full Adder (Dataflow and Behavioural Modelling)

Design Code:

Dataflow Modeling

```
module Fulladder_1(A,B,Cin,S,Cout);
  input A,B,Cin;
  output reg S,Cout;
 assign S=A^B^Cin;
 assign Cout = ((A&B)|(B&Cin)|(A&Cin))
endmodule
```

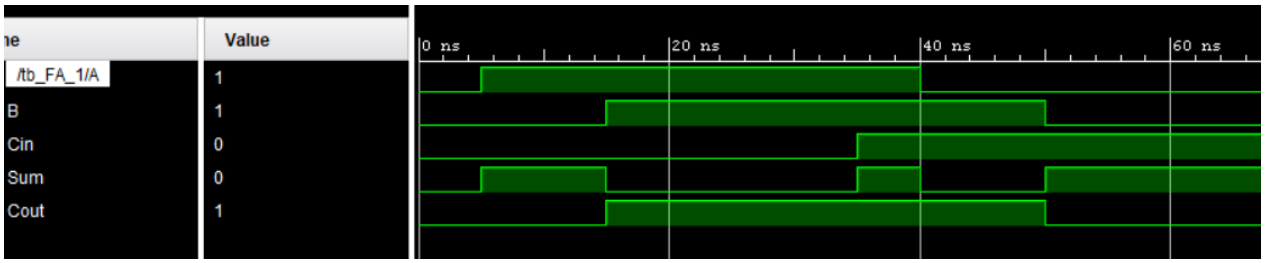
Behavioural Modelling

```
module Fulladder_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
```

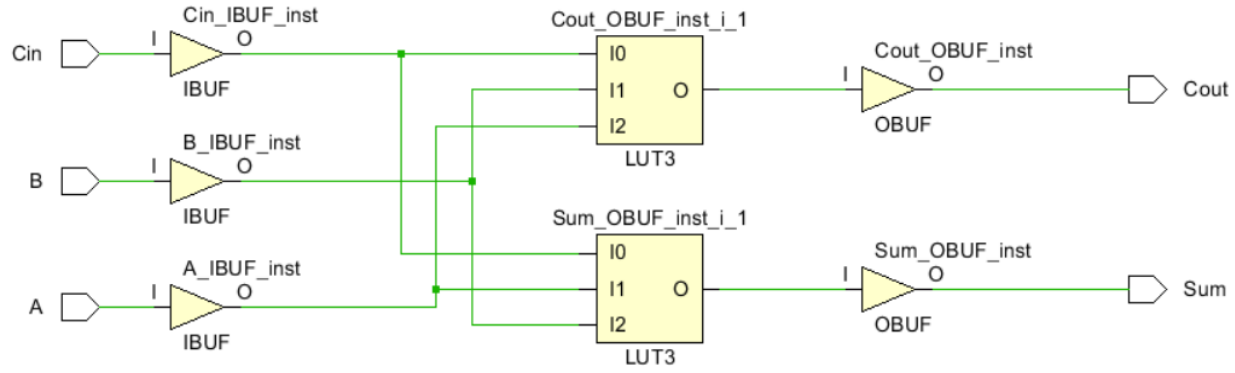
Testbench code:

```
module tb_Fulladder_1();
 reg a,b,cin;
 wire s, cout;
 Fulladder_1 dut(a,b,cin,s,cout);
 initial begin
   a = 1'b0;
   b = 1'b0;
   cin = 1'b0;
   $dumpfile("Fulladder_1.vcd");
   $dumpvars(1,tb_Fulladder_1);
   #100 $finish();
 end
 always begin
   #5 a = ~a;
   #10 b = ~b;
   #15 cin = ~cin;
 end
endmodule
```

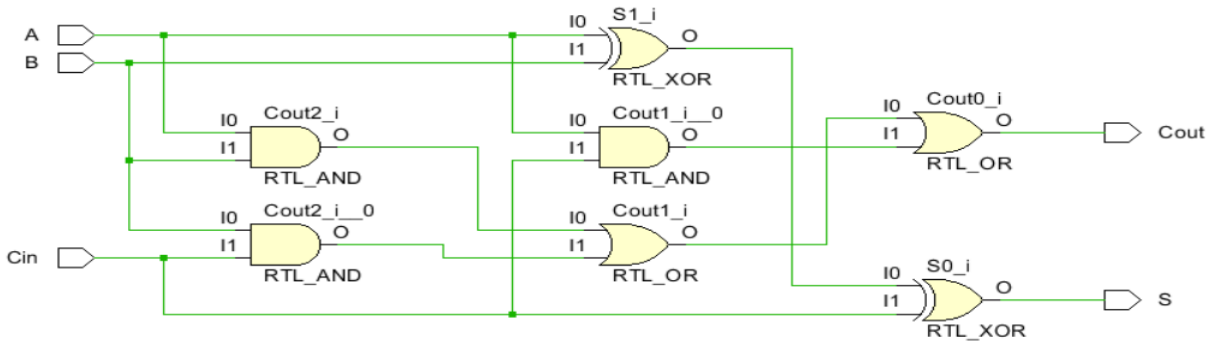
Waveform



Synthesis Schematic



RTL Schematic

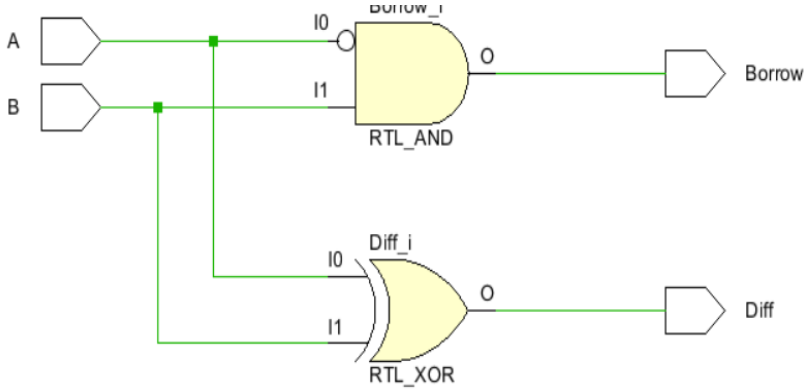


d) Half Subtractor

Dataflow Modelling:

```
module HS(
    input A,
    input B,
    output Diff,
    output Borrow
    );
    assign Diff=A^B;
    assign Borrow=((~A)&B);
endmodule
```

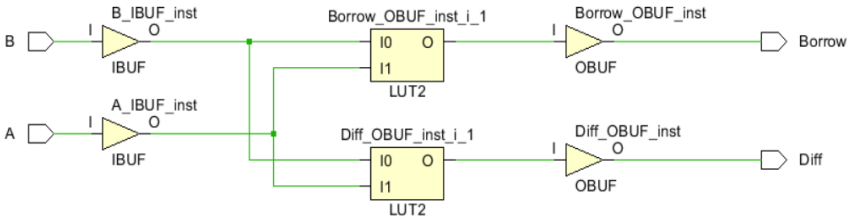
RTL Schematic



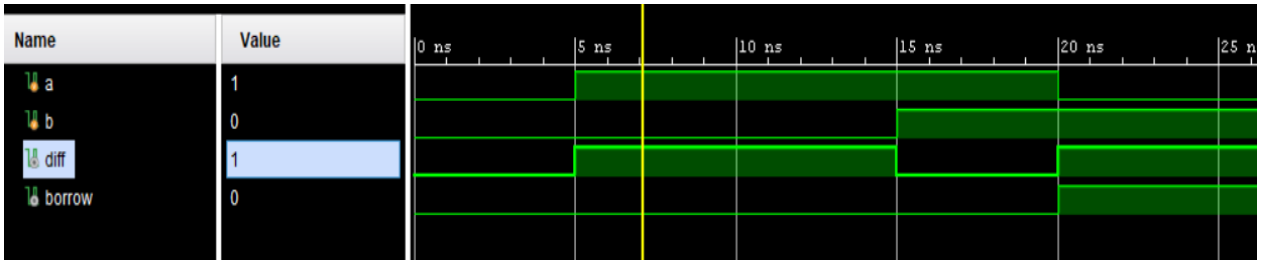
Testbench:

```
module tb_HS();
  reg a,b;
  wire diff,borrow;
  HS dut(a,b,diff,borrow);

  initial begin;
    $dumpfile("HS.vcd");
    $dumpvars(1,tb_HS);
    a=1'b 0;
    b=1'b 0;
    #35 $finish();
  end

  always begin;
    #5 a=~a;
    #10 b=~b;
  end
 endmodule
```

Synthesis Schematic



Waveform



e) Full Subtractor

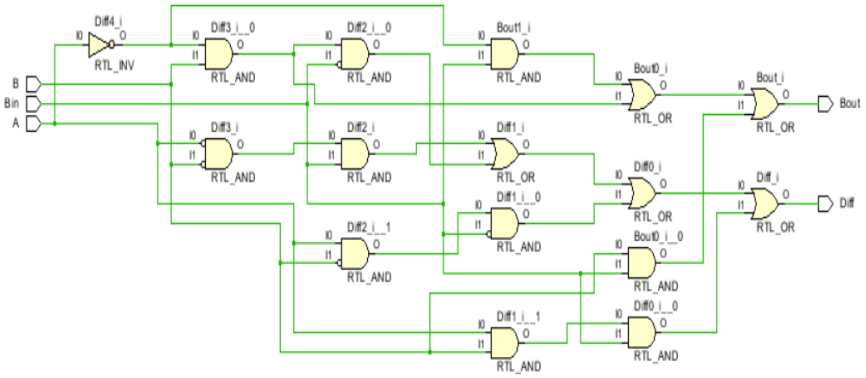
Dataflow Modelling:

```
module HS(
input A,
    input B,
    input Bin,
    output Diff,
    output Bout
    );
      assign Boutt= (((~A)&(Bin))|((~A)&(B))|((B)&(Bin)));
      assign Diff=(((~A)&(~B)&(Bin))|((~A)&(B)&(~Bin))|((A)&(~B)&(~Bin))|((A)&(B)&(Bin)));
endmodule
```

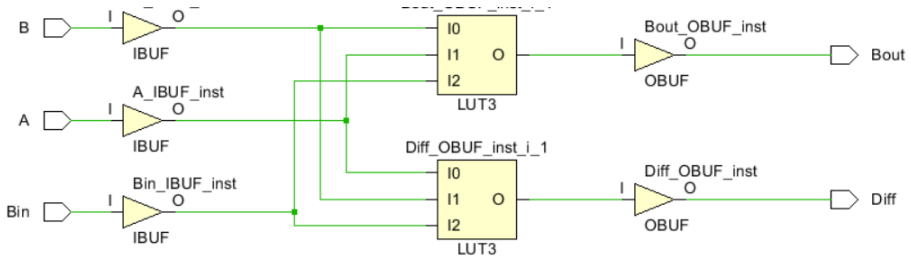
Testbench:

```
module tb_FS();
     reg a,b,bin;
     wire diff,bout;
     FS dut(a,b,bin,diff,bout);
    initial begin;
       $dumpfile("FS.vcd");
       $dumpvars(1,tb_FS);
       a=1'b 0;
       b=1'b 0;
       bin=1'b0;
       #100 $finish();
     end
     always begin;
       #5 a=~a;
       #10 b=~b;
       #15 bin=~bin;
     end
    endmodule
```

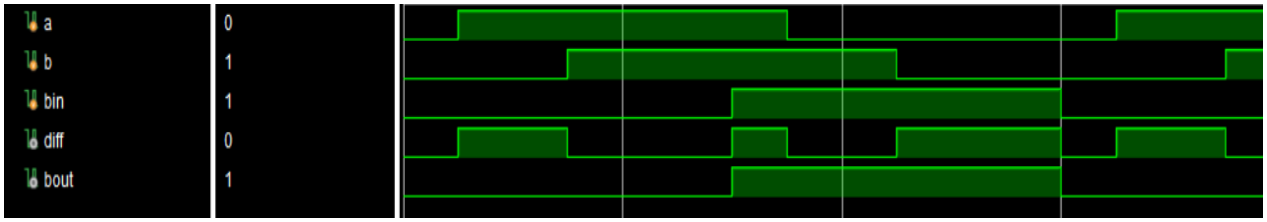
RTL Schematic



Synthesis Schematic



Waveform



Conclusion:

The Half Adder, Full Adder, Half Subtractor, and Full Subtractor were successfully designed and simulated in Verilog/VHDL using Xilinx Vivado. Different modeling styles (structural, dataflow, behavioral) were applied, and the simulation results matched the expected truth tables. The practical enhanced understanding of arithmetic circuits, hierarchical design, and HDL modeling methods.

Suggested Reference:

References used by the students:

Rubric wise marks obtained:

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

Experiment No: 6

Date: _____

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

Competency and Practical Skills: Basic Digital Design

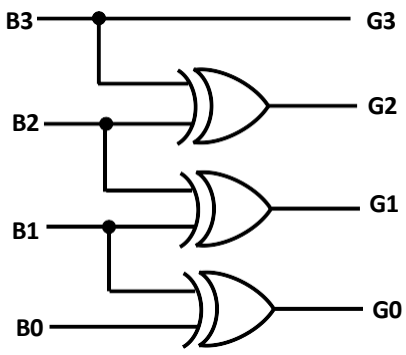
Relevant CO: CO5

Objectives: Designing of code converters

Equipment / Instruments: Laptop or Computer with Xilinx Vivado.

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
	

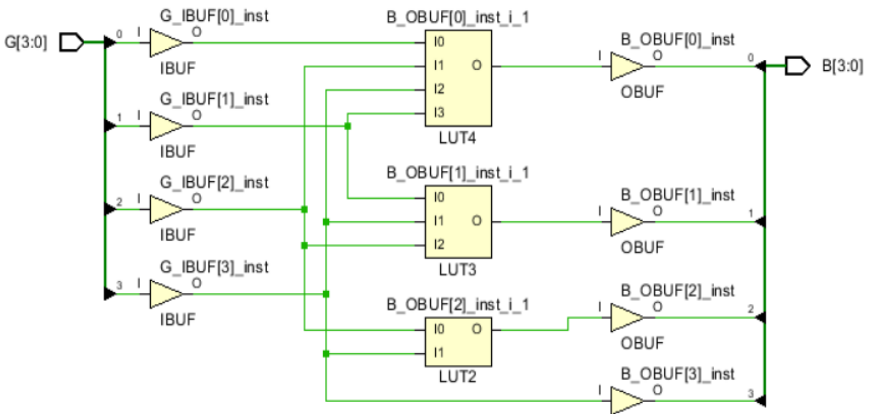
(1)Gray to Binary Encoder

Design Code:

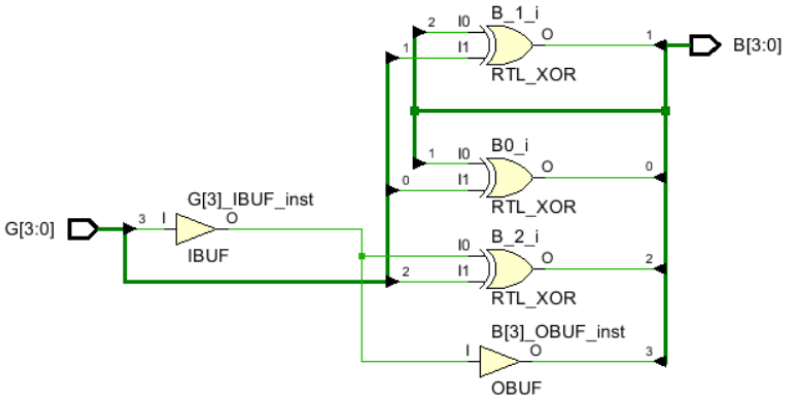
```
module gray_to_bin(G,B);
    input [3:0] G;
    output [3:0] B;
    wire [3:0] B;
    assign B[3] = G[3];
    assign B[2] = B[3] ^ G[2];
    assign B[1] = B[2] ^ G[1];
    assign B[0] = B[1] ^ G[0];
  endmodule
```

Testbench:

```
module tb_HS();
  reg a,b;
  wire diff,borrow;

  HS dut(a,b,diff,borrow);

  initial begin;
    $dumpfile("HS.vcd");
    $dumpvars(1,tb_HS);
    a=1'b 0;
    b=1'b 0;
    #35 $finish();
  end

  always begin;
    #5 a=~a;
    #10 b=~b;
  end
 endmodule
```

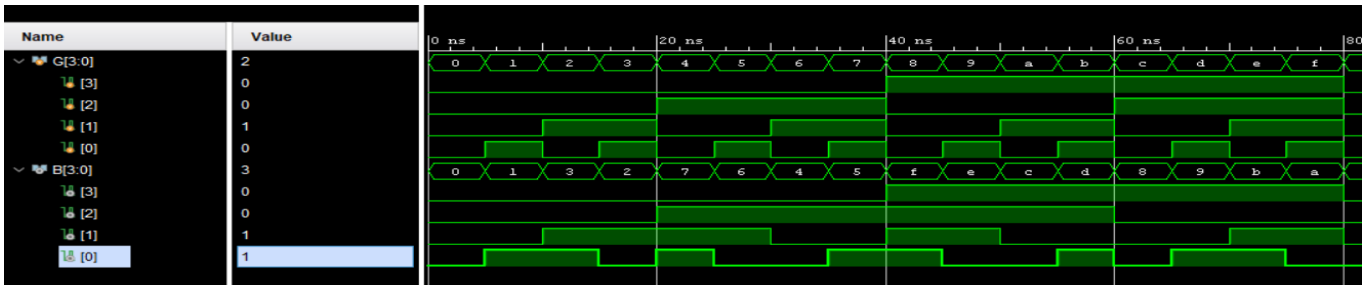


RTL Schematic



Synthesis Schematic

Waveform



(2)Binary to Gray Converter

Design Code:

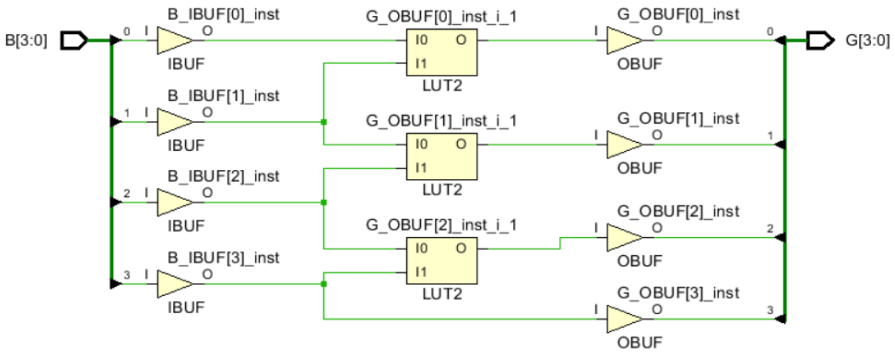
```
module bin_to_gray(B,G);
input [3:0] B;
output [3:0] G;
assign G[3] = B[3];
assign G[2] = B[3] ^ B[2];
assign G[1] = B[2] ^ B[1];
assign G[0] = B[1] ^ B[0];
endmodule
```

Testbench:

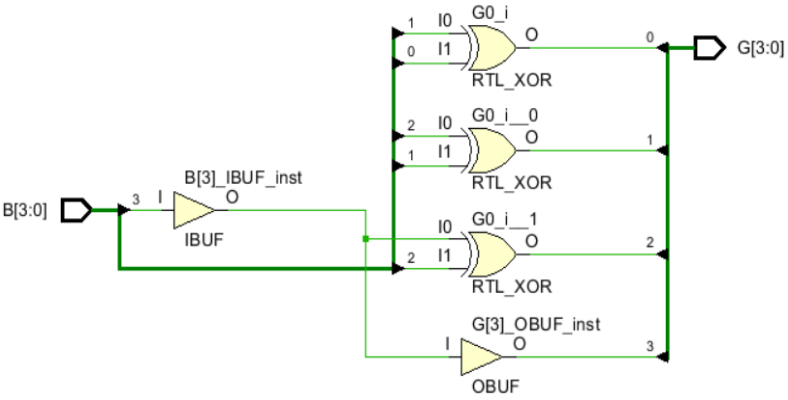
```
module tb_bin_to_gray();
  reg [3:0]B;
  wire [3:0]G;
  bin_g bg1(B,G);
  initial begin
    $dumpfile("bin_to_gray.vcd");
    $dumpvars(1,tb_bin_to_gray);
    B=4'b 0000;
    #100 $finish();
  end
  always begin
    #5 B=B+1;
  end
endmodule
```

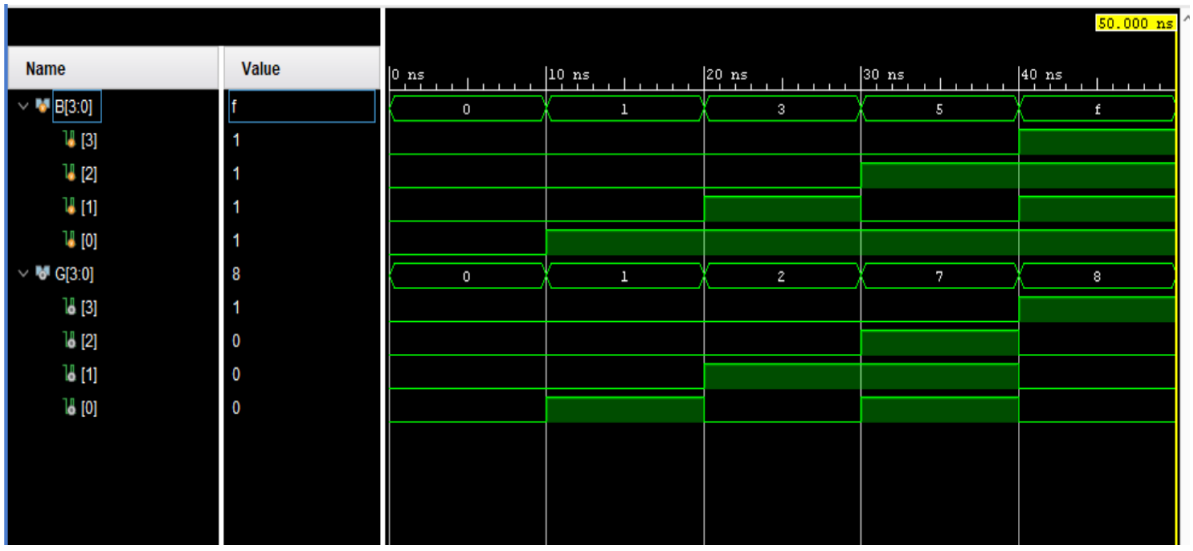
Waveform

RTL Schematic



Synthesis Schematic





Conclusion:

Binary-to-Gray and Gray-to-Binary converters were successfully designed and simulated in Verilog/VHDL using Xilinx Vivado. The simulation outputs matched the expected truth tables, confirming correct code conversion. This practical helped understand number system conversion and HDL implementation techniques.

Suggested Reference:

References used by the students:

Rubric wise marks obtained:

Rubrics	1	2	3	4	5	Total
Marks						

Experiment No: 7

Date: _____

Aim: Design Multiplexer and Demultiplexer using Verilog / VHDL.

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

Competency and Practical Skills: Basic Digital Design

Relevant CO: CO5

Objectives: Designing of Multiplexers and Demultiplexers

Equipment / Instruments: Laptop or Computer with Xilinx Vivado.

Basic Theory:

MUX:

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

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

Design Code:

(a) Dataflow Modeling

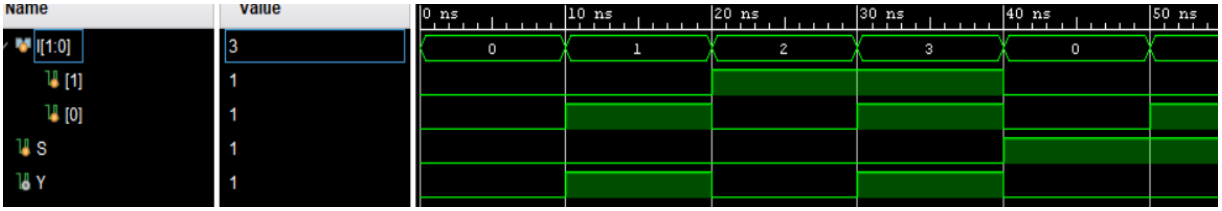
```
module mux2_1(
   input [1:0] I,
   input S,
   output Y
   );
   assign Y = (S) ? I[1] : I[0];
endmodule
```

(b) Behavioural Modeling

```
module mux2_1(
   input [1:0] I,
   input S,
   output reg Y
   );
always @(*) begin
       if (S == 0)
         Y = I[0];
       else
         Y = I[1];
     end
endmodule
```

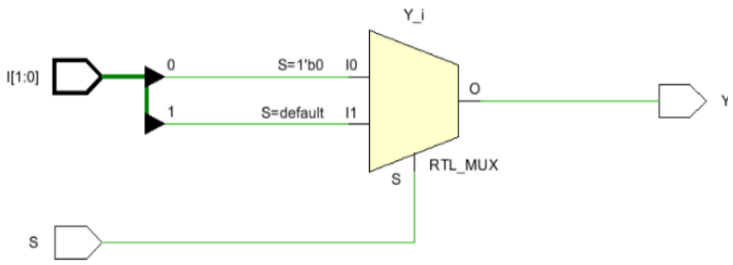
Testbench code:

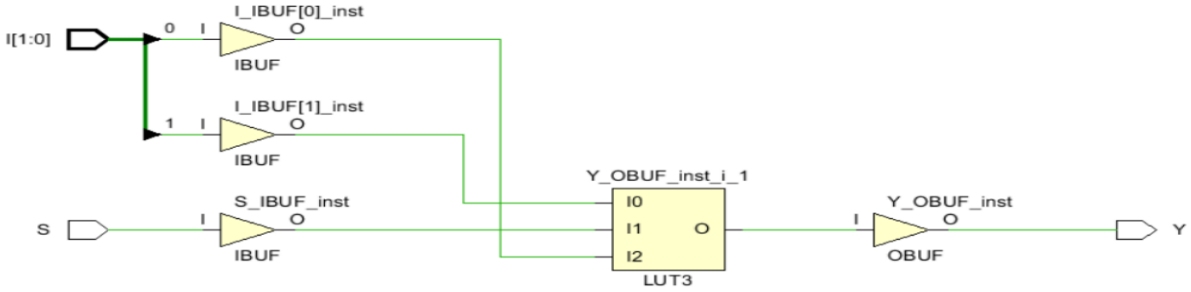
```
module tb_mux2_1;
    reg [1:0] I;
    reg S;
    wire Y;
    mux2_1 dut(I, S, Y);
    initial begin
     $dumpfile("mux2_1.vcd");
     $dumpvars(1, tb_mux2_1);
     // Apply test cases
     I = 2'b00; S = 0; #10;
     I = 2'b01; S = 0; #10;
     I = 2'b10; S = 0; #10;
     I = 2'b11; S = 0; #10;
     I = 2'b00; S = 1; #10;
     I = 2'b01; S = 1; #10;
     I = 2'b10; S = 1; #10;
     I = 2'b11; S = 1; #10;
     $finish;
    end
endmodule
```



Waveform

Synthesis Schematic





RTL Schematic

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

Design Code:

(a) Dataflow Modeling

```
module mux4_1(I,S,Y);
input [3:0] I;
input [1:0] S;
output Y;
assign Y = (S == 2'b00) ? I[0] :
           (S == 2'b01) ? I[1] :
           (S == 2'b10) ? I[2] :
                          I[3];
Endmodule
```

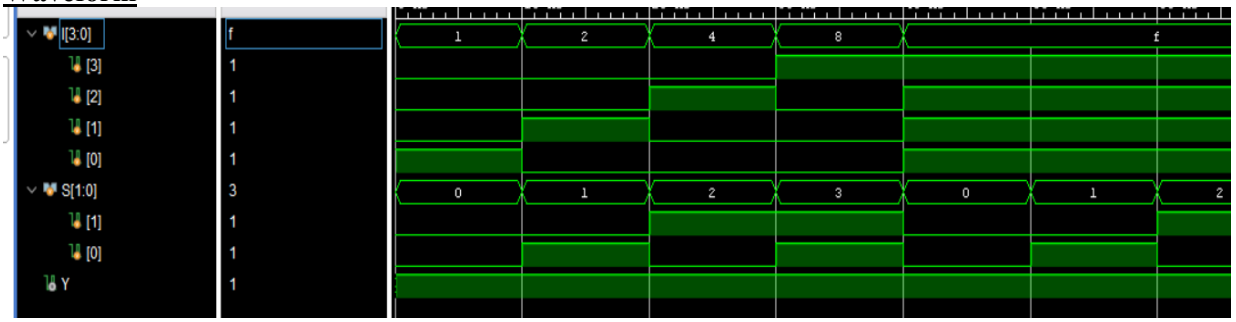
(b) Structural Modeling

```
module mux4_1(I,S,Y);
input [3:0] I;
input [1:0] S;
output Y;
wire w1,w2;
mux2_1 m1(.I({I[1],I[0]}),.S(S[0]),.Y(w1));
mux2_1 m2(.I({I[3],I[2]}),.S(S[0]),.Y(w2));
mux2_1 m3(.I({w2,w1}),.S(S[1]),.Y(Y));
endmodule
```

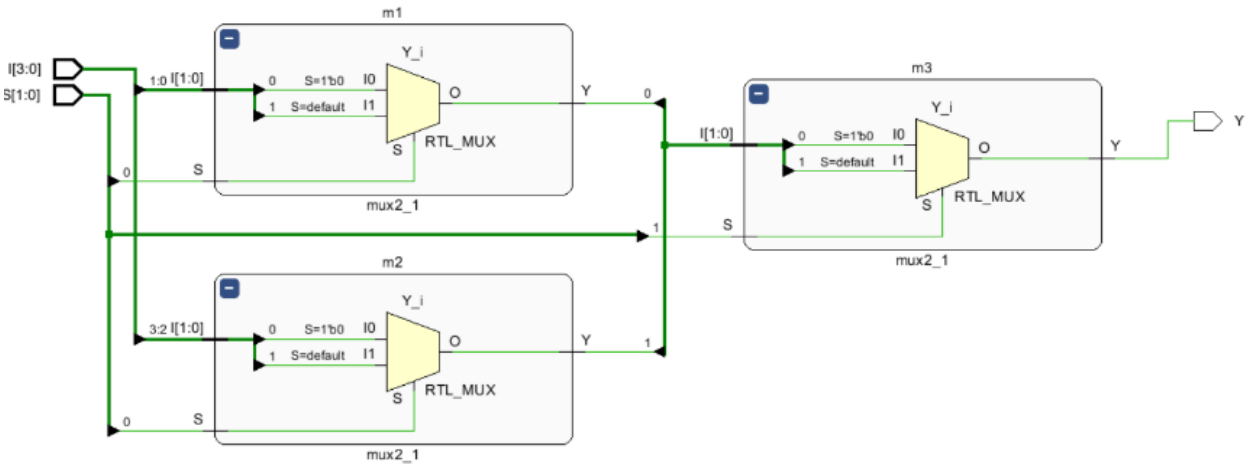
Testbench code:

```
module tb_mux4_1;
  reg [3:0] I;
  reg [1:0] S;
  wire Y;
 mux4_1 dut(I, S, Y);
  initial begin
    $dumpfile("mux4_1.vcd");
    $dumpvars(1, tb_mux4_1);
    // Apply test cases
    I = 4'b0001; S = 2'b00; #10;
    I = 4'b0010; S = 2'b01; #10;
    I = 4'b0100; S = 2'b10; #10;
    I = 4'b1000; S = 2'b11; #10;
    I = 4'b1111; S = 2'b00; #10;
    I = 4'b1111; S = 2'b01; #10;
    I = 4'b1111; S = 2'b10; #10;
    I = 4'b1111; S = 2'b11; #10;
    $finish;
  end
endmodule
```

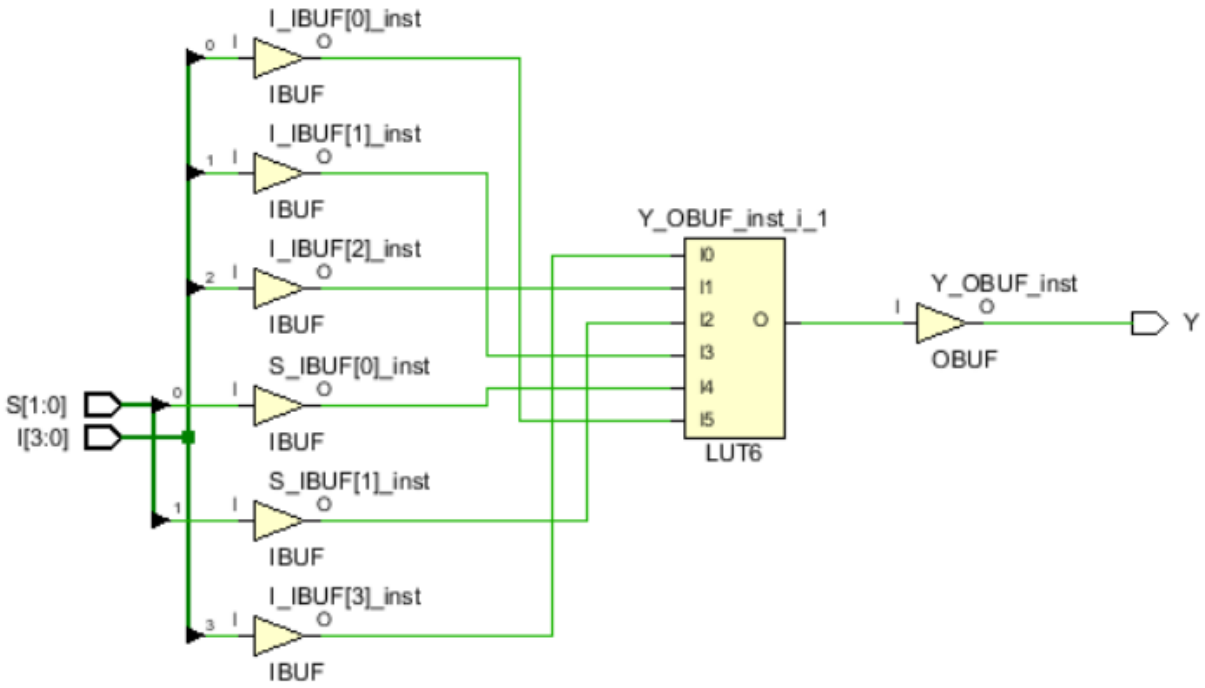
Waveform



RTL Schematic



Synthesis Schematic



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

Design Code:

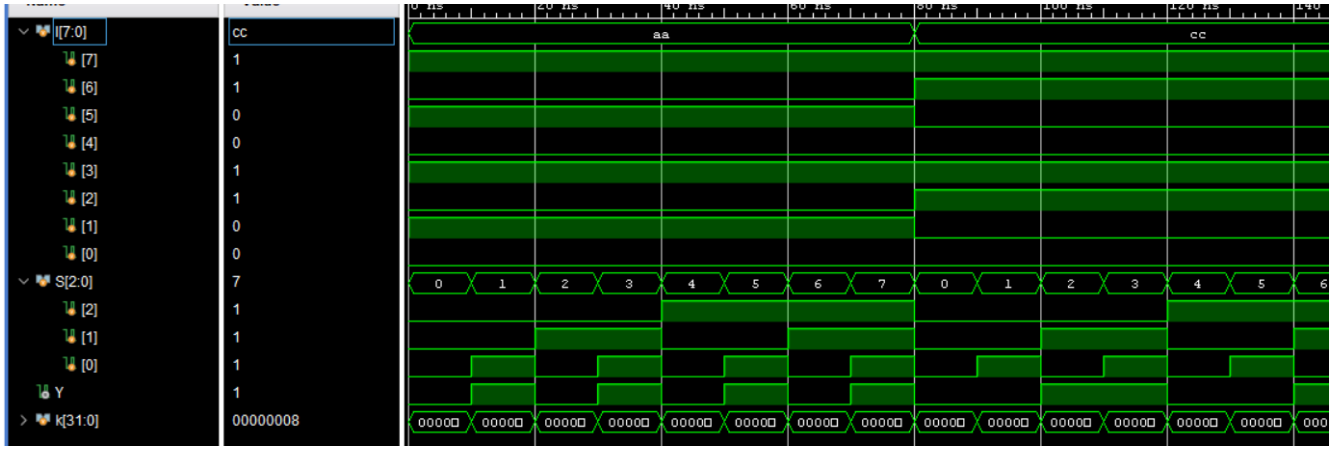
(a) Structural Modeling

```
module mux8_1(
   input [7:0] I,
   input [2:0] S,
   output Y
   );
   wire w1, w2;
      mux4_1 m1(.I(I[3:0]), .S(S[1:0]), .Y(w1));
      mux4_1 m2(.I(I[7:4]), .S(S[1:0]), .Y(w2));
      mux2_1 m3(.I({w2, w1}), .S(S[2]), .Y(Y));

endmodule
```

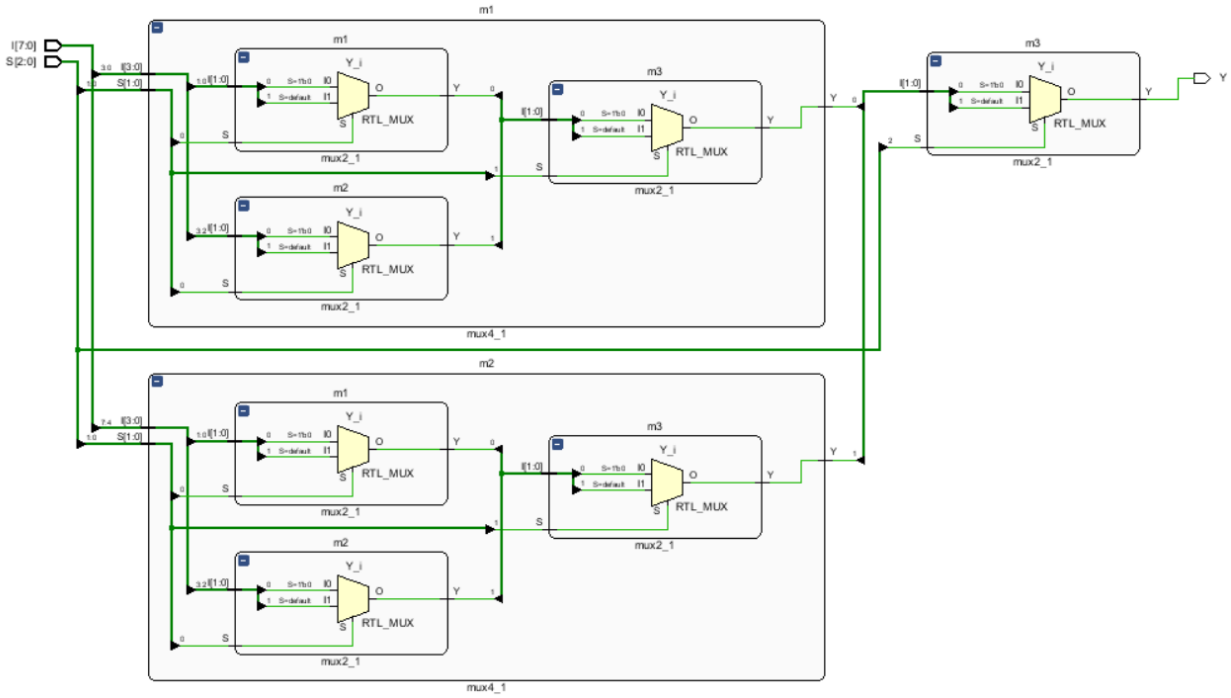
Testbench code:

```
module tb_mux8_1;
   reg [7:0] I;
   reg [2:0] S;
   wire Y;
   mux8_1 dut(.I(I), .S(S), .Y(Y));
   integer k;
   initial begin
      I = 8'b10101010;
      for (k = 0; k < 8; k = k + 1) begin
         S = k;
         #10;
      end
      I = 8'b11001100;
      for (k = 0; k < 8; k = k + 1) begin
         S = k;
         #10;
      end
      $finish;
   end
endmodule
```

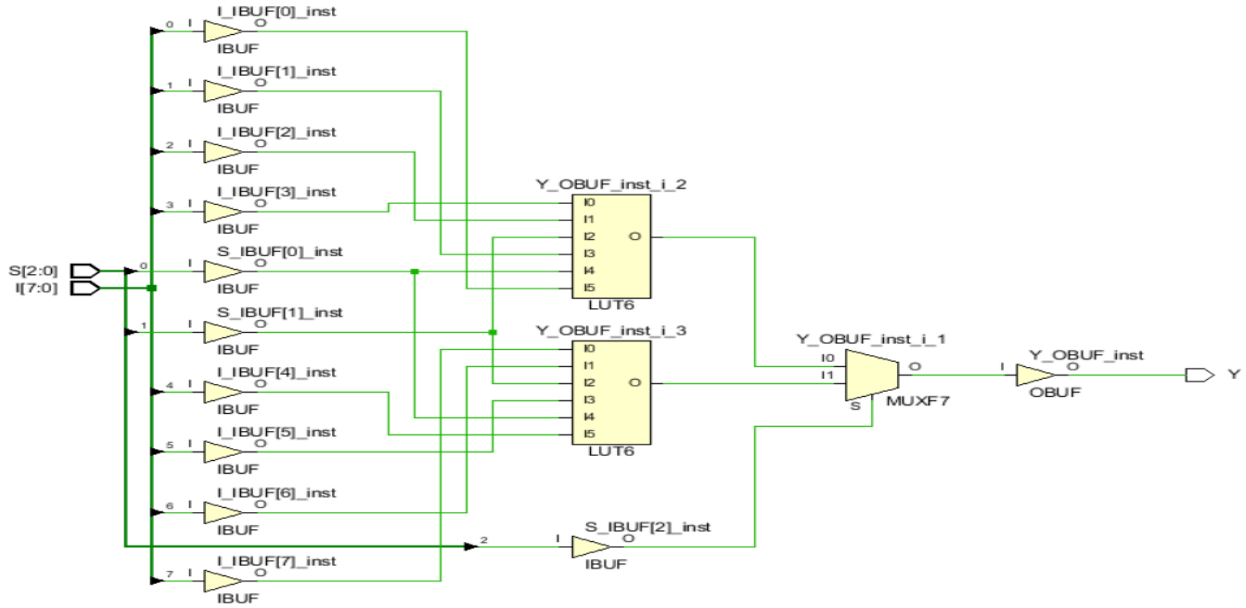
Waveform



RTL Schematic



Synthesis Schematic



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

Design Code:

(a) Structural Modeling

```
module mux16_1(
   input [15:0] I,
   input [3:0] S,
   output Y
   );
      wire w0, w1, w2, w3;
      mux4_1 m0(.I(I[3:0]),.S(S[1:0]),.Y(w0));
      mux4_1 m1(.I(I[7:4]),.S(S[1:0]),.Y(w1));
      mux4_1 m2(.I(I[11:8]),.S(S[1:0]),.Y(w2));
      mux4_1 m3(.I(I[15:12]),.S(S[1:0]),.Y(w3));
      mux4_1 m4(.I({w3, w2, w1,
w0}),.S(S[3:2]),.Y(Y));
endmodule
```

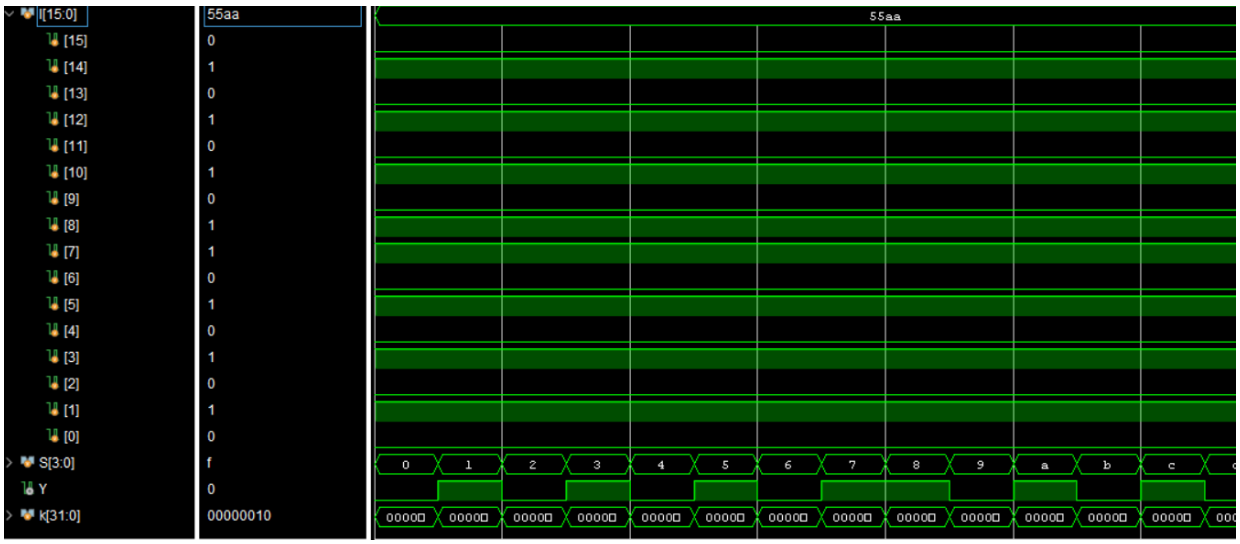
(b) Behavioural Modeling

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

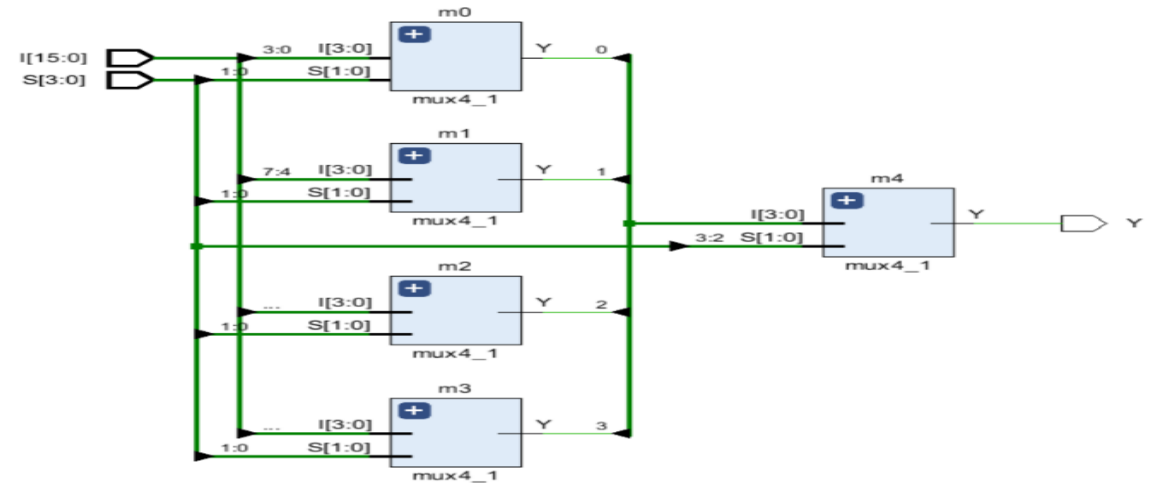
Testbench code:

```
module tb_mux8_1;
   reg [7:0] I;
   reg [2:0] S;
   wire Y;
   mux8_1 dut(.I(I), .S(S), .Y(Y));
   integer k;
   initial begin
      I = 8'b10101010;
      for (k = 0; k < 8; k = k + 1) begin
         S = k;
         #10;
      end
      I = 8'b11001100;
      for (k = 0; k < 8; k = k + 1) begin
         S = k;
         #10;
      end
      $finish;
   end
endmodule
```

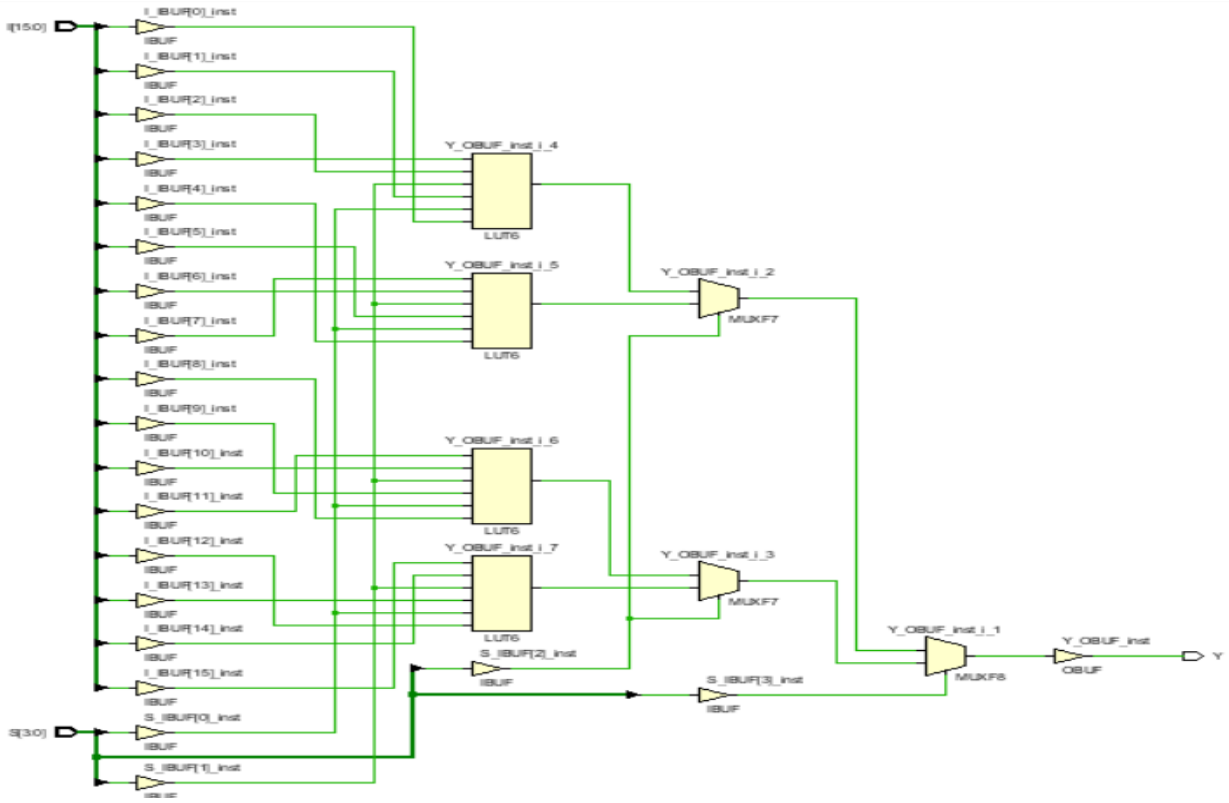
Waveform



RTL Schematic



Synthesis Schematic



(e)1:8 DEMUX

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

Design Code:

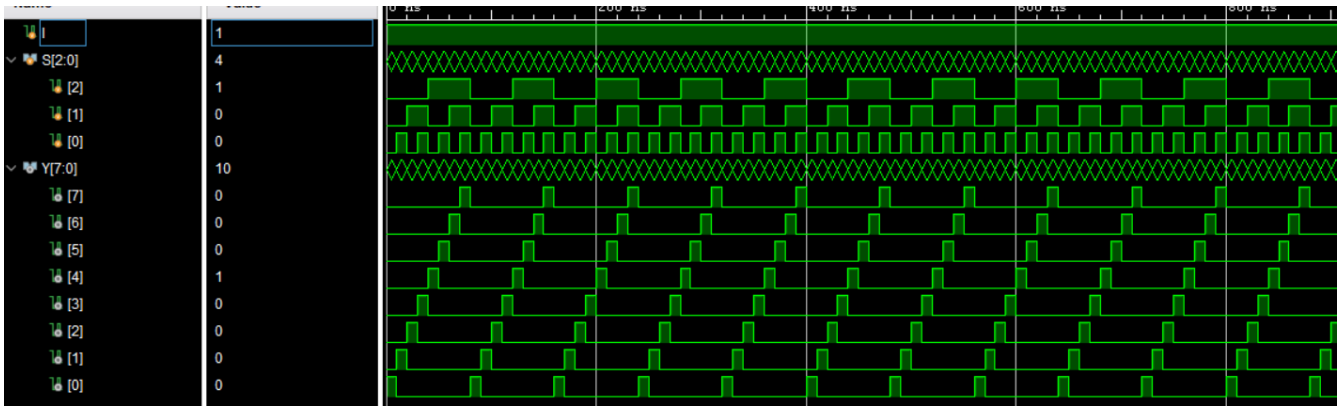
(a) Structural Modeling

```
module dmux1_8(
   input I,
   input [2:0] S,
   output [7:0] Y
    );
      assign Y[0] = I & (~S[2]) & (~S[1]) & (~S[0]);
      assign Y[1] = I & (~S[2]) & (~S[1]) & S[0];
      assign Y[2] = I & (~S[2]) & S[1] & (~S[0]);
      assign Y[3] = I & (~S[2]) & S[1] & S[0];
      assign Y[4] = I & S[2] & (~S[1]) & (~S[0]);
      assign Y[5] = I & S[2] & (~S[1]) & S[0];
      assign Y[6] = I & S[2] & S[1] & (~S[0]);
      assign Y[7] = I & S[2] & S[1] & S[0];
   endmodule
```

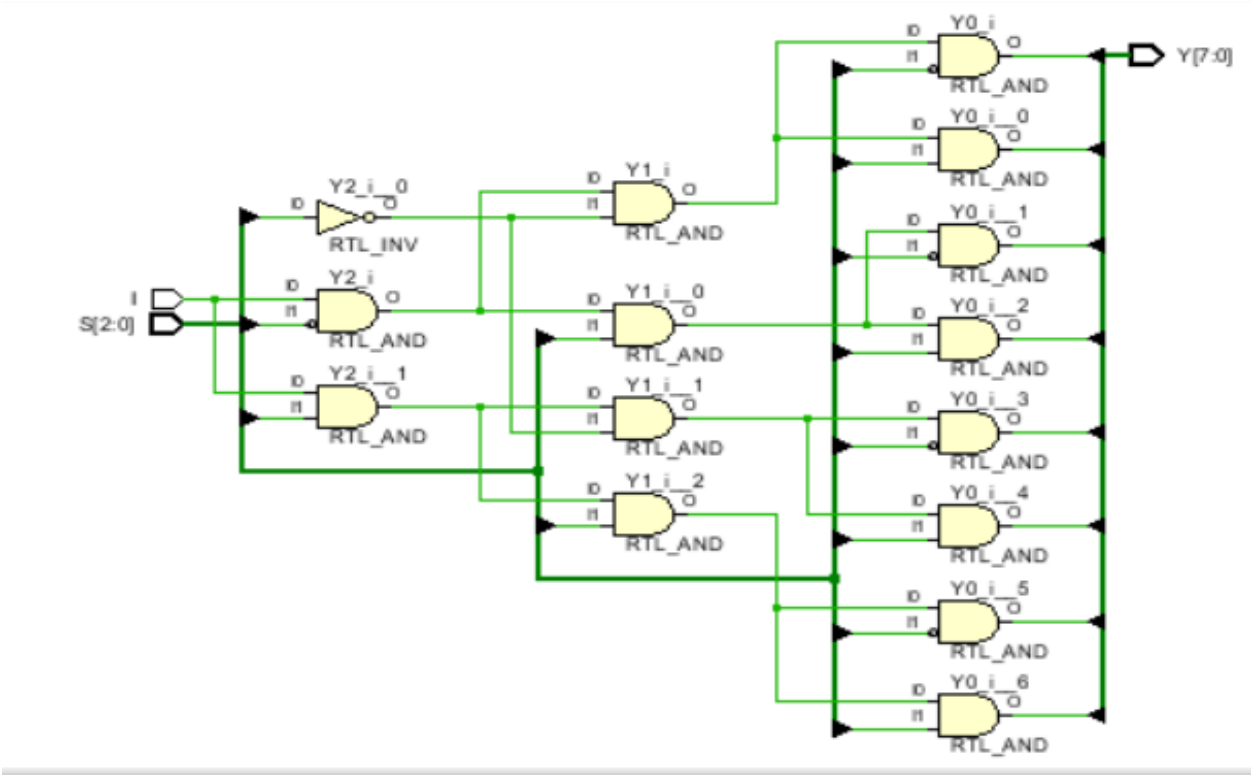
Testbench code:

```
module tb_dmux1_8( );
   reg I;
   reg [2:0]S;
   wire [7:0]Y;
   dmux1_8 dut(I,S,Y);
   initial begin
   $dumpfile("dmux1_8.vcd");
   $dumpvars(1,tb_dmux1_8);
 I=1;
   for(S=0;S<8;S=S+1) #10;
   $finish();
   end
endmodule
```

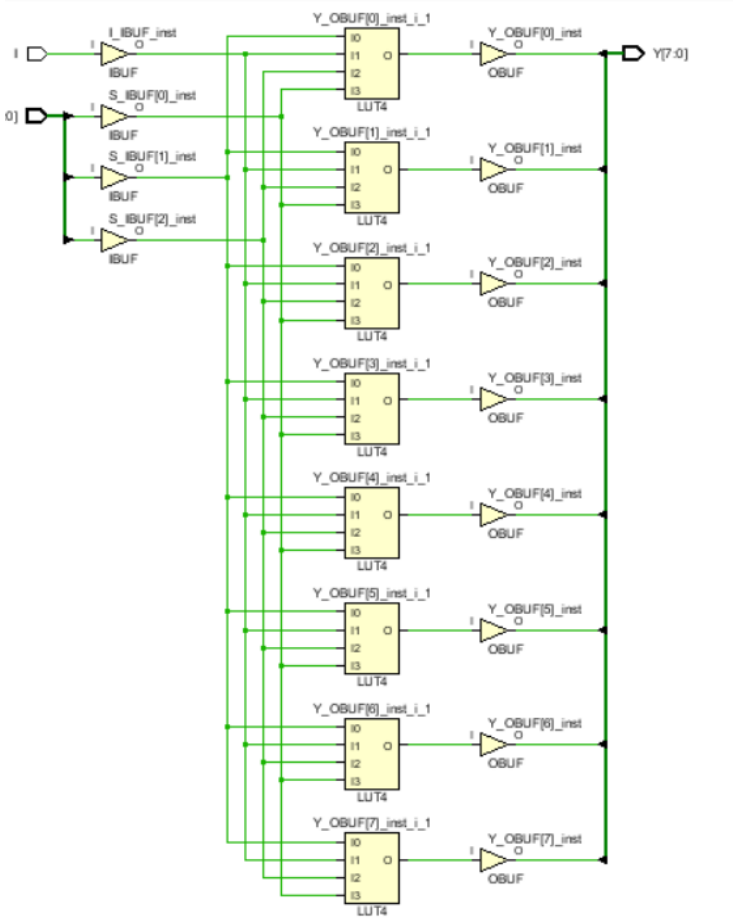
Waveform



RTL Schematic



Synthesis Schematic



Conclusion:

In this lab, we successfully designed and simulated various types of multiplexers and a demultiplexer using Verilog across multiple modeling styles-dataflow, behavioral, and structural.Each design was verified through simulation, confirming correct output behavior for all input combinations and selection lines.

Suggested Reference:

References used by the students:

Rubric wise marks obtained:

Rubrics	1	2	3	4	5	Total
Marks						

Experiment No: 8

Date: _____

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

Competency and Practical Skills:

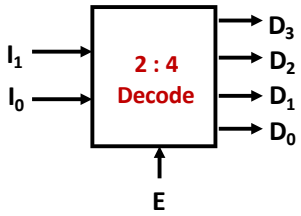
Relevant CO:

Objectives:

Equipment / Instruments: Laptop or Computer with Xilinx Vivado.

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

(a) 2-to-4 Decoder



Truth Table of 2-to-4 Decoder

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

Design Code:

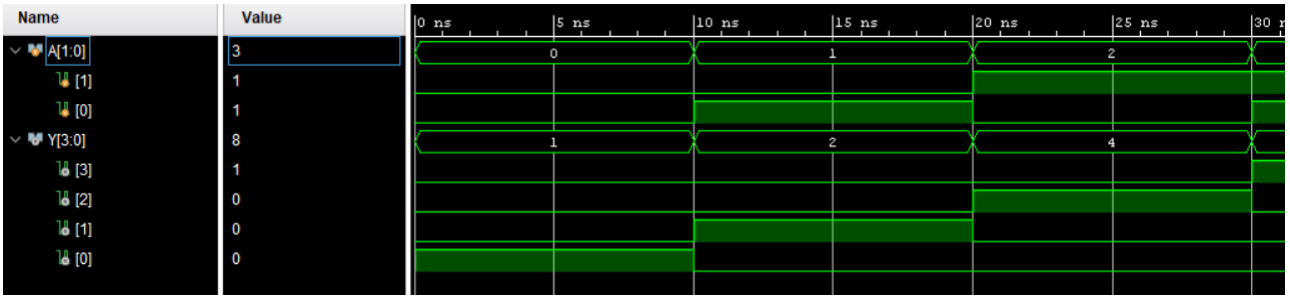
Dateflow Modeling

```
module decoder2_4(
   input [1:0] A,
   output [3:0] Y
   );
   assign Y[0] = ~A[1] & ~A[0];
   assign Y[1] = ~A[1] &  A[0];
   assign Y[2] =  A[1] & ~A[0];
   assign Y[3] =  A[1] &  A[0];

   endmodule
```

Testbench code:

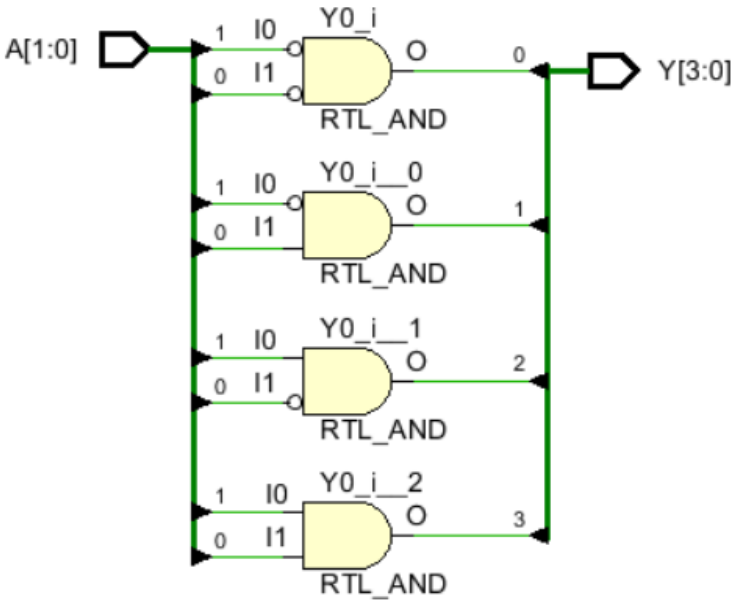
```
module tb_decoder2_4;
   reg [1:0] A;
   wire [3:0] Y;
   decoder2_4 dut(A,Y);
   initial begin
      $dumpfile("decoder2_4.vcd");
      $dumpvars(1, tb_decoder2_4);
      A = 2'b00; #10;
      A = 2'b01; #10;
      A = 2'b10; #10;
      A = 2'b11; #10;
   $finish;
   end
   endmodule
```

Waveform



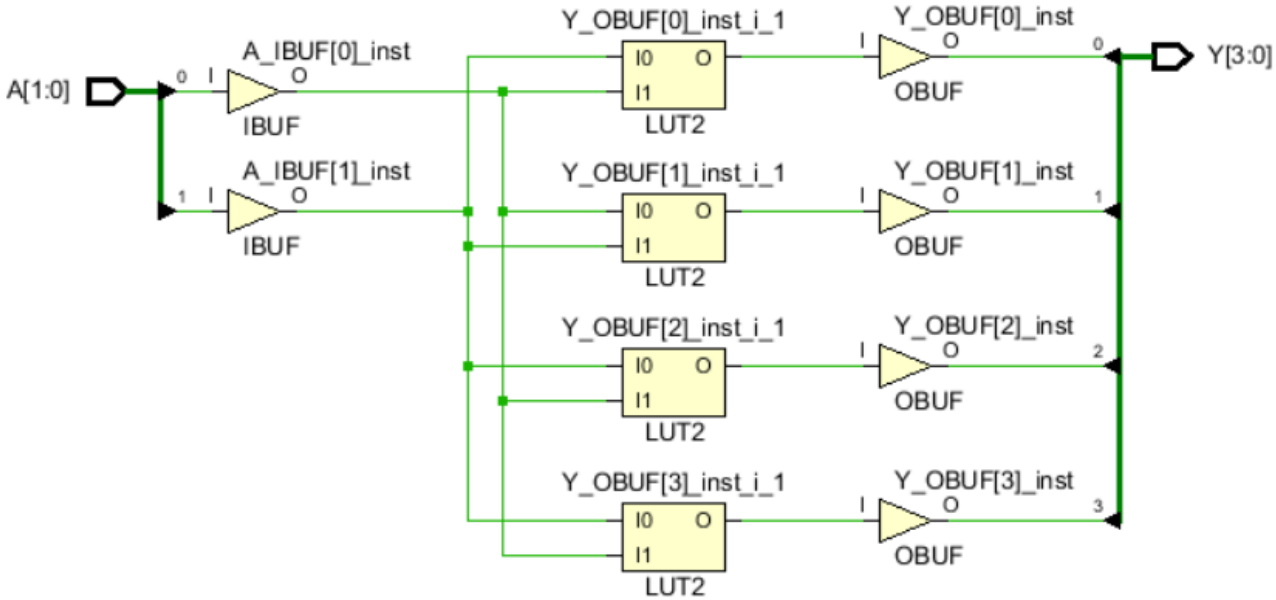
RTL

Schematic



Synthesis

Schematic



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

Design Code:

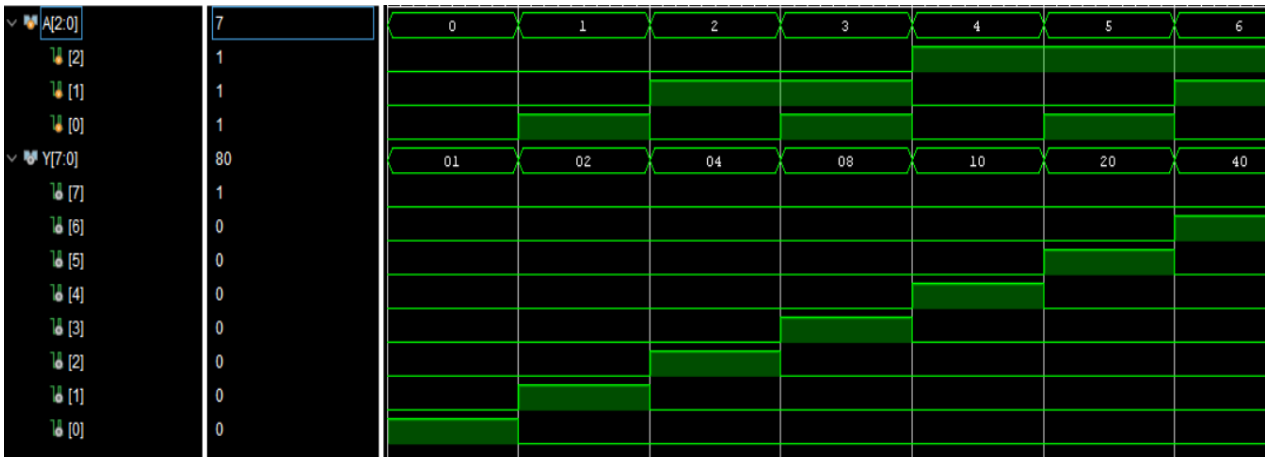
Dateflow Modeling

```
module decoder3_8(
   input [2:0] A,
   output [7:0] Y
   );
      assign Y[0] = (~A[2] & ~A[1] & ~A[0]);
      assign Y[1] = (~A[2] & ~A[1] &  A[0]);
      assign Y[2] = (~A[2] &  A[1] & ~A[0]);
      assign Y[3] = (~A[2] &  A[1] &  A[0]);
      assign Y[4] = ( A[2] & ~A[1] & ~A[0]);
      assign Y[5] = ( A[2] & ~A[1] &  A[0]);
      assign Y[6] = ( A[2] &  A[1] & ~A[0]);
      assign Y[7] = ( A[2] &  A[1] &  A[0]);
   endmodule
```

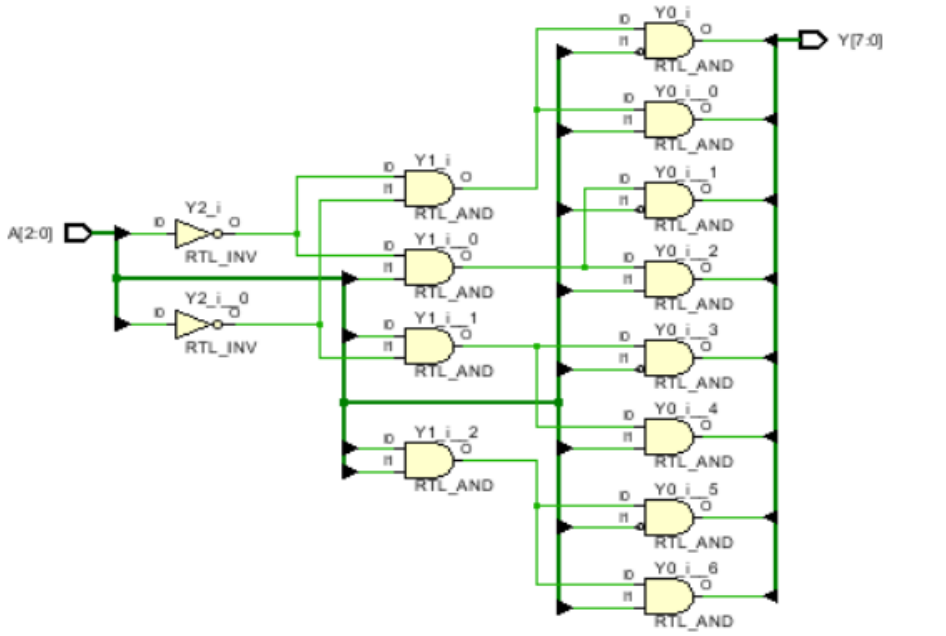
Testbench code:

```
module tb_decoder3_8;
      reg [2:0] A;
      wire [7:0] Y;
      decoder3_8 dut (A,Y);
      initial begin
         $dumpfile("decoder3_8.vcd");
         $dumpvars(0, tb_decoder3_8);
         A = 3'b000; #10;
         A = 3'b001; #10;
         A = 3'b010; #10;
         A = 3'b011; #10;
         A = 3'b100; #10;
         A = 3'b101; #10;
         A = 3'b110; #10;
         A = 3'b111; #10;
         $finish;
      end
   endmodule
```

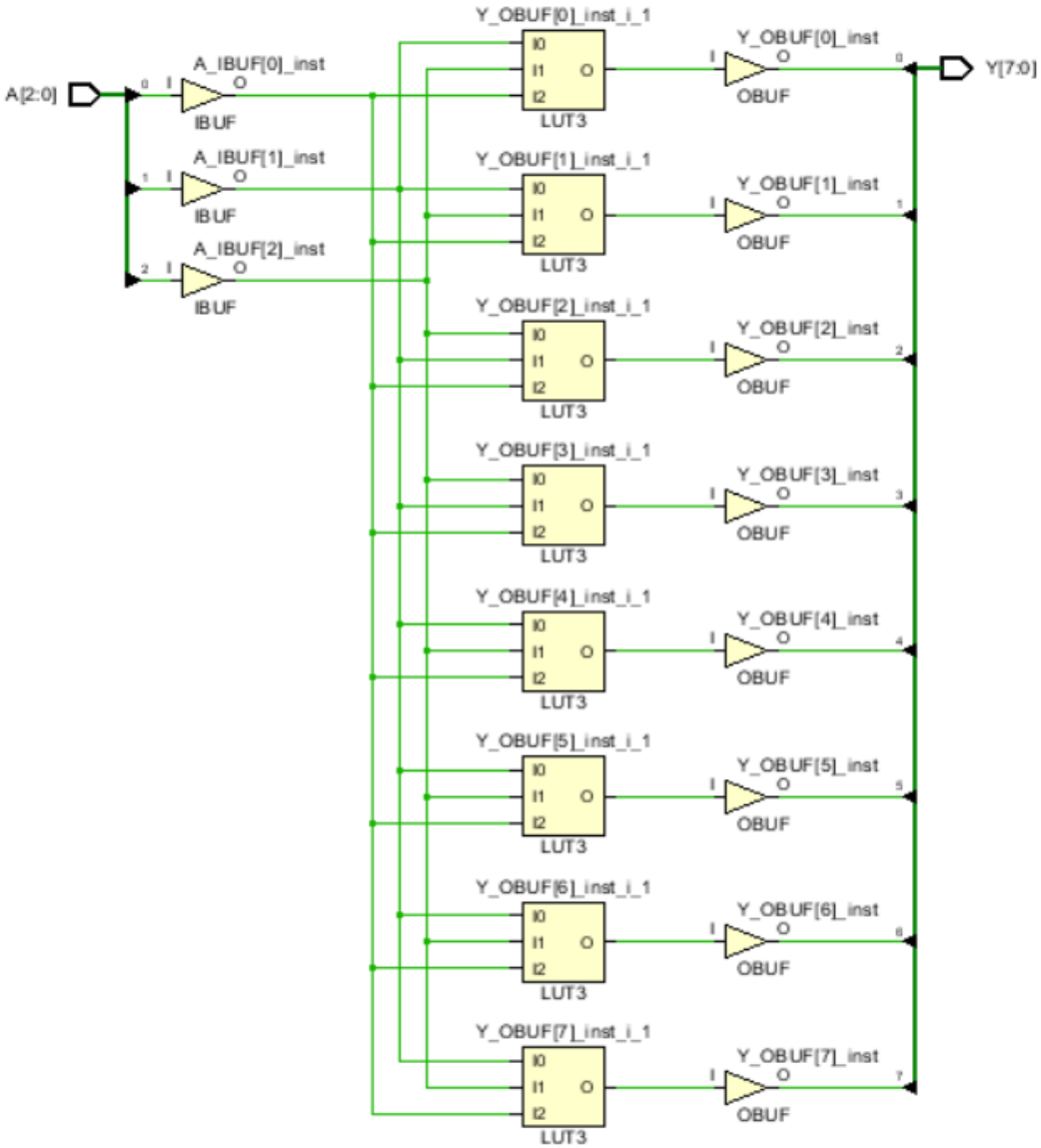
Waveforms:



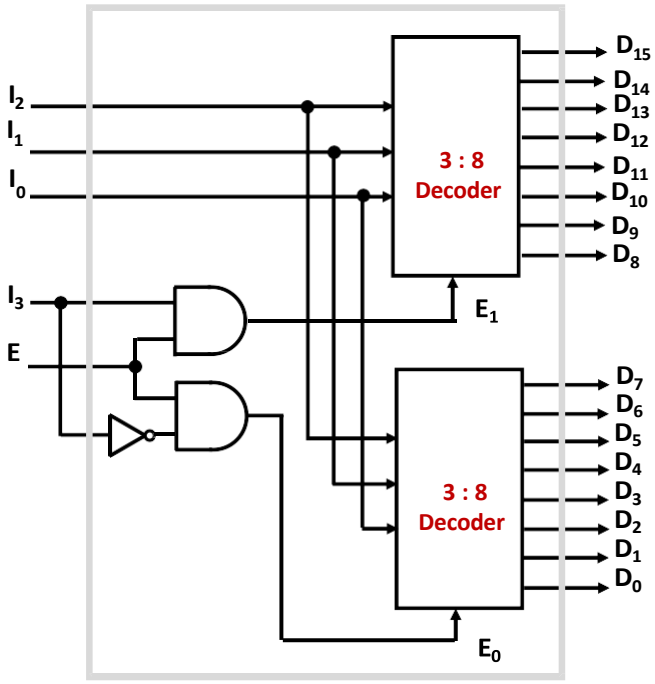
RTL Schematic



Synthesis Schematic



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



Design Code:

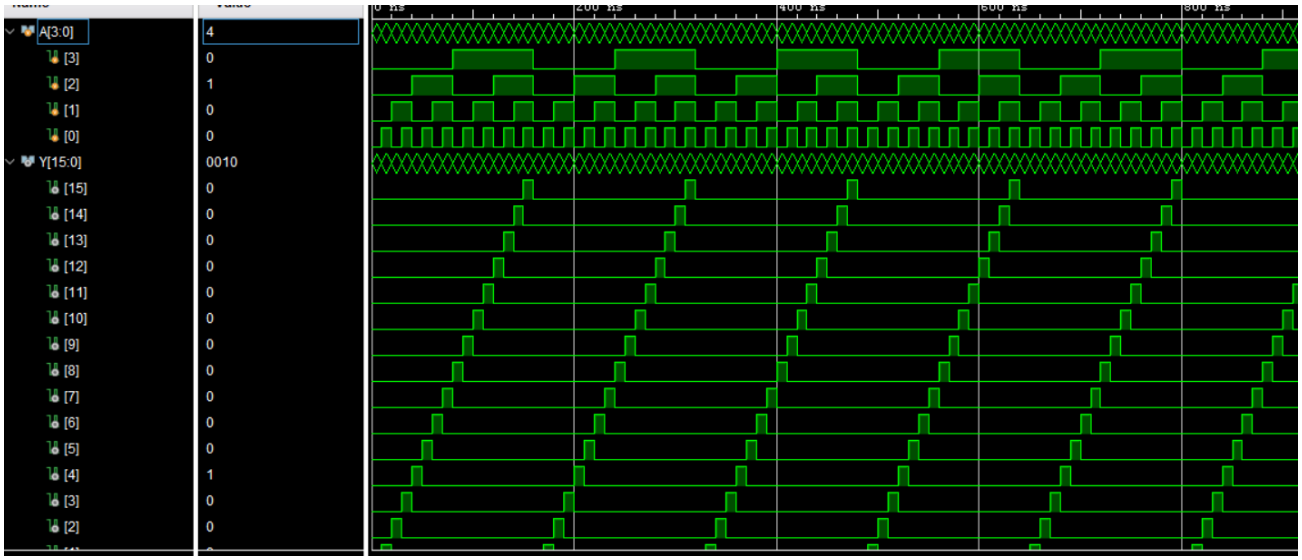
Structural Modeling

```
module decoder4_16(
   input [3:0] A,
   output [15:0] Y
   );
      decoder3_8 dec_low (.A(A[2:0]), .EN(~A[3]),
.Y(Y[7:0]));
      decoder3_8 dec_high(.A(A[2:0]), .EN(A[3]),
.Y(Y[15:8]));
   endmodule
```

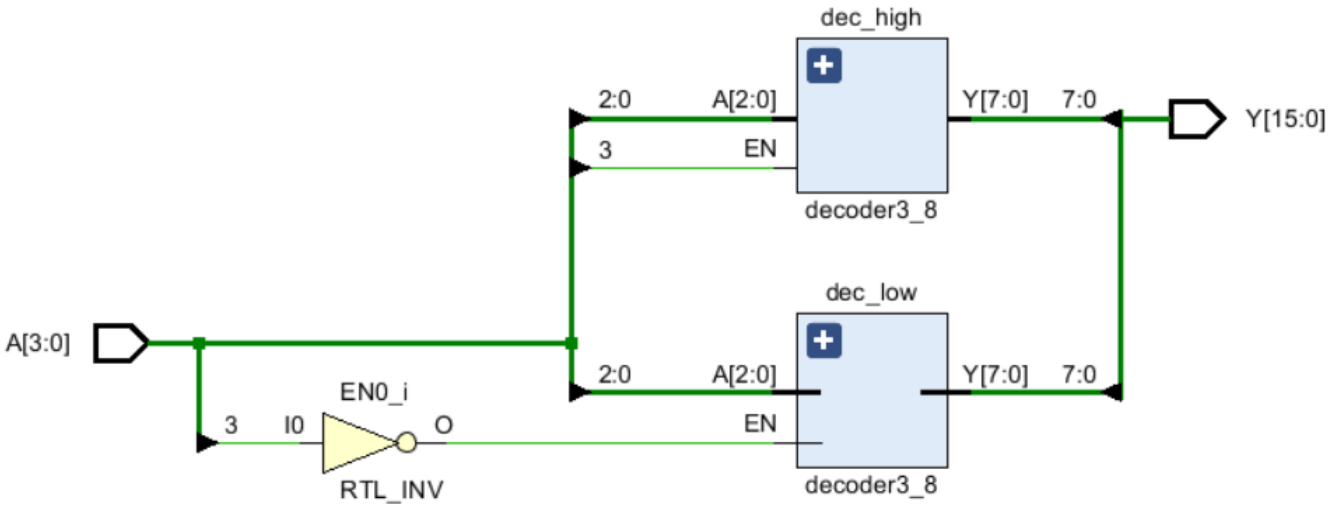
Testbench code:

```
module tb_decoder4_16( );
      reg [3:0] A;
      wire [15:0] Y;
      decoder4_16 dut (A,Y);
      initial begin
         $dumpfile("decoder4_16.vcd");
         $dumpvars(1, tb_decoder4_16);
         for (A = 0; A < 16; A = A + 1) #10;
         $finish;
      end
endmodule
```

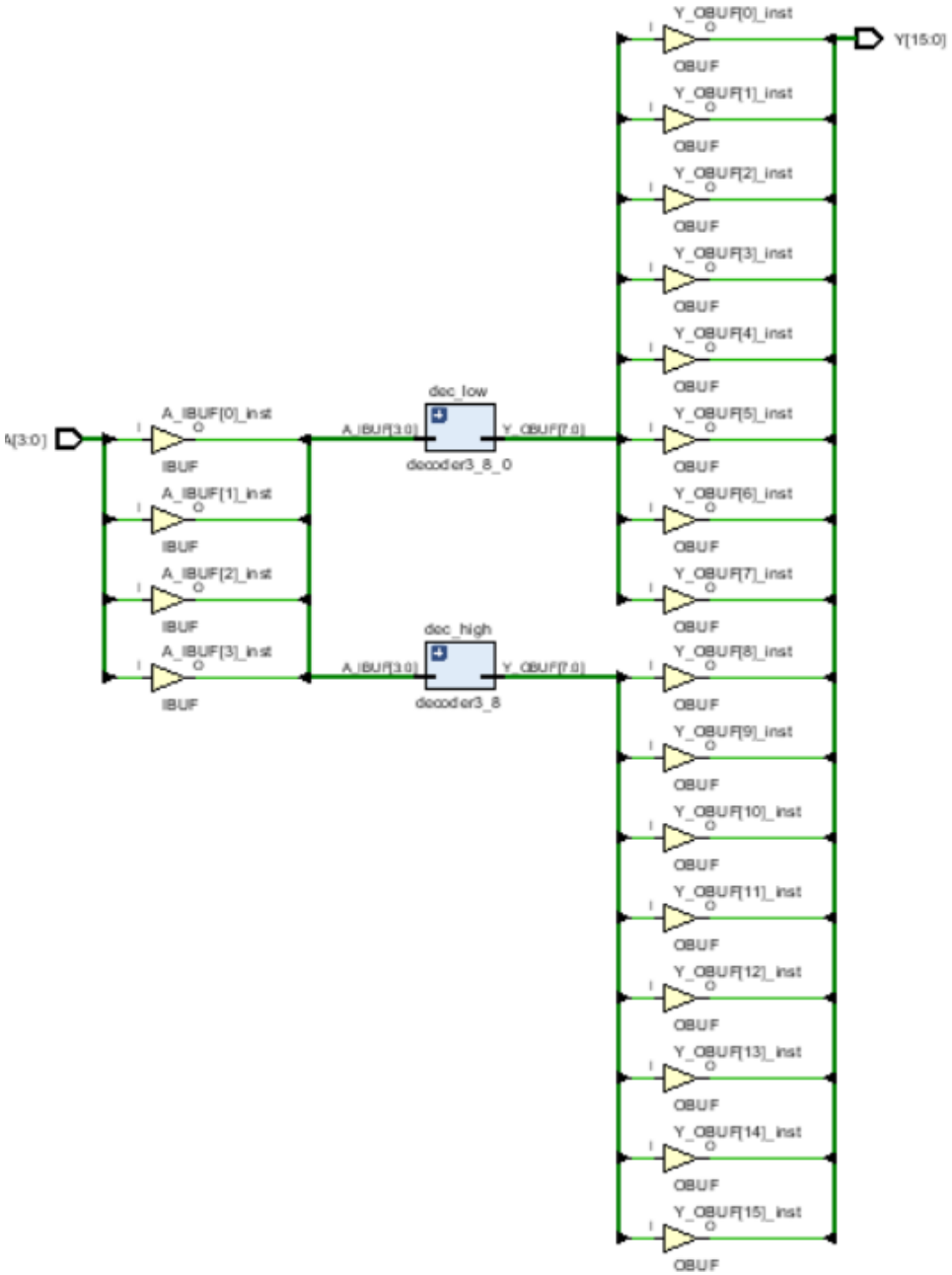
Waveform



RTL Schematic



Synthesis Schematic



Conclusion:

In this lab, we successfully designed and simulated 2:4, 3:8, and 4:16 decoders using Verilog. Each decoder was implemented using behavioral modeling. The lab reinforced key concepts in digital decoding and validated the designs through simulation.

Suggested Reference:

References used by the students:

Rubric wise marks obtained:

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

Experiment No: 9

Date: _____

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

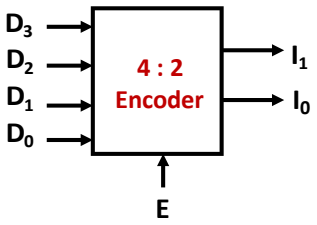
Competency and Practical Skills:

Relevant CO:

Objectives:

Equipment / Instruments: Laptop or Computer with Xilinx Vivado.

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



Truth Table of 4-to-2 Encoder

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

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

a) 4:2 Priority Encoder

Design Code:

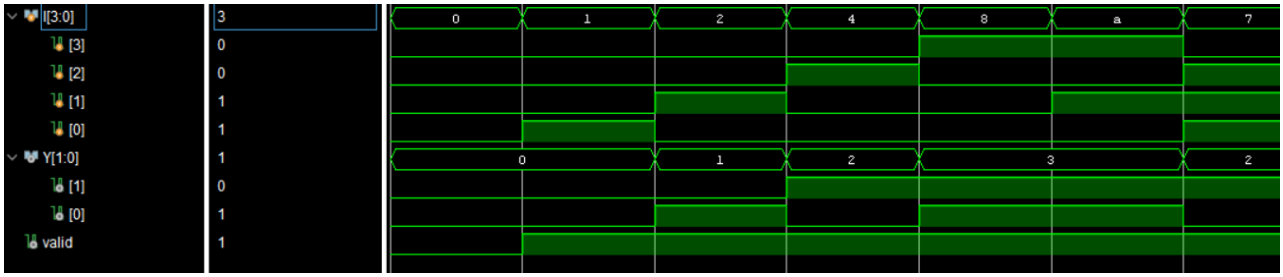
Dataflow Modeling

```
module encoder4_2(
   input [3:0] I,
   output [1:0] Y,
   output valid
   );
    assign Y[1] = I[3] | I[2];
    assign Y[0] = I[3] | (~I[2] & I[1]);
    assign valid = I[3] | I[2] | I[1] | I[0];
endmodule
```

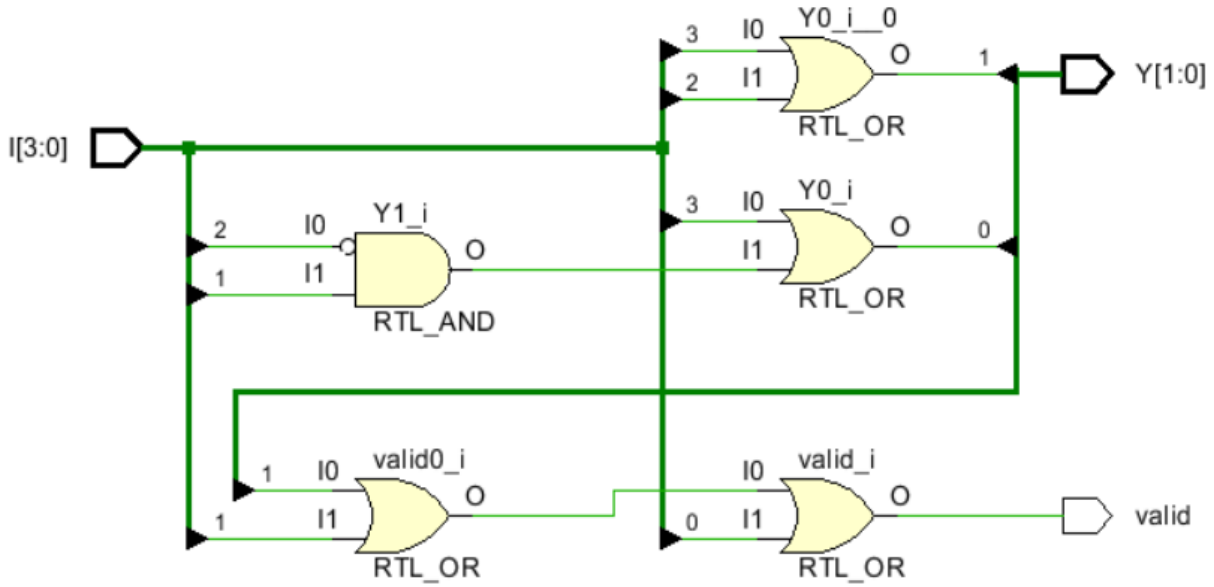
Testbench code:

```
module tb_encoder4_2;
   reg  [3:0] I;
   wire [1:0] Y;
   wire valid;
   encoder4_2 dut (.I(I), .Y(Y),
.valid(valid));
   initial begin
      $dumpfile("encoder4_2.vcd");
      $dumpvars(0, tb_encoder4_2);
      I = 4'b0000; #10;
      I = 4'b0001; #10;
      I = 4'b0010; #10;
      I = 4'b0100; #10;
      I = 4'b1000; #10;
      I = 4'b1010; #10;
      I = 4'b0111; #10;
      I = 4'b0011; #10;
      $finish;
   end
endmodule
```

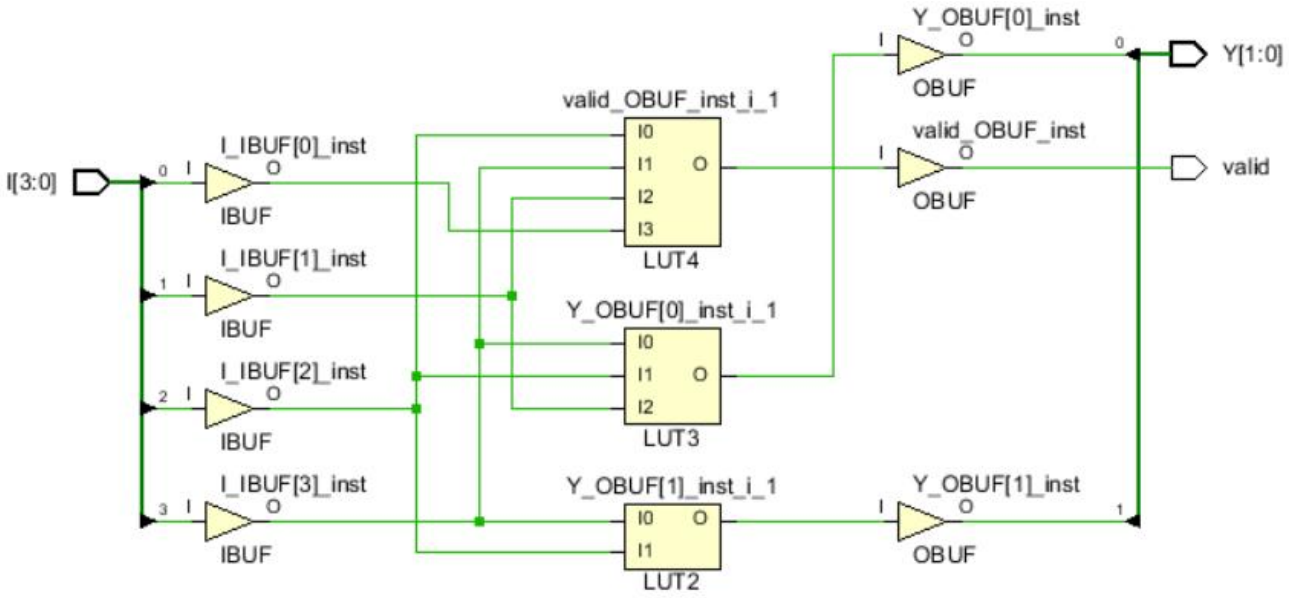
Waveforms



RTL Schematic



Synthesis Schematic



b) 8:3 Priority Encoder

Design Code:

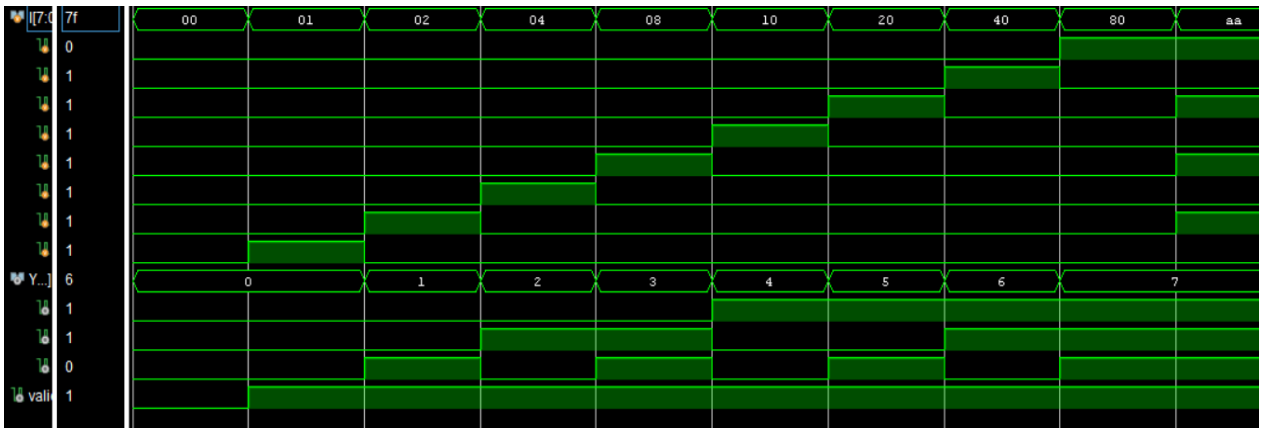
Dataflow Modeling

```
module encoder8_3(I, Y, valid);
    input [7:0] I;
    output [2:0] Y;
    output  valid;
      assign Y[2] = I[7] | I[6] | I[5] | I[4];
       assign Y[1] = I[7] | I[6] | (~I[5] & ~I[4] & (I[3] |
I[2]));
       assign Y[0] = I[7] | (~I[6] & (I[5] | (~I[4] & (I[3] |
(~I[2] & I[1])))));
       assign valid = |I;
endmodule
```

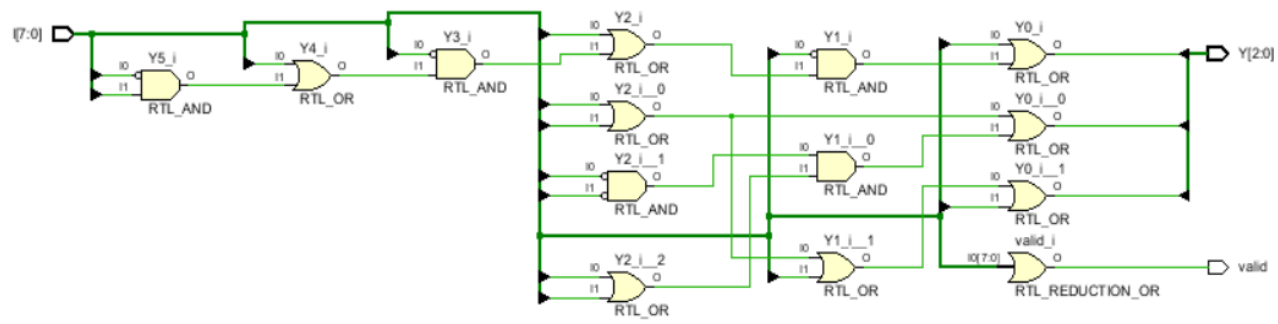
Testbench code:

```
module tb_encoder8_3;
    reg [7:0] I;
    wire [2:0] Y;
    wire valid;
    encoder8_3 dut (I,Y,valid);
    initial begin
        $dumpfile("encoder8_3.vcd");
        $dumpvars(1, tb_encoder8_3);
        I = 8'b00000000; #10;
        I = 8'b00000001; #10;
        I = 8'b00000010; #10;
        I = 8'b00000100; #10;
        I = 8'b00001000; #10;
        I = 8'b00010000; #10;
        I = 8'b00100000; #10;
        I = 8'b01000000; #10;
        I = 8'b10000000; #10;
        I = 8'b10101010; #10;
        I = 8'b01111111; #10;
        $finish;
    end
endmodule
```

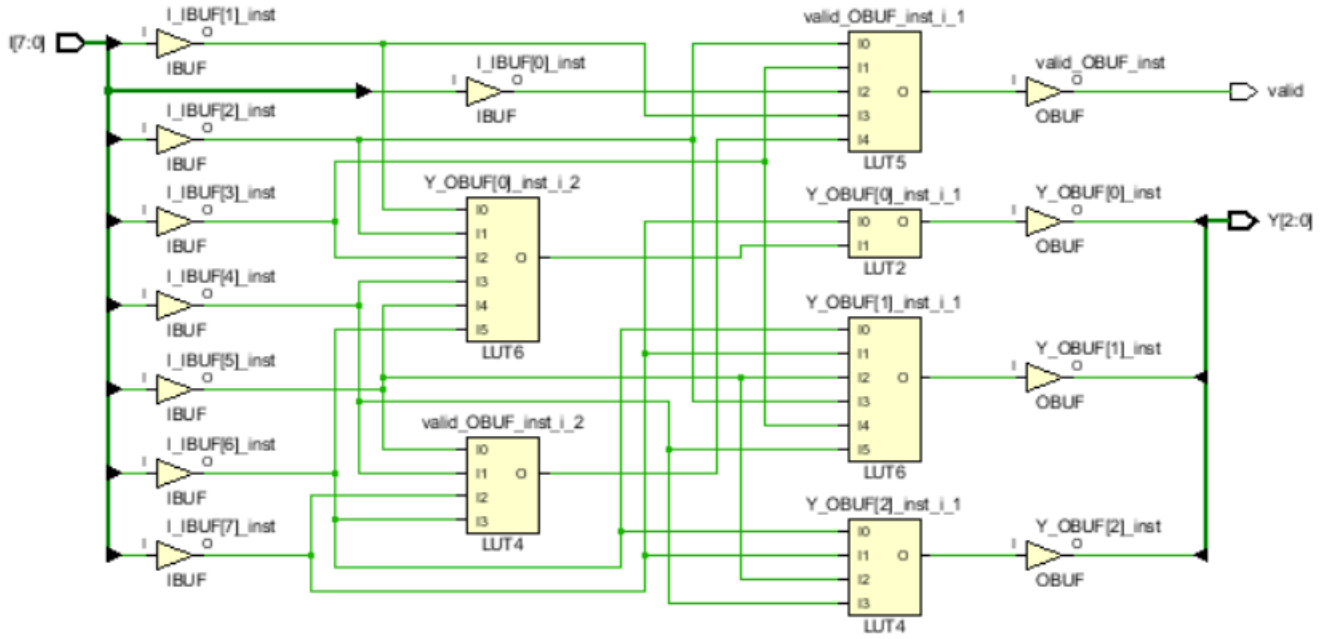
Waveform



RTL Schematic



Synthesis Schematic



Conclusion:

The 4:2 and 8:3 Priority Encoders were successfully designed and simulated in Xilinx Vivado using Verilog/VHDL. The simulation waveforms confirmed correct operation: the encoder outputs correspond to the highest-priority active input, with lower-priority inputs ignored. The results matched the expected truth tables, proving the design works correctly.

Suggested Reference:

References used by the students:

Rubric wise marks obtained:

Rubrics	1	2	3	4	5	Total
Marks						

Experiment No: 10

Date: _____

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

Competency and Practical Skills:

Relevant CO:

Objectives:

Equipment / Instruments: Laptop or Computer with Vivado.

Basic Theory:

A 4-bit comparator is a combinational circuit that compares two 4-bit binary numbers (A and B). The comparator has three outputs:

- **A > B** (A is greater than B)
- **A = B** (A is equal to B)
- **A < B** (A is less than B)

The comparison is done by checking the most significant bit (MSB) first. If the MSBs are equal, the next lower bit is compared, and so on until all four bits are checked. Using logic gates, the circuit generates the appropriate output depending on the relationship between the two binary inputs.

Program Code:

Design Code:

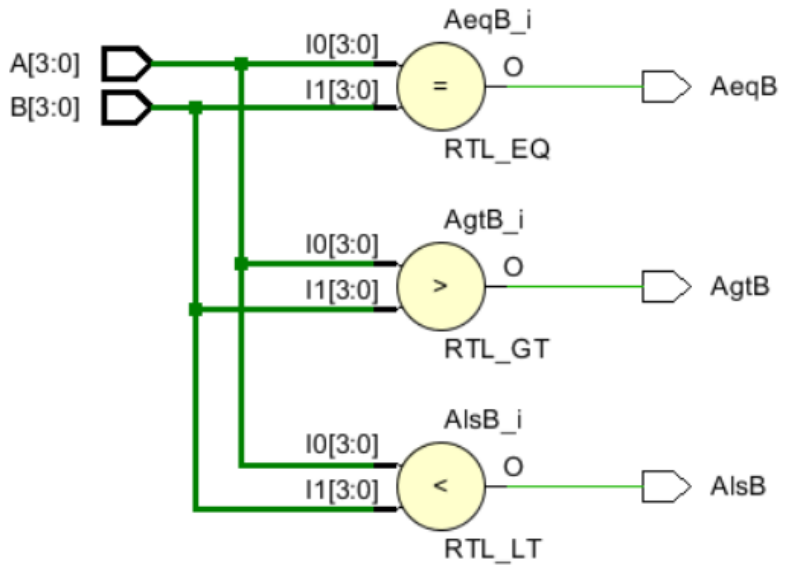
Dataflow Modeling

```
module comparator_4 (
input  [3:0] A, B,
output   AeqB,
output   AgtB,
output   AlsB
);
assign AeqB = (A == B);
assign AgtB = (A > B);
assign AlsB = (A < B);
endmodule
```

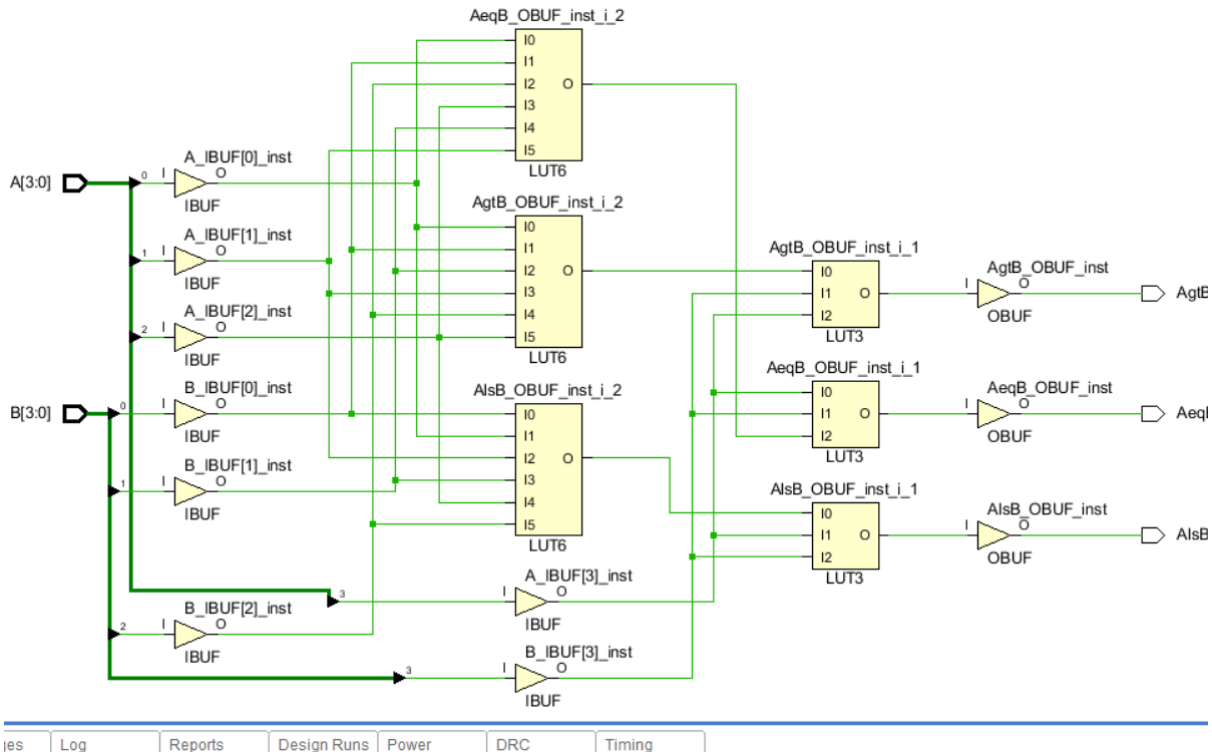
Testbench code:

```
module tb_comparator_4;
reg  [3:0] A, B;
wire AeqB, AgtB, AlsB;
comparator_4
dut(.A(A),.B(B),.AeqB(AeqB),.AgtB(AgtB),.AlsB(AlsB));
initial begin
$dumpfile("comparator_4.vcd");
$dumpvars(0, tb_comparator_4);
A = 4'b0000; B = 4'b0000; #10; // Equal
A = 4'b1010; B = 4'b0110; #10; // Greater
A = 4'b0011; B = 4'b0100; #10; // Less
A = 4'b1111; B = 4'b1111; #10; // Equal
A = 4'b1000; B = 4'b1100; #10; // Less
A = 4'b0111; B = 4'b0011; #10; // Greater
$finish;
end
endmodule
```

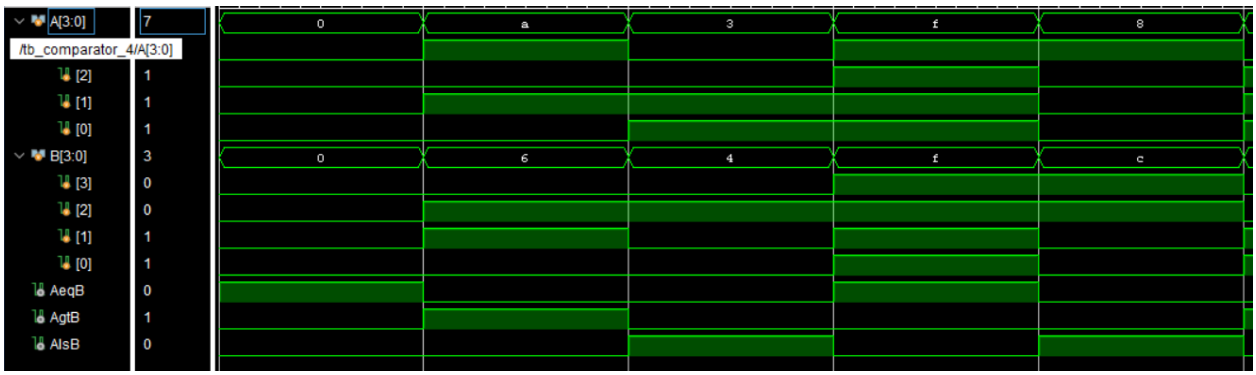
RTL Schematic:



Synthesis Schematic:



Output waveform:



Conclusion:

The 4-bit comparator was successfully designed using Verilog, comparing two 4-bit inputs to determine equality, greater-than, and less-than conditions. Simulation confirmed correct logical behavior across all input combinations, reinforcing concepts of bitwise comparison and control signal generation in digital design.

Quiz:

Suggested Reference:

References used by the students:

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 Vivado.

Basic Theory:

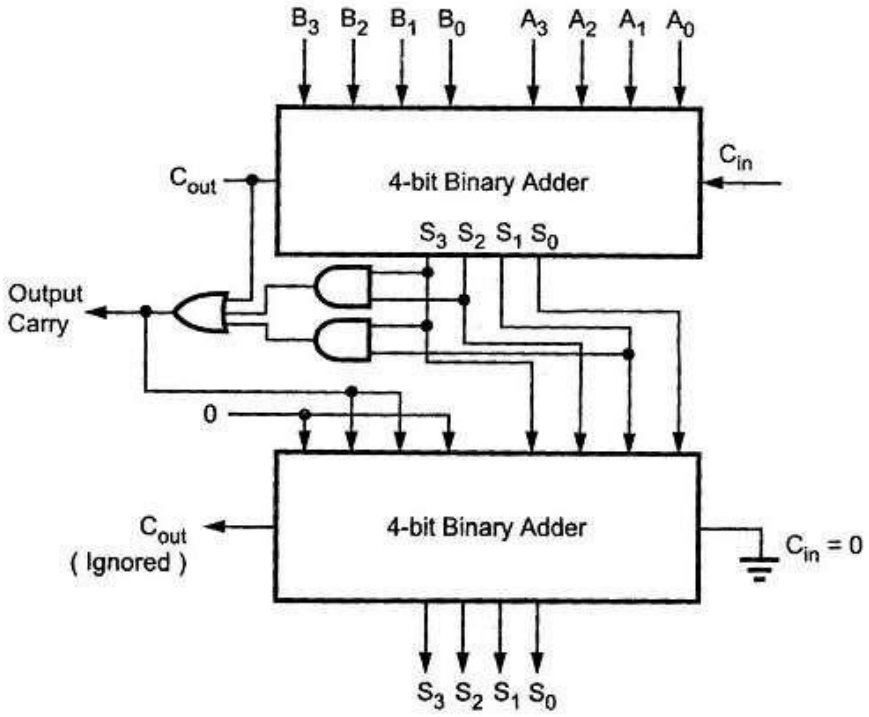
BCD Adder – Theory

- Adds two decimal digits in BCD form (0–9).
- If sum > 9, add 6 (0110) to correct result.
- Used in calculators and digital systems needing decimal outputs.

Ripple Carry Adder – Theory

- Made of full adders connected in series.
- Carry output of one adder goes into the next → "ripple effect."
- Simple design but slow for large numbers.

BCD Adder Block diagram:



Program Code:

(a)BCD Adder

Design Code:

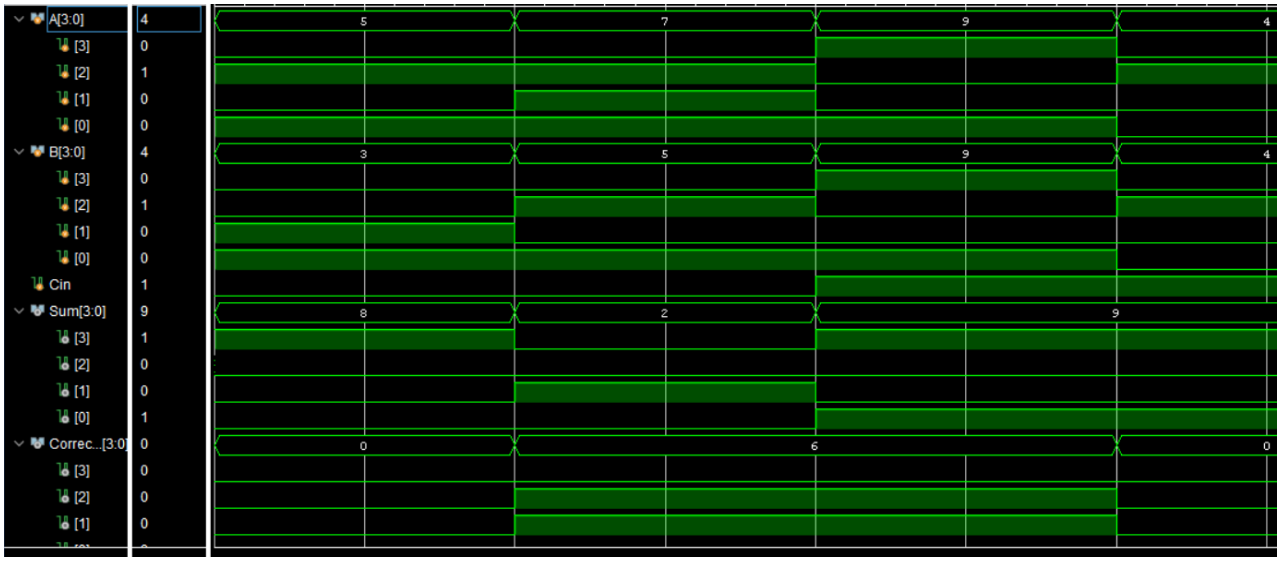
Dataflow Modeling

```
module bcd_adder(
   input [3:0] A,
   input [3:0] B,
   input Cin,
   output [3:0] Sum,
   output [3:0] Correction,
   output Cout
   );
     wire [4:0] temp_sum;
     wire correction_needed;
    assign temp_sum = A + B + Cin;
     assign correction_needed = (temp_sum > 9);
     assign Correction = correction_needed ? 4'b0110 :
4'b0000;
     assign {Cout, Sum} = temp_sum + Correction;
   endmodule
```

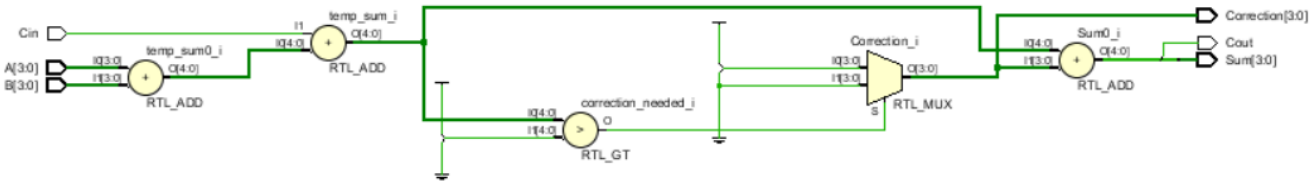
Testbench code:

```
module tb_bcd_adder;
  reg  [3:0] A, B;
  reg        Cin;
  wire [3:0] Sum, Correction;
  wire       Cout;
  bcd_adder uut (
    .A(A), .B(B), .Cin(Cin),
    .Sum(Sum), .Correction(Correction),
.Cout(Cout)
  );
  initial begin
    $dumpfile("bcd_adder.vcd");
    $dumpvars(0, tb_bcd_adder);
    A = 4'd5; B = 4'd3; Cin = 0; #10;
    A = 4'd7; B = 4'd5; Cin = 0; #10;
    A = 4'd9; B = 4'd9; Cin = 1; #10;
    A = 4'd4; B = 4'd4; Cin = 1; #10;
    $finish;
  end
endmodule
```

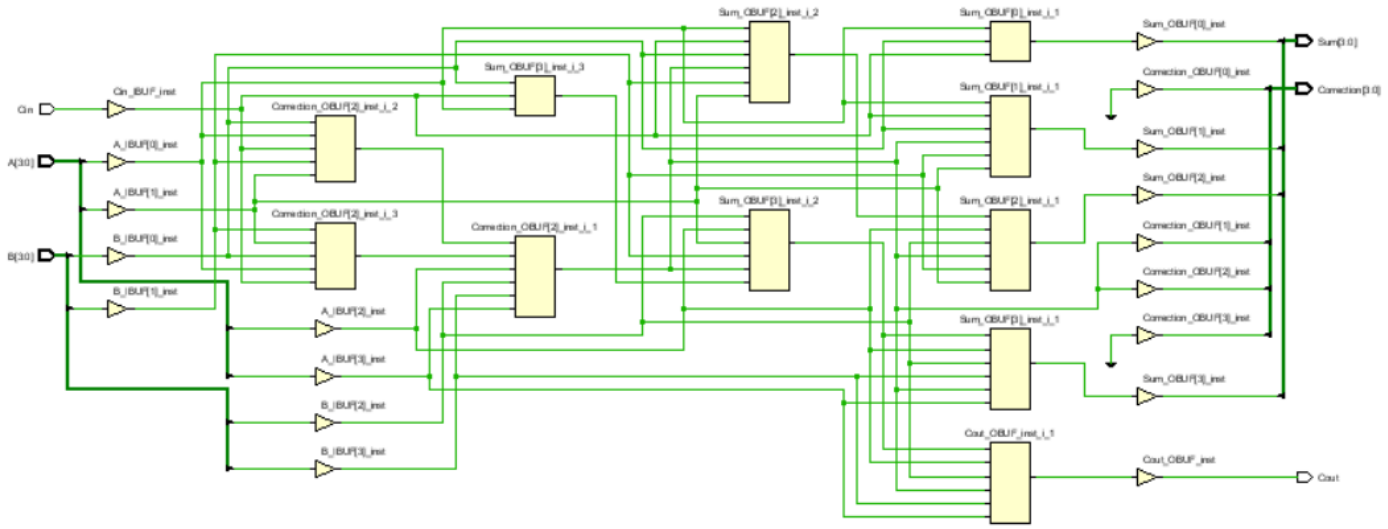
waveform:



RTL Schematic



Synthesis Schematic



(b)Ripple Carry Adder

Design Code:

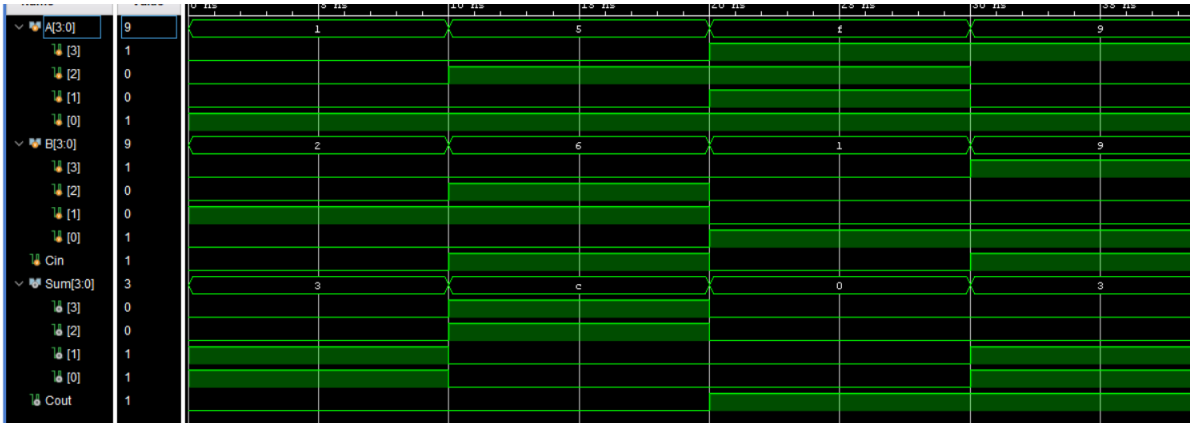
Structural Modelling:

```
module ripple_adder(
  input  [3:0] A,
  input  [3:0] B,
  input        Cin,
  output [3:0] Sum,
  output       Cout
);
  wire C1, C2, C3;
  FA_2 FA0 (A[0], B[0], Cin,  Sum[0], C1);
  FA_2 FA1 (A[1], B[1], C1,   Sum[1], C2);
  FA_2 FA2 (A[2], B[2], C2,   Sum[2], C3);
  FA_2 FA3 (A[3], B[3], C3,   Sum[3], Cout);
endmodule
```

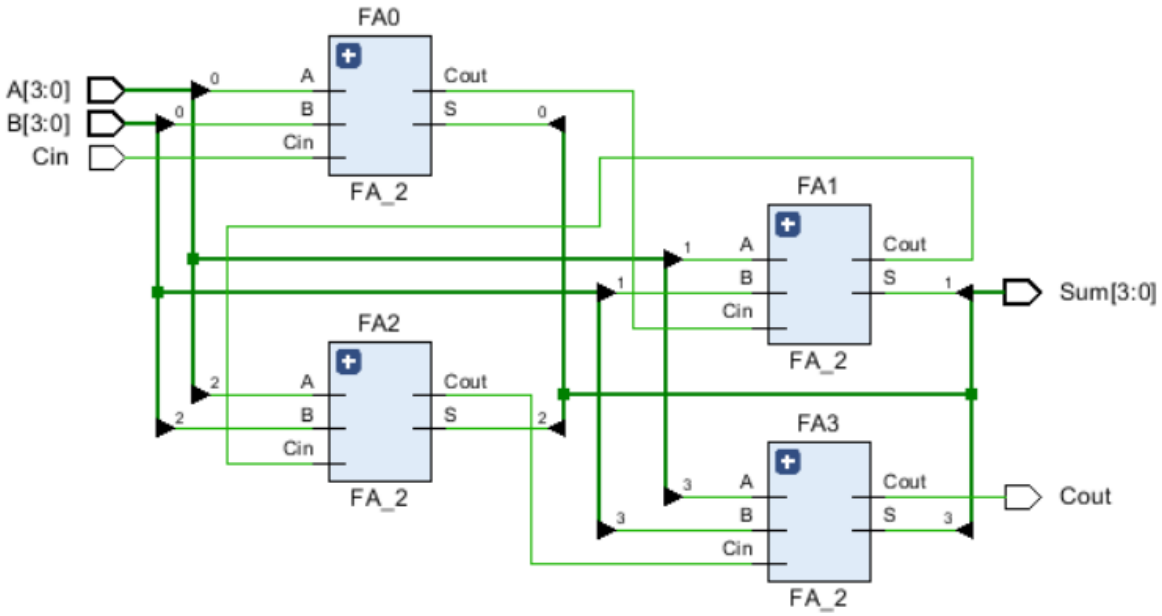
Testbench code:

```
module tb_ripple_adder;
  reg  [3:0] A, B;
  reg        Cin;
  wire [3:0] Sum;
  wire       Cout;

  ripple_adder uut (
    .A(A), .B(B), .Cin(Cin),
    .Sum(Sum), .Cout(Cout)
  );

  initial begin
    $dumpfile("ripple_adder.vcd");
    $dumpvars(0, tb_ripple_adder);

    A = 4'b0001; B = 4'b0010; Cin = 0; #10;
    A = 4'b0101; B = 4'b0110; Cin = 1; #10;
    A = 4'b1111; B = 4'b0001; Cin = 0; #10;
    A = 4'b1001; B = 4'b1001; Cin = 1; #10;
    $finish;
  end
endmodule
```

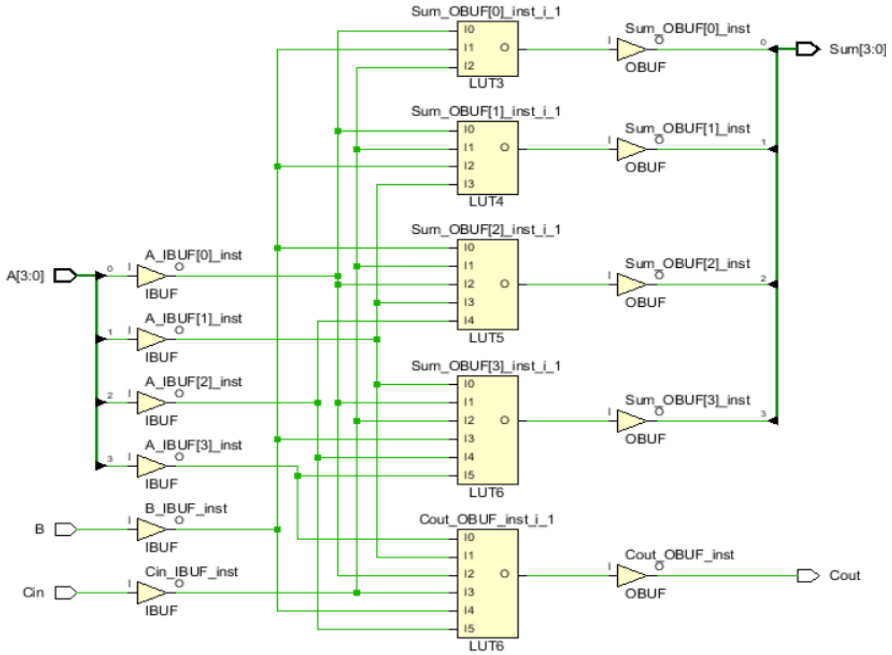
Waveform:



RTL Schematic



Synthesis Schematic



Conclusion:

The BCD adder and ripple carry adder were successfully designed using Verilog. Simulations verified correct binary and decimal addition, with proper carry propagation in the ripple carry structure. This lab reinforced concepts of arithmetic logic, modular design, and signal timing in digital systems.

Suggested Reference:

References used by the students:

Rubric wise marks obtained:

Rubrics	1	2	3	4	5	Total
Marks						

Experiment No: 16

Date: _____

Aim: Introduction to the LTspice tool.

Competency and Practical Skills:

Relevant CO:

Objectives:

Equipment / Instruments: Laptop or Computer with the LTspice tool.

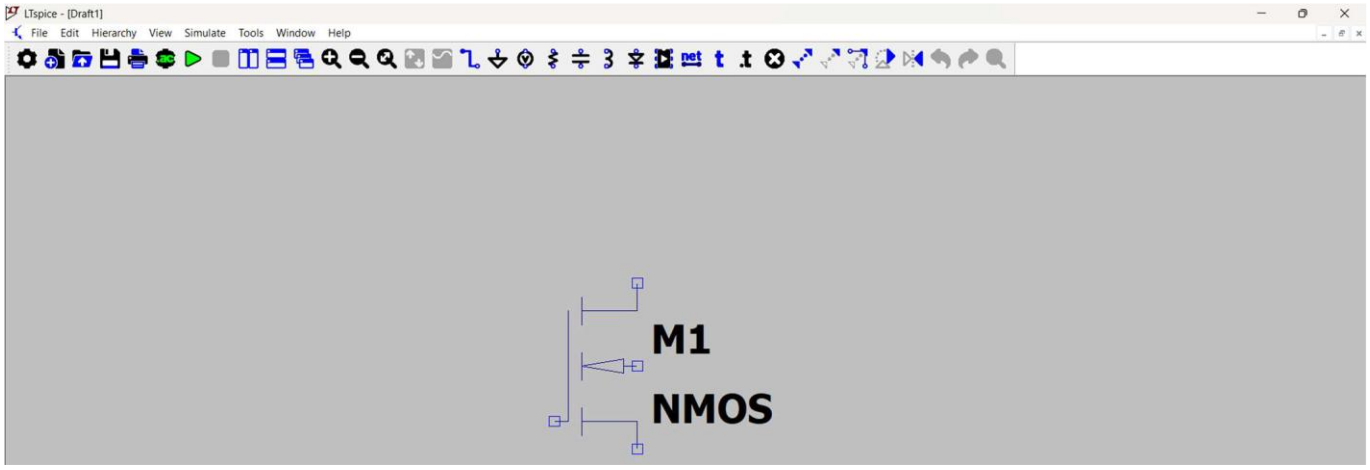
Introduction:

LTspice TOOL

LTspice is a widely used SPICE-based circuit simulator developed by Analog Devices.
It is primarily used for simulating analog, digital, and mixed-signal circuits.
Unlike layout tools such as Microwind, which focus on VLSI physical design, LTspice is dedicated to schematic-based design,
circuit simulation, and waveform analysis. It allows designers to verify the behavior of electronic circuits before hardware implementation.

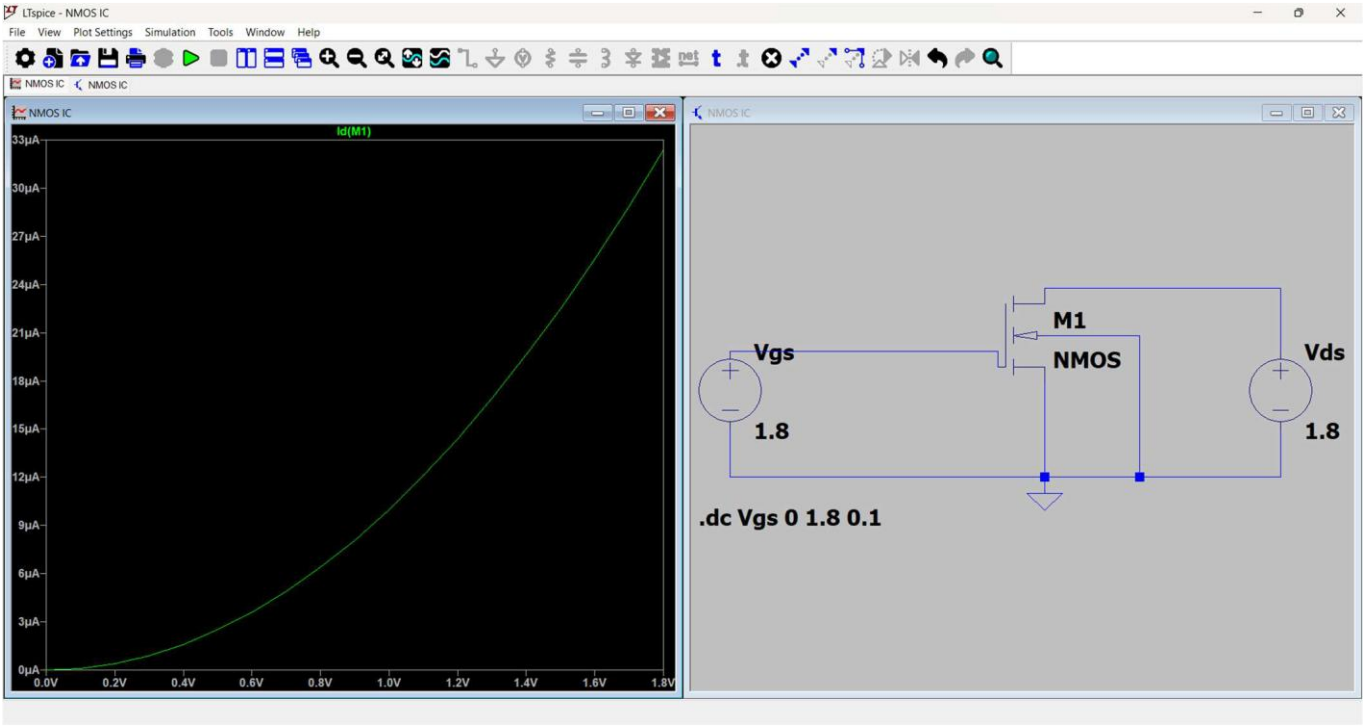
Key features include:

- **Schematic Capture**: Graphical entry of circuit diagrams using a wide range of standard
- components (R, L, C, diode, BJT, MOSFET, Op-Amps, etc.).
- Simulation Types Supported:
- Transient analysis (time-domain response)
- DC sweep analysis (IV curves, bias points)
- AC analysis (frequency response, Bode plots)
- Noise analysis
- FFT analysis for frequency spectrum
- Parameter sweep and Monte Carlo simulations
- **Device Models:** Built-in library of standard components and semiconductor devices. Users can
- also import SPICE models from manufacturers.
- **Subcircuits and Hierarchy:** Supports creation of subcircuits for hierarchical design.
- **Netlist Generator:** Automatically generates SPICE netlist for simulations.
- **Waveform Viewer:** Interactive plotting of voltage, current, power, and FFT analysis with cursor
- measurement.
- **User Flexibility:** Customizable parameters, scripting, and batch simulation.
- **Performance:** High-speed simulation engine optimized for switching power supplies and large
- analog circuits.



Technology Library Available:

- Standard SPICE device models: Resistors, capacitors, inductors, controlled sources.
- Semiconductor devices: Diodes, BJTs, MOSFETs, JFETs, MESFETs.
- Specialized models for operational amplifiers, comparators, voltage regulators.
- Support for user-defined models via .lib, .model, and .subckt.
- Behavioral sources for advanced mathematical expressions.



Conclusion: In this experiment, we learned the basics of LTspice and how to simulate circuits using DC, AC, and transient analysis. LTspice proved useful for verifying circuit behavior before hardware implementation

Suggested Reference:

References used by the students:

Rubric wise marks obtained:

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

Experiment No: 17

Date: _____

Aim: Implementation of Resistive load and CMOS inverters using LTspice.

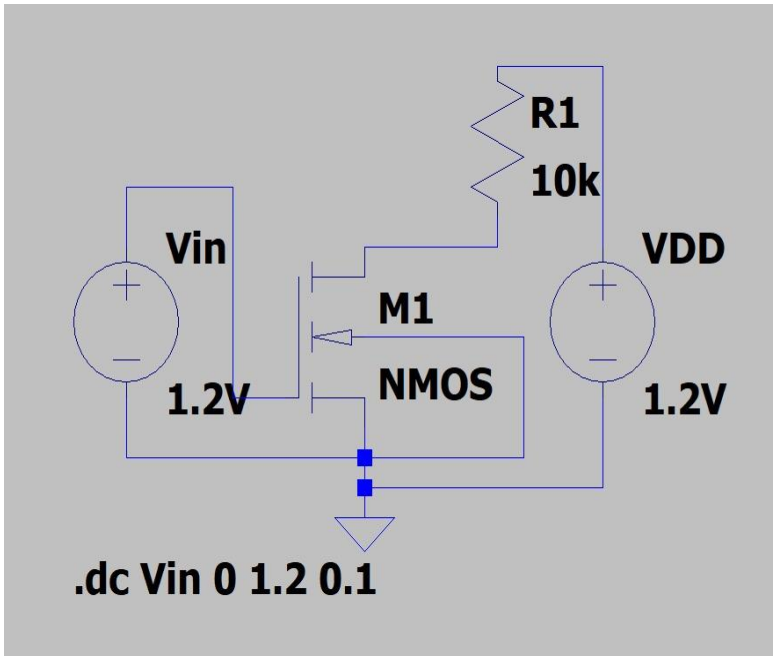
Competency and Practical Skills:

Relevant CO:

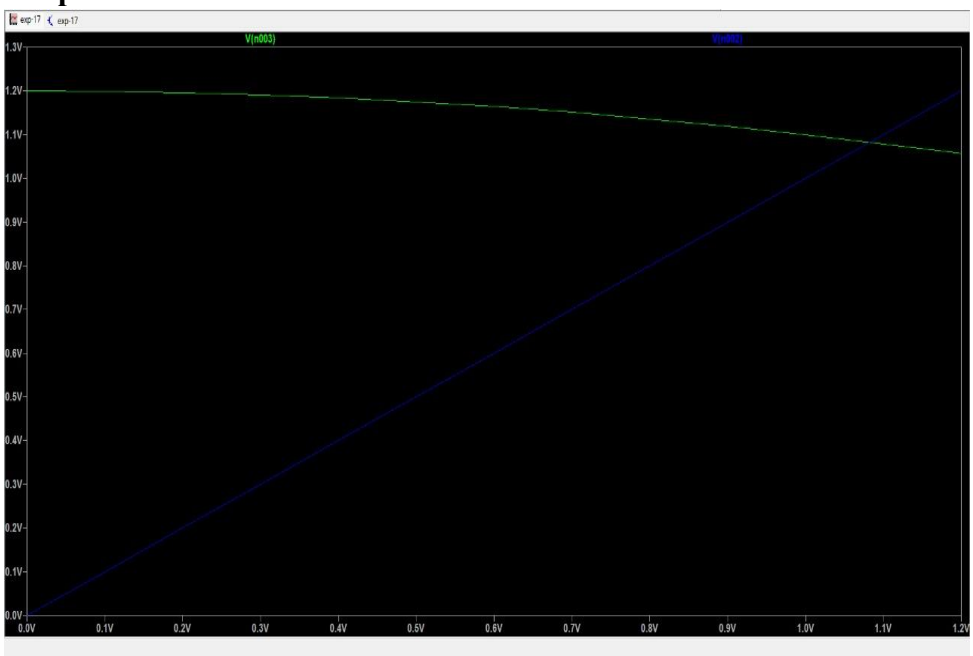
Objectives:

Equipment / Instruments: Laptop or Computer with LTspice Tool.

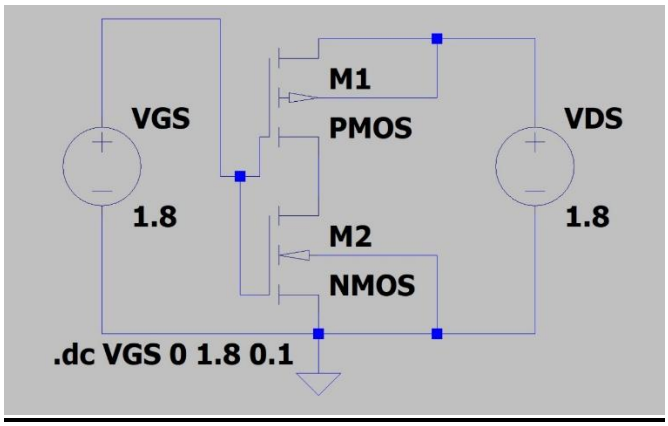
Resistive Load Inverter Schematic



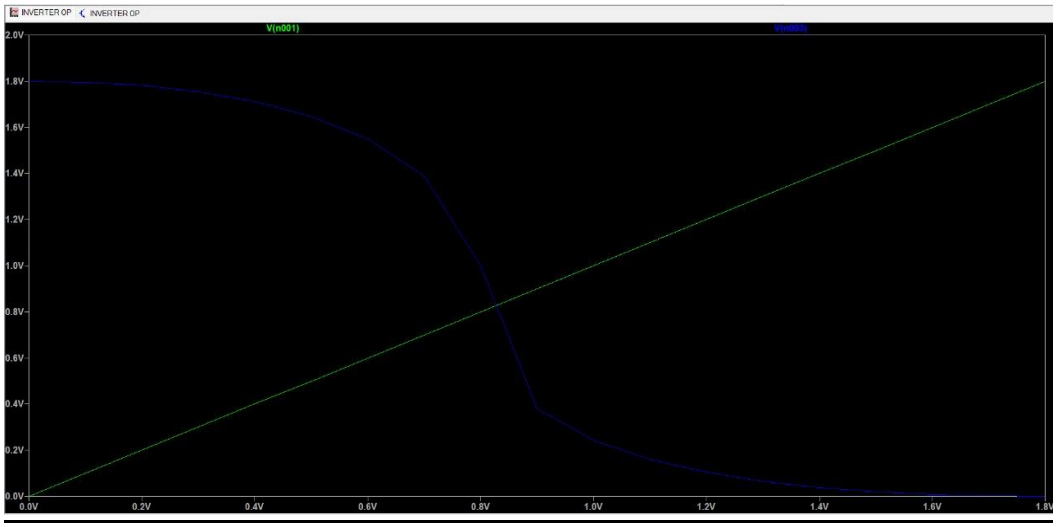
Output Waveform:



CMOS Inverter Schematic



Output Waveforms



Conclusion:

In this experiment, we performed implementation of Resistive load inverter and CMOS inverter with their output waveforms and schematics.

Suggested Reference:

References used by the students:

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 Microwind.

Competency and Practical Skills:

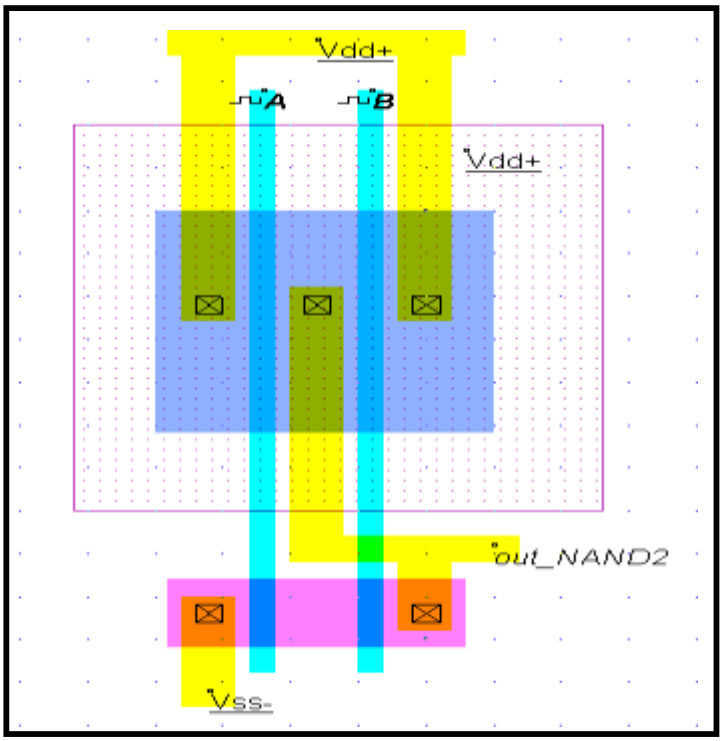
Relevant CO:

Objectives:

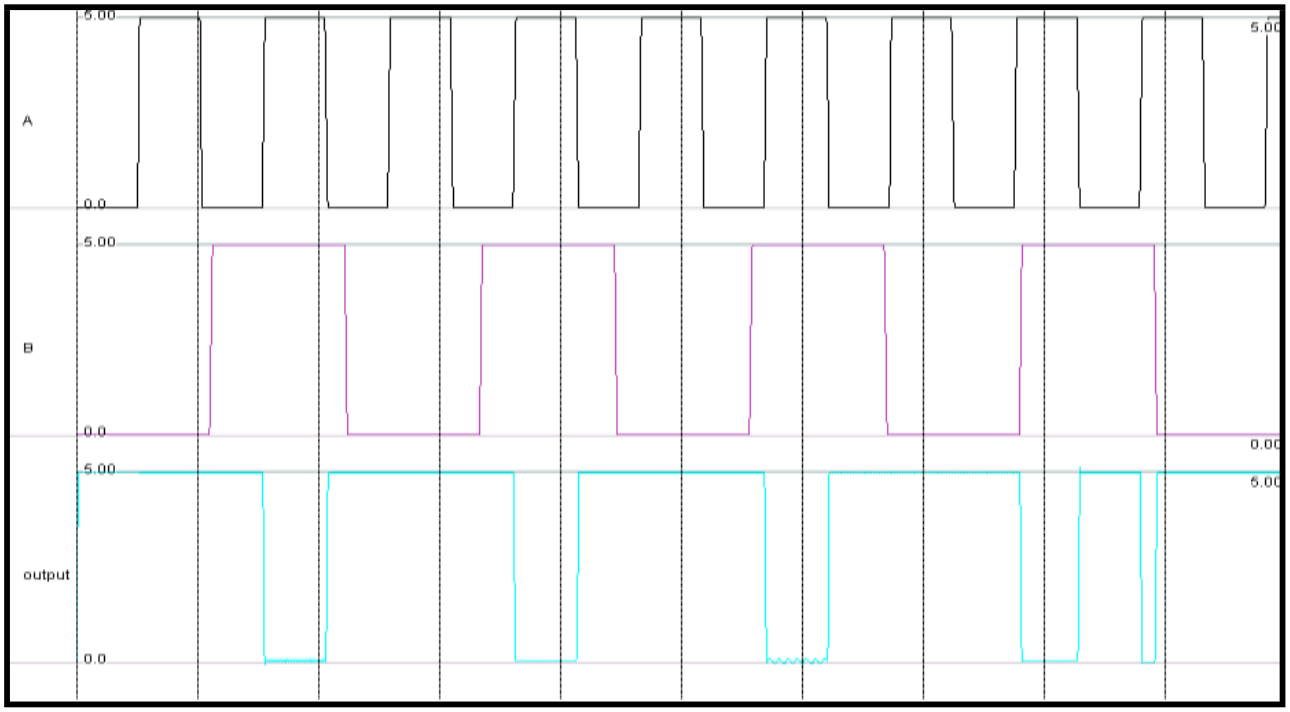
Equipment / Instruments: Laptop or Computer with Microwind Tool.

Basic Theory:

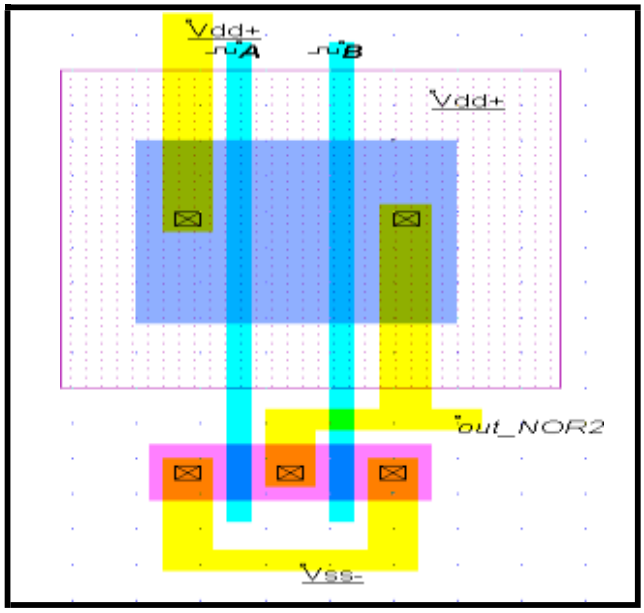
2-input NAND Gate Layout (CMOS 0.12um TECHNOLOGY using Microwind3)



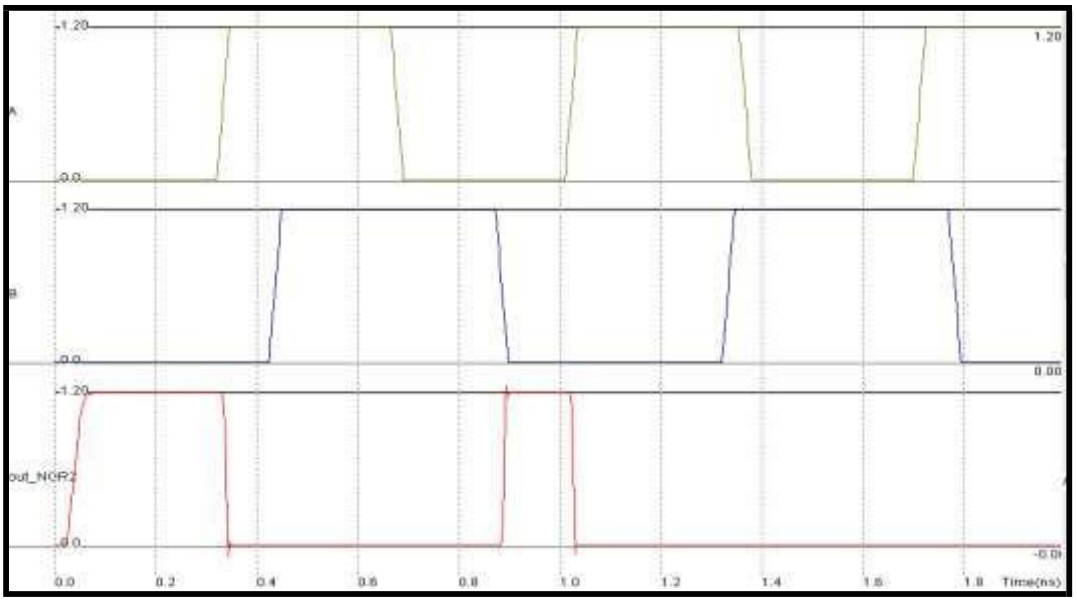
Simulation Waveforms



2-Input NOR Gate Layout (CMOS 0.12um TECHNOLOGY using Microwind3)



Simulation Waveforms



Conclusion:

Suggested Reference:

References used by the students:

Rubric wise marks obtained:

Rubrics	1	2	3	4	5	Total
Marks						