**DEPARTMENT OF ECE**

**WEST GODAVARI INSTITUTE OF SCIENCE & ENGINEERING**

**PRAKASARAOPALEM, AVAPADU, TADEPALLIGUDEM, W.G.DIST, A.P., INDIA**

# VLSI
# LAB MANUAL
Prepared

by

K.SUDHA RANI

Assistant professor

(R-20)

## III-B.Tech :I-Sem
**ELECTRONICS & COMMUNICATION ENGINEERING**

**WEST GODAVARI INSTITUTE OFSCIENCE & ENGINEERING**
Approved by AICTE & Affiliated to JNTU Kakinada.
**Prakasaraopalem, Avapadu, Tadepalligudem, W.G.Dist, A.P., INDIA**

**Department:ECE**

**WEST GODAVARI INSTITUTE OF SCIENCE & ENGINEERING**
**Prakasaraopalem, Avapadu, Tadepalligudem, W.G.Dist, A.P., INDIA**

# INSTITUTE VISION AND MISSION

## Vision of the Institution:

Promote academic excellence, research, innovation, and entrepreneurial skills to produce graduates with human values and leadership qualities to serve the nation

## Mission of the Institution:

Provide student-centric education and training on cutting-edge technologies to make the students globally competitive and socially responsible citizens. Create an environment to strengthen the research, innovation and entrepreneurship to solve societal problems.

**VLSI (Very-Large-Scale Integration) LABORATORY MANUAL**

(R20) III – B. Tech., II-Semester

# Index

**JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY KAKINADA**
**KAKINADA – 533 003, Andhra Pradesh, India**

**DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING**
**III Year – II Semester**

**VLSI DESIGN LAB (R20)**

**List of Experiment**

**PART (A): FPGA Level Implementation (Any Seven Experiments)**

**Note 1:** The students need to develop Verilog /VHDL Source code, perform simulation using relevant simulator and analyze the obtained simulation results using necessary Synthesizer

**Note 2:** All the experiments need to be implemented on the latest FPGA/CPLD Hardware in the Laboratory

1. Realization of Logic gates

**Design and Implementation of the following:**

2. 4-bit ripple carry and carry look ahead adder using behavioral, dataflow and structural modeling
3. a) 16:1 mux through 4:1 mux
   b) 3:8 decoder realization through 2:4 decoder
4. 8:3 encoder
5. 8-bit parity generator and checker
6. Flip-Flops
7. 8-bit synchronous up-down counter
8. 4-bit sequence detector through Mealy and Moore state machines.

**EDA Tools/Hardware Required:**

1. EDA Tool that supports FPGA programming including Xilinx Vivado /Altera (Intel)/ Cypress/Equivalent Industry standard tool along with corresponding FPGA hardware.
2. Desktop computer with appropriate Operating System that supports the EDA tools.

**PART (B): Back-end Level Design and Implementation (Any Five Experiments)**

**Note:** The students need to design the following experiments at schematic level using CMOS logic and verify the functionality. Further students need to draw the corresponding layout and verify the functionality including parasites. Available state of the art technology libraries can be used while simulating the designs using Industry standard EDA Tools.

**Design and Implementation of the following**

1a. Universal Gates
1b. An Inverter
2. Full Adder
3. Full Subtractor
4. Decoder
5. D-Flip-flop

**EDA Tools/Hardware Required:**

Mentor Graphics Software / Cadence/Synopsys/Tanner or Equivalent Industry Standard/CAD Tool.
Desktop computer with appropriate Operating System that supports the EDA tool

# INSTRUCTIONS TO THE STUDENTS

1. Students should come with thorough preparation for the experiment to be conducted.
2. Students should take prior permission from the concerned faculty before availing the leave.
3. Students should come with formals and to be present on time in the laboratory.
4. Students will not be permitted to attend the laboratory unless they bring the practical record fully completed in all respects pertaining to the experiment conducted in the previous class.
5. Students will be permitted to attend laboratory unless they bring the observation book fully completed in all respects pertaining to the experiment conducted in the present class.
6. They should obtain the signature of the staff-in-charge in the observation book after completing each experiment.
7. Practical record and observation book should be maintained neatly.

# Electronics & Communication Engineering

## Course Overview:

This course gives knowledge about the design, analysis, simulation of circuits used as building blocks in Very Large Scale Integration (VLSI) devices. Students can apply the concepts learnt in the lectures towards design of actual VLSI subsystem all the way from specification, modeling, synthesis and physical design. This lab provides hands-on experience on implementation of digital circuit designs using HDL language, which are required for development of various projects and research work.

## Objectives:

The course should enable the students to:

1. The ability to code and simulate any digital function in Verilog HDL.
2. Know the difference between synthesizable and non-synthesizable code.
3. Understand library modeling, behavioral code and the differences between them.
4. Understand the differences between simulator algorithms.
5. Learn good coding techniques per current industrial practices.
6. Understand logic verification using Verilog simulation.

## Course Outcomes:

After completion of the course, the student will be able to:

1. Describe Verilog hardware description languages (HDL).
2. Design Digital Circuits in Verilog HDL.
3. Write behavioral models of digital circuits.
4. Write Register Transfer Level (RTL) models of digital circuits.
5. Verify behavioral and RTL models.
6. Describe standard cell libraries and FPGAs.
7. Synthesize RTL models to standard cell libraries and FPGAs.
8. Implement RTL models on FPGAs and Testing & Verification.

Electronic design automation (EDA) or electronic computer-aided design software (ECAD) designs and develops electronic systems such as printed circuit boards (PCBs) and integrated circuits (ICs). It allows designers to build out different alternatives and options and compare them to each other. It also generates manufacturing documentation as part of the specification used to source, fabricate, and produce PCBs.

The rapidly growing EDA industry is best understood by looking at the definition of EDA.

**Electronics** includes anything electronic, from computer chips and cell phones to controls for automobiles, etc. Everything made by the electronics industry results from designers using EDA tools and services.

**Design** is the part of the production cycle where creativity, ingenuity, and new ideas are most valued. Designers build models to understand the behavior and complex interactions of millions of constituent parts in their designs to ensure completeness, correctness, and manufacturability of the final product. Many of the designers in this field include electrical and software engineers.

**Automation** demonstrates the increasing complexity in the electronics industry today. This complexity is enabled by Moore's Law (which states that the number of transistors in integrated circuits doubles every 18 months), which drives the need for automation. Engineers need to validate their concepts, model and analyze their designs, and identify and eliminate problems before making production commitments.EDA helps ensure correct designs.

# Very Large Scale Integration (VLSI)

VLSI is the process of creating an integrated circuit (IC) by combining thousands of transistors into a single chip. VLSI began in the 1970s when complex semiconductor and communication technologies were being developed. Before the introduction of VLSI technology most ICs had a limited set of functions they could perform.

The functionality of electronics equipment’s and gadgets has achieved a phenomenal while their physical sizes and weights have come down drastically. The major reason is due to the rapid advances in integration technologies, which enables fabrication of millions of transistors in a single Integrated Circuit (IC) or chip. IC is a device having multiple transistors with interconnects manufactured on a single silicon substrate. Integration with a complexity of 10’s of transistors is called Small Scale Integration, with 100’s is Medium Scale Integration (MSI), with 1000’s is Large Scale Integration (LSI), with 10,000 it is Very Large Scale Integration (VLSI) Systems of systems can

be implemented in a VLSI IC. However, with this rise in functionality of VLSI ICs, design problem has become huge and complex.

To address this complexly issue, after the design specifications are complete almost all the other steps are automated using CAD tools. However, even designs automated using CAD tools may

have bugs. Also, due to extremely large size of the design space it is not possible to verify correctness of the design under all possible situations. So technique are required that can verify, without exercising exhaustive input-output combinations, that the design meets all the input specifications; this technique is called formal verification. In VLSI designs millions of transistors are packed into a single chip. This leads to manufacturing defects and all the chips need to be physically tested by giving input signals from a pattern generator and comparing responses using a logic analyzer; this process is called Testing. So, in the process of manufacturing a VLSI IC there are three broad steps: **Design-Verification-Test**.

VLSI ICs can be divided into analog, digital or mixed-signal (both analog and digitalon the same chip) based on their functionality.

- Digital ICs can contain logic gates, flip-flops, multiplexers. Work using binary mathematics to process "one" and "zero" signals.
- Analog ICs, such as current mirrors, voltage followers, filters, OPAMPs etc. work by processing continuous signals.
- When single IC has both analog and digital components it is called mixed signal IC e.g, Analog to Digital Converter (ADC).

The automation algorithms and CAD tools are mainly available for digital ICs because transformation of design specifications to silicon implementation can be accomplished using logical procedures (which can be converted to algorithms and tools). However, most of the analog circuits design is like an “art” which is best performed by designers with “aid” of some CAD tools (which provides feedback to designer if the manual design is progressing fine etc.)

## VLSI Design flow

The VLSI IC circuits design flow is shown in the figure below.

- Specifications comes first, they describe abstractly the functionality, interface, and the architecture of the digital IC circuit to be designed.
- Architectural design is then created to analyze the design in terms of functionality, performance, compliance to given standards, and other specifications.
- RTL Description is done using HDLs. This RTL Description is simulated to test functionality. From here onwards we need the help of EDA tools.
- RTL Description is then converted to a gate-level netlist using logic synthesis tools. A gate-level net list is a Description of the circuit in terms of gates and connections between them, which are made in such a way that they meet the timing, power and area specifications.
- Finally a physical layout is made, which will be verified and then sent to fabrication.

The Figure provides a more simplified view of the VLSI design flow, taking into account the various representations, or abstractions of design - behavioral logic, circuit and mask layout. Note that the verification of design plays a very important role in every step during this process. The failure to properly verify a design in its early phases typically causes significant and expensive re-design at a later stage, which ultimately increases the time-to-market.

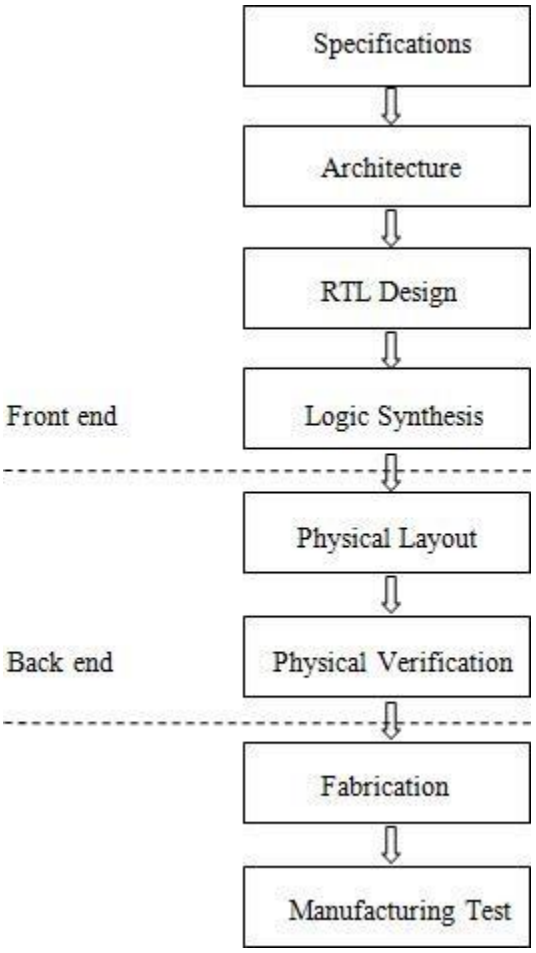



Figure 1 VLSI Design Flow

In the following, we will examine design methodologies and structured approaches which have been developed over the years to deal with both complex hardware and software projects. Regardless of the actual size of the project, the basic principles of structured design will improve the prospects of success. Some of the classical techniques for reducing the complexity of IC design are: Hierarchy, regularity, modularity and locality.

# DESIGN STYLES

In 1980s when industry observed the possibility of automating the VLSI physical design using CAD tools, a new design methodology has been introduced. This new design methodology was called semi-custom VLSI design, where the design on silicon is customized as per the required application, reducing the design time and cost involved.
In comparison with full custom VLSI where the complete layout will be hand drawn and every cell is designed as per the requirements the semi-custom has the following advantages.

- Separated design approach, front end and back end
- Reduced cost as the basic cells are reused
- Less design turnaround time.

In today ASIC industry the design is portioned into front end and back end as explained below.

1. Front end

a. Enter the design in one standard format (which EDA tools can understand)
b. Analyzing the requirements and high level design (identifying various blocksin design)
c. RTL design evolving the necessary micro architecture for the each block
d. VHDL, Verilog, other HDLs, Netlist etc.
e. Developing necessary test benches for functional verification.
f. Simulation and model verification using standard simulators
g. Integration of all the blocks and top level simulation.

2. Back end

a. Synthesizing the design, fixing any bugs (if any part of code is notsynthesizable)
b. Floor planning as the targeted silicon area
c. Invoking the ASIC back end tools (Mapping extracted Netlist cells totechnologyspecific cells)
d. Place and root as per the required timing and clock constraints
e. Extraction of models from synthesis outputs
f. Timing simulation and functional verification
g. Sending the design to the FAB and getting the chip manufactured

# Introduction to HDL

This section is a brief introduction to hardware design using a Hardware Description Language (HDL). A language describing hardware is quite different from C, Pascal, or other software languages. A computer program is dynamic, i.e., sharing the same resources, allocating resources when needed and not always optimized for maximum speed, optimal memory management, or lowest resource requirements. The main focus is functionality, but it is still not uncommon that software programs can behave quite unexpected. When problems arise, new versions of the programs are distributed by the vendor, usually with a new version number and a higher price tag. The demands on hardware design are high compared to software. Often it is not possible, or at least very tricky, to patch hardware

after fabrication. Clearly, the functionality must be correct and in addition how the code is written will affect the size and speedof the resulting hardware. Each mm2 of a chip costs money, lots of money. The amount of logic cells, memory blocks and input/output connections will affect the sizeof the design and therefore also the manufacturing cost. A software designer using a HDL has to be careful. The degrees of freedom compared with software design have dramatically increased and must be taken into account.

Hardware description languages such as Verilog differ from software programming languages because they include ways of describing the propagation time and signal strengths (sensitivity). There are two types of assignment operators; a blocking assignment (=), and a non-blocking (<=) assignment. The non-blocking assignment allows designers to describe a state-machine update without needing to declare and use temporary storage variables. Since these concepts are part of Verilog's language semantics, designers could quickly write descriptions of large circuits in a relatively compact and concise form. At the time of Verilog's introduction (1984), Verilog represented a tremendous productivity improvement for circuit designers who were already using graphical schematic capture software and specially written software programs to document and simulate electronic circuits.

The designers of Verilog wanted a language with syntax similar to the C programming language, which was already widely used in engineering software development. Like C, Verilog is case-sensitive and has a basic preprocessor (though less sophisticated than that of ANSI C/C++). Its control flow keywords (if/else, for, while, case, etc.) are equivalent, and its operator precedence is compatible with C. Syntactic differences include: required bit-widths for variable declarations, demarcation of procedural blocks (Verilog uses begin/end instead of curly braces {}), and many other minor differences. Verilog requires that variables be given a definite size. In C these sizes are assumed from the 'type' of the variable (for instance an integer type may be 8 bits).

A Verilog design consists of a hierarchy of modules. Modules encapsulate design hierarchy, and communicate with other modules through a set of declared input, output, and bidirectional ports. Internally, a module can contain any combination of the following: net/variable declarations (wire, reg, integer, etc.), concurrent and sequential statement blocks, and instances of other modules (sub-hierarchies). Sequential statements are placed inside a begin/end block and executed in sequential order within the block. However, the blocks themselves are executed concurrently, making Verilog a dataflow language.

Verilog's concept of 'wire' consists of both signal values (4-state: "1, 0, floating, undefined") and signal strengths (strong, weak, etc.). This system allows abstract modeling of shared signal lines, where multiple sources drive a common net. When a wire has multiple drivers, the wire's (readable) value is resolved by a function of the source drivers and their strengths.

A subset of statements in the Verilog language is synthesizable. Verilog modules that conform to a synthesizable coding style, known as RTL (register-transfer level), can be physically realized by synthesis software. Synthesis software algorithmically transforms the (abstract) Verilog source into a netlist, a logically equivalent description consisting only of elementary logic primitives (AND, OR, NOT, flip- flops, etc.) that are available in a specific FPGA or VLSI technology. Further manipulations to the netlist ultimately lead to a circuit fabrication blueprint (such as a photo mask set for an ASIC or a bitstream file for an FPGA). **HDL simulators are better than gate level simulators** for 2 reasons: portable modeldevelopment, and the ability to design complicated test benches that react to outputs from the model under test. Finding

model for a unique component for your particular gate level simulator can be a frustrating task; with an HDL language you can always write your own model. Also most gate level simulators are limited to simple waveform based test benches which complicate the testing of bus and microprocessor interface circuits.

- **Verilog** is a great low level language. Structural models are easy to design and Behavioral RTL code is pretty good. The syntax is regular and easy to remember. It is the fastest HDL language to learn and use. However Verilog lacks user defined data types and lacks the interface-object separation of the VHDL's entity- architecture model.

- **VHDL** is good for designing behavioral models and incorporates some of the modern object oriented techniques. It's syntax is strange and irregular, and the language is difficult to use. Structural models require a lot of code that interferes with the readability of the model.

# Xilinx Manual:

### 1. Introduction

Xilinx Tools is a suite of software tools used for the design of digital circuitsimplemented using Xilinx *Field Programmable Gate Array (FPGA) or Complex Programmable Logic Device (CPLD)*. The design procedure consists of (a) design entry, (b) synthesis and implementation of the design, (c) functional simulation and d)testing and verification. Digital designs can be entered in various ways using the above CAD tools: using a schematic entry tool, using a hardware description language (HDL) – Verilog or VHDL or a combination of both. In this lab we will only use the design flow that involves the use of VerilogHDL.

The CAD tools enable you to design combinational and sequential circuits starting with Verilog HDL design specifications. The steps of this design procedure are listedbelow:

1. Create Verilog design input file(s) using template driveneditor.
2. Compile and implement the Verilog designfile(s).
3. Create the test-vectors and simulate the design (functional simulation) withoutusing aPLD (FPGA orCPLD).
4. Assign input/output pins to implement the design on a targetdevice.
5. Download bitstream to an FPGA or CPLDdevice.
6. Test design on FPGA/CPLDdevice

A Verilog input file in the Xilinx software environment consists of the following segments:

1. ***Header:*** module name, list of input and output ports.
2. ***Declarations:*** input and output ports, registers and wires.
3. ***Logic Descriptions:*** equations, state machines and logic functions.
4. ***End:*** end module

## *2.* Creating *a NewProject*

Xilinx Tools can be started by clicking on the Project Navigator Icon on the Windows desktop. This should open up the Project Navigator window on your screen. This window shows (see Figure 1) the last accessed project.

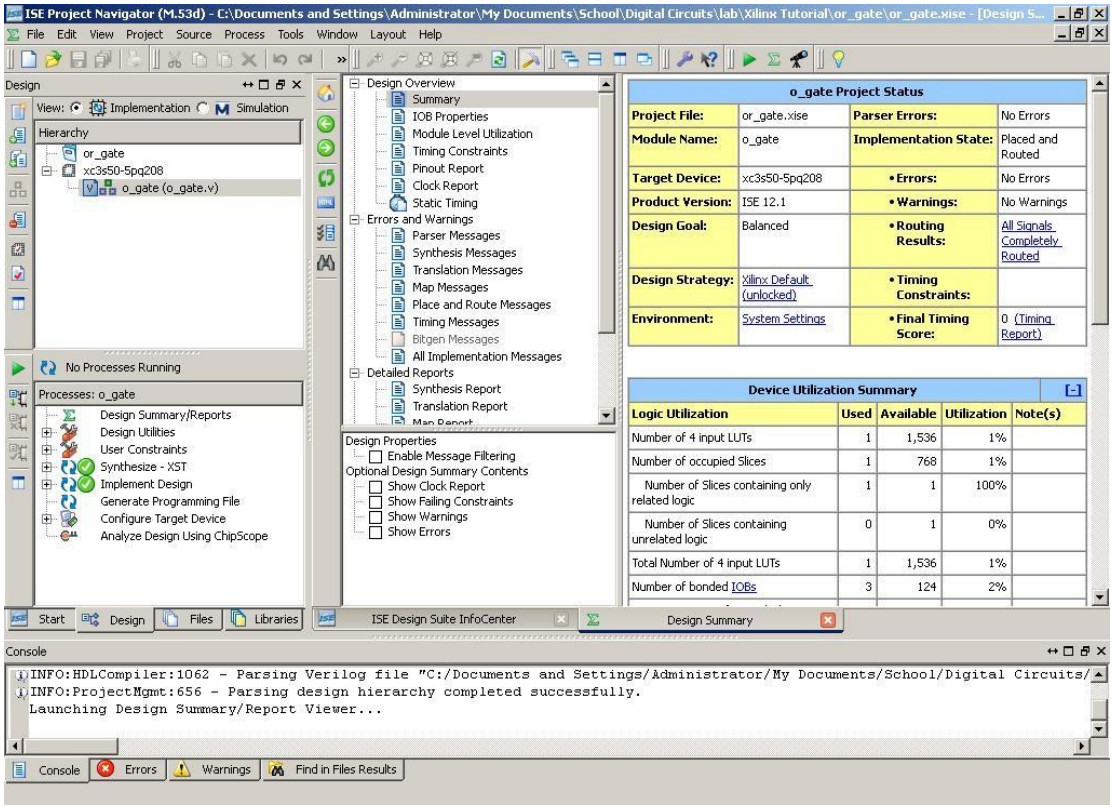



Figure 2: Xilinx Project Navigator window (snapshot from Xilinx ISE software)

### *Openinga* project

Select **File->New Project** to create a new project. This will bring up a new project window (Figure 2) on the desktop. Fill up the necessary entries as follows:

- **ProjectName**: Write the name of your newproject
- **Project Location**: The directory where you want to store the new project (Note: DO NOT specify the project location as a folder on Desktop or a folder in the Xilinx\bin directory. Your H: drive is the best place to put it. **The project location path is NOT to have any spaces in it eg: C:\Nivash\TA\new lab\sample exercises\o_gate is NOT to be used**) Leave the top level module type as HDL.Example: If the project name were "o_gate", enter "o_gate" as the project name andthen click "Next".

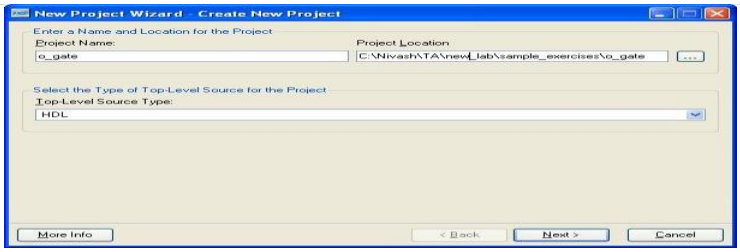



Figure 2: New Project Initiation window (snapshot from Xilinx ISE software)

Clicking on NEXT should bring up the following window:

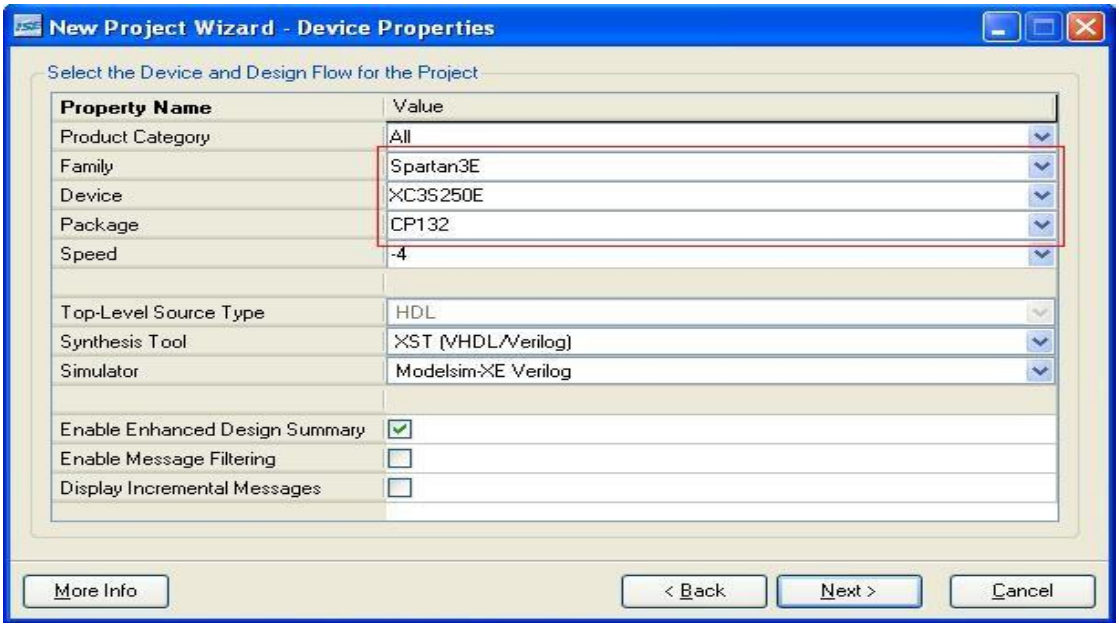



Figure 3: Device and Design Flow of Project (snapshot from Xilinx ISEsoftware)

For each of the properties given below, click on the '**value**' area and select from thelist of values thatappear.

- **Device Family**: Family of the FPGA/CPLD used. In this laboratory we will beusing the Spartan3EFPGA's.
- **Device**: The number of the actual device. For this lab you may enter **XC3S250E**
- (this can be found on the attached prototyping board)
- **Package**: The type of package with the number of pins. The Spartan FPGA used inthis lab is packaged in CP132package.
- **Speed Grade**: The Speed grade is"-4".
- **Synthesis Tool**: **XST**[VHDL/Verilog]
- **Simulator:** The tool used to simulate and verify the functionality of the design. Modelsim simulator is integrated in the Xilinx ISE. Hence choose "Modelsim-XE Verilog" as the simulator or even Xilinx ISE Simulator can beused.
- Then click on **NEXT** to save the entries.

All project files such as schematics, netlists, Verilog files, VHDL files, etc., will be stored in a subdirectory with the project name. A project can only have one top level HDL source file (or schematic). Modules can be added to the project to create a modular, hierarchical design.

In order to open an existing project in Xilinx Tools, select **File->Open Project** to show the list of projects on the machine. Choose the project you want and click **OK**.

Clicking on NEXT on the above window brings up the following window:

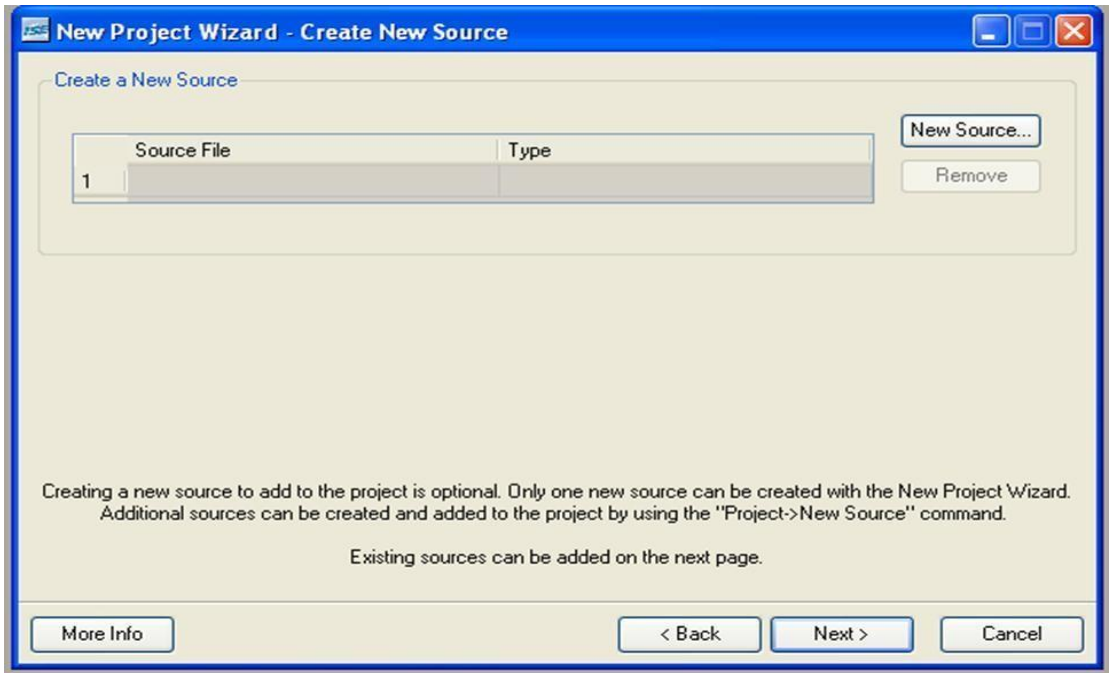



Figure 4: Create new source window (snapshot from Xilinx ISE software)

If creating a new source file, Click on the NEW SOURCE.

***Creating a Verilog HDL input file for a combinational logic design***

In this lab we will enter a design using a structural or RTL description using the Verilog HDL. You can create a Verilog HDL input file (**.v** file) using the HDL Editor available in the Xilinx ISE Tools (or any text editor).

In the previous window, click on the NEW SOURCE

A window pops up as shown in Figure 4. (*Note*: "***Add to project***" *option is selected by default. If you do not select it then you will have to add the new source file to the project manually*.)

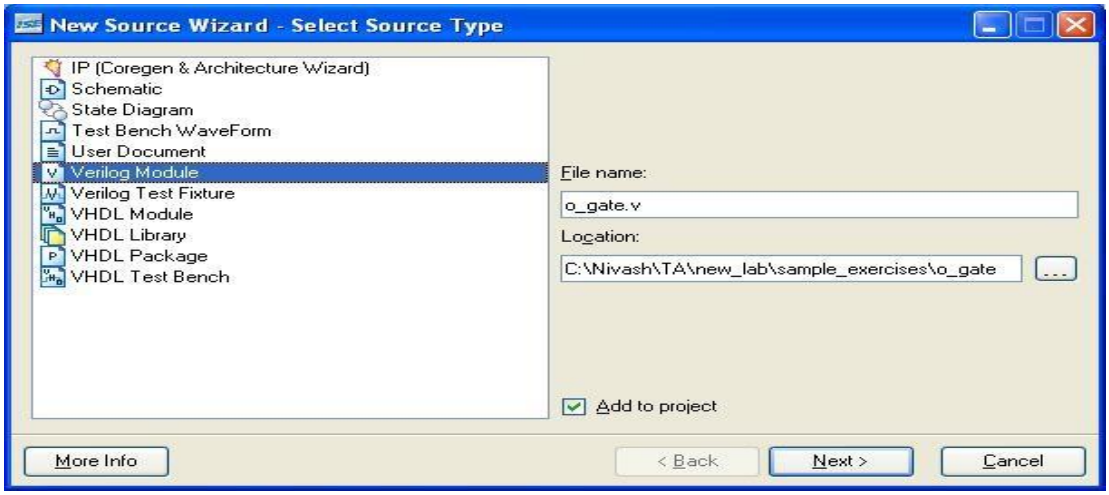



Figure 5: Creating Verilog-HDL source file (snapshot from Xilinx ISE software)

Select Verilog Module and in the "File Name:" area, enter the name of the Verilog source file you are going to create. Also make sure that the option Add to project is selected so that the source need not be added to the project again. Then click on Next to accept the entries. This pops up the following window (Figure 5).

In the **Port Name** column, enter the names of all input and output pins and specify the **Direction** accordingly. A Vector/Bus can be defined by entering appropriate bit numbers in the **MSB/LSB** columns. Then click on **Next>**to get a window showing all the new source information (Figure 6). If any changes are to be made, just click on **<Back** to go back and make changes. If everything is acceptable, click on **Finish**

**> Next > Next > Finish** tocontinue.

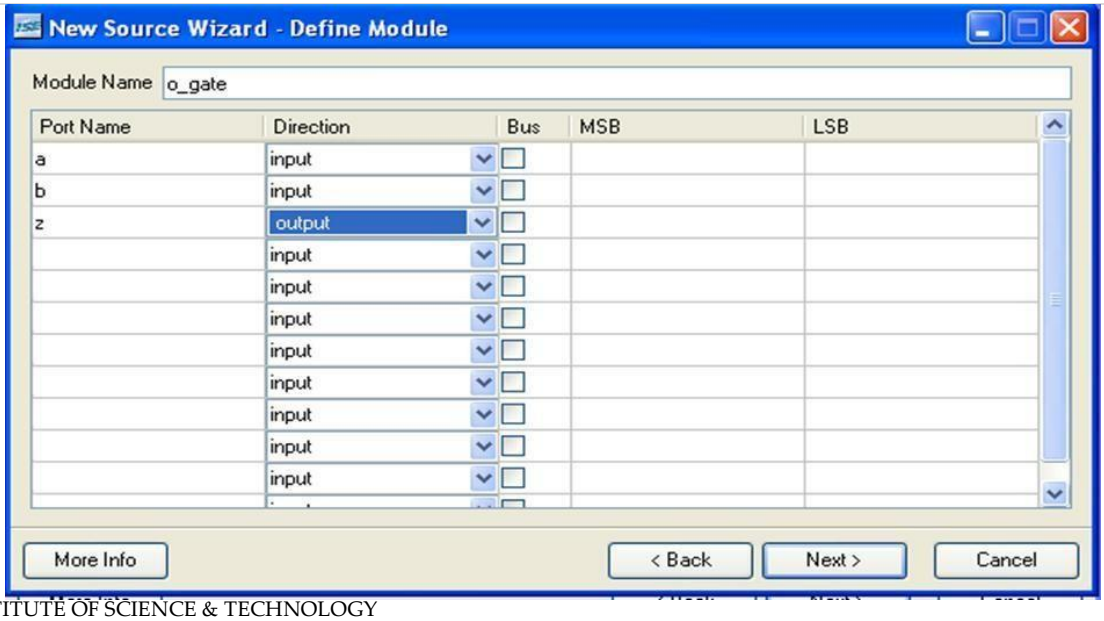


Figure 6: Define Verilog Source window (snapshot from Xilinx ISE software)

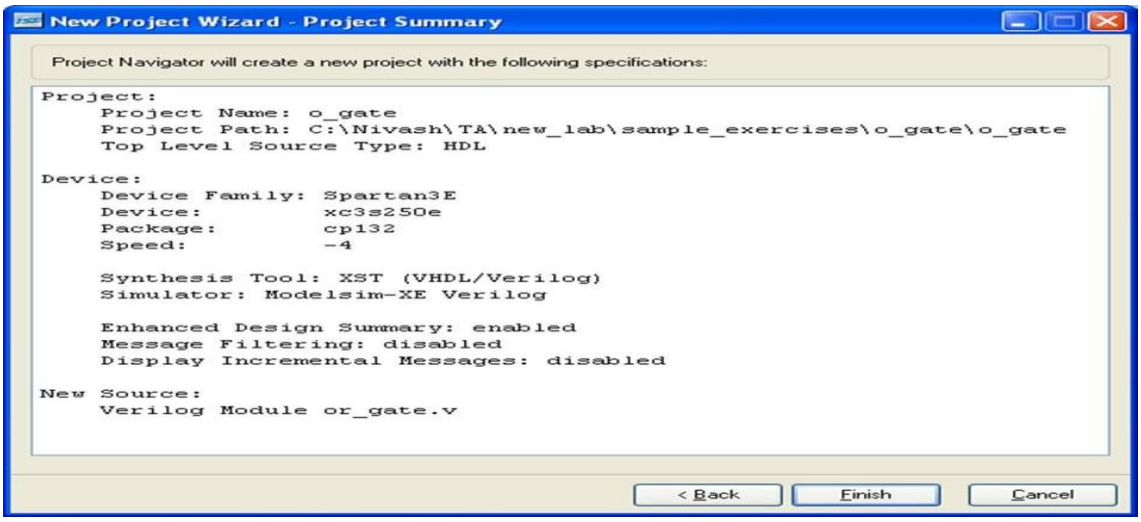



Figure 7: New Project Informationwindow(snapshot from Xilinx ISE software)

Once you click on **Finish**, the source file will be displayed in the sources window in the **Project Navigator** (Figure 1).

If a source has to be removed, just right click on the source file in the **Sources in Project window** in the **Project Navigator** and select **Remove**in that. Then select **Project -> Delete Implementation Data** from the Project Navigator menu bar to remove any relatedfiles.

**Editing the Verilog source *file***

The source file will now be displayed in the **Project Navigator** window (Figure 8). Thesource. Filewindowcanbeusedasatexteditortomakeanynecessarychangestothesourcefile.Allthe input/output pins will be displayed. Save your Verilog program periodically by selectingthe **File->Save** from the menu. You can also edit Verilog programs in any text editor andadd them to the project directory using "Add Copy Source".

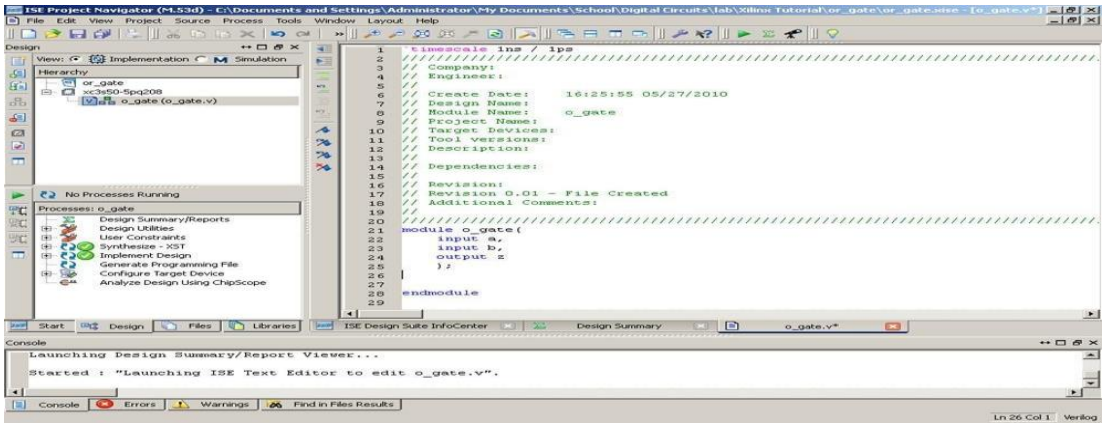



Figure 8: Verilog Source code editor window in the Project Navigator (fromXilinx ISE software)

## *3.* Functional Simulation of Combinational Designs

### Adding the test vectors

To check the functionality of a design, we have to apply test vectors and simulate the circuit. In order to apply test vectors, a test bench file is written. Essentially it will supply all the inputs to the module designed and will check the outputs of the module. Example: For the 2 input OR Gate, thesteps to generate the test bench are as follows:

In the Sources window (top left corner) right click on the file that you want to generate the test bench for and select 'New Source'

Provide a name for the test bench in the file name text box and select 'Verilog test fixture' among the file types in the list on the right side as shown in figure 9.

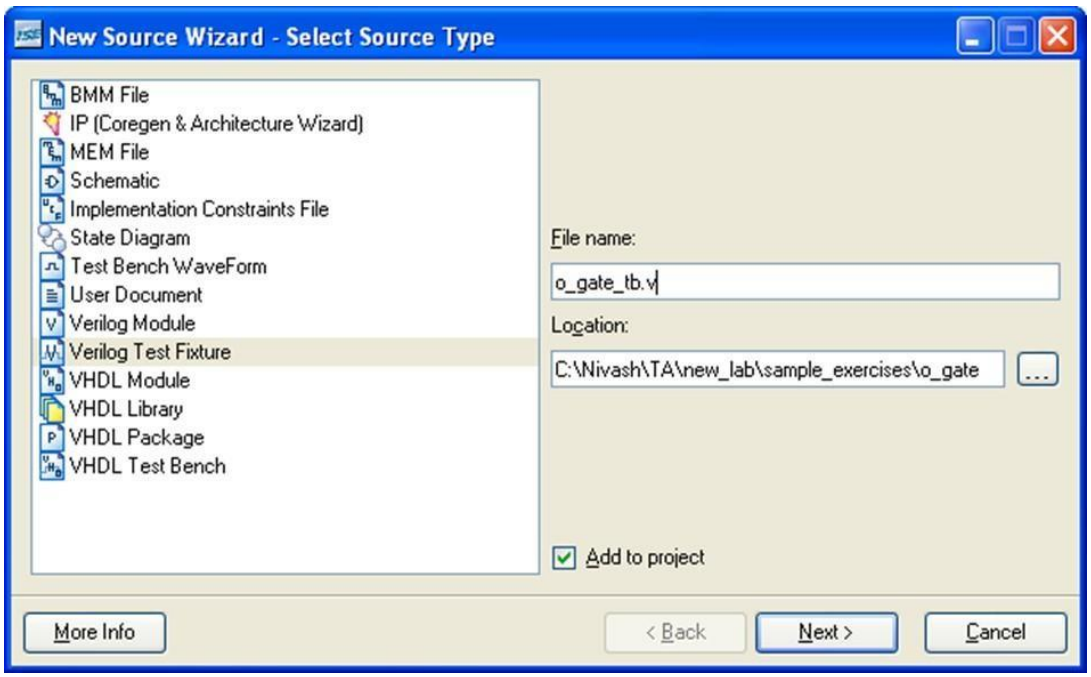



Figure 9: Adding test vectors to the design (snapshot from Xilinx ISE software)

Click on 'Next' to proceed. In the next window select the source file with which you wantto associate the test

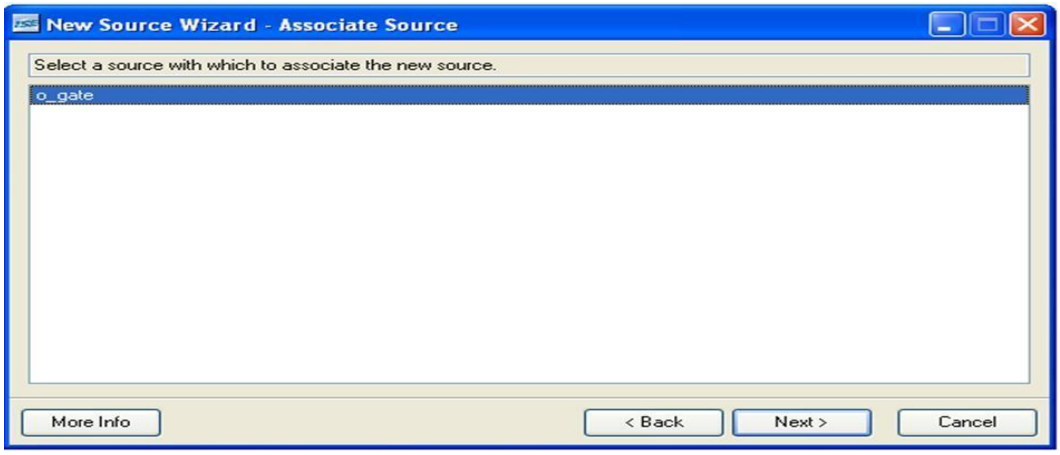
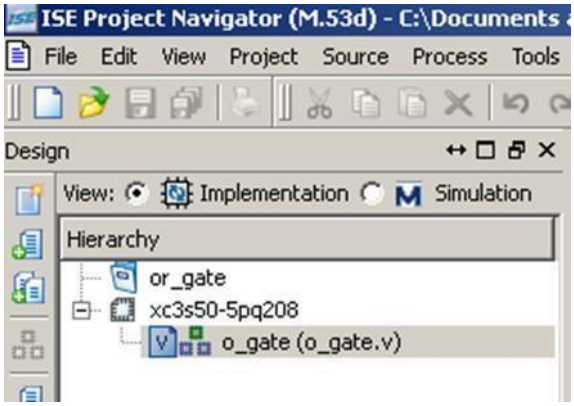



Figure 10: Associating a module to a testbench (snapshot from Xilinx ISEsoftware)

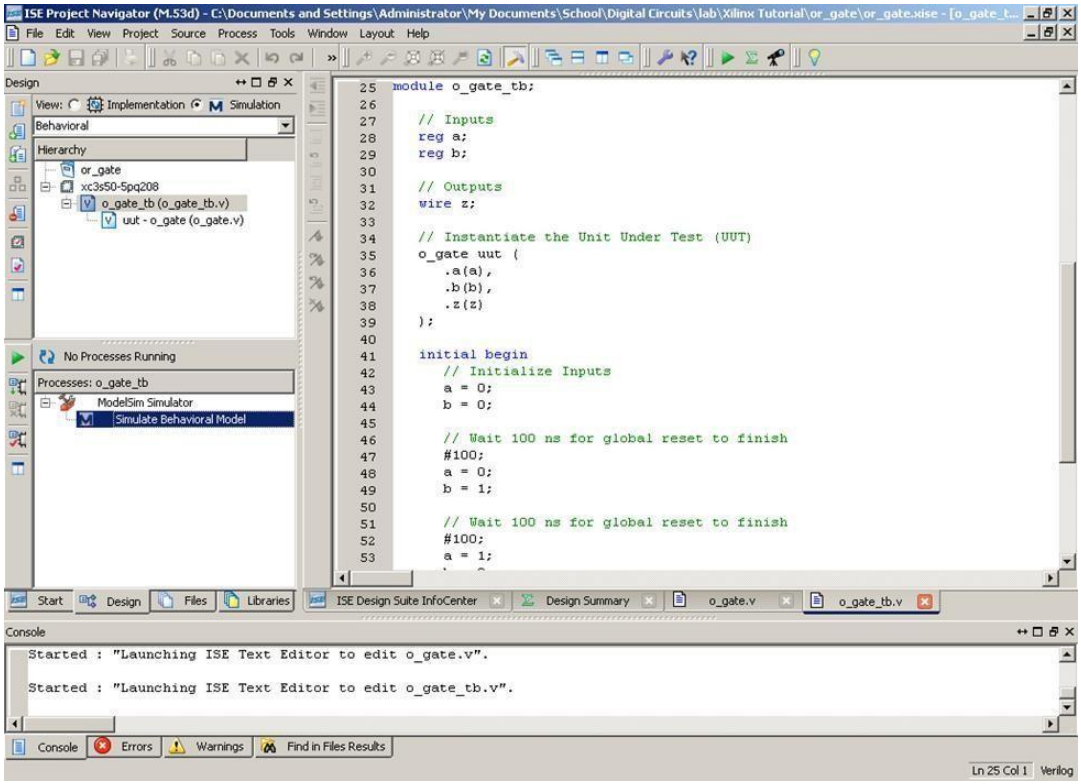
Click on Next to proceed. In the next window click on Finish. You will now be provided with a template for your test bench. If it does not open automatically click the radio button next to **Simulation .**




You should now be able to view your test bench template.

**Simulating and Viewing the OutputWaveforms**

Now under the **Processes window** (making sure that the test bench file in the **Sources window** is selected) expand the **ModelSim simulator Tab** by clicking on the add sign next to it. Double Click on **Simulate Behavioral Model**. You will probably receive a complier error. This is nothing to worry about – answer "No" whenasked if you wish to abort simulation. This should cause ModelSim to open. Wait for it to complete execution. If you wish to not receive the compiler error, right click on **Simulate Behavioral Model** and select process properties. Mark the checkbox next to "Ignore Pre- Complied Library Warning Check".




```
25  module o_gate_tb;
26
27    // Inputs
28    reg a;
29    reg b;
30
31    // Outputs
32    wire z;
33
34    // Instantiate the Unit Under Test (UUT)
35    o_gate uut (
36       .a(a),
37       .b(b),
38       .z(z)
39    );
40
41    initial begin
42       // Initialize Inputs
43       a = 0;
44       b = 0;
45
46       // Wait 100 ns for global reset to finish
47       #100;
48       a = 0;
49       b = 1;
50
51       // Wait 100 ns for global reset to finish
52       #100;
53       a = 1;
```

Figure 11: Simulating the design (snapshot from Xilinx ISE software)

**Saving the simulation results**

To save the simulation results, Go to the waveform window of the Modelsim simulator, Click on File -> Print to Postscript -> give desired filename and location.
Notethatbydefault,the waveform is“zoomed in”tothenanosecondlevel.Use the zoom controls to display the entire waveform.

Else a normal print screen option can be used on the waveform window and subsequentlystored in Paint.

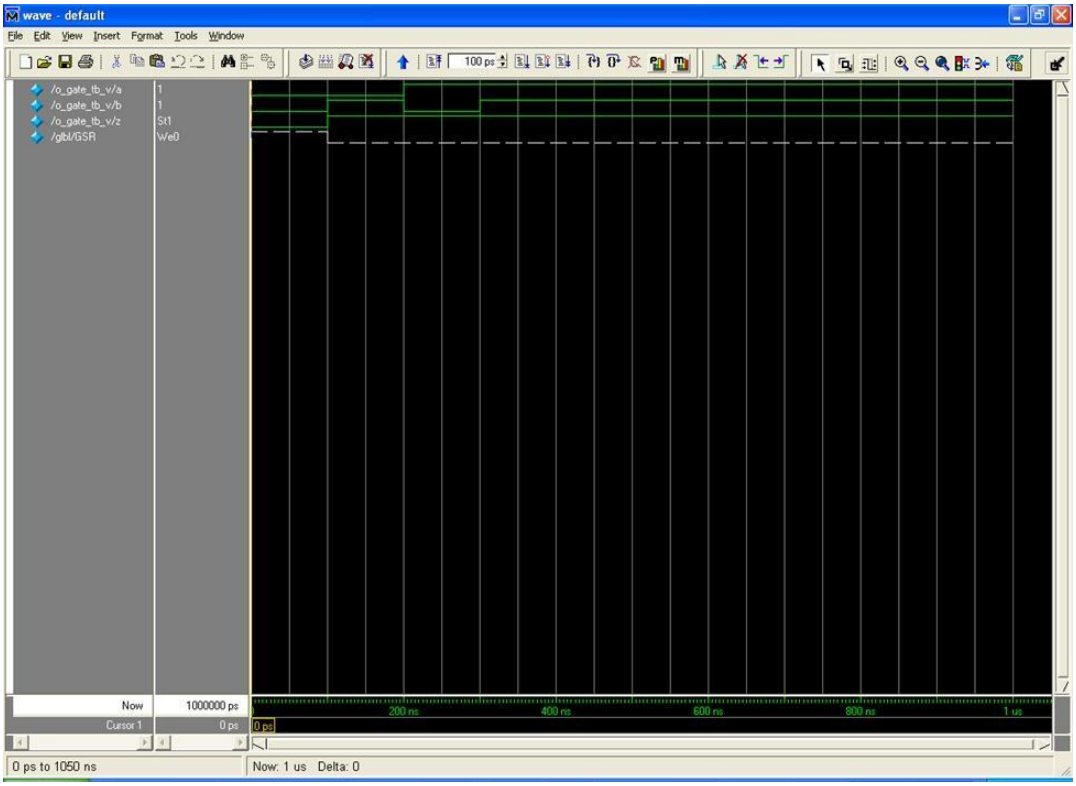



Figure 12: Behavioral Simulation output WaveformSnapshot fromModelSim)

For taking printouts for the lab reports, convert the black background to white in Tools ->Edit Preferences. Then click Wave Windows -> Wave Background attribute.

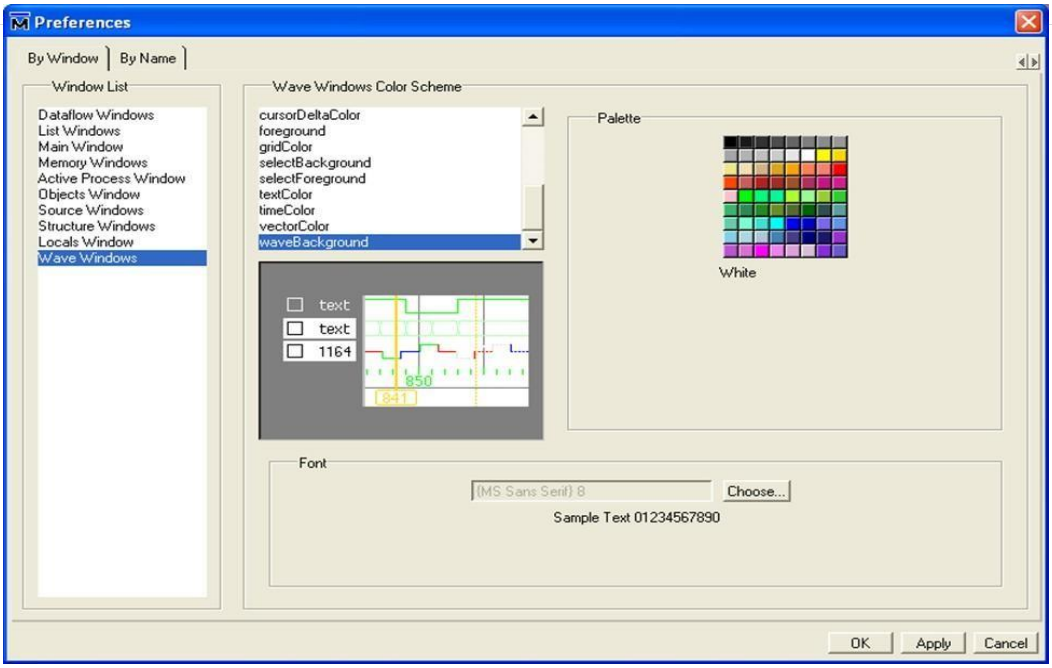



Figure 13: Changing Waveform Background in ModelSim

## Synthesis and Implementation of the Design

The design has to be synthesized and implemented before it can be checked for correctness, by running functional simulation or downloaded onto the prototyping board. With the top- level Verilog file opened (can be done by double-clicking that file) in the HDL editor window in the right half of the Project Navigator, and the view of the project being in the **Module view** , the **implement design** option can be seen in the **process view**. **Design entry utilities** and **Generate Programming File** options can also be seen in the process view. The former can be used to include user constraints, if any and the latter will be discussed later.

To synthesize the design, double click on the **Synthesize Design** option in the Processes window.

To implement the design, double click the **Implement design** option in the **Processes window**. It will go through steps like **Translate, Map and Place & Route**. If any of these steps could not be done or done with errors, it will place a **X** mark in front of that, otherwise a tick mark will be placed after each of them to indicate the successful completion. If everything is done successfully, a tick mark will be placed before the **Implement Design** option. If therearewarnings, one can see mark in front of the option indicating that there are some warnings. One can look at the warnings or errors in the**Console** window present at the bottom of the Navigator window. ***Every time the design file is saved; all these marks disappear asking for a fresh compilation***.

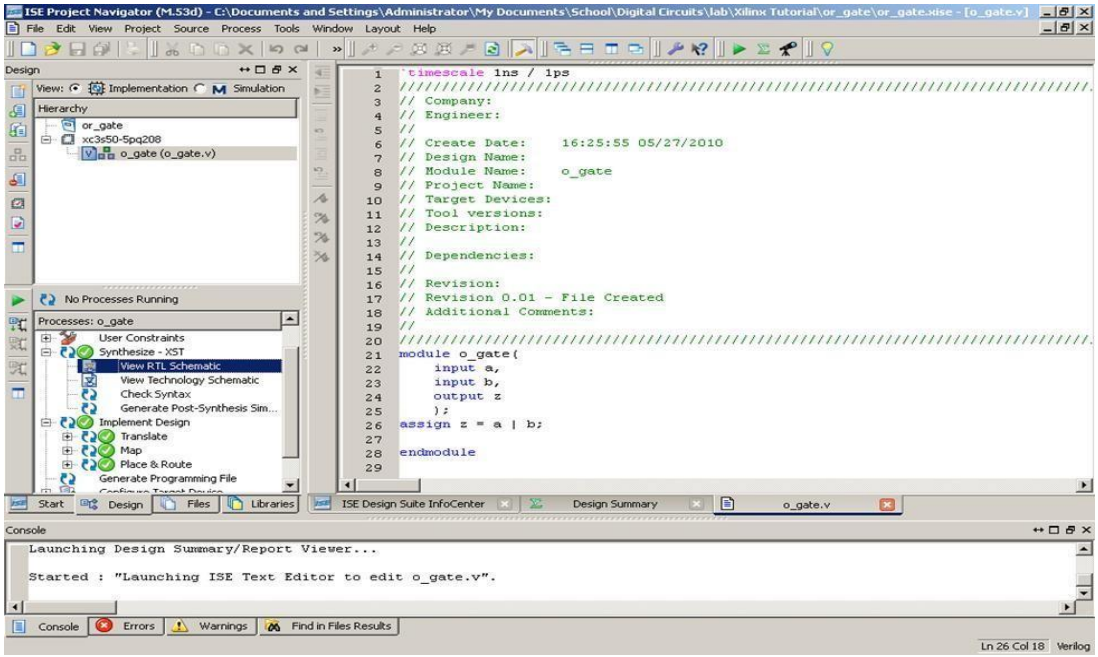



Figure 14: Implementing the Design (snapshot from Xilinx ISE software)

By double clicking it opens the top level module showing only input(s) and output(s) as shown below.

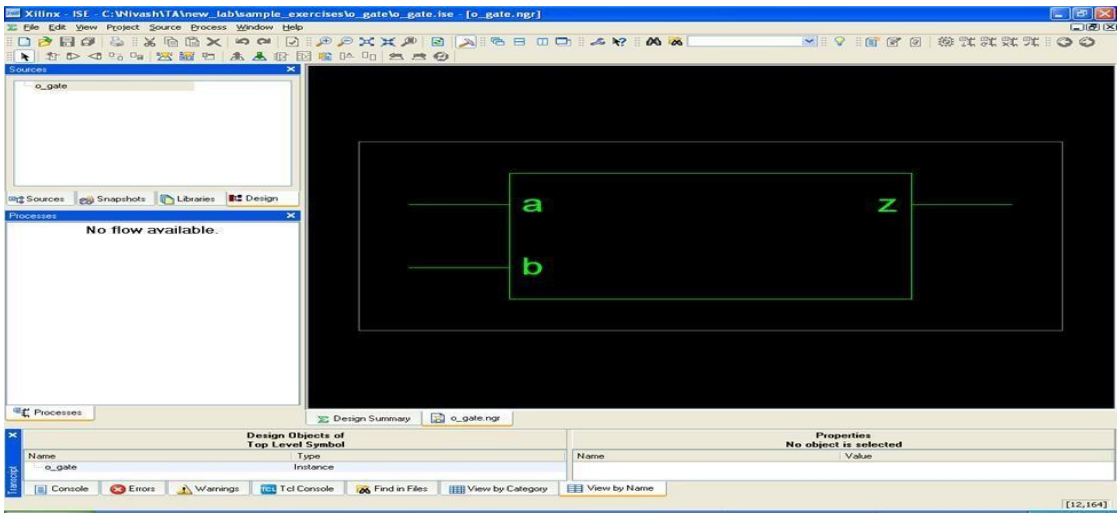



Figure 15: Top Level Hierarchy of the design

By double clicking the rectangle, it opens the realized internal logic asshown below.

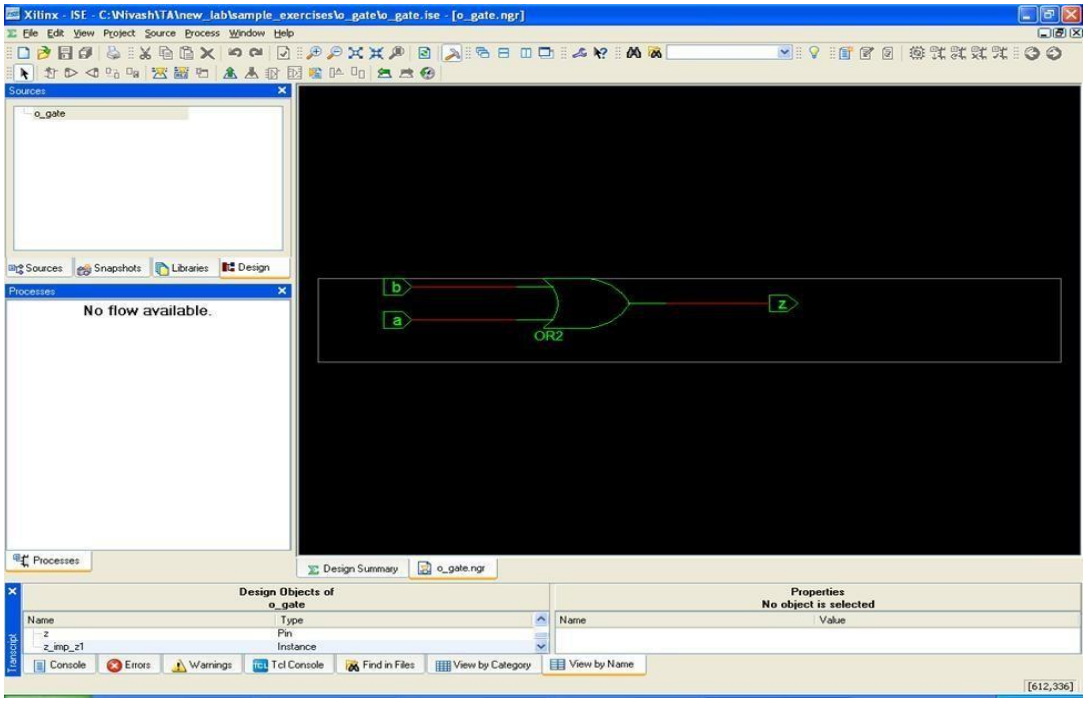



Figure 16: Realized logic by the XilinxISE for the verilog code

# EXPERIMENT- 1

**Design and Simulation of** Realization of Logic gates **using all the modeling styles and Synthesis of all the logic gates using Verilog HDL**

**Aim:** To Implement and verify the functionality of AND gate using Xilinx ISE

**Apparatus required: Electronics Design Automation Tools used**

- Xilinx ISE Simulator tool
- Xilinx XST Synthesis tool
- Xilinx Spartan FPGA Board
- JTAG cable
- Adaptor 5v/4A

**Theory:**

**Logic gates:**

A logic gate is an idealized or physical device implementing a Boolean function; that is, it performs a logical operation on one or more binary inputs, and produces a single binary output.

**Procedure:**

1. Double click on Xilinx Design Suite Icon.

2. Select new project in file menu. 3. Enter the project name and location as shown below and press Next .

4. Select the Family, Device, Package and speed as per the requirements and press Next

5. Create a new source by using new source icon or right click on the device/project folder to create new source.

6. Select the appropriate source type and enter the file name in New Source Wizard window and press Next .

7. Enter the architecture name – dataflow/behavioral/structural, port name and select the direction. This will create .v source file. Press Next and finish the initial project creation.

8. Write complete VHDL/Verilog code implementation and save.

9. Click on implementation and check for syntax using "Check syntax" option under synthesize tab. If any error, edit and correct VHDL/Verilog code and repeat check syntax until zero errors.

10. Double click on ISIM simulator by selecting simulation mode to complete the functional simulation of your design.

**Part –B Hardware Interfacing Procedure :**

1. Repeat the steps 1 to 10 from the procedure for software experiments .

2. Make the connection between appropriate FRC's of the FPGA board and the DIP switch connector of the GPIOcard-2/

3. Make the connection between appropriate FRC's of the FPGA board and the LED connector of the GPIOcard-2.

4. Right click on the device and select "New Source", Select the option "Implementation constraint File" and provide the file name and click on next and then hit Finish. This creates an .ucf file.

5. Double click on the added .ucf file and assign the pin numbers to inputs and outputs referring to FRC sheet using the syntax as shown.Save the constraint file.

6. Connect USB programmer for FPGA between FPGA kit and USB port of your computer.

7. Go to process window, select the VHDL or Verilog file and click on "configure target device".

8. Click OK for the warning below.

9. Select boundary scan to impact the target device.

10. Right click on the impact window to establish a connection between system and FPGA by selecting "INTIALIZE CHAIN" option.

11. Both prom device and FPGA device gets identified after step 10 and bypass the procedure to select only FPGA which of main interest.

12. Now choose device 2(FPGA XC3S400) and hit ok to complete the impact.

13. Now right click on the device to assign a new .bit file by selecting an option "ASSIGN NEW CONFIGURATION FILE".

**Boolean equations:**

AND Gate:Y = (A.B)

OR Gate: Y = (A + B)

NAND Gate: Y = (A.B)'

NOR Gate: Y = (A+B)'

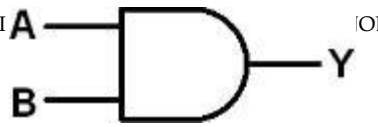
XOR Gate: Y = A.B' + A'.B

XNOR Gate: Y = A.B + A'.B'

NOT gate: Y=A'

**AND Gate - Block diagram:**





**OR gate - Truth Table**

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Verilog program for AND gate:**

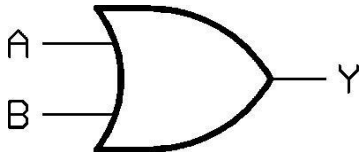
```
module andg (A, B, Y);
        input A, B;
        output Y;
        assign Y = A & B;
endmodule
```

**Verilog test bench  program for AND gate:**

```
module andg_tb;
reg A, B;
wire Y;
andg andgate(.A(A), .B(B),.Y(Y));
initial begin
      A =1'b0;B= 1'b0;
      #10 A =1'b0;B= 1'b1;
      #10 A =1'b1;B= 1'b0;
      #10 A =1'b1;B= 1'b1;
      #10
  $finish;
   end
always @(Y)
   $display( "time =%0t \tINPUT VALUES: \t A=%b B =%b \t output value Y  =%b",$time,A,B,Y);
endmodule
```

**Wave Form:**

**OR Gate - Block diagram:**




**OR gate - Truth Table**

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**Verilog program for OR gate:**

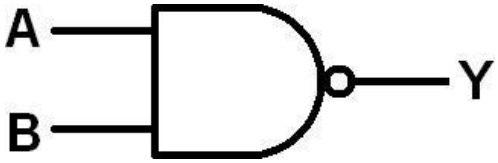
```
module org (A, B, Y);
        input A, B;
        output Y;
        assign Y = A | B;
endmodule
```

**Verilog test bench  program for AND gate:**

```
module org_tb;
reg A, B;
wire Y;
andg andgate(.A(A), .B(B),.Y(Y));
initial begin
      A =1'b0;B= 1'b0;
      #10 A =1'b0;B= 1'b1;
      #10 A =1'b1;B= 1'b0;
      #10 A =1'b1;B= 1'b1;
      #10
         $finish;
   end
always @(Y)
   $display( "time =%0t \tINPUT VALUES: \t A=%b B =%b \t output value Y  =%b",$time,A,B,Y);
endmodule
```

**Wave form:**

**NAND Gate - Block diagram:**




**OR gate - Truth Table**

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

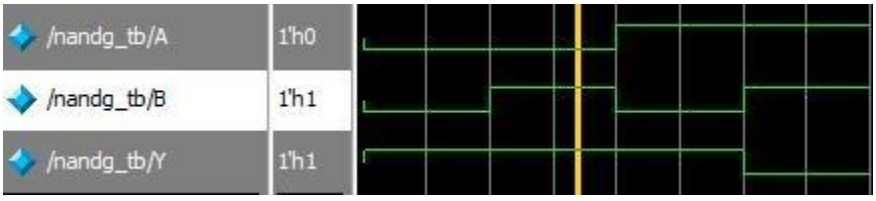
**Verilog program for nand gate:**

```
module nandg (A, B, Y);

input A, B;

output Y;

assign Y = ~(A & B);

endmodule
```

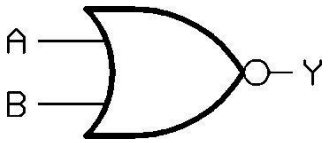
**Verilog testbench program for nand gate:**

```
module nandg_tb;
reg A, B;
wire Y;
nandg nandgate(.A(A), .B(B),.Y(Y));
initial begin
      A =1'b0;B= 1'b0;
      #10 A =1'b0;B= 1'b1;
      #10 A =1'b1;B= 1'b0;
      #10 A =1'b1;B= 1'b1;
      #10 $finish;
   end
always @(Y)
   $display( "time =%0t \tINPUT VALUES: \t A=%b B =%b \t output value Y =%b", $time,A,B,Y);
endmodule
```

**Wave Form:**



**NOR Gate - Block diagram:**




**NOR gate - Truth Table**

| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

**Verilog program for XOR gate:**

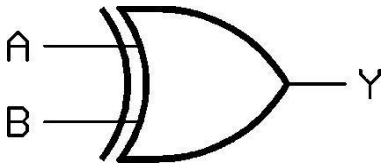
```
module norg (A, B, Y);
        input A, B;
        output Y;
        assign Y = ~(A | B);
endmodule
```

**Verilog teshbench program for NOR gate:**

```
module norg_tb;
reg A, B;
wire Y;
norg norgate(.A(A), .B(B),.Y(Y));
initial begin
      A =1'b0;B= 1'b0;
      #10 A =1'b0;B= 1'b1;
      #10 A =1'b1;B= 1'b0;
      #10 A =1'b1;B= 1'b1;
      #10
         $finish;
   end
always @(Y)
   $display( "time =%0t \tINPUT VALUES: \t A=%b B =%b \t output value Y  =%b",$time,A,B,Y);
endmodule
```

**Wave Form:**

**XOR Gate - Block diagram:**




**NOR gate - Truth Table**

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

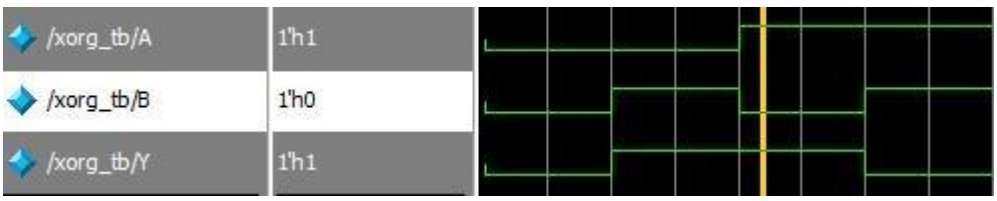
**Verilog program for XOR gate:**

```
module xorg_dataflow (A, B, Y);
input A, B;
output Y;
assign Y = A ^ B ;
endmodule
```

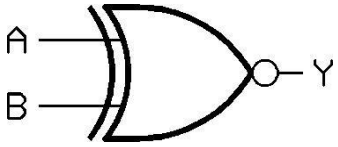
**Verilog teshbench program for XOR gate:**

```
module xorg_tb;
reg A, B;
wire Y;
xorg xorgate(.A(A), .B(B),.Y(Y));
initial begin
        A =1'b0;B= 1'b0;
        #10 A =1'b0;B= 1'b1;
        #10 A =1'b1;B= 1'b0;
        #10 A =1'b1;B= 1'b1;
        #10
            $finish;
    end
always @(Y)
    $display( "time =%0t \tINPUT VALUES: \t A=%b B =%b \t output value Y  =%b",$time,A,B,Y);
endmodule
```

**Wave Form:**



**XNOR Gate - Block diagram:**




**NOR gate - Truth Table**

| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Verilog program for XNOR gate:**

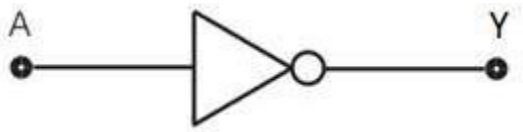
```
module xnorg_dataflow (A, B, Y);

input A, B;

output Y;

assign Y = ~(A ^ B) ;

endmodule
```

**Verilog testbench program for XNOR gate:**

```
module xnorg_tb;
reg A, B;
wire Y;
xnorg xnorgate(.A(A), .B(B),.Y(Y));
initial begin
        A =1'b0;B= 1'b0;
        #10 A =1'b0;B= 1'b1;
        #10 A =1'b1;B= 1'b0;
        #10 A =1'b1;B= 1'b1;
        #10
            $finish;
    end
always @(Y)
    $display( "time =%0t \tINPUT VALUES: \t A=%b B =%b \t output value Y  =%b",$time,A,B,Y);
endmodule
```

**Wave Form:**

**NOT Gate - Block diagram:**




**NOT_gate - Truth Table**

| A | Y |
|---|---|
| 0 | 1 |
| 1 | 0 |

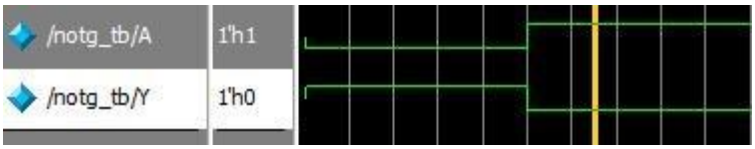
**Verilog program for NOT gate:**

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

**Verilog testbench program for NOT gate:**

```
module notg_tb;
reg A;
wire Y;
notg norgate(.A(A),.Y(Y));
initial begin
       A =1'b0;
       #10 A =1'b1;
       #10
          $finish;
   end
always @(Y)
   $display( "time =%0t \tINPUT VALUES: \t A=%b  output value Y  =%b",$time,A,Y);
endmodule
```

**Wave Form:**




**Result :**

# EXPERIMENT 2

## HDL CODE FOR CARRY LOOK AHEAD ADDER

## Aim

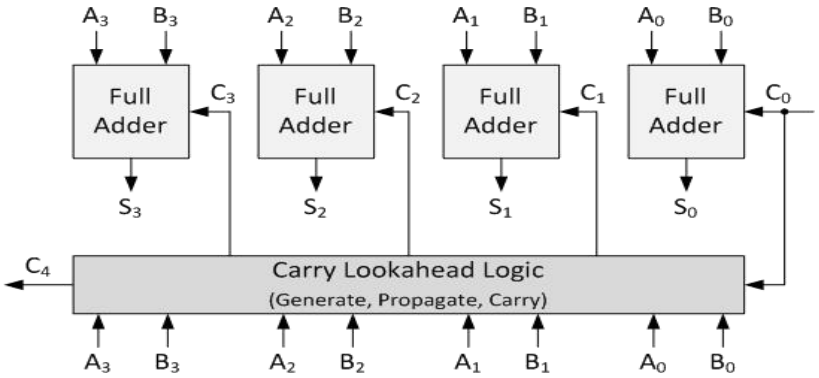
To design and simulate the HDL code for 4-bit ripple carry and carry look ahead adder using behavioural, dataflow and structural modeling

**Apparatus required: Electronics Design Automation Tools used**

- Xilinx ISE Simulator tool
- Xilinx XST Synthesis tool
- Xilinx Spartan FPGA Board
- JTAG cable
- Adaptor 5v/4A

**Theory:**

A carry-lookahead adder improves speed by reducing the amount of time required to determine carry bits. It can be contrasted with the simpler, but usually slower, ripple carry adder for which the carry bit is calculated alongside the sum bit, and each bit must wait until the previous carry has been calculated to begin calculating its own result and carry bits. The carry-look ahead adder calculates one or more carry bits before the sum, which reduces the wait time to calculate the result of the larger value bits. The CLA solves the problem of delay it takes to propagate the carry, by calculating the carry signal in advance based on the input signal . The working of this adder can be understood by manipulating Boolean expressions dealing with full adder.



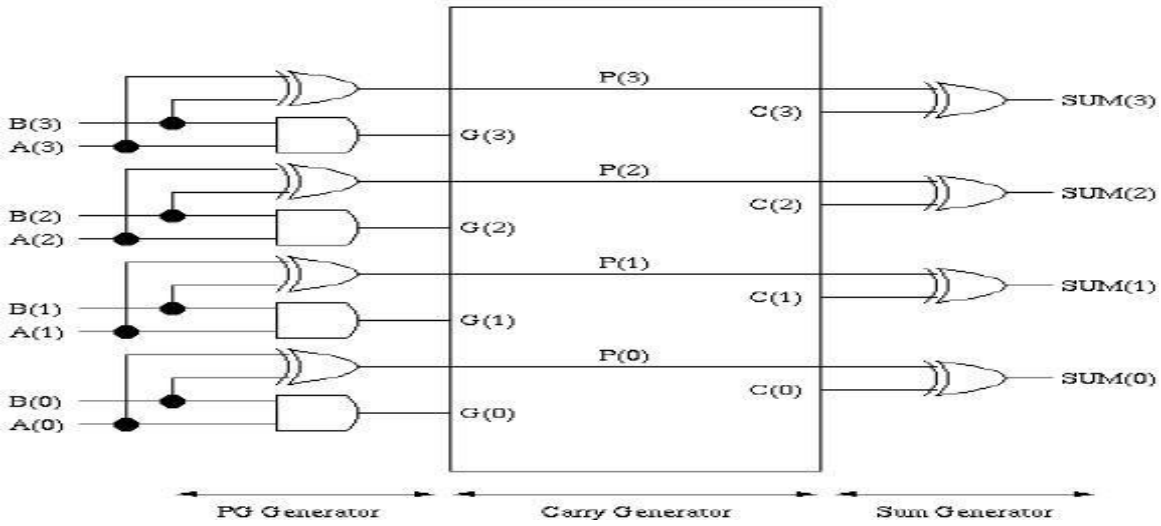



Figure Carry-look ahead Adder

**Truth Table:**

| A | B | Cin | Cout | Condition |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | No carry generation |
| 0 | 0 | 1 | 0 | |
| 0 | 1 | 0 | 0 | |
| 0 | 1 | 1 | 1 | No carry propagate |
| 1 | 0 | 0 | 0 | |
| 1 | 0 | 1 | 1 | |
| 1 | 1 | 0 | 1 | Carry generate |
| 1 | 1 | 1 | 1 | |

**Verilog Code**

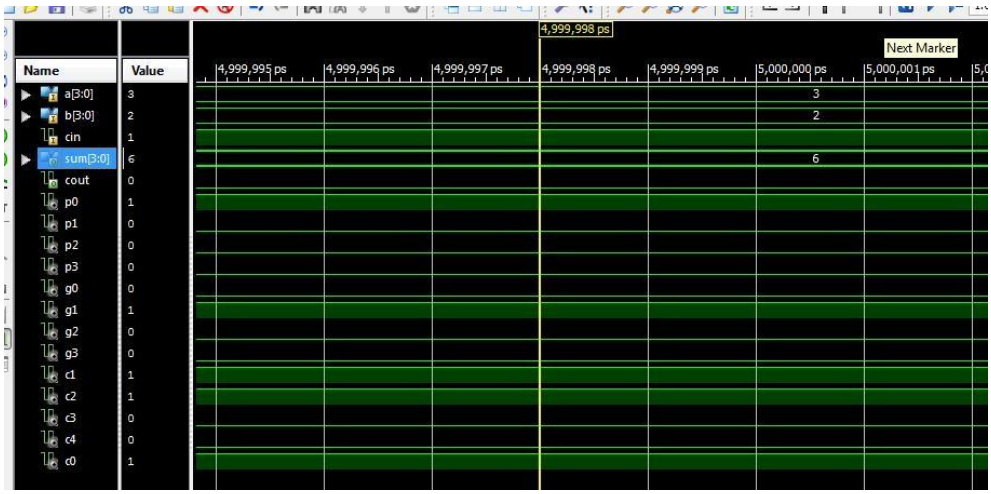
```
module p21(a,b,cin,sum,cout);input[3:0] a,b;
input cin;
output [3:0] sum;output cout;
wire p0,p1,p2,p3,g0,g1,g2,g3,c1,c2,c3,c4;assign p0=(a[0]^b[0]),
p1=(a[1]^b[1]),
p2=(a[2]^b[2]),
p3=(a[3]^b[3]);
                    assign g0=(a[0]&b[0]),
                        g1=(a[1]&b[1]),
g2=(a[2]&b[2]),
g3=(a[3]&b[3]);
assign c0=cin, c1=g0|(p0&cin),
c2=g1|(p1&g0)|(p1&p0&cin),
c3=g2|(p2&g1)|(p2&p1&g0)|(p1&p1&p0&cin),
c4=g3|(p3&g2)|(p3&p2&g1)|(p3&p2&p1&g0)|(p3&p2&p1&p0&cin);
 assign sum[0]=p0^c0,
sum[1]=p1^c1,
sum[2]=p2^c2,
sum[3]=p3^c3;
assign cout=c4;
endmodule
```

# Test Bench:

```
module TestModule;
// Inputs
reg [3:0] a;
reg [3:0] b;
reg cin;

// Outputs
wire [3:0] sum;
wire cout;

// Instantiate the Unit Under Test (UUT)
CLA_Adder uut (
.a(a),
.b(b),
.cin(cin),
.sum(sum),
.cout(cout)
```

```
);
initial begin
// Initialize Inputs
a = 0;
b = 0;
cin = 0;
// Wait 100 ns for global reset to finish
#100;
a = 5;
b = 6;
cin = 1;
// Wait 100 ns for global reset to finish
#100;
end
endmodule
```

**Waveform:**




**Result:**

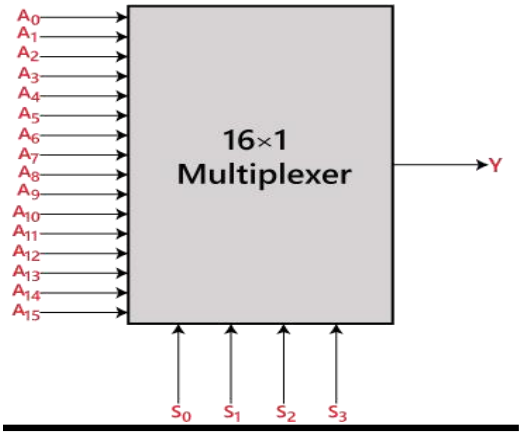
# EXPERIMENT 3

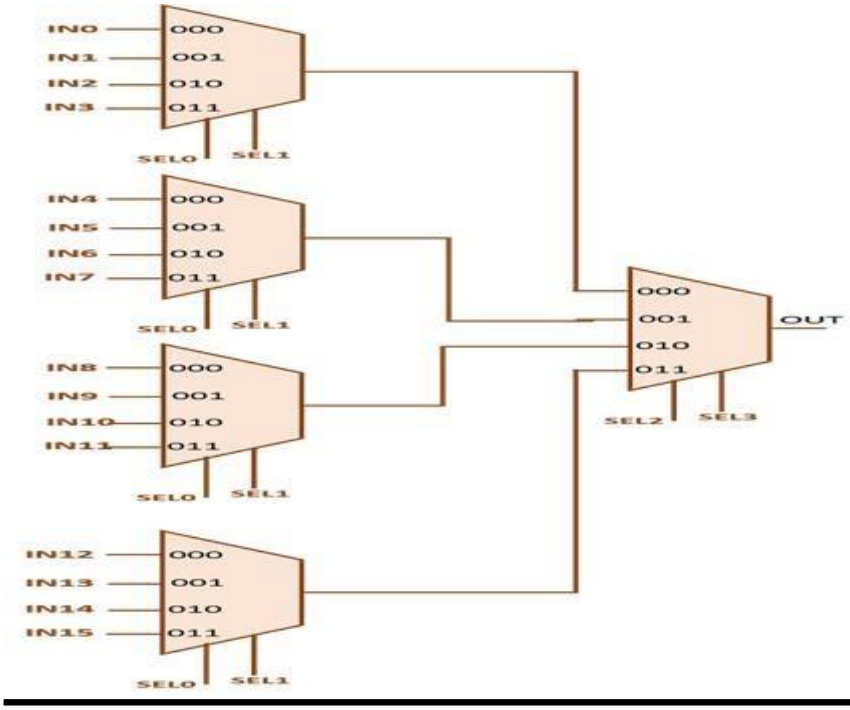
**Aim:** To design the 16 x 1 Multiplexer using 4 x 1 MUX using Verilog and simulate the design

**Apparatus required: Electronics Design Automation Tools used**

- Xilinx ISE Simulator tool
- Xilinx XST Synthesis tool
- Xilinx Spartan FPGA Board
- JTAG cable
- Adaptor 5v/4A

**Block diagram:**



**TRUTH TABLE:**

| INPUTS | | | | Output |
|---|---|---|---|---|
| $S_0$ | $S_1$ | $S_2$ | $S_3$ | Y |
| 0 | 0 | 0 | 0 | $A_0$ |
| 0 | 0 | 0 | 1 | $A_1$ |
| 0 | 0 | 1 | 0 | $A_2$ |
| 0 | 0 | 1 | 1 | $A_3$ |
| 0 | 1 | 0 | 0 | $A_4$ |
| 0 | 1 | 0 | 1 | $A_5$ |
| 0 | 1 | 1 | 0 | $A_6$ |
| 0 | 1 | 1 | 1 | $A_7$ |
| 1 | 0 | 0 | 0 | $A_8$ |
| 1 | 0 | 0 | 1 | $A_9$ |
| 1 | 0 | 1 | 0 | $A_{10}$ |
| 1 | 0 | 1 | 1 | $A_{11}$ |
| 1 | 1 | 0 | 0 | $A_{12}$ |
| 1 | 1 | 0 | 1 | $A_{13}$ |
| 1 | 1 | 1 | 0 | $A_{14}$ |
| 1 | 1 | 1 | 1 | $A_{15}$ |

**HDL Program File for 4:1 MUX [ MUX4X1.v]**

```
module mux4to1_gate(out,in,sel);

input [0:3] in;

input [0:1] sel;
```

```
output out;

wire a,b,c,d,n1,n2,a1,a2,a3,a4;

not n(n1,sel[1]);

not nn(n2,sel[0]);

and (a1,in[0],n1,n2);

and (a2,in[1],n2,sel[1]);

and (a3,in[2],sel[0],n1);

and (a4,in[3],sel[0],sel[1]);

or or1(out,a1,a2,a3,a4);

endmodule
```

**HDL Program File for 16:1 MUX [ MUX16X1.v**

```
module mux16to1(out,in,sel);

input [0:15] in;

input [0:3] sel;

output out;

wire [0:3] ma;

mux4to1_gate mux1(ma[0],in[0:3],sel[2:3]);

mux4to1_gate mux2(ma[1],in[4:7],sel[2:3]);

mux4to1_gate mux3(ma[2],in[8:11],sel[2:3]);

mux4to1_gate mux4(ma[3],in[12:15],sel[2:3]);

mux4to1_gate mux5(out,ma,sel[0:1]);

endmodule
```

**HDL Test Bench File for 16:1 MUX [TESTMUX16.v]**

```
module testmux_16;

reg [0:15] in;
```

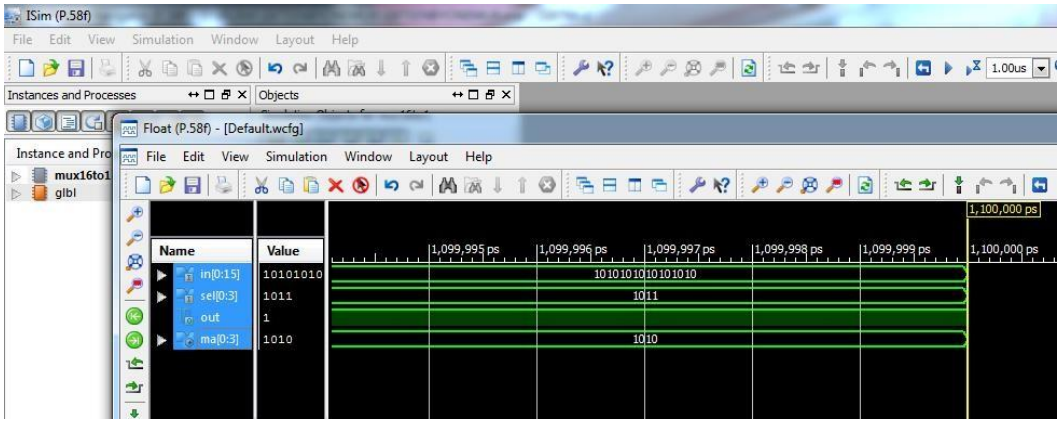
```
reg [0:3] sel;
wire out;
mux16to1 mux(out,in,sel);
initial
begin
$monitor("in=%b | sel=%b | out=%b",
in,sel,out);
end
initial
begin
in=16'b1000000000000000; sel=4'b0000;
#30 in=16'b0100000000000000; sel=4'b0001;
#30 in=16'b0010000000000000; sel=4'b0010;
#30 in=16'b0001000000000000; sel=4'b0011;
#30 in=16'b0000100000000000; sel=4'b0100;
#30 in=16'b0000010000000000; sel=4'b0101;
#30 in=16'b0000001000000000; sel=4'b0110;
#30 in=16'b0000000100000000; sel=4'b0111;
#30 in=16'b0000000010000000; sel=4'b1000;
#30 in=16'b0000000001000000; sel=4'b1001;
#30 in=16'b0000000000100000; sel=4'b1010;
#30 in=16'b0000000000010000; sel=4'b1011;
#30 in=16'b0000000000001000; sel=4'b1100;
#30 in=16'b0000000000000100; sel=4'b1101;
#30 in=16'b0000000000000010; sel=4'b1110;
```

#30 in=16'b0000000000000001; sel=4'b1111;

end

endmodule

**Waveform**




**Result:**

b) 3:8 decoder realization through 2:4 decoder

**Aim:** To design the 3:8 decoder realization through 2:4 decoder using Verilog and simulate the design

**Apparatus required: Electronics Design Automation Tools used**

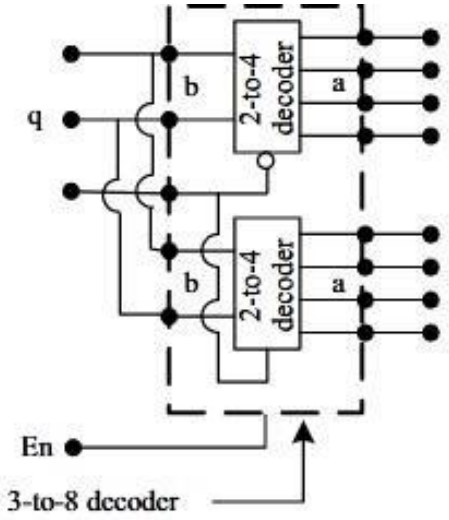
- Xilinx ISE Simulator tool
- Xilinx XST Synthesis tool
- Xilinx Spartan FPGA Board
- JTAG cable
- Adaptor 5v/4A

**THEORY:**

As a decoder is a combinational circuit takes an n-bit binary number and produces an output on one of 2n output lines. A decoder is a multiple input, multiple output logic circuit that converts coded inputs into coded outputs where the input and output codes are different. The enable inputs must be ON for the decoder to function, otherwise its outputs assumes a 'disabled' output code word. Decoding is necessary in applications such as data multiplexing, seven segment display and memory address decoding.

In a 2-to-4 binary decoder, two inputs are decoded into four outputs hence it consists of two input linesand 4 output lines. Only one output is active at any time while the other outputs are maintained at logic 0 and the output which is held active or high is determined the two binary inputs A1 and A0. The 3-to-8 decoders have an "Enable" input each (designated 'en' – one being of the active high and the other of the active low type); these are connected to the most significant bit of the 4-b it input to form the4-to-16 decoder. The 3-to-8 decoder can again be formed in terms of two 2-to-4 decoders in the same manner as shown in Figure

**Circuit Diagram:**




**Truth Table:**

| Inputs | | | | Outputs | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| EN | q2 | q1 | q0 | PP7 | PP6 | PP5 | PP4 | PP3 | PP2 | PP1 | PP0 |
| 0 | × | × | × | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**2X 4 Decoder**

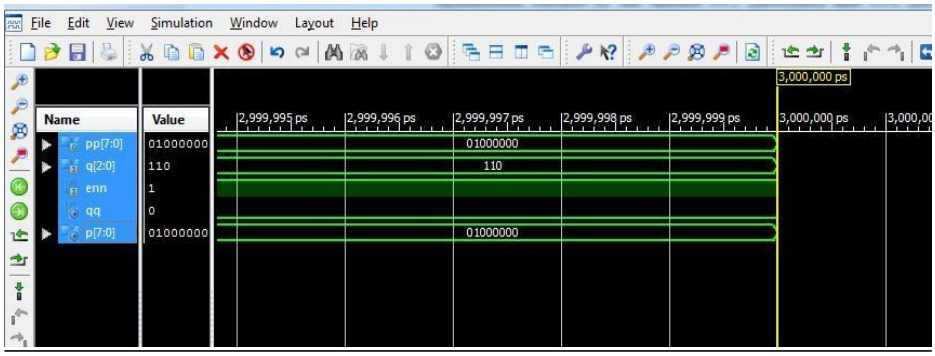
```
module dec2_4 (a,b,en);
output [3:0] a;
input [1:0]b; input en;
wire [1:0]bb;
```

```
not(bb[1],b[1]),(bb[0],b[0]);
and(a[0],en, bb[1],bb[0]),(a[1],en, bb[1],b[0]),
(a[2],en, b[1],bb[0]),(a[3],en, b[1],b[0]);
endmodule
//test bench
module tst_dec2_4();
wire [3:0]a;
reg[1:0] b; reg en;
dec2_4 dec(a,b,en);
initial
begin
{b,en} =3'b000;
#2{b,en} =3'b001;
#2{b,en} =3'b011;
#2{b,en} =3'b101;
#2{b,en} =3'b111;
end
initial
$monitor ($time , "output a = %b, input b = %b ",
a, b);
endmodule
```

**Verilog Code for 3 TO 8 DECODER**

```
module dec3_8(pp,q,enn);
output[7:0]pp;
input[2:0]q;
input enn;
wire qq;
wire[7:0]p;
not(qq,q[2]);
dec2_4 g1(.a(p[3:0]),.b(q[1:0]),.en(qq));
dec2_4 g2(.a(p[7:4]),.b(q[1:0]),.en(q[2]));
and g30(pp[0],p[0],enn);
and g31(pp[1],p[1],enn);
and g32(pp[2],p[2],enn);
and g33(pp[3],p[3],enn);
and g34(pp[4],p[4],enn);
and g35(pp[5],p[5],enn);
and g36(pp[6],p[6],enn);
and g37(pp[7],p[7],enn);
endmodule
```

## Waveform




## Result:

# EXPERIMENT 4

**Design of 8-to-3 encoder (without and with parity) usingVerilog HDL**

**Aim** To design the 8x3 encoder using Verilog and simulate the design

**Apparatus required:- Electronics Design Automation Tools used:-**

- Xilinx Spartan 3 FPGA
- Xilinx ISE Simulator tool
- Xilinx XST Synthesis tool
- Xilinx Project Navigator 8.1i
- JTAG cable
- Adaptor 5v/4A

**THEORY:**

An encoder is a combinational logic circuit that essentially performs a "reverse" of decoder functions. An encoder has 2^N input lines and N output lines. In encoder the output lines generate the binary code corresponding to input value. An encoder accepts an active level on one of its inputs, representing digit, such as a decimal or octal digits, and converts it to a coded output such as BCD or binary. Encoders can also be devised to encode various symbols and alphabetic characters. The process of converting from familiar symbols or numbers to a coded format is called ***encoding***. An encoder has a number of input lines, only one of which input is activated at a given time and produces an N-bit output code, depending on which input is activated.

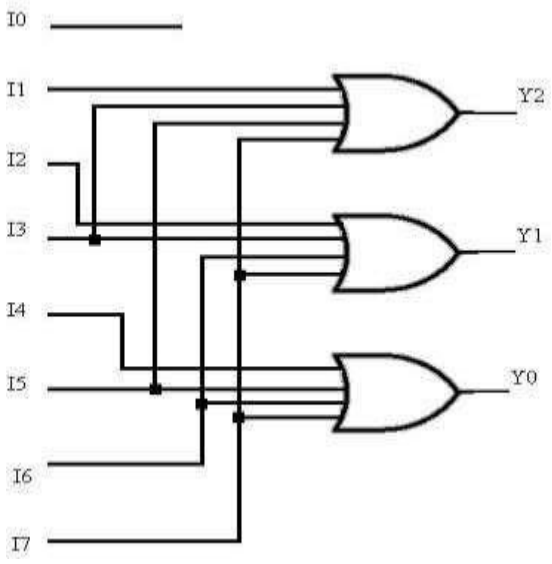
For an 8-to-3 binary encoder with inputs I0-I7 the logic expressions of the outputs Y0-Y2 are:

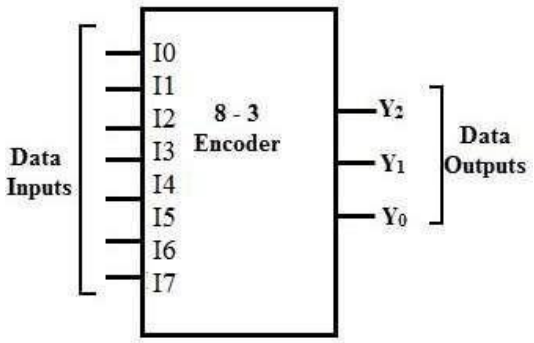
Y0 = I1 + I3 + I5 + I7

Y1= I2 + I3 + I6 + I7

Y2 = I4 + I5 + I6 +I7

**Circuit Diagram, Block diagram and Truth Table:**




| I0 | I1 | I2 | I3 | I4 | I5 | I6 | I7 | Y2 | Y1 | Y0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

**Verilog program for 8x3 encoder structural:**

```
module encoder_8_to_3(input [7:0] I,output reg [2:0] Y );
                or(Y[2],I[4],I[5],I[6],I[7]);
                or(Y[1],I[2],I[3],I[6],I[7]);
                or(Y[0],I[1],I[3],I[5],I[7]);
endmodule
```

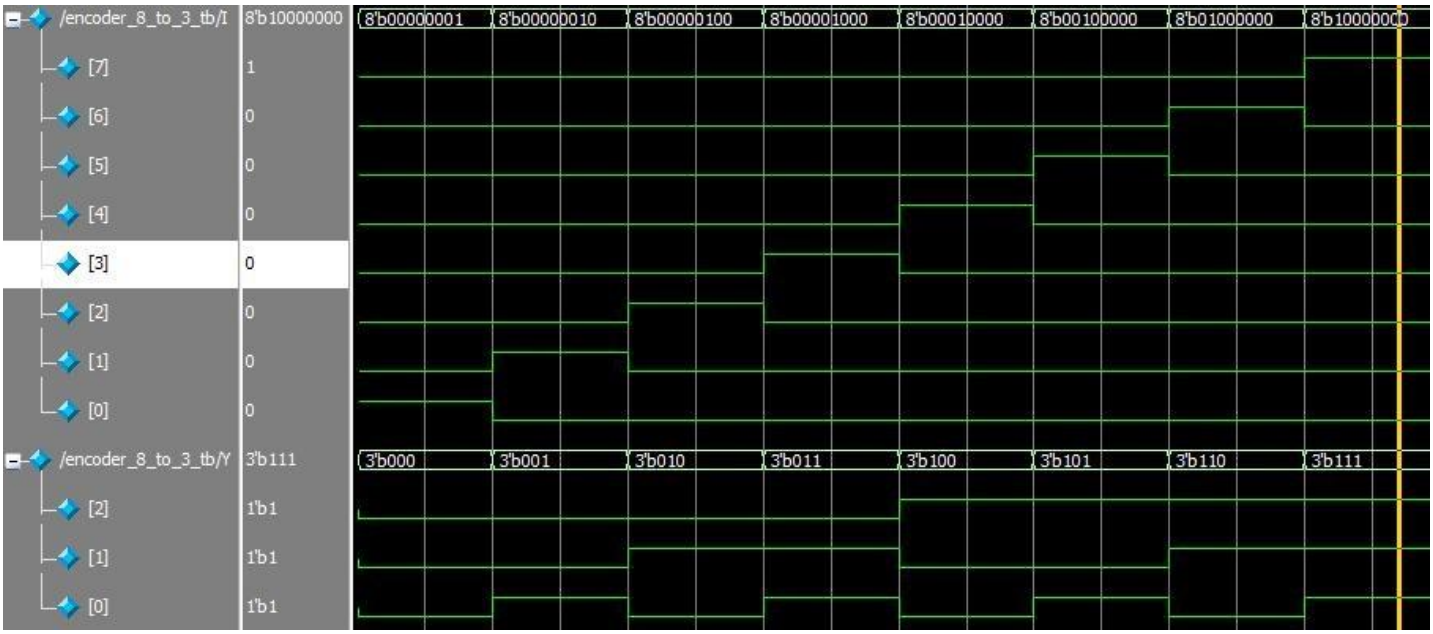
**Verilog program for 8x3 encoder behavioral:**

```
module encoder_8_to_3(input [7:0] I,output reg [2:0] Y );
always@(*)
        begin
        case(I)
                8'b00000001: Y<= 3'b000;
                8'b00000010: Y <= 3'b001;
                8'b00000100: Y <= 3'b010;
                8'b00001000: Y <= 3'b011;
                8'b00010000: Y <= 3'b100;
                8'b00100000: Y <= 3'b101;
                8'b01000000: Y <= 3'b110;
                8'b10000000: Y <= 3'b111;
                default: Y<= 3'bxxx;
endcase
end
endmodule
```

**Verilog testbench program for 8x3 encoder behavioral:**

```
module encoder_8_to_3_tb;
                reg [7:0] I;
                wire [2:0] Y;
encoder_8_to_3 encoder(.I(I),.Y(Y));
initial begin
        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$stop;
        end
always @(Y)
   $display("time =%0t \tINPUT VALUES: \t I=%b  \t output value Y  = %b ",$time,I, Y);
endmodule
```

## Expected Wave form:




**Result:**

# Experiment 5:

**Aim :Design of** 8-bit parity generator and checker

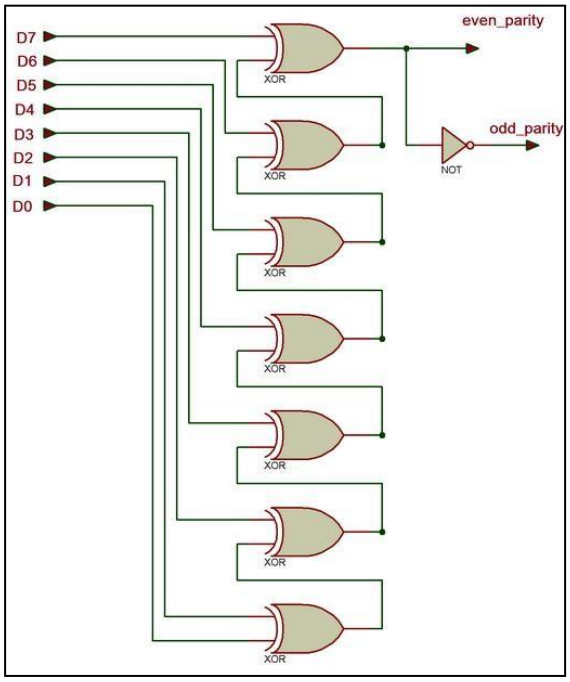
**Apparatus required: - Electronics Design Automation Tools used: -**

- Xilinx ISE Simulator tool
- Xilinx XST Synthesis tool
- Xilinx Spartan FPGA Board
- JTAG cable
- Adaptor 5v/4A

## Theory:

The function odd-parity is defined within the module parity-check in Figure 9.2. It generates a parity bit. The parity bit is 1 if the number of one-bits in the byte is odd. Otherwise it is zero. The module has an 8-bit vector input and a flag input – en. It has an output chk. Whenever the flag goes high, the function odd-parity is called. It returns the parity bit value and assigns it to chk in the module. parity-check is an example with a single-bit output-type function in it. The function has no local variables in it.

## Circuit Diagram



**Truth Table**

| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | Even_parity | Odd_parity |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |

**VHDL Code for parity generator**

```
library ieee;
use ieee.std_logic_1164.all;
entity parity is
     port( data:in bit_vector(7 downto 0);
              even_p,odd_p: out bit);
end parity;
architecture parity_gen of parity is
signal temp : bit_vector(5 downto 0);
begin
   temp(0)<=data(0) xor data(1);
   temp(1)<=temp(0) xor data(2);
   temp(2)<=temp(1) xor data(3);
   temp(3)<=temp(2) xor data(4);
   temp(4)<=temp(3) xor data(5);
   temp(5)<=temp(4) xor data(6);
   even_p <= temp(5) xor data(7);
   odd_p <= not(temp(5) xor data(7));
end parity_gen;
```

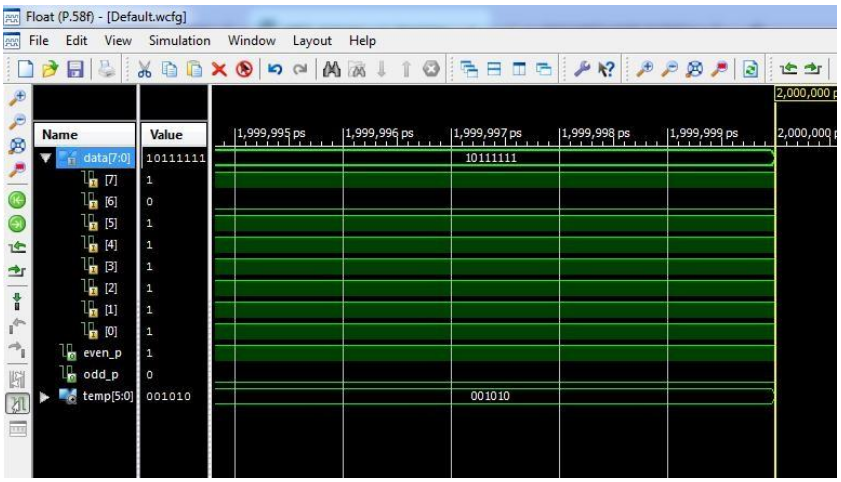
**VHDL Code for parity checker**

```
library ieee;
use ieee.std_logic_1164.all;
entity parity_chk is
     port( data:in bit_vector(7 downto 0);
              p: in bit;
              e: out bit);
end parity_chk;
architecture parity_arch of parity_chk is
signal temp : bit_vector(6 downto 0);
```

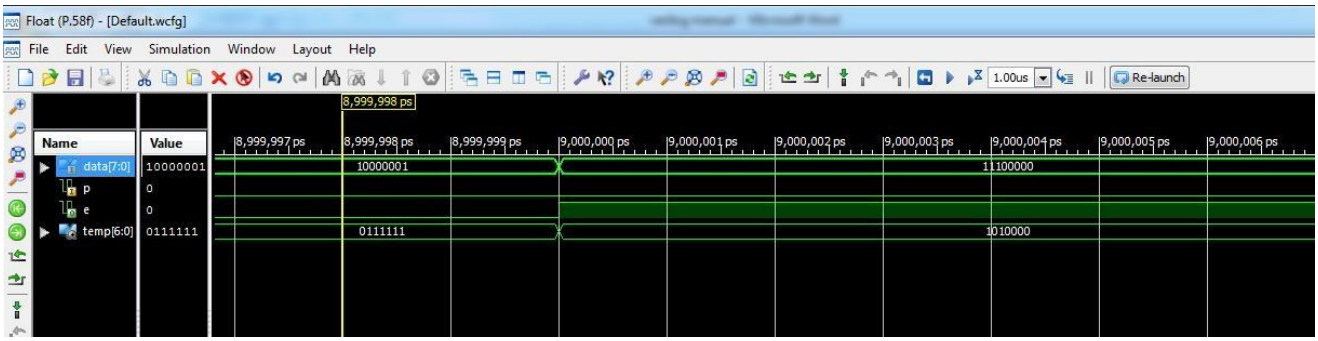
```
  begin
    temp(0)<=data(0) xor data(1);
    temp(1)<=temp(0) xor data(2);
    temp(2)<=temp(1) xor data(3);
    temp(3)<=temp(2) xor data(4);
    temp(4)<=temp(3) xor data(5);
    temp(5)<=temp(4) xor data(6);
    temp(6) <= temp(5) xor data(7);
    e <= p xor temp(6);
end parity_arch;
```

## Waveform:

**Parity Generator:**




**Parity checker:**




**Result:**

# Experiment 6

**Design of Latches and flip flops: D-latch SR, D,JK, T**

**Aim:** Design of Latches and flip flops (D-latch SR, D,JK, T) using Verilog and simulates the design

**Apparatus required: - Electronics Design Automation Tools used: -**

- Xilinx ISE Simulator tool
- Xilinx XST Synthesis tool
- Xilinx Spartan FPGA Board
- JTAG cable
- Adaptor 5v/4A
-

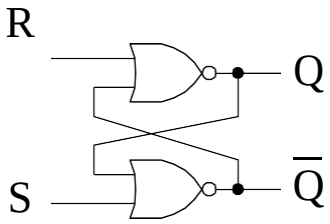
**Theory:**

LATCH AND FLIP-FLOP

*Latch* and *flip-flop* are memory devices and implemented using *bistable* circuit - its output will remain in either 0 or 1 state. The output state of a *latch* is controlled by its excitation input signals. A *flip-flop* (FF) is predominately controlled by a *clock* and its output state is determinted by its excitation input signals. *Note that if the clock is replaced by a gated control signal, the flip-flop becomes a gated latch.*

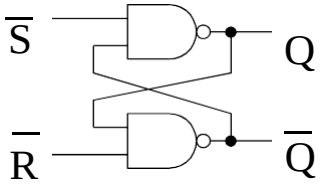
a. RS (reset-set) latch circuit

When S (set) is set to 1, the output Q will be set to 1. Likewise, when R (reset) is set to 1, the output Q will be set to 0. It is invalid to set both S and R to 1.




| S | R | Q | $\overline{Q}$ | |
|---|---|---|---|---|
| 0 | 0 | Unchanged | | |
| 0 | 1 | 0 | 1 | |
| 1 | 0 | 1 | 0 | |
| 1 | 1 | 0 | 0 | Invalid input |

NOR gate implementation



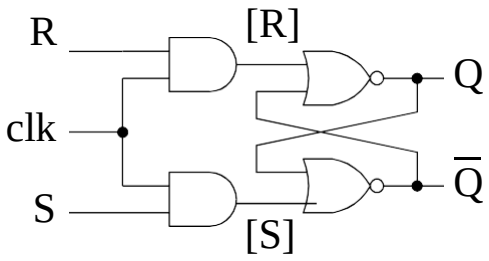

| $\overline{S}$ | $\overline{R}$ | Q | $\overline{Q}$ |
|---|---|---|---|
| 0 | 0 | Invalid input | |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | Unchanged | |

NAND gate implementation

Note that the input is ***active high*** for NOR gate implementation, whereas the input is ***active low*** for NAND gate implementation.

b. Clocked RS FF

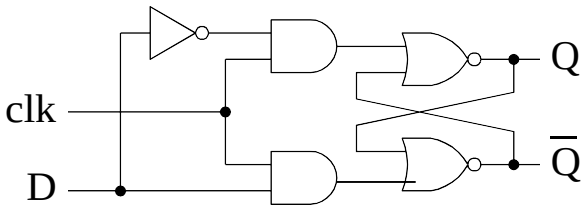
The major problem of RS latch is its susceptibility to voltage noise which could change the output states of the FF. With the clocked RS FF, the problem is remedied. With the clock held low, [S] & [R] held low, the output remains unchanged. With the clock held high, the output follows R & S. Thus the output will be latched its states when the clock goes low.




| | | clk= 1 | clk = 0 |
|---|---|---|---|
| S | R | $Q_{n+1}$ | $Q_{n+1}$ |
| 0 | 0 | $Q_n$ | $Q_n$ |
| 0 | 1 | 0 | $Q_n$ |
| 1 | 0 | 1 | $Q_n$ |
| 1 | 1 | Invalid i/p | $Q_n$ |

c. D-type FF

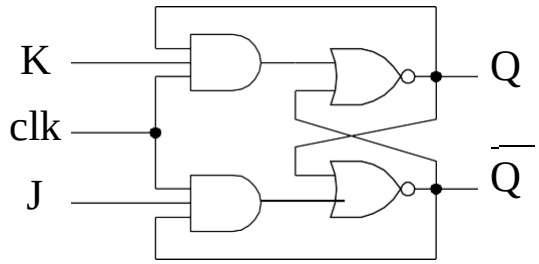
The D-type FF remedies the indeterminate state problem that exists when both inputs to a clocked RS FF are high. The schematic is identical to a RS FF except that an inverter is used to produce a pair of complementary input.




| clk | D | $Q_{n+1}$ |
|---|---|---|
| 0 | x | $Q_n$ |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

d. JK FF

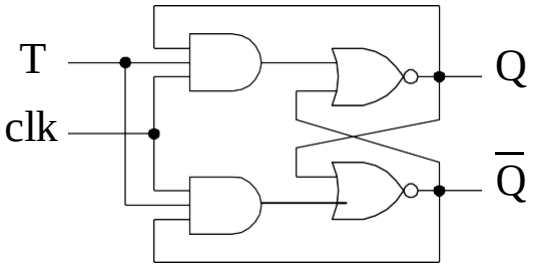
The JK FF is a refinement of the RS FF in that the undetermined state of the RS type is defined in the JK type. Inputs J and K behave like inputs S and R to set and reset (clear) the FF, respectively. The input marked J is for *set* and the input marked K is *reset*.




| | | clk=0 | clk=1 |
|---|---|---|---|
| J | K | $Q_{n+1}$ | $Q_{n+1}$ |
| 0 | 0 | $Q_n$ | $Q_n$ |
| 0 | 1 | $Q_n$ | 0 |
| 1 | 0 | $Q_n$ | 1 |
| 1 | 1 | $Q_n$ | $\overline{Q_n}$ |

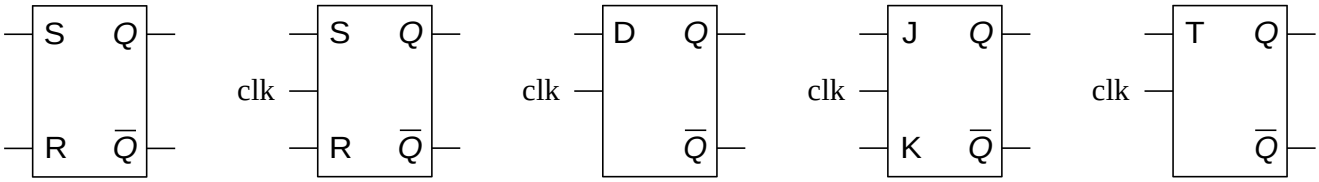
e. T-type FF

The toggle (T) FF has a clock input which causes the output state changed for each clock pulse if T is in its active state. It is useful in counter design.




| clk | T | $Q_{n+1}$ |
|---|---|---|
| 0 | X | $Q_n$ |
| 1 | 0 | $Q_n$ |
| 1 | 1 | $\overline{Q}_n$ |

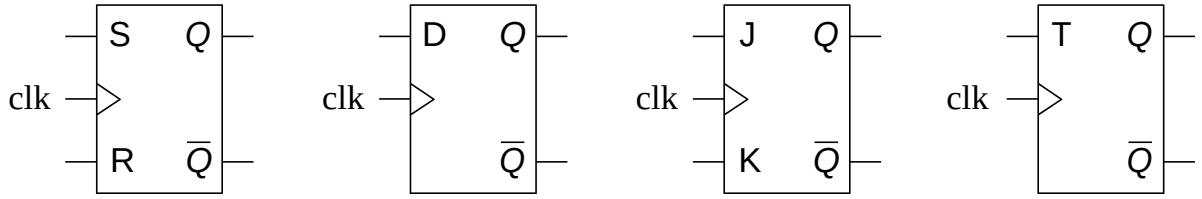
*Logic smybols of various latch and **level-triggered** flip-flops*

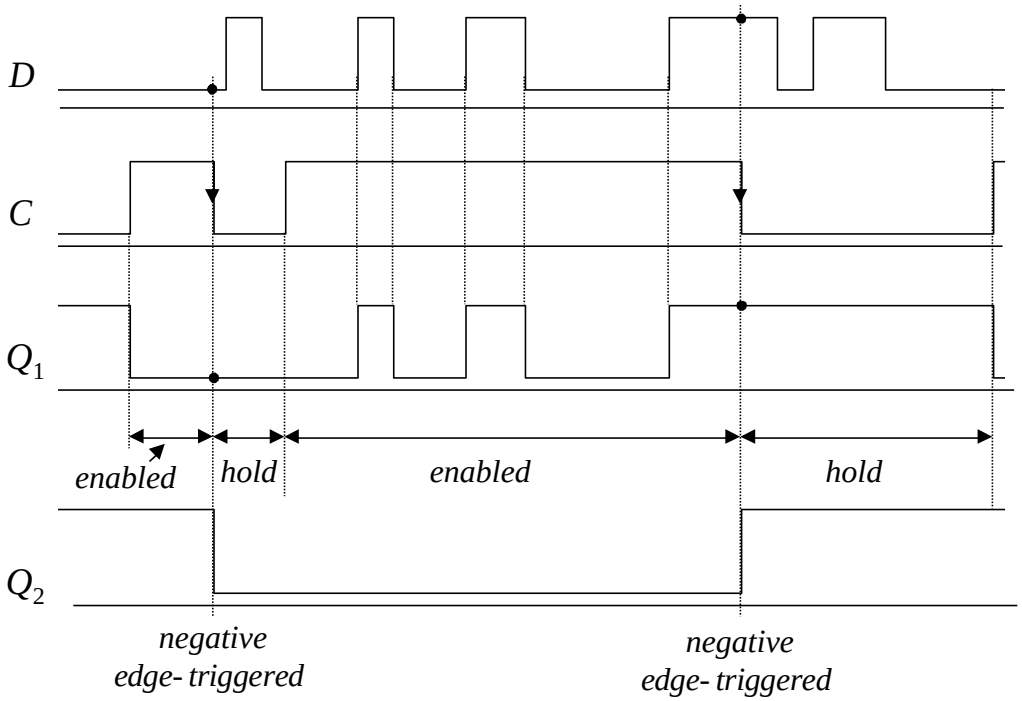



f. Edge-triggered FF

Clocked FF is a ***level-triggered*** device, its output responses to the input during the clock active period and this is referred to as the "0" and "1" catching problem. For sequential synchronous circuit, the data transfer is required to be synchronized with the clock signal. Additional circuit is included in the FF to ensure that it will only response to the input at the transition edge of the clock pulse. These type of devices are called ***edge-triggered*** FFs.

*Logic smybols of various **edge-triggered** flip-flops*

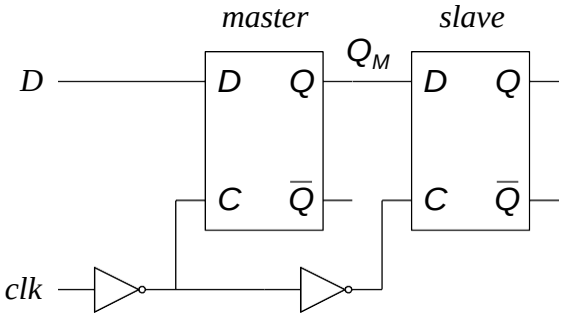




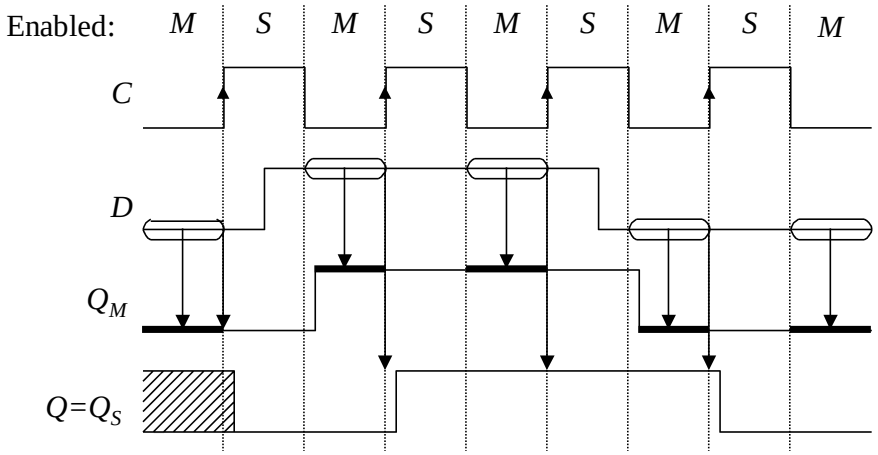

Timing diagram for a gated D latch and a negative edge-triggered D FF

g. Master-slave FF

A master-slave type FF consists of two FFs, a master stage and a slave stage. The output states responding to the inputs are transmitted to slave output through the master stage in different time slots of the clock pulse. Hence the output will not be affected by the undesirable changes at the input after the output of the master FF has been latched to the slave FF.




Master-slave D flip-flop




Timing diagram

h. FF timing parameters

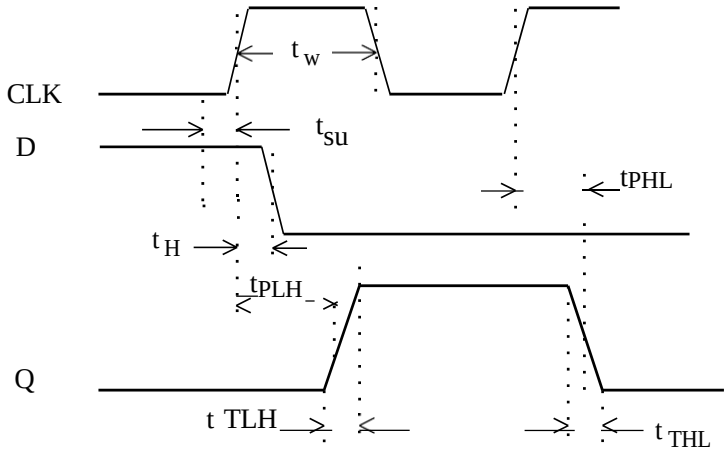
*Propagation delay*: propagation delay for a FF is the amount of time it takes for the output of the FF to change its state from a clock trigger or asynchronous set or rest. It is defined from the 50% point of the input pulse to the 50% point of the output pulse. Propagation delay is specified as $t_{PHL}$ - the propagation time from a HIGH to a LOW, and as $t_{PLH}$ - the propagation time from a LOW to a HIGH.

*Output transition time*: the output transition time is defined as the rise time or fall time of the output. The $t_{TLH}$ is the 10% to 90% time, or LOW to HIGH transition time. The $t_{THL}$ is the 90% to 10% time, or the HIGH to LOW transition time.

*Setup time*: the setup time is defined as the interval immediately preceding the active transition of the clock pulse during which the control or data inputs must be stable (at a valid logic level). Its parameter symbol is $t_{su}$.

*Hold time*: the hold time is the amount of time that the control or data inputs must be stable after the clock trigger occurs. The $t_H$ is the parameter symbol for hold time.

*Minimum pulse width*: the minimum pulse width is required to guarantee a correct state change for the flip-flop. It is usually denoted by $t_w$.



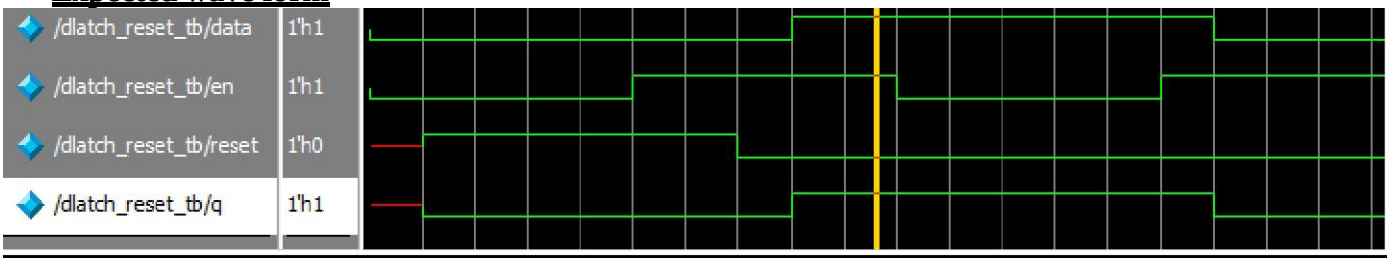
**Verilog program for D-latch**

```
module dlatch_reset (data, en,  reset,  q);
    input data, en, reset ;
    output q;
    reg q;
always @ ( en or reset or data)
    if (reset) begin
    q <= 1'b0;
    end else if (en) begin
    q <= data;
    end
endmodule //End Of Module dlatch_reset
```

**Verilog testbench program for D-latch**

```
module  dlatch_reset_tb;
    reg data, en, reset ;
    wire q;
    dlatch_reset dlatch(data, en,  reset,  q);

    initial
    begin
    en=0;
    data = 0;
    #5 reset = 1;
    #30 reset = 0;
    $monitor($time, "\ten=%b\t ,reset=%b\t, data=%b\t, q=%b",en,reset,data,q);
    #160 $finish;
    end
    always #25 en = ~en;
    always #40 data = ~data;
 endmodule
```

**Expected wave form**



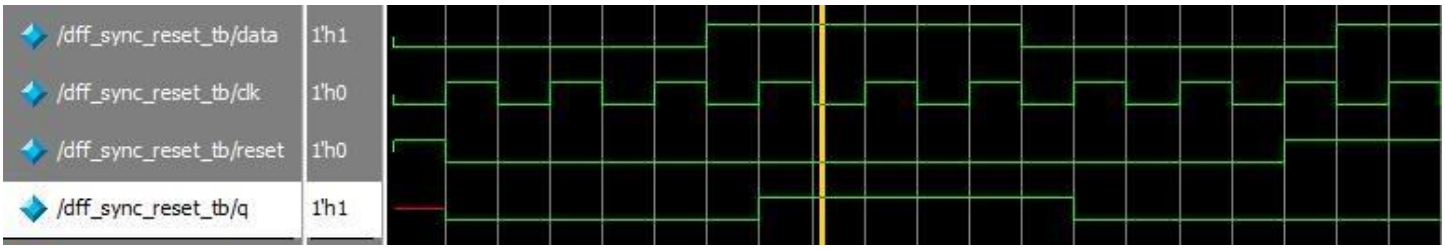
**Verilog program for D-flip flop with sync reset**

```
module dff_sync_reset (
                        data  ,  //  Data   Input
                        clk   ,  //  Clock  Input
                        reset , // Reset input q
                        // Q output
                        );
    input data, clk, reset ;
    output q;
    reg q;
always @ ( posedge clk)
    if (reset) begin
            q <= 1'b0;
    end  else begin
            q <= data;
end
endmodule
```

**Verilog testbench program for D-flip flop with sync reset**

```
module dff_sync_reset_tb;
    reg data, clk, reset ;
    wire q;
    dff_sync_reset dffr (.data(data), .clk(clk), .reset(reset)  ,.q(q));
    initial
    begin
    clk=0;
    data = 0;
    reset = 1;
            #5 reset = 0;
            #80 reset = 1;
            $monitor($time, "\tclk=%b\t ,reset=%b\t, data=%b\t, q=%b",clk,reset,data,q);#100
            $finish;
            end
            always #5 clk = ~clk;
            always #30 data = ~data;
 endmodule
```

**Expected Waveform**



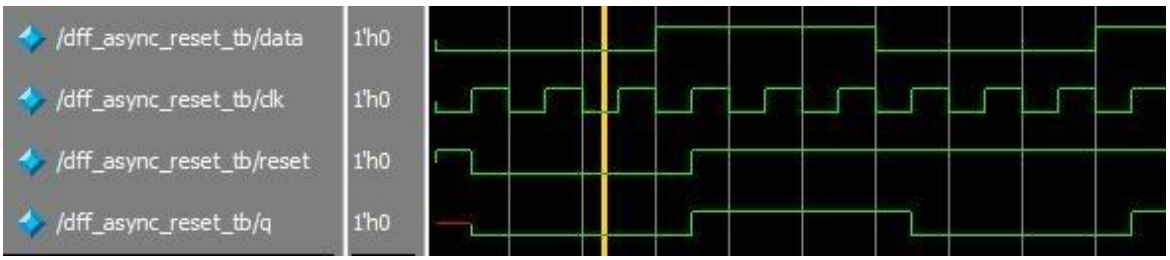
**Verilog testbench program for D-flip flop with async reset**

```
module dff_async_reset (
                        data , // Data Input
                        clk    , // Clock Input
                        reset , // Reset input
                        q         // Q output
                        );
input data, clk, reset ;
     output q;
     reg q;
always @ ( posedge clk or negedge reset)
     if (~reset) begin
               q <= 1'b0;
               end  else begin
               q <= data;
               end
endmodule
```

**Verilog testbench program for D-flip flop with async reset**

```
module dff_async_reset_tb;
     reg data, clk, reset ;
     wire q;
dff_async_reset dffr (.data(data), .clk(clk), .reset(reset) ,.q(q));
     initial
     begin
     clk=0;
     data = 0;
     reset = 1;
     #5 reset = 0;
     #30 reset = 1;
     $monitor($time, "\tclk=%b\t ,reset=%b\t, data=%b\t,  q=%b",clk,reset,data,q);
     #100 $finish;
     end
     always #5 clk = ~clk;
     always #30 data = ~data;
 endmodule
```

**Expected Waveform**



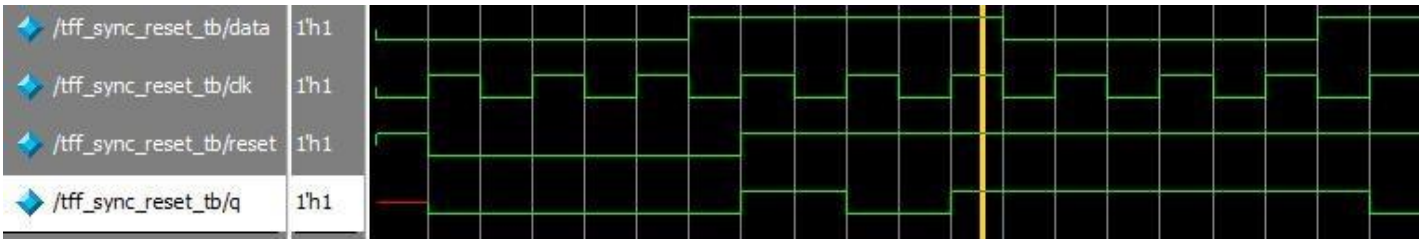
**Verilog program for T-flip flop with sync reset**

```
module tff_sync_reset (
                        data , // Data Input
                        clk , // Clock Input
                        reset , // Reset input
                        q  // Q output
                        );
    input data, clk, reset ;
    output q;
    reg q;
    always @ ( posedge clk)
            if (~reset) begin
                        q <= 1'b0;
                        end else if (data) begin
            q <= !q;
    end
endmodule
```

**Verilog testbench program for T-flip flop with sync reset**

```
module tff_sync_reset_tb;
    reg data, clk, reset ;
    wire q;
tff_sync_reset tffr (.data(data), .clk(clk), .reset(reset)  ,.q(q));
    initial
    begin
    clk=0;
    data = 0;
    reset = 1;
    #5 reset = 0;
    #30 reset = 1;
    $monitor($time, "\tclk=%b\t ,reset=%b\t, data=%b\t,  q=%b",clk,reset,data,q);
    #100 $finish;
    end
    always #5 clk = ~clk;
    always #30 data = ~data;
 endmodule
```

**Expected Waveform**



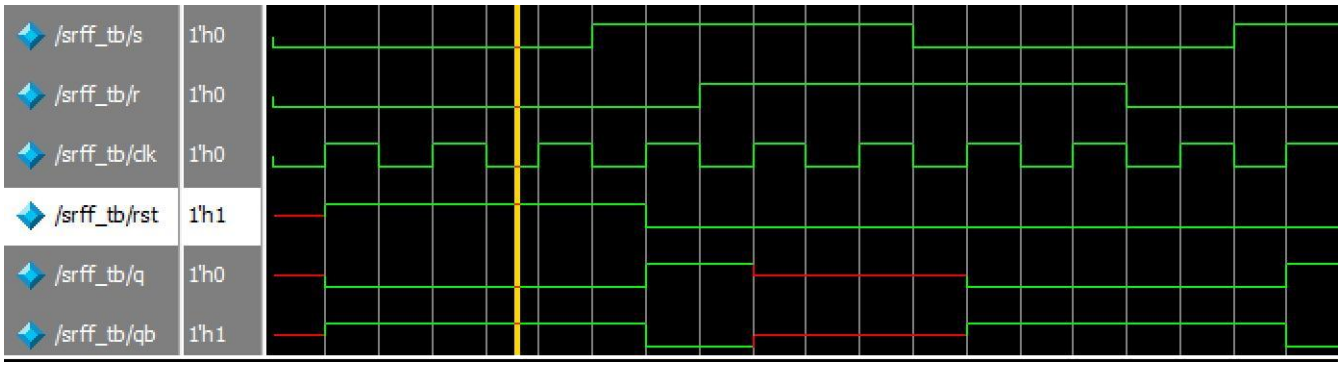
**Verilog program for SR-flip flop**

```
module srff(s,r,clk,rst,q,qb);
    input s,r,clk,rst;
    output q,qb;
    wire s,r,clk,rst,qb;
    reg q;
always @ (posedge clk)
    begin
    if(rst)
    q<=1'b0;
            else if (s==1'b0 && r==1'b0) q<=q;
            else if (s==1'b0&& r==1'b1) q<=1'b0;
            else if (s==1'b1 && r==1'b0) q<=1'b1;
            else if (s==1'b1 && r==1'b1) q<=1'bx;
            end
            assign qb=~q;
endmodule
```

**Verilog testbench program for SR-flip flop**

```
module srff_tb;
    reg s,r,clk,rst;
    wire q,qb;
    srff srflipflop(.s(s),.r(r),.clk(clk),.rst(rst),.q(q),.qb(qb));
    initial
    begin
    clk=0;
    s = 0; r = 0;
    #5 rst = 1;       #30 rst = 0;
    $monitor($time, "\tclk=%b\t ,rst=%b\t, s=%b\t,r=%b\t, q=%b\t, qb=%b",clk,rst,s,r,q,qb);
    #100 $finish;
    end
    always #5  clk = ~clk;
    always #30 s = ~s;
    always #40 r = ~r;
 endmodule
```

**Expected Waveform**



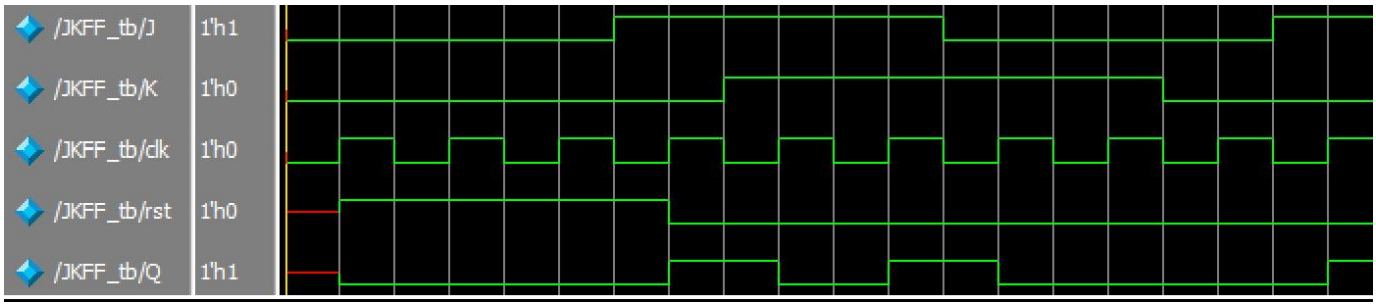
**Verilog program for JK-flip flop**

```
module JKFF ( input J,input K, input clk, input rst, output reg Q);
    always @(posedge clk or posedge rst) //asynch reset
    begin
                if(rst == 1)
                        begin
                        Q <= 0;
                        end
                else begin
                        case({J, K})
                        2'b00: Q <= Q; //no change
                        2'b01: Q <= 1'b0; //Clear
                        2'b10: Q <= 1'b1; //Set
                        2'b11: Q <= ~Q; //Complement
                        endcase
                end
    end
endmodule
```

**Verilog testbench program for JK-flip flop with**

```
module JKFF_tb;
    reg J,K,clk,rst;
    wire Q;
    JKFF JKflipflop(.J(J),.K(K),.clk(clk),.rst(rst),.Q(Q));
    initial
    begin
    clk=0;  J = 0; K = 0;
    #5 rst = 1;
    #30 rst = 0;
    $monitor($time, "\tclk=%b\t ,rst=%b\t, J=%b\t,K=%b\t, Q=%b",clk,rst,J,K,Q);
    #100 $finish;
    end
    always #5  clk = ~clk;
    always #30 J = ~J;
    always #40 K = ~K;
 endmodule
```

**Expected Waveform**

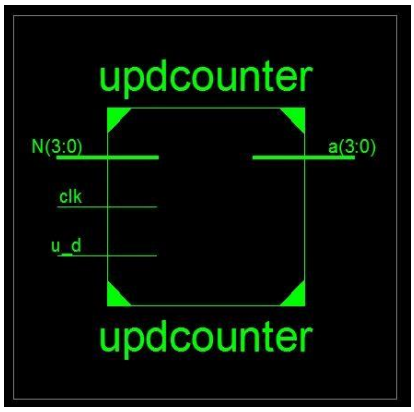


# EXPERIMENT 7: 8bit Up/Down Counter

**Aim:** To Design and implement an 8bit Up/Down Counter with enable input and synchronous clear

**Apparatus required :-** Electronics Design Automation Tools used:-

- Xilinx ISE Simulator tool
- Xilinx XST Synthesis tool




Verilog code:

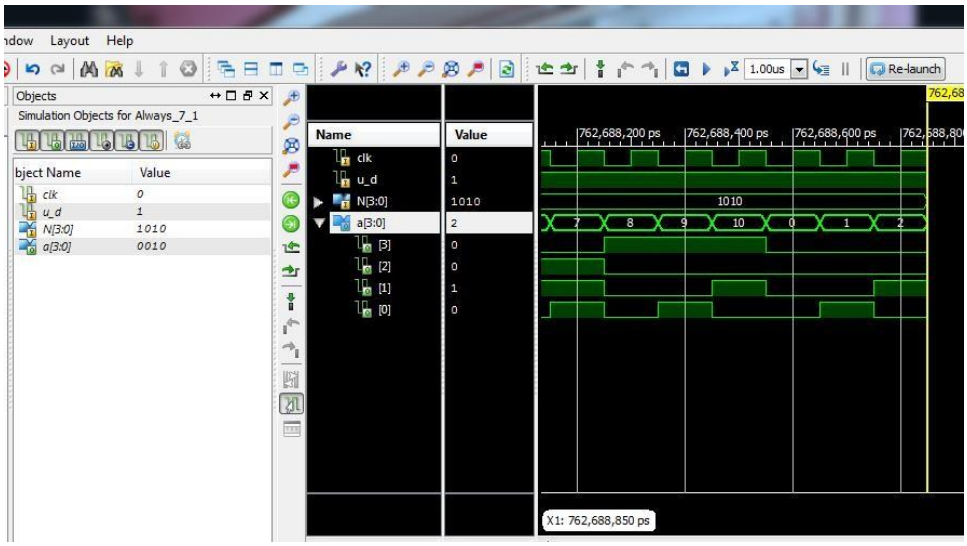
```
module updcounter(a,clk,N,u_d);
input clk,u_d;
input[3:0]N;
output[3:0]a;
reg[3:0]a;
initial a =4'b0000;
always@(negedge clk)
a=(u_d)?((a==N)?4'b0000:a+1'b1):((a==4'b0000)?N:a-
1'b1);
endmodule
```

**Test bench**

```
module tst_updcounter();//TEST_BENCH
reg clk,u_d;
reg[3:0]N;
wire[3:0]a;
updcounter c2(a,clk,N,u_d);
initial
begin
N = 4'b0111;
u_d = 1'b0;
clk = 0;
end
always #2 clk=~clk;
```

```
always #34u_d=~u_d;
initial $monitor
($time,"clk=%b,N=%b,u_d=%b,a=%b",clk,N,u_d,a);
initial #64 $stop;
endmodule
```

## Waveform




BEHAVIORAL MODELING — 1

```
# 0clk=0,N=0111,u_d=0,a=0111
# 2clk=1,N=0111,u_d=0,a=0111
# 4clk=0,N=0111,u_d=0,a=0110
# 6clk=1,N=0111,u_d=0,a=0110
# 8clk=0,N=0111,u_d=0,a=0101
# 10clk=1,N=0111,u_d=0,a=0101
# 12clk=0,N=0111,u_d=0,a=0100
# 14clk=1,N=0111,u_d=0,a=0100
# 16clk=0,N=0111,u_d=0,a=0011
# 18clk=1,N=0111,u_d=0,a=0011
# 20clk=0,N=0111,u_d=0,a=0010
# 22clk=1,N=0111,u_d=0,a=0010
# 24clk=0,N=0111,u_d=0,a=0001
# 26clk=1,N=0111,u_d=0,a=0001
# 28clk=0,N=0111,u_d=0,a=0000
# 30clk=1,N=0111,u_d=0,a=0000
# 32clk=0,N=0111,u_d=0,a=0111
# 34clk=1,N=0111,u_d=1,a=0111
# 36clk=0,N=0111,u_d=1,a=0000
```

```
# 38clk=1,N=0111,u_d=1,a=0000
# 40clk=0,N=0111,u_d=1,a=0001
# 42clk=1,N=0111,u_d=1,a=0001
# 44clk=0,N=0111,u_d=1,a=0010
# 46clk=1,N=0111,u_d=1,a=0010
# 48clk=0,N=0111,u_d=1,a=0011
# 50clk=1,N=0111,u_d=1,a=0011
# 52clk=0,N=0111,u_d=1,a=0100
# 54clk=1,N=0111,u_d=1,a=0100
# 56clk=0,N=0111,u_d=1,a=0101
# 58clk=1,N=0111,u_d=1,a=0101
# 60clk=0,N=0111,u_d=1,a=0110
# 62clk=1,N=0111,u_d=1,a=0110
```

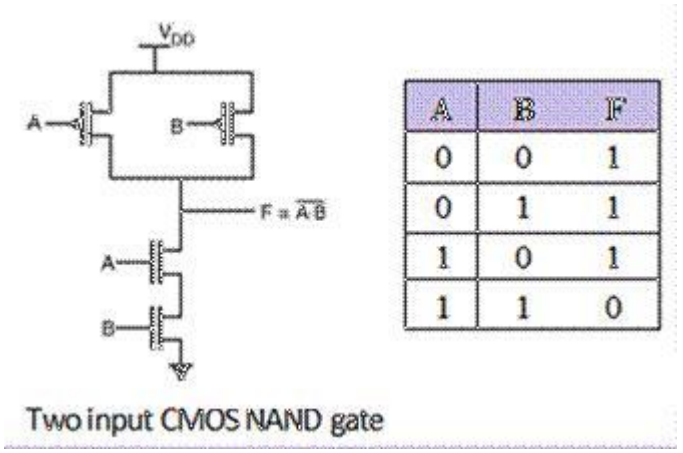
## Result

# Experiment 8: Design and Implementation Universal Gates

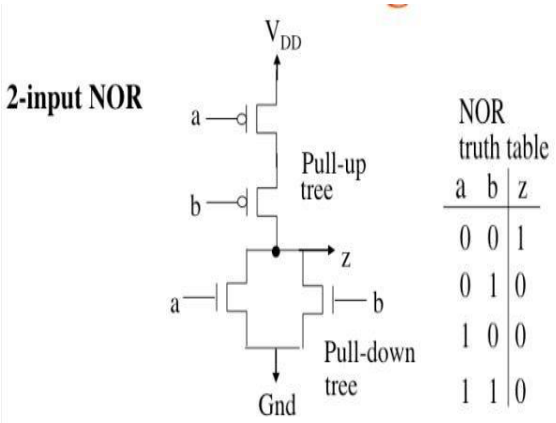
**Aim:** To design and implementation of universal gates

**Tools:** Tanner/ Mentor Graphics-Pyxis, AMS, Calibre

(i). NAND Gate:
Circuit Diagram:



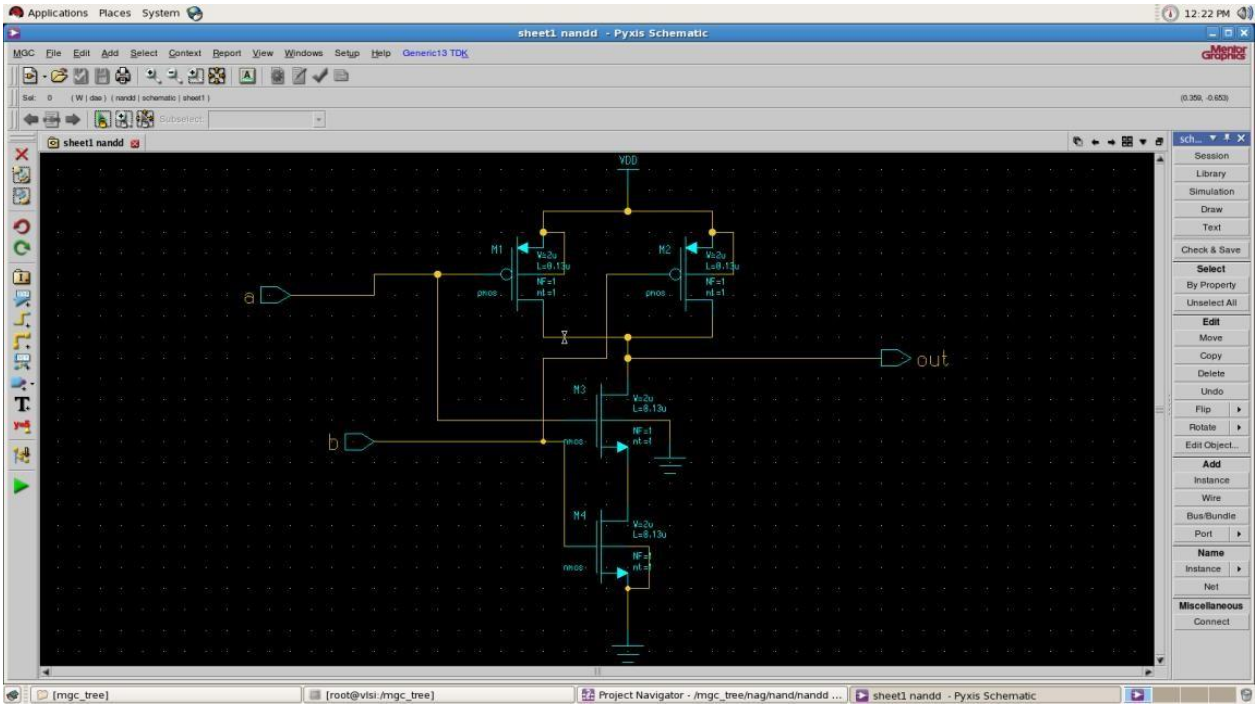

| A | B | F |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

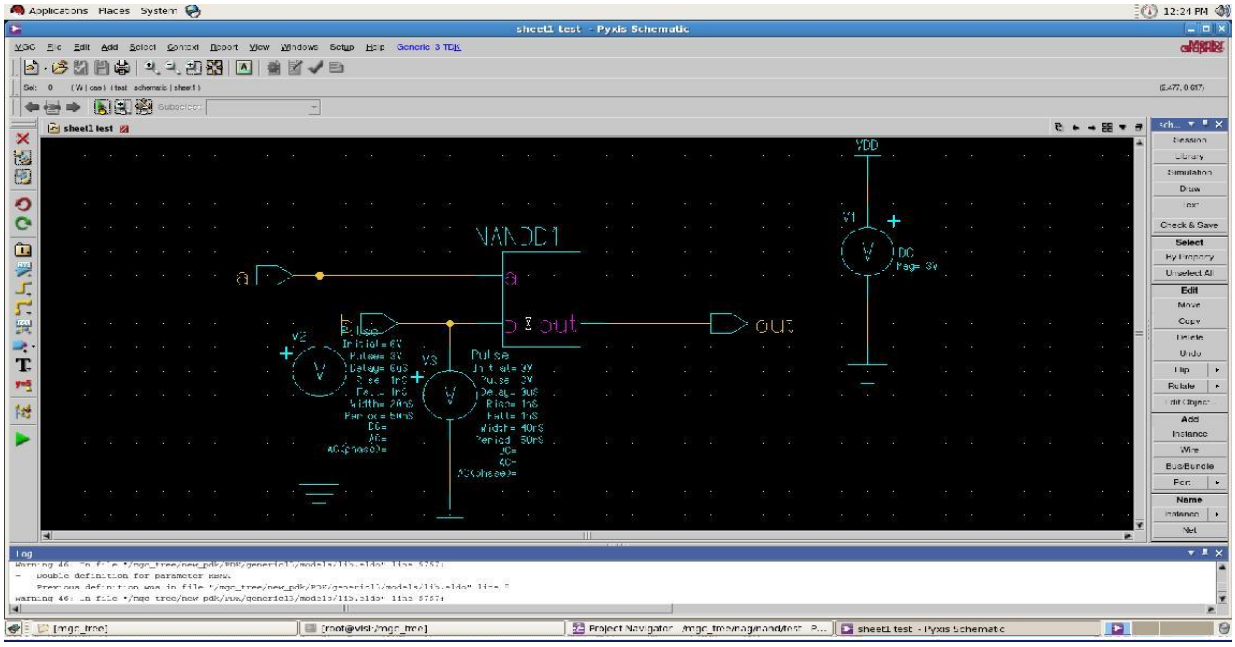



**PROCEDURE:**

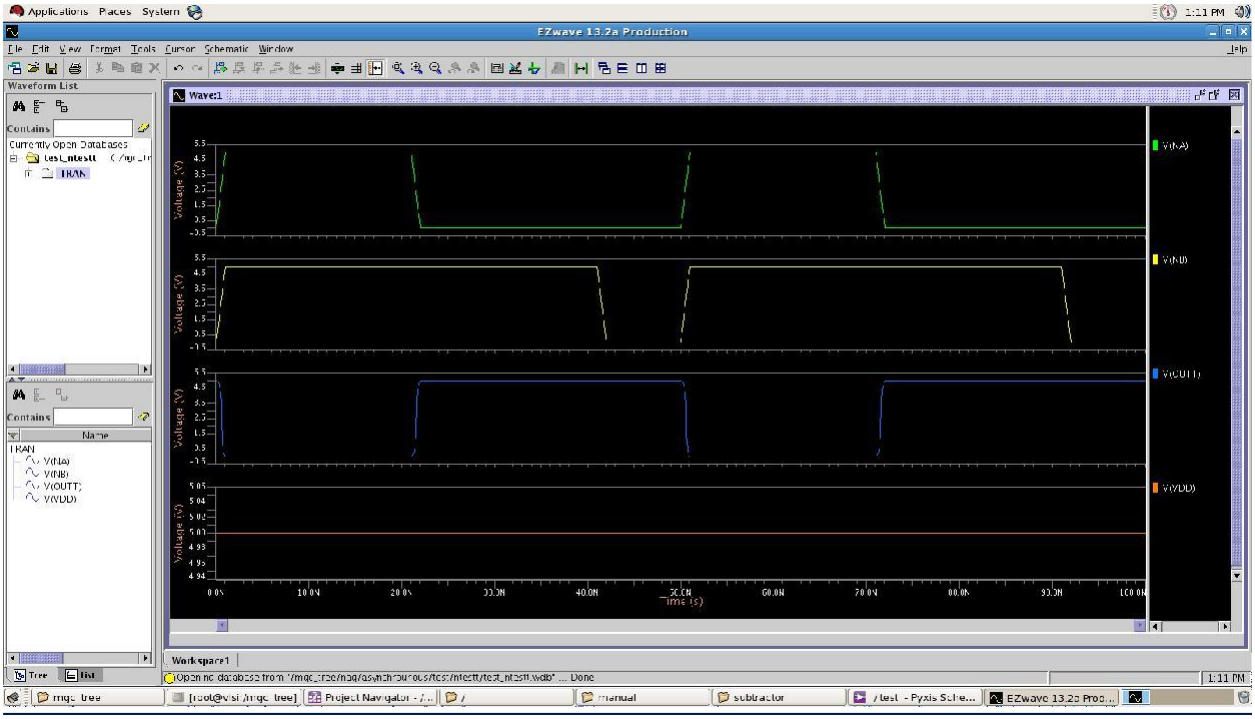
1. Connect the Circuit as shown in the circuit diagram using Pyxis Schematic tool
2. Enter into Simulation mode.
3. Setup the Analysis and library.
4. Setup the required analysis.
5. Probe the required Voltages
6. Run the simulation.
7. Observe the waveforms in EZ wave.

## Testbench:

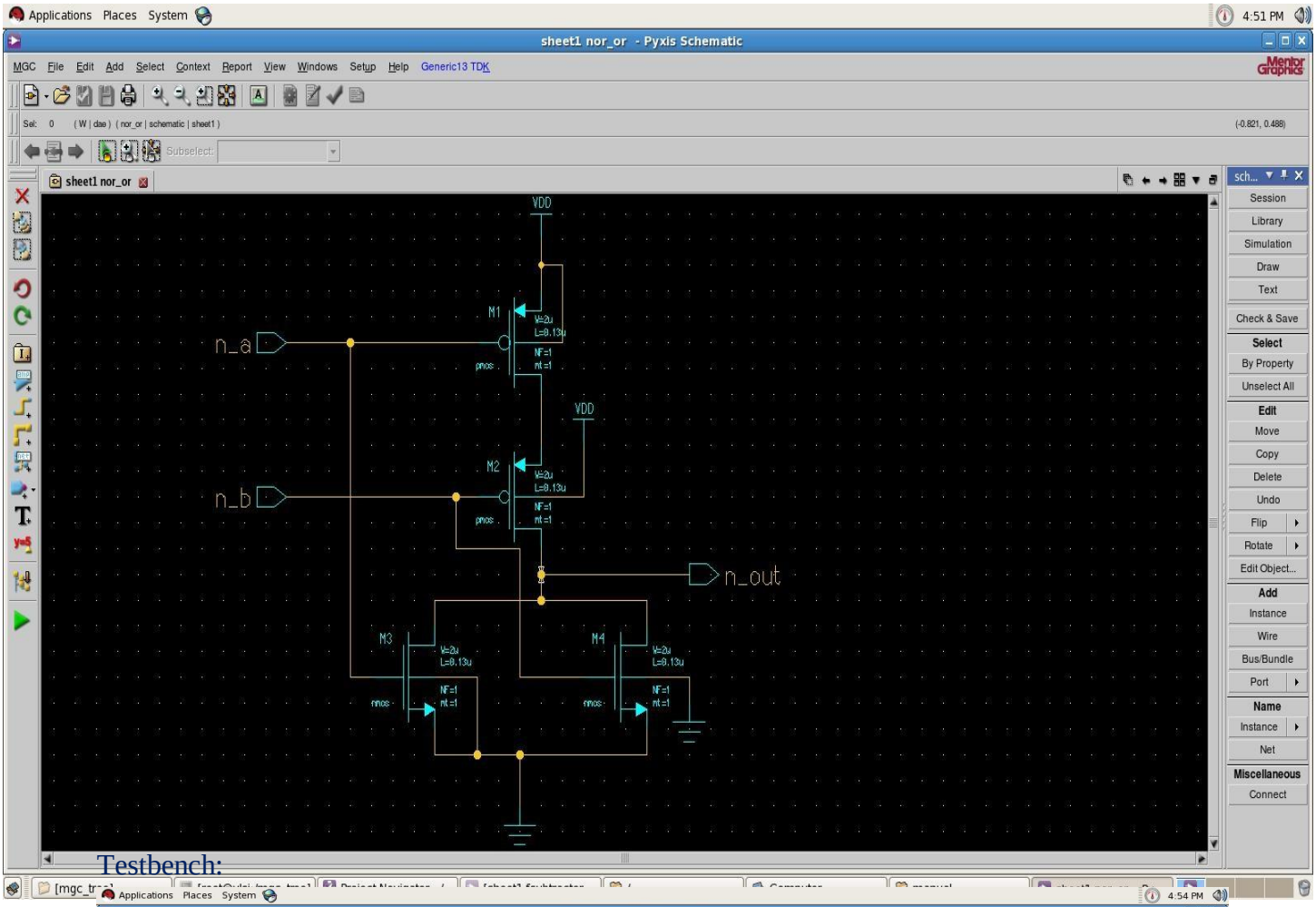

sheet1 nandd - Pyxis Schematic
VDD
M1
M2
M3
M4
a
b
out
nmos


sheet1 test - Pyxis Schematic
VDD
V1
VANDD1
a
b
out
V2
V3
Pulse

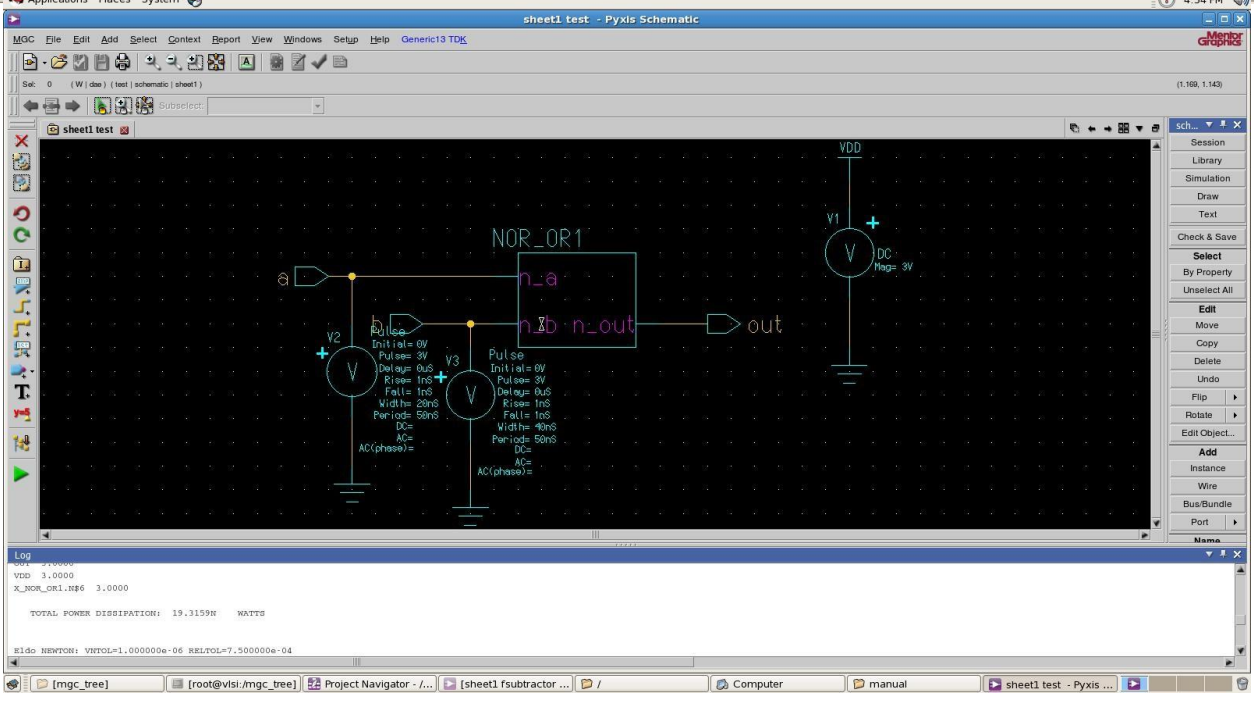
## Waveforms


EZwave 13.2a Production
Wave1
Waveform List
Contains
Currently Open Databases
Name
V(VDD)
Workspace1

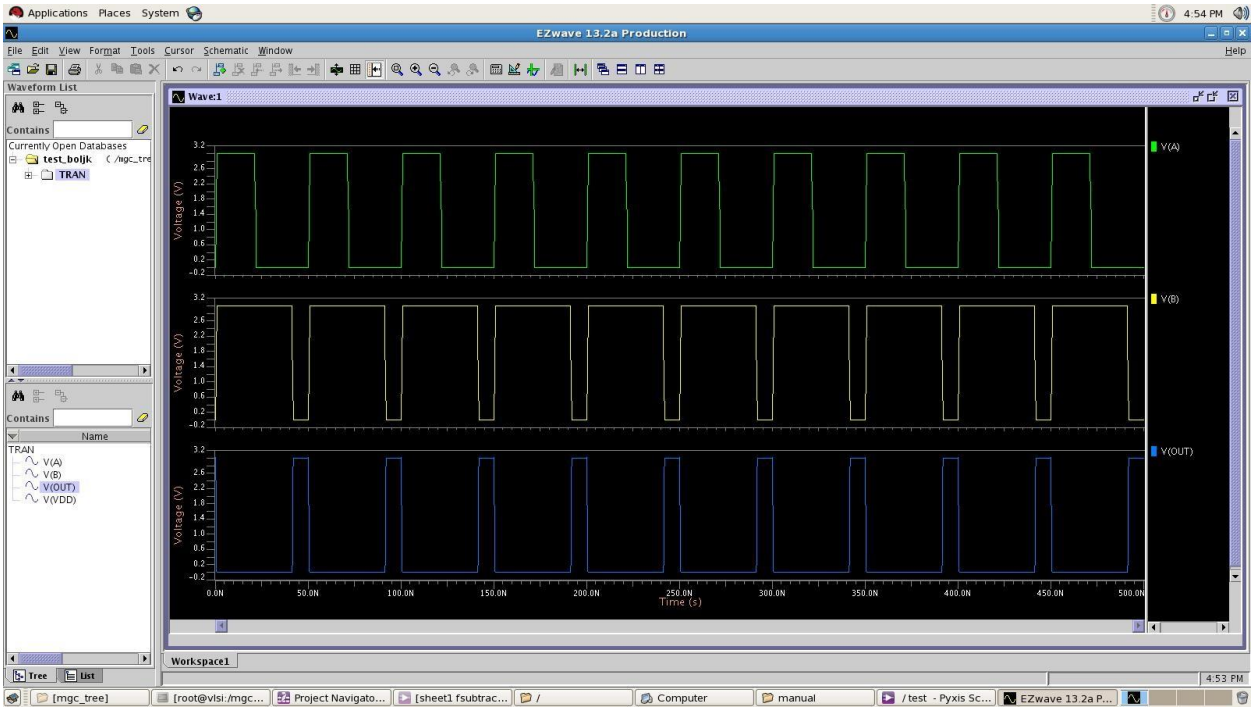
## NOR Gate:




## Testbench:



WAVEFORMS:




RESULT:

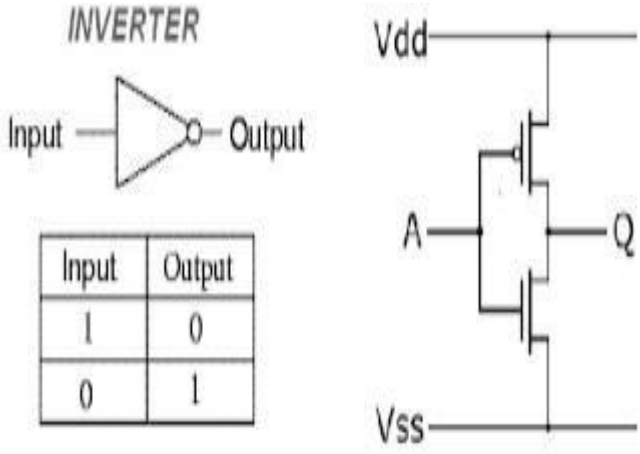
# Experiment 9: Design and Implementation of an Inverter

**AIM:** To design and Implementation of an Inverter

**TOOLS:** Tanner/ Mentor Graphics-Pyxis, AMS,

Calibre. **CIRCUIT**

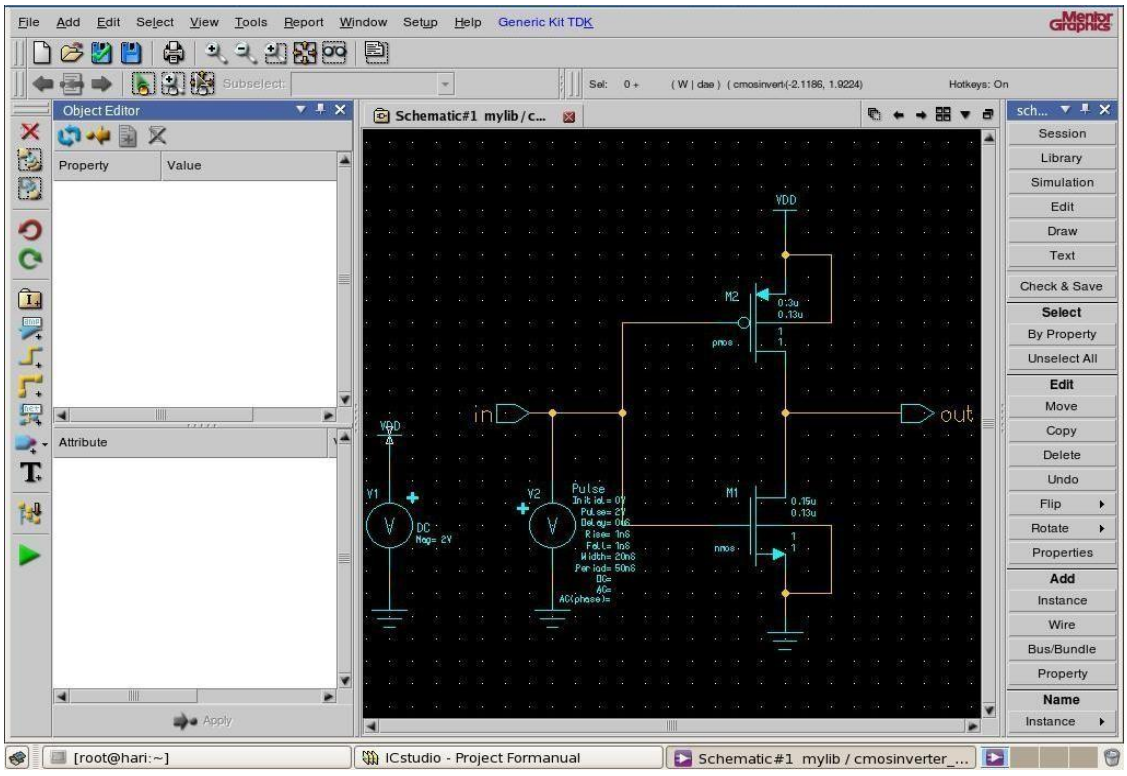
**DIAGRAM:**



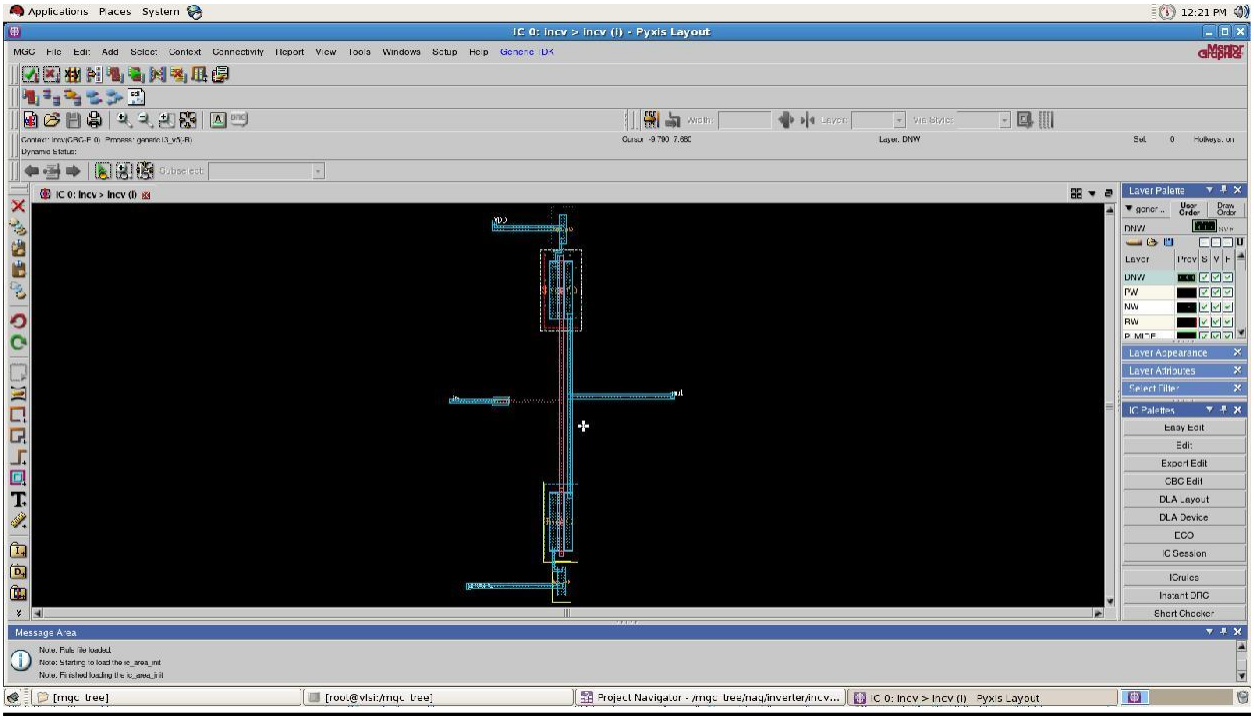
## PROCEDURE:

1. Connect the Circuit as shown in the circuit diagram using Pyxis Schematic tool
2. Enter into Simulation mode.
3. Setup the Analysis and library.
4. Setup the required analysis.
5. Probe the required Voltages
6. Run the simulation.
7. Observe the waveforms in EZ wave.
8. Draw the layout using Pysis Layout.
9. Perform Routing using IRoute
10. Perform DRC, LVS, PEX.

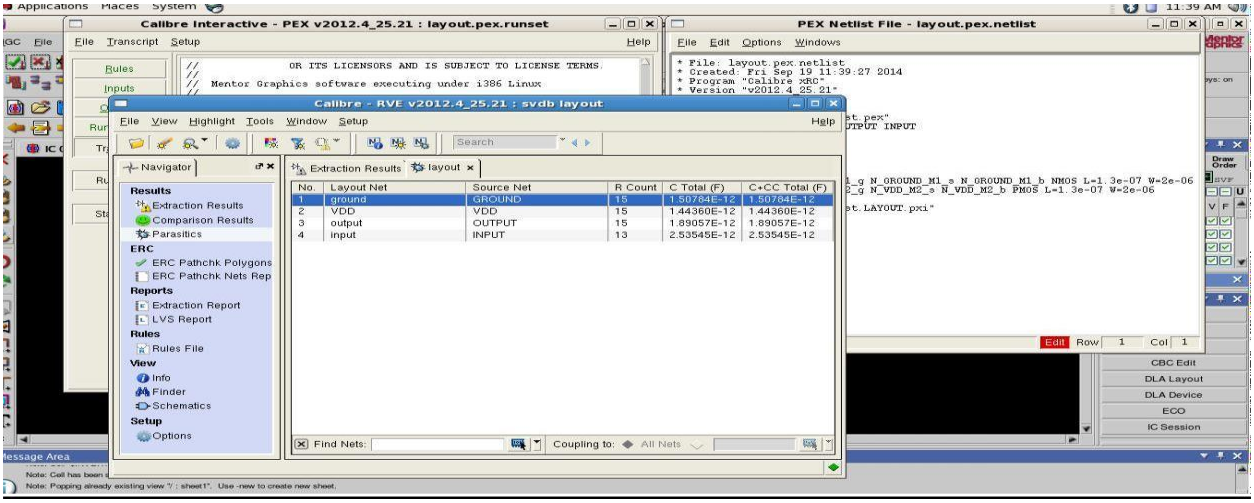
## Result:



## Layout:




## Result Verification Environment:

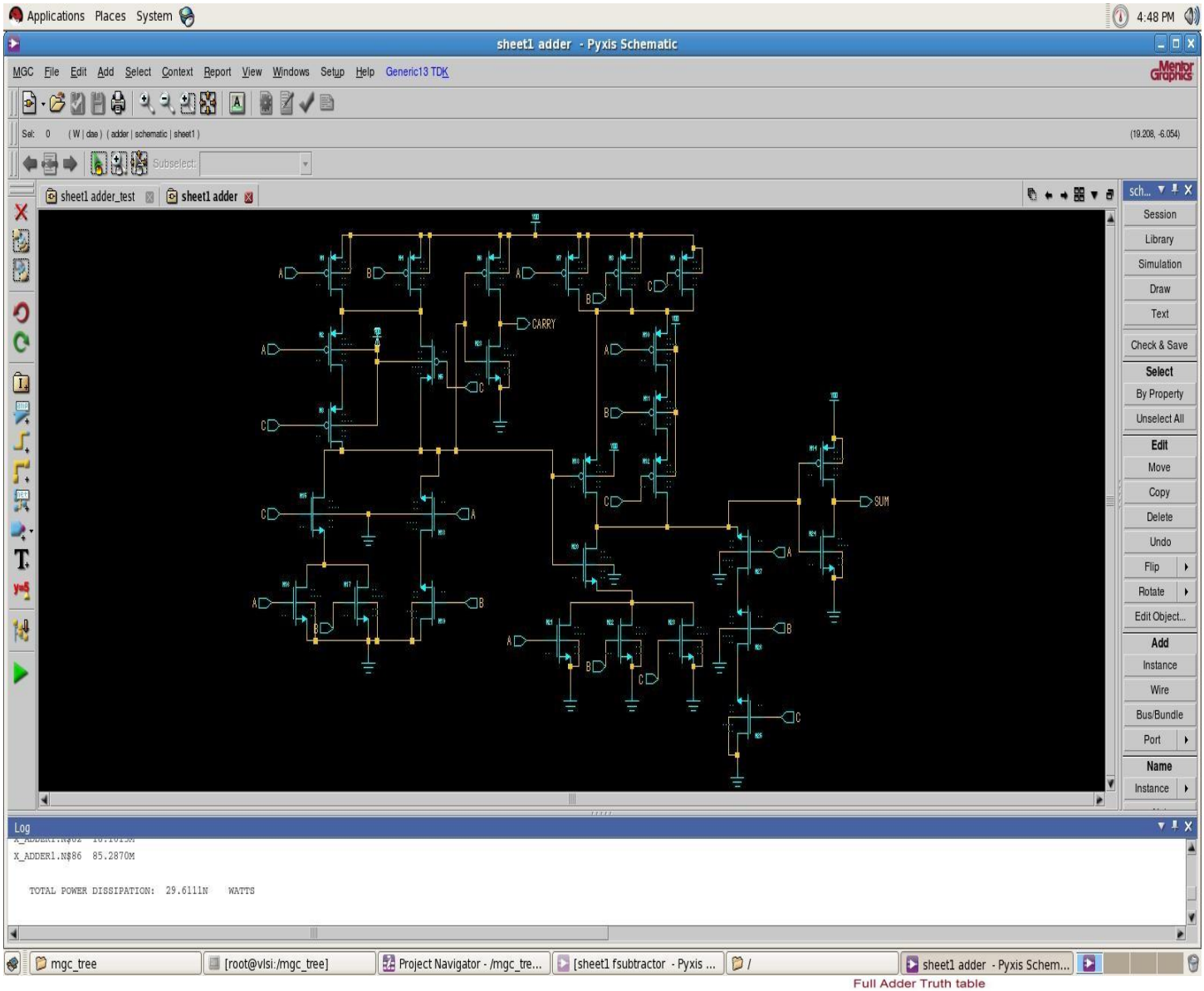


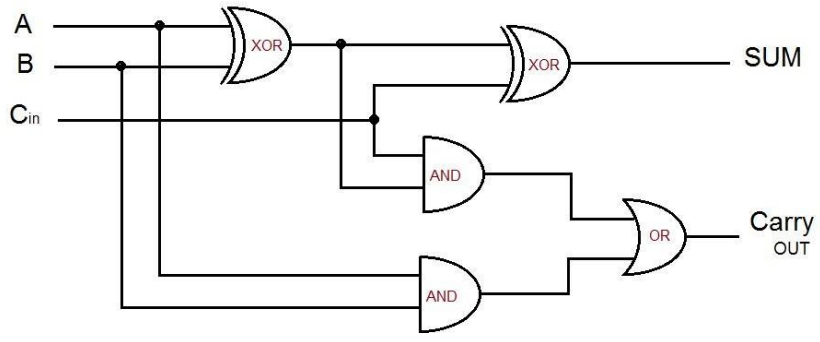
# Experiment 10: Design and Implementation of Full Adder

**AIM:** To design and Implementation of an Fulladder

**TOOLS:** Tanner/ Mentor Graphics-Pyxis, AMS, Calibre

**CIRCUIT DIAGRAM:**

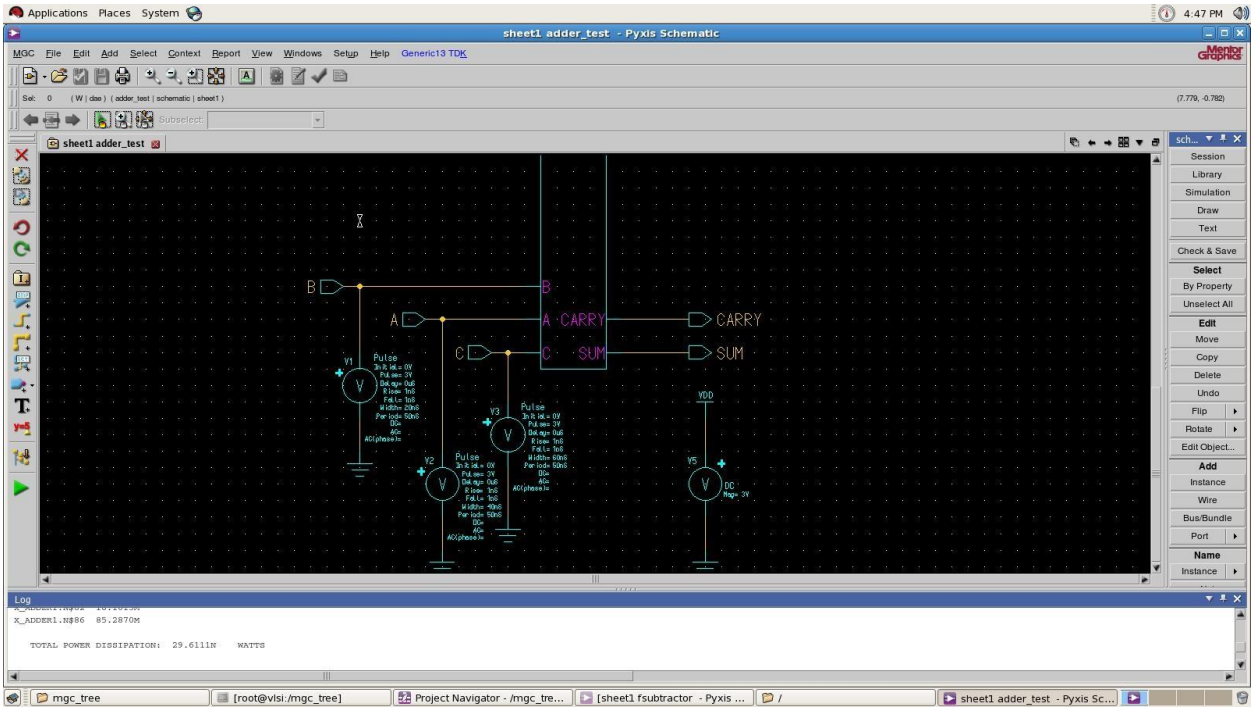


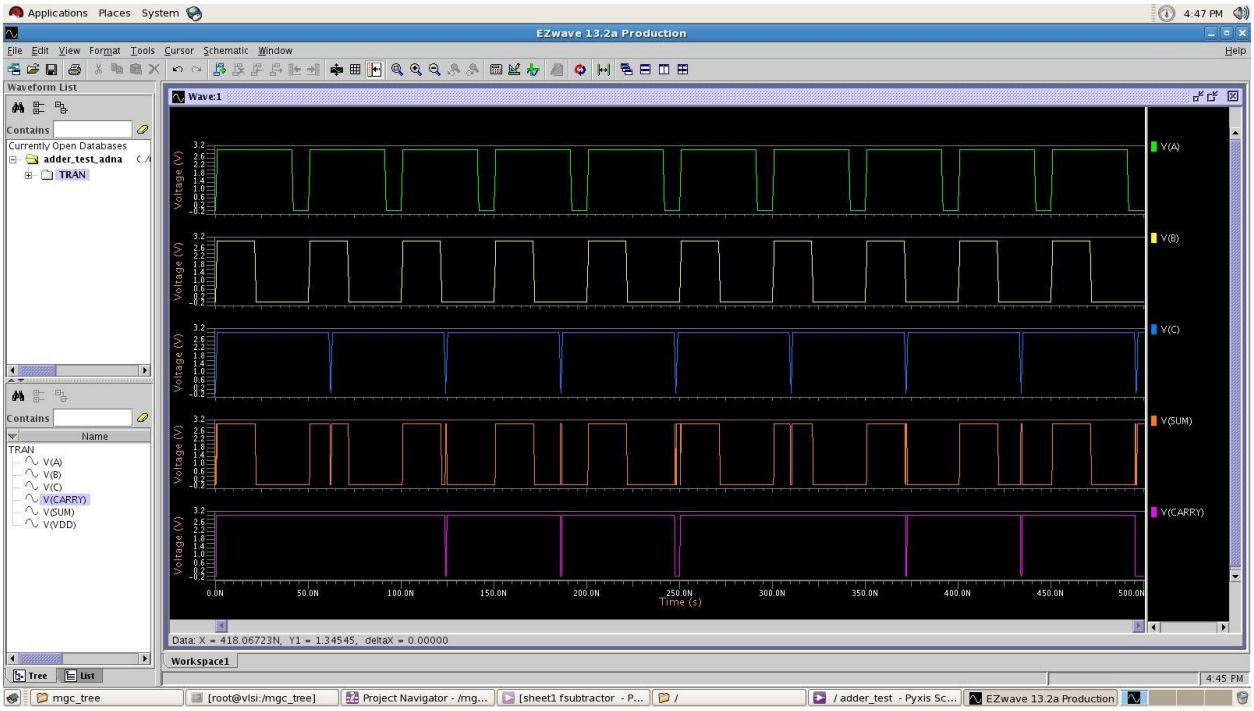

Full Adder Truth table

| INPUTS | | | OUTPUTS | |
|---|---|---|---|---|
| A | B | $C_{in}$ | SUM | CARRY 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 |

## Testbench:




## Waveforms:



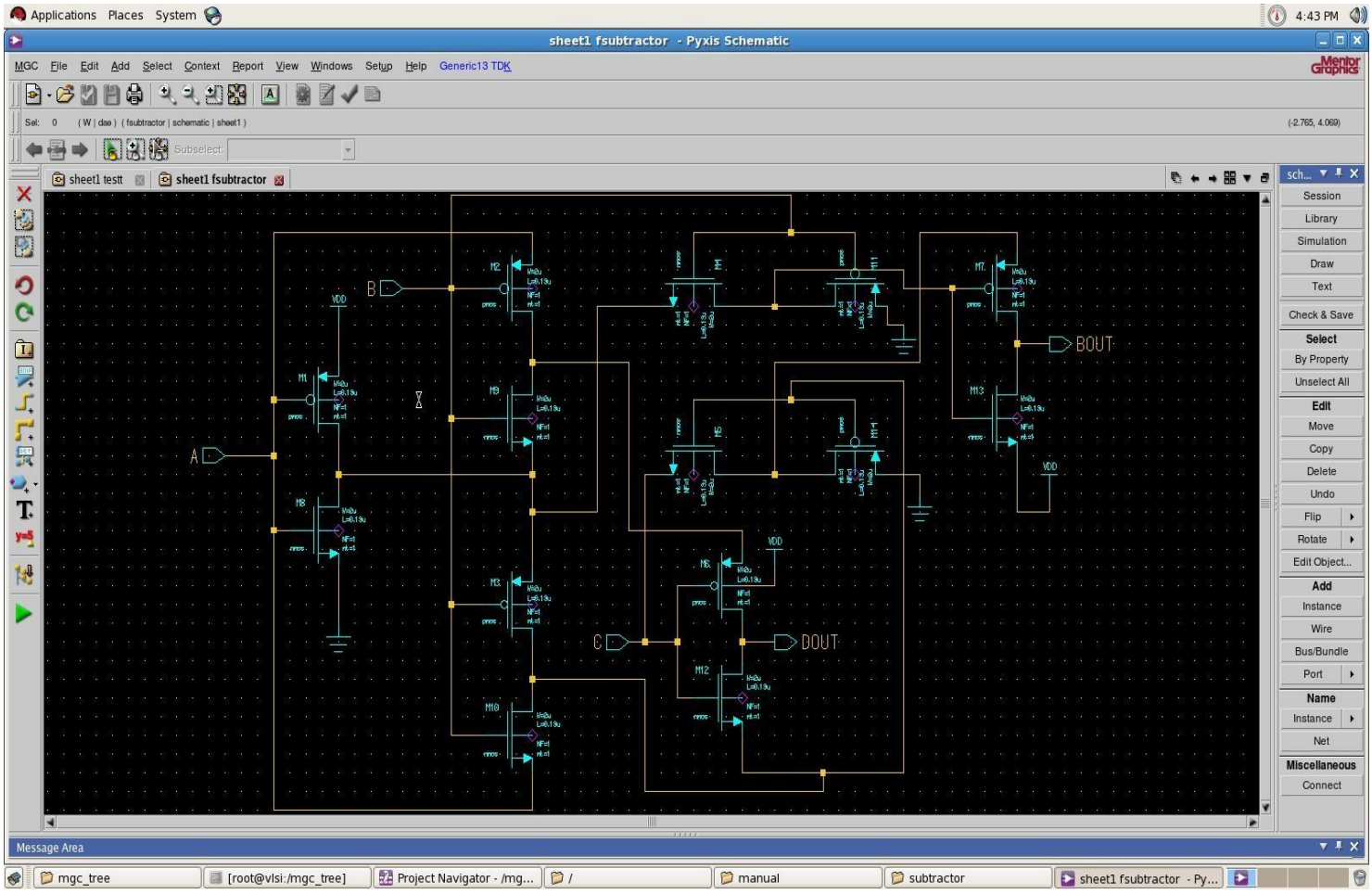

# Result:

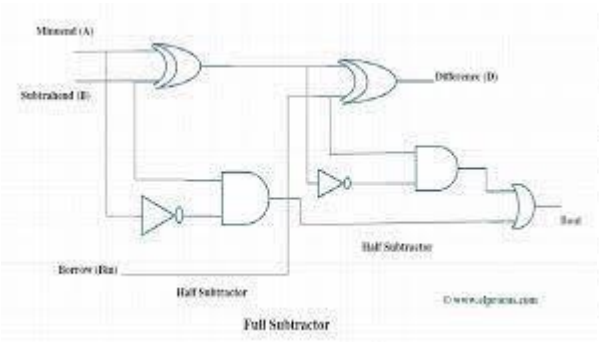
# Experiment 11: Design and Implementation of Full Subtractor

**AIM:** To design and Implementation of an Full-subtractor

**TOOLS:** Tanner/ Mentor Graphics-Pyxis, AMS, Calibre

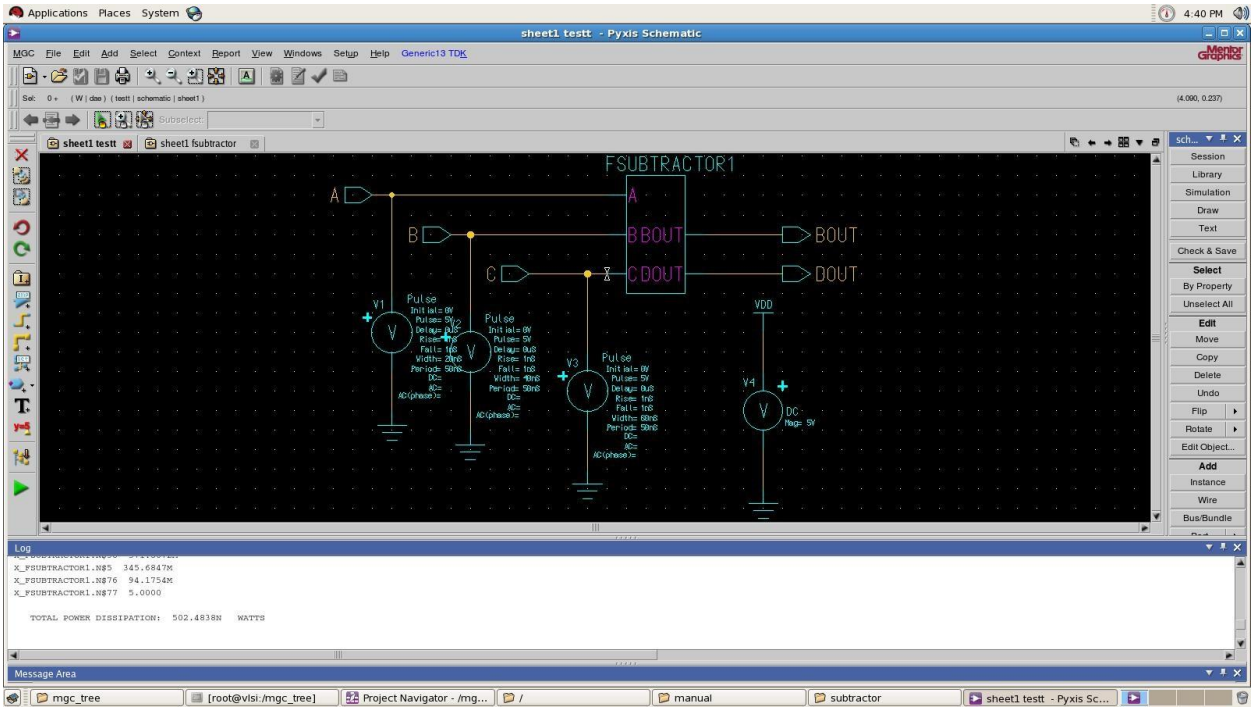
**CIRCUIT DIAGRAM:**



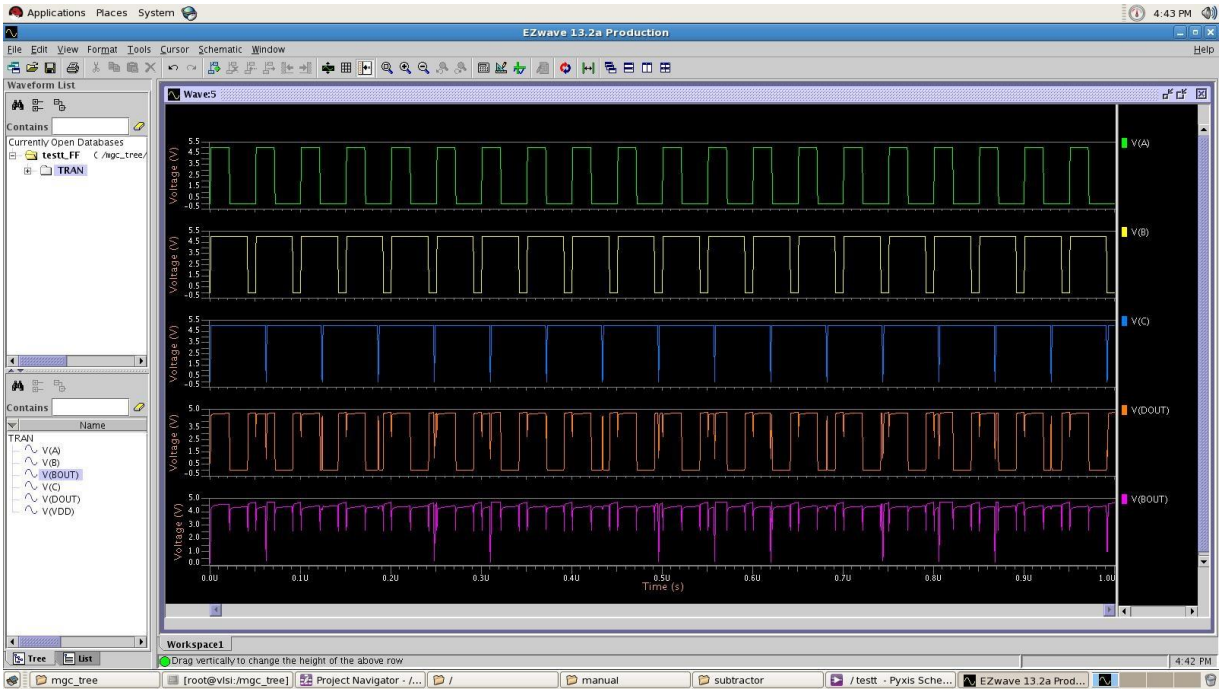




| INPUT | | | OUTPUT | |
|---|---|---|---|---|
| A | B | Bin | D | Bout |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

## Testbench:




**OUTPUT WAVEFORMS:**




**RESULT:**

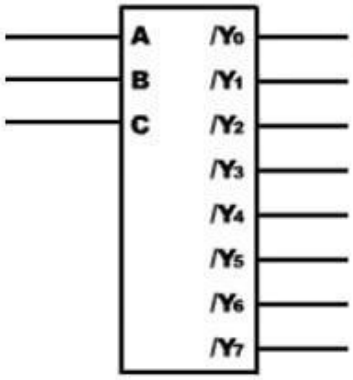
# Experiment 12: Design and Implementation of Decoder

**AIM:** To design and Implementation of Decoder

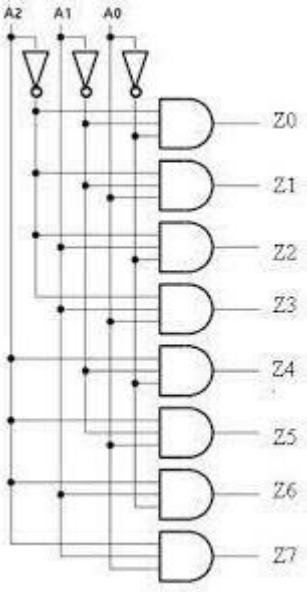
**TOOLS:** Tanner/ Mentor Graphics-Pyxis, AMS,

Calibre

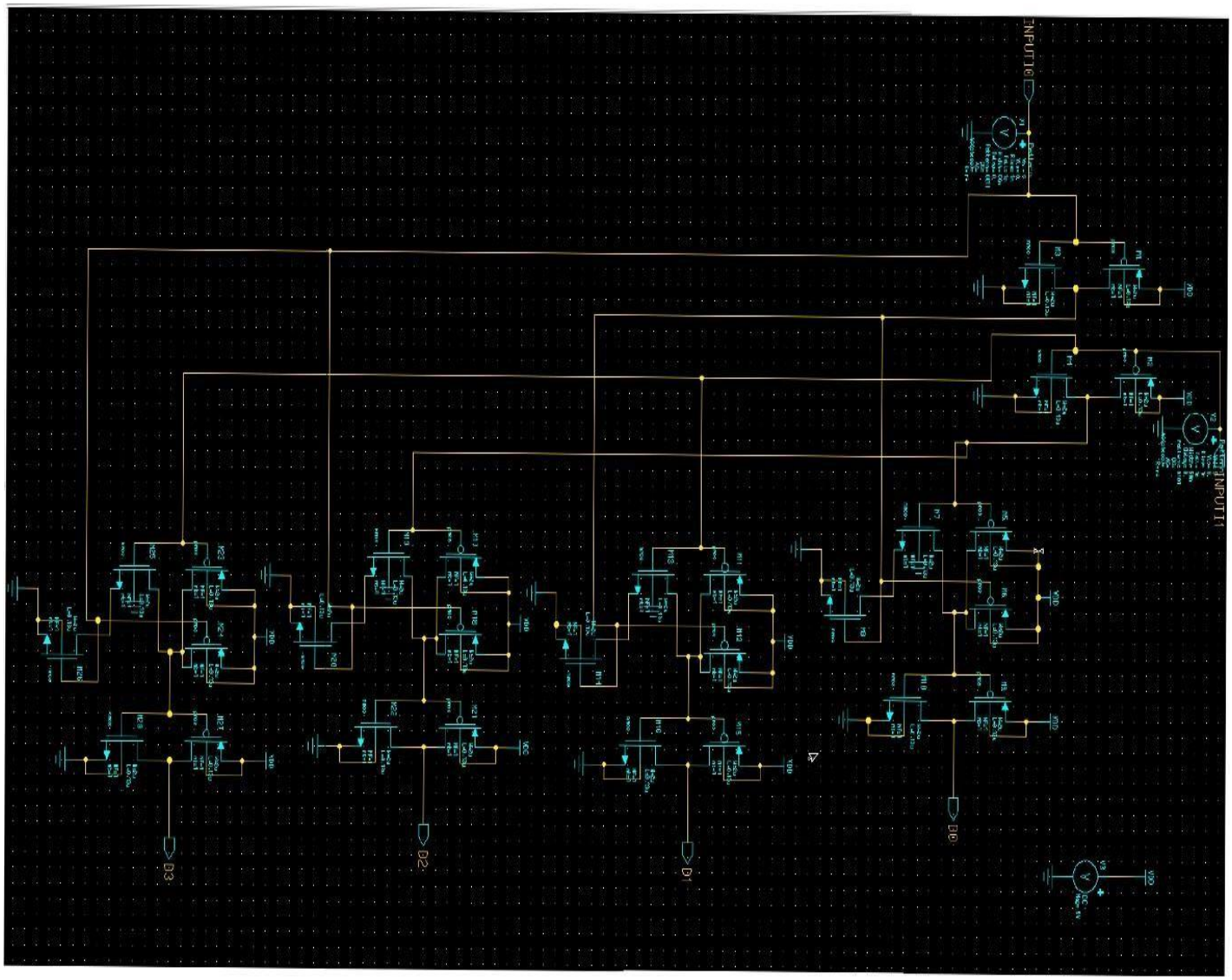
**CIRCUIT DIAGRAM:**



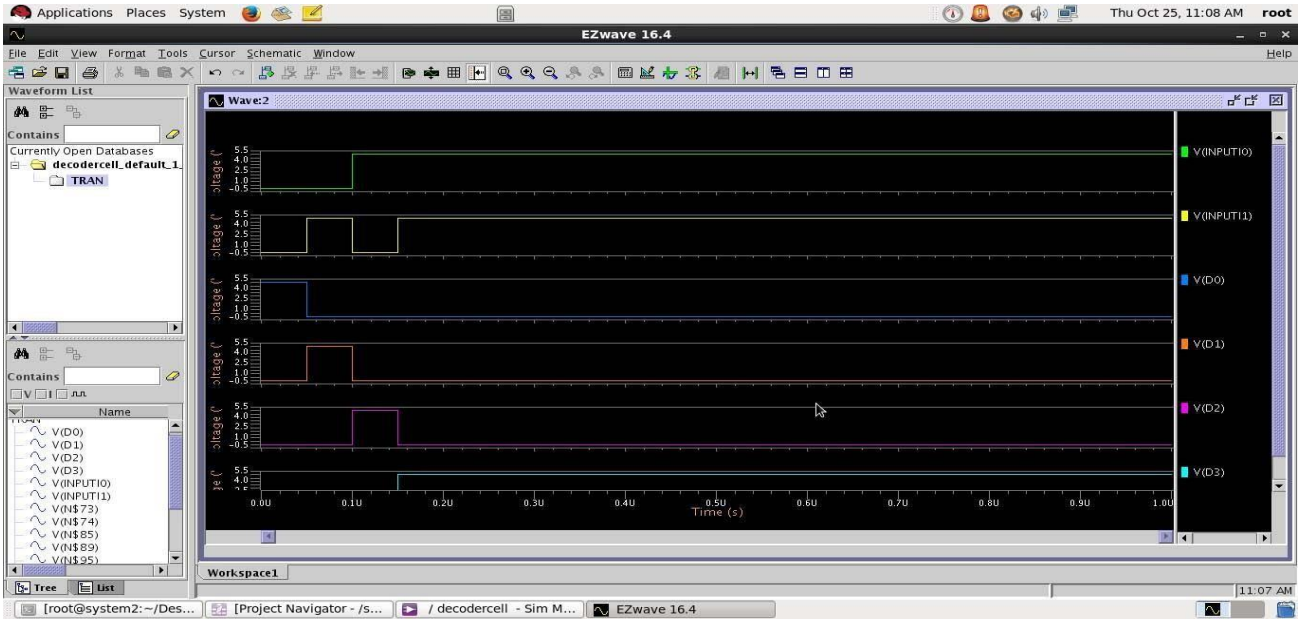

| Input | | | Output | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| A | B | C | /Y0 | /Y1 | /Y2 | /Y3 | /Y4 | /Y5 | /Y6 | /Y7 |
| L | L | L | L | H | H | H | H | H | H | H |
| H | L | L | H | L | H | H | H | H | H | H |
| L | H | L | H | H | L | H | H | H | H | H |
| H | H | L | H | H | H | L | H | H | H | H |
| L | L | H | H | H | H | H | L | H | H | H |
| H | L | H | H | H | H | H | H | L | H | H |
| L | H | H | H | H | H | H | H | H | L | H |
| H | H | H | H | H | H | H | H | H | H | L |



## Circuit diagram:



## WAVEFORMS:



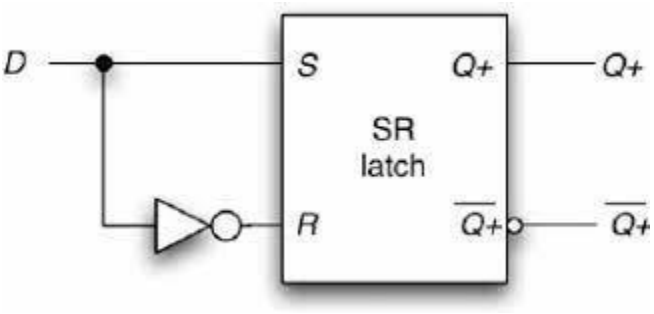

**RESULT:**

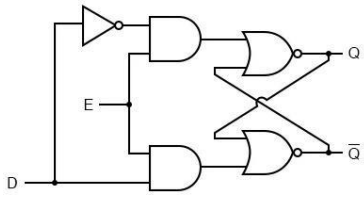
## Experiment 12 Design and Implementation of an D-Latch

**AIM:** To design and Implementation of an D-Latch

**TOOLS:** Tanner/ Mentor Graphics-Pyxis, AMS.

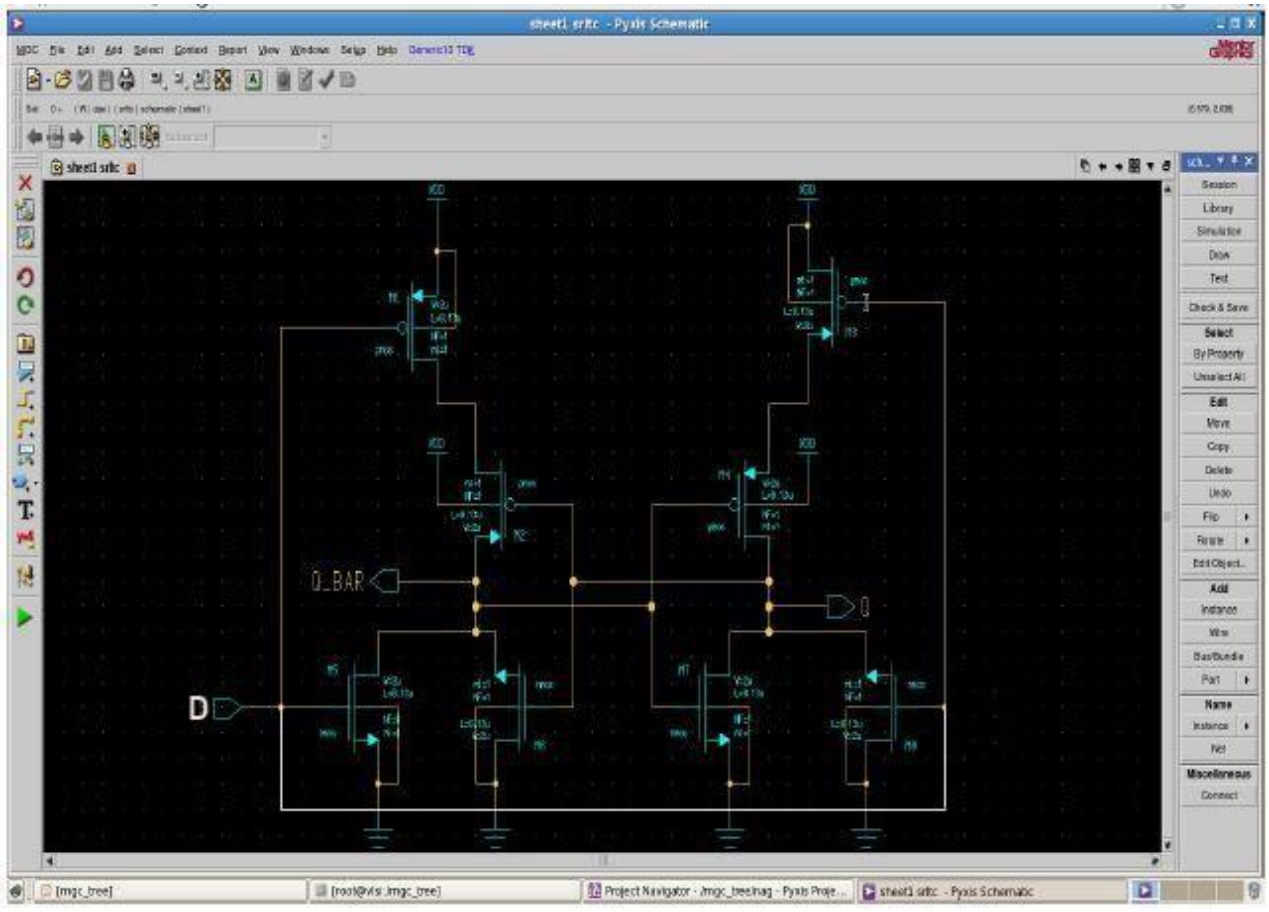
**CIRCUIT DIAGRAM:**



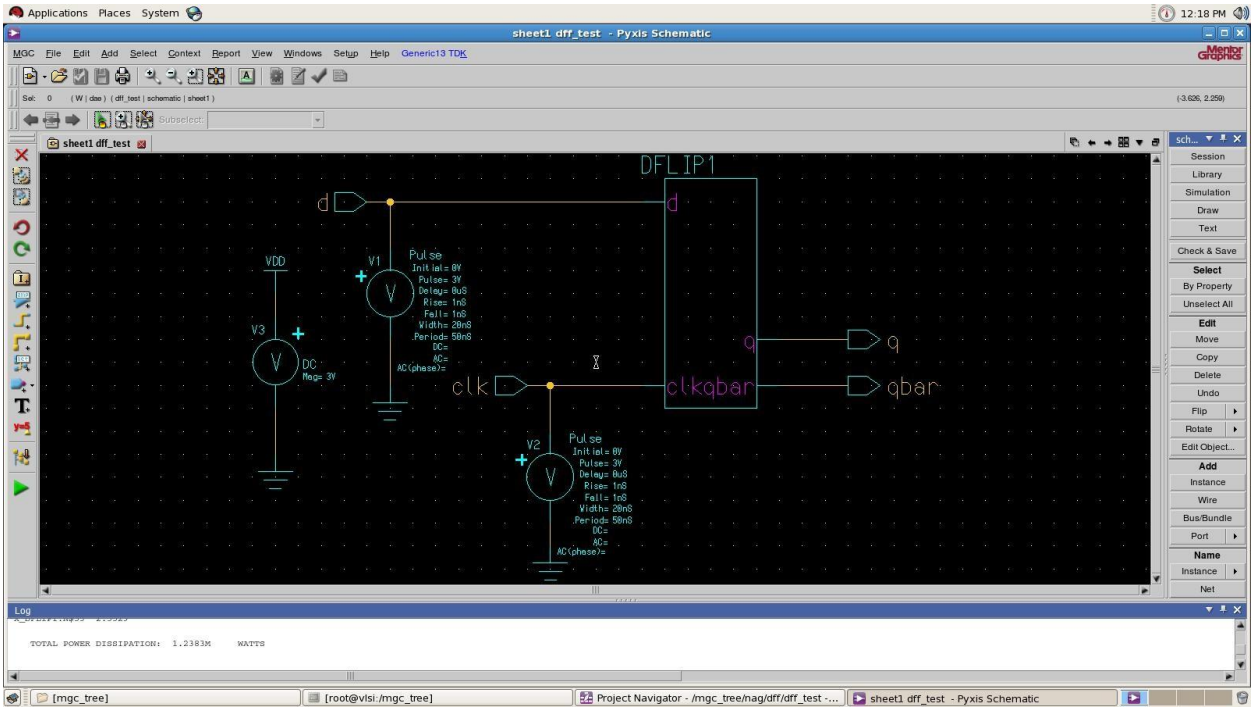




| E | D | Q | $\overline{Q}$ |
|---|---|---|---|
| 0 | 0 | latch | latch |
| 0 | 1 | latch | latch |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

**CIRCUIT DIAGRAM:**



## Testbench:




## WAVEFORMS:


## Result:

# Additional Experiment

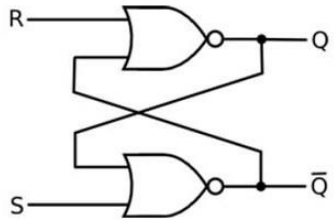
## Experiment 14: Design and Implementation of RS-Latch

**AIM:** To design and Implementation of RS Latch

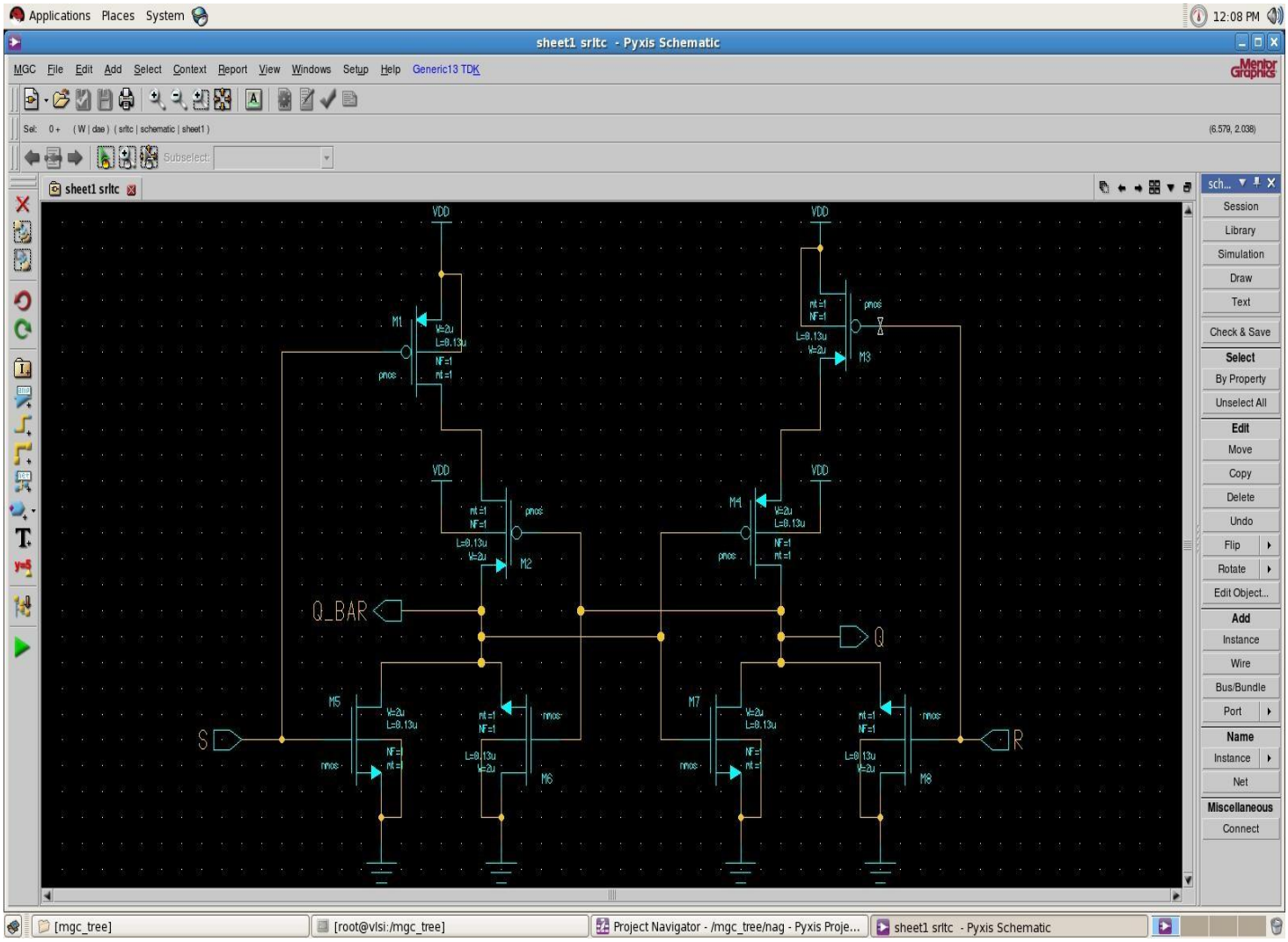
**TOOLS:** Tanner/Mentor Graphics - Pyxis, AMS,

Calibre.**CIRCUIT DIAGRAM:**

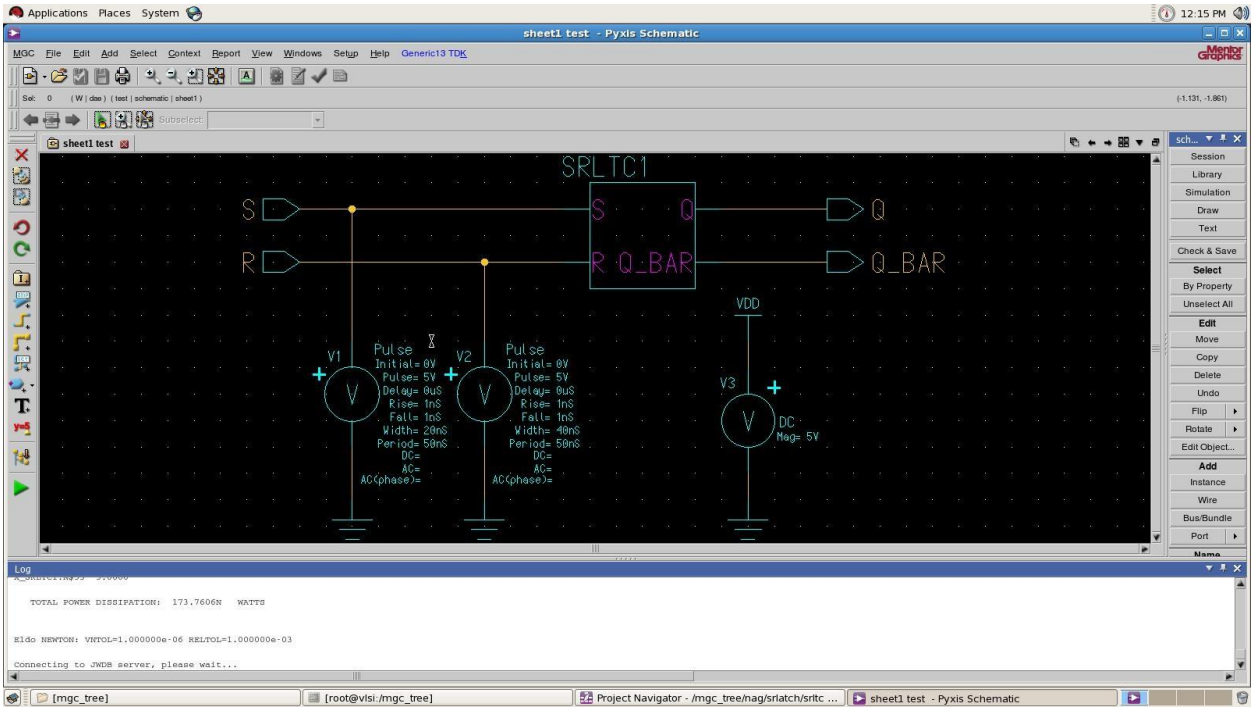
SR Latch based on NOR Gate



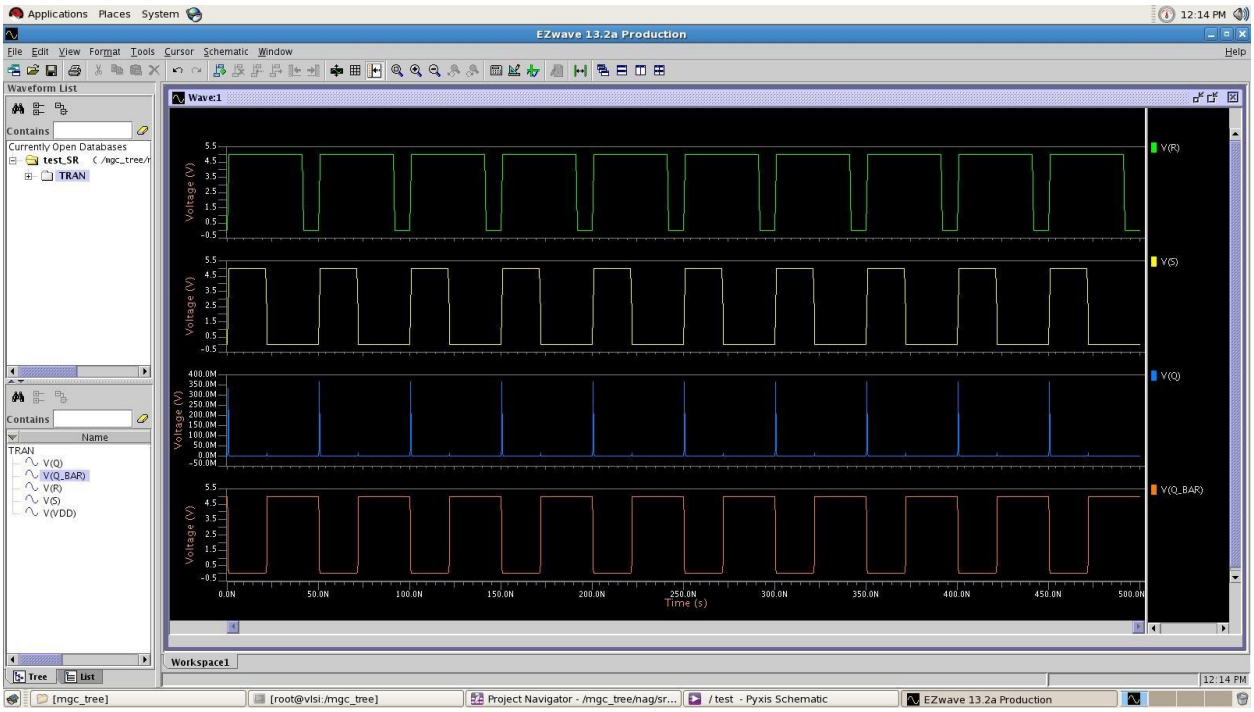

| S | R | Q | $\overline{Q}$ | Operation |
|---|---|---|---|---|
| 0 | 0 | Q | $\overline{Q}$ | Hold |
| 1 | 0 | 1 | 0 | Set |
| 0 | 1 | 0 | 1 | Reset |
| 1 | 1 | 0 | 0 | Not allowed |


Applications Places System
12:08 PM
sheet1 srltc - Pyxis Schematic
MGC File Edit Add Select Context Report View Windows Setup Help Generic13 TDK
sheet1 srltc
VDD
M1
M2
M3
M4
M5
M6
M7
M8
Q_BAR
Q
S
R
Session
Library
Simulation
Draw
Text
Check & Save
Select
By Property
Unselect All
Edit
Move
Copy
Delete
Undo
Flip
Rotate
Edit Object...
Add
Instance
Wire
Bus/Bundle
Port
Name
Instance
Net
Miscellaneous
Connect
[mgc_tree]
[root@vlsi:/mgc_tree]
Project Navigator - /mgc_tree/nag - Pyxis Proje...
sheet1 srltc - Pyxis Schematic

## Testbench:




## WAVEFORMS:




## Result:

Experiment Practice & Viva Answers