



**ACS** College of Engineering  
Approved by AICTE New Delhi, Affiliated to VTU, Belagavi  
(A Unit of RajaRajeswari Group of Institutions)



# **Department of Electronics and Communication Engineering**

## **HARDWARE DISCRIPTION LABORATORY MANUAL**

**Subject: Practical Components of IPCC(21EC32**

**Prepared by**

**Mrs. Vijaya Dalawai**

**Assistant Professor, Dept. Of ECE**

**Dr. H B Bhuvaneswari**

**HOD, Dept. Of ECE**



**Affiliated to Visvesvaraya Technological**

**University, Belagavi, Karnataka - 590018**

**2021-22**

**CONTENTS**

<b>Sl. NO</b>	<b>Title</b>
1	Syllabus
2	Cycles of Experiments
3	Overview of HDL lab
4	Introduction to FPGA
4	PART A - Combinational & Sequential Circuits Programs
5	PART B -Interfacing Programs
6	Viva Questions

**PROGRAM OUTCOMES (POS)**

**Engineering Graduates will be able to:**

**PO1. Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

**PO2. Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

**PO3. Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

**PO4. Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

**PO5. Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

**PO6. The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

**PO7. Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

**PO8. Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

**PO9. Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

**PO10. Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

**PO11. Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

**PO12. Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.  
*Acs college of engineering, Bangalore* *Dep. of ECE*

change.

**PROGRAM SPECIFIC OUTCOMES (PSOS)**

At the end of graduation the student will be able,

- To comprehend the fundamental ideas in Electronics and Communication Engineering and apply them to identify, formulate and effectively solve complex engineering problems using latest tools and techniques.
- To work successfully as an individual pioneer, team member and as a leader in assorted groups, having the capacity to grasp any requirement and compose viable solutions.
- To be articulate, write cogent reports and make proficient presentations while yearning for continuous self-improvement.
- To exhibit honesty, integrity and conduct oneself responsibly, ethically and legally; holding the safety and welfare of the society paramount.

**Program Educational Objectives (PEOs)**

- Graduates will have a successful professional career and will be able to pursue higher education and research globally in the field of Electronics and Communication Engineering thereby engaging in lifelong learning.
- Graduates will be able to analyse, design and create innovative products by adapting to the current and emerging technologies while developing a conscience for environmental/ societal impact.
- Graduates with strong character backed with professional attitude and ethical values will have the ability to work as a member and as a leader in a team.
- Graduates with effective communication skills and multidisciplinary approach will be able to redefine problems beyond boundaries and develop solutions to complex problems of today's society.

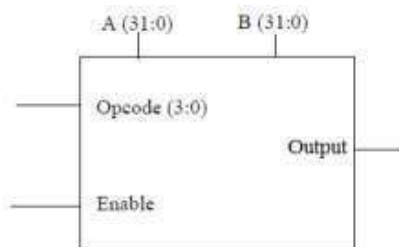
## HDL LABORATORY (18ECL58)

<b>Subject Code</b>	<b>:</b>	<b>18ECL58</b>	<b>I.A. Marks</b>	<b>:</b>	<b>40</b>
<b>Hours/Week</b>	<b>:</b>	<b>03</b>	<b>Exam Hours</b>	<b>:</b>	<b>03</b>
<b>Total Hours</b>	<b>:</b>	<b>36</b>	<b>Exam Marks</b>	<b>:</b>	<b>60</b>

**VTU SYLLABUS****PART A**

- 1 Write a Verilog program for the following combinational designs
  - a. 2 to 4 decoder
  - b. 8 to 3 (encoder without priority & with priority)
  - c. 8 to 1 multiplexer.
 4 bit binary to gray converter
- 2 Model in Verilog for a full adder and add functionality to perform logical operations of XOR, XNOR, AND and OR gates. Write test bench with appropriate input patterns to verify the modeled behaviour.

- 3 Write a Verilog code to model 32 bit ALU using the schematic diagram shown Below



- ALU should use combinational logic to calculate an output based on the four bit op-code input.
- ALU should pass the result to the out bus when enable line in high, and tri-state the out bus when the enable line is low.
- ALU should decode the 4 bit op-code according to the example given below.

OPCODE	ALU Operation
1.	A+B
2.	A-B
3.	A Complement
4.	A*B
5.	A AND B

		6.	A OR B	
		7.	A NAND B	
		8.	A XOR B	
4	Write Verilog code for SR, D and JK and verify the flip flop.			
5	Write Verilog code for 4-bit BCD synchronous counter.			
6	Write Verilog code for counter with given input clock and check whether it works as clock divider performing division of clock by 2, 4, 8 and 16. Verify the functionality of the code.			
	<b>PART-B</b>			
7	Write a Verilog code to design a clock divider circuit that generates 1/2, 1/3rd and 1/4th clock from a given input clock. Port the design to FPGA and validate the functionality through oscilloscope.			
8	Interface a DC motor to FPGA and write Verilog code to change its speed and direction.			
9	Interface a Stepper motor to FPGA and write Verilog code to control the Stepper motor rotation which in turn may control a Robotic Arm. External switches to be used for different controls like rotate the Stepper motor (i) +N steps if Switch no.1 of a Dip switch is closed (ii) +N/2 steps if Switch no. 2 of a Dip switch is closed (iii) -N steps if Switch no. 3 of a Dip switch is closed etc.			
10	Interface a DAC to FPGA and write Verilog code to generate Sine wave of frequency F KHz (eg. 200			
11	KHz) frequency. Modify the code to down sample the frequency to F/2 KHz. Display the Original and			
12	Down sampled signals by connecting them to an oscilloscope.			

## Introduction to HDL

An HDL is a programming language used to describe electronic circuit essentially digital logic circuits. It can be used to describe the operation, design and organization of a digital circuit. It can also be used to verify the behaviour by means of simulations. The principle difference between HDL and other programming languages is that HDL is a concurrent language whereas the others are procedural i.e. single threaded. HDL has the ability to model multiple parallel processes like adders, flip-flops etc which execute automatically and independently of each other. It is like building many circuits that can operate independently of each other.

The two widely used HDLs are:

- VHDL: Very High Speed Integrated Circuits HDL
- Verilog HDL

**VHDL (VHSIC Hardware Description Language)** is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits. **VHDL** can also be used as a general purpose parallel programming language.

Verilog, standardized as IEEE 1364, is a hardware description language (HDL) used to model electronic systems. It is most commonly used in the design and verification of digital circuits at the register-transfer level of abstraction. It is also used in the verification of analog circuits and mixed-signal circuits, as well as in the design of genetic circuits.

## Difference between Verilog and VHDL

1. VHDL is based on Pascal and ADA while Verilog is based on C language.
2. VHDL is strongly typed i.e., does not allow the intermixing, or operation of variables, with different classes whereas Verilog is weakly typed.
3. VHDL is case insensitive and Verilog is case sensitive.
4. Verilog is easier to learn compared to VHDL.
5. Verilog has very simple data types, while VHDL allows users to create more complex data types.
6. Verilog lacks the library management, like that of VHDL.

## FPGA DESIGN FLOW

1. **Design Entry** – the first step in creating a new design is to specify its structure and functionality. This can be done either by writing an HDL model using some text editor or drawing a schematic diagram using schematic editor.
2. **Design Synthesis** – next step in the design process is to transform design specification into a more suitable representation that can be further processed in the later stages in the design flow. This representation is called the netlist. Prior to netlist creation synthesis tool checks the model syntax and analyse the hierarchy of your design which ensures that your design is optimized for the design architecture you have selected. The resulting netlist is saved to a Native Generic Circuit (NGC) file (for Xilinx® Synthesis Technology (XST) compiler) or an Electronic Design Interchange Format (EDIF) file (for Precision, or Synplify/Synplify Pro tools).



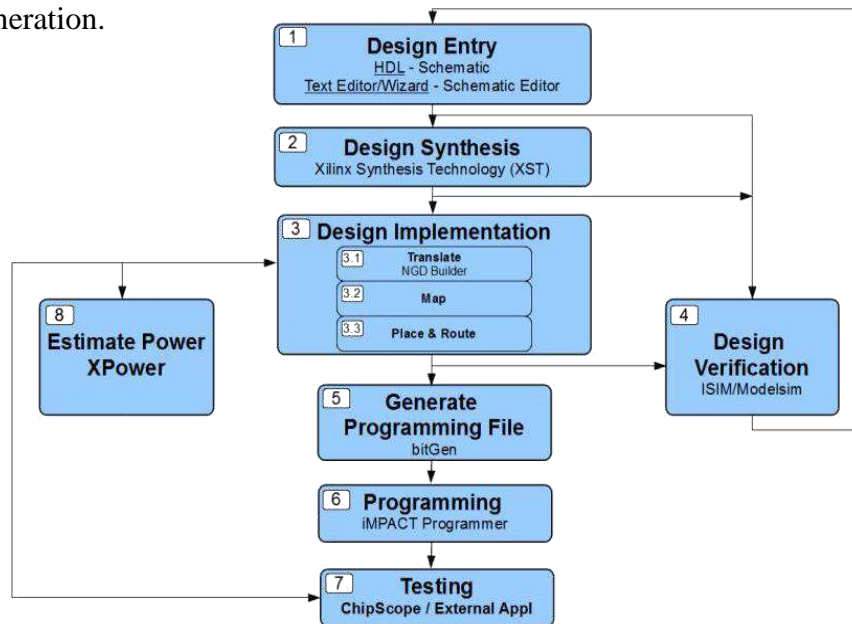
### 3. Design Implementation

Implementation step maps netlist produced by the synthesis tool onto particular device's internal structure. It consists from three steps:

**Translate step** – merges all incoming netlists and constraints into a Xilinx Native Generic Database (NGD) file.

**Map step** - maps the design, specified by an NGD file, into available resources on the target FPGA device, such as LUTs, Flip-Flops, BRAMs,... As a result, an Native Circuit Description (NCD) file is created.

**Place and Route step** - takes a mapped Native Circuit Description (NCD) file, places and routes the design, and produces an NCD file that is used as input for bit stream generation.



**Figure: FPGA Design Flow**

4. **Design Verification** – is very important step in design process. Verification is comprised of seeking out problems in the HDL implementation in order to make it compliant with the design specification. A verification process reduces to extensive simulation of the HDL code. Design Verification is usually performed using two approaches: Simulation and Static Timing Analysis.

There are two types of simulation:

- **Functional (Behavioral) Simulation** – enables you to simulate or verify a code syntax and functional capabilities of your design. This type of simulation tests your design decisions before the design is implemented and allows you to make any necessary changes early in the design process. In functional (behavioral) simulation no timing information is provided.
- **Timing Simulation** – allows you to check does the implemented design meet all functional and timing requirements and behaves as you expected. The timing simulation uses the detailed information about the signal delays as they pass through various logic and memory components and travel over connecting wires. Using this information it is possible to accurately simulate the behaviour of the implemented design. This type of simulation is performed after the design has been placed and routed for the target PLD, because accurate signal delay information can now be estimated. A process of relating

accurate timing information with simulation model of the implemented design is called Back-Annotation.

- **Static Timing Analysis** – helps you to perform a detailed timing analysis on mapped, placed only or placed and routed FPGA design. This analysis can be useful in evaluating timing performance of the logic paths, especially if your design doesn't meet timing requirements. This method doesn't require any type of simulation.
5. **Generate Programming File** – this option runs BitGen, the Xilinx bitstream generation program, to create a bitstream file that can be downloaded to the device.
  6. **Programming** – iMPACT Programmer uses the output from the Generate Programming File process to configure your target device.
  7. **Testing** – after configuring your device, you can debug your FPGA design using the Xilinx ChipScope Pro tool or some external logic analyzer.
  8. **Estimate Power** – after implementation, you can use the XPower Analyzer for estimation and power analysis. XPower Analyzer is delivered with ISE Design Suite. With this tool you can estimate power, based on the logic and routing resources of the actual design.

## ABOUT XILINX ISE SOFTWARE

Xilinx ISE (Integrated Synthesis Environment) is a software tool produced by Xilinx for synthesis and analysis of HDL designs, enabling the developer to synthesize ("compile") their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer.

Xilinx ISE is a design environment for FPGA(Field programmable gate arrays) products from Xilinx, and is tightly-coupled to the architecture of such chips, and cannot be used with FPGA products from other vendors. The Xilinx ISE is primarily used for circuit synthesis and design, while ISIM or the ModelSim logic simulator is used for system-level testing

## STEPS TO EXECUTE A PROGRAM

1) Starting the ISE software

Start \_ program \_ XILINX ISE 7 \_ Project Navigator

2) Creating a New Project in ISE

A project is a collection of all files necessary to create and to download a design to a selected FPGA or CPLD devices.

Project name:

**Project location:**

Top-Level Source Type: HDL

Click **Next** to move to the project properties page.

3) Fill in the properties in the table as shown below

**Device Family:** Spartan 3

**Device:** XC3S50

**Package:** PQ208Speed

**Speed:** -5

Top-Level Module Type: HDL

**HDL Synthesis Tool: XST(VHDL/VERILOG)****Simulator: ISE Simulator (VHDL/ Verilog)**

## 4) Creating an HDL Source

Create a top-level HDL file for the design. Determine the language that you wish to use (Verilog module or VHDL module).

This simple AND Gate design has two inputs: A and B. This design has one output called C

- ☐ Click New Source in the New Project Wizard to add one new source to your project.
- a) Select **VERILOG MODULE** as the source type in the New Source dialog box.
- b) Type in the file name **for ex:** and\_gate
- c) Verify that the Add to project checkbox is selected.
- d) Click Next.
- e) Define the ports for your Verilog source.

In the Port Name column, type the **port names** on three separate rows: A, B and C.

In the Direction column, indicate whether **each port is an input, output, or inout**.

For A and B, select in from the list. For C, select out from the list.

## 5) Click next in the Define Verilog Source dialog box.

6) Click Finish in the New Source Information dialog box to complete the new source file template. Click Next in the New Project Wizard. Click next again.

## 7) Click Finish in the New Project Information dialog box.

ISE creates and displays the new project in the Sources in Project window and adds the and\_gate.v file to the project.

8) Double-click on the **and\_gate.v** file in the Sources in Project window to open the Verilog file in the ISE Text Editor.

The and\_gate.v file contains:

Module name with the inputs and outputs declared.

9) Add the relationship between input and output after the input and output declared in module. Save the file by selecting File > Save.

10) When the source files are complete, the next step is to check the syntax of the design. Syntax errors and typos can be found using this step.

- a) Select the counter design source in the **ISE Sources window** to display the related processes in the Processes for Source window.
- b) Click the “+” next to the **Synthesize-XST** process to expand the hierarchy.
- c) Double-click the **Check Syntax** process.

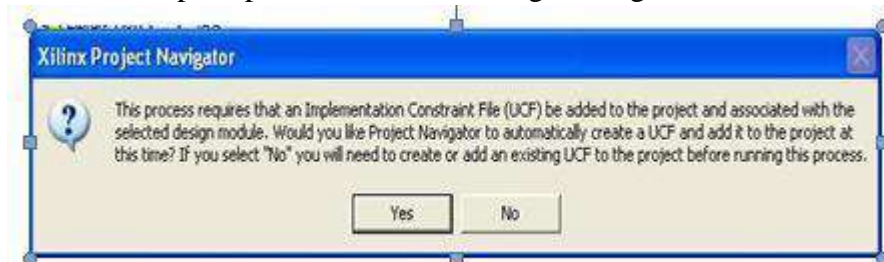
11) When an ISE process completes, you will see a status indicator next to the process name.

- a) If the process completed successfully, a **green check** mark appears.
- b) If there were errors and the process failed, a red X appears.

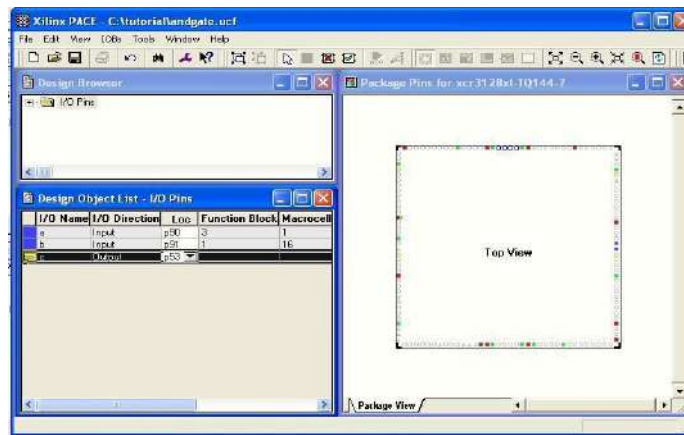
- c) A yellow exclamation point means that the process completed successfully, but some Warnings occurred.
- d) An orange question mark means the process is out of date and should be run again.
- e) Look in the Console tab of the Transcript window and read the output and status messages produced by any process that you run.

Caution! You must correct any errors found in your source files. If you continue without valid syntax, you will not be able to simulate or synthesize your design.

- 12) After the successful check syntax in the process Examine RTL diagrams.
- 13) To Create Testbench waveform, Right click on file name in source window, and\_gate.v and add source.
- 14) Add testbench waveform source with a new file name and click next.
- 15) A timing window pops up. Click on combinatorial and click next.
- 16) A graphical window of input and output appears. Make changes according to the truth table and save.
- 17) <file\_name>.tb file is added to the project.
- 18) In source window change implementation to behavioral simulation.
- 19) In process window click on Xilinx ISE simulator and RUN. Output window appears. Analyze the waveforms according to the truth table.
- 20) Double-click the Assign Package Pins process found in the User Constraints process group. ISE runs the Synthesis and Translate step and automatically creates a User Constraints File(UCF). You will be prompted with the following message.



- 21) Click Yes to add the UCF file to your project. The file is added to your project and is visible in the Sources in Project.
- 22) Now the Xilinx Pin out and Area Constraints Editor (PACE) opens.
- 23) You can see your I/O Pins listed in the Design Object List window. Enter a pin location for each pin in the Loc column as specified below  
A: P1, B:P2, C:P3
- 24) Click on the Package View tab at the bottom of the window to see the pins you just added. Put your mouse over grid number to verify the pin assignment.



25) Close PACE

Creating Configuration Data

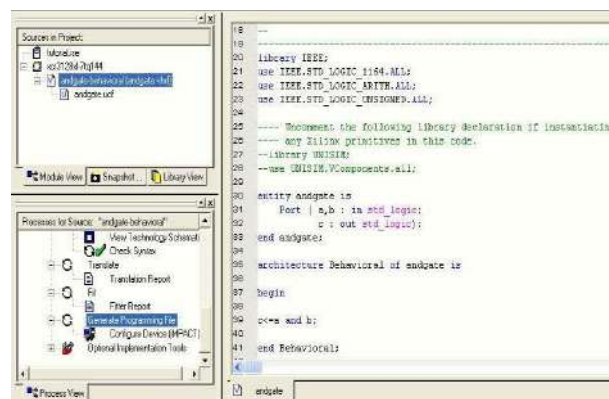
The Program File is an encoded file that is the equivalent of the design in a form that can be downloaded into the CPLD device.

The final phase in the software flow is to generate a program file and configure the device

## Generating a Program File

The Program File is created. It is written into a file called andgate.jed This is the actual configuration data

1. Double Click the Generate Programming File process located near the bottom of the Processes for Source window.

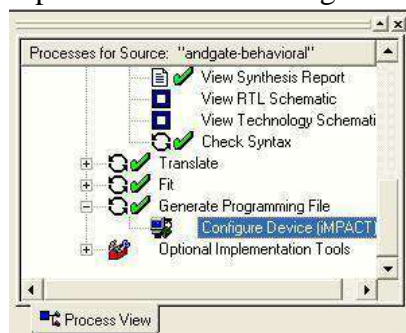


This section provides simple instructions for configuring a Spartan-3 xc3s200 device connected to your PC.

**Note:** Your board must be connected to your PC before proceeding. If the device on your board does not match the device assigned to the project, you will get errors. Please refer to the IMPACT Help for more information. To access the help, select Help > Help Topics

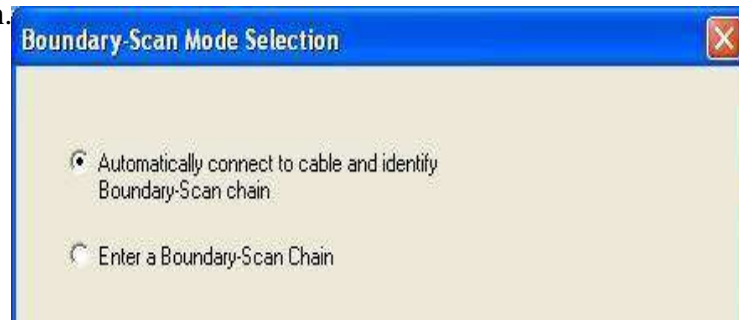
To configure the device:

1. Click the "+" sign to expand the Generate Programming File processes.





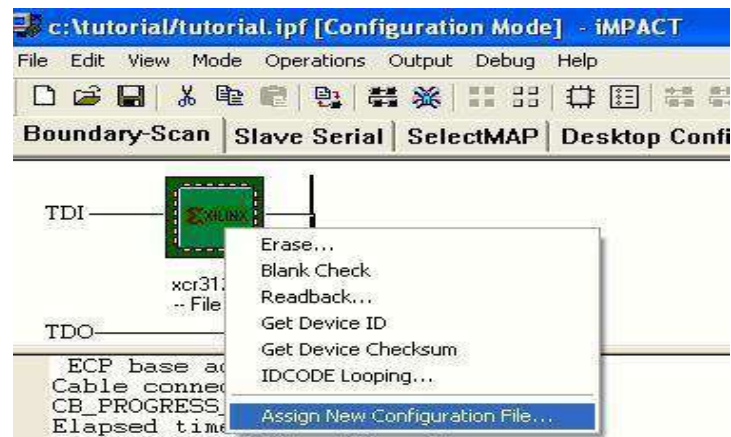
2. Double click on the Configure device IMPACT
3. In the Configure Devices dialog box, verify that Boundary-Scan Mode is selected and Click Next
4. Verify that Automatically connect to cable and identify Boundary-Scan chain is selected and click Finish.



5. If you get a message saying that there was one device found, click OK to continue



6. The iMPACT will now show the detected device, right click the device and select New Configuration File.



7. The Assign New Configuration File dialog box appears. Assign a configuration file to each device in the JTAG chain. Select the andgate.jed file and click Open
8. Right-click on the counter device image, and select Program... to open the Program Options dialog box.
9. Click OK to program the device. ISE programs the device and displays Programming Succeeded if the operation was successful
10. Close IMPACT without saving

# BASIC PROGRAM – ALL LOGIC GATES








**Aim:** Write Verilog code to realize all the logic gates

**Learning Objective:** To study the Verilog code for all the logic gates

Algorithm:

- ✓ Start
- ✓ Initialize Input & output ports. .
- ✓ Construct the truth table and extract the expression.
- ✓ Write the Verilog code using a dataflow modeling style.
- ✓ verify the functionality of design with the truth table
- ✓ observe the timing diagram and verify
- ✓ End the program.

Logic Gates and Truth Table:

Name	NOT	AND	NAND	OR	NOR	XOR	XNOR																																																																																																
Alg. Expr.	$\overline{A}$	$AB$	$\overline{AB}$	$A+B$	$\overline{A+B}$	$A\oplus B$	$\overline{A\oplus B}$																																																																																																
Symbol																																																																																																							
Truth Table	<table><tr><th>A</th><th>X</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	X	0	1	1	0	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	B	A	X	0	0	0	0	1	0	1	0	0	1	1	1	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	B	A	X	0	0	1	0	1	1	1	0	1	1	1	0	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	B	A	X	0	0	0	0	1	1	1	0	1	1	1	1	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	B	A	X	0	0	1	0	1	0	1	0	0	1	1	0	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	B	A	X	0	0	0	0	1	1	1	0	1	1	1	0	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	B	A	X	0	0	1	0	1	0	1	0	0	1	1	1
A	X																																																																																																						
0	1																																																																																																						
1	0																																																																																																						
B	A	X																																																																																																					
0	0	0																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					
B	A	X																																																																																																					
0	0	1																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
B	A	X																																																																																																					
0	0	0																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	1																																																																																																					
B	A	X																																																																																																					
0	0	1																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	0																																																																																																					
B	A	X																																																																																																					
0	0	0																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
B	A	X																																																																																																					
0	0	1																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					



**VERILOG CODE :**

## 1. AND gate

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

## 2. OR gate

```
module or_gate (a,b,c);  
    input a;  
    input b;  
    output c;  
    assign c= a|b;  
endmodule
```

## 3. NOT gate

```
module not_gate (a,c);  
    input a;  
    output c;  
    assign c= ~a;  
endmodule
```

## 4. NAND gate

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

## 5. NOR gate

```
module nor_gate (a,b,c);  
    input a;  
    input b;  
    output c;  
    assign c= ~(a|b);  
endmodule
```

## 6. XOR gate

```
module xor_gate (a,b,c);  
    input a;  
    input b;
```

```
        output c;
        assign c= a^b;
endmodule
```

#### 7. XNOR gate

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

### VERILOG CODE

```
module gates(a_in, b_in, not_op, and_op, nand_op, or_op, nor_op, xor_op, xnor_op);
    input a_in, b_in;
    output not_op, and_op, nand_op, or_op, nor_op, xor_op, xnor_op;

    assign not_op= ~a_in;
    assign and_op=a_in&b_in; assign nand_op=~(a_in&b_in);
    assign or_op=a_in|b_in;
    assign nor_op=~(a_in|b_in); assign xor_op=a_in^b_in;
    assign xnor_op=~(a_in^b_in);
endmodule
```

**Result:** The Simulation has carried out and verified with respect to truth table.

**Outcomes:** Familiar with Verilog HDL Program, usage of Xilinx software and understand ISE Simulator.

# PROGRAM - 2 ADDERS

**AIM:** Write a Verilog code to describe the functions of a Full Adder .

**Learning Objective:** To study the working and writing HDL code for Adders.

**Algorithm:**

- ✓ Start
- ✓ Initialize Input & output ports. .
- ✓ Construct the truth table and extract the expression also draw the logic circuit.
- ✓ Write the Verilog code using a dataflow, behavioral and structural modeling styles with respect to the truth table, expression and logic circuit.
- ✓ verify the functionality of design referring to truth table
- ✓ observe the timing diagram
- ✓ End the program.

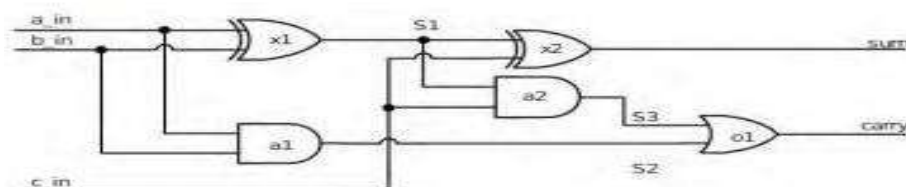
**Block Diagram:**



**Expression:**

$sum = a\_in \oplus b\_in \oplus c\_in;$   
 $carry = (a\_in.b\_in) + (b\_in.c\_in) + (a\_in.b\_in);$

**Logic Diagram:**



**Truth Table:**

Inputs			Outputs	
a_in	b_in	c_in	sum	carry
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

**CODE :**

i) Verilog - Data Flow Style :

module fulladder(a\_in, b\_in, c\_in, sum, carry):

```

input a_in, b_in, c_in;
output sum, carry;
    assign sum = a_in ^ b_in ^ c_in;
    assign carry = (a_in & b_in) | (b_in & c_in) | (a_in & c_in);
endmodule

```

## ii) Verilog - Behavioral Style:

```

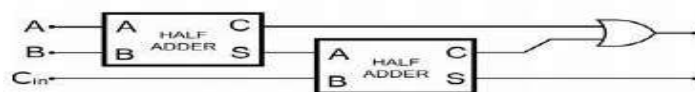
module fulladder(a,b,c, sum, carry);
    input [2:0] a,b,c;
    output sum,carry;
    reg sum,carry;
    always@(a,b,c)
    begin
        case ({a,b,c})
            3'b000: {sum,carry}=2'b00;
            3'b001: {sum,carry}=2'b10;
            3'b010: {sum,carry}=2'b10;
            3'b011: {sum,carry}=2'b01;
            3'b100: {sum,carry}=2'b10;
            3'b101: {sum,carry}=2'b01;
            3'b110: {sum,carry}=2'b01;
            3'b111: {sum,carry}=2'b11;
            default: {sum,carry}=2'bxx;
        endcase
    end
endmodule

```

**Block Diagram:**



**Logic Diagram:**



**Truth Table:**

Inputs			Outputs	
a in	b in	c in	sum	carry
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

## iii) Verilog - Structural Style

```

module full_adder (a,b,c,sum,carry);
    input a,b,c;
    output sum,carry;

```

```

wire s1,c1,c2,c3;
xor(s1,a,b);
xor(s,c,s1);
and(c1,a,b);
and(c2,s1,cin);
or(carry,c1,c2);
endmodule

```

### (Extra Stuff)

#### iv) Verilog - Structural Style (Using two half adders) :

```

module fulladder (a_in, b_in, c_in, sum, carry);
    input a_in, b_in, c_in;
    output sum, carry;
    wire temp1, temp2, temp3;
        halfadder ha1 (a_in, b_in, temp1, temp2);
        halfadder ha2 (c_in, temp1, sum, temp3);
        or g3 (carry,temp3,temp1);
endmodule

```

```

module halfadder(a, b, s, c);
    input a, b;
    output s, c;
    xor g1 (s, a, b);
    and g2 (c, a, b);
endmodule

```

**Result:** The Simulation has carried out and verified with respect to truth table.

**Outcomes:** Be able to design a model in three modeling style such as dataflow, behavioral and structural.

## PROGRAM 3 - ARITHMETIC LOGIC UNIT

**AIM:** Write a Verilog code to a model for 32 bit ALU for given schematic diagram.

**Learning Objective:** Design of ALU unit and knowing the operation of ALU.

OP-CODE	ALU OPERATION
1.	A+B

2.	A-B
3.	A Complement
4.	A*B
5.	A AND B
6.	A OR B
7.	A NAND B
8.	A XOR B

Algorithm:

- ✓ Start
- ✓ Initialize Input & output ports. .
- ✓ Write the Verilog code using a behavioral modeling style for a given opcode
- ✓ verify the functionality of design referring to truth table
- ✓ observe the timing diagram
- ✓ End the program.

VERILOG CODE:

```
module alu(a, b, opcode,en,y,y_mul);
    input [31:0] a;
    input [31:0] b;
    input en;
    input [2:0] opcode;
    output [31:0] y;
    output[63:0]y_mul;
    reg [31:0] y;
    reg [63:0] y_mul;
    always @(a, b , opcode)
    begin
        if (en==1)
            case (opcode)
                3'b000:y=a+b;
                3'b001:y=a-b;
                3'b010:y=~a;
                3'b011:y_mul=a*b;
                3'b100:y= a&b;
                3'b101:y=a|b;
                3'b110:y=~(a&b);
                3'b111:y=a^b;
            default:begin end
        endcase
    end
```

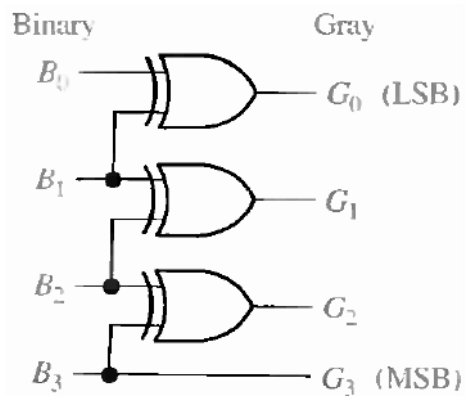
```
                else
                    begin
                        y=32'bz;
                        y_mul=64'bz
                    end
                end
            end
        endmodule
```

**Result:** The Simulation has carried out and verified with respect to truth table.

**Outcomes:** Be able to design a small digital circuit and functional verification is learned.

## 4 BIT BINARY TO GRAY

Logic Diagram :



**Truth Table :**

Binary Inputs (b_in)				Gray Outputs (g_op)			
b_in[3]	b_in[2]	b_in[1]	b_in[0]	g_op[3]	g_op[2]	g_op[1]	g_op[0]
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	0	0	1	1	1
0	1	1	1	0	1	0	1
0	1	1	0	0	1	0	0
1	0	0	1	1	1	0	0
1	0	0	0	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0



## K-MAP FOR G3:

B1B0 \ B3B2	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	1
10	1	1	1	1

Equation for G3= B3

## K-MAP FOR G2:

B1B0 \ B3B2	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	0	0	0	0
10	1	1	1	1

Equation for G2= B3<sup>''</sup> B2 + B3 B2<sup>''</sup>

$$G2 = B3 \text{ XOR } B2$$

## K-MAP FOR G1:

B1B0 \ B3B2	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	1	1	0	0
10	0	0	1	1

Equation for G1= B1<sup>''</sup> B2 + B1 B2<sup>''</sup>

$$G1 = B1 \text{ XOR } B2$$

## K-MAP FOR G0:

B1B0 \ B3B2	00	01	11	10
00	0	1	0	1
01	0	1	0	1
11	0	1	0	1
10	0	1	0	1

Equation for G0= B1<sup>''</sup> B0 + B1 B0<sup>''</sup>

$$G0 = B1 \text{ XOR } B0$$

## VERILOG CODE:

```

module binary_gray(b_in, g_op);
    input [3:0] b_in;
    output [3:0] g_op;
    assign g_op[3] = b_in[3];
    assign g_op[2] = b_in[3] ^ b_in[2];
    assign g_op[1] = b_in[2] ^ b_in[1];

```

```
        assign g_op[0] = b_in[1] ^ b_in[0];  
endmodule
```

**Result:** The Simulation has carried out and verified with respect to truth table.

**Outcomes:** Be able to model digital systems at several levels of abstractions and also able to write the Verilog HDL code for different combinational circuits by using truth table in dataflow and behavioral model.

## 8:3 Encoder [Without Priority]

VERILOG CODE :

```
module encoder8_3(en, a_in, y_op);  
    input en;  
    input [7:0] a_in;  
    output [2:0] y_op;  
    reg [2:0] y_op;  
    always @ (a_in,en)  
    begin  
        if(en==1 )  
            y_op=3'bzzz;  
        else  
            case (a_in)
```

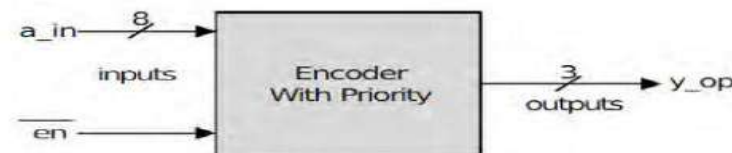
```

8'b00000001: y_op = 3'b000;
8'b00000010: y_op = 3'b001;
8'b00000100: y_op = 3'b010;
8'b00001000: y_op = 3'b011;
8'b00010000: y_op = 3'b100;
8'b00100000: y_op = 3'b101;
8'b01000000: y_op = 3'b110;
8'b10000000: y_op = 3'b111;
default: y_op = 3'bxxx;
endcase
end
endmodule

```

## ii b) 8:3 Encoder [With Priority]

**Block Diagram:**



**Truth Table:**

Inputs								Outputs		
a_in(7)	a_in(6)	a_in(5)	a_in(4)	a_in(3)	a_in(2)	a_in(1)	a_in(0)	y_op(2)	y_op(1)	y_op(0)
1	X	X	X	X	X	X	X	1	1	1
0	1	X	X	X	X	X	X	1	1	0
0	0	1	X	X	X	X	X	1	0	1
0	0	0	1	X	X	X	X	1	0	0
0	0	0	0	1	X	X	X	0	1	1
0	0	0	0	0	1	X	X	0	1	0
0	0	0	0	0	0	1	X	0	0	1
0	0	0	0	0	0	0	1	0	0	0

### VERILOG CODE:

```

module prio_enco(en, a_in, y_op);
    input en;
    input [7:0] a_in;
    output [2:0] y_op;
    reg [2:0] y_op;
    always @ (a_in,en)
    begin
        if(en==1) y_op = 3'bzzz;
        if(a_in[7] == 1) y_op = 3'b000;
        if(a_in[6] == 1) y_op = 3'b001;
        if(a_in[5] == 1) y_op = 3'b010;
        if(a_in[4] == 1) y_op = 3'b011;
        if(a_in[3] == 1) y_op = 3'b100;
        if(a_in[2] == 1) y_op = 3'b101;
        if(a_in[1] == 1) y_op = 3'b110;
        if(a_in[0] == 1) y_op = 3'b111;
        default: y_op=3'bxxx;
    end
end

```

endmodule

### iii) 8:1 MULTIPLEXER

**Block Diagram:****Truth Table:**

Inputs											Output
sel (2)	sel (1)	sel (0)	i_in (7)	i_in (6)	i_in (5)	i_in (4)	i_in (3)	i_in (2)	i_in (1)	i_in (0)	y_out
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	1

#### VERILOG CODE :

```

module mux8_1(en,i_in, sel, y_out);
    input en;
    input [7:0] a_in;
    input [2:0] sel;
    output y_out;
    reg y_out;
    always@ (i_in,sel )
    begin
        if(en==1)
            y_out=1'bz;
        else
            case (sel)
                3'b000:y_out=i_in[0];
                3'b001: y_out=i_in[1];
                3'b010: y_out=i_in[2];
                3'b011: y_out=i_in[3];
                3'b100: y_out=i_in[4];
                3'b101: y_out=i_in[5];
                3'b110: y_out=i_in[6];
                3'b111: y_out=i_in[7];
            endcase
        end
    end
endmodule

```

```
endmodule
```

```
end
```

## 2 to 4 decoder

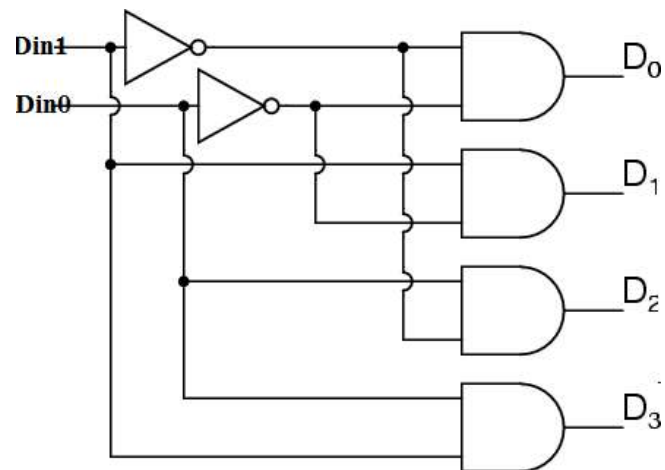
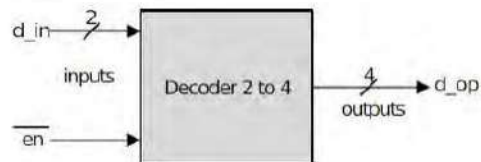


Figure: 2 to 4 Decoder

**Truth Table:**

Inputs			Outputs			
en	d_in(1)	d_in(0)	d_op(3)	d_op(2)	d_op(1)	d_op(0)
1	X	X	Z	Z	Z	Z
0	0	0	0	0	0	1
0	0	1	0	0	1	0
0	1	0	0	1	0	0
0	1	1	1	0	0	0

**VERILOG CODE : Structural code for 2 to 4 decoder**

```

module 2to4dec( input [1:0] d_in, output [3:0] d_op);
wire d0_bar, d1_bar;
not a1(d0_bar, d_in[0]);
not a2(d1_bar, d_in[1]);
and a3(d_op[0],d1_bar,d0_bar);
and a4(d_op[1],d1_bar,d_in[0]);
and a5(d_op[2],d_in[1],d0_bar);
and a6(d_op[3],d_in[1],d_in[0]);

endmodule

```

## PROGRAM 4 - FLIP FLOPS

**AIM:** Develop the Verilog code for the following Flip-Flops:

- SR FF
- D FF
- JK FF
- T FF

**Learning Objective:** To Study and write the Verilog code for mention Flip-Flops

Algorithm:

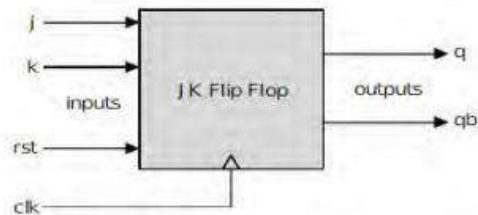
- ✓ Start
- ✓ Initialize Input & output ports. .
- ✓ Construct the truth table.

- ✓ Write the Verilog code using a behavioral modeling style with respect to the truth table
- ✓ verify the functionality of design referring to truth table
- ✓ observe the timing diagram
- ✓ End the program.

**Note:** The same Algorithm follows for all types of flip flops.

#### a. JK FLIP-FLOP

**Block Diagram:**



**Truth Table:**

Inputs				Outputs		
rst	clk	j	k	q	qb	Action
1	↑	X	X	q	qb	No Change
0	↑	0	0	q	qb	No Change
0	↑	0	1	0	1	Reset
0	↑	1	0	1	0	Set
0	↑	1	1	q'	q'	Toggle

#### VERILOG CODE :

```
module jk_ff(jk, clk, rst, q, qb);
    input [1:0]jk;
    input rst, clk;
    output q,qb;
```

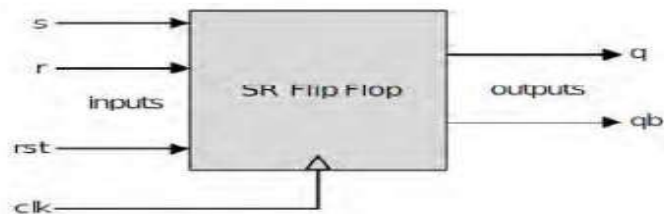
```

reg q,qb;
always @ (posedge clk)
begin
    if (rst==1)
        begin
            q=0;
            qb=1;
        end
    else
        case (jk)
            2'b00: begin
                q=q; qb=qb;
            end
            2'b01: begin
                q=0; qb=1;
            end
            2'b10: begin
                q=1; qb=0;
            end
            2'b11: begin
                q=~q; qb=~qb;
            end
            default:begin end
        endcase
    end
endmodule

```

## b. SR FLIPFLOP

**Block Diagram:**



**Truth Table:**

Inputs				Outputs		
rst	clk	s	r	q	qb	Action
1	↑	X	X	q	qb	No Change
0	↑	0	0	q	qb	No Change
0	↑	0	1	0	1	Reset
0	↑	1	0	1	0	Set
0	↑	1	1	-	-	Illegal

## VERILOG CODE :

```

module sr_ff(sr, clk, rst, q, qb);
    input [1:0]sr;
    input rst, clk;

```



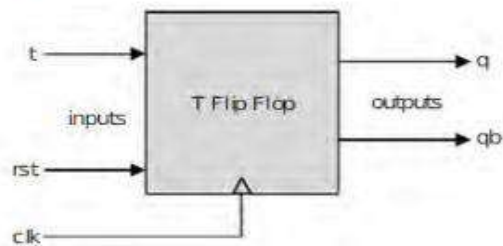
```

output q,qb;
reg q,qb;
    always @ (posedge clk)
    begin
        if (rst==1)
        begin
            q=0; qb=1;
        end
        else
        case (sr)
            2'b00: begin q=q; qb=qb; end
            2'b01: begin q=0; qb=1; end
            2'b10: begin q=1; qb=0; end
            2'b11: begin q=1'bx; qb=1'bx; end
            default:begin end
        endcase
    end
endmodule

```

### c. T- FLIPFLOP

**Block Diagram:**



**Truth Table:**

Inputs			Outputs		
rst	clk	t	q	qb	Action
1	↑	X	q	qb	No Change
0	↑	0	q	qb	No Change
0	↑	1	q'	q'	Toggle

Algorithm:

Start

Initialize T is reset and CLK as input ,a and qn as ouput

If clk="1" and an event on the positive pulse

If t=0 then q=1 else q=0 qb=1

Stop

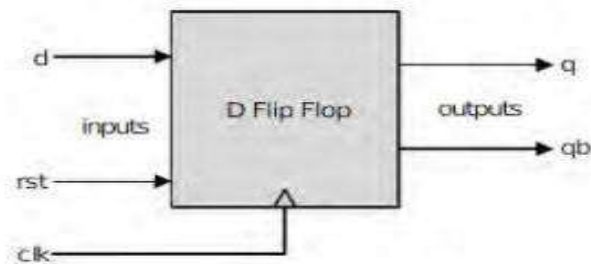
## VERILOG CODE :

```

module tff (t,clk,rst, q,qb);
  input t,clk,rst;
  output q,qb;
  reg q,qb;
  reg temp=0;
  always@(posedge clk,posedge rst)
  begin
    if (rst==0)
    begin
      case(t)
        1'b0:q=q;
        1'b1:q= ~q;
      endcase
      qb=~q;
    end
  end
endmodule

```

## D-FLIPFLOP

**Block Diagram:****Truth Table:**

Inputs			Outputs		
rst	clk	d	q	qb	Action
1	↑	X	q	qb	No Change
0	↑	0	0	1	Reset
0	↑	1	1	0	Set

## VERILOG CODE :

```

module d_ff(d, rst, clk, q, qb);
  input d;
  input rst;
  input clk;
  output q;
  output qb;
  reg q,qb;
  always@(posedge clk)

```

```
begin
    if (rst==1)
        begin
            q=0; qb=1;
        end
    else
        begin
            q=d; qb=~d;
        end
    end
end
```

endmodule

**Result:** The Simulation has carried out and verified with respect to truth table.

**Outcomes:** Be able to model a memory system with clock.

## **PROGRAM 5 - COUNTERS**

**AIM:** Design 4 bit binary, BCD Counter (Synchronous reset and Asynchronous reset) and “any sequence” Counters.

**Learning Objective:** To study and write the code for Sequential circuits.

Algorithm:

- ✓ Start
- ✓ Initialize Input & output ports. .
- ✓ Construct the truth table.
- ✓ Write the Verilog code using a behavioral modeling style with respect to the truth table
- ✓ verify the functionality of design referring to truth table
- ✓ observe the timing diagram
- ✓ End the program.

### **A. BINARY COUNTER**

```
module binary_counter (clk, rst, bin_count);
    input clk, rst;
    output [3:0] bin_count;
    reg [3:0] bin_count;
    initial
        bin_count = 4'b0000;
    always @ (posedge clk)
        begin
            if (rst)
                bin_count = 3'b0000;
            else
                bin_count = bin_count + 1'b1;
        end
endmodule
```

### **B. BCD COUNTER**

```
module BCD_Counter ( clk ,reset ,dout );
    input clk ;
    input reset ;
    output [3:0] dout ;
```

```
reg [3:0] dout ;  
    initial  
    dout = 0 ;  
    always @ (posedge clk)  
    begin  
        if (reset)  
            dout = 0;
```

```

        else
        if (dout<=9)
            dout = dout + 1;
        else
            if (dout==9) begin
                dout <= 0;
            end
        end
    end
endmodule

```

## Synchronous Counter

**Block diagram:**



**Truth table:**

Clock	rst	bin_out(3)	bin_out(2)	bin_out(1)	bin_out(0)
X	X	0	0	0	0
↑	1	0	0	0	0
↑	1	0	0	0	1
↑	1	0	0	1	0
↑	1	0	0	1	1
↑	1	0	1	0	0
↑	1	0	1	0	1
↑	1	0	1	1	0
↑	1	0	1	1	1
↑	1	1	0	0	0
↑	1	1	0	0	1
↑	1	1	0	1	0
↑	1	1	0	1	1
↑	1	1	1	0	0
↑	1	1	1	0	1
↑	1	1	1	1	0
↑	1	1	1	1	1

### C. SYNCHRONOUS With Reset - UP COUNTER

```

module syn_up_counter (clk ,rst ,enable ,up_count);
    input clk ;
    input rst ;
    input enable ;

```

```
output [3:0] up_count;
wire clk ;
wire rst ;
wire enable ;
reg [3:0] up_count;
always @ (posedge clock)
begin
    if (reset == 1'b1)
    begin
        up_count <= 4'b0000;
    end
    else
    if (enable == 1'b1)
    begin
        up_count <= up_count+ 1;
    end
end
endmodule
```

#### D. SYNCHRONOUS With Reset– DOWNCOUNTER

```
module syn_dwn_counter (clk ,rst ,enable ,dwn_count);
input clk ;
input rst ;
input enable ;
output [3:0] dwn_count;
wire clk ;
wire rst ;
wire enable ;
reg [3:0] dwn_count;
always @ (posedge clock)
begin
    if (reset == 1'b1)
    begin
        dwn_count <= 4'b0000;
    end
    else
    if (enable == 1'b1)
    begin
        dwn_count <= dwn_count- 1;
    end
end
endmodule
```

#### E. SYNCHRONOUS With Clear – UP-DOWN COUNTER

```
module sync_up_dwn_counter(cnt,clk,up_dwn,clr);
input clk,clr;
input up_dwn;
output [3:0] cnt;
reg [3:0]cnt;
```

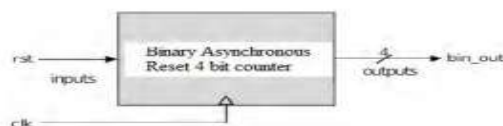
```

initial cnt = 1'd0;
always @(posedge clk)
begin
    case(clr)
        1'b1 : cnt = 1'd0;
        default : begin
            case(up_dwn)
                1'b0 : cnt = cnt - 4'b0001;
                default : cnt = cnt + 1'b1;
            endcase
        end
    endcase
end
endmodule

```

## ASynchronous Counter

**Block diagram:**



**Truth table:**

Clock	rst	bin_out(3)	bin_out(2)	bin_out(1)	bin_out(0)
X	0	0	0	0	0
↑	1	0	0	0	0
↑	1	0	0	0	1
↑	1	0	0	1	0
↑	1	0	0	1	1
↑	1	0	1	0	0
↑	1	0	1	0	1
↑	1	0	1	1	0
↑	1	0	1	1	1
↑	1	1	0	0	0
↑	1	1	0	0	1
↑	1	1	0	1	0
↑	1	1	0	1	1
↑	1	1	1	0	0
↑	1	1	1	0	1
↑	1	1	1	1	0
↑	1	1	1	1	1

### F. ASYNCHRONOUS With Reset - UP COUNTER

```

module counter (clk, clr, enable, asy_up);
input clk, clr, enable;
output [3:0] asy_up;
reg [3:0] tmp;

always @(posedge clk or posedge clr)
begin
    if (clr)
        tmp = 4'b0000;

```



```
        else
            if (enable)
                tmp = tmp + 1'b1;
            end
        assign asy_up = tmp;
    endmodule
```

### G. ASYNCHRONOUS With Reset – DOWNCOUNTER

```
module counter (clk, clr, enable, asy_dwn);
    input clk, clr, enable;
    output [3:0] asy_dwn;
    reg [3:0] tmp;

    always @(posedge clk or posedge clr)
    begin
        if (clr)
            tmp = 4'b0000;
        else
            if (enable)
                tmp = tmp - 1'b1;
            end
        assign asy_dwn = tmp;
    endmodule
```

### H. ASYNCHRONOUS With Clear – UP-DOWN COUNTER

```
module async_up_dwn_counter (clk,cnt,up_dwn,clr);
    input up_dwn,clr,clk;
    output [3:0]cnt;
    reg [3:0]cnt;
    always @(posedge clk)
    begin
        if(clr == 1'b1)
            cnt = 4'b0000;
        else
            begin
                case(up_dwn)
                    1'b0 : begin
                        if(cnt[1:0] == 2'b10)
                            cnt[1:0] = 2'b01;
                        else if(cnt[2:0] == 3'b100)
                            cnt[2:0] = 3'b011;
                        else if(cnt[3:0] == 4'b1000)
                            cnt[3:0] = 4'b0111;
                        else if(cnt[3:0] == 4'b0000)
                            cnt[3:0] = 4'b1111;
                        else
                            cnt[0] = ~cnt[0];
                    end
                endcase
            end
    end
```

```

                                end
1'b1 : begin
    if(cnt[1:0] == 2'b01)
        cnt[1:0] = 2'b10;
    else if(cnt[2:0] == 3'b011)
        cnt[2:0] = 3'b100;
    else if(cnt[3:0] == 4'b0111)
        cnt[3:0] = 4'b1000;
    else if(cnt[3:0] == 4'b1111)
        cnt[3:0] = 4'b0000;
    else
        cnt[0] = ~cnt[0];
    end
endcase
end
end
endmodule
```



```
counter[6]= 8'd251;
counter[7]= 8'd255;
counter[8]= 8'd255;
counter[9]= 8'd251;
counter[10]= 8'd244;
counter[11]= 8'd232;
counter[12]= 8'd218;
counter[13]= 8'd192;
counter[14]= 8'd182;
counter[15]= 8'd161;
counter[16]= 8'd139;
counter[17]= 8'd116;
counter[18]= 8'd94;
counter[19]= 8'd73;
counter[20]= 8'd54;
counter[21]= 8'd37;
counter[22]= 8'd23;
counter[23]= 8'd11;
counter[24]= 8'd4;
counter[25]= 8'd4;
counter[26]= 8'd11;
counter[27]= 8'd23;
counter[28]= 8'd37;
counter[29]= 8'd54;
counter[30]= 8'd73;
counter[31]= 8'd94;
counter[32]= 8'd116;
counter[33]= 8'd128;
end

always @(posedge clk)
begin
    if (clk == 1'b1)
        begin
            div <= div + 1'b 1;
        end
    end
end

assign clkdiv = div[8];

always @(posedge(clkdiv))
begin
    if(i>34)
        begin
            i=0;
        end

    dac_out <= counter[i];
    i = i + 1;

end

endmodule
```

**//Extra Stuff****Square**

```
module sqwave (clk,rst,dac_out);

input clk;
input rst;
output reg [7:0] dac_out;

//reg [7:0] dac_out;
reg [7:0] counter;
reg [15:0] div;
wire clkdiv;

always @(posedge clk)
begin
if (clk == 1'b1)
begin
div <= div + 1'b 1;
end
end

assign clkdiv = div[8];

always @(posedge(clkdiv))
begin
if (rst == 1'b1)
begin
counter <= 8'b 00000000;
end

counter <= counter + 1;

end

always @(counter)
begin
if (counter <= 128)
begin
dac_out <= 8'b 11111111;
end
else
begin
dac_out <= 8'b 00000000;
end
end

endmodule
```

## Triangle

```
module tri_wave ( clk, rst, dac_out);

input clk;
input rst;
output [7:0] dac_out;

reg [7:0] dac_out;
reg [7:0] counter;
reg [15:0] div;
wire clkdiv;

always @(posedge clk)
begin : process_1
if (clk == 1'b 1)
begin
div <= div + 1'b 1;
end
end

assign clkdiv = div[8];

always @(posedge(clkdiv))
begin : process_2
if (rst == 1'b 1)
counter <= 8'b 00000000;
end
counter <= counter + 1;

if(counter < 128)
dac_out = dac_out + 1;
else
dac_out = dac_out - 1;
end

endmodule
```

## Ramp

```
Module ramp_wave ( clk, rst, dac_out);

input clk;
input rst;
output [7:0] dac_out;

reg [7:0] dac_out;
reg [7:0] counter;
reg [15:0] div;
wire clkdiv;

always @(posedge clk)
begin : process_1
if (clk == 1'b 1)
begin
```

```
        div <= div + 1'b 1;
    end
end

assign clkdiv = div[8];

always @(posedge(clkdiv))
    begin : process_2
        if (rst == 1'b 1)
            begin
                counter <= 8'b 00000000;
            end
        counter <= counter + 1;
        dac_out = dac_out - 1;
    end

endmodule
```

**User Constraint File (UCF):**

```
NET "clk"      LOC = "p79"
NET "rest"     LOC = "p21"
NET "dout<0>"  LOC = "p187"
NET "dout<1>"  LOC = "p185"
NET "dout<2>"  LOC = "p190"
NET "dout<3>"  LOC = "p189"
NET "dout<4>"  LOC = "p194"
NET "dout<5>"  LOC = "p191"
NET "dout<6>"  LOC = "p197"
NET "dout<7>"  LOC = "p196"
```

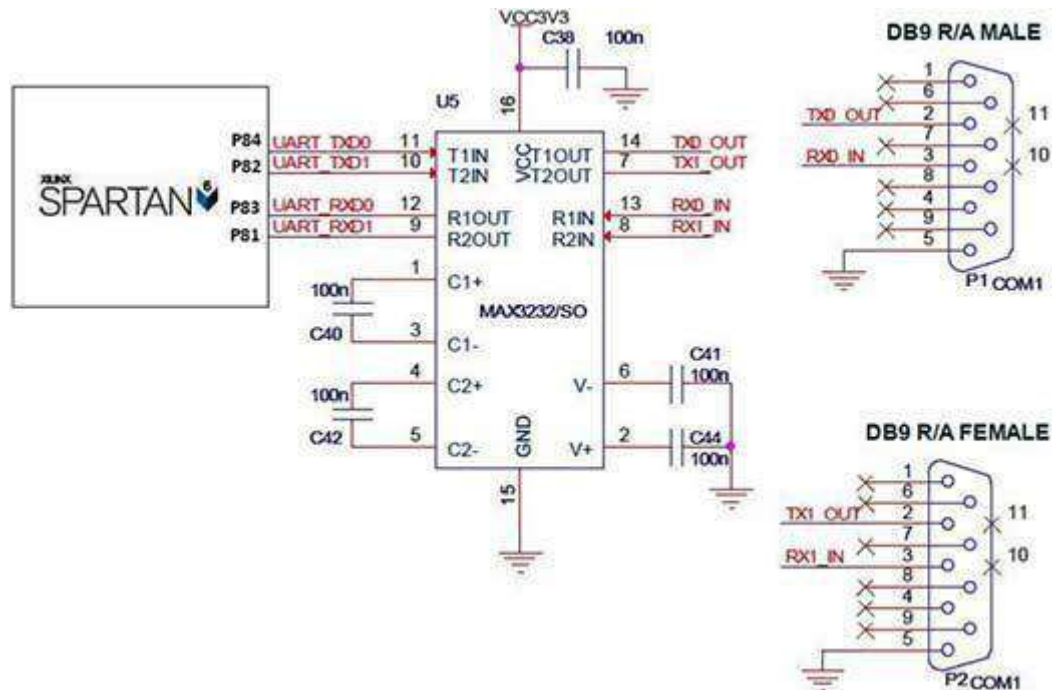
**Result:** The different waveforms are generated and observed in CRO.

**Outcomes:** Design, and interface a DAC using HDL be able to generate different waveforms using DAC on CRO

## **PROGRAM 5 – ANALOG TO DIGITAL CONVERTER**

**AIM:** Write HDL code to accept Analog signal, Temperature sensor and display the data on LCD or Seven segment displays..

**Learning Objective:** To study HDL code to simulate Analog to Digital Converter (ADC) using temperature sensor.



```

library IEEE --analog to digital converter
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

```

entity adc is

```

    Port (
        addr : out std_logic_vector(1 downto 0);
        chin : in std_logic_vector(1 downto 0);
        strt : out std_logic;
        EOC : in std_logic;
        dout : in std_logic_vector(7 downto 0);
        clk1 : in std_logic;
        oen1 : out std_logic;
        oen2 : out std_logic;
        oen3 : out std_logic;
        oen4 : out std_logic;
        oen5 : out std_logic;
        oen6 : out std_logic;
        disp : out std_logic_vector(7 downto 0));
end adc;

```

architecture Behavioral of adc is

```

    type state is (state1,state2,state3,state4,state5);

```



```
--locally used signals declaration
signal current_state, next_state:state;
signal soen1 :          std_logic:= '1';
signal soen2 :          std_logic:= '1';
signal clk_count :      std_logic_vector(8 downto 0):= (others => '0');
signal clk_dsp :        std_logic_vector(1 downto 0);
signal check :          std_logic_vector(7 downto 0);
signal address :        std_logic_vector(3 downto 0):= "0000";
signal temp2 :          std_logic_vector(7 downto 0):= "00000000";
signal data :           std_logic_vector(7 downto 0);
signal clk              :          std_logic;

begin
addr <= "00" when chin = "00" else
      "01" when chin = "01" else
      "10" when chin = "10" else
      "11";

oen3 <= '0';
oen4 <= '0';
oen5 <= '0';
oen6 <= '0';

p1:process(clk1)
begin
    if clk1'event and clk1 = '1' then
        clk_count <= clk_count + 1;
        clk_dsp <= clk_count(8 downto 7);
        clk <= clk_count(6);
    end if;
end process p1;

p2:process(clk_dsp)
begin
    case clk_dsp is
        when "00" => soen2 <= '1';
                                soen1 <= '0';
        when "01" => soen1 <= '1';
                                soen2 <= '0';
        when others => soen1 <= '0';
                                soen2 <= '0';
    end case;
end process p2;

pp:process(clk)
begin

    if clk'event and clk = '1' then
        current_state <= next_state;
```

```

        case current_state is
            when state1 =>
                strt <= '1';           --enabling start/ale

                next_state <= state2;

            when state2 =>
--start/ale low (pulse width 5 usec)
                strt <= '0';
--checking eoc
                if EOC = '0' then

                    next_state <= state3;

                end if;
            when state3 =>
--start/ale low (pulse width 5 usec)
-- making oe low to read eoc
--checking eoc
                if dout(7) = '1' then
                    if EOC = '1' then

                        next_state <= state4;

                    end if;

                when state4 =>
                    next_state <= state5;
                    when state5 => -- jump to start
                        check <= dout;
                        next_state <= state1;
                end case;
            end if;

        end process pp;

p4:process(clk1)
type t_mem is array(0 to 15) of std_logic_vector(7 downto 0);
variable mem_data: t_mem:=
    ("00111111", "00000110", "01011011", "01001111",           --0123
     "01100110", "01101101", "01111101", "00000111",           --4567
     "01111111", "01101111", "01110111", "01111100",           --89ab
     "00111001", "01011110", "01111001", "01110001");          --cdef
variable adv : integer := 0;
begin
    if clk1'event and clk1 = '1' then
        adv := conv_integer(address(3 downto 0));
        data <= mem_data(adv);
    end if ;
end process p4;

```

```
oen1 <= soen1;
oen2 <= soen2;

p5: process(clk)
begin
    if clk1'event and clk1 = '1' then
        if ( soen2 = '1' ) then
            address <= check(3 downto 0);
            disp <= data;
        else
            if (soen1 = '1' ) then
                address <= check(7 downto 4);
                disp <= data;
            end if;
        end if;
    end if;
end process p5 ;

end behavioral;
```

**User Constraint File (UCF):**

```
NET "addr<0>" LOC = "p171" | IOSTANDARD = LVTTTL ;
NET "addr<1>" LOC = "p172" | IOSTANDARD = LVTTTL ;
```

```
NET "chin<0>" LOC = "p29" | IOSTANDARD = LVTTTL ;
NET "chin<1>" LOC = "p27" | IOSTANDARD = LVTTTL ;
```

```
NET "clk1" LOC = "p79" | IOSTANDARD = LVTTTL ;
```

```
NET "disp<0>" LOC = "p10" | IOSTANDARD = LVTTTL ;
NET "disp<1>" LOC = "p11" | IOSTANDARD = LVTTTL ;
NET "disp<2>" LOC = "p12" | IOSTANDARD = LVTTTL ;
NET "disp<3>" LOC = "p13" | IOSTANDARD = LVTTTL ;
NET "disp<4>" LOC = "p15" | IOSTANDARD = LVTTTL ;
NET "disp<5>" LOC = "p16" | IOSTANDARD = LVTTTL ;
NET "disp<6>" LOC = "p18" | IOSTANDARD = LVTTTL ;
NET "disp<7>" LOC = "p19" | IOSTANDARD = LVTTTL ;
```

```
NET "dout<0>" LOC = "p196" | IOSTANDARD = LVTTTL ;
NET "dout<1>" LOC = "p197" | IOSTANDARD = LVTTTL ;
NET "dout<2>" LOC = "p191" | IOSTANDARD = LVTTTL ;
NET "dout<3>" LOC = "p194" | IOSTANDARD = LVTTTL ;
NET "dout<4>" LOC = "p189" | IOSTANDARD = LVTTTL ;
NET "dout<5>" LOC = "p190" | IOSTANDARD = LVTTTL ;
NET "dout<6>" LOC = "p185" | IOSTANDARD = LVTTTL ;
NET "dout<7>" LOC = "p187" | IOSTANDARD = LVTTTL ;
```

```
NET "EOC" LOC = "p183" | IOSTANDARD = LVTTTL ;
```



```
wire [3:0] dout;
reg [20:0] div;
wire clkdiv;
reg [3:0] shift_reg;

always @(posedge clk)
begin : process_1
    if (clk === 1'b 1)
        begin
            div <= div + 1'b 1;
        end
    end
assign clkdiv = div[16];
always @(negedge reset or posedge clkdiv)
begin : process_2
    if (reset === 1'b 0)
        begin
            shift_reg <= 4'b 0001;
        end
    else if (clkdiv === 1'b 1 )
        begin
```

```
        if (dir === 1'b 1)
        begin
            shift_reg <= {shift_reg[0], shift_reg[3:1]};
        end
        else
        begin
            shift_reg <= {shift_reg[2:0], shift_reg[3]};
        end
    end
end
```

```
assign dout =  
shift_reg;  
endmodule
```

**User Constraint File (UCF):**

```
NET "clk"      LOC = "p79" | IOSTANDARD =  
LVTTTL ; NET "rest"      LOC = "p21" |  
IOSTANDARD = LVTTTL ; NET "dir1"  LOC = "p29"  
| IOSTANDARD = LVTTTL ; NET "dout<0>" LOC =  
"p169" | IOSTANDARD = LVTTTL ; NET "dout<1>"  
LOC = "p175" | IOSTANDARD = LVTTTL ; NET  
"dout<2>" LOC = "p176" | IOSTANDARD = LVTTTL  
; NET "dout<3>" LOC = "p178" | IOSTANDARD =  
LVTTTL ;
```

**Result:** The Motor is successfully interfaced and run.

**Outcomes:** Design, and interface a stepper motor using HDL Also control and reversedirections of the motor using HDL.