EAST POINT COLLEGE OF ENGINEERING & TECHNOLOGY

Department of Computer Science and Engineering

'Jnana Prabha', Virgo Nagar Post, Bengaluru-560049

Academic Year: 2023-24

LABORATORY MANUAL

SEMESTER	**: III**
SUBJECT	**: Digital Design and Computer Organization**
SUBCODE	**: BCS 302**

NAME: ____

USN: ____

SECTION: ____

BATCH: ____

PROGRAM OUTCOMES

Engineering Graduates will able to:

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

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

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.

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.

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

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.

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.

Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
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.

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.

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.

EAST POINT COLLEGE OF ENGINEERING & TECHNOLOGY

INSTITUTE VISION AND MISSION

VISION

The East Point College of Engineering and Technology aspires to be a globally acclaimed institution, recognized for excellence in engineering education, applied research and nurturing students for holistic development.

MISSION

M1: To create engineering graduates through quality education and to nurture innovation, creativity and excellence in teaching, learning and research

M2: To serve the technical, scientific, economic and societal developmental needs of our communities

M3: To induce integrity, teamwork, critical thinking, personality development and ethics in students and to lay the foundation for lifelong learning

EAST POINT COLLEGE OF ENGINEERING & TECHNOLOGY

Department of Computer Science and Engineering

DEPARTMENT VISION AND MISSION

VISION

The department aspires to be a Centre of excellence in Computer Science & Engineering to develop competent professionals through holistic development.

MISSION

M1: To create successful Computer Science Engineering graduates through effective pedagogies, the latest tools and technologies, and excellence in teaching and learning.

M2: To augment experiential learning skills to serve technical, scientific, economic, and social developmental needs.

M3: To instill integrity, critical thinking, personality development, and ethics in students for a successful career in Industries, Research, and Entrepreneurship.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO 1: To produce graduates who can perform technical roles to contribute effectively in software industries and R&D Centre

PEO 2: To produce graduates having the ability to adapt and contribute in key domains of computer science and engineering to develop competent solutions.

PEO 3: To produce graduates who can provide socially and ethically responsible solutions while adapting to new trends in the domain to carve a successful career in the industry

PROGRAM SPECIFIC OUTCOMES (PSOs)

PSO1: To conceptualize, model, design, simulate, analyze, develop, test, and validate computing systems and solve technical problems arising in the field of computer science & engineering.

PSO2: To specialize in the sub-areas of computer science & engineering systems such as cloud computing, Robotic Process Automation, cyber security, big data analytics, user interface design, and IOT to meet industry requirements.

PSO3: To build innovative solutions to meet the demands of the industry using appropriate tools and techniques.

COURSE LEARNING OBJECTIVES

CLO 1: Demonstrate the functionalities of binary logic system

CLO 2: Explain the working of combinational and sequential logic system

CLO 3: Realize the basic structure of computer system

CLO 4: Explain the approaches involved in achieving communication between processor and I/O devices.

CLO 5: Illustrate the working of I/O operations and processing unit

COURSE OUTCOMES

At the end of the course the student will be able to:

CO1: Apply the K–Map techniques to simplify various Boolean expressions.

CO 2: Design different types of combinational and sequential circuits along with Verilog programs.

CO 3: Describe the fundamentals of machine instructions, addressing modes and Processor performance.

CO 4: Explain the approaches involved in achieving communication between processor and I/O devices.

CO 5: Analyze internal Organization of Memory and Impact of cache/Pipelining on Processor Performance.

Sl. NO	**Experiments** **Simulation packages preferred: Multisim, Modelsim, PSpice or any other relevant**
1	Given a 4-variable logic expression, simplify it using appropriate technique and simulate the same using basic gates.
2	Design a 4 bit full adder and subtractor and simulate the same using basic gates.
3	Design Verilog HDL to implement simple circuits using structural, Data flow and Behavioural model.
4	Design Verilog HDL to implement Binary Adder-Subtractor – Half and Full Adder, Half and Full Subtractor.
5	Design Verilog HDL to implement Decimal adder.
6	Design Verilog program to implement Different types of multiplexer like 2:1, 4:1 and 8:1.
7	Design Verilog program to implement types of De-Multiplexer.
8	Design Verilog program for implementing various types of Flip-Flops such as SR, JK and D.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

CIE for the theory component of the IPCC (maximum marks 50)

- IPCC means practical portion integrated with the theory of the course.
- CIE marks for the theory component are **25 marks** and that for the practical component is **25 marks**.
  25 marks for the theory component are split into **15 marks** for two Internal Assessment Tests (Two Tests, each of 15 Marks with 01-hour duration, are to be conducted) and **10 marks** for other assessment methods mentioned in 22OB4.2. The first test at the end of 40-50% coverage of thesyllabus and the second test after covering 85-90% of the syllabus.
- Scaled-down marks of the sum of two tests and other assessment methods will be CIE marks for thetheory component of IPCC (that is for **25 marks)**.
- The student has to secure 40% of 25 marks to qualify in the CIE of the theory component of IPCC.

CIE for the practical component of the IPCC

- **15 marks** for the conduction of the experiment and preparation of laboratory record, and **10 marks** for the test to be conducted after the completion of all the laboratory sessions.
- On completion of every experiment/program in the laboratory, the students shall be evaluated including viva-voce and marks shall be awarded on the same day.
- The CIE marks awarded in the case of the Practical component shall be based on the continuous evaluation of the laboratory report. Each experiment report can be evaluated for 10 marks. Marks of all experiments'

- write-ups are added and scaled down to **15 marks**.
- The laboratory test **(duration 02/03 hours)** after completion of all the experiments shall be conducted for 50 marks and scaled down to **10 marks.**
- Scaled-down marks of write-up evaluations and tests added will be CIE marks for the laboratory component of IPCC for **25 marks**.
- The student has to secure 40% of 25 marks to qualify in the CIE of the practical component of the IPCC.

SEE for IPCC

Theory SEE will be conducted by University as per the scheduled timetable, with common questionpapers for the course (**duration 03 hours**)

1. The question paper will have ten questions. Each question is set for 20 marks.
2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), **should have a mix of topics** under that module.
3. The students have to answer 5 full questions, selecting one full question from each module.
4. Marks scored by the student shall be proportionally scaled down to 50 Marks

The theory portion of the IPCC shall be for both CIE and SEE, whereas the practical portion will have a CIE component only. Questions mentioned in the SEE paper may include questions from the practical component.

Index

I VERILOG INTRODUCTION

Verilog is a hardware description language (HDL) that describes the functionality of hardware design and the synthesis tool converts hardware descriptions into an actual design that has combinational and sequential elements. Verilog language is simpler than VHDL Verilog is based on C language, whereas VHDL is based on Ada and Pascal languages. Latest Verilog standard: IEEE Standard 1364-2005

Verilog is basically a structural and behavior language and defines four abstraction levels to implement modules. With respect to the external environment, the module is viewed as identical irrespective of abstraction levels. But internal module implementation differs based on abstraction as described below.

1. **Gate level** – The module implementation is similar to the gate-level design description in terms of logic gates and interconnections between them.
2. **Dataflow level –** The module implementation depends on data flow specification i.e. how data flows and processes in the design circuit.
3. **Switch level** – The module implementation requires switch level knowledge to implement a design in terms of storage nodes, switches. This is the lowest level of abstraction.

4. Behavior level – The module implementation is similar to C language programming that includes algorithmic level implementation without worrying about hardware implementation details.Above abstraction levels are also commonly mentioned with modeling terminology. The design can be implemented with a combination of gate-level, data flow, and behavioral modeling. The commonly heard term RTL (Register Transfer Level) in digital design is used for a combination of data flow and behavior modeling.

II HOW TO USE A ISE DESIGN SUITE 14.7

Running Verilog code in Xilinx ISE Design Suite 14.7 involves several steps from creating your project to simulating your design.

1. Install and Launch Xilinx ISE Design Suite 14.7:

Download and Install Xilinx ISE Design Suite 14.7 from the Xilinx Website. Once the installation is complete, launch Xilinx ISE Design Suite.

2. Create a New Project:

Click on “File”> “New Project”

The New project wizard will open. Follow the prompts and provide a name and location for your project. Make sure to specify your project type (HDL or Schematic).

Select “VHDL” or “Verilog” as the source type,depending on your code.

3. **Add Design Files:**

   In the New Project Wizard,you will be prompted to add design files. Click "Next and Select your Verilog source code files.
   You can also add any constraint files(XDC files)if your design requires them.

4. **Set the Top Module:**

   Select the top module of your design.This is the main module you want to synthesize.If your design has multiple modules,select the top-level module that you want to program.

5. **Specify Project Settings:**

   Configure any additional settings in the New project Wizard,such as the target device, simulator and othe project properties.

6. **Create a Test Bench:**

   In your Project hierarchy,right-click on your Verilog module and choose "New Source". Select "HDL" and choose "VHDL Testbench" or Verilog TestBench depending on your preference.Follow the wizard to create testbench file.

7. **Edit Testbench Code:**

   Open the testbench file and write the Verilog or VHDL code to create stimulus for your design and check the outputs. The testbench code should instantiate your Verilog module and apply inputs,as well as monitor the outputs.

8. **Simulate the Design:**

   In Xilinx ISE,you can use the built-in simulator,ISim,to rum simulations.Click on "Simulate" in the hierarchy panel on the left.Right-click "Run Behavioral Simulation" or "Run Post-Implementation Simulation",depending on your stage of development.

9. **View Simulation Results:**

   The simulation waveform viewer will open,showing the results of your testbench simulation. Analyze the waveforms to verify your design functions correctly.

Experiment-1

1. Given a 4-variable logic expression, F(A,B,C,D)= ∑ m (0,1,2,4,5,6,8,9,10,12,13) simplify it using appropriate technique and simulate the same using basic gates.

AIM: To simplify given four variable logical expression using K-map and simulate using basic gates.

SIMULATOR USED: ISE DESIGN SUIT 14.7

THEORY:

To minimize a Boolean expression we can employ any one of the followingtechniques:

(i) Boolean Algebra

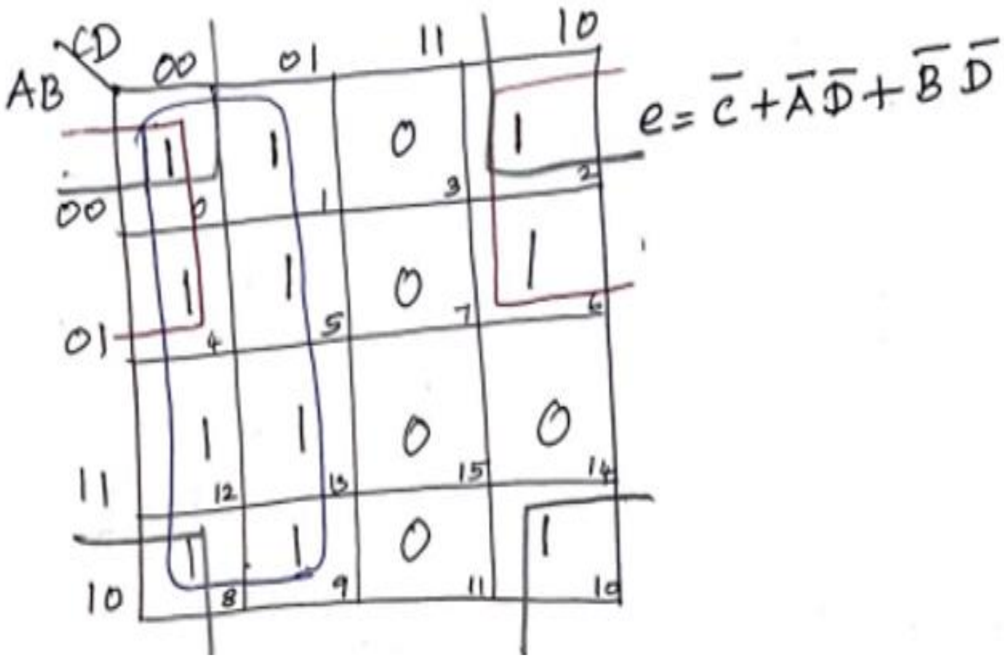
(ii) Karnaugh maps.

Before we proceed to simplification techniques, two forms of the Boolean expression must be noted.

1. Sum of product (SOP): Ex: ABC+AB+AC

2. Product of Sum (POS): Ex: (A+B+C) (A+B) +(A+C)

4-Variable Boolean expression Simplification using k-Map. $F(A,B,C,D) = \Sigma m(0,1,2,4,5,6,8,9,10,12,13)$



Sol: **e = c' +a' d'+b'd'**

Truth Table

Decimal Number	a	b	c	d	Output e
0	0	0	0	0	1
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	0
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	0
15	1	1	1	1	0

PROCEDURE:

1. Simplify the given expression using techniques like Boolean algebra and Karnaugh-map to minimize number of gates required.
2. Create Verilog code for implementation and view the logic gates and their interconnections in your design using RTL Schematic .

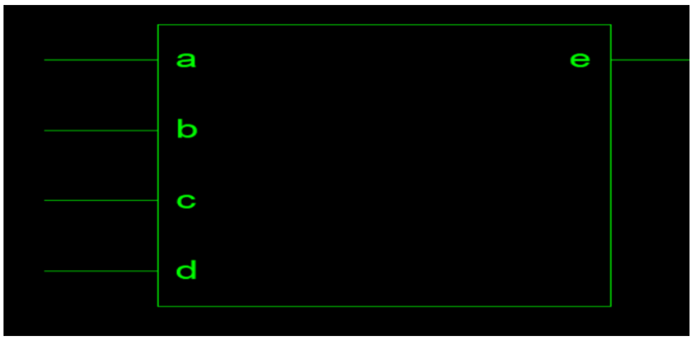


Fig 1.1 Simple circuit

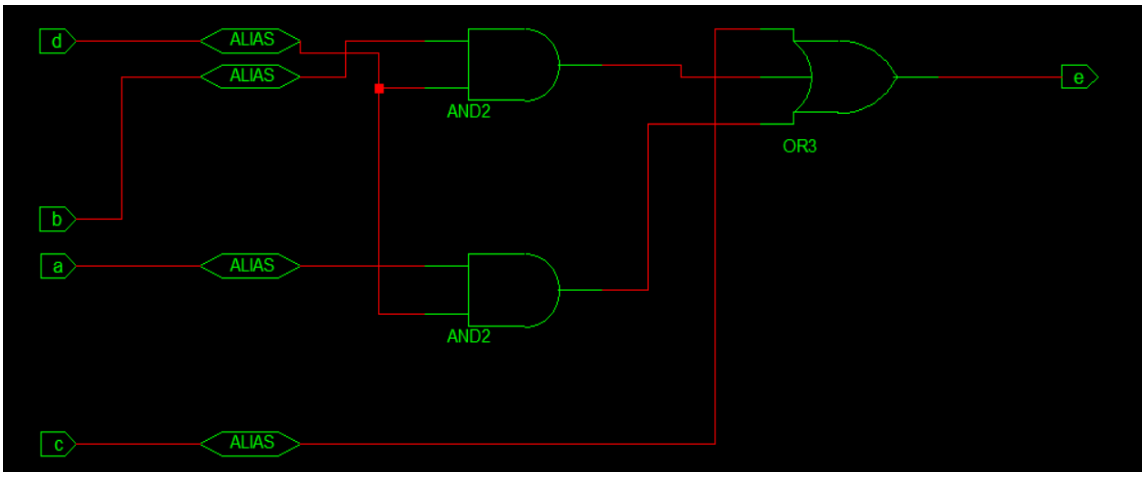


Fig 1.2

RTL Schematic For **e = c’ +a’ d’+b’d’**

Imple mentation of 4- Variable Boolean expression using gates.

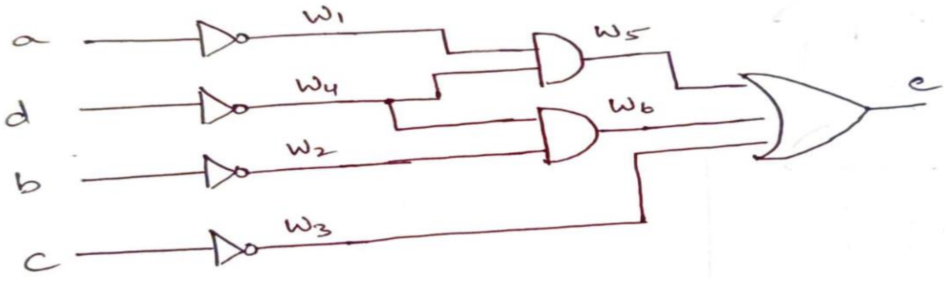


Fig 1.3:Implementation of four Variable Boolean expression

Verilog Code for Four Variable expression

```
 module b4v(a,b,c,d,e);
input a,b,c,d;
output e;
wire w1,w2,w3,w4,w5,w6;
not n1(w1,a);
not n2(w2,b);
not n3(w3,c);
not n4(w4,d);
and a1(w5,w4,w1);
and a2(w6,w4,w2);
or or1(e,w5,w6,w3);
endmodule
```

This Verilog code models the logic expression using basic gates and assigns the result to output signal ‘e’.

3. Create a test bench code to verify the functionality of Verilog design and obtain the corresponding simulation waveform. The test bench code includes input and monitors the output to compare it with expected results.

Test Bench Code:

```
 module b4v_tb;
        // Inputs
        reg a;
        reg b;
        reg c;
        reg d;
        // Outputs
        wire e;
        // Instantiate the Unit Under Test (UUT)
        b4v uut (
                .a(a),
                .b(b),
                .c(c),
                .d(d),
                .e(e)
        );
        initial begin
                 $monitor("a=%bb=%bc=%bd=%be=%b",a,b,c,d,e);
                #10 a=0;b=0;c=0;d=0;
                   #10 a=0;b=0;c=0;d=1;
                #10 a=0;b=0;c=1;d=0;
                #10 a=0;b=0;c=1;d=1;
                #10 a=0;b=1;c=0;d=0;
                #10 a=0;b=1;c=0;d=1;
                #10 a=0;b=1;c=1;d=0;
                #10 a=0;b=1;c=1;d=1;
                #10 a=1;b=0;c=0;d=0;
                #10 a=1;b=0;c=0;d=1;
                #10 a=1;b=0;c=1;d=0;
                #10 a=1;b=0;c=1;d=1;
                #10 a=1;b=1;c=0;d=0;
                #10 a=1;b=1;c=0;d=1;
                #10 a=1;b=1;c=1;d=0;
        #10 a=1;b=1;c=1;d=1;
        end
endmodule
```

Simulation Output

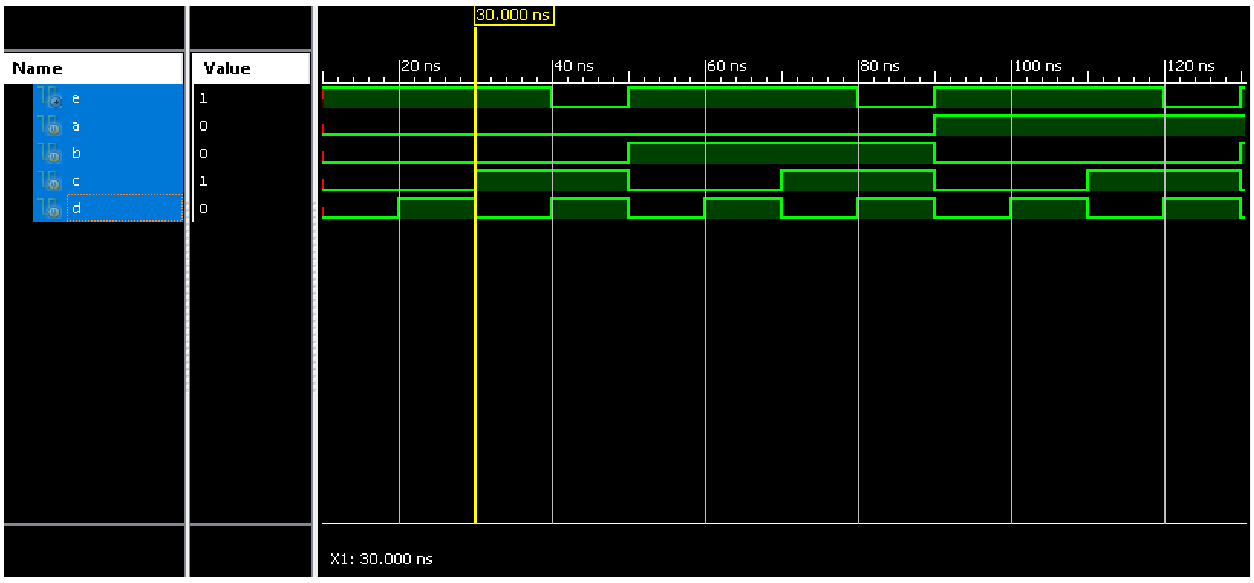


Fig 1.3 Simulation Output Waveform

(iii) Get the output from the console output window at the time of simulation

Console Window Output

```
a=0b=0c=0d=0e=1
a=0b=0c=0d=1e=1
a=0b=0c=1d=0e=1
a=0b=0c=1d=1e=0
a=0b=1c=0d=0e=1
a=0b=1c=0d=1e=1
a=0b=1c=1d=0e=1
a=0b=1c=1d=1e=0
a=1b=0c=0d=0e=1
a=1b=0c=0d=1e=1
a=1b=0c=1d=0e=1
a=1b=0c=1d=1e=0
a=1b=1c=0d=0e=1
a=1b=1c=0d=1e=1
a=1b=1c=1d=0e=0
a=1b=1c=1d=1e=0
```

EXPERIMENT-2

Design Verilog HDL to implement Binary Adder– Half , Full , and 4-bit Full adder.

AIM: To design i) Half adder

ii) Full adder

iii) 4-bit full adder using basic logic gates and simulate their operations.

SIMULATOR USED: ISE DESIGN SUIT 14.7

THEORY:

i) **Half Adder:** Half adder is a basic combinational design that can add two single bits and results to a sum and carry bit as an output.

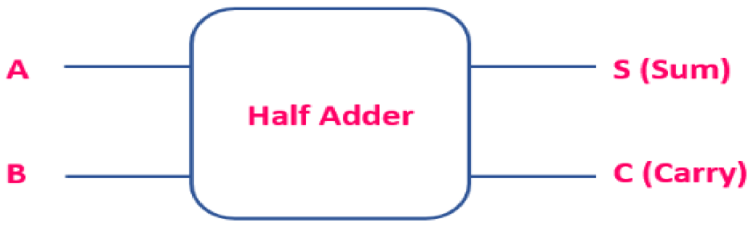
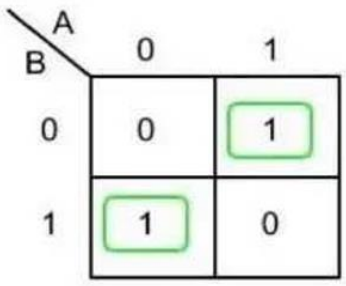


Fig 2.1 Block Diagram of Half Adder

Truth Table For Half Adder

Inputs		Outputs	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



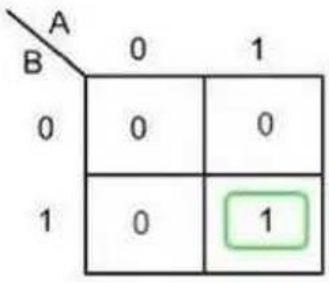


Fig 2.2 K-MAP for SUM **Fig 2.3 K-MAP for CARRY**

The simplified expression for K-map are 1) Sum =A’B+AB’

2) Carry=AB

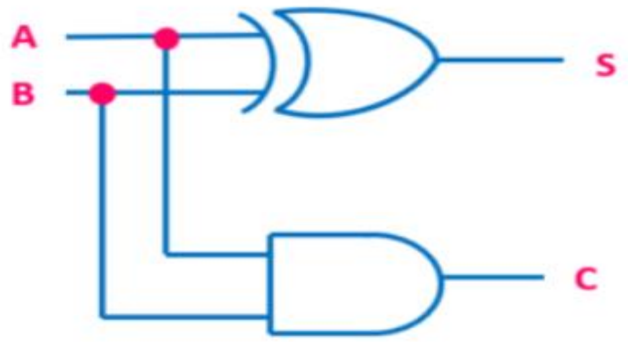


Fig 2.4 Logic Circuit for Half Adder

The half adder circuit is suitable for the addition of two bits at the LSB (Least Significant Bit) position. Because during the addition of two bits at the LSB position, there is no incoming carry. But whenever there is an incoming carry (C_{in}) along with the two bits, then a full adder circuit can be used.

Procedure

1. Write a Verilog Code using appropriate modelling style (Structural , data flow, or behavioral) to describe the desired logic. Ensure Your Verilog code includes input and output ports, logical operations and any necessary components. Obtain RTL Schematic for Half adder.

Verilog Code For Half Adder

```
module ha(A,B, S,C);
   input A,B;
   output S,C;
          xor(S,A,B);
       and(C,A,B);
endmodule
```

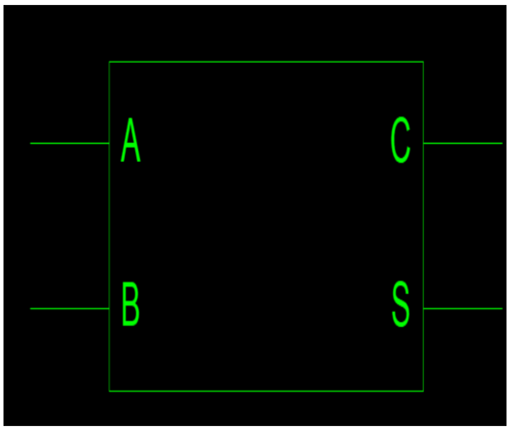


Fig 2.5 Simple Circuit

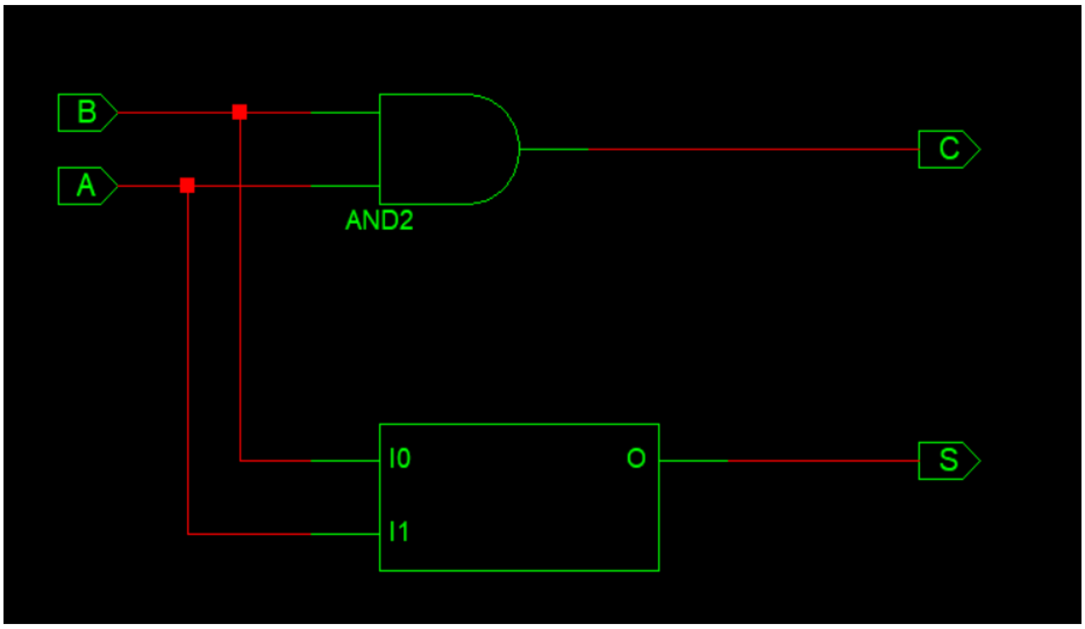


Fig 2.6 RTL Schematic for Half Adder

2. Create a testbench module that instantiates your Verilog module, defines test inputs and applies stimulus and generate the simulation waveform.

Testbench code for Half Adder

```
module ha_tb_v;
        // Inputs
        reg A;
        reg B;

        // Outputs
        wire S;
        wire C;

        // Instantiate the Unit Under Test (UUT)
        ha uut (
                .A(A),
                .B(B),
                .S(S),
                .C(C)
        );
        initial begin
                A = 0;B = 0;

                #10 A=0;B=1;
                #10 A=1;B=0;
                  #10 A=1;B=1;
        end

endmodule
```

Simulation Output

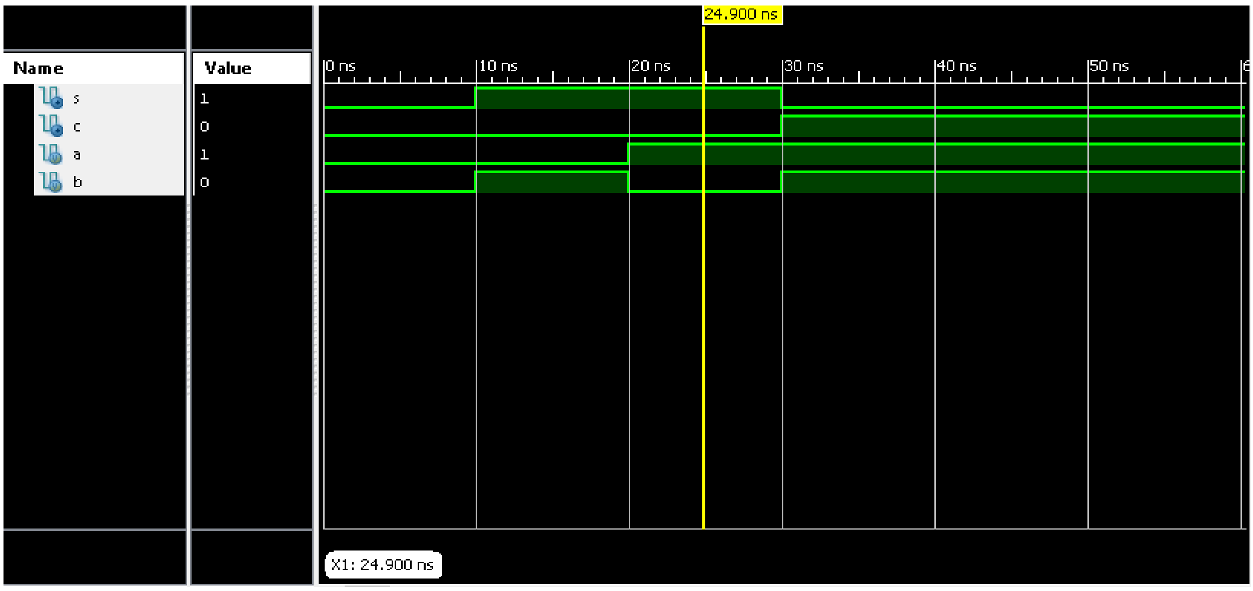


Fig 2.7 RTL Simulation Waveform for Half Adder.

3. Review the console output for results and debugging information.

Console Window Output

```
A=0 B=0 S=0, C=0
A=0 B=1 S=1, C=0
A=1 B=0 S=1, C=0
A=1  B=1  S=0,  C=1
```

ii) Full Adder

The full adder is the combinational circuit that adds the two bits along with the incoming carry (C_{in}) and generates the sum bit (S) and an outgoing carry bit (C_{out}) as an output. The truth table of the full adder is shown below.

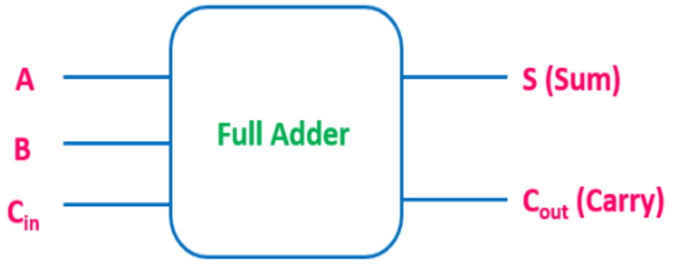
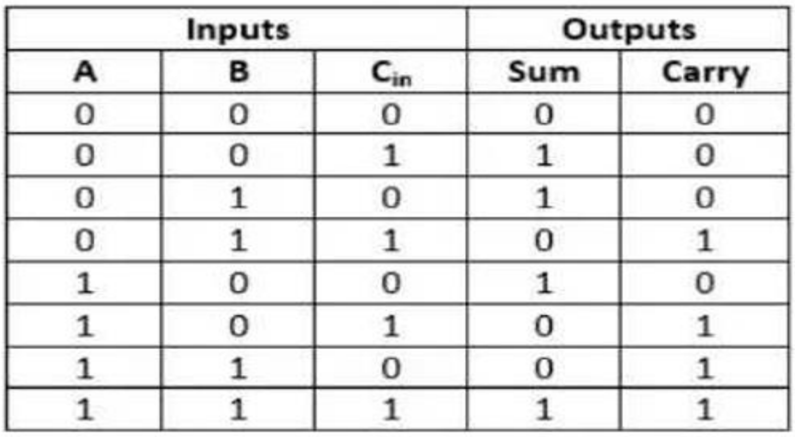


Fig 2.8 Block Diagram for Full Adder

Truth table for Full Adder



Inputs			Outputs	
A	B	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

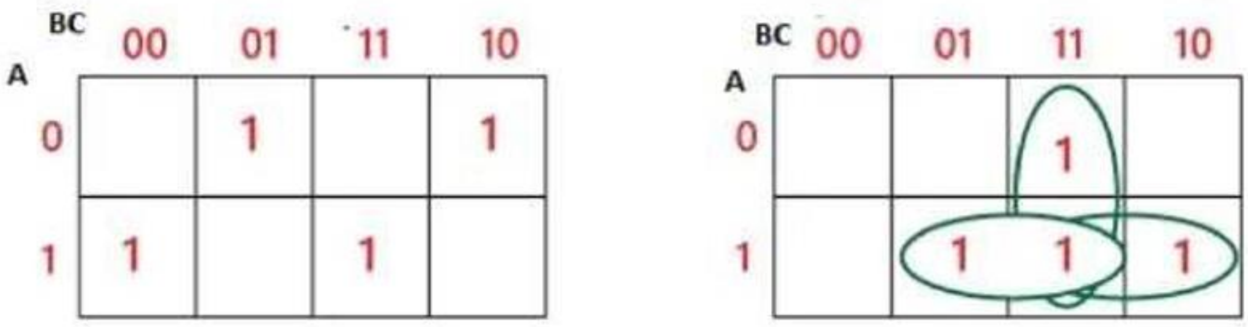


Fig2.9 K-Map For SUM and CARR

The Simplified expressions are

Sum = A’B’C+A’BC’+AB’C’+ABC

Carry = AB+BC+CA

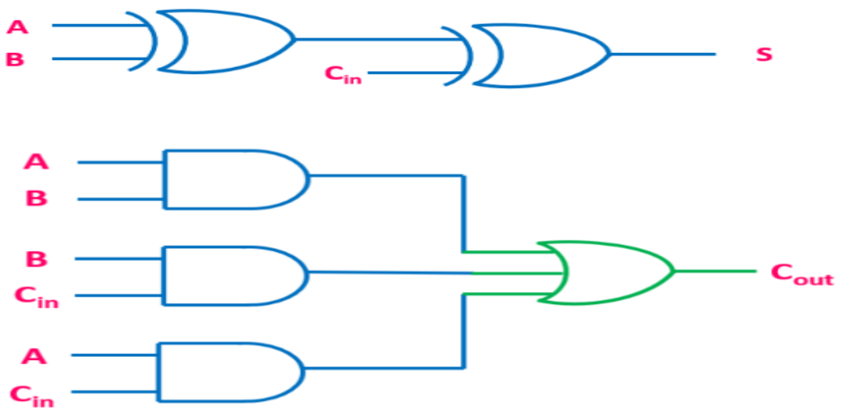


Fig 2.10 logic circuit for the sum (S) and the carry (C_{out}) output

Procedure

1. Write a Verilog Code using appropriate modelling style (Structural , data flow, or behavioral) to describe the desired logic. Ensure Your Verilog code includes input and output ports, logical operations and any necessary components. Obtain RTL Schematic for Full adder.

Verilog Code For Full Adder

```
module fa(A,B,Cin, S,CO);
   input A,B,Cin;
   output reg S,CO;
        always @ (*)
        begin
        S=A^B^Cin;
        CO= (A&B)|(A&Cin)|(Cin&A);
   end
endmodule
```

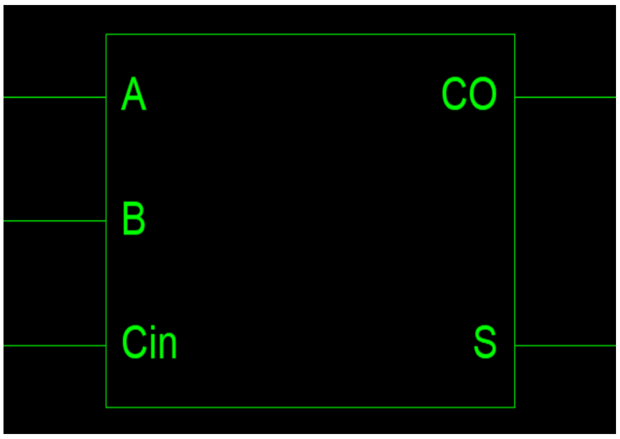


Fig 2.11 Simple Circuit

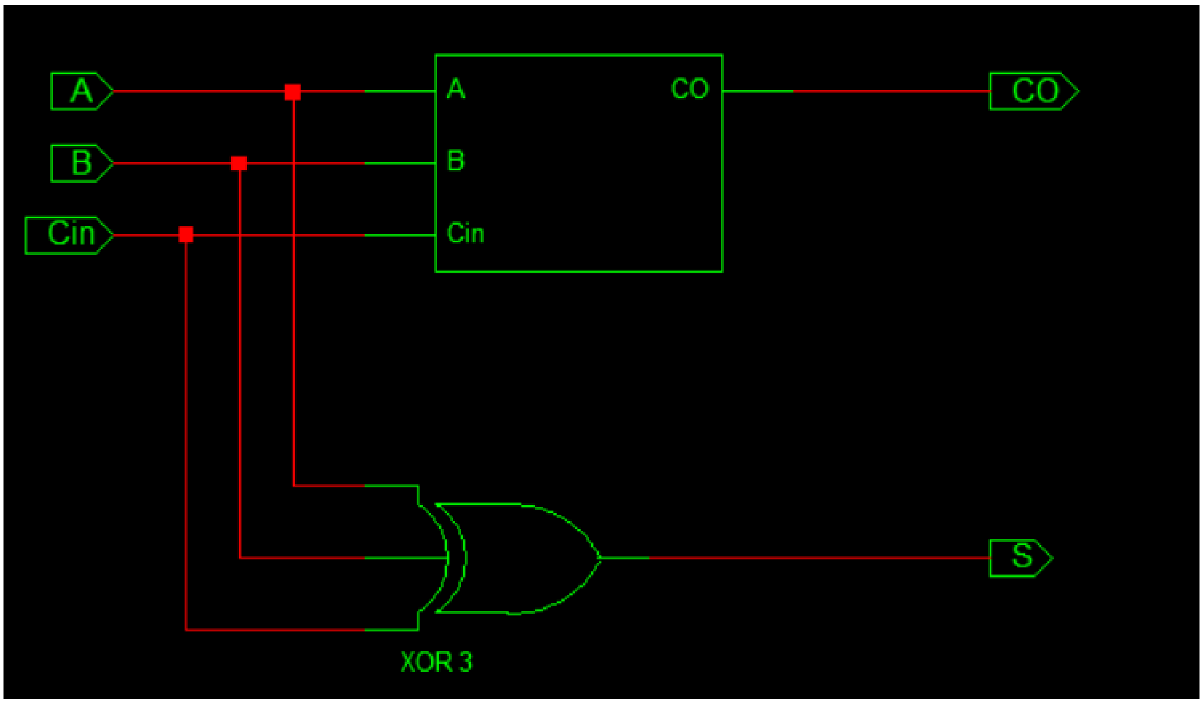


Fig 2.12 RTL Schematic for Full Adder

2. Create a testbench module that instantiates your Verilog module, defines test inputs and applies stimulus and generate the simulation waveform.

Testbench code for Full Adder

```
module fa_tb_v;
        // Inputs
        reg A;
        reg B;
        reg Cin;
        // Outputs
        wire S;
        wire CO;
        // Instantiate the Unit Under Test (UUT)
        fa uut (
                .A(A),
                .B(B),
                .Cin(Cin),
                .S(S),
                .CO(CO)
        );
        initial begin
                // Initialize Inputs
         $monitor("A=%bB=%bCin=%bS=%bCO=%b",A,B,Cin,S,CO);
                #10 A=0;B=0;Cin=0;
                  #10 A=0;B=0;Cin=1;
                #10 A=0;B=1;Cin=0;
                #10 A=0;B=1;Cin=1;
                #10 A=1;B=0;Cin=0;
                #10 A=1;B=0;Cin=1;
                #10 A=1;B=1;Cin=0;
                #10 A=1;B=1;Cin=1;
        end
    endmodule
```

Simulation Output

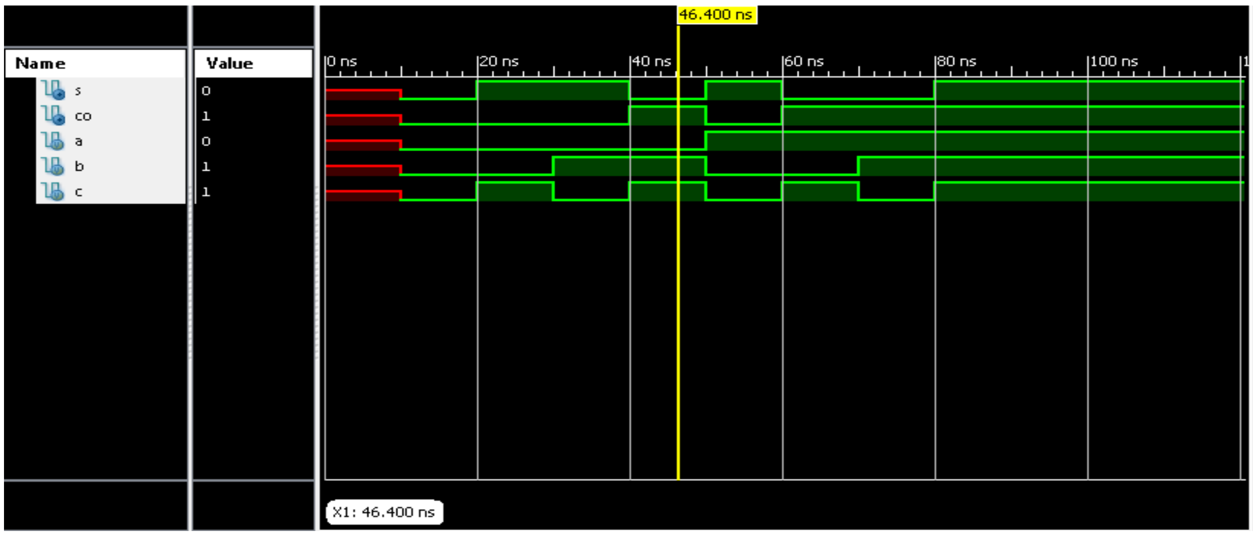


Fig 2.13 RTL Simulation Waveform for Full Adder.

3. Review the console output for results and debugging information.

Console Window Output

```
A=0 B=0 Cin=0 S=0 CO=0
A=0 B=0 Cin=1 S=1 CO=0
A=0 B=1 Cin=0 S=1 CO=0
A=0 B=1 Cin=1 S=0 CO=1
```

A=1 B=0 Cin=0 S=1 CO=0

A=1 B=0 Cin=1 S=0 CO=1

A=1 B=1 Cin=0 S=0 CO=1

A=1 B=1 Cin=1   S=1   CO=1

iii) Four-Bit Full Adder

Binary adders are implemented to add two binary numbers. So in order to add two 4 bit binary numbers, we will need to use 4 full-adders.

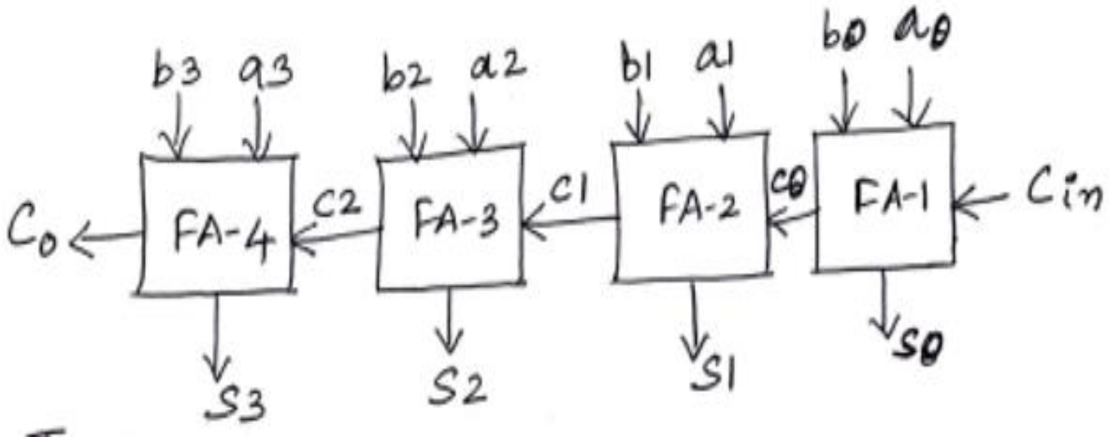


Fig 2.14 Block Diagram of 4-bit Full Adder

Truth Table for Four Bit Full Adder

S.NO	cin	a3	a2	a1	a0	b3	b2	b1	b0	s3	s2	s1	s0	c0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	1	0	0	0	1	0	0	1	0	0
3	0	0	0	1	0	0	0	1	0	0	1	0	0	0
4	0	0	0	1	1	0	0	1	1	0	1	1	0	0
5	0	0	1	0	0	0	1	0	0	1	0	0	0	0
6	0	0	1	0	1	0	1	0	1	1	0	1	0	0
7	0	0	1	1	0	0	1	1	0	1	1	0	0	0
8	0	0	1	1	1	0	1	1	1	1	1	1	0	0
9	0	1	0	0	0	1	0	0	0	0	0	0	0	1
10	0	1	0	0	1	1	0	0	1	0	0	1	0	1
11	0	1	0	1	0	1	0	1	0	0	1	0	0	1
12	0	1	0	1	1	1	0	1	1	0	1	1	0	1
13	0	1	1	0	0	1	1	0	0	1	0	0	0	1
14	0	1	1	0	1	1	1	0	1	1	0	1	0	1
15	0	1	1	1	0	1	1	1	0	1	1	0	0	1
16	0	1	1	1	1	1	1	1	1	1	1	1	0	1

Procedure

1. Write a Verilog Code using appropriate modelling style (Structural , data flow, or behavioral) to describe the desired logic. Ensure Your Verilog code includes input and output ports, logical operations and any necessary components. Obtain RTL Schematic for 4-bit Full adder.

Verilog Code For 4-Bit Full Adder

```
module fa4bit(s,co,a,b,cin);
   input [3:0]a,b;
   input cin;
   output [3:0]s;
   output co;
          wire [2:0]c;
   fa_behav fa1(s[0],c[0],a[0],b[0],cin);
          fa_behav fa2(s[1],c[1],a[1],b[1],c[0]);
          fa_behav fa3(s[2],c[2],a[2],b[2],c[1]);
          fa_behav fa4(s[3],co,a[3],b[3],c[2]);
endmodule


module fa_behav(s,co,a,b,c);
   input a,b,c;
   output reg s,co;
   always @ (*)
          begin
          s=a^b^c;
          co= (a&b)|(b&c)|(c&a)
end
endmodule
```

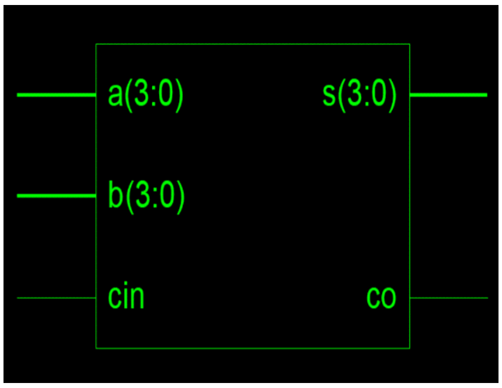


Fig 2.15 Simple Circuit

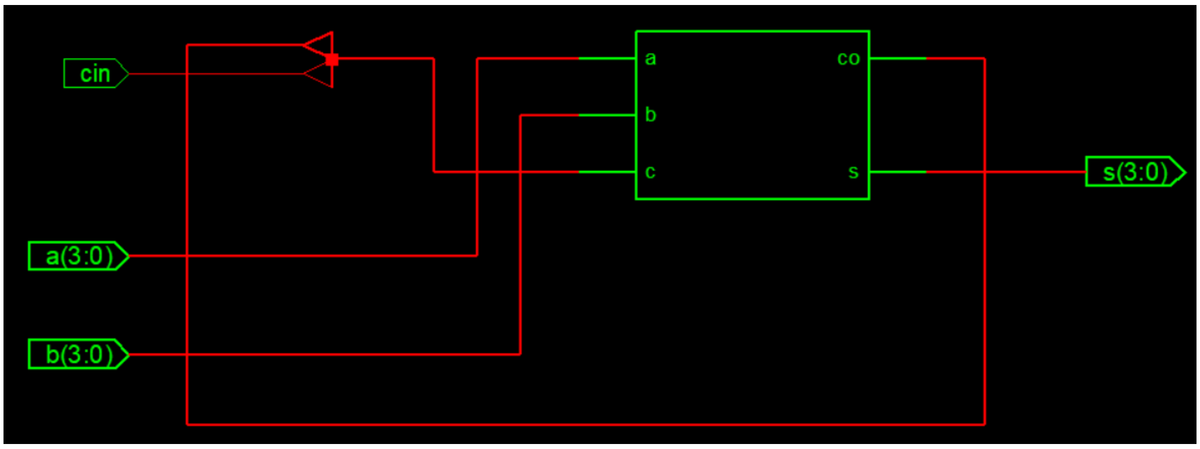


Fig 2.16 RTL Schematic for 4-bit Full Adder

2. Create a testbench module that instantiates your Verilog module, defines test inputs and applies stimulus and generate the simulation waveform.

Testbench code for 4-Bit Full Adder

```
module fa4bit_tb;
        // Inputs
        reg [3:0] a;
        reg [3:0] b;
        reg cin;
        // Outputs
        wire [3:0] s;
        wire co;
   integer i;
        // Instantiate the Unit Under Test (UUT)
        fa4bit uut (
                .s(s),
                .co(co),
                .a(a),
                .b(b),
                .cin(cin)
        );
initial begin
                // Initialize Inputs
                 //$monitor("a=0x%0h b=0x%0h c=%b s=0x%0h co=ox%0h",a,b,c,s,co);
                #10 a =4'b0000; b=4'b0000;cin = 0;
                #10 a =4'b0001; b=4'b0001;
                #10 a =4'b0010; b=4'b0010;
                #10 a =4'b0011; b=4'b0011;
                #10 a =4'b0100; b=4'b0100;
                #10 a =4'b0101; b=4'b0101;
                #10 a =4'b0110; b=4'b0110;
                #10 a =4'b0111; b=4'b0111;
                #10 a =4'b1000; b=4'b1000;
                #10 a =4'b1001; b=4'b1001;
                #10 a =4'b1010; b=4'b1010;
                #10 a =4'b1011; b=4'b1011;
                #10 a =4'b1100; b=4'b1100;
                #10 a =4'b1101; b=4'b1101;
                #10 a =4'b1110; b=4'b1110;
                #10 a =4'b1111; b=4'b1111;
                // Wait 100 ns for global reset to finish
                #100;
                // using for loop to random the values to the input
                //for(i=0 ; i<5 ; i=i+1) begin
                  // #10 a <= $random;
   //b <= $random;
                //end
        end
endmodule
```

Simulation Output

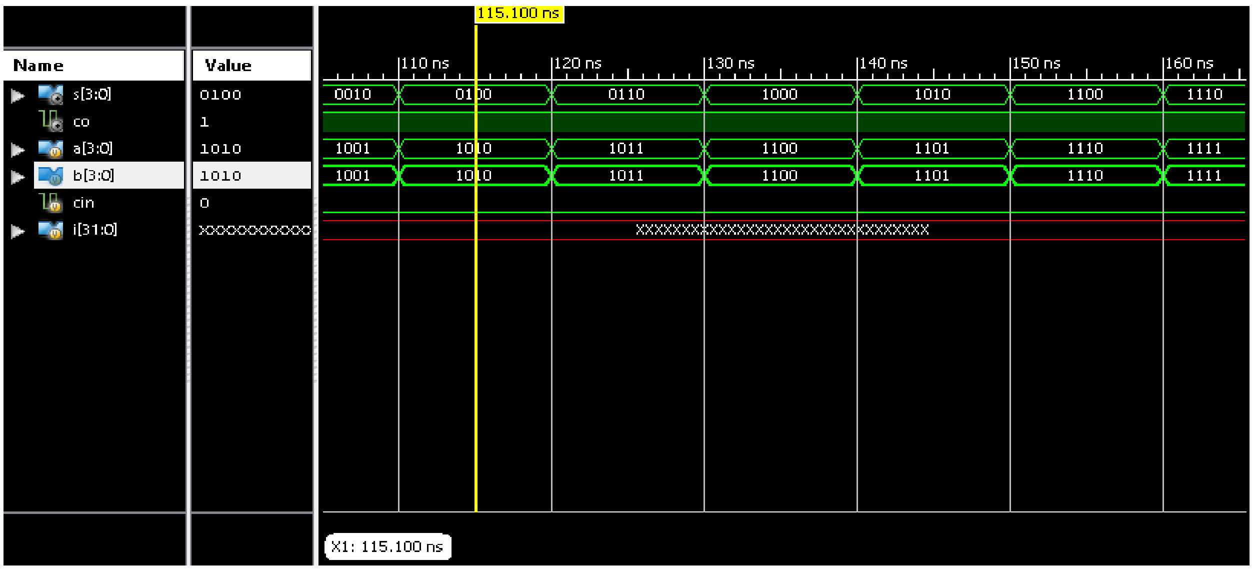


Fig 2.17 RTL Simulation Waveform for 4-bitFull Adder.

3. Review the console output for results and debugging information.

Console Window Output

```
a=0000 b=0000 c=0 s=0000 co=0
a=0001 b=0001 c=0 s=0010 co=0
a=0010 b=0010 c=0 s=0100 co=0
a=0011 b=0011 c=0 s=0110 co=0
a=0100 b=0100 c=0 s=1000 co=0
a=0101 b=0101 c=0 s=1010 co=0
a=0110 b=0110 c=0 s=1100 co=0
a=0111 b=0111 c=0 s=1110 co=0
a=1000 b=1000 c=0 s=0000 co=1
a=1001 b=1001 c=0 s=0010 co=1
a=1010 b=1010 c=0 s=0100 co=1
a=1011 b=1011 c=0 s=0110 co=1
a=1100 b=1100 c=0 s=1000 co=1
a=1101 b=1101 c=0 s=1010 co=1
a=1110 b=1110 c=0 s=1100 co=1
a=1111 b=1111 c=0 s=1110 co=1
```

EXPERIMENT-3

Design Verilog HDL to implement Binary Substractor– Half , Full , and 4-bit Full Subtractor.

AIM: To design i) Half Subtractor ii) Full Subtractoriii) 4-bit full Subtractor using basic logic gates and simulate their operations.

SIMULATOR USED: ISE DESIGN SUIT 14.7

THEORY:

i) Half Subtractor

A half-subtractor is a combinational logic circuit that have two inputs and two outputs (i.e. difference and borrow). The half subtractor produces the difference between the two binary bits at the input and also produces a borrow output (if any). In the subtraction (A-B), A is called as Minuend bit and B is called as Subtrahend bit.

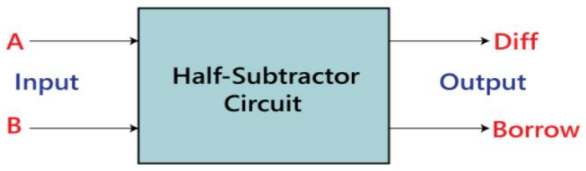


Fig 3.1 Block Diagram of Half Subtractor

Truth Table For Half Subtractor

Inputs		Outputs	
A	B	Diff	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

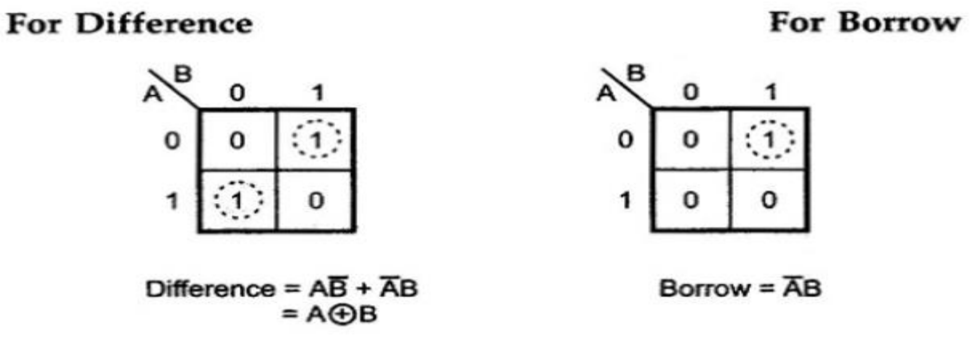


Fig 3.2 K-Map for Difference and Barrow

The SOP form of the **Diff** and **Borrow** is as follows:

Diff=A'B+AB'

Borrow = A'B

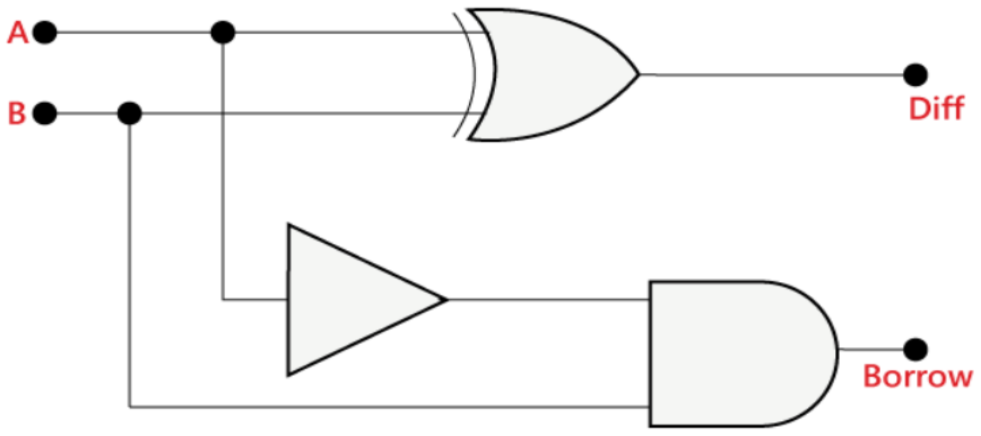


Fig 3.3 Logic circuit for the Half Subtractor

Procedure

1. Write a Verilog code for Half Subtractor and obtain the RTL Schematic

Verilog Code for Half Subtractor

```
module hsub(a,b,d,bo);
   input a,b;
   output d,bo;
    wire w1;
    not n1(w1,a);
    xor x1(d,a,b);
    and a1(bo,w1,b);
endmodule
```

Fig 3.4 Simple Circuit

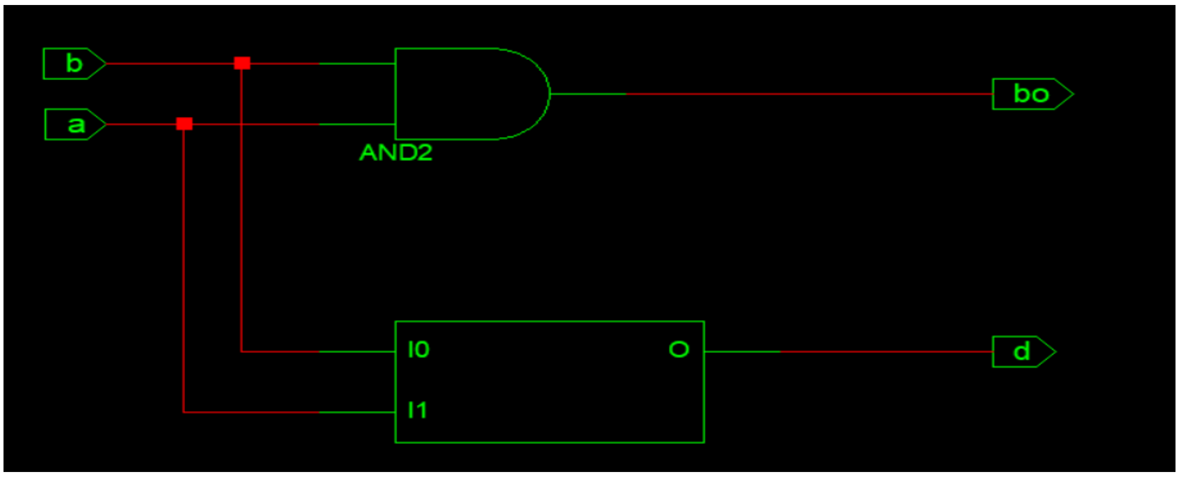


Fig 3.5 RTL Schematic for Half Subtractor

2. Write a test bench code for Half Subtractor and obtain the RTL Simulation Waveform.

Test Bench Code For Half Subtractor

```
module hsub_tb
 // Inputs
 reg a;
 reg b;
 // Outputs
 wire d;
 wire bo;
 // Instantiate the Unit Under Test (UUT)
 hsub uut (
      .a(a),
      .b(b),
      .d(d),
      .bo(bo));
 initial begin
       $monitor("a=%bb=%bd=%bbo=%b",a,b,d,bo);
      #10 a=0;b=0;
   #10 a=0;b=1;
      #10 a=1;b=0;
      #10 a=1;b=1;
 end
endmodule
```

Simulation Output

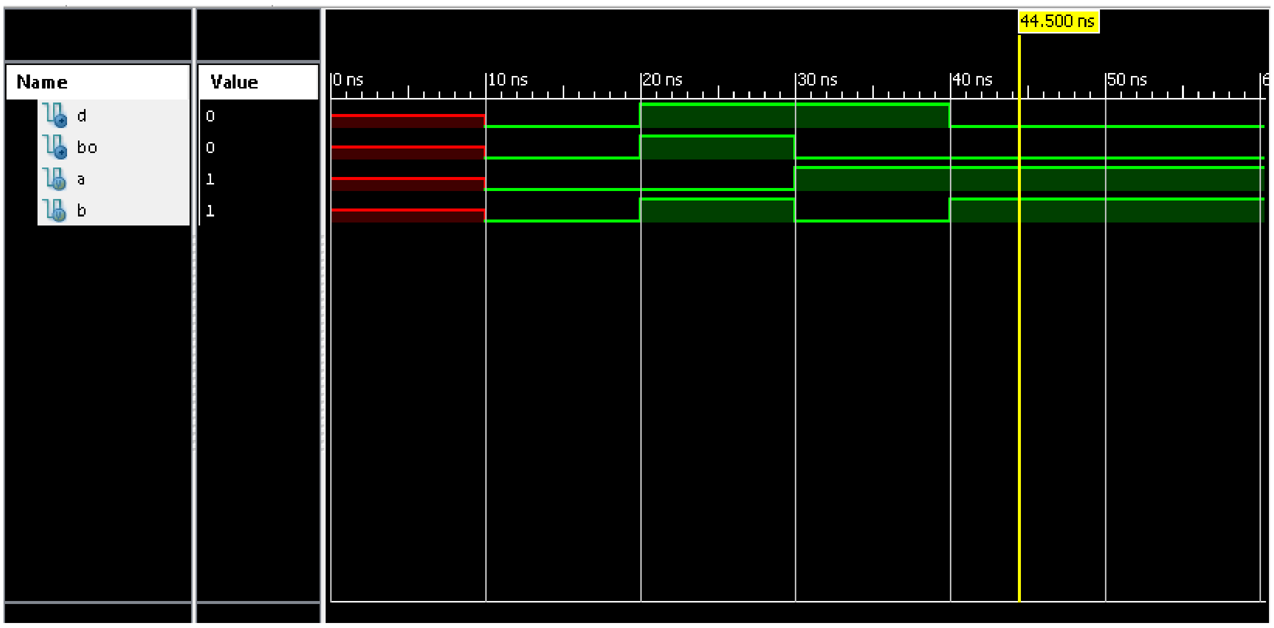


Fig 3.6 RTL Simulation Waveform for Half Subtractor

3. Review the console output for results and debugging information.

Console Window Output

```
a=0b=0d=0bo=0
a=0b=1d=1bo=1
a=1b=0d=1bo=0
a=1b=1d=0bo=0
```

ii) Full Subtractor

The Half Subtractor is used to subtract only two numbers. To overcome this problem, a full subtractor was designed. The full subtractor is used to subtract three 1-bit numbers A, B, and C, which are minuend, subtrahend, and borrow, respectively. The full subtractor has three input states and two output states i.e., diff and borrow.

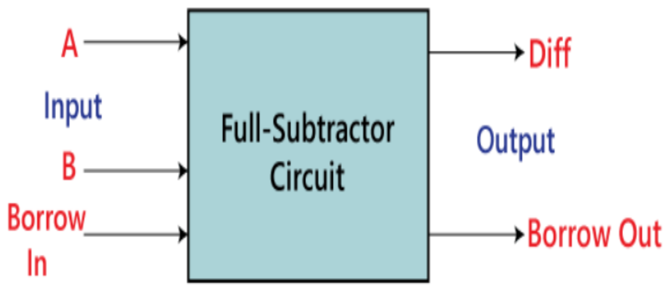
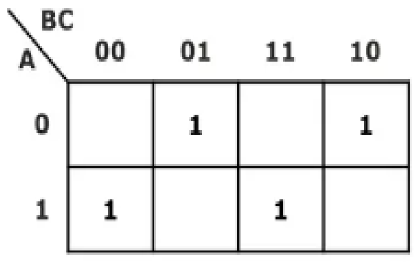


Fig 3.7 Block Diagram of Full Subtractor

Truth Table For Full Subtractor

Inputs			Outputs	
A	B	$Borrow_{in}$	Diff	Borrow
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



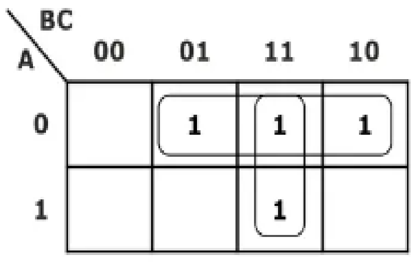


Fig 3.8 K-Map for Difference and Barrow

Logical expression for difference –

```
D    = A'B'Bin + A'BBin' + AB'Bin' + ABBin
     = Bin(A'B' + AB)  + Bin'(AB' + A'B)
     = Bin( A XNOR B) + Bin'(A XOR B)
     = Bin (A XOR B)'  +  Bin'(A XOR B)
     = Bin XOR (A XOR B)
     = (A XOR B) XOR Bin
```

Logical expression for borrow –

```
Bout = A'B'Bin + A'BBin' + A'BBin + ABBin
     = Bin(AB + A'B') + A'B(Bin + Bin')
     = Bin( A XNOR B) + A'B
     = Bin (A XOR B)' + A'B
```

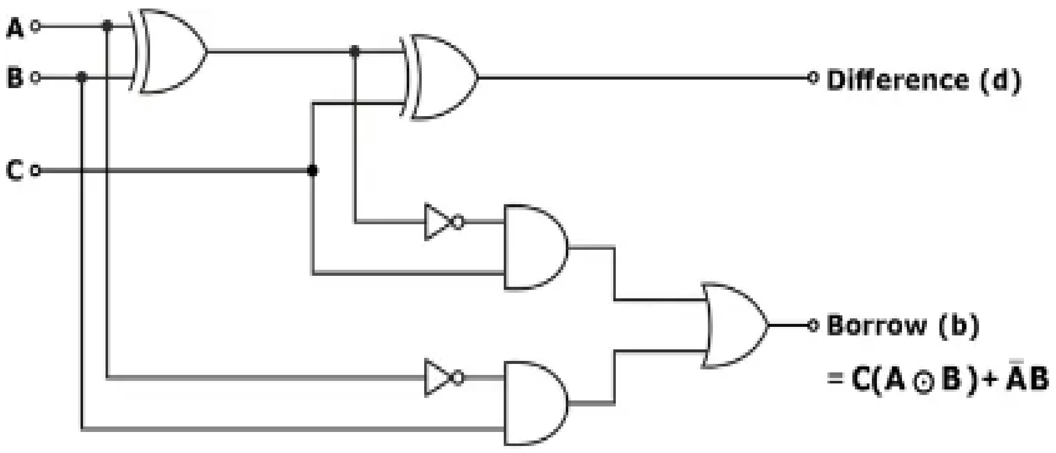


Fig 3.9 Logic circuit for the Full Subtractor

Procedure

1. Write a Verilog code for Full Subtractor and obtain the RTL Schematic

Verilog Code for Full Subtractor

```
module fsub(a,b,bin,d,bout);
   input  a,b,bin;
   output d,bout;
   wire w1,w2,w3,w4,w5;
    xor xor1(w1,a,b);
    xor xor2(d,w1,bin);
    not not1(w2,w1);
    not not2(w3,a);
    and and1(w4,w2,bin);
    and and2(w5,w3,b);
    or or1(bout,w4,w5);
endmodule
```

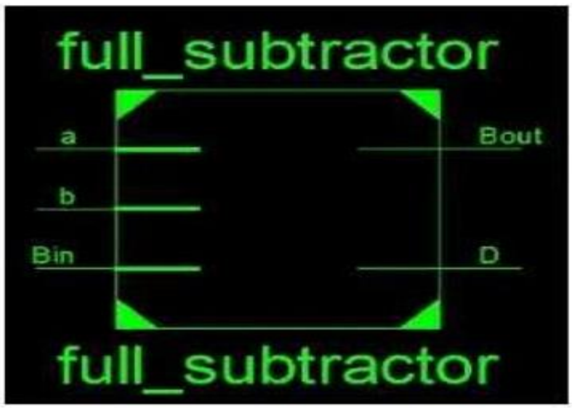


Fig 3.10 Simple Circuit

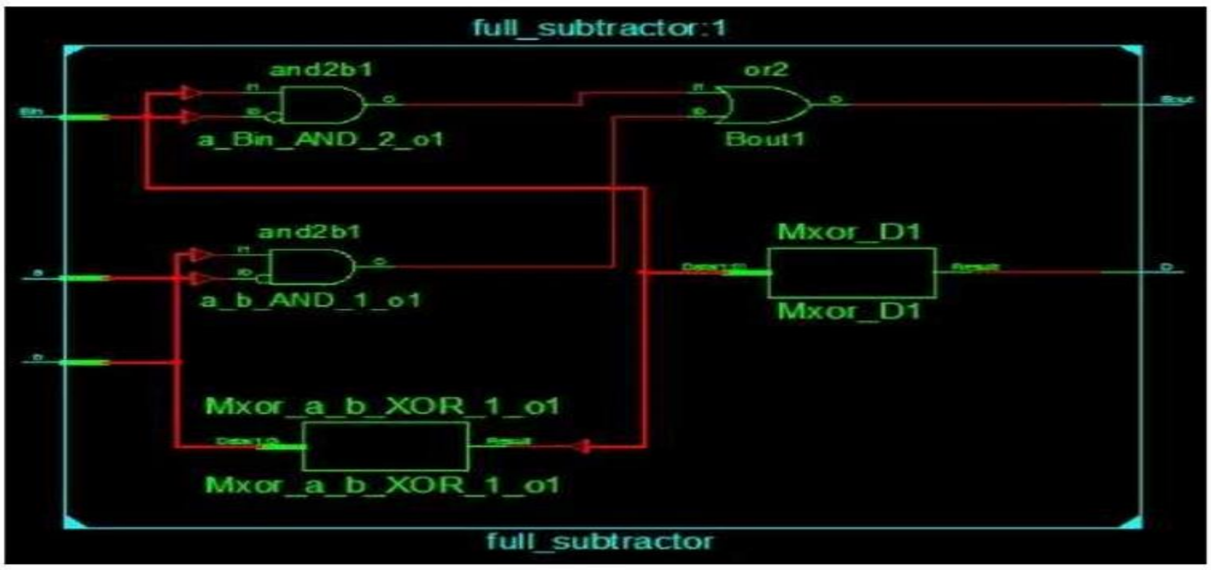


Fig 3.11 RTL Schematic for Full Subtractor

2. Write a test bench code for Full Subtractor and obtain the RTL Simulation Waveform.

Test Bench Code For Full Subtractor

```
module fsub_tb;
  // Inputs
  reg a;
  reg b;
  reg bin;
  // Outputs
  wire d;
  wire bout;
  // Instantiate the Unit Under Test (UUT)
  fsub uut (
    .a(a),
    .b(b),
    .bin(bin),
    .d(d),
    .bout(bout)
  );
  initial begin
     $monitor("a=%b b=%b bin=%b d=%b bout=%b",a,b,bin,d,bout);
    #10 a=0;b=0;bin=0;
    #10 a=0;b=0;bin=1;
    #10 a=0;b=1;bin=0;
    #10 a=0;b=1;bin=1;
    #10 a=1;b=0;bin=0;
    #10 a=1;b=0;bin=1;
    #10 a=1;b=1;bin=0;
    #10 a=1;b=1;bin=1;
  end
   endmodule
```

Simulation Output

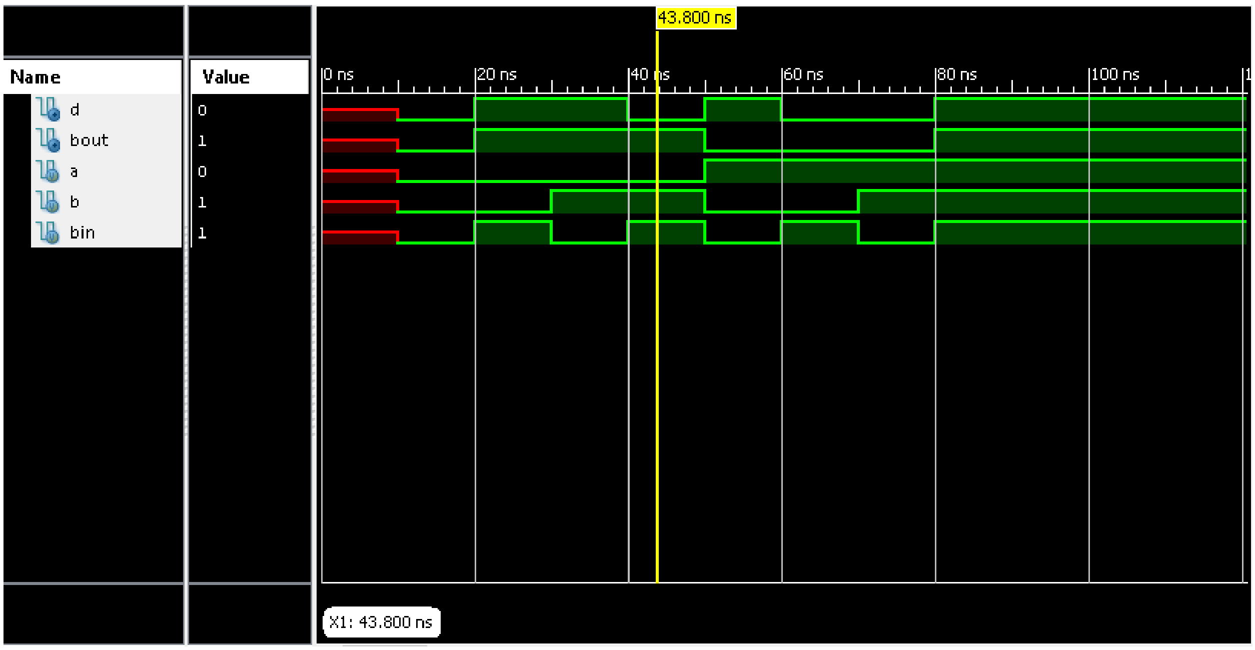


Fig 3.12 RTL Simulation Waveform for Full Subtractor

3. Review the console output for results and debugging information.

Console Window Output

```
a=0 b=0 bin=0 d=0 bout=0
a=0 b=0 bin=1 d=1 bout=1
a=0 b=1 bin=0 d=1 bout=1
a=0 b=1 bin=1 d=0 bout=1
a=1 b=0 bin=0 d=1 bout=0
a=1 b=0 bin=1 d=0 bout=0
a=1 b=1 bin=0 d=0 bout=0
a=1 b=1 bin=1 d=1 bout=1
```

iii) Four-Bit Full Subtractor

Here, A4, A3, A2, A1 is minuend and B4, B3, B2, B1 is subtrahend. S4, S3, S2, S1 is result of substraction where C4 is final carry which is ignored.

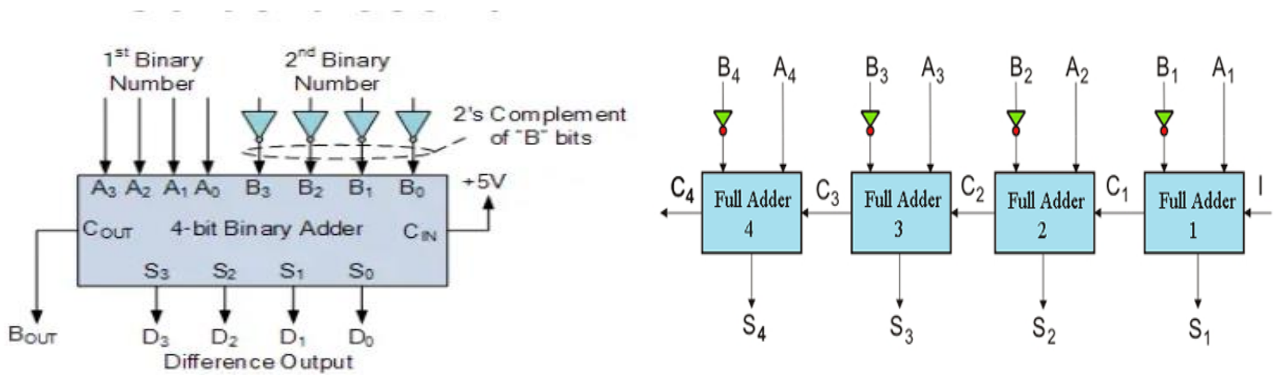


Fig 3.13 Block Diagram of 4-bit Full Subtractor

Truth Table for Four Bit Full Subtractor

Bin	A				B				D				Bout
Bin	A3	A2	A1	A0	B3	B2	B1	B0	D3	D2	D1	D0	Bout
1	0	0	0	0	1	1	1	1	0	0	0	1	0
	0	0	0	1	1	1	0	0	0	0	1	1	0
	0	0	1	0	1	1	0	1	0	1	0	1	0
	0	0	1	1	1	1	0	0	0	1	1	1	0
	0	1	0	0	1	0	1	1	1	0	0	1	0
	0	1	0	1	1	0	1	0	1	0	1	1	0
	0	1	1	0	0	0	1	1	1	1	0	1	0
	0	1	1	1	1	0	0	0	1	1	1	1	0
	1	0	0	0	0	1	1	1	0	0	0	1	1
	1	0	0	1	0	1	1	0	0	0	1	1	1
	1	0	1	0	0	1	0	1	0	1	0	1	1
	1	0	1	1	0	1	0	0	0	1	1	1	1
	1	1	0	0	0	0	1	1	1	0	0	1	1
	1	1	0	1	0	0	1	0	1	0	1	1	1
	1	1	1	0	0	0	0	1	1	1	0	1	1
	1	1	1	1	0	0	0	0	1	1	1	1	1

Procedure

1. Write a Verilog Code using appropriate modelling style (Structural , data flow, or behavioral) to describe the desired logic. Ensure Your Verilog code includes input and output ports, logical operations and any necessary components.

Verilog Code For 4-Bit Full Subtractor

```
module sub4b(a,b,bin,d,bout);
   input bin;
   output bout ;
   input [3:0] a,b ;
          output [3:0] d ;
          wire [2:0] bi;
          fa_behav sub1(a[0],~b[0],bin,d[0],bi[0]);
          fa_behav sub2(a[1],~b[1],bi[0],d[1],bi[1]);
          fa_behav sub3(a[2],~b[2],bi[1],d[2],bi[2]);
          fa_behav sub4(a[3],~b[3],bi[2],d[3],bout);
endmodule
          module fa_behav(a,b,c,s,co);
   input a,b,c;
   output reg s,co;
   always @ (*)
          begin
          s=a^b^c;
          co= (a&b)|(b&c)|(c&a);
   end
endmodule
```

2. Create a testbench module that instantiates your Verilog module, defines test inputs and applies stimulus and generate the simulation waveform.

Testbench code for 4-Bit Full Subtractor

```
    module sub4b_tb;
      // Inputs
      reg [3:0] a;
      reg [3:0] b;
      reg bin;
      // Outputs
      wire [3:0] d;
      wire bout;
    //integer i;
      // Instantiate the Unit Under Test (UUT)
      sub4b uut (
        .a(a),
        .b(b),
        .bin(bin),
        .d(d),
        .bout(bout)
    );
    initial begin
        // Initialize Inputs
         //$monitor("a=0x%0h b=0x%0h c=%b s=0x%0h co=ox%0h",a,b,c,s,co);
        #10 a =4'b0000; b=4'b1111;bin =1;
        #10 a =4'b0001; b=4'b1110;
        #10 a =4'b0010; b=4'b1101;
        #10 a =4'b0011; b=4'b1100;
        #10 a =4'b0100; b=4'b1011;
        #10 a =4'b0101; b=4'b1010;
        #10 a =4'b0110; b=4'b1001;
        #10 a =4'b0111; b=4'b1000;
        #10 a =4'b1000; b=4'b0111;
        #10 a =4'b1001; b=4'b0110;
        #10 a =4'b1010; b=4'b0101;
        #10 a =4'b1011; b=4'b0100;
        #10 a =4'b1100; b=4'b0011;
        #10 a =4'b1101; b=4'b0010;
        #10 a =4'b1110; b=4'b0001;
        #10 a =4'b1111; b=4'b0000;

        // Wait 100 ns for global reset to finish
        #100;
        // using for loop to random the values to the input
        //for(i=0 ; i<5 ; i=i+1) begin
          // #10 a <= $random;
                    // b <= $random;
        //end
    end
endmodule
```

Simulation Output

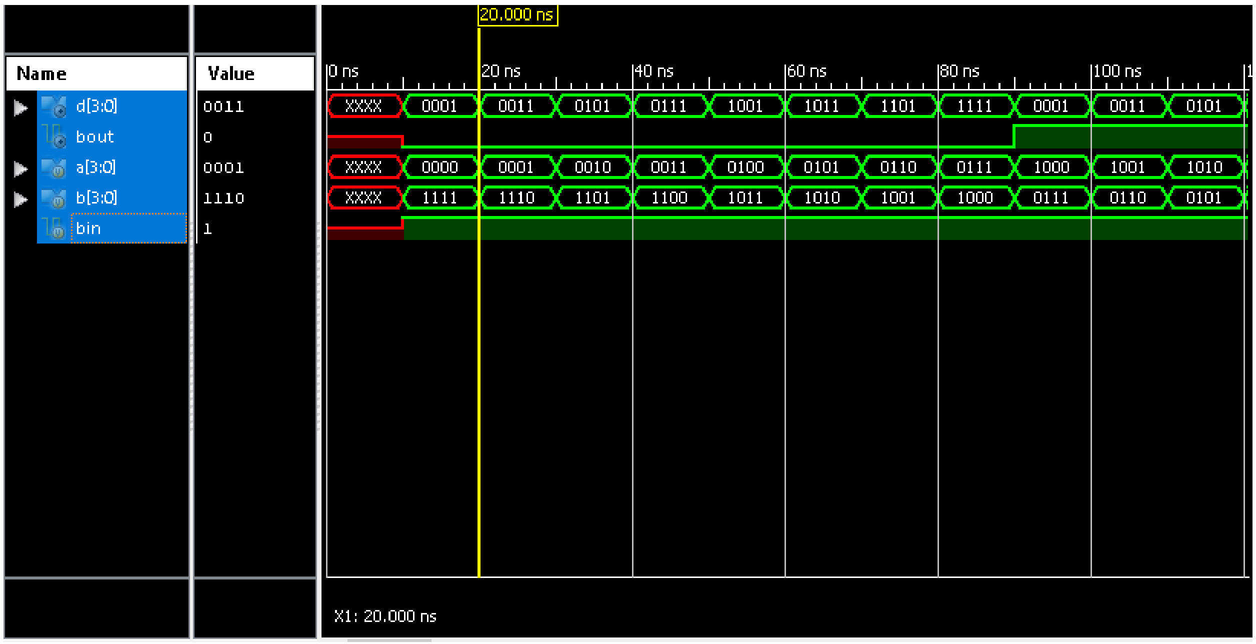


Fig 3.16 RTL Simulation Waveform for 4-bit Full Subtractor.

CONSOLE WINDOW OUTPUT

```
a=0000 b=1111 bin=1 d=0001 bout=0

a=0001 b=1110 bin=1 d=0011 bout=0

a=0010 b=1101 bin=1 d=0101 bout=0

a=0011 b=1100 bin=1 d=0111 bout=0

a=0100 b=1011 bin=1 d=1001 bout=0

a=0101 b=1010 bin=1 d=1011 bout=0

a=0110 b=1001 bin=1 d=1101 bout=0

a=0111 b=1000 bin=1 d=1111 bout=0

a=1000 b=0111 bin=1 d=0001 bout=1

a=1001 b=0110 bin=1 d=0011 bout=1

a=1010 b=0101 bin=1 d=0101 bout=1

a=1011 b=0100 bin=1 d=0111 bout=1

a=1100 b=0011 bin=1 d=1001 bout=1

a=1101 b=0010 bin=1 d=1011 bout=1

a=1110 b=0001 bin=1 d=1101 bout=1

a=1111 b=0000 bin=1 d=1111 bout=1
```

EXPERIMENT-4

Design Verilog HDL to implement Simple circuits using structural, Data flow and Behavioural model

AIM: To design a simple circuit using structural, Data flow and Behavioural model .

SIMULATOR USED: ISE DESIGN SUIT 14.7

THEORY:

i) Structural Modeling

Description : Structural modeling describes a design in terms of interconnected hardware components or modules.It explicitly defines the components and their interconnections.

Use Cases: This modeling style is suitable for designing and connecting pre-defined components or IP blocks, such as multiplexers, adders, and flip-flops.

Implementation : On structural modeling you instantiate predefined components and connect them to create the desired circuit.

Example Verilog code for a 2 to 1 multiplexer using structural modeling:

```
module mux2x1(
   input I0,I1,S,
   output Y );
wire w1,w2,w3;
not(w1,S);
and(w2,I0,w1);
and(w3,I1,S);
or(Y,w2,w3);
endmodule
```

ii) Data Flow Modeling

Description : Data flow modeling specifies how data flows through the circuit. It defines operations on data as function of inputs, similar to equations in mathematics.

Use Cases: Data flow modeling is suitable for designing the desired functionality of circuit without specifying its structure explicitly.

Implementation : You describe the behaviour of the circuit using assignments and combinational logic equations.

Example Verilog code for and gate using Data flow modeling:

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

iii) Behavioural Modeling

Description : iii) Behavioural modeling focuses on high-level functionality,describing how a module behaves without specifying the underlying hardware details.It’s often used for algorithimicdescriptions.

Use Cases: Behavioural modeling is useful for designing modules where the internal hardware is not critical or when you want to abstract away hardware details.

Implementation : You use procedural constructs like always blocks and if-else statements to specify how the modules behaves based on inputs.

Example Verilog code for and full adder using Behavioural modeling:

```
module full_adder(
input a,
input b,
input cin,
output sum,
output cout
);
always @(*) begin
  // Calculate the sum
  sum = a ^ b ^ cin;
  // Calculate the carry-out
  cout = (a & b) | (a & cin) | (b & cin);
end
endmodule
```

Procedure

1. Minimize the following Boolean expression f(A,B,C) = Σm(1,3,6,7)

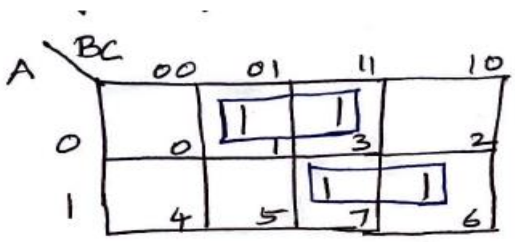


Fig 4.1 K-Map

Thus minimized Boolean expression is F(A,B,C) = AB+A’C

Truth Table

S.no	a	b	c	e
1	0	0	0	0
2	0	0	1	1
3	0	1	0	0
4	0	1	1	1
5	1	0	0	0
6	1	0	1	0
7	1	1	0	1
8	1	1	1	1

2.Write the Verilog code for the given expression using structural flow, data flow and behavioural model.

Obtain the RTL Schematic.

Structural Modeling

Verilog Code :

```
module design_all1(a,b,c,e);
    input a,b,c;
    output e ;
            wire w1,w2,w3;
            not n1(w1,a);
            and a1(w2,a,b);
            and a2(w3,w1,c);
            or or1(e,w2,w3);
endmodule
```

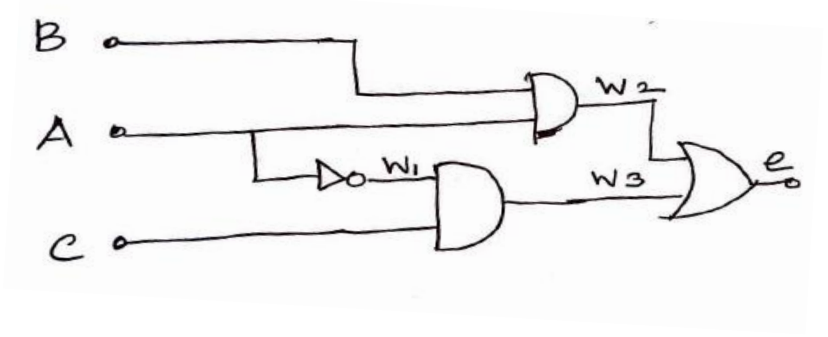


Fig 4.2:Implementation of three Variable Boolean expression

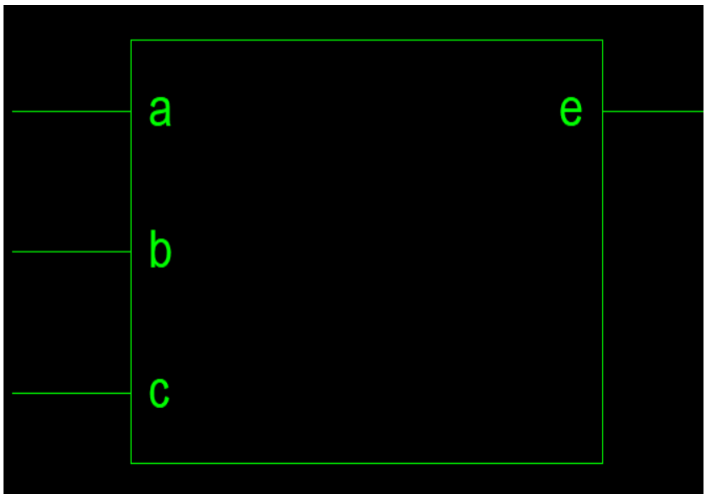


Fig 4.2 Simple Circuit

Data Flow Modeling

Verilog Code :

```
module designall_2(a,b,c,e);
   input a,b,c;
   output e;
          assign e = (a&b)|(~a&c);
endmodule
```

Behavioural Modeling

Verilog Code :

```
module designall_3(a,b,c,e);
   input a,b,c;
   output reg e;
   always @ (*)
          begin
          e = (a&b)|(~a&c);
   end
endmodule
```

2. Write a testbench code for the give expression using any model and obtain the simulation waveform

Test Bench Code

```
    module design_all1_tb;
      // Inputs
      reg a;
      reg b;
      reg c;
      // Outputs
      wire e;
    // Instantiate the Unit Under Test (UUT)
    design_all1 uut (
      .a(a),
      .b(b),
      .c(c),
      .e(e)
    );

    initial begin
        $monitor("a=%bb=%bc=%be=%b",a,b,c,e);
      #10 a=0;b=0;c=0;
      #10 a=0;b=0;c=1;
      #10 a=0;b=1;c=0;
      #10 a=0;b=1;c=1;
      #10 a=1;b=0;c=0;
      #10 a=1;b=0;c=1;
      #10 a=1;b=1;c=0;
      #10 a=1;b=1;c=1;
    end
endmodule
```

Simulation Output

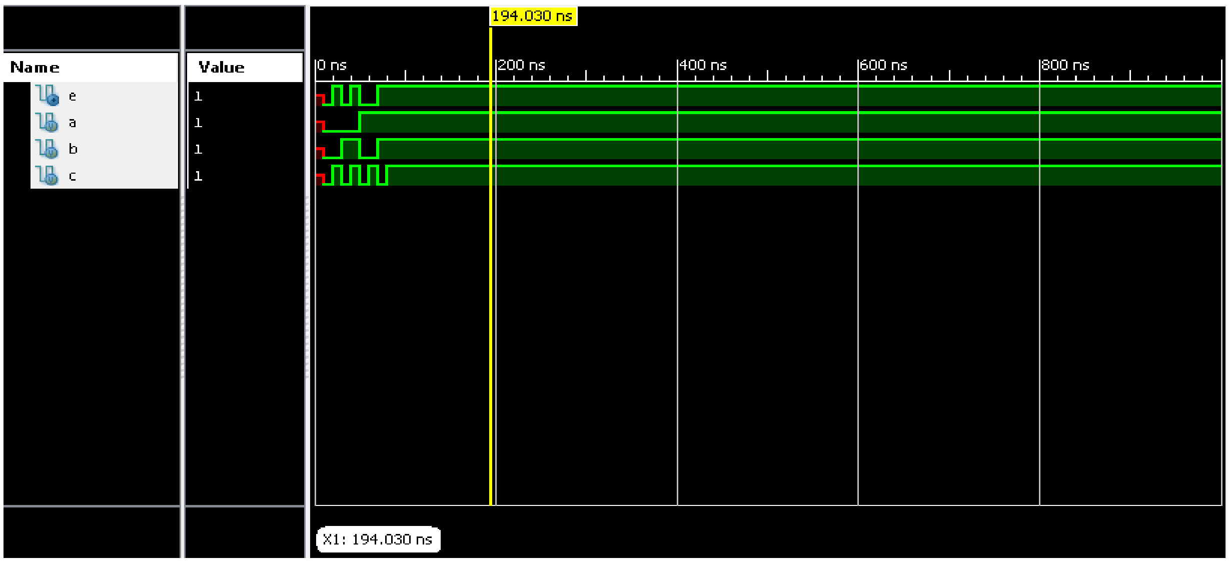


Fig 4.3 RTL Simulation Waveform

Console Window Output

```
a=0b=0c=0e=0
a=0b=0c=1e=1
a=0b=1c=0e=0
a=0b=1c=1e=1
a=1b=0c=0e=0
a=1b=0c=1e=0
a=1b=1c=0e=1
a=1b=1c=1e=1
```

<u>EXPERIMENT-5</u>

Design Verilog HDL to implement Decimal adder or BCD Adder

AIM: To design a decimal or BCD adder using Verilog HDL.

SIMULATOR USED: ISE DESIGN SUIT 14.7

THEORY: The digital systems handles the decimal number in the form of binary coded decimal numbers (BCD). A BCD Adder Circuit that adds two BCD digits and produces a sum digit also in BCD. BCD numbers use 10 digits, 0 to 9 which are represented in the binary form 0 0 0 0 to 1 0 0 1, i.e. each BCD digit is represented as a 4-bit binary number. When we write BCD number say 526, it can be represented as

$$\begin{tikzcd}
5 \arrow[d] & 2 \arrow[d] & 6 \arrow[d] \\
0101 & 0010 & 0110
\end{tikzcd}$$

Here, we should note that BCD cannot be greater than 9.

The addition of two BCD numbers can be best understood by considering the three cases that occur when two BCD digits are added.

Sum Equals 9 or less with carry 0

Let us consider additions of 3 and 6 in BCD.

$$\begin{array}{rcccccl}
6 & 0 & 1 & 1 & 0 & \leftarrow & \text{BCD for 6} \\
+\ 3 & 0 & 0 & 1 & 1 & \leftarrow & \text{BCD for 3} \\
\hline
9 & 1 & 0 & 0 & 1 & \leftarrow & \text{BCD for 9}
\end{array}$$

The addition is carried out as in normal binary addition and the sum is 1 0 0 1, which is BCD code for 9.

Sum greater than 9 with carry 0

Let us consider addition of 6 and 8 in BCD

$$\begin{array}{rcccccl}
6 & 0 & 1 & 1 & 0 & \leftarrow & \text{BCD for 6} \\
+\ 8 & 1 & 0 & 0 & 0 & \leftarrow & \text{BCD for 8} \\
\hline
14 & 1 & 1 & 1 & 0 & \leftarrow & \text{Invalid BCD number}
\end{array}$$

The sum 1 1 1 0 is an invalid BCD number. This has occurred because the sum of the two digits exceeds 9. Whenever this occurs the sum has to be corrected by the addition of six (0110) in the invalid BCD number, as shown below

```
      6              0   1   1   0      ←  BCD for 6
  +   8              1   0   0   0      ←  BCD for 8
  -------          ----------------
     14              1   1   1   0      ←  Invalid BCD number
                 +   0   1   1   0      ←  Add 6 for correction
                   ----------------
 0   0   0   1       0   1   0   0
                                        ←  BCD for 14
 \_____1_____/      \______4_____/
```

After addition of 6 carry is produced into the second decimal position.

Sum equals 9 or less with carry 1

Let us consider addition of 8 and 9 in BCD

```
    8                           1   0   0   0    ←  BCD for 8
 +  9                           1   0   0   1    ←  BCD for 9
 ------      ----------------------------------
   17        0   0   0   1      0   0   0   1    ←  Incorrect BCD result
```

In this, case, result (0001 0001) is valid BCD number, but it is incorrect. To get the correct BCD result correction factor of 6 has to be added to the least significant digit sum, as shown below

```
    8                           1   0   0   0    ←  BCD for 8
 +  9                           1   0   0   1    ←  BCD for 9
 ------      ----------------------------------
   17        0   0   0   1      0   0   0   1    ←  Incorrect BCD result
         +   0   0   0   0      0   1   1   0    ←  Add 6 for correction
             ----------------------------------
             0   0   0   1      0   1   1   1    ←  BCD for 17
```

Going through these three cases of BCD addition we can summarise the BCD addition procedure as follows :

Add two BCD numbers using ordinay binary addition.

If four-bit sum is equal to or less than 9, no correction is needed. The sum is in proper BCD form.

If the four-bit sum is greater than 9 or if a carry is generated from the four-bit sum, the sum is invalid.

To correct the invalid sum, add 01102 to the four-bit sum. If a carry results from this addition, add it to the next higher-order BCD digit.

Thus to implement BCD Adder Circuit we require :

- 4-bit binary adder for initial addition

- Logic circuit to detect sum greater than 9 and
- One more 4-bit adder to add 0110 in the sum if sum is greater than 9 or carry is 1.

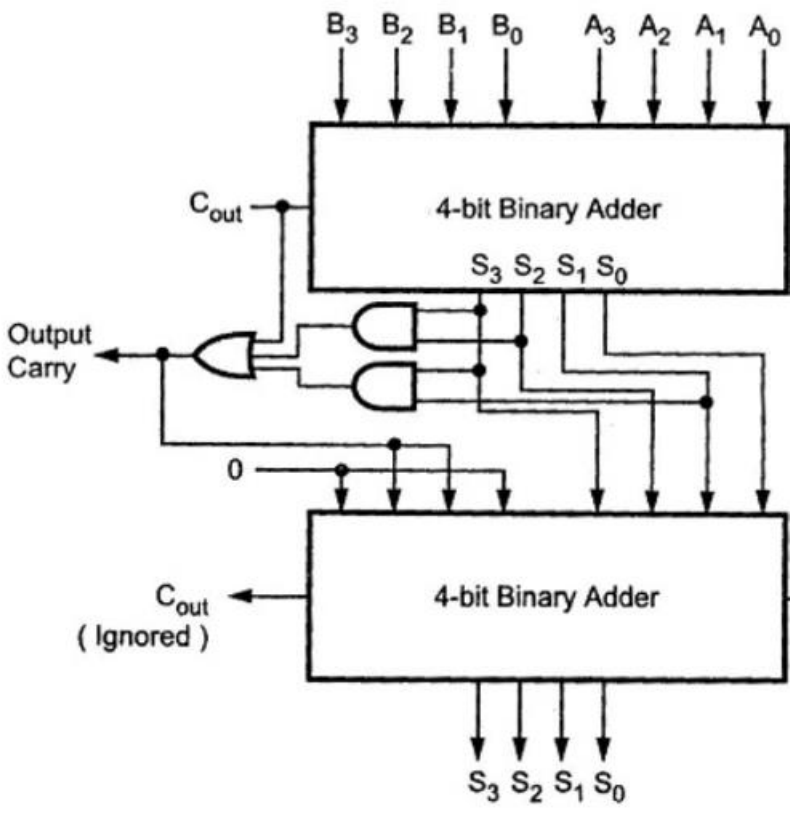


Fig 5.1 Logic Circuit For BCD Adder

The two BCD numbers, are first added in the top 4-bit binary adder to produce a binary sum. When the output carry is equal to zero (i.e. when sum ≤ 9 and Cout = 0) nothing (zero) is added to the binary sum. When it is equal to one (i.e. when sum > 9 or Cout = 1), binary 0110 is added to the binary sum through the bottom 4-bit binary adder. The output carry generated from the bottom binary adder can be ignored, since it supplies information already available at the output-carry terminal.

Procedure

1.Write the Verilog code for BCD adder .

Verilog Code For BCD adder

```
module bcd(a,b,sum,carry);
//declare the inputs and outputs of the module with their sizes.
   input [3:0] a,b;

   output [3:0] sum;
   output carry;
   //Internal variables
```

```
    reg [5:0] sum_temp;
    reg [3:0] sum;
    reg carry;
//always block for doing the addition
    always @(a,b)
    begin
     sum_temp = a+b;
        if(sum_temp > 9)    begin
            sum_temp = sum_temp+6; //add 6, if result is more than 9.
            carry = 1;  //set the carry output
            sum = sum_temp[3:0];    end
        else    begin
            carry = 0;
            sum = sum_temp[3:0];
        end
    end
endmodule
```

2.Write a Test Bench Code for BCD Adder and Obtain RTL Simulation Waveform

Test Bench Code for BCD Adder

```
module bcd_tb;

// Inputs

            reg [3:0] a;
            reg [3:0] b;
            //reg carry_in;
            // Outputs
            wire [3:0] sum;
            wire carry;
            // Instantiate the Unit Under Test (UUT)
            bcd uut (
                        .a(a),
                        .b(b),
                        //.carry_in(carry_in),
                        .sum(sum),
                        .carry(carry));
            initial begin
                 $monitor("a=%d b=%d  sum=%b, carry = %b",a,b,sum,carry);
       a = 0;  b = 0;     #100;
        a = 6;  b = 9;    #100;
        a = 3;  b = 3;    #100;
        a = 4;  b = 5;    #100;
        a = 8;  b = 2;    #100;
        a = 9;  b = 9;   #100;
 end
end module
```

Simulation OutPut

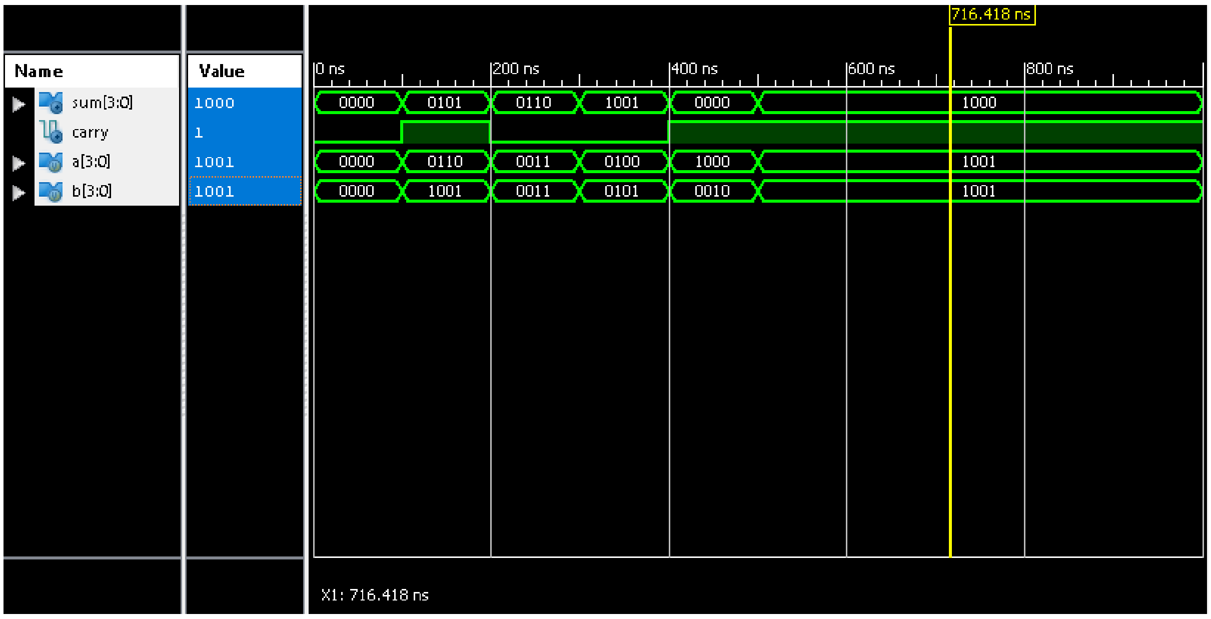


Fig 5.4 RTL Simulation Waveform

CONSOLE WINDOW OUTPUT

```
a= 0 b= 0 sum=0000, carry = 0
a= 6 b= 9 sum=0101, carry = 1
a= 3 b= 3 sum=0110, carry = 0
a= 4 b= 5 sum=1001, carry = 0
a= 8 b= 2 sum=0000, carry = 1
a= 9 b= 9 sum=1000, carry = 1
```

EXPERIMENT-6

Design Verilog Program to implement Different types of Multiplexer like 2:1,4:1,8:1

AIM: To design and implement Different types of Multiplexer like 2:1,4:1,8:1 using Verilog HDL.

SIMULATOR USED: ISE DESIGN SUIT 14.7

THEORY:

Multiplexer:

A multiplexer is a combinational circuit that has 2^n input lines and a single output line. Simply, the multiplexer is a multi-input and single-output combinational circuit. The binary information is received from the input lines and directed to the output line. On the basis of the values of the selection lines, one of these data inputs will be connected to the output.

Unlike encoder and decoder, there are n selection lines and 2^n input lines. So, there is a total of 2^N possible combinations of inputs. A multiplexer is also treated as Mux.

i) 2:1 Multiplexer:

In 2×1 multiplexer, there are only two inputs, i.e., a and b, 1 selection line, i.e., S and single outputs, i.e., Y. On the basis of the combination of inputs which are present at the selection line S, one of these 2 inputs will be connected to the output. The block diagram and the truth table of the 2×1 multiplexer are given below.

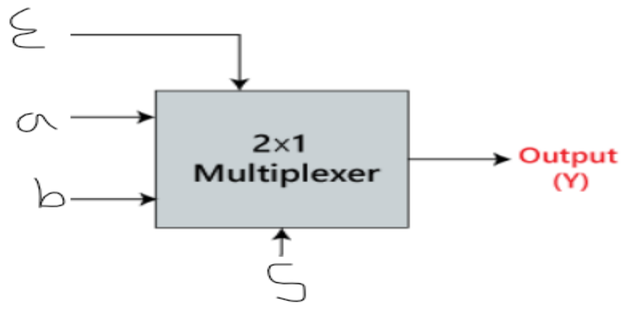


Fig 6.1 Block Diagram for 2:1 Mux

Truth Table for 2:1 MUX

Enable	Select Line	Inputs		Output
E	s	a	b	y
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

Procedure

1. Write a Verilog Code for 2:1 MUX

Verilog Code for 2:1 MUX

```
module mux2(a,b,s, y);
   input a,b,s;
   output reg y;
          always@(s,a,b)
begin
    case({s})
    1'b0:y=a;
    1'b1:y=b;
endcase
end
endmodule
```

2. Write a Test Bench code for 2:1 MUX and obtain Simulation Waveform

Test Bench Code for 2:1 MUX

```
module mux2_tb_v;
         // Inputs
         reg a;
         reg b;
         reg s;

         // Outputs
         wire y;

         // Instantiate the Unit Under Test (UUT)
         mux2 uut (
                  .a(a),
                  .b(b),
                  .s(s),
                  .y(y)
         );
initial begin
                  s=0;a=0;b=0;
                  #10;s=1;a=0;b=0;
     #10s=0;a=0;b=1;
#10s=1;a=0;b=1;
#10s=0;a=1;b=0;
#10s=1;a=1;b=0;
#10s=0;a=1;b=1;
#10s=1;a=1;b=1;
         end

endmodule
```

Simulation Output

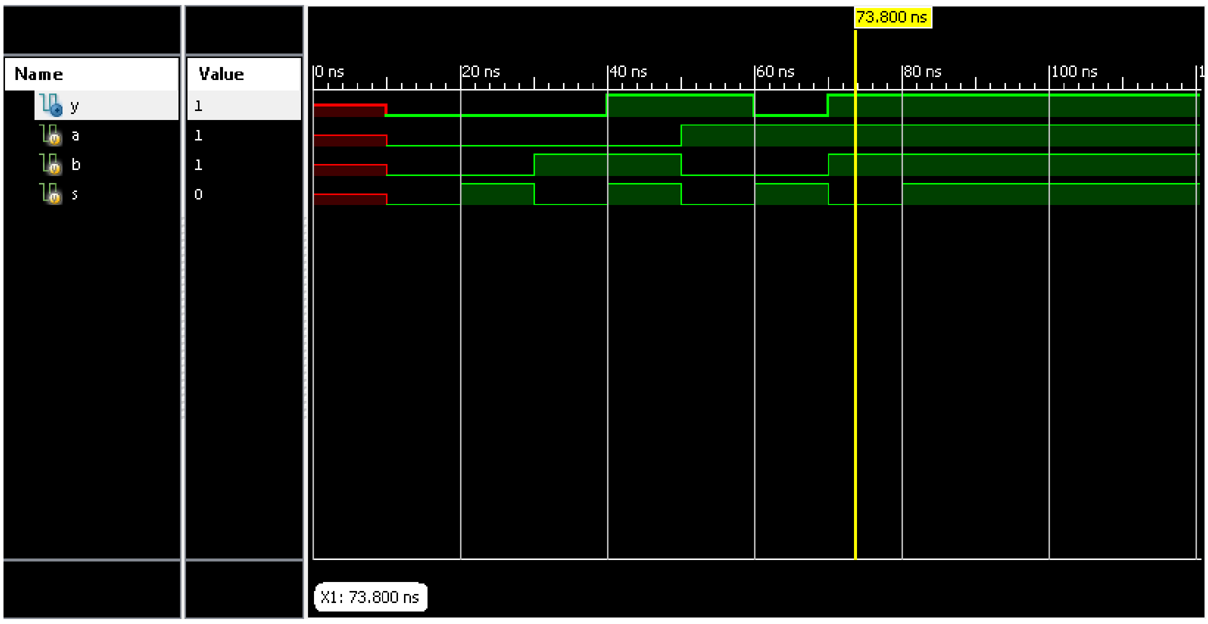


Fig 6.2 Simulation Waveform for 2:1 MUX

ii) 4:1 Multiplexer:

4x1 Multiplexer has four data inputs I3, I2, I1 & I0, two selection lines s1 & s0 and one output Y. The block diagram of 4x1 Multiplexer is shown in the following figure.

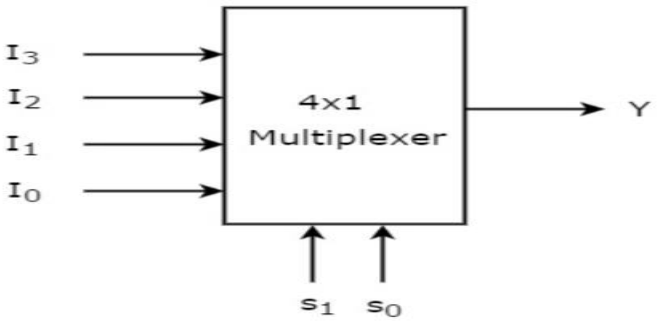


Fig 6.3 Block Diagram for 4:1 Mux

Truth Table for 4:1 MUX

Inputs		Output
s 1	s0	y
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Procedure

1. Write a Verilog Code for 4:1 MUX

Verilog Code for 4:1 MUX

```
module mux4(I, s, y);
    input [3:0] I;
    input [1:0] s;
    output reg y;
   always@(I,s)
            begin
               case(s)
                     2'b 00:y=I[0];
                     2'b 01:y=I[1];
                     2'b 10:y=I[2];
                     2'b 11:y=I[3];
            endcase
end

endmodule
```

2. Write a Test Bench code for 4:1 MUX and obtain Simulation Waveform

Test Bench Code for 4:1 MUX

```
module mux4_tb_v;
            // Inputs
            reg [3:0] I;
            reg [1:0] s;

            // Outputs
            wire y;

            // Instantiate the Unit Under Test (UUT)
            mux4 uut (
                        .I(I),
                        .s(s),
                        .y(y)
            );
            initial begin
                        // Initialize Inputs
                        I = 4'b 1100;
                        s = 2'b 00;
#100 s=2'b 01;
#100 s=2'b 10;
#100 s=2'b 11;
            end

endmodule
```

Simulation Output

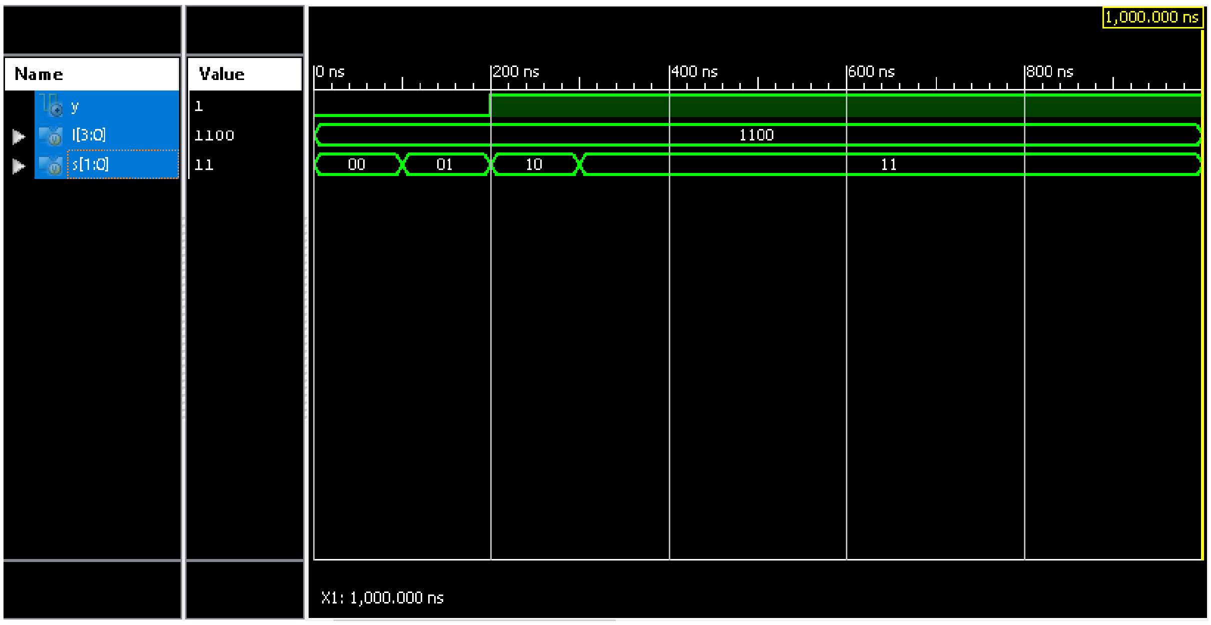


Fig 6.4 Simulation Waveform for 4:1 MUX

iii) 8:1 Multiplexer:

iv) An 8-to-1 multiplexer is a digital device that selects one of the eight inputs lines to the output line by using three-bit selection line. The block diagram of 8-to-1 Mux is shown in Figure 1. A 2n-to-1 multiplexer needs n bit selection line to select one of the 2n inputs to the output.

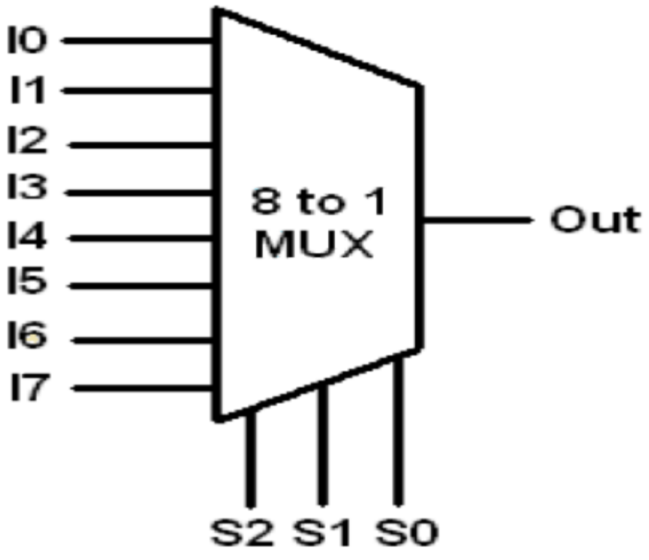


Fig 6.5 Block Diagram for 8:1 Mux

Truth Table for 8:1 MUX

Inputs			Output
s 2	s1	s0	y
0	0	0	I0
0	0	1	I1
0	1	0	I2
0	1	1	I3
1	0	0	I4
1	0	1	I5
1	1	0	I6
1	1	1	I7

Procedure

1. Write a Verilog Code for 8:1 MUX

Verilog Code for 8:1 MUX

```
module mux8(I, s, y);
    input [7:0] I;
    input [2:0] s;
    output reg y;    always@(I,s)
            begin
                case(s)
                         3'b 000:y=I[0];
                         3'b 001:y=I[1];
                         3'b 010:y=I[2];
                         3'b 011:y=I[3];
                          3'b 100:y=I[4];
                          3'b 101:y=I[5];
                          3'b 110:y=I[6];
                          3'b 111:y=I[7];
            endcase
end

endmodule
```

2. Write a Test Bench code for 8:1 MUX and obtain Simulation Waveform

Test Bench Code for 8:1 MUX

```
module mux8_tb_v;
            // Inputs
            reg [7:0] I;
            reg [2:0] s;
            // Outputs
            wire y;
            // Instantiate the Unit Under Test (UUT)
            mux8 uut (
                        .I(I),
                    .s(s),
                    .y(y)
          );
initial begin
I = 8'b 10101100;
      s = 3'b 000;
#100 s=3'b 001;
#100 s=3'b 010;
#100 s=3'b 011;
#100 s=3'b 100;
 #100 s=3'b 101;
 #100 s=3'b 110;
#100 s=3'b 111;

          end

endmodule
```

Simulation Output

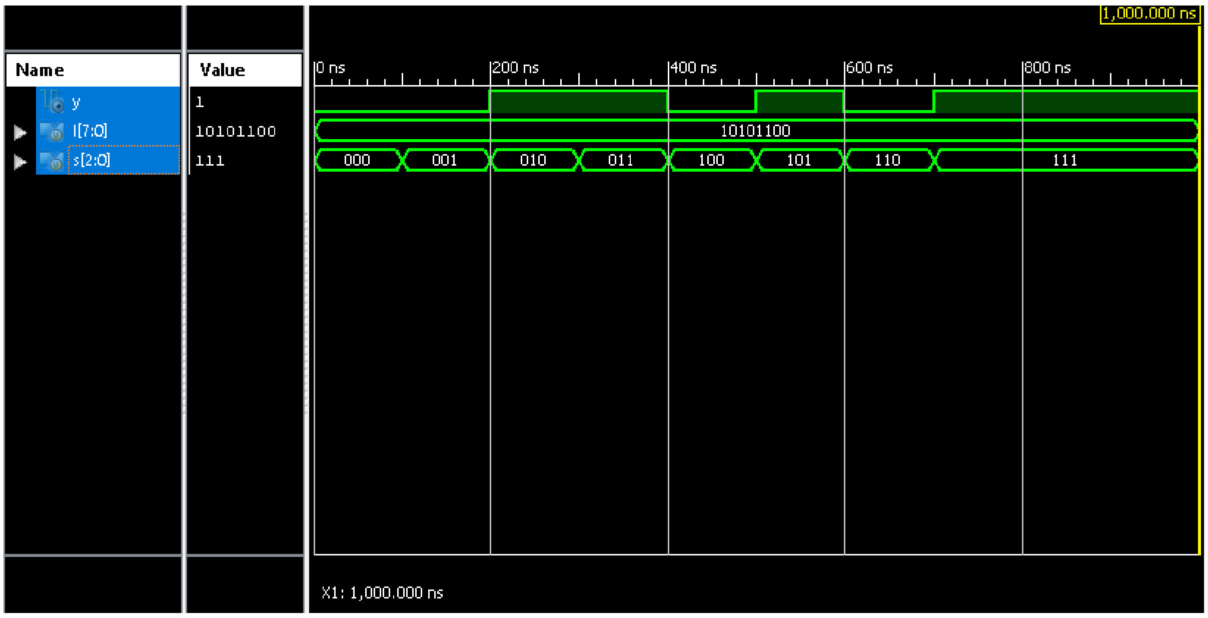


Fig 6.6 Simulation Waveform for 8:1 MUX

EXPERIMENT-7

Design Verilog Program to implement Different types of De-Multiplexer like 1:2,1:4,1:8

AIM: To design and implement Different types of De-Multiplexer like 1:2,1:4,1:8 using Verilog HDL.

SIMULATOR USED: ISE DESIGN SUIT 14.7

THEORY:

De-Multiplexer:

The demultiplexer is a combinational logic circuit designed to switch one common input line to one of several seperate output line. The data distributor, known more commonly as the demultiplexer or "Demux" for short. The demultiplexer takes one single input data line and then switches it to any one of a number of individual output lines one at a time.

i) 1:2 De-Multiplexer:

A 1-to-2 demultiplexer consists of one input line, two output lines and one select line. The signal on the select line helps to switch the input to one of the two outputs. The figure below shows the block diagram of a 1-to-2 demultiplexer with additional enable input.

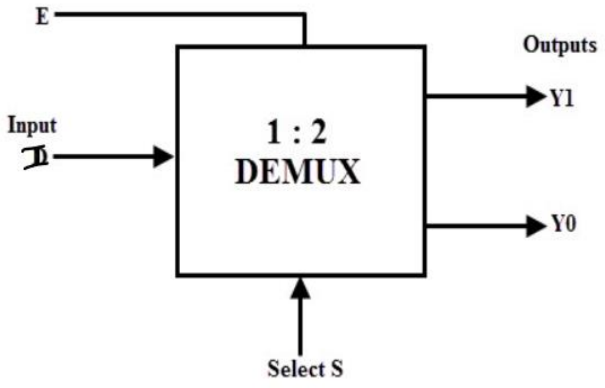


Fig 7.1 Block Diagram for 1:2 DMux

Truth Table for 1:2 DMUX

Enable	Input	Select Line	Output	
E	I	S	Y0	Y1
1	I	0	I	0
1	I	1	0	I

The logical expression of the term Y is as follows:

Y0=S'.I
Y1=S.I

Procedure

1. Write a Verilog Code for 1:2 DMUX

Verilog Code for 1:2 DMUX

```
module dmt(s,I, y0,y1);
   input s,I;
   output y0,y1;
   wire w1;
   not n1(w1,s);
          and a1(y0,w1,I);
          and a2(y1,s,I);
 endmodule
```

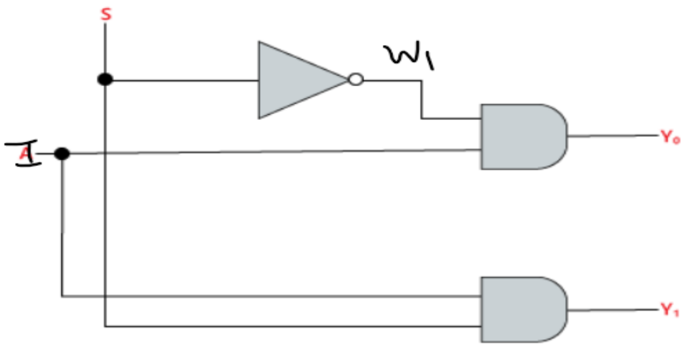


Fig 7.2 Logic Circuit for 1:2 DMUX

2. Write a Test Bench code for 1:2 DMUX and obtain Simulation Waveform

Test Bench Code for 1:2 DMUX

```
module dmt_tb_v;
        // Inputs
        reg s;
        reg I;
        // Outputs
        wire y0;
        wire y1;
        // Instantiate the Unit Under Test (UUT)
        dmt uut (
                .s(s),
                .I(I),
                .y0(y0),
                .y1(y1));
        initial begin
                I = 1;
                s=0;
                #100
                s=1;
        end
endmodule
```

Simulation Output

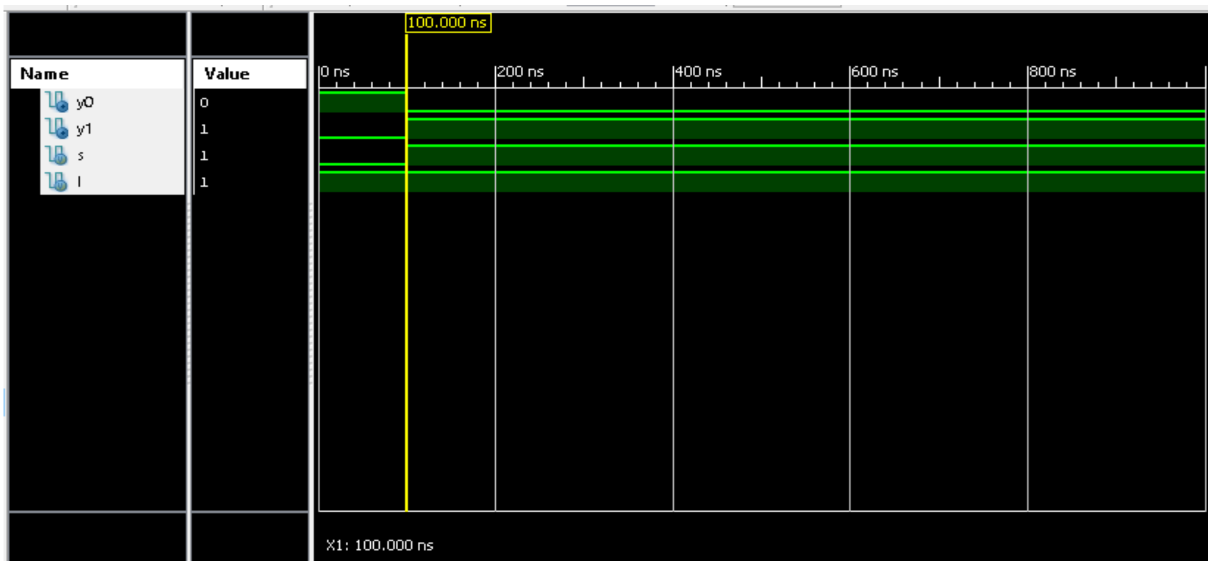


Fig 7.3 Simulation Waveform for 1:2 DMUX

ii) 1:4 De-Multiplexer:

In 1 to 4 De-multiplexer, there are total of four outputs, i.e., Y0, Y1, Y2, and Y3, 2 selection lines, i.e., S0 and S1 and single input, i.e., A. On the basis of the combination of inputs which are present at the selection lines S0 and S1, the input be connected to one of the outputs. The block diagram and the truth table of the 1×4 multiplexer are given below.

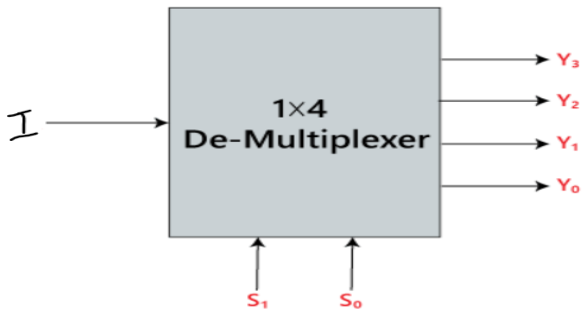


Fig 7.4 Block Diagram for 1:4 DMux

Truth Table for 1:4 DMUX

Input	Select Lines		Outputs			
	S0	S1	Y3	Y2	Y1	Y0
I	0	0	0	0	0	1
I	0	1	0	0	1	0
I	1	0	0	1	0	0
I	1	1	1	0	0	0

The logical expression of the term Y is as follows:

Y0=S1' S0' I
Y1=S0' S1 I
Y2=S0 S1' I
Y3=S0 S1 I

Procedure

1. Write a Verilog Code for 1:4 DMUX

Verilog Code for 1:4 DMUX

```
module dmf(s0,s1,I, y0,y1,y2,y3);
   input s0,s1,I;
   output y0,y1,y2,y3;     wire w1,w2;
          not n1(w1,s0);
          not n2(w2,s1);
          and a1(y0,w1,w2);
          and a2(y1,w1,s1);
          and a3(y2,w2,s0);
          and a4(y3,s0,s1);
endmodule
```

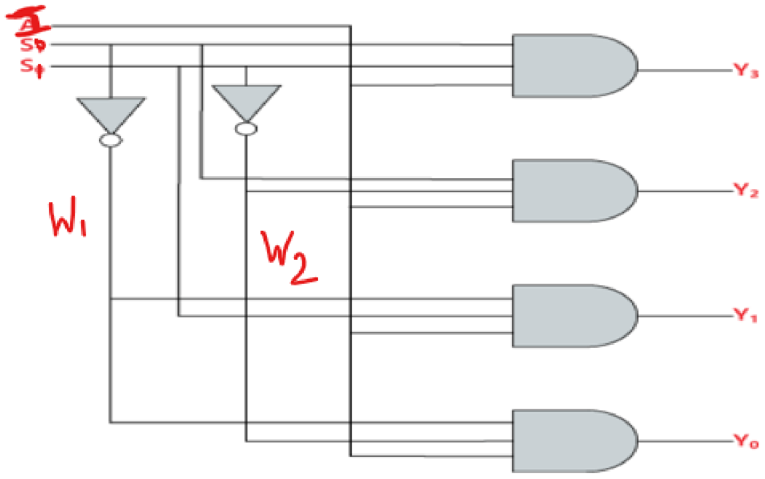


Fig 7.5 Logic Circuit for 1:4 DMUX

2. Write a Test Bench code for 1:4 DMUX and obtain Simulation Waveform

Test Bench Code for 1:4 DMUX

```
module dmf_tb_v;

        // Inputs
        reg s0;
        reg s1;
        reg I;

        // Outputs
        wire y0;
        wire y1;
        wire y2;
        wire y3;
        // Instantiate the Unit Under Test (UUT)
        dmf uut (
                .s0(s0),
                .s1(s1),
                .I(I),
                .y0(y0),
                .y1(y1),
                .y2(y2),
                .y3(y3)
        );
        initial begin

                I = 1;
                #10 s0=0;s1=0;
                #10 s0=0;s1=1;
                #10 s0=1;s1=0;
                 #10 s0=1;s1=1;
        end
    endmodule
```

Simulation Output

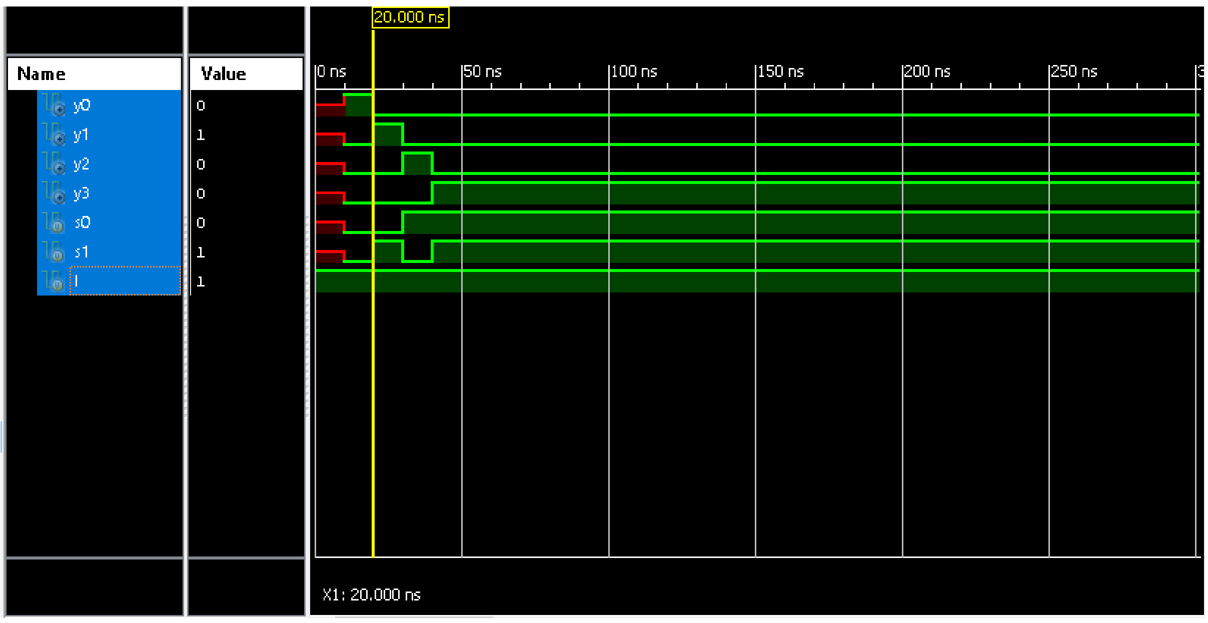


Fig 7.6 Simulation Waveform for 1:4 DMUX

iii) 1:8 De-Multiplexer:

In 1 to 8 De-multiplexer, there are total of eight outputs, i.e., Y0, Y1, Y2, Y3, Y4, Y5, Y6, and Y7, 3 selection lines, i.e., S0, S1and S2 and single input, i.e., A. On the basis of the combination of inputs which are present at the selection lines S0, S1 and S2, the input will be connected to one of these outputs. The block diagram and the truth table of the 1×8 de-multiplexer are given below..

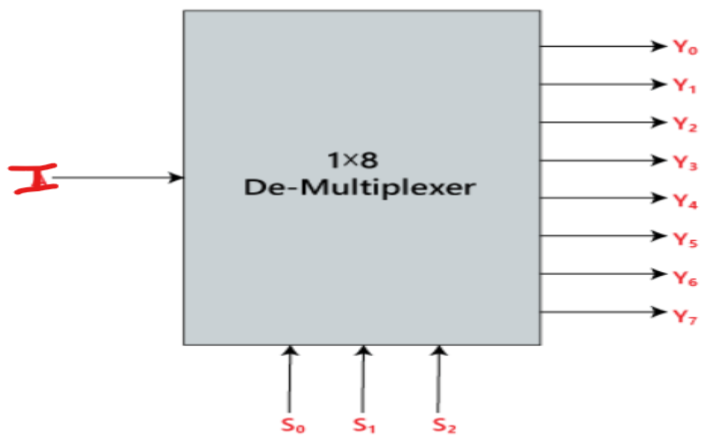


Fig 7.7 Block Diagram for 1:8 DMux

Truth Table for 1:8 DMUX

Input line	S0	S1	S2	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
I	0	0	0	0	0	0	0	0	0	0	1
I	0	0	1	0	0	0	0	0	0	1	0
I	0	1	0	0	0	0	0	0	1	0	0
I	0	1	1	0	0	0	0	1	0	0	0
I	1	0	0	0	0	0	1	0	0	0	0
I	1	0	1	0	0	1	0	0	0	0	0
I	1	1	0	0	1	0	0	0	0	0	0
I	1	1	1	1	0	0	0	0	0	0	0

Procedure

1. Write a Verilog Code for 1:8 DMUX

Verilog Code for 1:8 DMUX

```
module dme(s0,s1,s2,I, y0,y1,y2,y3,y4,y5,y6,y7);
   input s0,s1,s2,I;
   output y0,y1,y2,y3,y4,y5,y6,y7;     wire w0,w1,w2;
          not n1(w0,s0);
          not n2(w1,s1);
          not n3(w2,s2);
          and a0(y0,w0,w1,w2);
```

```
        and a1(y1,w0,w1,s2);
        and a2(y2,w0,s1,w2);
        and a3(y3,w0,s1,s2);
        and a4(y4,s0,w1,w2);
         and a5(y5,s0,w1,s2);
        and a6(y6,s0,s1,w2);
        and a7(y7,s0,s1,s2);
endmodule
```

2. Write a Test Bench code for 1:8 DMUX and obtain Simulation Waveform

The logical expression of the term Y is as follows:

Y_0=S0'.S1'.S2'.I
Y_1=S0'.S1'.S2.I
Y_2=S0'.S1.S2'.I
Y_3=S0'.S1.S2.I
Y_4=S0.S1'.S2'.I
Y_5=S0.S1'.S2.I
Y_6=S0.S1.S2'.I
Y_7=S0.S1.S2.I

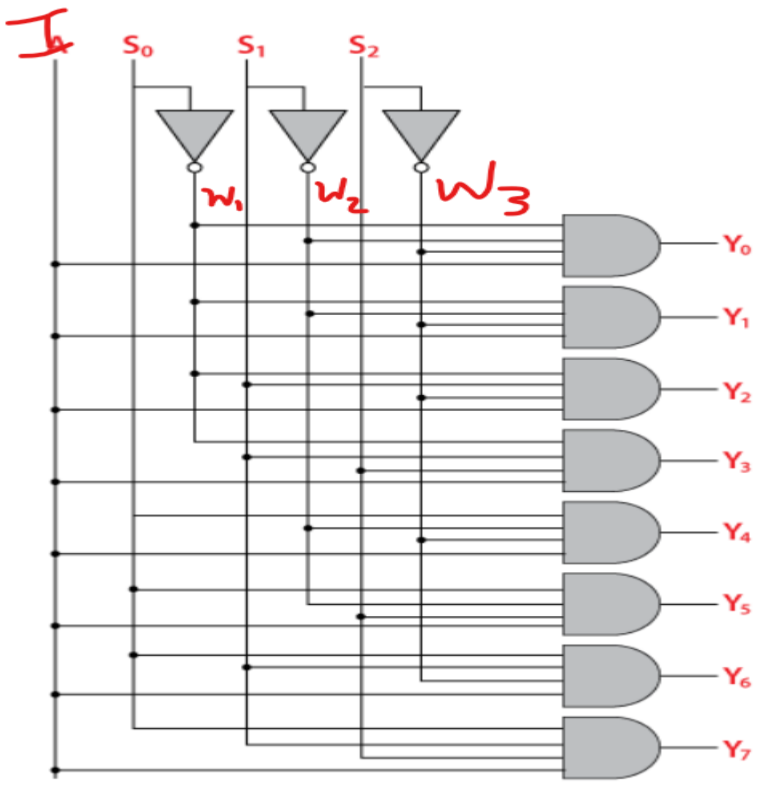


Fig 7.8 Logic Circuit for 1:8 DMUX

Test Bench Code for 1:8 DMUX

```
module dm8_tb_v;
        // Inputs
        reg s0;
        reg s1;
        reg s2;
        reg I;
        // Outputs
        wire y0;
        wire y1;
        wire y2;
        wire y3;
```

```
	wire y4;
	wire y5;
	wire y6;
	wire y7;
	// Instantiate the Unit Under Test (UUT)
	dme uut (
		.s0(s0),
		.s1(s1),
		.s2(s2),
		.I(I),
		.y0(y0),
		.y1(y1),
		.y2(y2),
		.y3(y3),
		.y4(y4),
		.y5(y5),
		.y6(y6),
		.y7(y7)
	);
	initial begin
		I = 1;
		#10 s0 = 0;s1 = 0;s2 = 0;
		#10 s0 = 0;s1 = 0;s2 = 1;
		#10 s0 = 0;s1 = 1;s2 = 0;
		#10 s0 = 0;s1 = 1;s2 = 1;
		#10 s0 = 1;s1 = 0;s2 = 0;
		#10 s0 = 1;s1 = 0;s2 = 1;
		#10 s0 = 1;s1 = 1;s2 = 0;
		#10 s0 = 1;s1 = 1;s2 = 1;
	end
endmodule
```

Simulation Output

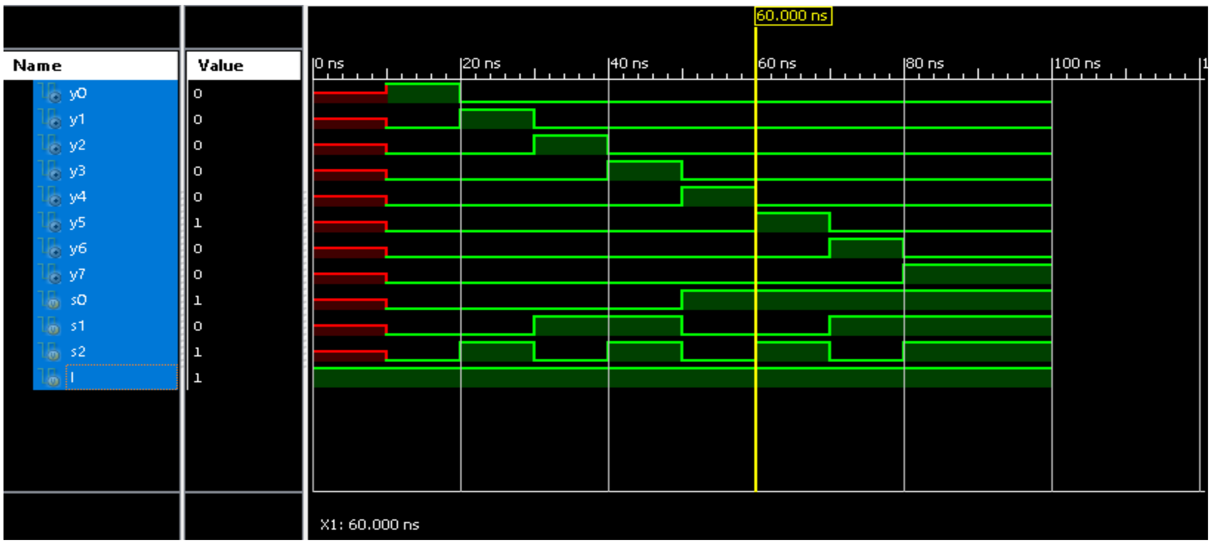


Fig 7.9 Simulation Waveform for 1:8 DMUX

EXPERIMENT-8

Design Verilog Program for implementing various types of Flip-flops like SR,JK & D

AIM: To design and implement various types of Flip-flop like SR,JK & D using Verilog HDL.

SIMULATOR USED: ISE DESIGN SUIT 14.7

THEORY:

A flip-flop is a fundamental building block in digital electronics and is a type of bistable multivibrator. It is a circuit that has two distinct output states and is capable of storing one bit of information. Flip-flops are widely used in sequential circuits, memory devices, and digital systems. There are several types of flip-flops, each with its own characteristics. Here are some common types:

i) S-R Flip Flop

This is the most common flip-flop among all. This simple flip-flop circuit has a set input (S) and a reset input (R). In this system, when you Set "S" as active, the output "Q" would be high, and "Q'" would be low. Once the outputs are established, the wiring of the circuit is maintained until "S" or "R" go high, or power is turned off.

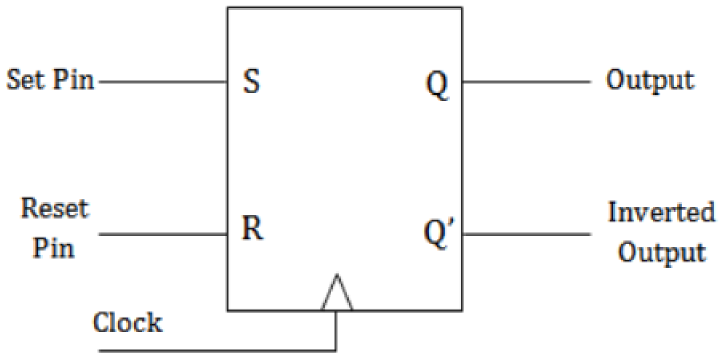


Fig 8.1 Symbol of S R Flip flop

Truth Table of S-R Flip flop

Clock	S	R	q	qb	Action
+ve edge	0	0	q	qb	No change
+ve edge	0	1	0	1	Reset
+ve edge	1	0	1	0	Set
+ve edge	1	1	X	X	Not Defined

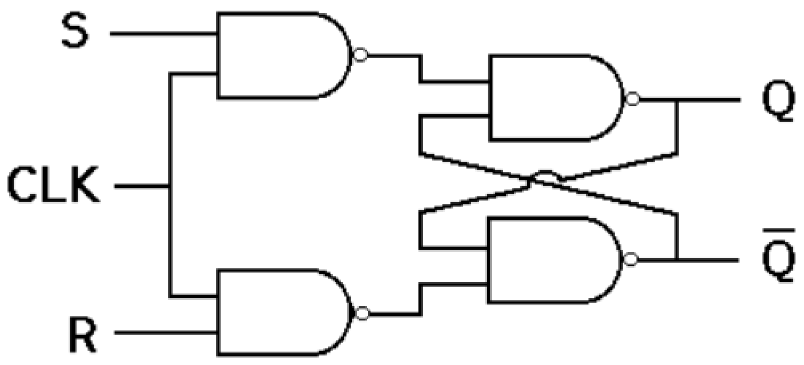


Fig 8.2 Logic circuit of S-R Flip Flop Using NAND Gates

Procedure

1. Write a Verilog code for S-R Flip flip.

Verilog Code for S-R Flip Flop

```
module srff(s,r,c, q,qb);
   input s,r,c;
   output reg q,qb;
          always @(posedge c)
          begin
            case({s,r})
                  2'b 00:q=q;
                  2'b 01:q=1'b 0;
                  2'b 10:q=1'b 1;
                  2'b 11:q=1'b x;
          endcase
          qb=~q;
          end

endmodule
```

2. Write the Test Bench code for SR Flip flop and obtain simulation waveform.

Test Bench Code for SR Flip Flop

```
module srff_tb_v;
        // Inputs
        reg s;
        reg r;
        reg c;
        // Outputs
        wire q;
        wire qb;
        // Instantiate the Unit Under Test (UUT)
        srff uut (
                .s(s),
                .r(r),
                .c(c),
                .q(q),
                .qb(qb)
        );
        initial begin
                s = 0;r = 0;c = 0;
                #100;s = 0;r = 1;
                 #100;s = 1;r = 0;
                #100;s = 1;r = 1;
                end
        always #10 c=~c;

endmodule
```

Simulation Output

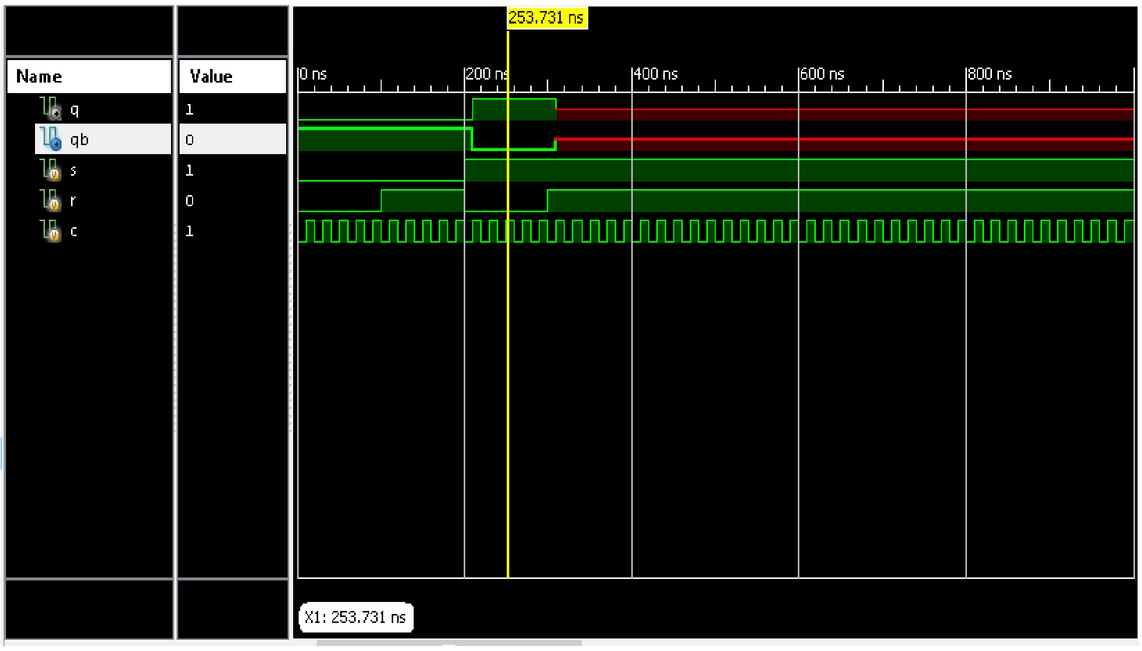


Fig 8.3 Simulation Waveform for SR Flip Flop

ii) J-K Flip Flop

Due to the undefined state in the SR flip-flops, another flip-flop is required in electronics. The JK flip-flop is an improvement on the SR flip-flop where S=R=1 is not a problem. The name JK flip-flop is termed from the inventor Jack Kilby from texas instruments

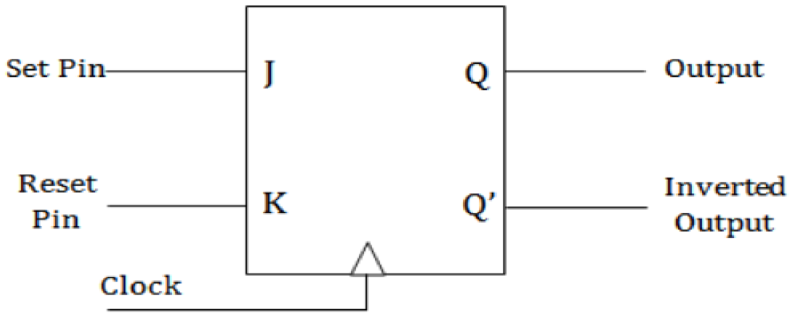


Fig 8.4 Symbol of JK Flip flop

Truth Table of J-K Flip flop

Clock	J	K	q	qb	Action
+ve edge	0	0	q	qb	No change
+ve edge	0	1	0	1	Reset
+ve edge	1	0	1	0	Set
+ve edge	1	1	qb	q	Toggle

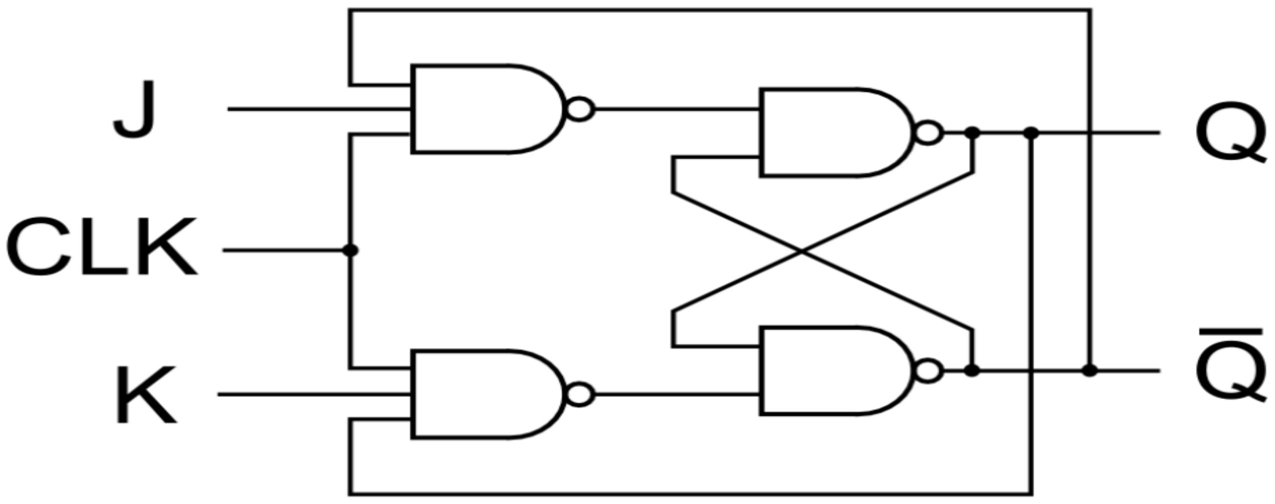


Fig 8.5 Logic circuit of J-K Flip Flop Using NAND Gates

Procedure

1. Write a Verilog code for J-K Flip flip.

Verilog Code for J-K Flip Flop

```
 module jkf(j,k,c, q,qb);
 input j,k,c;
 output reg q=0,qb=1;
          always@(posedge c)
         begin
           case({j,k})
                  2'b 00:q=q;
                  2'b 01:q=1'b 0;
                  2'b 10:q=1'b 1;
                  2'b 11:q=~q;
         endcase
         qb=~q;
         end

endmodule
```

2. Write the Test Bench code for JK Flip flop and obtain simulation waveform.

Test Bench Code for JK Flip Flop

```
module jkf_tb;
        // Inputs
        reg j;
        reg k;
        reg c;
        // Outputs
        wire q;
        wire qb;

        // Instantiate the Unit Under Test (UUT)
        jkf uut (
                .j(j),
                .k(k),
                .c(c),
                .q(q),
                .qb(qb)
        );
        initial begin
                j = 0;k = 0;c = 0;
               #100;j = 0;k = 1;
               #100;j = 1;k = 0;
                #100;j = 1;k = 1;
        end
        always #10 c=~c;
endmodule
```

Simulation Output

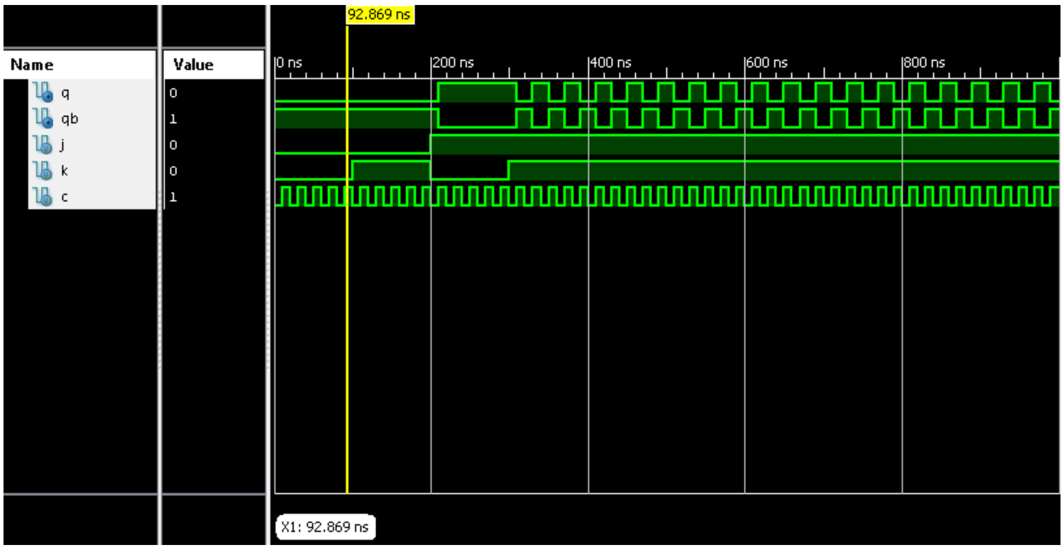


Fig 8.6 Simulation Waveform for JK Flip Flop

iii) D Flip Flop

D flip-flop is a better alternative that is very popular with digital electronics. They are commonly used for counters and shift registers and input synchronization. In the D flip-flops, the output can only be changed at the clock edge, and if the input changes at other times, the output will be unaffected.

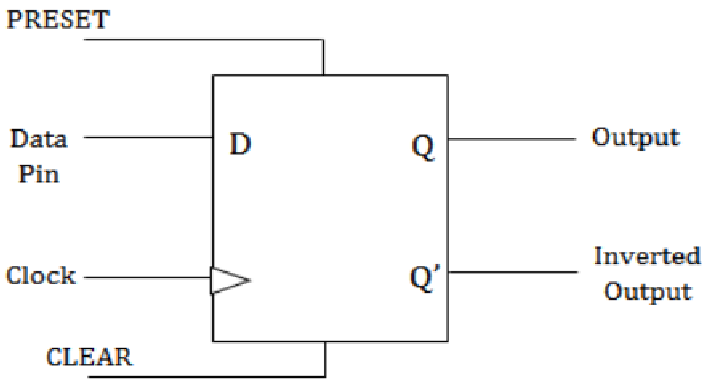


Fig 8.7 Symbol of D Flip flop

Truth Table of D Flip flop

Clock	D	q	qb	Action
+ve edge	0	0	1	Reset
+ve edge	1	1	0	Set

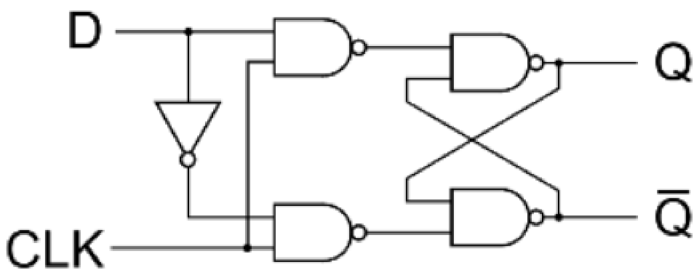


Fig 8.8 Logic circuit of D Flip Flop Using NAND Gates

Procedure

1. Write a Verilog code for D Flip flip.

Verilog Code for D Flip Flop

```
module dff(d,c, q,qb);
   input d,c;
   output reg q,qb;
           always@(posedge c)
          begin
             case({d})
                     1'b 0:q=0;
                     1'b 1:q=1'b 1;
   endcase
          qb=~q;
          end

endmodule
```

2. Write the Test Bench code for JK Flip flop and obtain simulation waveform.

Test Bench Code for D Flip Flop

```
module dff_tb_v;
         // Inputs
         reg d;
         reg c;
         // Outputs
         wire q;
         wire qb;
         // Instantiate the Unit Under Test (UUT)
         dff uut (
                  .d(d),
                  .c(c),
                  .q(q),
                  .qb(qb)
         );
         initial begin
                  d = 0;c = 0;
```

```
                #100;d=1;
        end
            always #10 c=~c;
endmodule
```

Simulation Output

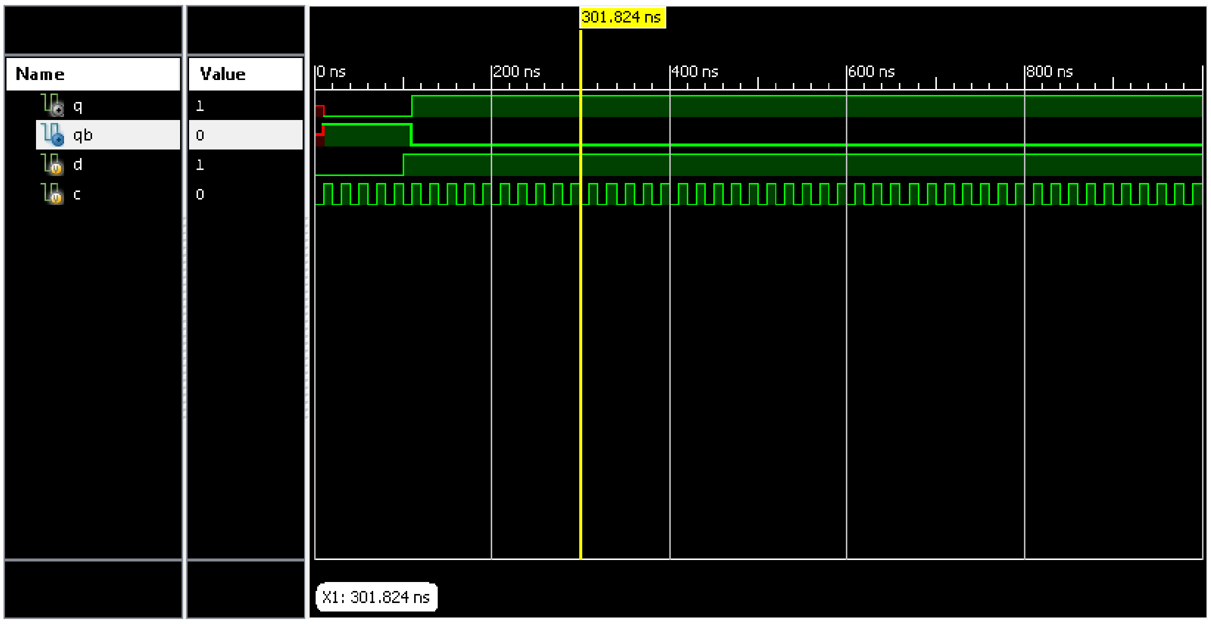


Fig 8.9 Simulation Waveform for JK Flip Flop

ADDITIONAL EXPERIMENTS

EXPERIMENT-10

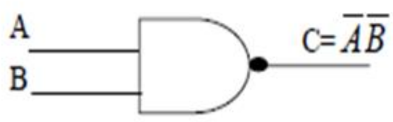
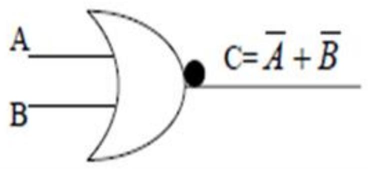
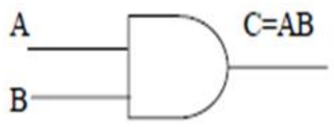
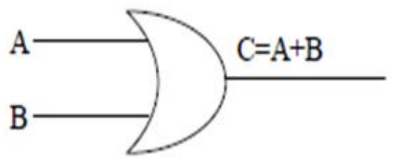
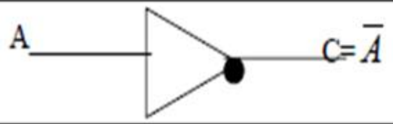
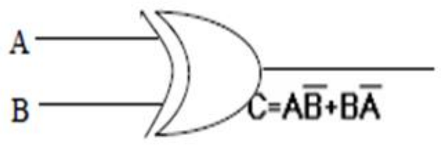
Design Verilog Program for implementing T-Flip-flop

AIM: To design and implement T-Flip Flop using Verilog HDL.

SIMULATOR USED: ISE DESIGN SUIT 14.7

THEORY:

TRUTH TABLE &SYMBOLS:

S.NO	GATE	SYMBOL	INPUTS		OUTPUT
			A	B	C
1.	NAND IC 7400		0	0	1
			0	1	1
			1	0	1
			1	1	0
2.	NOR IC 7402		0	0	1
			0	1	0
			1	0	0
			1	1	0
3.	AND IC 7408		0	0	0
			0	1	0
			1	0	0
			1	1	1
4.	OR IC 7432		0	0	0
			0	1	1
			1	0	1
			1	1	1
5.	NOT IC 7404		1	-	0
			0	-	1
6.	EX-OR IC 7486		0	0	0
			0	1	1
			1	0	1
			1	1	0

TRUTH TABLE:

a	b	and_out	or_out	not_out	nand_out	nor_out	xor_out	xnor_ou t
0	0	0	0	1	1	1	0	1
0	1	0	1	1	1	0	1	0
1	0	0	1	0	1	0	1	0
1	1	1	1	0	0	0	0	1

PROGRAM:

```
  module gates ( a, b,and_out, or_out, not_out, nand_out, nor_out, xor_out, xnor_out );
  input a,b;
  outputand_out, or_out, not_out, nand_out, nor_out, xor_out, xnor_out;
  assign and_out = a & b;
  assign  or_out=  a  | b,
  assign  not_out=~a ,
  assign nand_out= ~(a & b),
  assign nor_out= ~(a | b),
  assign xor_out= a ^ b;
  assign xnor_out = ~(a ^ b);
endmodule
```

PROCEDURE:

1. Write a Verilog code for all logic gates.
2. Write a Test Bench Code for all Logic Gates.

RESULT:

The functions of all the logic gates have been realized using Verilog HDL code.

EXPERIMENT-10

Design Verilog Program for implementing T-Flip-flop

AIM: To design and implement T-Flip Flop using Verilog HDL.

SIMULATOR USED: ISE DESIGN SUIT 14.7

THEORY:

i) T Flip Flop

A T flip-flop is like a JK flip-flop. These are basically single-input versions of JK flip-flops. This modified form of the JK is obtained by connecting inputs J and K together. It has only one input along with the clock input. The T flip-flop has a single input, T (toggle), and a clock input.

When the clock input transitions, the output toggles (flips) if the T input is 1; otherwise, it maintains its state.

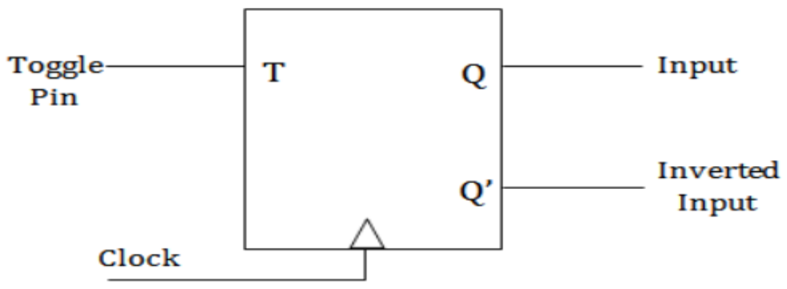


Fig 10.1 Symbol of D Flip flop

Truth Table of T Flip flop

Clock	T	q	qb	Action
+ve edge	0	0	1	No Change
+ve edge	1	1	0	Toggle

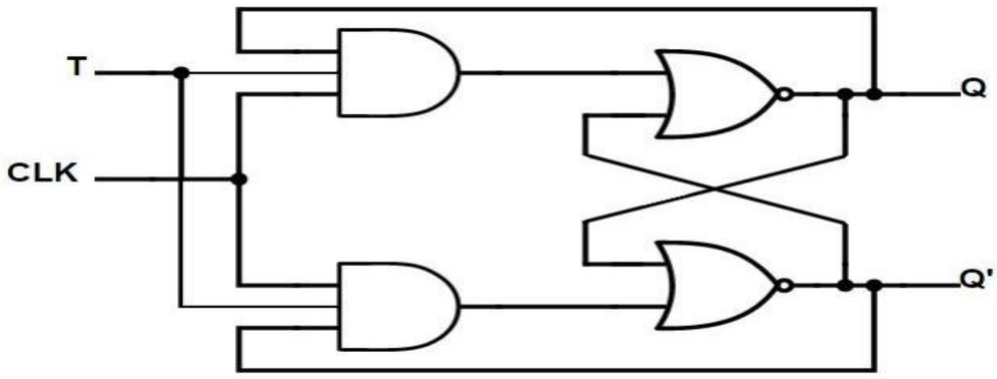


Fig 10.2 Logic circuit of T Flip Flop

Procedure

1. Write a Verilog code for T Flip flip.

Verilog Code for T Flip Flop

```
module tff(clk,t,q);
input clk,t;
output reg q;

always @ (posedge clk)
begin
   if(t == 0)
      q <= q;
   else
      q = ~q;
end

endmodule
```

2. Write the Test Bench code for T Flip flop and obtain simulation waveform.

Test Bench Code for T Flip Flop

```
module tf_tb_v;
        // Inputs
        reg clk;
        reg rst;
        reg t;

        // Outputs
        wire q;

        // Instantiate the Unit Under Test (UUT)
        tff uut (
                .clk(clk),
                .rst(rst),
                .t(t),
                .q(q)
        );
initial begin
clk = 1;
forever #10 clk=~clk;
end
initial begin
                #10 rst = 0;
                #10 rst=1;
                #10 t = 0;
                #10 t=1;
                #10 t=0;
                #10 t=1;
                #2000 $finish;
        end
```

```
endmodule
```

Simulation Output

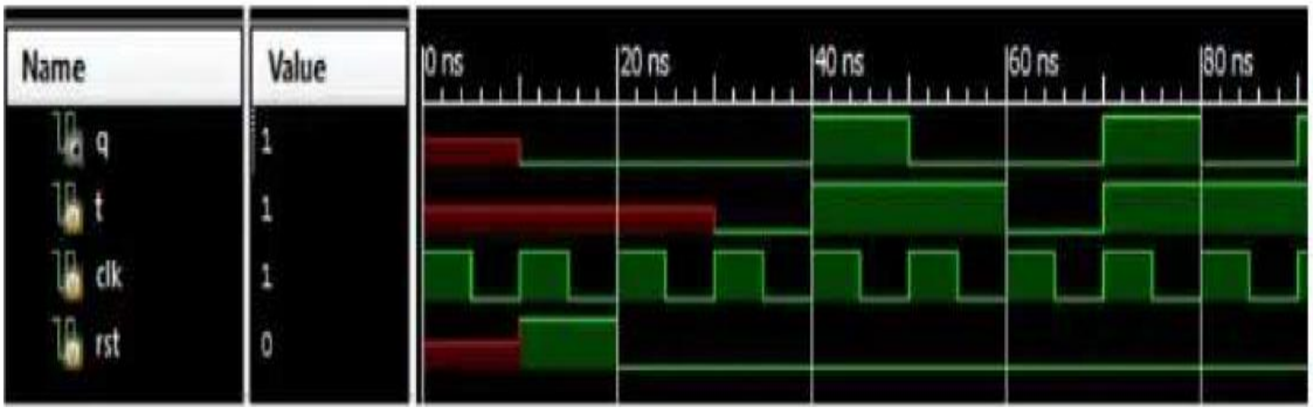


Fig 10.3 Simulation Waveform for T Flip Flop

VIVA QUESTION & ANSWERS

1. **What is HDL?**
   Hardware description language is a computer aided design (CAD). Tool to design and synthesis of digital system. HDL language is similar to language.
2. **HDL language is similar to which language?**
   It is similar to C Language.
3. **Justify the statement "Debugging the design is easy " in HDL.**
   Yes, because HDL packages implementation simulator & test benches
4. **List the Hardware Description Language?**
   VHDL & Verilog
5. **What is the abbreviation of VHDL?**
   VHDL means Very High Speed Integrated Circuit(VHSIC) hardware description language.
6. **What is the VHDL standard ?**
   The updated standard in 1993 is IEEE standard 1076-1993.
7. **Write the general structure of VHDL model?**
   Entityentity_name is
   port(define input and output port); endentity_name;
   architecturearchitecture_name of entity_name is begin
   statements; endarchitecture_name;
8. **Write the general structure for verilog?**
   Module modulename(input and output variable); input ;
   output;
   statement;
   endmodule;
9. **Which package is attached with VHDL program?**
   std_logic_1164 package is attached with VHDL program.
10. **What is the Verilog HDL standard and who is maintaining it?**
    IEEE standard 1364-1995 is the verilog HDL standard and it is maintain by Verilog international organization.
11. **In VHDL, what are the modes that the ports can take?**
    in, out, inout, buffer, linkage.
12. **Explain the function of the modes of the port.**
    in-The port is only an i/p and appears only on the right hand side of the statement. out-The port is only an o/p and appears only on the left and right hand side of the statement.
    inout-The port can be used as both an i/p & o/p.
    buffer-The port can be used as both i/p & o/p but should have only one source.
13. **Explain the structure of the verilog module.**
    The verilog module has two parts, Declaration & Body. Declaration- name, inputs and outputs of the module are listed. Body-shows the relationship between the inputs & outputs.
14. **Which of two Hardware Description Language is case Sensitive?**
    Verilog is case Sensitive.
15. **How should the module be terminated in verilog?**
    The module is terminated by the predefined word endmodule.
16. **What are the modes that exists in verilog ports?**
    input : The port is only an i/p Port.
    output : The port is only an o/p Port.
    inout : The port can be used as both an i/p and o/p.
17. **How are the operators broadly classified?**

Logical- AND,OR,XOR.
Relational =,=,<,<,>,> Arithmetic +,-,*,
Shift to move the bits of an objects in a certain direction right or left.

18. What are the modes that exists in verilog ports?
input : The port is only an i/p Port.
output : The port is only an o/p Port.
inout : The port can be used as both an i/p and o/p.

19. How are the operators broadly classified?
Logical- AND,OR,XOR.
Relational =,=,<,<,>,> Arithmetic +,-,*,
Shift to move the bits of an objects in a certain direction right or left.

20. State the different types of Logical Operators.
Logical Operators are AND, OR, NAND, NOR, NOT and Exclusive-OR.

21. Write the verilog bitwise logical operators.
AND - &, OR - |, NAND - ~(&), NOR - ~(|), EX-OR - ^, EX-NOR - ~ ^, NOT- ~.

22. What are Boolean Logical Operators, Give example.
The Boolean Logical operatorsoperate on 2 operands, the result is Boolean 0 or 1. Verilog Boolean logical operators are &&-AND Operator, ||-OR Operator.

23. What are Reduction Operators. Give examples.
The Reduction Opetators operate on a single Operand and the result is Boolean. Example of verilog Reduction Logical Operators are:
&-Reduction AND
| -Reduction OR
~&-Reduction NAND
~| -Reduction NOR
^ -Reduction XOR
~ ^ - Reduction XNOR
! -Negation.

24. What are relational Operators. Give example.
Relational Operators are used to compare the values of two objects and the result is Boolean 0 or 1, the VHDL relational operators are:
= Equality
/= Inequality
<Less than
<= Less than or equal
>Greater than
>= Greater than or equal.

25. What are Arithmetic Operators. State few arithmetic Operators in HDL? Arithmetic Operators performs various operators like.

VHDL Arithmetic Operators	Verilog Arithmetic Operators
+ Addition	+ Addition
- Subtraction	- Subtraction
*Multiplication	* Multiplication
/ Division	/ Division
Mod -Modulus	% Modulus
Rem -Remainder	
** Exponent	** Exponent

Absabsolute	
{,} Concatenate	&Concatenate

25. **What are shift & rotate operators?**
Shift Left represents multiplication by 2. Shift Right represents division by 2.
Example of VHDL shift operators.
ASll 1- Shift A one position left logical. ASrl 2- Shift A two position right logical. ASla 1 -Shift A one position left arithmetic. ASla 2 -Shift A two position right arithmetic. A rol 1-RotateA one position left.
A rol 1-Rotate A one position right. Verilog Shift Operators.
A<<1 Shift a one position left logical A>>2 Shift a two position right.
26. **What are the different types of VHDL data types?**
The VHDL data types are broadly classified into 5 types.
1)Scalar Type -Bit type Boolean
Integer Real Character Physical
User defined type Severity type
2)Composite Type-Bit vector type Array type
Record
3)Access Type
4)File Type
5)Other Types -Std_logic_type Std_logic_vector type Signed
Unsigned
27. **What are the different types of data types in Verilog?**
Nets, Registers, Vectors, Integers, Parameters, Real, Array.
28. **What are the different styles of writing the description?**
Behavioral, Structural, Switch level, Data flow, Mixed language.
29. **What is Behavioral Description?**
Behavioral description models the system as to how the o/p's behave with the i/p's. In behavioral description the architecture includes predefined word process in VHDL and always/initial Verilog module.
30. **What is Structural Description?**
Structural description model the system as component or gates.
Key word component is used in the architecture(VHDL) if gate construct. In Verilog and, or, not is included in the module.
31. **Explain what is Switch level description?**
The system is described using switches or transistors. The verilog key words nmos, pmos, cmos, transistors describes the system.
32. **Explain what is data flow?**
The data flow describes how the systems signal flows from the inputs to the outputs. The description is done by writing the Boolean functions of the outputs.
33. **What are the advantages of using mixed type description?**
Mixed type description use more than one type or style of the previously mentioned descriptions.
34. **What is the function of data flow descriptions?**
Data flow descriptions simulates the system by showing how the signa flows system inputs to outputs.
35. **How is signal assignment done in HDL?**
In VHDL the signal assignment operator <= is used & in verilog, the predefined word assign is used.
36. **How do you declare a constant in HDL?**

A constant is VHDL is declared by using the predefined word constant and in verilog it is declared by its type like time or integer.
To assign a value to a constant assignment operator or initial value assignment operator :=is used in VHDL & = in verilog.

37. **Write a time delay signed assignment statement.**
    To assign a delay time to a signed - assigned statement the predefined word after is used in VHDL & in Verilog it is # (Delay time).
    Ex : S1 - sel and b after 10 nsec - VHDL. assign # 10 S1=sel& b// Verilog.
38. **Define Vector data types.**
    A vector is a data type that declares an array of similar elements suchas to declare an object that has a width of more than 1 bit.
39. **What is the difference between syntax error and semantic error?**
    Syntax error is those that result from not following the rules of the language. it terminates compilation of the program.
    A semantic error is an error in the mechanics of the statement. it may not terminate the program, but the outcome of the program may not be as expected.
40. **What is the function of Behavioral Description?**
    The behavioral description describes the system by showing how the outputs behave according to changes in its inputs. In behavioral description one need not know the logic diagram of the system but one should know how the output behaves in response to change in the output.
41. **What are the two phase of execution in HDL?**
    The two phases of execution in HDL are Calculation and Assignment.
42. **What do you mean by Sequential Calculation?**
    Sequential calculation means the calculation of a statement will not wait until the proceeding statement is assigned only until the calculation is done.
43. **Which are the Sequential statements that are assigned with behavioral description?**
    If statement, else-if, loop statements, for loop, forever, report, repeat, next-exit.
44. **State the difference between signal and variable assignment.**
    A process is written based on signal assignment statements and then another process is written based on variable assignment statements. The difference can be observed by the simulation wave forms.
45. **When do use loop statements and what is its advantage?**
    Loops is used to repeat the execution of statements written inside the body. The number of repetitions is controlled by the range of an index parameter. The loop allows the code to be shortened**.**
46. **When is Structural Description best suited?**
    It is best suited when the digital logic of the system hardware components is known.
47. **What type of components are used in structural description?**
    Components can be gate level: AND, OR, NOT, XOR, XNOR gates. Components can be of higher logic level such as Register Transfer Level (RTL) or processor level.
48. **What type of statements are written in Structural Description and why?**
    Statements are “Concurrent “ in nature. At any simulation time, all statements that have an event are executed concurrently.
49. **Difference between VHDL & Verilog structural description.**
    Verilog recognizes all the primitive gates such as AND, OR, NOT, XOR, XNOR. Basic VHDL packages do not recognize any gates unless the package is limited to one more libraries, packages that have gate description