Visvesvaraya Technological University

"Jnana Sangama", Belagavi – 590018

Third Semester B.E.

[As per Choice Based Credit System (CBCS) Scheme]

(For Internal Circulation Only)

"Digital Design and Computer Organization (BCS302)"

IPCC Laboratory Manual

(For Reference Only)

Name	
USN	
Section	
Lab Batch	
Day/Time	

Kalpataru Institute of Technology, Tiptur -572 201

Department of Artificial Intelligence and Machine Learning.

AY: 2024-2025

VISVESVARAYA TECHNOLOGICAL UNIVERSITY, BELAGAVI

Integrated Professional Core Course (IPCC)

DIGITAL DESIGN AND COMPUTER ORGANIZATION

LAB MANUAL

BCS302

III Semester

KALPATARU INSTITUTE OF TECHNOLOGY, TIPTUR

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND

MACHINE LEARNING

2024-25

KALPATARU INSTITUTE OF TECHNOLOGY

(Accredited by NAAC with B+, Approved by A.I.C.T.E. New Delhi, Recognized by Govt. of Karnataka & Affiliated to V.T U., Belagavi)

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

Vision and Mission of the Institution

Vision

"To bring forth technical graduates of high caliber with a strong character and to uphold the spiritual and cultural values of our country."

Mission

"To impart quality technical and managerial education at graduate and post graduate levels through our dedicated and well qualified faculty."

Vision and Mission of the AI & ML Department

Vision

"To create a community of AI and ML specialists distinguished by their technical excellence and moral principles, who champion responsible innovation and honor our country's spiritual and cultural traditions, advancing technology for the betterment of society."

Mission

M1: "To advance AI and ML through ground-breaking development, encouraging students and faculty to pursue innovative solutions that address global challenges and drive societal progress."
M2: "To provide a supportive and inclusive environment that nurtures the intellectual and personal growth of our students, preparing them to become leaders in AI&ML."
M3: "To integrate the nation's spiritual and cultural values into the AI and ML curriculum, fostering respect for our heritage and guiding students to consider the cultural impacts of their innovations."

KALPATARU INSTITUTE OF TECHNOLOGY

(Accredited by NBA, Approved by A.I.C.T.E. New Delhi, Recognized by Govt. of Karnataka & Affiliated to V.T U., Belagavi)

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

Program Outcomes	
a.	**Engineering Knowledge:** Apply knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.
b.	**Problem Analysis: Identify,** formulate, research literature and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences and engineering sciences
c.	**Design/ Development of Solutions**: Design solutions for complex engineeringproblems and design system components or processes that meet specified needs with appropriate consideration for public health and safety, cultural, societal and environmental considerations.
d.	**Conduct investigations of complex problems** using research-based knowledge and research methods including design of experiments, analysis and interpretation of data and synthesis of information to provide valid conclusions.
e.	**Modern Tool Usage**: Create, select and apply appropriate techniques, resources andmodern engineering and IT tools including prediction and modeling to Complex engineering activities with an under- standing of the limitations.
f.	**The Engineer and Society**: Apply reasoning informed by contextual knowledge to assess societal, health, safety, legal and cultural issues and theConsequent responsibilities relevant to professional engineering practice.
g.	**Environment and Sustainability**: Understand the impact of professional Engineering solutions in societal and environmental contexts and demonstrate knowledge of and need for sustainable development.
h.	**Ethics**: Apply ethical principles and commit to professional ethics and Responsibilities and norms of engineering practice.
i.	**Individual and Team Work:** Function effectively as an individual, and as amember or leader in diverse teams and in multi disciplinary settings.
j.	**Communication:** Communicate effectively on complex engineering activities withthe 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**.**
k.	**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.
l.	**Project Management and Finance:** Demonstrate knowledge and understanding of 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.
Program Specific Outcomes	
m.	**PSO1**: The ability to comprehend, analyse, and apply knowledge of human cognition, Artificial Intelligence, Machine Learning, and data engineering to real-world problems, addressing future challenges.
n.	**PSO2**: The ability to cultivate computational knowledge and project development skills, utilizing innovative tools and techniques to address problems in Deep Learning, Machine Learning, and Artificial Intelligence.

Course objectives:

- To demonstrate the functionalities of binary logic system
- To explain the working of combinational and sequential logic system
- To realize the basic structure of computer system
- To illustrate the working of I/O operations and processing unit

Course outcomes (Course Skill Set):

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

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

CO2: Design different types of combinational and sequential circuits along with Verilog programs.

CO3: Describe the fundamentals of machine instructions, addressing modes and Processor performance. CO4: Explain the approaches involved in achieving communication between processor and I/O devices.

CO5: Analyze internal Organization of Memory and Impact of cache/Pipelining on Processor Performance.

CONTENTS

I. Verilog Introduction

- **Verilog** is a language that describes how electronic hardware (like circuits and chips) should work. Think of it as a blueprint for hardware.
- Once you write your hardware design in Verilog, **synthesis tools** can convert it into actual hardware designs.
- Verilog is easier to use compared to other HDLs like VHDL because it's based on the programming languages **C** and **Ada**.
- The latest version of Verilog follows the **IEEE Standard 1364-2005**.

Abstraction Levels in Verilog:

Verilog allows you to design hardware at different levels of detail, known as **abstraction levels**. There are four of these levels:

1. **Gate Level**: At this level, you design using logic gates (like AND, OR, NOT) and their connections. This is the most basic level, where you think in terms of individual components and how they are wired together.
2. **Dataflow Level**: Here, the focus is on how data moves through the system. You describe how data flows between different parts of the design, rather than worrying about the actual gates used.
3. **Switch Level**: This is the lowest level, where you deal with switches and storage nodes. It requires deep knowledge of how the hardware works at a very granular level (like understanding the behaviour of transistors)
4. **Behavioural Level**: At this level, you write code similar to traditional programming (like C) without focusing on the hardware details. This is where you describe what the system should do, rather than how it should do it

Steps to Use Xilinx ISE Design Suite 14.2:

1. **Install and Launch Xilinx ISE Design Suite 14.2:**
    - If you don't already have the software, download it from the Xilinx website.
    - Install the software on your computer, and once installed, open the Xilinx ISE Design Suite.
2. **Create a New Project:**
    - Go to the **File** menu and select **New Project**.
    - A wizard will guide you through the steps to create your project.
    - Give your project a **name** and specify where you want to **save it**.
    - Choose whether your project is based on **HDL (Hardware Description Language)** or **schematic design**.
    - If you're using Verilog, select **Verilog** as your source type.
3. **Add Design Files:**
    - The wizard will prompt you to add the necessary files for your project.
    - Click **Next** and upload your **Verilog** source code files.
    - If your design requires it, you can also add **constraint files (XDC files)**, which help control the timing and physical layout of your design.
4. **Set the Top Module:**
    - The top module is the main part of your design that you want to work on.
    - If your project has multiple modules, choose the one that you want to synthesize and program onto the FPGA.
5. **Specify Project Settings:**
    - Lastly, you can configure additional settings for your project, such as:
    - **Target device:** The specific FPGA model you are using.

- **Simulator:** The tool you'll use to test your design.
- Other **properties** that affect how your project is compiled and run.

6. **Create a Testbench:**
   - In your project hierarchy (the list of files/modules in your project), right-click on your Verilog module and choose **"New Source."**
   - Select **"HDL"** and then choose either **"VHDL Testbench"** or **"Verilog Testbench"**, depending on the language you're using.
   - A wizard will guide you through the creation of the testbench file.
7. **Edit Testbench Code:**
   - Open the testbench file you just created.
   - In this file, write code that creates inputs (stimulus) for your design and monitors the outputs.
   - The testbench should activate your Verilog module, apply input signals, and observe the results to verify if your design behaves as expected.
8. **Simulate the Design:**
   - In Xilinx ISE, use the built-in simulator called **"Isim"** to run your simulations.
   - Go to the **hierarchy panel** on the left side and click on **"Simulate."**
   - Right-click and choose either **"Run Behavioural Simulation"** (for testing functionality) or **"Run Post-Implementation Simulation"** (for testing after synthesis).
9. **View Simulation Results:**
   - The simulation tool will show the results in a **waveform viewer**.
   - These waveforms represent the signals in your design over time. Analyse them to ensure your design functions correctly.

1. Given a 4-variable logic expression, simplify it using appropriate technique and simulate the same using basic gates.

E=A'B'CD'+A'BCD'+ABCD'+AB'CD'+AB'C'D'+AB'C'D+AB'CD

1.1 AIM: To simplify the given 4-variable logic expression using k-map and simulate using basic gates.

SIMULATOR USED: ISE Design Suite 14.2

1.2 THEORY:

A Karnaugh Map (K-map) is a graphical representation of a truth table, which is a systematic way to simplify and optimize Boolean algebra expressions used in digital circuit design. K-maps are particularly helpful for minimizing the number of logic gates in a given logic function.

Basic Concepts:

Truth Table: A truth table is a tabular representation of all possible input combinations and their corresponding output values for a logic function. It helps define the behavior of the function.

Boolean Algebra: Boolean algebra is a mathematical system used to manipulate binary variables and perform logical operations such as AND, OR, and NOT. It is essential for simplifying and optimizing logic functions.

Karnaugh Map Overview:

Grid Structure: A Karnaugh Map is a grid where each cell represents a unique combination of input variables. The size of the grid depends on the number of variables in the logic function.

Binary Values: Inside each cell, you write the binary value (0 or 1) that corresponds to the output of the logic function for that specific input combination.

Simplification Process:

Grouping Cells: The primary goal of a K-map is to identify groups of adjacent cells containing 1s. These groups are called "minterms" or "maxterms," and they represent patterns of input values where the output is 1. The larger the group, the more simplification can be achieved.

Implicants: An "implicant" is a group of adjacent cells that can be used to represent a part of the logic function. Implicants can be as simple as individual cells or as complex as a group covering multiple cells.

Prime Implicants: Prime implicants are the most significant groups that cannot be further combined with other groups. They are essential for minimizing the logic expression.

Minimization: The K-map simplification process involves covering all 1s in the truth table using the fewest and largest groups of cells. This minimizes the number of terms in the logic expression.

Rules for Grouping:

Groups Must Be Powers of 2: Groups must contain 1, 2, 4, 8, or more cells, as powers of 2. They should be rectangular or square, and adjacent cells must be covered.

Groups Can "Wrap Around": Groups can "wrap around" the edges of the K-map, allowing cells at opposite sides of the grid to be part of the same group.

Using the Simplified Expression:

Applying the Simplified Expression: Once the logic function is simplified, you can use the simplified expression to design a digital circuit using logic gates such as AND, OR, and NOT gates.

In summary, a Karnaugh Map is a graphical method for simplifying and optimizing Boolean algebra expressions used in digital logic design. By identifying groups of 1s and minimizing the logic function, K-maps help reduce the number of logic gates needed to implement the function efficiently.

1.3 PROCEDURE:

Step 1: Simplify the expression using techniques like Boolean algebra and Karnaugh maps (K-maps) to minimize the number of gates required. In this case, I used a Karnaugh map to map the values from the truth table for illustration:

Table 1.1 Truth Table for

E=A'B'CD'+A'BCD'+ABCD'+AB'CD'+AB'C'D'+AB'C'D+AB'CD

Decimal Number	A	B	C	D	Output E
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	1
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	0
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	1
12	1	1	0	0	0
13	1	1	0	1	0
14	1	1	1	0	1
15	1	1	1	1	0

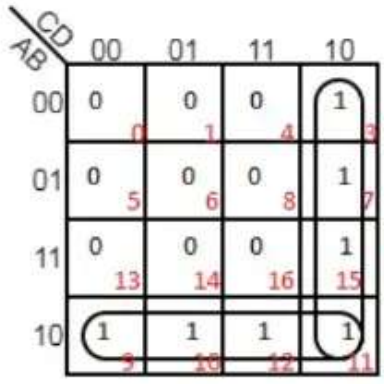


Figure 1.1 K-Map for E=A'B'CD'+A'BCD'+ABCD'+AB'CD'+AB'C'D'+AB'C'D+AB'CD

Group the 1s in the K-map to find the simplified expression:

$$F(A, B, C, D) = CD'+AB'$$

To implement this simplified expression using basic gates (AND, OR, NOT), you can use the following gates:

CD': Use an AND gate with inputs C and D'.

AB': Use an AND gate with inputs A and B'.

The final result is obtained by using an OR gate to combine the outputs of the above AND gates.

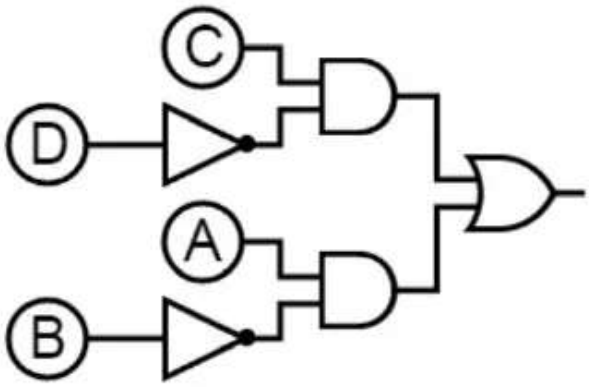


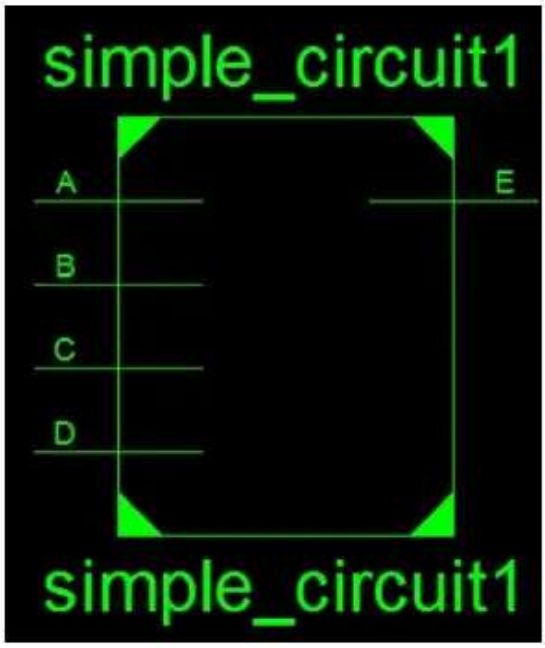
Figure 1.2 Logic Circuit for

E=A'B'CD'+A'BCD'+ABCD'+AB'CD'+AB'C'D'+AB'C'D+AB'CD

Step 2: Create the Verilog code for this implementation and View the logic gates and their interconnections in your design using the RTL schematic

```
module simple_circuit1( A,B,C,D,E);
    output E;
    input A,B,C,D;
    wire w1,w2,w3,w4;
    not n1(w1,D);
    not n2(w2,B);
    and a1(w3,C,w1);
    and a2(w4,A,w2);
    or o1(E,w3,w4);
endmodule
```

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



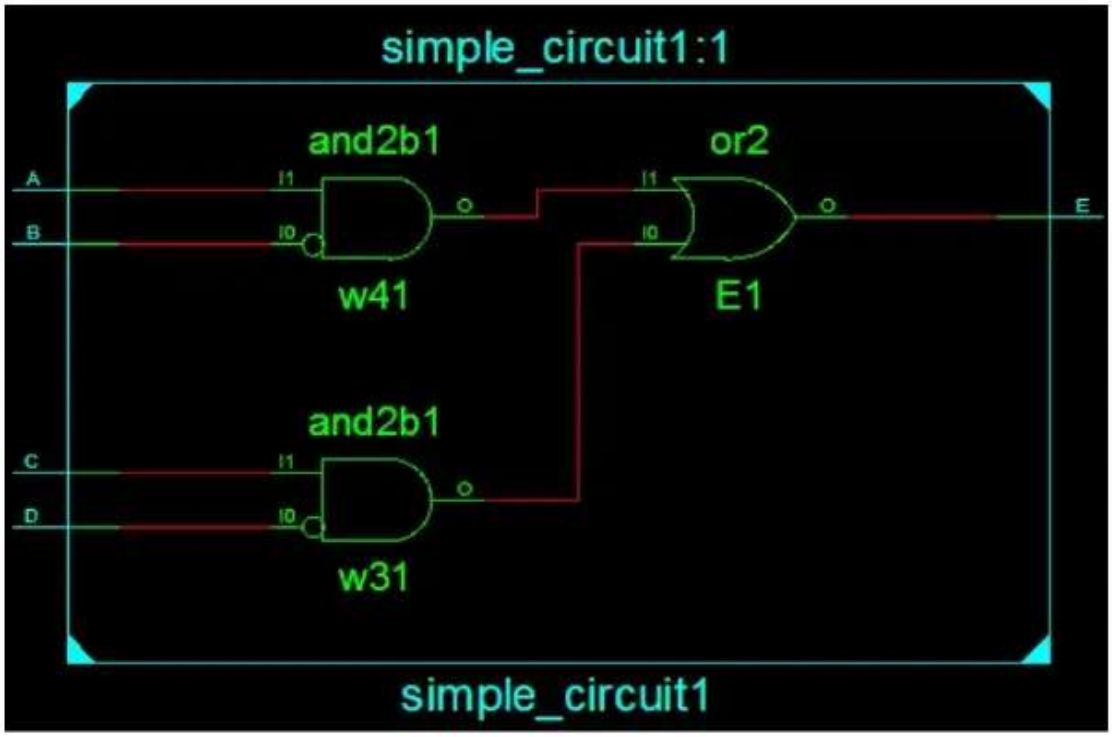


Figure 1.3 RTL schematic for CD'+AB'

Step 3: Create a testbench to verify the functionality of the Verilog design and obtain the corresponding RTL simulation waveform. A testbench includes inputs, a clock signal (if applicable), and monitors the output to compare it with expected results.

TestBench Code:

```
module test_simplecircuit;
        reg A;
        reg B;
        reg C;
        reg D;
        wire E;
        simple_circuit1 uut (
                .A(A),
                .B(B),
                .C(C),
                .D(D),
                .E(E)
        );
        initial begin
                $monitor("A = %b B = %b C = %b D = %b E = %b", A,B,C,D,E);
                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
    initial #1000 $finish;
endmodule
```

This testbench provides values for inputs A, B, C, and D, monitors the output E, and displays the result.

After the simulation completes, you'll obtain simulation results. These results will typically include waveforms and log messages. The primary focus is on the behavior of the output signal (E) under various input conditions. Analyze the results to ensure your design is functioning as expected.

In your simulation output, you should observe the value of E as it changes over time for the provided input conditions. Verify that it matches your expected results based on the logic expression and the Verilog code.

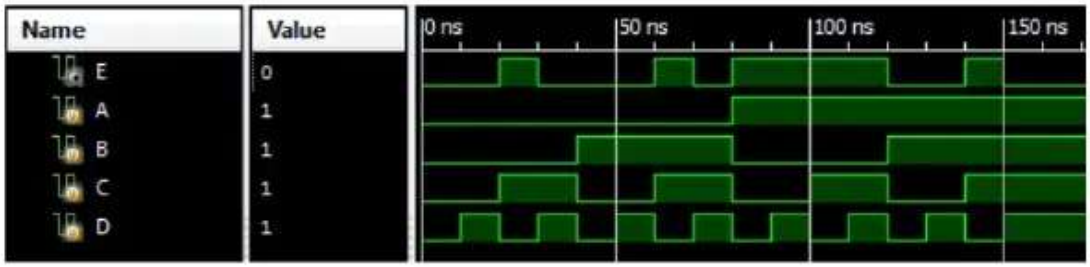


Figure 1.4 RTL simulation waveform for CD'+AB'

Step 4: Get the Output from the console/output window at the time of simulation.

A = 0 B = 0 C = 0 D = 0 E = 0

A = 0 B = 0 C = 0 D = 1 E = 0

A = 0 B = 0 C = 1 D = 0 E = 1

A = 0 B = 0 C = 1 D = 1 E = 0

A = 0 B = 1 C = 0 D = 0 E = 0

A = 0 B = 1 C = 0 D = 1 E = 0

A = 0 B = 1 C = 1 D = 0 E = 1

A = 0 B = 1 C = 1 D = 1 E = 0

A = 1 B = 0 C = 0 D = 0 E = 1

A = 1 B = 0 C = 0 D = 1 E = 1

A = 1 B = 0 C = 1 D = 0 E = 1

A = 1 B = 0 C = 1 D = 1 E = 1

A = 1 B = 1 C = 0 D = 0 E = 0

A = 1 B = 1 C = 0 D = 1 E = 0

A = 1 B = 1 C = 1 D = 0 E = 1

A = 1 B = 1 C = 1 D = 1 E = 0

1.4 APPLICATIONS:

1. **Digital Circuit Design:** K-maps are widely used in digital circuit design to simplify and optimize Boolean expressions, reducing the complexity of circuits and improving efficiency.
2. **Troubleshooting and Debugging**: K-maps can be used to analyze and troubleshoot complex digital systems to understand their behavior.

1.5 EXPECTED VIVA QUESTIONS:

1. Can you explain the basic steps involved in simplifying a logic expression?
2. What are the commonly used techniques for simplifying logic expressions?
3. Given the expression E = A'B'CD' + A'BCD' + ABCD' + AB'CD' + AB'C'D' + AB'C'D + AB'CD, how would you go about simplifying it?
4. Can you apply the Karnaugh map method to simplify the given expression?
5. After simplifying the expression, what is the simplified Boolean function?
6. Which gates will you use to implement the simplified expression using basic logic gates?
7. How do you simulate a logic circuit using basic gates on a digital logic simulator or hardware?
8. What is the advantage of simplifying a logic expression before implementing it with basic gates?
9. Can you provide a step-by-step demonstration of implementing the simplified expression using basic gates on a digital logic simulator?
10. Are there any best practices or considerations to keep in mind when implementing logic expressions with basic gates to ensure efficient and error-free circuit design?

2. Design a 4 bit full adder and subtractor and simulate the same using basic gates.

2.1 AIM: To design a 4-bit full adder and subtractor using basic logic gates and simulate their operations

SIMULATOR USED: ISE Design Suite 14.2

2.2 THEORY:

A 4-bit ripple carry adder cum subtractor is a digital logic circuit that can perform both addition and subtraction operations on 4-bit binary numbers. This combination of functions is achieved by using a single circuit with an additional control input that determines whether the operation should be addition or subtraction.

Inputs and Outputs:

The circuit has two sets of 4-bit inputs, A and B, representing the numbers to be operated on. It also has a control input, often labeled as "Subtract" or "Borrow-In" (M or K), which determines the operation mode. When the control input is set to 0, the circuit performs addition, and when it's set to 1, it performs subtraction.The circuit produces two 4-bit outputs: the result, which can be either the sum or the difference, and a carry/borrow output (Cout) that can be used as the carry-in or borrow-out for the next stage.

Addition Mode (Control Input = 0):

In this mode, the circuit behaves as a 4-bit ripple carry adder.Each bit position in the adder performs a bit-wise addition of the corresponding bits from A and B.The carry-out from one bit position is used as the carry-in for the next higher-order bit position.

Subtraction Mode (Control Input = 1):

In this mode, the circuit behaves as a 4-bit ripple carry subtractor.

Each bit position in the subtractor performs a bit-wise subtraction of the corresponding bits from A and B.

The borrow-out from one bit position is used as the borrow-in for the next higher-order bit position.

Adder-Subtractor Operation:

In both addition and subtraction modes, the operation is carried out bit by bit, just like in a regular 4-bit adder or subtractor. The control input (Bin) determines whether a borrow or

carry is generated at each bit position. In addition mode, a carry is generated if needed, and in subtraction mode, a borrow is generated if needed.

The final result output is obtained by concatenating the individual bit results from each bit position.The carry-out (in addition mode) or borrow-out (in subtraction mode) from the most significant bit is typically provided as the Cout output.

Table 2.1 Truth Table for 4 bit full adder and subtractor

M/K	A	B	C	Carry/Borrow	Sum/Difference
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	1	1
0	0	1	1	1	0
0	1	0	0	0	1
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	1	0	1
1	0	1	0	0	1
1	0	1	1	1	0
1	1	0	0	0	1
1	1	0	1	1	0
1	1	1	0	1	0
1	1	1	1	1	1

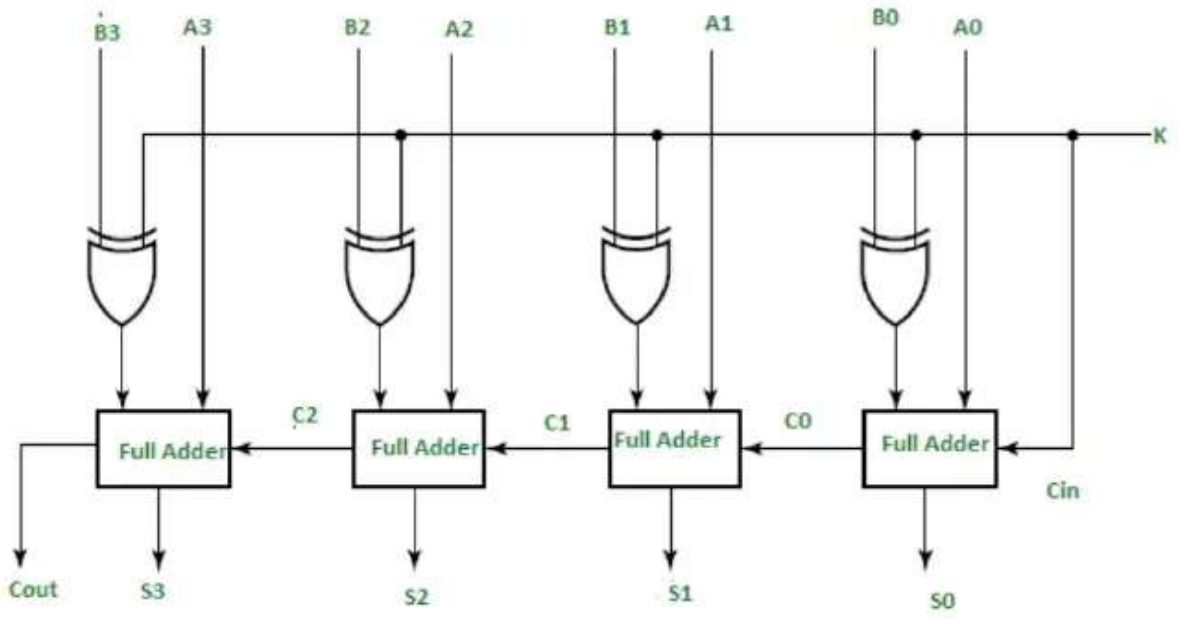


Figure 2.1 Logic circuit for 4 bit full adder and subtractor

In summary, a 4-bit ripple carry adder cum subtractor is a versatile circuit that can handle both addition and subtraction operations on 4-bit binary numbers based on a control input. It offers flexibility in arithmetic operations while utilizing a common hardware platform.

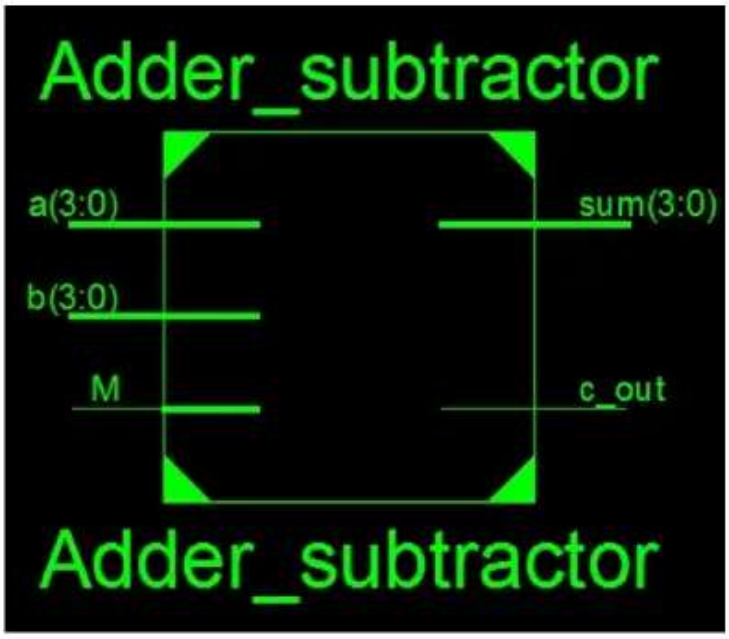
2.3 PROCEDURE:

Step 1: Creating a Ripple Carry Adder in Verilog involves describing the behavior of the adder in a high-level hardware description language. It performs both addition and subtraction based on a control input. Here, we'll use M as the control input, where M = 0 corresponds to addition, and M = 1 corresponds to subtraction. Here is a simple algorithm for designing a 4-bit Ripple Carry Adder using Verilog:

Verilog code:

```
module Add_half (
input a, b,
output c_out, sum
);
xor G1 (sum, a, b);
and G2 (c_out, a, b);
```

```
endmodule
module Add_full (
input a, b, c_in,
output c_out, sum
);
wire w1, w2, w3, w4;
Add_half M1 (a, b, w1, w2);
Add_half M2 (w2, c_in, w3, sum);
or O1 (c_out, w1, w3);
endmodule
module Adder_subtractor (
input [3:0] a, b, // A and B input
input M, // mode input
output [3:0] sum,
output c_out
);
wire [3:0] x;
wire c1, c2, c3;
xor G0 (x[0], b[0], M);
xor G1 (x[1], b[1], M);
xor G2 (x[2], b[2], M);
xor G3 (x[3], b[3], M);
Add_full m0 (.a(a[0]), .b(x[0]), .c_in(M), .sum(sum[0]), .c_out(c1));
Add_full m1 (.a(a[1]), .b(x[1]), .c_in(c1), .sum(sum[1]), .c_out(c2));
Add_full m2 (.a(a[2]), .b(x[2]), .c_in(c2), .sum(sum[2]), .c_out(c3));
Add_full m3 (.a(a[3]), .b(x[3]), .c_in(c3), .sum(sum[3]), .c_out(c_out));
endmodule
```



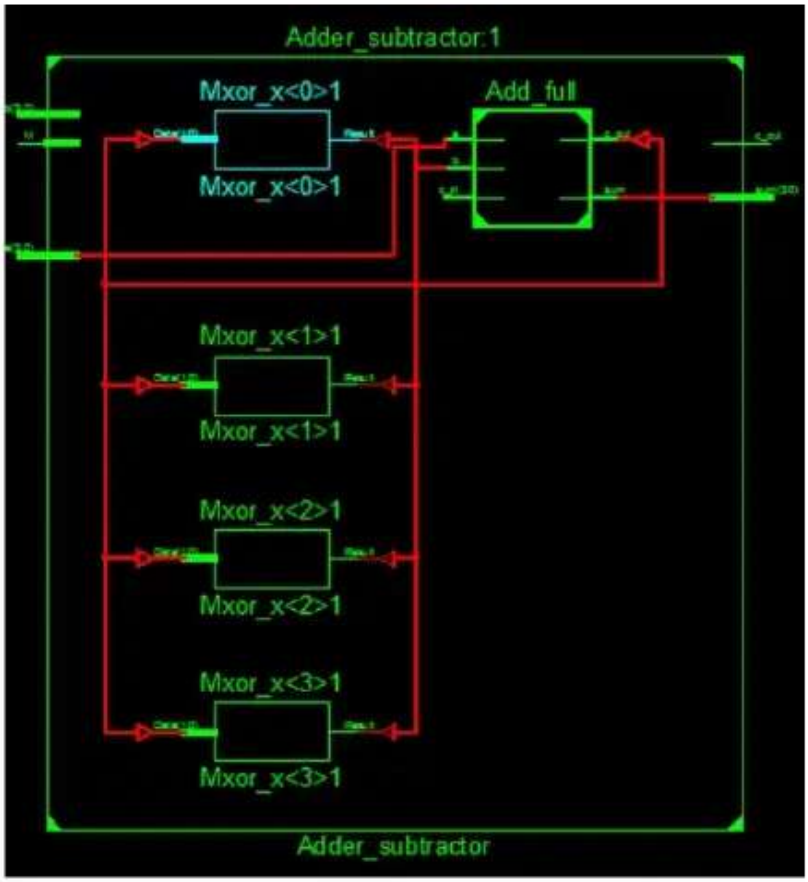


Figure 2.2 RTL Schematic for 4 bit full adder and subtractor

Step 2: Write the Testbench code to obtain the simulation waveform. Here is a simple testbench to verify the functionality of the Ripple Carry Add Sub module:

```
module top;
reg [3:0] a, b;
reg M;
wire [3:0] sum;
wire c_out;
integer i,j;
Adder_subtractor dut (.a(a), .b(b), .M(M), .c_out(c_out), .sum(sum));
initial begin
$monitor("a = %b b = %b M = %b c_out = %b sum = %b", a, b, M, c_out, sum);
M = 1'b0;
for (i = 5; i < 8 ; i = i + 1) begin
for (j = 6; j < 9 ; j = j + 1) begin
a = i; b = j; #20;
end
end
M = 1'b1;
for (i = 6; i < 8 ; i = i + 1) begin
for (j = 0; j < 5 ; j = j + 1) begin
a = i; b = j; #20;
end
end
end
endmodule
```

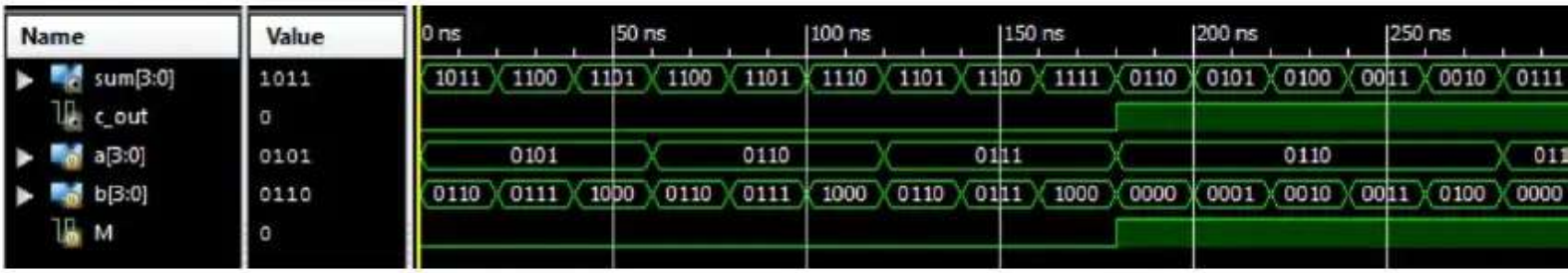


Figure 2.3 RTL simulation waveform for 4 bit full adder and subtractor

Step 3: Obtain the Output:

When you run the simulation, you should see the output messages and signal values in the console or simulation log. The messages will be displayed as the simulation progresses. Make sure that your simulation tool's console or log window is visible and open during simulation to view the output.

a = 0101 b = 0110 M = 0 c_out = 0 sum = 1011
a = 0101 b = 0111 M = 0 c_out = 0 sum = 1100
a = 0101 b = 1000 M = 0 c_out = 0 sum = 1101
a = 0110 b = 0110 M = 0 c_out = 0 sum = 1100
a = 0110 b = 0111 M = 0 c_out = 0 sum = 1101
a = 0110 b = 1000 M = 0 c_out = 0 sum = 1110
a = 0111 b = 0110 M = 0 c_out = 0 sum = 1101
a = 0111 b = 0111 M = 0 c_out = 0 sum = 1110
a = 0111 b = 1000 M = 0 c_out = 0 sum = 1111
a = 0110 b = 0000 M = 1 c_out = 1 sum = 0110
a = 0110 b = 0001 M = 1 c_out = 1 sum = 0101
a = 0110 b = 0010 M = 1 c_out = 1 sum = 0100
a = 0110 b = 0011 M = 1 c_out = 1 sum = 0011
a = 0110 b = 0100 M = 1 c_out = 1 sum = 0010
a = 0111 b = 0000 M = 1 c_out = 1 sum = 0111
a = 0111 b = 0001 M = 1 c_out = 1 sum = 0110
a = 0111 b = 0010 M = 1 c_out = 1 sum = 0101
a = 0111 b = 0011 M = 1 c_out = 1 sum = 0100
a = 0111 b = 0100 M = 1 c_out = 1 sum = 0011

2.4 APPLICATIONS:

Arithmetic Operations:

Calculator and Computer Arithmetic: 4-bit ripple carry adder/subtractors are used in the arithmetic logic units (ALUs) of microprocessors and microcontrollers for performing arithmetic operations. They can add or subtract 4-bit binary numbers as part of more complex calculations.

Binary Operations: These circuits are used in various binary operations, including binary addition, binary subtraction, and complement operations. They are essential in digital systems for performing these fundamental operations.

Data Processing:

Data Compression: In data compression algorithms, arithmetic operations are used to encode and decode data efficiently. A 4-bit ripple carry adder/subtractor can play a role in these processes, allowing for compact data representation.

Encryption and Decryption: In encryption and decryption algorithms, such as those used in secure communication systems, these circuits are used to perform bit-level operations on data to ensure security and privacy.

Error Detection and Correction:

Error-Correcting Codes: In communication systems, error-correcting codes are used to detect and correct errors in transmitted data. These circuits can help in implementing these codes by performing arithmetic operations on encoded data.

Parity Bit Generation and Checking: In systems that use parity bits for error checking and correction, 4-bit adder/subtractor circuits can be used to calculate and verify parity bits.

Control and Processing:

State Machines: In digital systems with sequential logic and state machines, these circuits are used to transition between states and control system behavior based on inputs and conditions.

Comparator Circuits: They are used in comparator circuits to compare two binary numbers and determine their relationship, such as greater than, less than, or equal.

General Purpose Digital Logic:

Digital Filters: In digital signal processing, adders/subtractors can be used in digital filters to perform signal processing tasks like filtering and modulation.

Data Routing and Multiplexing: In multiplexers and demultiplexers, these circuits can help with data routing and selection, allowing different data sources to be combined or separated.

Digital Counters: They are used in digital counters to keep track of events or perform counting operations in digital systems, such as in timers and frequency dividers.

In summary, 4-bit ripple carry adder/subtractor circuits have a wide range of applications in digital systems, particularly in arithmetic operations, data processing, error detection and correction, and control. Their versatility and ability to perform both addition and subtraction make them fundamental components in digital electronics.

2.5 EXPECTED VIVA QUESTIONS

1. Explain the concept of a full adder. What are its inputs and outputs?
2. Can you describe the truth table of a full adder? How many input combinations are there, and how many outputs does it have?
3. What is the purpose of a carry-in (Cin) in a full adder?
4. Walk me through the design process of a 4-bit full adder. What are the major components you need to consider?
5. Which basic logic gates (e.g., AND, OR, XOR, etc.) are required to build a full adder circuit?
6. How do you represent subtraction using a full adder? Explain the concept of two's complement.
7. What is the significance of the carry-out (Cout) in a full adder?
8. Can you describe the inputs and outputs of a 4-bit subtractor circuit? How does it differ from a full adder?
9. Explain how you can simulate the 4-bit full adder and subtractor using basic gates in a digital design tool like Xilinx ISE or similar software.
10. What are the steps to create a testbench for your design? How do you set up test cases for simulation?
11. What is the expected result when adding two 4-bit binary numbers using your full adder? Can you provide an example?
12. How would you verify the correctness of your full adder and subtractor simulations?
13. What are the limitations or potential challenges in this design, and how can you address them?
14. What applications can you think of for a 4-bit full adder and subtractor circuit?

15. Discuss the implications of propagation delay in your circuit. How can you minimize it?
16. What are the differences between a ripple carry adder and a carry-lookahead adder?
17. Explain the trade-offs between area and speed in digital circuit design.
18. How do you ensure that your design is efficient in terms of gate count and power consumption?
19. What are some ways to optimize the performance of your 4-bit full adder and subtractor circuit?
20. Can you demonstrate your understanding by drawing the logic diagrams for a 4-bit full adder and subtractor?

3. Design Verilog HDL to implement simple circuits using structural, Data flow and Behavioural model.

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

SIMULATOR USED: ISE Design suite 14.2

3.2 THEORY:

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: In 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 multiplexer_2to1 (input A, B, S, output Y);
  and gate1 (Y, A, S);
  and gate2 (Y, B, ~S);
endmodule
```

Data Flow Modeling:

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

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

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

Example Verilog code for a 2-to-1 multiplexer using data flow modeling:

```
module multiplexer_2to1 (input A, B, S, output Y);
  assign Y = (S) ? A : B;
endmodule
```

Behavioral Modeling:

Description: Behavioral modeling focuses on high-level functionality, describing how a module behaves without specifying the underlying hardware details. It's often used for algorithmic descriptions.

Use Cases: Behavioral 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 module behaves based on inputs.

Example Verilog code for a simple 4-bit adder using behavioral modeling:

```
module adder_4bit (input [3:0] A, B, output [3:0] Y);
  always @(A, B)
    Y = A + B;
endmodule
```

When designing Verilog HDL code using these modeling styles, consider the following best practices:

1. Ensure that your code is well-documented with comments to describe the functionality and purpose of each module.
2. Test your design thoroughly using testbenches to verify its correctness and simulate its behavior.
3. Pay attention to the coding style and naming conventions to maintain readability and consistency in your code.
4. Be aware of simulation and synthesis implications, as different modeling styles may lead to different synthesis results.
5. Familiarize yourself with the specific Verilog syntax and constructs related to each modeling style.
6. Keep in mind the target hardware and design constraints when choosing the appropriate modeling style for your project.
7. Using the correct modeling style for your design and adhering to good coding practices will help you create Verilog HDL code that is efficient, maintainable, and suited to your specific design requirements.

3.3 PROCEDURE:

Step 1: Minimize the following boolean function-F(A, B, C, D) = Σm(0, 1, 2, 5, 7, 8, 9, 10, 13, 15)

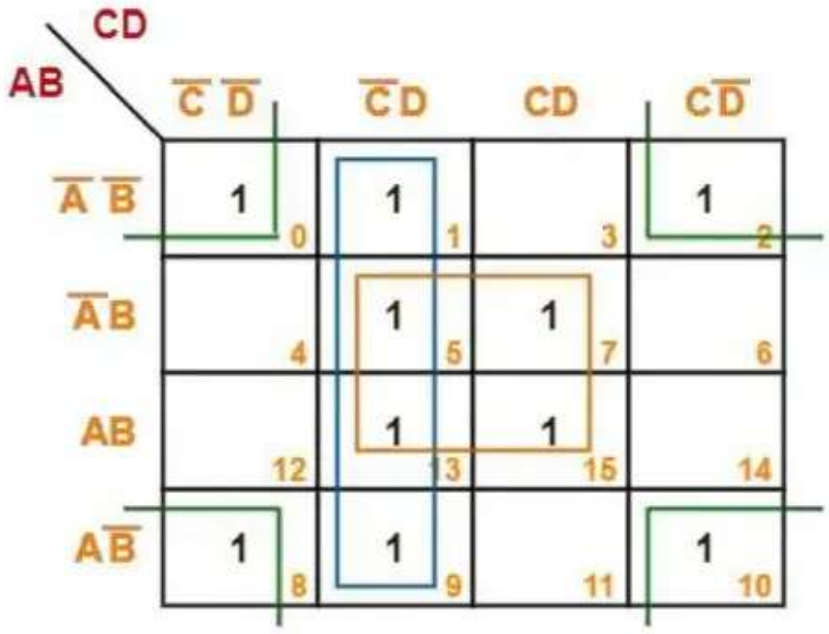


Figure 3.1: K map

Thus, minimized boolean expression is: F(A, B, C, D) = BD + C'D + B'D'.

Step 2: Write the verilog code for the given expression using structural, Data flow and Behavioural model and obtain the RTL schematic.

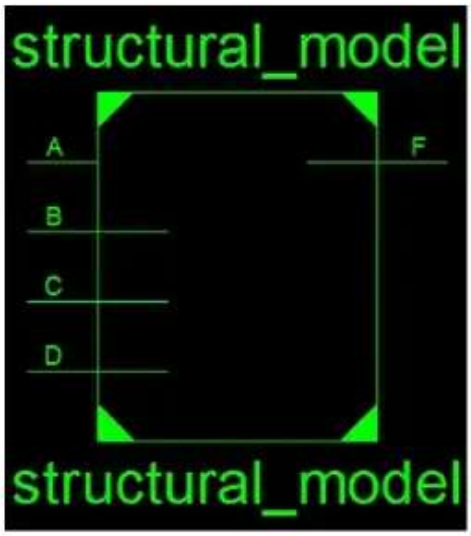
3.3.1 Structural model:

In the structural model, you explicitly instantiate logic gates and connect them to implement the expression. For this expression, you'll need three AND gates and an OR gate.

Verilog code:

```
module structural_model (
   input A, B, C, D,
   output F
);
wire w1,w2,w3;
```

```
// Implement the product terms using AND gates
and gate1 (w1, B, D);
and gate2 (w2, ~C, D);
and gate3 (w3, ~B, ~D);
// Implement the OR gate to combine the product terms
or or_gate (F,w1,w2,w3);
endmodule
```



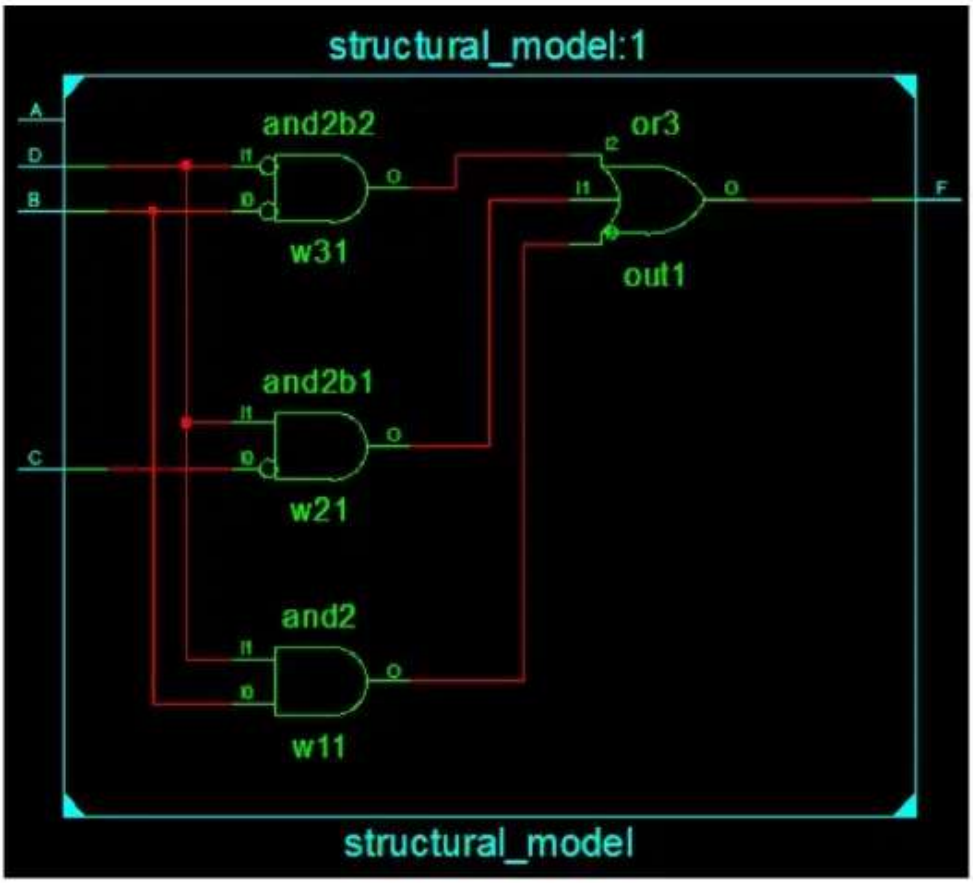


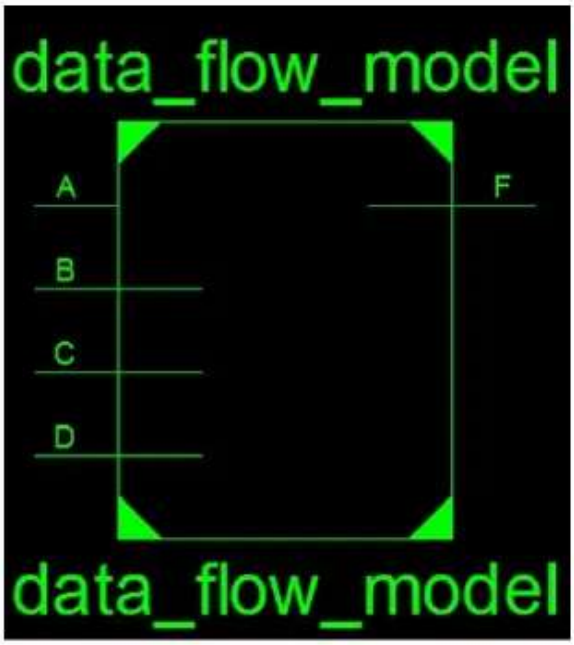
Figure 3.2 RTL Schematic of the given expression using Structural modelling

3.3.2 Dataflow model:

In the data flow model, you describe how the output depends on the input through equations. This approach uses assign statements to define the behavior.

Verilog code:

```
module data_flow_model (
   input A, B, C, D,
   output F
);
assign w1 = B & D;
assign w2 = (~C) & D;
assign w3 = (~B) & (~D);
assign F = w1 | w2 | w3;
endmodule
```



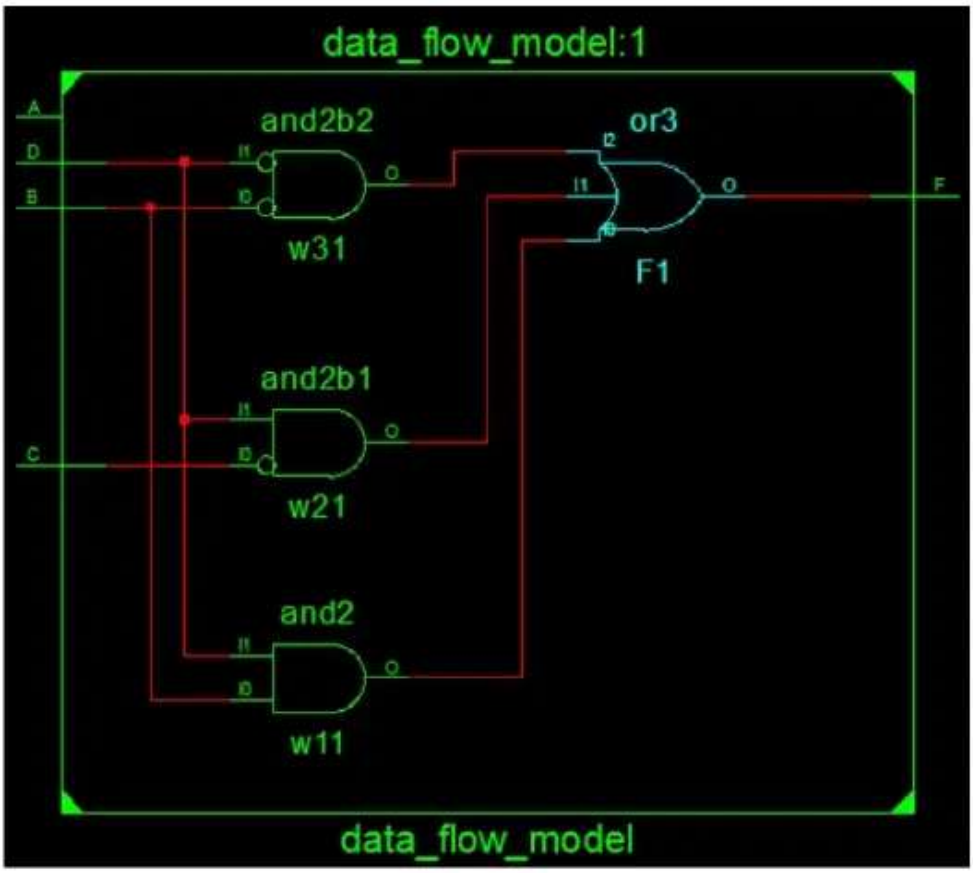


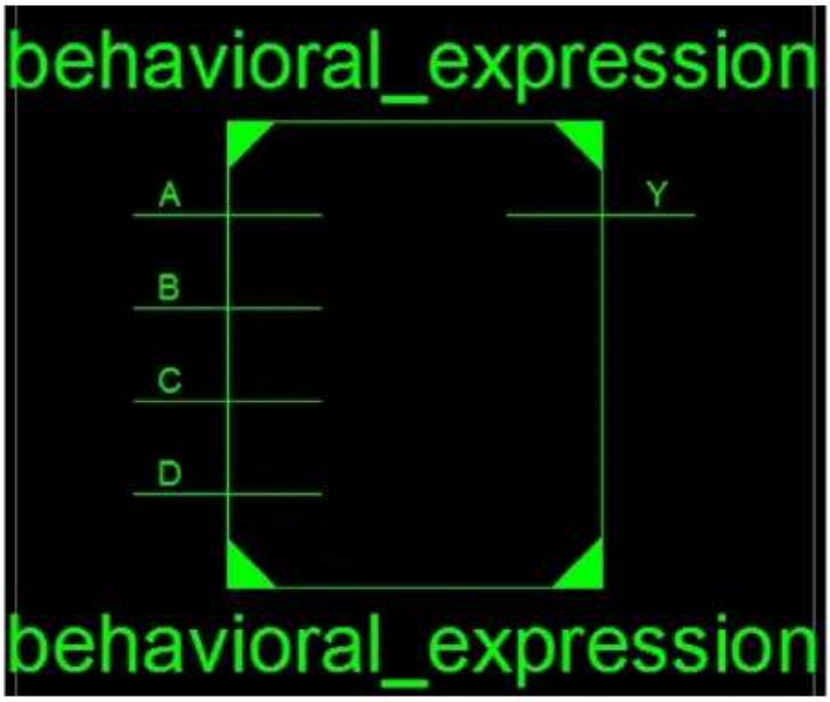
Figure 3.3 RTL Schematic of the given expression using dataflow modelling

3.3.3 Behavioral Model:

In the behavioral model, you describe the circuit's behavior using high-level constructs like if-else statements.

Verilog code:

```
module behavioral_expression (
   input A, B, C, D,
   output reg Y
);
always @* begin
   if (A) begin
      Y = B & D;
   end else if (C) begin
      Y = ~C & D;
   end else begin
      Y = ~B & ~D;
   end
end
endmodule
```



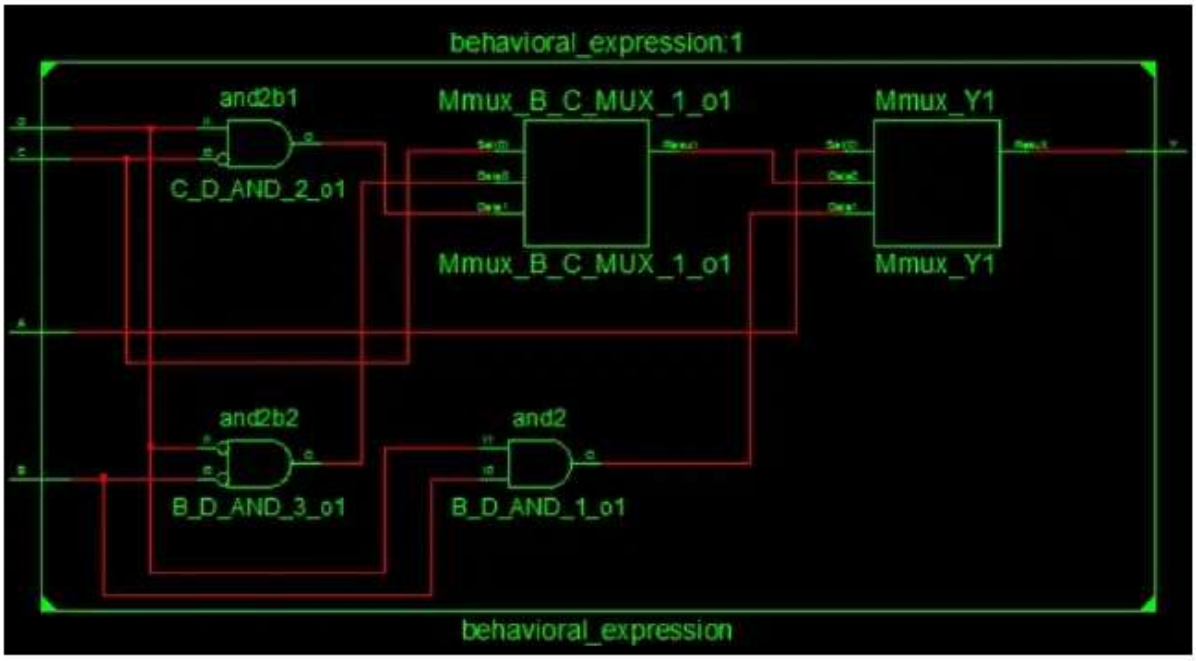


Figure 3.4 RTL Schematic of the given expression using behavioural modelling

Step 3: Write the Testbench code for the given expression using any model and get the RTL waveform.

Testbench code:

```
module simple_cir;
        // Inputs
        reg A;
        reg B;
        reg C;
        reg D;
        // Outputs
        wire Y;
        behavioral_expression uut (
                .A(A),
                .B(B),
                .C(C),
                .D(D),
                .Y(Y)
        );
        initial begin
          $monitor("A = %b B = %b C = %b D = %b Y = %b", A,B,C,D,Y);
                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
    initial #10000 $finish;
endmodule
```

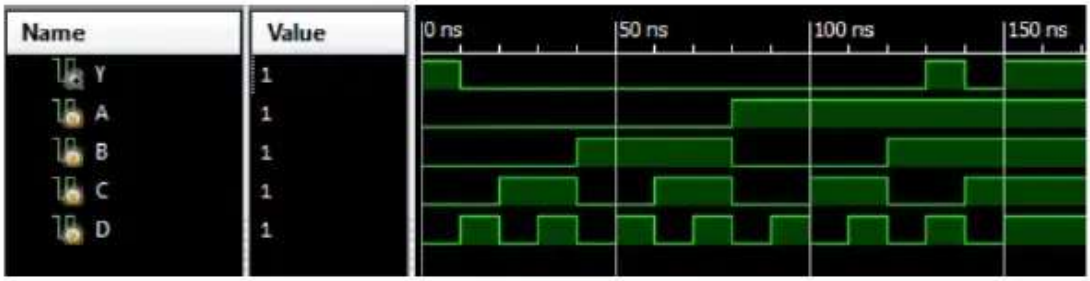


Figure 3.5 RTL Simulation waveform

Step 4: Output:

Get the output from the console window after the simulation process is done.

A = 0 B = 0 C = 0 D = 0 Y = 1

A = 0 B = 0 C = 0 D = 1 Y = 0

A = 0 B = 0 C = 1 D = 0 Y = 0

A = 0 B = 0 C = 1 D = 1 Y = 0

A = 0 B = 1 C = 0 D = 0 Y = 0

A = 0 B = 1 C = 0 D = 1 Y = 0

A = 0 B = 1 C = 1 D = 0 Y = 0

A = 0 B = 1 C = 1 D = 1 Y = 0

A = 1 B = 0 C = 0 D = 0 Y = 0

A = 1 B = 0 C = 0 D = 1 Y = 0

A = 1 B = 0 C = 1 D = 0 Y = 0

A = 1 B = 0 C = 1 D = 1 Y = 0

A = 1 B = 1 C = 0 D = 0 Y = 0

A = 1 B = 1 C = 0 D = 1 Y = 1

A = 1 B = 1 C = 1 D = 0 Y = 0

A = 1 B = 1 C = 1 D = 1 Y = 1

3.4 APPLICATIONS:

1. Dataflow Modeling:

Applications: Dataflow modeling is primarily used for describing the flow of data through a digital circuit and defining the logical relationships between inputs and outputs. It is suitable for applications where the emphasis is on expressing how data is transformed from one form to another.

Combinational logic circuits: Dataflow modeling is particularly useful for describing the behavior of combinational logic circuits, where the outputs are solely dependent on the current inputs.

Arithmetic and mathematical operations: It's used to model operations like addition, subtraction, multiplication, and division.

Data transformations: Dataflow modeling is used to describe data transformations, such as data formatting, encoding, and decoding.

Multiplexers, demultiplexers, and data selectors: These components are often described using dataflow modeling.

2. Behavioral Modeling:

Applications: Behavioral modeling focuses on describing the functionality and operation of a digital system at a higher level of abstraction. It is used to capture the intended behavior of a circuit or system without specifying its internal structure. This style is particularly useful when the emphasis is on system-level functionality.

Algorithmic designs: Behavioral modeling is ideal for representing algorithms and complex computational processes.

Control systems: It is used to describe the control logic that manages the operation of a digital system.

State machines: Behavioral modeling is suitable for describing finite state machines and state transitions.

High-level descriptions: It is often used for top-level design descriptions before diving into lower-level details.

3. Structural Modeling:

Applications: Structural modeling describes a digital system as an interconnected network of individual logic gates and components. It is used to provide a detailed and explicit representation of the physical layout and connectivity of a circuit.

Hierarchical design: Structural modeling is essential for building complex digital systems by creating a hierarchy of interconnected modules.

Component libraries: It is used to design and use custom or standard component libraries, including adders, multipliers, flip-flops, and more.

ASIC and FPGA designs: Structural modeling is crucial for describing the physical layout of an application-specific integrated circuit (ASIC) or field-programmable gate array (FPGA).

In practice, a digital design project often involves a combination of these modeling styles. For example, you might use behavioral modeling for the high-level functionality, dataflow modeling to describe the data processing within specific components, and structural modeling to implement the physical layout of the circuit. The choice of modeling style depends on the specific design requirements and the level of abstraction needed at each stage of the design process.

3.5 EXPECTED VIVA QUESTIONS:

1. Explain the difference between structural, data flow, and behavioral modeling in Verilog. When would you choose one over the other for a specific design?
2. What are the primary components or constructs used in each modeling style in Verilog?
3. Describe the components needed for structural modeling. How do you instantiate and connect these components to implement a specific expression or logic?
4. Can you explain the purpose of module instantiation and how you connect modules in structural modeling?
5. Provide an example of a simple expression, and write Verilog code for it using structural modeling.
6. Explain how data flows through a design when using data flow modeling. How is it different from structural modeling?
7. What constructs in Verilog are commonly used for data flow modeling, and how do you use them to represent the logic of an expression?

8. Give an example of a Verilog code for a simple expression using data flow modeling.
9. Discuss the concept of high-level behavioral modeling in Verilog. When is it appropriate to use this modeling style, and what are its advantages?
10. Explain how procedural blocks like always and if-else statements are used in behavioral modeling.
11. Provide an example of a Verilog code for a simple expression using behavioral modeling.
12. Given an expression (e.g., a Boolean expression, arithmetic operation), demonstrate how you would approach implementing it in Verilog using structural, data flow, and behavioral models.
13. What are the key considerations for selecting the appropriate modeling style for a specific expression?
14. How would you validate and test your Verilog code for the expression using simulation and testbenches?
15. Explain how you would set up a simulation environment to test the Verilog code you've written for the expression.
16. What factors should you consider when synthesizing the Verilog code for your expression for actual hardware implementation?
17. Discuss potential challenges or issues that may arise when transitioning from simulation to synthesis, especially with different modeling styles.
18. How can you optimize your Verilog code to improve performance and reduce resource usage for the given expression?
19. What are some common optimization techniques for each modeling style (structural, data flow, and behavioral)?

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

4.1 AIM: To Design and implement Binary Adder-Subtractor – Half and Full Adder, Half and Full Subtractor using verilog HDL

Simulator used: ISE Design Suite 14.2

4.2 HALF ADDER:

4.2.1 THEORY:

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

Figure 4.1 Block Diagram of Half Adder

Table 4.1 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

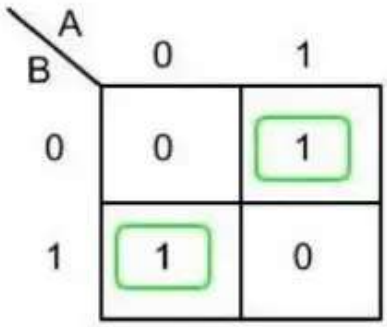


Figure 4.2 K map for Sum

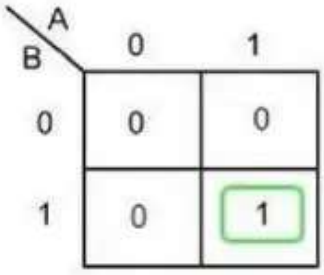


Figure 4.3 K map for Carry

The simplified expression from k map are :

Sum=A'B+AB'

Carry = AB

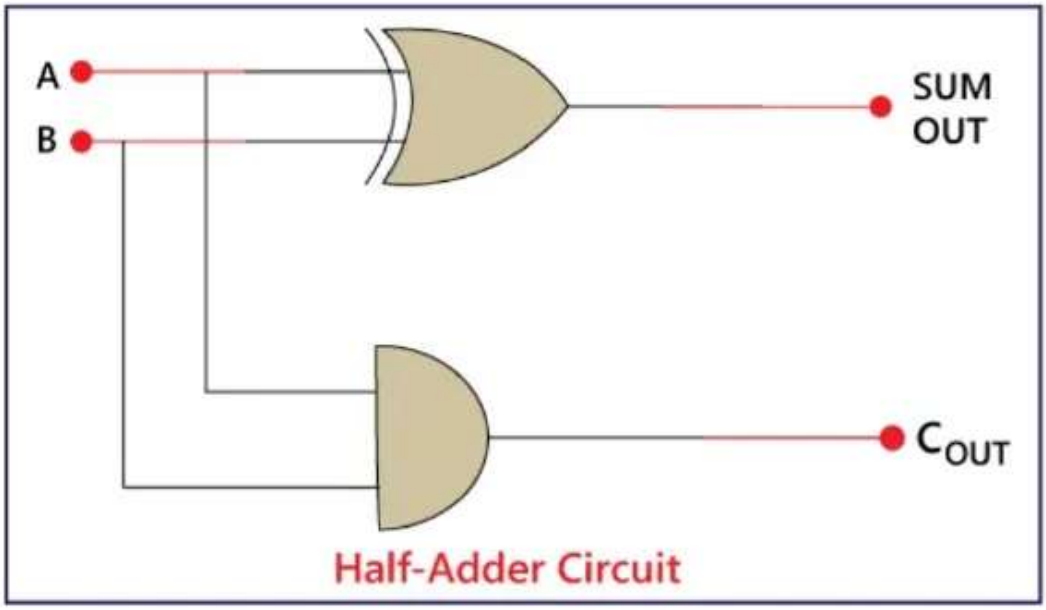


Figure 4.4 Logic circuit of Half Adder

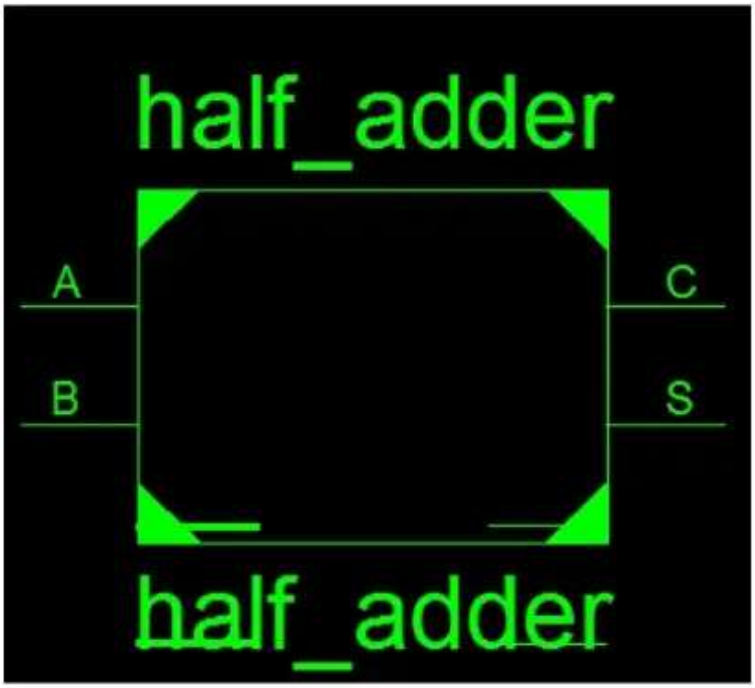
As half adder considers only two bits so along with the addition of two single bits, it cannot accommodate an extra carry bit from the previously generated result. Hence, it is called a half-adder.

4.2.2 PROCEDURE:

Step 1: Write Verilog code using the appropriate modeling 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 the RTL schematic for Half adder.

Verilog code for Half adder

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



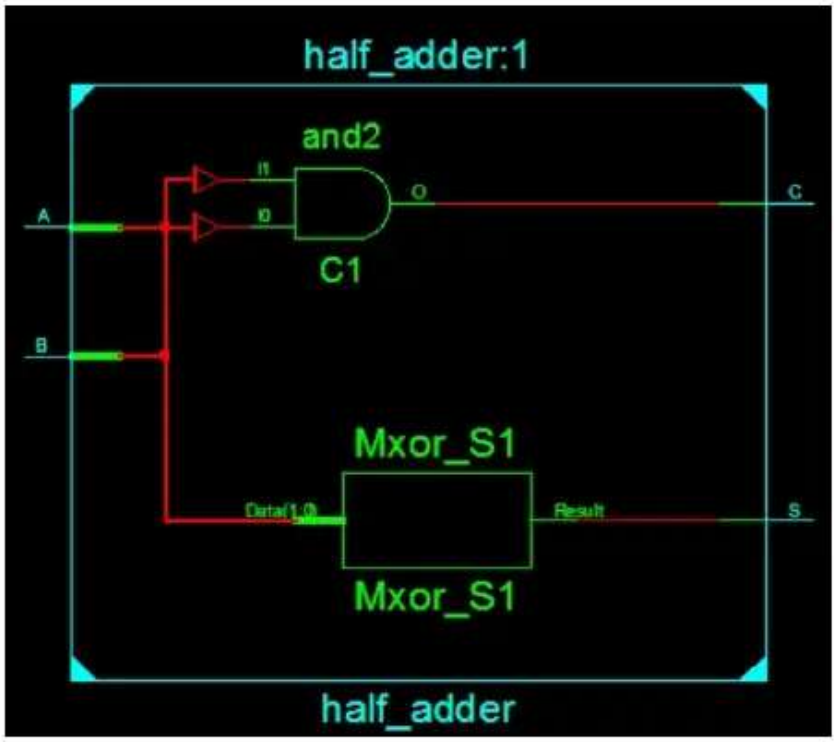


Figure 4.5 RTL Schematic for Half adder

Step 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 tb_half_adder();
reg a , b ;
wire s , c;
half_adder ckt(a,b,s,c);
initial
begin
        a = 1'b0;
        b = 1'b0;
        #100
        a = 1'b1;
        b = 1'b0;
        #100
```

```
        a = 1'b0;
        b = 1'b1;
        #100
        a = 1'b1;
        b = 1'b1;
end
endmodule
```

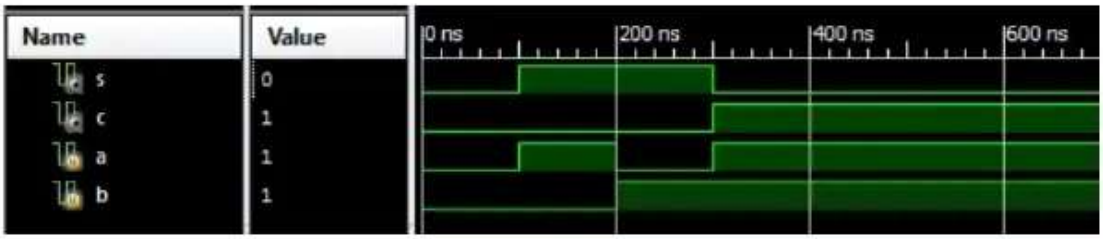


Figure 4.6 RTL Simulation waveform for code for Half adder

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

At time 0: a=0 b=0, s=0, c=0

At time 100000: a=1 b=0, s=1, c=0

At time 200000: a=0 b=1, s=1, c=0

At time 300000: a=1 b=1, s=0, c=1

4.2.3 APPLICATIONS OF HALF ADDER:

1. Half adders are used in various digital circuits, including arithmetic logic units (ALUs) and adder-subtractor units.
2. They are the basic building blocks for constructing full adders and multi-bit adders.
3. Half adders can be used to add two binary numbers to get the least significant bits of the result.

4.3 FULL ADDER

4.3.1 THEORY

The full adder adds three single-bit input and produce two single-bit output. Thus, it is useful when an extra carry bit is available from the previously generated result.

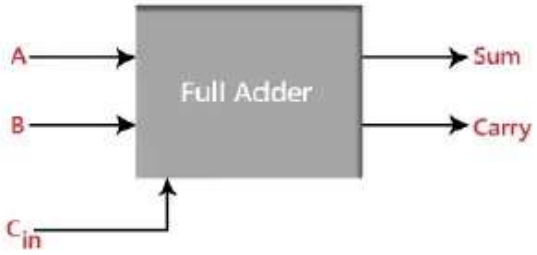


Figure 4.7 Block Diagram of Full adder

Table 4.2 Truth Table of 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

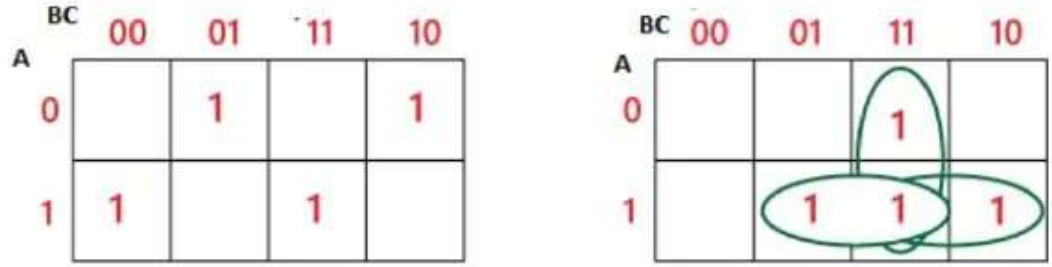


Figure 4.8:K map for Sum and Carry

The simplified expressions are

$$Sum = A'B'C + A'BC + AB'C' + ABC$$

$$Carry = AB + AC + BC$$

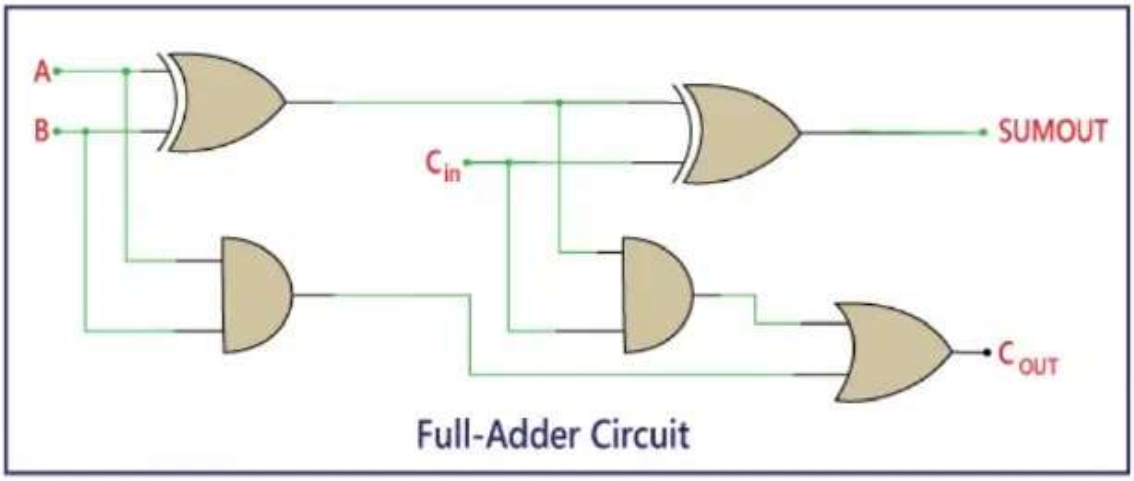
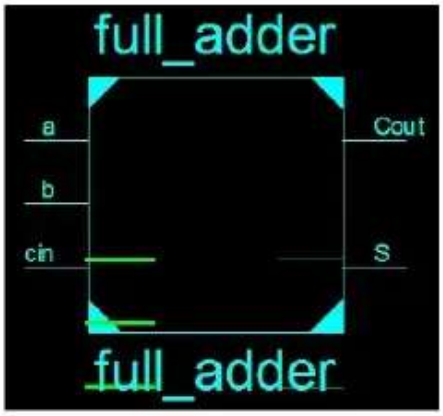


Figure 4.9: Logic circuit for Full adder

4.3.2 PROCEDURE:

Step 1: Write the Verilog code for full adder and obtain the RTL schematic

```
module full_adder(input a, b, cin, output S, Cout);
  assign S = a ^ b ^ cin;
  assign Cout = (a & b) | (b & cin) | (a & cin);
endmodule
```



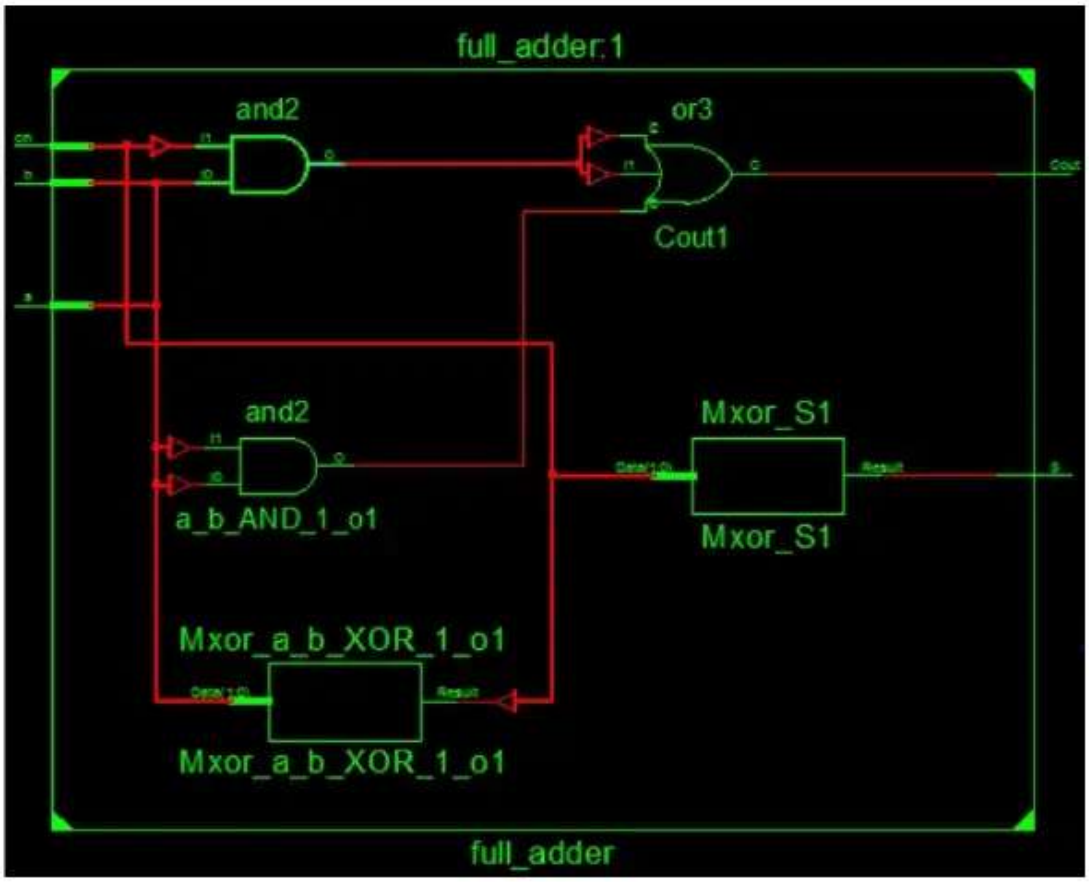


Figure 4.10 RTL Schematic for Full adder

Step 2: Write the testbench code for full adder and obtain the RTL simulation waveform

Testbench code:

```
module tb_top;
 reg a, b, c;
 wire s, c_out;

 full_adder fa(a, b, c, s, c_out);
 initial begin
  $monitor("At time %0t: a=%b b=%b, cin=%b, sum=%b, carry=%b",$time, a,b,c,s,c_out);
  a = 0; b = 0; c = 0; #1;
  a = 0; b = 0; c = 1; #1;
  a = 0; b = 1; c = 0; #1;
  a = 0; b = 1; c = 1; #1;
  a = 1; b = 0; c = 0; #1;
```

```
  a = 1; b = 0; c = 1; #1;
  a = 1; b = 1; c = 0; #1;
  a = 1; b = 1; c = 1;
 end
endmodule
```

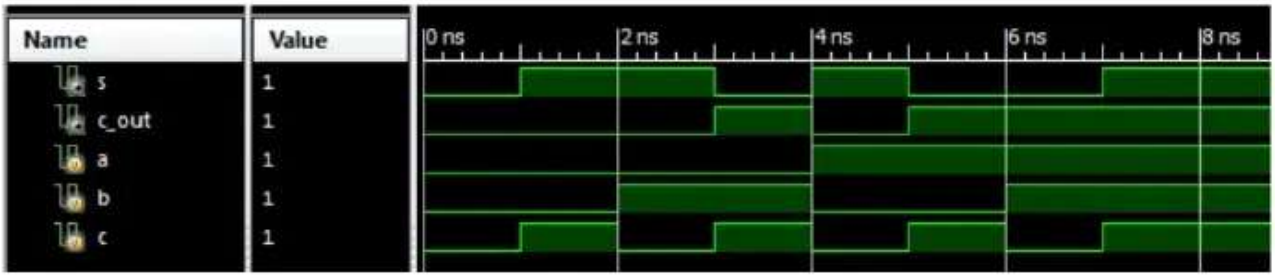


Figure 4.11 RTL Simulation waveform for Full Adder

Step 3: Get the result from the output window

At time 0: a=0 b=0, cin=0, sum=0, carry=0

At time 1000: a=0 b=0, cin=1, sum=1, carry=0

At time 2000: a=0 b=1, cin=0, sum=1, carry=0

At time 3000: a=0 b=1, cin=1, sum=0, carry=1

At time 4000: a=1 b=0, cin=0, sum=1, carry=0

At time 5000: a=1 b=0, cin=1, sum=0, carry=1

At time 6000: a=1 b=1, cin=0, sum=0, carry=1

At time 7000: a=1 b=1, cin=1, sum=1, carry=1

4.3.3 CONSTRUCTION OF FULL ADDER USING TWO HALF ADDER

To build a Full Adder, you can use two Half Adders and an OR gate as follows:

1. Use the first Half Adder to add A and B, producing a partial sum (S1) and a partial carry-out (Cout1).
2. Use the second Half Adder to add S1 and the third input, Cin, producing the final sum (S) and another partial carry-out (Cout2).
3. Combine Cout1 and Cout2 using an OR gate to get the final carry-out (C) of the Full Adder.

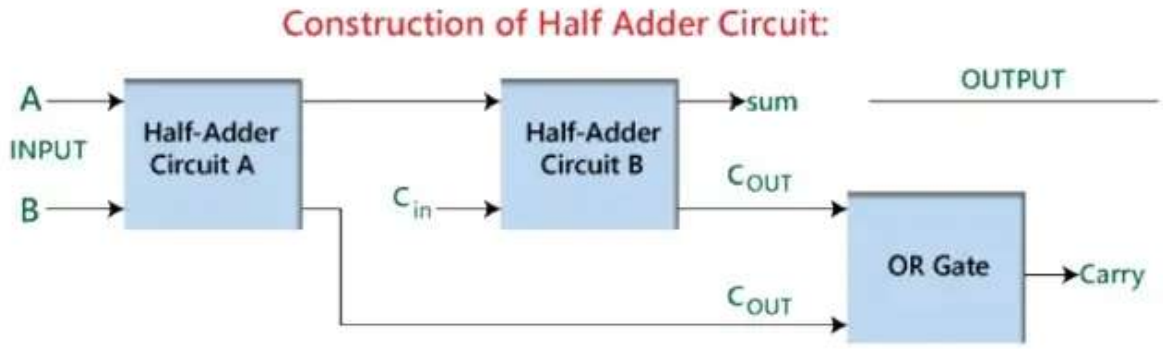


Figure 4.12 Block diagram of full adder using two half adders

Procedure :

Step 1: Write the Verilog code for full adder and obtain the RTL schematic

```
module half_addr(input a, b, output s, c);
  assign s = a^b;
  assign c = a & b;
endmodule
module full_adder(input a, b, cin, output s_out, c_out);
  wire s, c0, c1;
  half_addr HA1 (a, b, s, c0);
  half_addr HA2 (s, cin, s_out, c1);
  assign c_out = c0 | c1;
endmodule
```

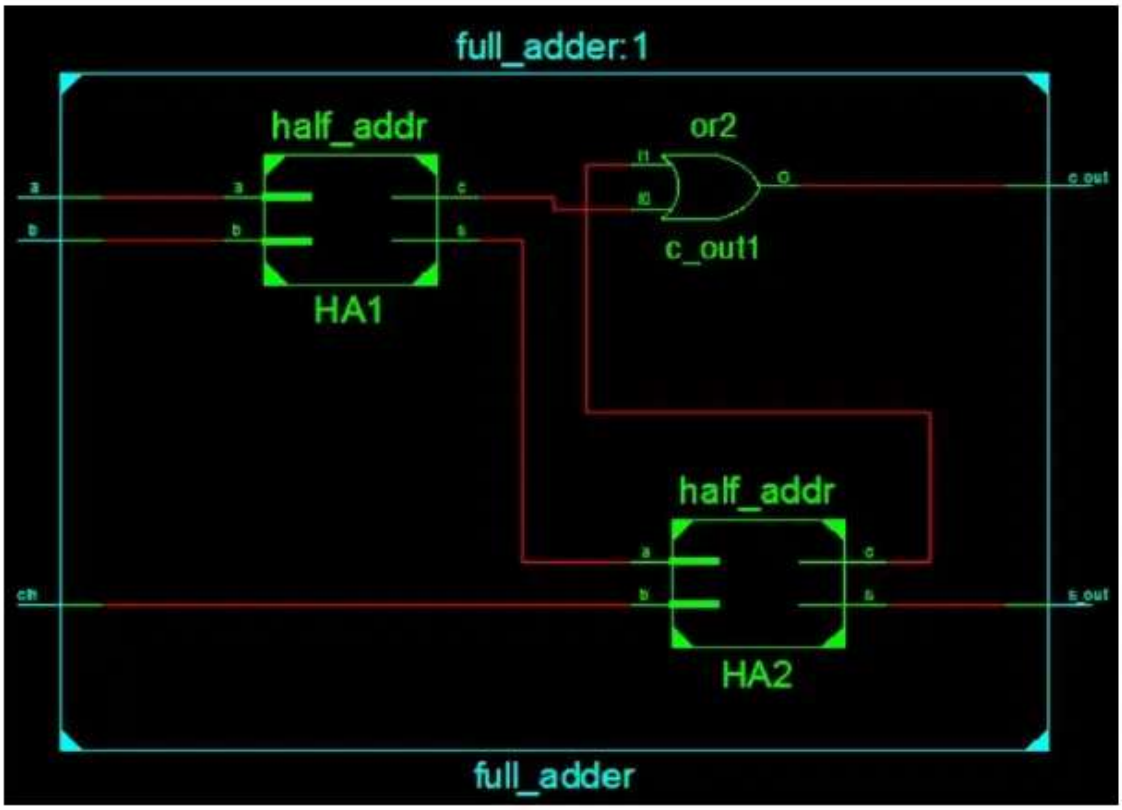


Figure 4.13 RTL Schematic for Full adder using half adders

Step 2: Write the testbench code for full adder and obtain the RTL simulation waveform

Testbench code. Here the testbench code for full adder using two half adders is same as that of the testbench code of full adder.

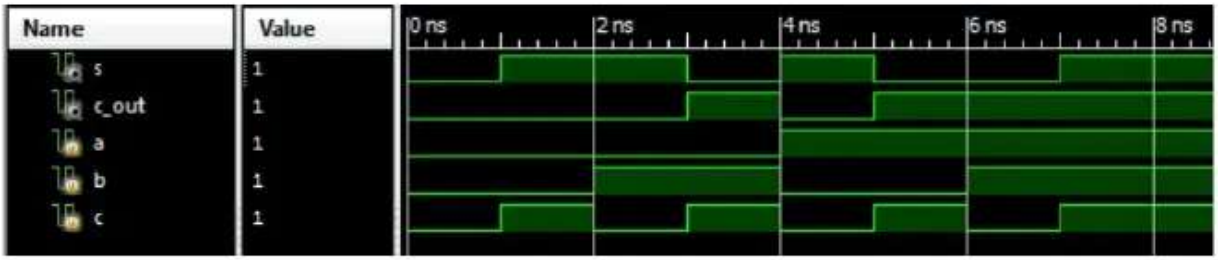


Figure 4.14 RTL Simulation waveform for Full Adder using half adders

Step 3: Get the result from the output window

At time 0: a=0 b=0, cin=0, sum=0, carry=0

At time 1000: a=0 b=0, cin=1, sum=1, carry=0

At time 2000: a=0 b=1, cin=0, sum=1, carry=0

At time 3000: a=0 b=1, cin=1, sum=0, carry=1

At time 4000: a=1 b=0, cin=0, sum=1, carry=0

At time 5000: a=1 b=0, cin=1, sum=0, carry=1

At time 6000: a=1 b=1, cin=0, sum=0, carry=1

At time 7000: a=1 b=1, cin=1, sum=1, carry=1

4.3.4 APPLICATIONS:

1. **Arithmetic Logic Units (ALUs):** Full adders are used as the building blocks of Arithmetic Logic Units in microprocessors and digital calculators. They perform binary addition for multi-bit inputs and produce the sum and carry-out.
2. **Binary Counters:** In digital counters, full adders are employed to accumulate binary numbers as they count. They help create more complex sequential circuits and counting mechanisms.

4.4 HALF SUBTRACTOR

4.4.1 THEORY:

The half subtractor works opposite to the half adder as it substracts two single bits and results in a difference bit and borrow bit as an output. As half subtractor considers only two bits so along with the subtraction of two single bits, it cannot accommodate an extra borrow bit from the previously generated result. Hence, it is called a half-subtractor.

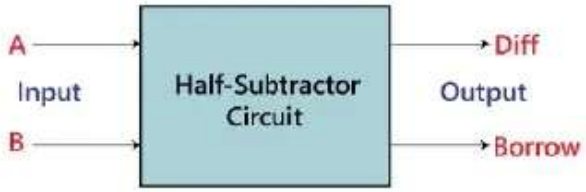


Figure 4.14 Block Diagram of the Half subtractor

Table 4.3 Truth Table of Half Subtractor

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

The simplified expressions are:

Diff= A'B+AB'

Borrow = A'B

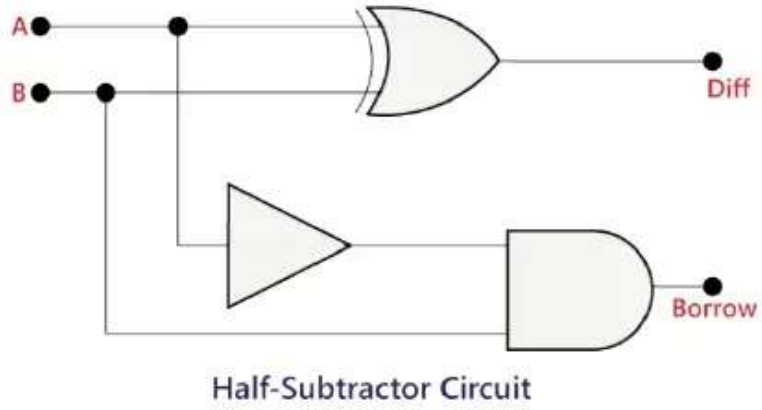


Figure 4.15 Logic circuit for Half Subtractor

4.4.2 PROCEDURE:

Step 1: Write the Verilog code for Half Subtractor and obtain the RTL schematic

```
module half_subtractor(input a, b, output D, B);
  assign D = a ^ b;
  assign B = ~a & b;
endmodule
```

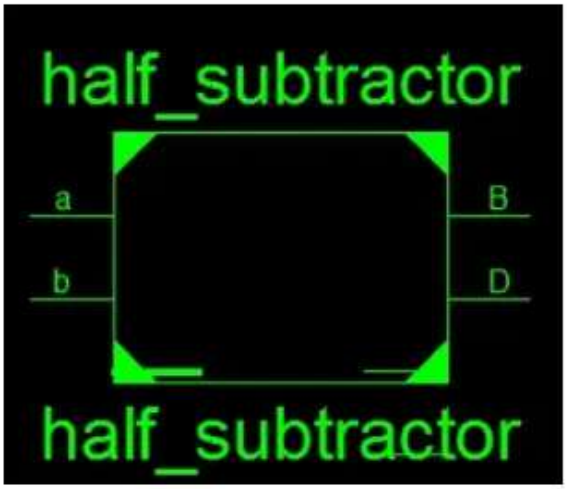


Figure 4.16 RTL Schematic for Half Subtractor

Step 2: Write the testbench code for Half Subtractor and obtain the RTL simulation waveform

Testbench code:

```
module tb_top;
 reg a, b;
 wire D, B;
 half_subtractor hs(a, b, D, B);
 initial begin
  $monitor("At time %0t: a=%b b=%b, difference=%b, borrow=%b",$time, a,b,D,B);
  a = 0; b = 0;
  #1;
  a = 0; b = 1;
  #1;
  a = 1; b = 0;
  #1;
  a = 1; b = 1;
 end
endmodule
```

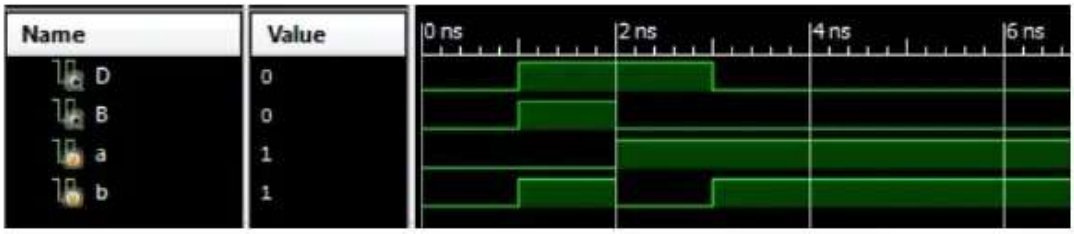


Figure 4.17 RTL Simulation waveform for Half subtractor

Step 3: Get the result from the output window

At time 0: a=0 b=0, difference=0, borrow=0

At time 1000: a=0 b=1, difference=1, borrow=1

At time 2000: a=1 b=0, difference=1, borrow=0

At time 3000: a=1 b=1, difference=0, borrow=0

4.4.3 APPLICATIONS:

1. Binary Subtraction: A half subtractor is used to perform binary subtraction, generating the difference and borrow. It is used in digital calculators, logic circuits, and microprocessor components when subtraction is required.
2. Sequential Circuits: Half subtractors can be integrated into sequential circuits where subtraction is necessary, such as in state machines or digital signal processing.

4.5 FULL SUBTRACTOR

4.5.1 THEORY:

A full subtractor is designed to accommodate the extra borrow bit from the previous stage. Thus it has three single-bit inputs and produces two single-bit output

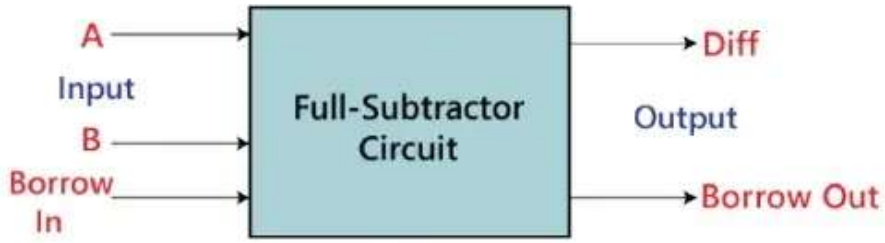


Figure 4.18 Block Diagram of the Full subtractor

Table 4.4 Truth Table of 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

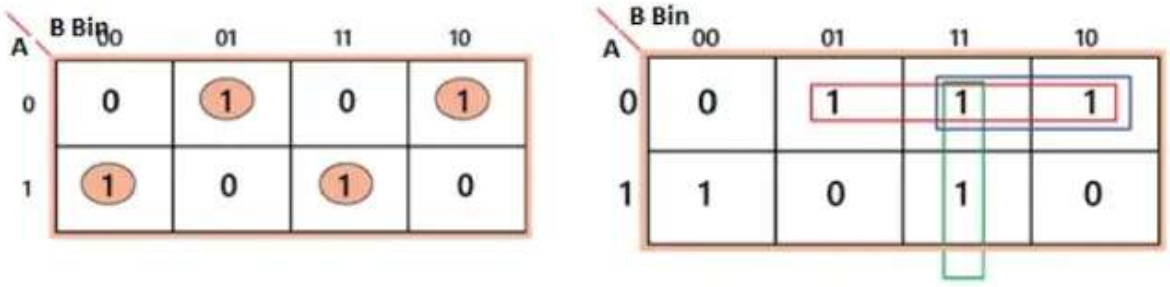


Figure 4.19 K map for Diff and Borrow of full subtractor

The SOP form can be obtained with the help of K-map as:

Diff=xy' z'+x' y' z+xyz+x'yz'

Borrow=x' z+x' y+yz

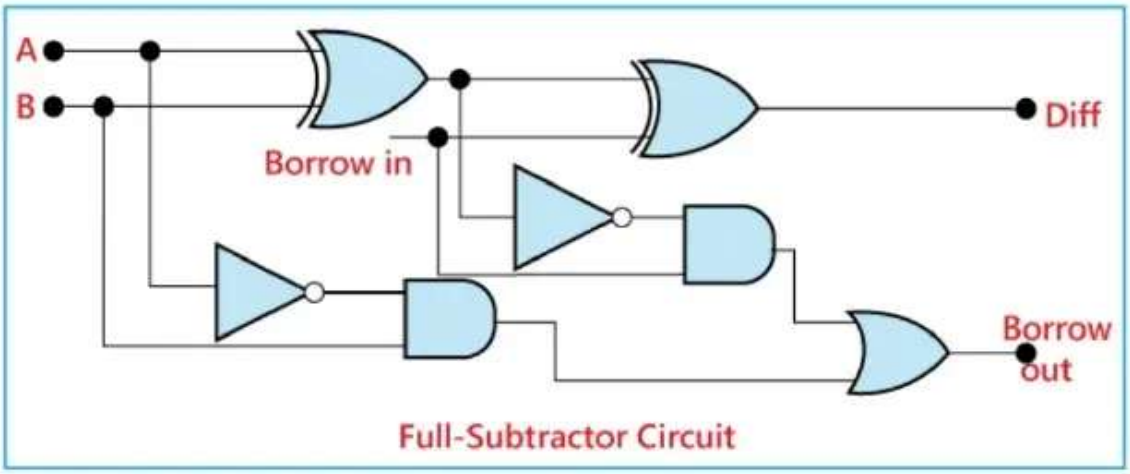
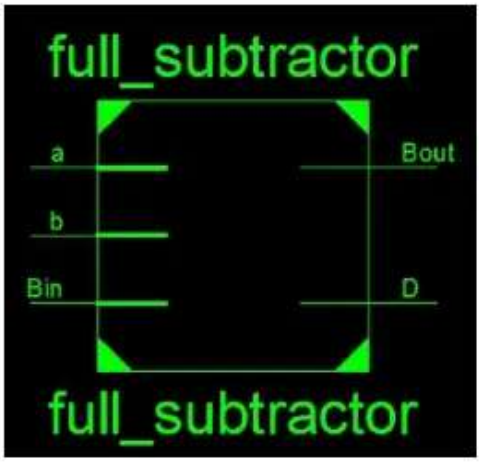


Figure 4.20: Logic circuit for Full Subtractor

4.5.2 PROCEDURE:

Step 1: Write the Verilog code for Full Subtractor and obtain the RTL schematic

```
module full_subtractor(input a, b, Bin, output D, Bout);
  assign D = a ^ b ^ Bin;
  assign Bout = (~a & b) | (~(a ^ b) & Bin);
endmodule
```



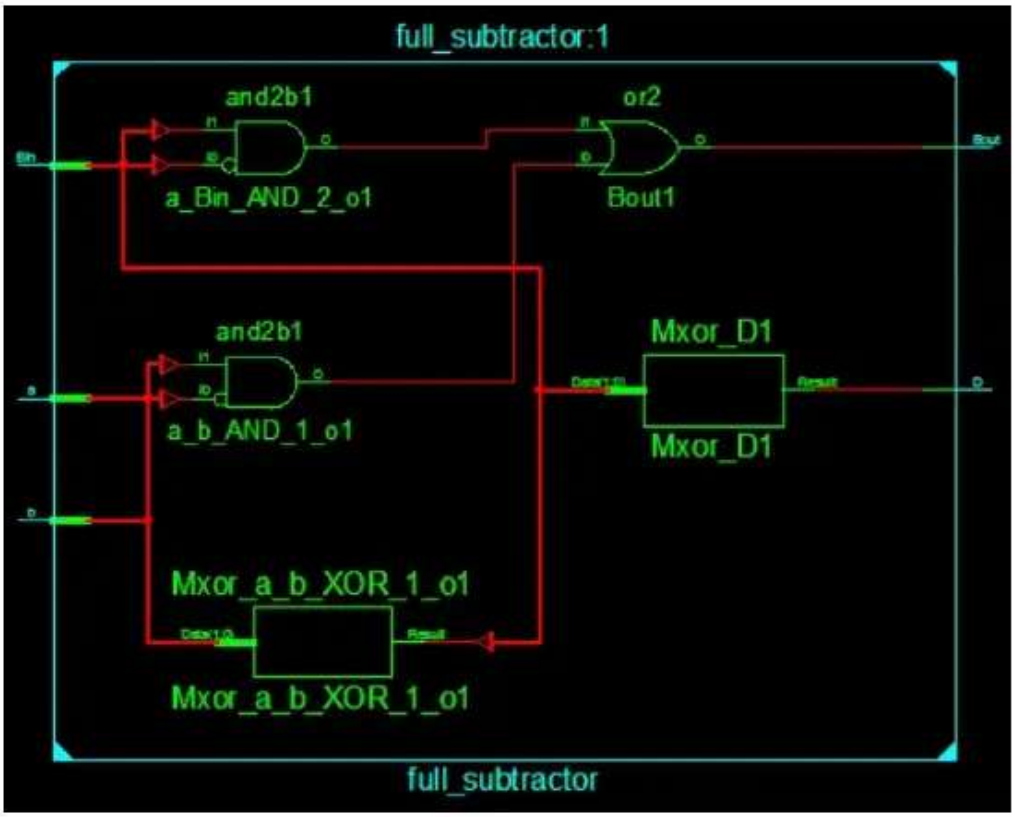


Figure 4.21 RTL Schematic for Full Subtractor

Step 2: Write the testbench code for Full Subtractor and obtain the RTL simulation waveform

Testbench code:

```
module tb_top;
 reg a, b, Bin;
 wire D, Bout;
 full_subtractor fs(a, b, Bin, D, Bout);
initial begin
 $monitor("At time %0t: a=%b b=%b, Bin=%b, difference=%b, borrow=%b",$time, a,b,Bin,D,Bout);
  a = 0; b = 0; Bin = 0; #1;
  a = 0; b = 0; Bin = 1; #1;
  a = 0; b = 1; Bin = 0; #1;
  a = 0; b = 1; Bin = 1; #1;
  a = 1; b = 0; Bin = 0; #1;
```

```
  a = 1; b = 0; Bin = 1; #1;
  a = 1; b = 1; Bin = 0; #1;
  a = 1; b = 1; Bin = 1;
 end
endmodule
```

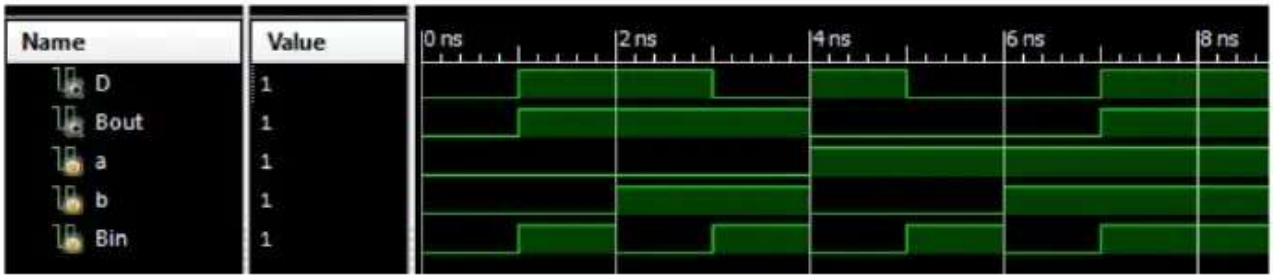


Figure 4.22 RTL Simulation waveform for Full Subtractor

Step 3: Get the result from the output window

At time 0: a=0 b=0, Bin=0, difference=0, borrow=0

At time 1000: a=0 b=0, Bin=1, difference=1, borrow=1

At time 2000: a=0 b=1, Bin=0, difference=1, borrow=1

At time 3000: a=0 b=1, Bin=1, difference=0, borrow=1

At time 4000: a=1 b=0, Bin=0, difference=1, borrow=0

At time 5000: a=1 b=0, Bin=1, difference=0, borrow=0

At time 6000: a=1 b=1, Bin=0, difference=0, borrow=0

At time 7000: a=1 b=1, Bin=1, difference=1, borrow=1

4.5.3 FULL SUBTRACTOR USING TWO HALF SUBTRACTOR:

Theory: To build a Full Subtractor, you can use two Half Subtractors and an OR gate as follows:

1. Use the first Half Subtractor to subtract B from A, producing a partial difference (D1) and a partial borrow (Bout1).
2. Use the second Half Subtractor to subtract Bout1 from D1 and subtract the third input, Bin, producing the final difference (D) and another partial borrow (Bout2).
3. Combine Bout1 and Bout2 using an OR gate to get the final borrow-out (Bout) of the Full Subtractor.

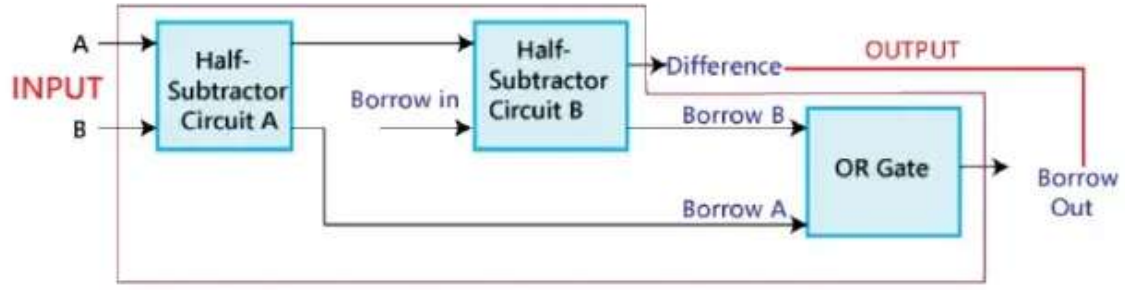


Figure 4.23: Logic circuit for Full subtractor using half subtractors

Procedure:

Step 1: Write the Verilog code for Full subtractor and obtain the RTL schematic

```
module or_gate(a0, b0, c0);
input a0, b0;
output c0;
assign c0 = a0 | b0;
endmodule
module xor_gate(a1, b1, c1);
input a1, b1;
output c1;
assign c1 = a1 ^ b1;
endmodule
module and_gate(a2, b2, c2);
input a2, b2;
output c2;
assign c2 = a2 & b2;
endmodule
module not_gate(a3, b3);
input a3;
output b3;
assign b3 = ~ a3;
endmodule
module half_subtractor(a4, b4, c4, d4);
```

```
input a4, b4;
output c4, d4;
wire x;
xor_gate u1(a4, b4, c4);
and_gate u2(x, b4, d4);
not_gate u3(a4, x);
endmodule
module full_subtractor(A, B, Bin, D, Bout);
input A, B, Bin;
output D, Bout;
wire p, q, r;
half_subtractor u4(A, B, p, q);
half_subtractor u5(p, Bin, D, r);
or_gate u6(q, r, Bout);
endmodule
```

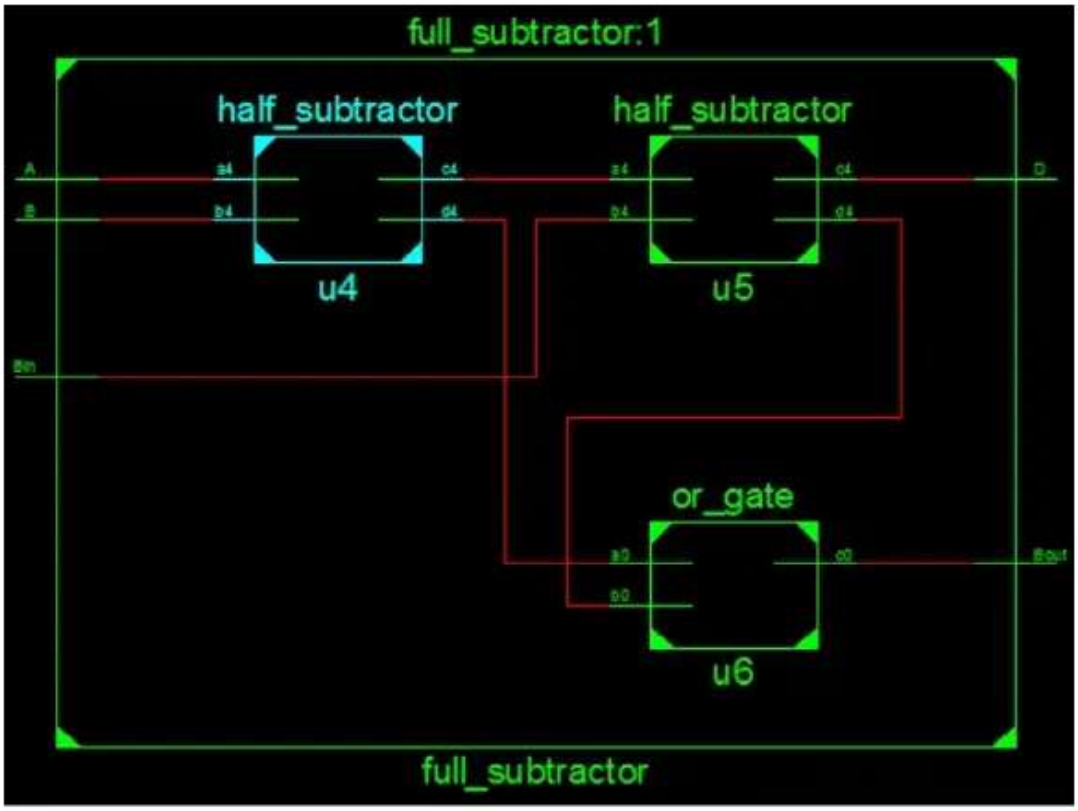


Figure 4.24 RTL Schematic for Full subtractor using half subtractors

Step 2: Write the testbench code for Full subtractor and obtain the RTL simulation waveform

Testbench code:

```
`timescale 1ns / 1ps
module top;
 reg a, b, bin;
 wire d, bout;
 full_subtractor instantiation(.A(a), .B(b), .Bin(bin), .D(d), .Bout(bout));
 initial
 begin
   $dumpfile("xyz.vcd");
   $dumpvars;
   a=0;
   b=0;
   bin=0;
   #10000 $finish;
 end
always #40 a=~a;
always #20 b=~b;
always #10 bin=~bin;
 always @(a or b or bin)
   $monitor("At TIME(in ns)=%t, A=%d B=%d Bin=%d D = %d Bout = %d", $time, a, b, bin, d,
bout);
Endmodule
```

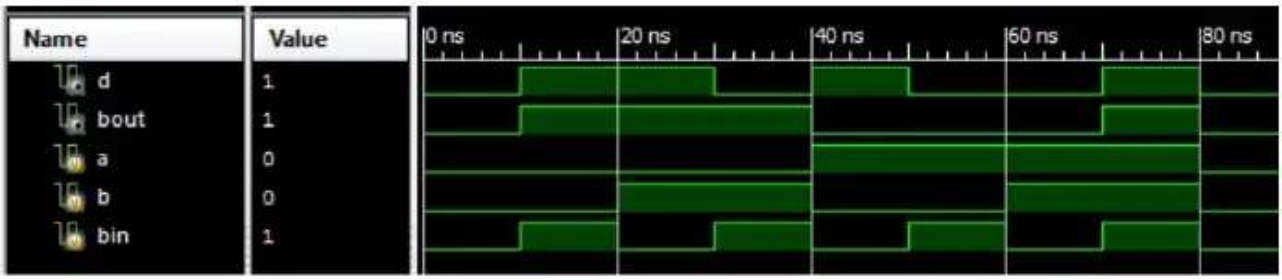


Figure 4.25 RTL Simulation waveform for Full subtractor using half subtractors

Step 3: Get the result from the output window

At TIME(in ns)= 0, A=0 B=0 Bin=0 D = 0 Bout = 0

At TIME(in ns)= 10000, A=0 B=0 Bin=1 D = 1 Bout = 1

At TIME(in ns)= 20000, A=0 B=1 Bin=0 D = 1 Bout = 1

At TIME(in ns)= 30000, A=0 B=1 Bin=1 D = 0 Bout = 1

At TIME(in ns)= 40000, A=1 B=0 Bin=0 D = 1 Bout = 0

At TIME(in ns)= 50000, A=1 B=0 Bin=1 D = 0 Bout = 0

At TIME(in ns)= 60000, A=1 B=1 Bin=0 D = 0 Bout = 0

At TIME(in ns)= 70000, A=1 B=1 Bin=1 D = 1 Bout = 1

4.5.4 APPLICATIONS:

1. **Arithmetic Logic Units (ALUs):** Full Subtractors are essential components of Arithmetic Logic Units, which are a key part of microprocessors and digital calculators. They enable the microprocessor to perform subtraction operations.
2. **Digital Calculators:** Full Subtractors are used in digital calculators to perform subtraction operations. They help users perform arithmetic calculations quickly and accurately.
3. **Subtraction with Borrow:** Full Subtractors are particularly useful when performing binary subtraction with the need for a borrow-out. This is important for carrying out subtraction in multi-bit binary numbers.
4. **Digital Signal Processing:** In applications involving digital signal processing, Full Subtractors can be used to manipulate and process digital signals. They are particularly useful for subtraction operations in signal processing algorithms.
5. **State Machines:** Full Subtractors can be used in digital systems and state machines where subtraction is part of the control logic. This can be important for applications that involve conditional branching and decision-making.
6. **Counters and Sequential Circuits:** Full Subtractors can be incorporated into digital counters and sequential circuits where subtraction is required as part of the counting and sequencing process. They help manage the transition between states in these circuits.

7. **Multi-bit Subtraction:** In cases where multi-bit subtraction is necessary, Full Subtractors can be cascaded together to create more complex subtractors. This is essential for subtracting larger binary numbers.
8. **Parallel Subtraction:** Full Subtractors are used for parallel subtraction of multi-bit binary numbers, especially in applications that require high-speed data processing.
9. **Binary Coded Decimal (BCD) Arithmetic**: Full Subtractors play a role in BCD arithmetic operations, which are common in applications like digital clocks, timers, and numerical displays.

In summary, Full Subtractors are fundamental components in digital electronics, with applications in microprocessors, digital calculators, digital signal processing, state machines, and various other digital systems where binary subtraction is required. They are essential for performing accurate and efficient subtraction operations in digital circuits.

4.6 EXPECTED VIVA QUESTIONS:

Half Adder:

1. What is the purpose of a half adder in digital circuits?
2. Can you explain the truth table of a half adder and its outputs?
3. Describe the logic diagram or circuit for a half adder.
4. How many inputs and outputs does a half adder have, and what are they?
5. What happens when you apply binary addition to the inputs of a half adder?
6. How is the carry-out (Cout) determined in a half adder?
7. In what scenarios is a half adder used in digital design?
8. Can you explain how to construct a full adder using two half adders?
9. How does a half adder differ from a full adder, in terms of inputs and outputs?
10. What are the implications of using a half adder in multi-bit binary addition?"

Full Adder:

1. What is the primary purpose of a full adder in digital circuits?
2. Explain the truth table of a full adder, including its inputs and outputs.
3. Describe the logic diagram or circuit for a full adder.
4. How does a full adder handle binary addition compared to a half adder?
5. What is the significance of the carry-in (Cin) input in a full adder?

6. Can you provide an example of a real-world application where full adders are used?
7. Discuss the relationship between full adders and half adders in multi-bit addition.
8. How can you cascade full adders to add multi-bit binary numbers?
9. What is the difference between the carry-out (Cout) of a full adder and the carry-in (Cin) of the next stage?
10. What are the challenges associated with designing efficient full adders for high-speed operations?"

Half Subtractor:

1. What is the primary function of a half subtractor in digital circuits?
2. Describe the truth table of a half subtractor, including its inputs and outputs.
3. Explain the logic diagram or circuit for a half subtractor.
4. How is binary subtraction carried out in a half subtractor?
5. What is the significance of the borrow-out (Bout) output in a half subtractor?
6. Can you provide an example of a practical application where half subtractors are used?
7. Discuss the differences between half adders and half subtractors.
8. How can you cascade half subtractors to subtract multi-bit binary numbers?
9. Explain the concept of borrow propagation in half subtractors.
10. What challenges might arise when using half subtractors for complex subtraction operations?"

Full Subtractor:

1. What is the primary role of a full subtractor in digital circuits?
2. Describe the truth table of a full subtractor, including its inputs and outputs.
3. Explain the logic diagram or circuit for a full subtractor.
4. How does a full subtractor perform binary subtraction compared to a half subtractor?
5. What are the uses of the borrow-in (Bin) input in a full subtractor?
6. Can you provide an example of a real-world application where full subtractors are used?
7. Compare the differences between full adders and full subtractors.
8. How can you cascade full subtractors to subtract multi-bit binary numbers?
9. Discuss borrow propagation and borrow generation in full subtractors.
10. What challenges may arise when implementing full subtractors for complex subtraction operations?"

5 Design Verilog HDL to implement Decimal or BCD adder.

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

SIMULATOR USED: ISE Design suite 14.2

5.2 THEORY:

A Binary Coded Decimal (BCD) Adder is a digital circuit that performs addition operations on Binary Coded Decimal numbers, which are a way of representing decimal digits (0-9) using 4-bit binary numbers.

BCD Representation:

In BCD representation, each decimal digit (0-9) is represented by a 4-bit binary code. For example, decimal 0 is represented as 0000, decimal 1 as 0001, and so on, up to decimal 9, which is represented as 1001.

BCD Addition:

BCD addition is the process of adding two BCD numbers together, producing a BCD result. BCD addition involves performing binary addition while taking into account carry propagation between the digits. It ensures that the result remains in BCD format.

BCD Adder Components:

A BCD adder typically consists of several 4-bit binary adders, each designed to add two 4-bit BCD digits and generate a 4-bit BCD sum. These 4-bit binary adders are used to handle the addition of individual digits, and the carry generated by each digit's addition is propagated to the next higher-order digit.

Carry Propagation in BCD Addition:

The key aspect of BCD addition is carry propagation. When a BCD digit addition results in a carry, this carry must be propagated to the next higher-order BCD digit. For example, when adding 9 (1001) and 5 (0101), the addition of the ones digit (1+1) generates a carry of 1, which needs to be added to the tens digit.

In summary, a BCD adder is a digital circuit that performs addition operations on Binary Coded Decimal numbers, allowing for accurate and efficient decimal arithmetic. It involves performing binary addition while managing carry propagation to ensure the result remains in

BCD format. BCD adders find applications in various fields, particularly in scenarios where decimal numbers need to be processed and displayed accurately.

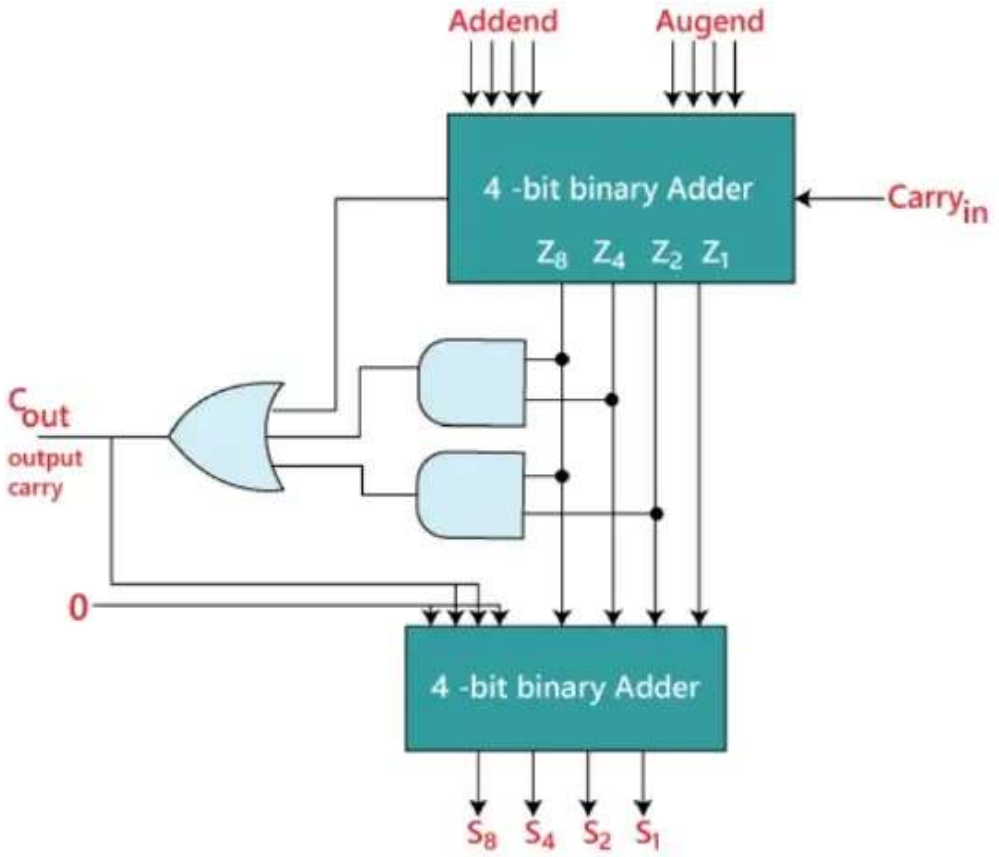


Figure 5.1: Logic circuit of BCD adder

Table 5.1 Truth Table of BCD adder

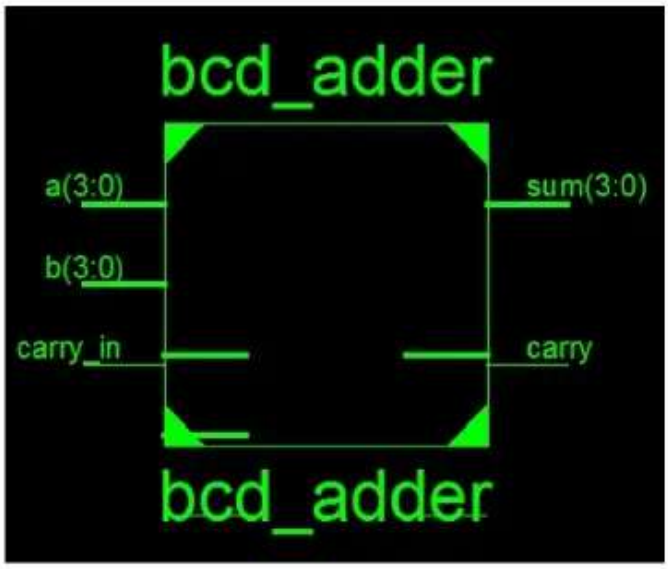
Binary Sum						BCD Sum						Decimal
K	Z_8	Z_4	Z_2	Z_1		C	S_8	S_4	S_2	S_1		
0	0	0	0	0	S	0	0	0	0	0		0
0	0	0	0	1	A	0	0	0	0	1		1
0	0	0	1	0	M	0	0	0	1	0		2
.	.	.	.	.	E	.	.	.	.	.		.
.	.	.	.	.		.	.	.	.	.		.
.	.	.	.	.	C	.	.	.	.	.		.
.	.	.	.	.	O	.	.	.	.	.		.
0	1	0	0	0	D	0	1	0	0	0		8
0	1	0	0	1	E	0	1	0	0	1		9
10 to 19 Binary and BCD codes are not the same												
0	1	0	1	0		1	0	0	0	0		10
0	1	0	1	1		1	0	0	0	1		11
0	1	1	0	0		1	0	0	1	0		12
0	1	1	0	1		1	0	0	1	1		13
0	1	1	1	0		1	0	1	0	0		14
0	1	1	1	1		1	0	1	0	1		15
1	0	0	0	0		1	0	1	1	0		16
1	0	0	0	1		1	0	1	1	1		17
1	0	0	1	0		1	1	0	0	0		18
1	0	0	1	1		1	1	0	0	1		19

5.3 PROCEDURE:

Step 1: Write the Verilog code for BCD adder and obtain the RTL schematic

```
//module declaration with inputs and outputs
module bcd_adder(a,b,carry_in,sum,carry);
//declare the inputs and outputs of the module with their sizes.
   input [3:0] a,b;
   input carry_in;
   output [3:0] sum;
   output carry;
   //Internal variables
   reg [4:0] sum_temp;
```

```
    reg [3:0] sum;
    reg carry;
//always block for doing the addition
    always @(a,b,carry_in)
    begin
        sum_temp = a+b+carry_in; //add all the inputs
        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
```



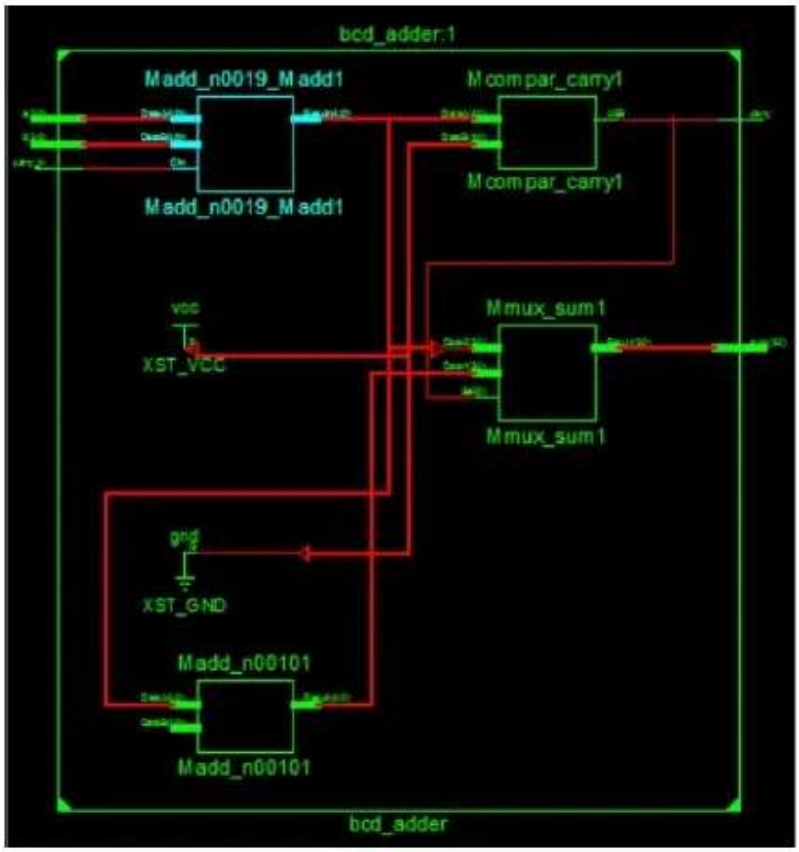


Figure 5.2 RTL Schematic for BCD adder

Step 2: Write the testbench code for BCD adder and obtain the RTL simulation waveform

Testbench code:

Figure 5.3 RTL Simulation waveform for BCD Adder

5.4 APPLICATIONS:

BCD adders are used in applications where decimal arithmetic is required, such as calculators, digital clocks, and some specialized data processing systems.

1. **Digital Calculators:** BCD adders are used in calculators to perform addition and subtraction operations in decimal format, providing accurate and reliable results.
2. **Digital Clocks:** BCD adders help maintain and update the time in digital clocks by adding seconds, minutes, hours, and other time components.
3. **Data Conversion:** BCD adders are used in data conversion applications where binary data needs to be converted to BCD format or vice versa.
4. **Numeric Displays:** BCD adders are used to update numeric displays, such as 7-segment displays, to show the correct BCD representation of a number.

5.5 EXPECTED VIVA QUESTIONS:

1. What does BCD stand for, and how is it different from binary representation?
2. Explain the BCD representation of decimal digits (0-9) using 4-bit binary codes.
3. How does BCD addition differ from binary addition?
4. Can you walk me through the process of adding two BCD numbers using a BCD adder?
5. What is the significance of carry propagation in BCD addition?
6. Why is it important to ensure that the result of a BCD addition remains in BCD format?
7. What are the key components or building blocks in a BCD adder?
8. How is a 4-bit binary adder used in a BCD adder?
9. Describe the role of each 4-bit binary adder in a BCD adder.
10. How does carry propagation work in a BCD adder when adding BCD digits?
11. Can you provide an example of BCD addition where carry propagation occurs?
12. What happens if there is a carry-out from the most significant BCD digit during addition?
13. Discuss the practical applications of BCD adders in digital systems.
14. How are BCD adders used in digital calculators and numeric displays?
15. Explain the role of BCD adders in maintaining and updating digital clocks.
16. In what scenarios would you need to convert binary data to BCD format, and how can a BCD adder be useful in this context?
17. How do BCD adders facilitate data processing and conversion in digital systems?
18. Discuss the trade-offs and challenges in designing efficient BCD adders for high-speed applications.

6 Design Verilog program to implement Different types of multiplexer like 2:1, 4:1 and 8:1.

6.1 AIM: To design and implement Different types of multiplexer like 2:1, 4:1 and 8:1 using verilog HDL.

Simulator used: ISE Design suite 14.2

Mutliplexer

A multiplexer (MUX) is a combinational circuit that connects any one input line (out of multiple N lines) to the single output line based on its control input signal (or selection lines).Usually, for 'n' selection lines, there are N = 2^n input lines.N:1 denotes it has 'N' input lines and one output line.

6.2 2:1 Multiplexer

6.2.1 Theory

A 2:1 multiplexer (mux) is a digital logic circuit that has two data inputs, one control input, and one output(i.e 2:1 MUX has 2 input lines and one select line.). It selects one of the two data inputs based on the state of the control input. The term "2:1" indicates the ratio of data inputs to the multiplexer, where there are two data inputs (I0 and I1) and one output (Y).

1. Data Input I0 is selected when the control input (sel) is LOW (0).
2. Data Input I1 is selected when the control input (sel) is HIGH (1).

The selected data input is transmitted to the output, allowing you to switch between two data sources based on the value of the control input. 2:1 multiplexers are fundamental building blocks in digital design and are used in various applications, such as data routing, signal selection, and control logic in digital systems. They provide a way to select and pass data from one of the two inputs to the output, making them versatile components in digital circuits.

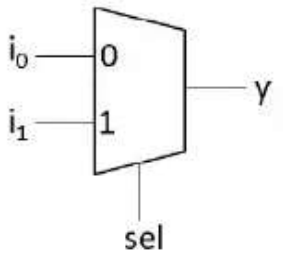


Figure 6.1 Block Diagram of 2:1 Mux

Table 6.1 Truth table for 2:1 multiplexer

sel	y
0	I_0
1	I_1

6.2.2 PROCEDURE:

Step 1: Write the Verilog code for 2:1 Mux and obtain the RTL schematic

```
module mux_2_1( input sel, input i0, i1, output y);
   assign y = sel ? i1 : i0;
endmodule
```

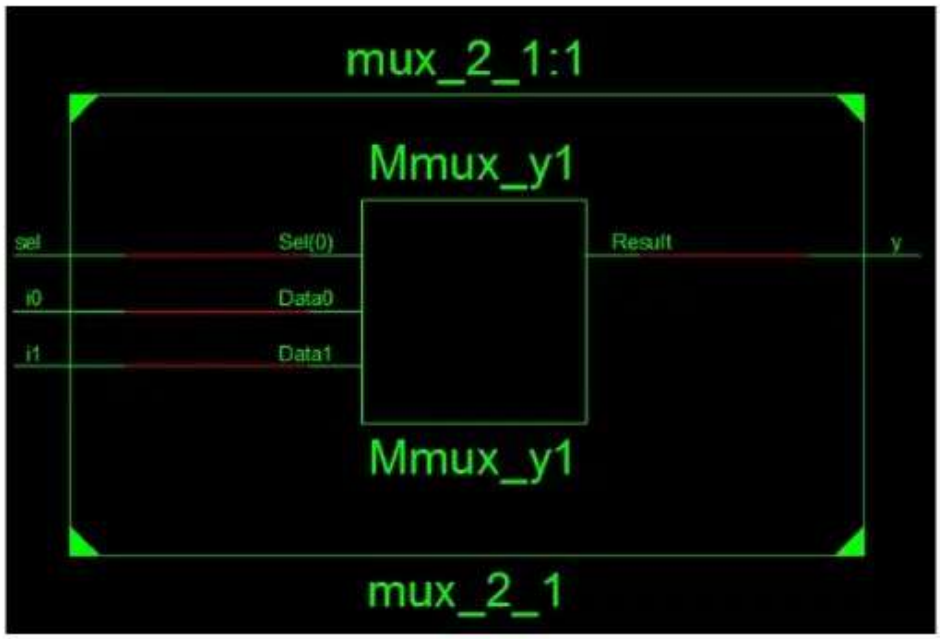


Figure 6.2 RTL Schematic for 2:1 multiplexer

Step 2: Write the testbench code for 2:1 Mux and obtain the RTL simulation waveform

```
module mux_tb;
 reg i0, i1, sel;
 wire y;
 mux_2_1 mux(sel, i0, i1, y);
 initial begin
  $monitor("sel = %h: i0 = %h, i1 = %h --> y = %h", sel, i0, i1, y);
  i0 = 0; i1 = 1;
  sel = 0;
  #1;
  sel = 1;
 end
endmodule
```

Figure 6.3 Output Window for 2:1 multiplexer

Step 3: Get the result from the output window

```
sel = 0: i0 = 0, i1 = 1 --> y = 0
sel = 1: i0 = 0, i1 = 1 --> y = 1
```

6.3 4:1 MULTIPLEXER

6.3.1 THEORY:

A 4:1 multiplexer (mux) is a digital logic circuit that has four data inputs, two control inputs, and one output(i.e 4:1 has 4 input lines and two select lines). It selects one of the four data inputs based on the combination of control inputs. The term "4:1" indicates the ratio of data inputs to the multiplexer, where there are four data inputs (I0, I1, I2, and I3) and one output (Y).

The combination of two control inputs, typically labeled as Sel[0] and Sel[1], determines which of the four data inputs is transmitted to the output. The two control inputs can take on four possible combinations (00, 01, 10, and 11), and each combination selects one of the four data inputs.

For example, with S0 and S1 as control inputs:

1. When Sel[1] = 0 and Sel[0] = 0, the mux selects Data Input I0 to pass to the output.
2. When Sel[1]= 0 and Sel[0] = 1, the mux selects Data Input I1.
3. When Sel[1]= 1 and Sel[0] = 0, the mux selects Data Input I2.
4. When Sel[1] = 1 and Sel[0] = 1, the mux selects Data Input I3.

4:1 multiplexers are commonly used in digital design and serve various purposes, including data routing, signal selection, and control logic in digital systems. They enable the selection of one of four data sources based on the values of the control inputs, making them valuable components in digital circuits.

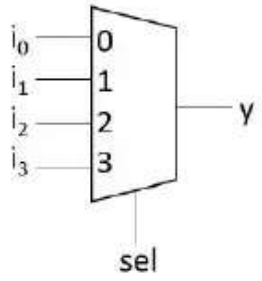


Figure 6.4 Block Diagram of 4:1 Mux

Table 6.2 Truth table for 4:1 multiplexer

Sel[0]	Sel[1]	y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

6.3.2 PROCEDURE:

Step 1: Write the Verilog code for 4:1 Mux and obtain the RTL schematic

Verilog code for 4:1 multiplexer

```
module mux_example( input [1:0] sel, input i0,i1,i2,i3, output reg y);
  always @(*) begin
  case(sel)
    2'h0: y = i0;
    2'h1: y = i1;
    2'h2: y = i2;
    2'h3: y = i3;
    default: $display("Invalid sel input");
  endcase
 end
endmodule
```

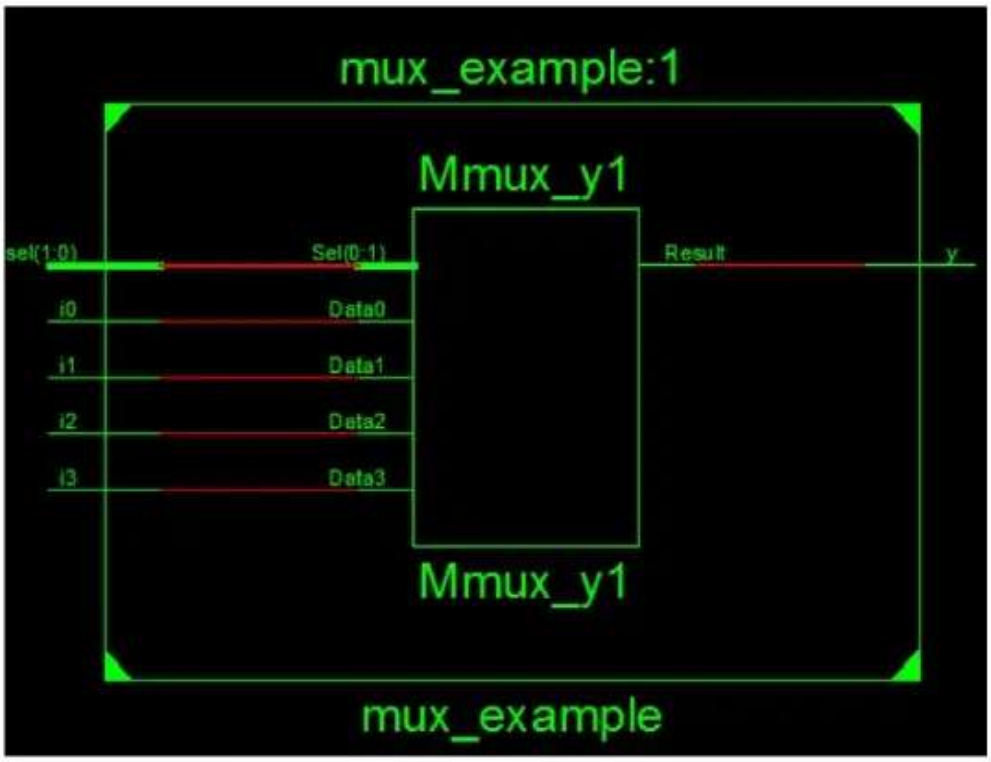


Figure 6.5 RTL Schematic for 4:1 multiplexer

Step 2: Write the testbench code for 4:1 Mux and obtain the RTL simulation waveform

Testbench code:

```
module tb;
 reg [1:0] sel;
 reg i0,i1,i2,i3;
 wire y;
  mux_example mux(sel, i0, i1, i2, i3, y);
 initial begin
  $monitor("sel = %b -> i3 = %0b, i2 = %0b ,i1 = %0b, i0 = %0b -> y = %0b", sel,i3,i2,i1,i0,
y);
  {i3,i2,i1,i0} = 4'h5;
  repeat(6) begin
    sel = $random;
    #5;
  end
 end
endmodule
```

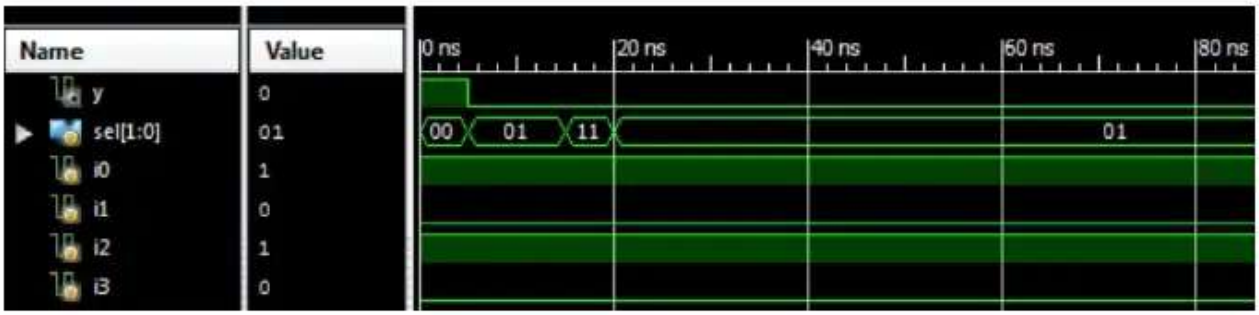


Figure 6.6 Output Window for 4:1 multiplexer

Step 3: Get the result from the output window

sel = 00 -> i3 = 0, i2 = 1 ,i1 = 0, i0 = 1 -> y = 1

sel = 01 -> i3 = 0, i2 = 1 ,i1 = 0, i0 = 1 -> y = 0

sel = 11 -> i3 = 0, i2 = 1 ,i1 = 0, i0 = 1 -> y = 0

sel = 01 -> i3 = 0, i2 = 1 ,i1 = 0, i0 = 1 -> y = 0

6.4 8:1 MULTIPLEXER

6.4.1 THEORY

An 8:1 multiplexer (mux) is a digital logic circuit that has eight data inputs, three control inputs, and one output (i.e 8:1 has 8 input lines and three select lines). It selects one of the eight data inputs based on the combination of control inputs. The term "8:1" indicates the ratio of data inputs to the multiplexer, where there are eight data inputs (D0, D1, D2, D3, D4, D5, D6, and D7) and one output (Y).

The combination of three control inputs, typically labeled as Sel[2], Sel[1], and Sel[0], determines which of the eight data inputs is transmitted to the output.

The three control inputs can take on eight possible combinations (000, 001, 010, 011, 100, 101, 110, and 111), and each combination selects one of the eight data inputs.

For example, with S2, S1, and S0 as control inputs:

When Sel[2] = 0, Sel[1] = 0, and Sel[0]= 0, the mux selects Data Input D0 to pass to the output.

When Sel[2] = 0, Sel[1] = 0, and Sel[0]= 1, the mux selects Data Input D1.

When Sel[2]= 0, Sel[1] = 1, and Sel[0] = 0, the mux selects Data Input D2.

And so on, with the other combinations determining which data input is selected.

8:1 multiplexers are fundamental components in digital design and are used in various applications, including data routing, signal selection, and control logic in digital systems. They allow you to select one of eight data sources based on the values of the control inputs, making them valuable in complex digital circuits.

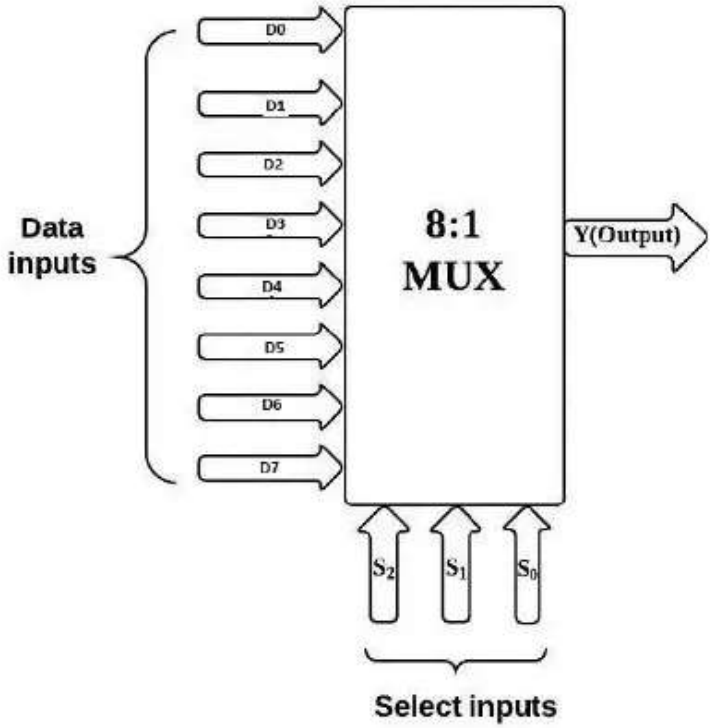


Figure 6.7 Block Diagram of 8:1 Mux

Table 6.3 Truth table for 8:1 multiplexer

Sel[2]	Sel[1]	Sel[0]	y
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

6.4.2 PROCEDURE:

Step 1: Write the Verilog code for 8:1 Mux and obtain the RTL schematic

```
module Mulitplexer(d0,d1,d2,d3,d4,d5,d6,d7,sel,out);
input d0,d1,d2,d3,d4,d5,d6,d7;
input [2:0] sel;
```

```
output reg out;
always@(sel)
begin
case(sel)
3'b000:out=d0;
3'b001:out=d1;
3'b010:out=d2;
3'b011:out=d3;
3'b100:out=d4;
3'b101:out=d5;
3'b110:out=d6;
3'b111:out=d7;
endcase
end
endmodule
```

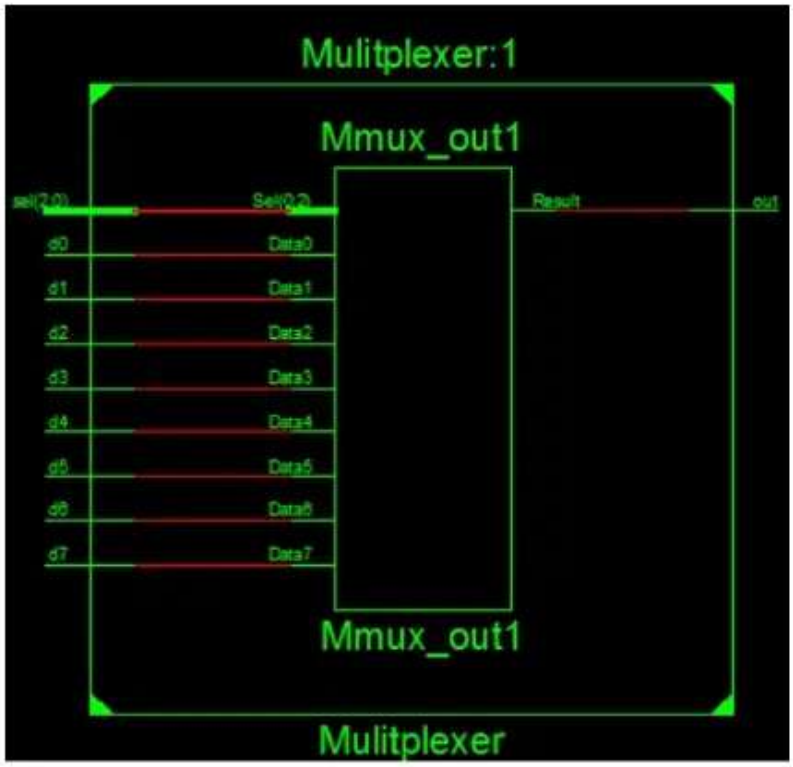


Figure 6.8 RTL Schematic for 8:1 multiplexer

Step 2: Write the testbench code for 8:1 Mux and obtain the RTL simulation waveform

Testbench code:

```
module TestModule;
// Inputs
reg d0;
reg d1;
reg d2;
reg d3;
reg d4;
reg d5;
reg d6;
reg d7;
reg [2:0] sel;
// Outputs
wire out;
// Instantiate the Unit Under Test (UUT)
Mulitplexer uut (
.d0(d0),
.d1(d1),
.d2(d2),
.d3(d3),
.d4(d4),
.d5(d5),
.d6(d6),
.d7(d7),
.sel(sel),
.out(out)
);
initial begin
// Initialize Inputs
d0 = 0;
```

```
d1 = 0;
d2 = 0;
d3 = 0;
d4 = 0;
d5 = 0;
d6 = 0;
d7 = 0;
sel = 0;
// Wait 100 ns for global reset to finish
#100;
d0 = 0;
d1 = 0;
d2 = 0;
d3 = 0;
d4 = 1;
d5 = 1;
d6 = 0;
d7 = 1;
sel = 5;
// Wait 100 ns for global reset to finish
#100;
// Add stimulus here
end
endmodule
```

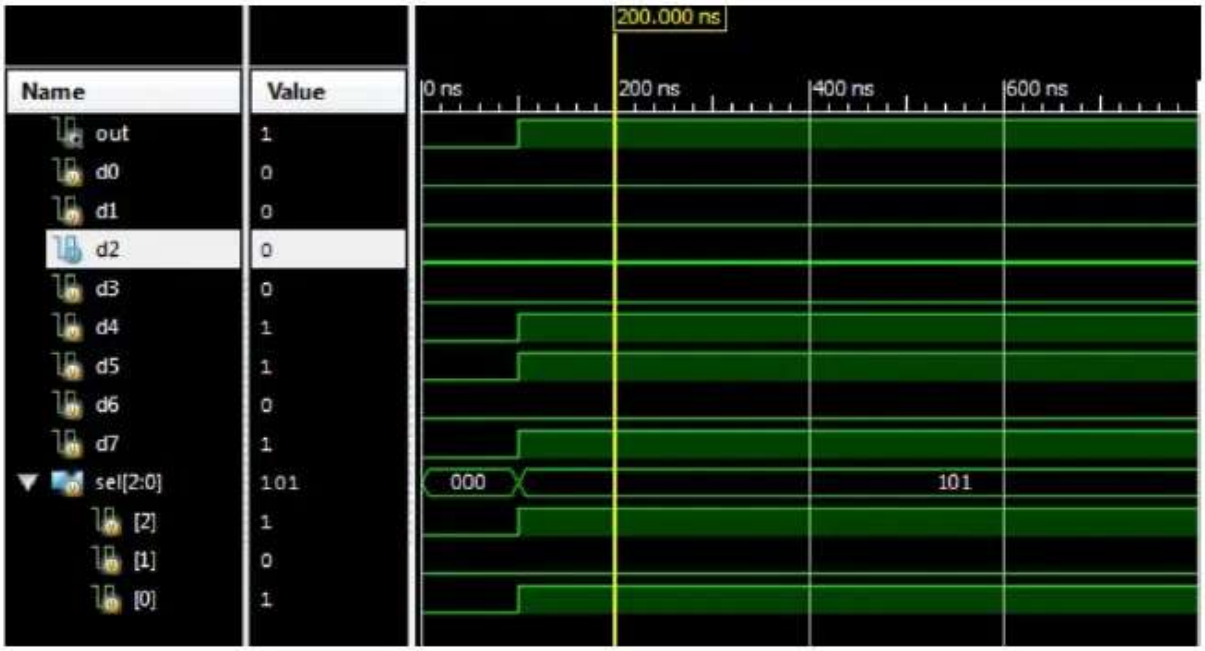


Figure 6.9 RTL simulation waveform for 8:1 multiplexer

6.5 APPLICATION OF MUX:

MUXes are versatile components that find numerous applications, including:

1. **Data Routing**: MUXes are often used for data routing in digital systems. They can select one of many input sources and route the data to the desired destination. For example, in a computer system, a MUX can be used to select data from various peripherals and send it to the CPU for processing.
2. **Memory Addressing:** In memory systems, MUXes are used to select a specific memory location or register based on the address input. This is essential for reading and writing data in memory or accessing specific registers in a microcontroller or microprocessor.
3. **ALU Operations:** Arithmetic Logic Units (ALUs) in microprocessors use MUXes to select the operation to be performed (addition, subtraction, AND, OR, etc.) based on control signals. This allows the ALU to perform a wide range of arithmetic and logic operations.
4. **Control Logic:** MUXes are used in control units of microprocessors to make decisions and control various aspects of the processor's operation. They can be used to select different control signals based on the current state or instruction being executed.

5. **Data Compression:** MUXes can be used in data compression algorithms to select the most significant bits of data for compression, reducing the amount of data to be transmitted or stored.
6. **Signal Multiplexing:** In communication systems, MUXes are used to combine multiple input signals onto a single communication channel for transmission, and at the receiving end, they are used to select and separate the desired signals.
7. **Display Controllers**: In display controllers, MUXes can be used to control various aspects of display operations, such as selecting the source of display data, choosing which portion of the display to update, and managing display refresh rates.
8. **Test and Measurement Equipment**: MUXes are essential components in test and measurement equipment, enabling the selection of different test signals for analysis and measurement purposes.
9. **Analog-to-Digital Conversion:** MUXes are used in analog-to-digital converters (ADCs) to select one of several analog input channels for digitization. This allows for the conversion of multiple analog signals into digital form.
10. **Digital Audio and Video Switching**: In audio and video systems, MUXes are used for selecting audio or video inputs, routing signals to various outputs, and managing switching between different sources.
11. **Security Systems:** MUXes are employed in security systems for sensor selection and monitoring. They can select different sensors (e.g., motion detectors, cameras) to activate or monitor based on security conditions.

These applications demonstrate the versatility and importance of MUXes in digital systems, where they are used to efficiently manage data, control, and signal routing, making them a critical component in various electronic devices and systems.

6.6 EXPECTED VIVA QUESTIONS:

2:1 MUX:

What is a 2:1 MUX, and what does the "2:1" ratio signify?

Can you describe the typical symbol or representation of a 2:1 MUX?

Explain the purpose of a select (or control) input in a 2:1 MUX.

How many data inputs and how many control inputs does a 2:1 MUX have?

Walk me through the truth table for a 2:1 MUX.

What happens when the select input is 0 or 1 in a 2:1 MUX?

Provide an example of a real-world application where a 2:1 MUX is used.

How can you use a 2:1 MUX to implement a logic function or create a signal switch?

4:1 MUX:

Explain the concept of a 4:1 MUX, including the meaning of the "4:1" ratio.

What is the typical symbol for a 4:1 MUX, and how is it different from the 2:1 MUX symbol?

How many data inputs and how many control inputs are found in a 4:1 MUX?

Can you provide the truth table for a 4:1 MUX?

Describe a situation where you might use a 4:1 MUX to perform signal selection.

How does a 4:1 MUX differ from cascading two 2:1 MUXes to achieve the same result?

What role does a 4:1 MUX play in data routing and signal selection in digital systems?

8:1 MUX:

Define an 8:1 MUX and explain the significance of the "8:1" ratio.

Illustrate the symbol or representation of an 8:1 MUX.

How many data inputs and control inputs are present in an 8:1 MUX?

Could you provide the truth table for an 8:1 MUX?

Give an example of a practical scenario where an 8:1 MUX is employed.

Explain how you can use an 8:1 MUX to simplify complex logic functions.

What benefits does an 8:1 MUX offer in terms of signal selection and data routing in digital systems?

7 Design Verilog program to implement types of De-Multiplexer.

7.1 AIM: To design and implement different types of De-Mux using Verilog HDL

Simulator used: ISE Design Suite 14.2

Definition:

A De-Multiplexer (DEMUX) is a combinational circuit that works exactly opposite to a multiplexer. A DEMUX has a single input line that connects to any one of the output lines based on its control input signal (or selection lines). Usually, for 'n' selection lines, there are N = 2^n output lines. 1:N denotes one input line and 'N' output lines.

7.2 1:2 DE-MULTIPLEXER

7.2.1 THEORY

A 1:2 demux takes a single input and directs it to one of two possible outputs based on the value of its control input (i.e 1:2 De-Mux has one select line and 2 output lines).The control input specifies which of the two output lines the input signal is routed to.

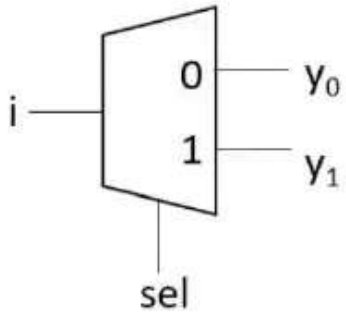


Figure 7.1 Block Diagram of 1:2 De-Mux

Table 7.1 Truth table for 1:2 De-Mux

Sel	Y_0	Y_1
0	i	0
1	0	i

7.2.2 PROCEDURE:

Step 1: Write the Verilog code for 1:2 De-Mux and obtain the RTL schematic

module demux_2_1(

```
 input sel,
 input i,
 output y0, y1);
 assign {y0,y1} = sel?{1'b0,i}: {i,1'b0};
endmodule
```

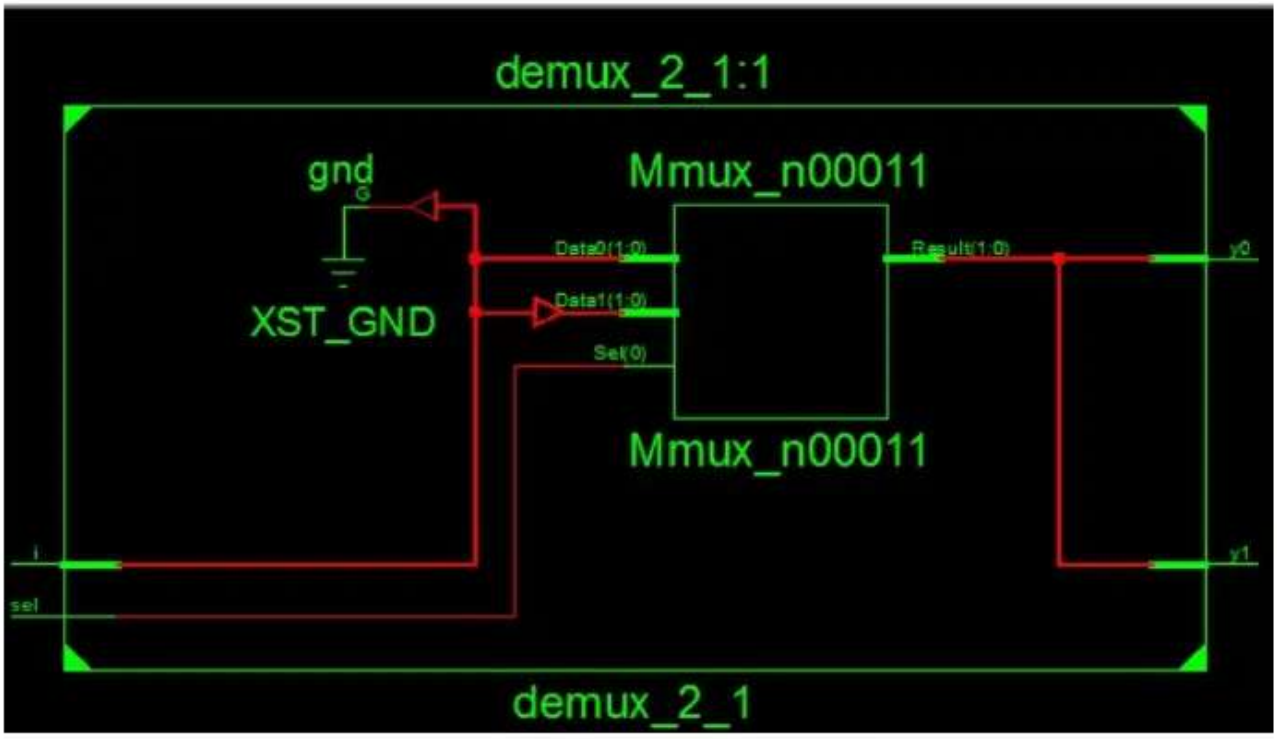


Figure 7.2 RTL Schematic for 1:2 DeMux

Step 2: Write the testbench code for 1:2 De-MUX and obtain the RTL simulation waveform

```
module demux_tb;
 reg sel, i;
 wire y0, y1;
 demux_2_1 demux(sel, i, y0, y1);
 initial begin
  $monitor("sel = %h: i = %h --> y0 = %h, y1 = %h", sel, i, y0, y1);
  sel=0; i=0; #1;
  sel=0; i=1; #1;
  sel=1; i=0; #1;
```

```
  sel=1; i=1; #1;
 end
endmodule
```

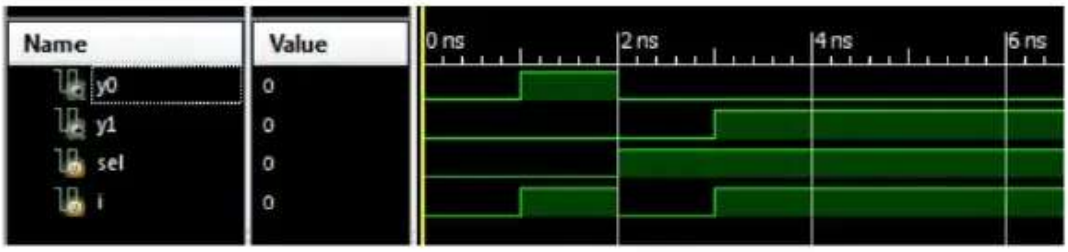


Figure 7.3 RTL Simulation window for 1:2 DeMux

Step 3: Get the result from the output window

sel = 0: i = 0 --> y0 = 0, y1 = 0

sel = 0: i = 1 --> y0 = 1, y1 = 0

sel = 1: i = 0 --> y0 = 0, y1 = 0

sel = 1: i = 1 --> y0 = 0, y1 = 1

7.3 1:4 Demultiplexer

7.3.1 Theory

A 1:4 demux takes a single input and directs it to one of four possible outputs based on the combination of its control inputs (1:4 De-Mux has one select line and 4 output lines).The combination of control inputs (typically two control lines) specifies which of the four output lines the input signal is routed to.

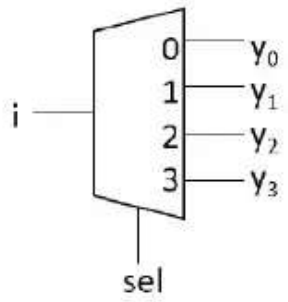


Figure 7.4 Block Diagram of 1:4 De-Mux

Table 7.2 Truth table for 1:4 De-Mux

Sel[0]	Sel[1]	Y_0	Y_1	Y_2	Y_3
0	0	i	0	0	0
0	1	0	i	0	0
1	0	0	0	i	0
1	1	0	0	0	i

7.3.2 PROCEDURE:

Step 1: Write the Verilog code for 1:4 De-Mux and obtain the RTL schematic

```
module demux_1_4(
 input [1:0] sel,
 input  i,
 output reg y0,y1,y2,y3);
 always @(*) begin
  case(sel)
   2'h0: {y0,y1,y2,y3} = {i,3'b0};
   2'h1: {y0,y1,y2,y3} = {1'b0,i,2'b0};
   2'h2: {y0,y1,y2,y3} = {2'b0,i,1'b0};
   2'h3: {y0,y1,y2,y3} = {3'b0,i};
   default: $display("Invalid sel input");
  endcase
 end
endmodule
```

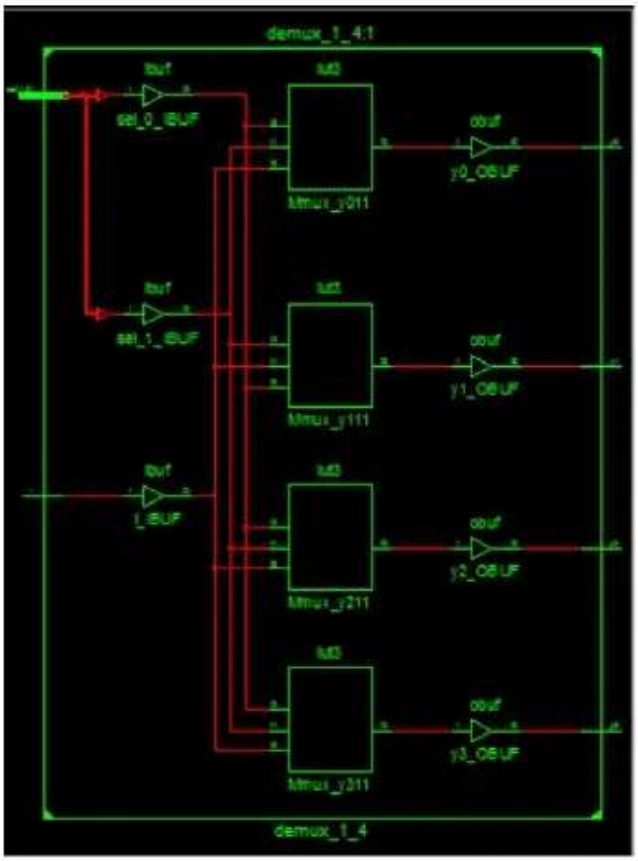


Figure 7.5 RTL Schematic for 1:4 De-Mux

Step 2: Write the testbench code for 1:4 De-MUX and obtain the RTL simulation waveform

```
module tb;
 reg [1:0] sel;
 reg i;
 wire y0,y1,y2,y3;
 demux_1_4 demux(sel, i, y0, y1, y2, y3);
 initial begin
  $monitor("sel = %b, i = %b -> y0 = %0b, y1 = %0b ,y2 = %0b, y3 = %0b", sel,i, y0,y1,y2,y3);
  sel=2'b00; i=0; #1;
  sel=2'b00; i=1; #1;
  sel=2'b01; i=0; #1;
  sel=2'b01; i=1; #1;
  sel=2'b10; i=0; #1;
```

```
   sel=2'b10; i=1; #1;
   sel=2'b11; i=0; #1;
   sel=2'b11; i=1; #1;
 end
endmodule
```

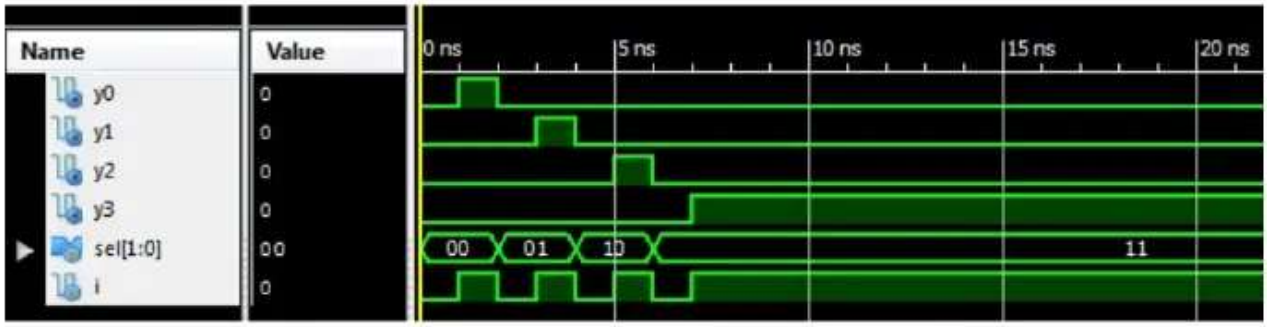


Figure 7.6 RTL Simulation window for 1:4 De-Mux

Step 3: Get the result from the output window

sel = 00, i = 0 -> y0 = 0, y1 = 0 ,y2 = 0, y3 = 0

sel = 00, i = 1 -> y0 = 1, y1 = 0 ,y2 = 0, y3 = 0

sel = 01, i = 0 -> y0 = 0, y1 = 0 ,y2 = 0, y3 = 0

sel = 01, i = 1 -> y0 = 0, y1 = 1 ,y2 = 0, y3 = 0

sel = 10, i = 0 -> y0 = 0, y1 = 0 ,y2 = 0, y3 = 0

sel = 10, i = 1 -> y0 = 0, y1 = 0 ,y2 = 1, y3 = 0

sel = 11, i = 0 -> y0 = 0, y1 = 0 ,y2 = 0, y3 = 0

sel = 11, i = 1 -> y0 = 0, y1 = 0 ,y2 = 0, y3 = 1

7.4 1:8 DEMULTIPLEXER

7.4.1 THEORY:

A 1:8 demux takes a single input and directs it to one of eight possible outputs based on the combination of its control inputs (1:8 De-Mux has one select line and 8 output lines).The combination of control inputs (typically three control lines) specifies which of the eight output lines the input signal is routed to.

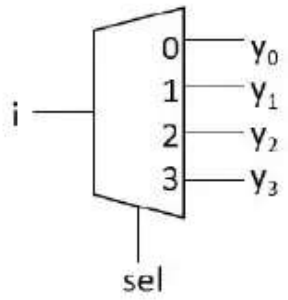


Figure 7.7 Block Diagram of 1:8 De-Mux

Table 7.3 Truth table for 1:8 De-Mux

A	S_2	S_1	S_0	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
1	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	0	0	0
1	0	1	0	0	0	1	0	0	0	0	0
1	0	1	1	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	1	0	0	0
1	1	0	1	0	0	0	0	0	1	0	0
1	1	1	0	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	0	0	1

7.4.2 PROCEDURE:

Step 1: Write the Verilog code for 1:8 De-MUX and obtain the RTL schematic

```
module demux_1_8(y,s,a);
output reg [7:0]y;
input [2:0]s;
input a;
always @(*)
begin
y=0;
case(s)
3'd0: y[0]=a;
3'd1: y[1]=a;
3'd2: y[2]=a;
```

```
3'd3: y[3]=a;
3'd4: y[4]=a;
3'd5: y[5]=a;
3'd6: y[6]=a;
3'd7: y[7]=a;
endcase
end
endmodule
```

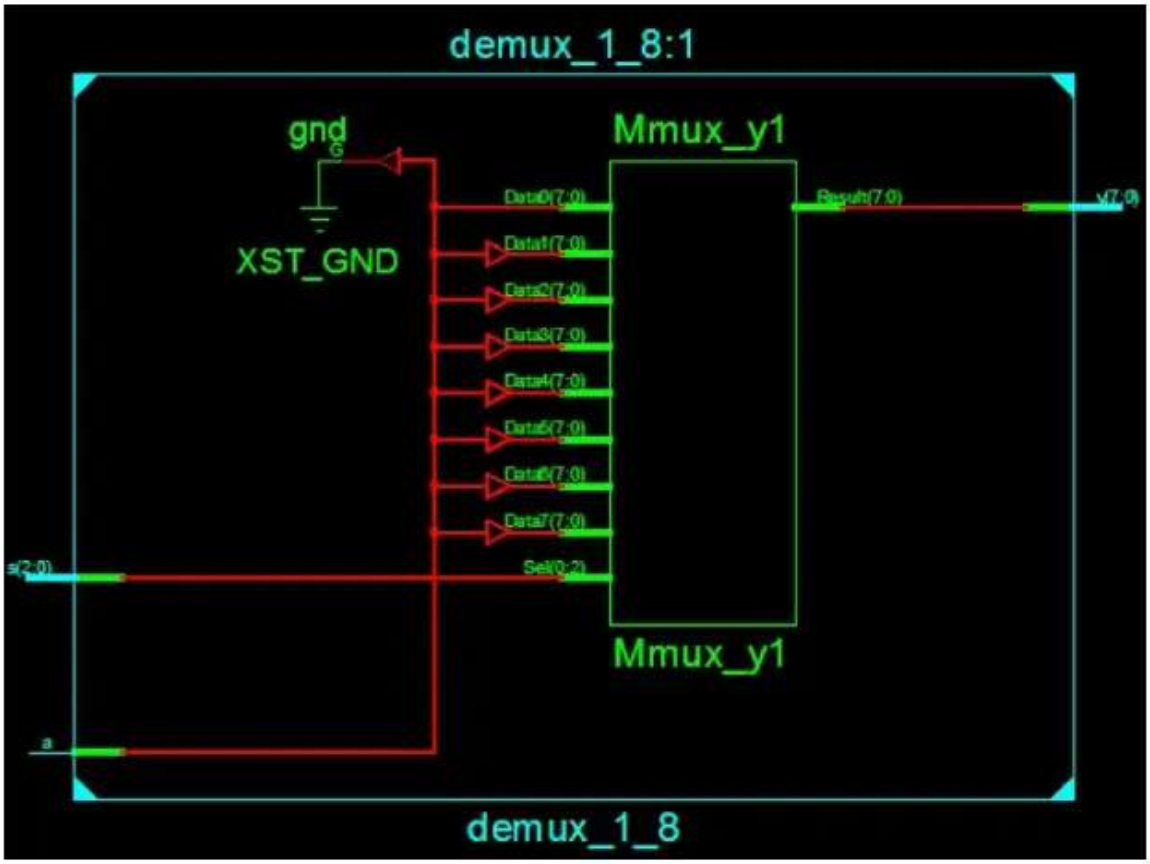


Figure 7.8 RTL Schematic for 1:8 De-Mux

Step 2: Write the testbench code for 1:8 De-Mux and obtain the RTL simulation waveform

```
module test_demux;
reg [2:0]S;
reg A;
wire [7:0]Y;
```

```
demux_1_8 mydemux(.y(Y), .a(A), .s(S));
initial begin
A=1;
S=3'd5;
#30;
A=0;
S=3'd1;
#30;
A=1;
S=3'd1;
#30;
S=3'd6;
#30;
S=3'd0;
#30;
end
endmodule
```

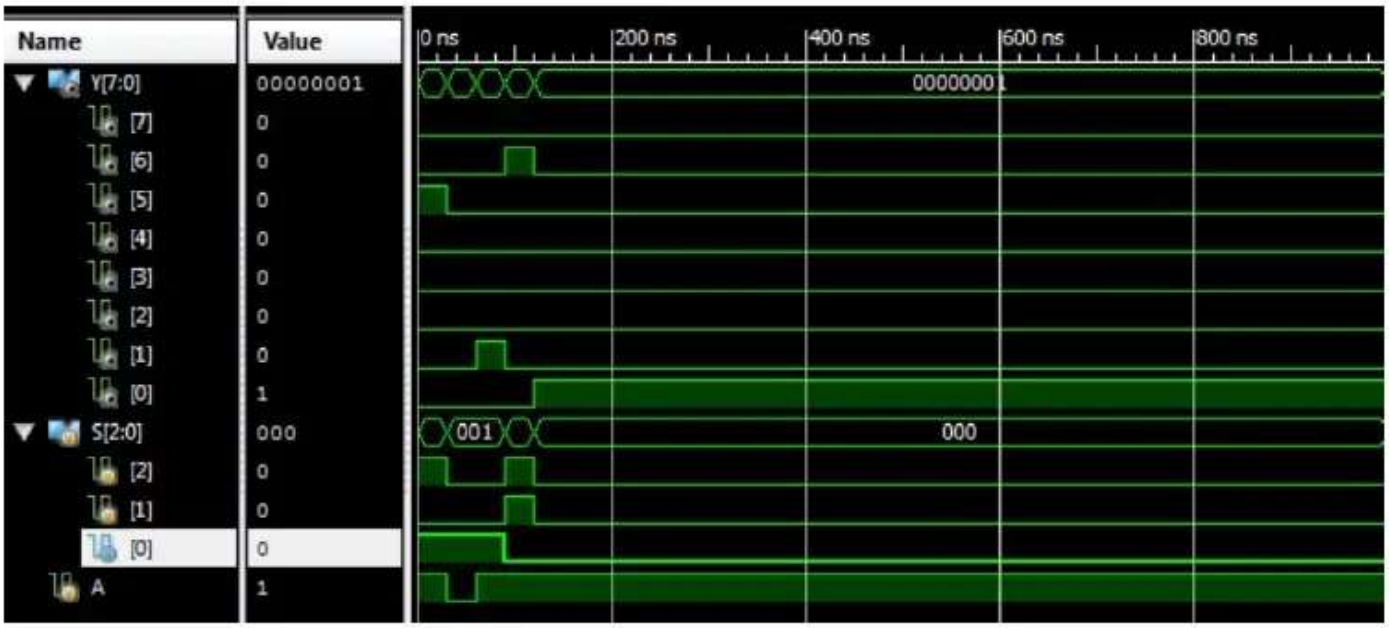


Figure 7.9 RTL Simulation window for 1:8 De-Mux

7.5 APPLICATIONS:

Demultiplexers have a wide range of applications in digital electronics and communication systems. Here are some common applications:

1. **Data Distribution:** One of the primary uses of a Demux is data distribution. It can take a single input signal and distribute it to multiple output channels, allowing for data transmission to various destinations. This is commonly used in data buses and communication networks.
2. **Digital Demodulation:** In digital communication systems, Demuxes are used to demodulate signals. For example, in demodulating frequency-shift keying (FSK) signals, a Demux can separate the incoming signal into its constituent frequency components, which represent different binary values.
3. **Memory Access:** Demuxes are used in memory addressing. They help select a specific memory location for reading or writing data based on the address input. In this application, a Demux routes data to a specific memory cell.
4. **Display Control:** In display technologies like LED and LCD displays, Demuxes are used to control individual segments or pixels. A single input signal can be routed to select specific display elements, allowing for the display of text, graphics, or images.
5. **Audio Routing**: In audio systems, Demuxes can be used for routing audio signals to different speakers or audio channels. This allows users to control which speakers or channels receive the audio signal.
6. **Video Signal Routing:** Demuxes are used in video signal routing applications. They can route a video signal to various display devices or sections of a display.
7. **Sensor Selection**: In systems with multiple sensors, such as environmental monitoring systems, Demuxes can be used to select specific sensors for data collection. This is useful for analyzing different sensor inputs.
8. **Digital Control**: In digital control systems, Demuxes can help distribute control signals to various actuators or devices. For example, in a robotic control system, a Demux can route control signals to specific motors or components.
9. **Signal Analysis:** Demuxes are used in signal analysis and testing equipment. They can route signals to different analyzers or instruments for analysis, measurement, or troubleshooting.

10. **Telecommunications:** In telecommunications systems, Demuxes are used for demultiplexing multiplexed signals. For example, in time-division multiplexing (TDM) systems, a Demux separates the multiplexed signals into their individual channels.
11. **Digital to Analog Conversion**: In digital-to-analog conversion (DAC), a Demux can route digital data to different analog channels, enabling the generation of analog signals with specific characteristics.
12. **Digital Switching:** In digital switches, Demuxes are used to route digital signals from one source to multiple destinations, facilitating switching operations.

These applications demonstrate the versatility of Demultiplexers in digital electronics, enabling signal distribution, control, and routing in various digital systems and communication networks.

7.6 EXPECTED VIVA QUESTIONS:

1. What are the common features shared by all Demux configurations, regardless of their ratios?
2. How do Demuxes contribute to signal routing, control, and data distribution in digital systems?
3. Explain the concept of reverse operations between a Mux and a Demux.
4. Can you illustrate a scenario where a multiplexer (MUX) followed by a Demux is used in data transmission and reception?
5. What challenges or considerations are involved in designing Demuxes for high-speed data applications?

1:2 Demux:

1. What does "1:2" signify in a 1:2 Demux, and how many output lines does it have?
2. Explain the basic operation of a 1:2 Demux, including the control input.
3. What is the truth table for a 1:2 Demux?
4. Can you provide a typical symbol or representation for a 1:2 Demux?
5. Describe a real-world application where a 1:2 Demux is used.
6. How does a 1:2 Demux differ from a multiplexer (MUX) in terms of its operation?
7. What are the advantages of using a 1:2 Demux in certain scenarios?

1:4 Demux:

1. What is the significance of "1:4" in a 1:4 Demux, and how many output lines does it have?
2. Walk me through the operation of a 1:4 Demux, including the control inputs.
3. Provide the truth table for a 1:4 Demux.
4. Illustrate the symbol or representation of a 1:4 Demux.
5. Explain an application where a 1:4 Demux is employed to demultiplex data.
6. How does a 1:4 Demux compare to cascading multiple 1:2 Demuxes to achieve the same result?
7. What are the key features that make a 1:4 Demux suitable for specific tasks?

1:8 Demux:

1. What does "1:8" represent in a 1:8 Demux, and how many output lines does it have?
2. Describe the operation of a 1:8 Demux, particularly the control inputs.
3. Provide the truth table for a 1:8 Demux.
4. Show the typical symbol or representation of a 1:8 Demux.
5. Can you explain a practical application where a 1:8 Demux is utilized?
6. Discuss the advantages of using a 1:8 Demux over cascading smaller Demuxes (e.g., 1:2 or 1:4) in terms of efficiency and simplicity.

8. Design Verilog program for implementing various types of Flip-Flops such as SR, JK and D.

8.1 AIM: To design and implement various types of Flip-Flops such as SR, JK and D using Verilog HDL

Simulator used: ISE Design suite 14.2

8.2 Definition of Flipflop

A flip-flop is a fundamental digital circuit that stores binary information. It is commonly used in digital electronics and sequential logic circuits to store and control the state of a particular bit of information. There are various types of flip-flops, with the most common being the D flip-flop, JK flip-flop, SR flip-flop, and T flip-flop. Here is a general theory of flip-flops:

Basic Characteristics:

1. **State Holding:** Flip-flops can be in one of two stable states (0 or 1), which makes them suitable for storing binary information over time.
2. **Clock Signal:** Most flip-flops have a clock input that determines when they should change their state. The change usually occurs on the rising or falling edge of the clock signal.
3. **Inputs and Outputs:** Flip-flops typically have a data input (D), a clock input (CLK or CP), a reset input (R or CLR), a preset input (P or SET), and a Q and Q' (complement) output.

8.3 SR FLIPFLOP:

8.3.1 THEORY

An SR flip-flop is a fundamental bistable multivibrator circuit in digital electronics.

1. It has two inputs: Set (S) and Reset (R), and two outputs: Q and Q'.
2. The Q output is set to '1' when the S input is '1,' and it remains '1' until the R input is '1,' at which point Q becomes '0.' The output Q' is the complement of Q.

An SR flip-flop is often used for state storage and control applications.

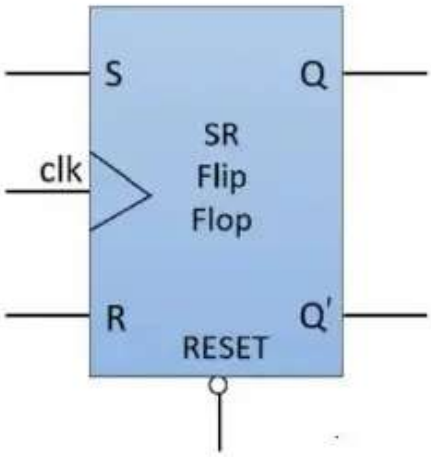


Figure 8.1: Block Diagram of SR flipflop

Table 8.1: Truth Table of SR Flipflop

Clock Edge	S	R	Q_{n+1}	Description
↓	X	X	Q_n	No Change (Store previous input)
↑	0	0	Q_n	No Change (Store previous input)
↑	1	0	1	Set Q to 1
↑	0	1	0	Reset Q to 0
↑	1	1	X	Invalid state

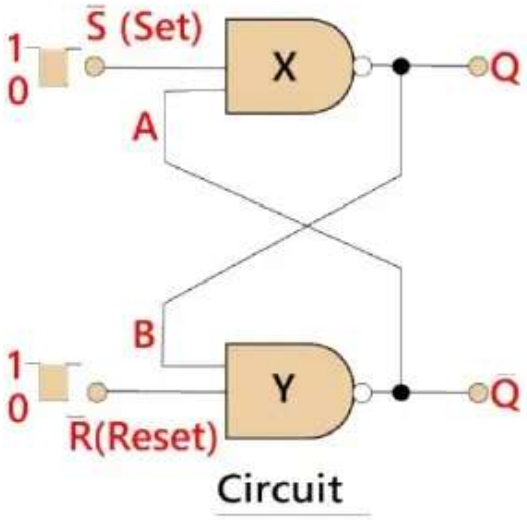
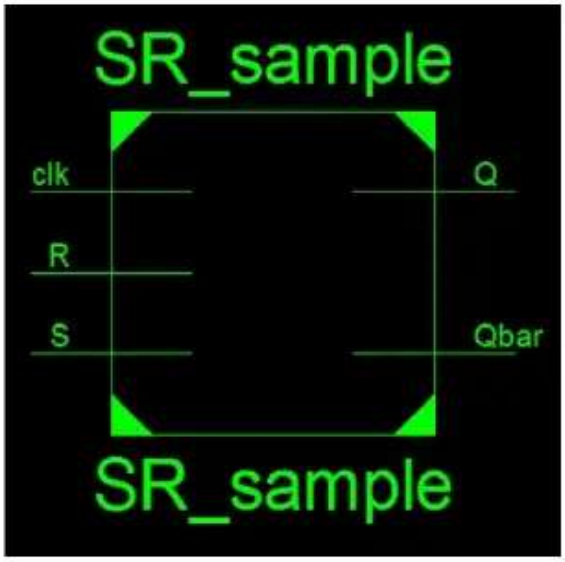


Figure 8.2 Logic circuit for SR Flipflop

8.3.2 PROCEDURE

Step 1: Write the Verilog code for SR flipflop and obtain the RTL schematic

```
module SR_sample(input S,
   input R,
   input clk,
   output Q,
   output Qbar
   );
   reg M,N;
always @(posedge clk) begin
 M <= !(S & clk);
 N <= !(R & clk);
end
assign Q = !(M & Qbar);
assign Qbar = !(N & Q);
```



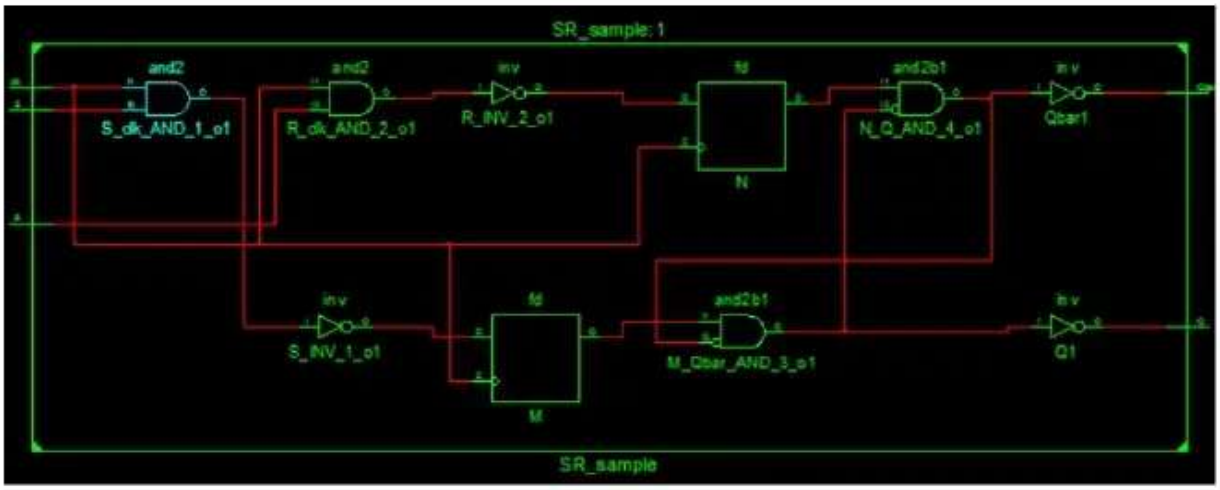


Figure 8.3 RTL Schematic for SR Flipflop

Step 2: Write the testbench code for SR Flipflop and obtain the RTL simulation waveform

Testbench code:

```
module TestSR;
// Inputs
reg S;
reg R;
reg clk;
// Outputs
wire Q;
wire Qbar;
SR_sample uut (
.S(S),
.R(R),
.clk(clk),
.Q(Q),
.Qbar(Qbar)
);
initial begin
   $monitor("At time %0t:S = %b, R = %b -> Q = %0b, Qbar = %0b",$time,S,R,clk,Q,Qbar);
S = 0;
   R = 0;
```

```
    clk = 0;
  fork
  #2 S = 0;
  #2 R = 1;
  #4 S = 0;
  #4 R = 0;
  #6 S = 0;
  #6 R = 1;
  #8 S = 1;
  #8 R = 0;
  #10 S = 1;
  #10 R = 1;
  join
 end
always #1 clk =! clk;
endmodule
```

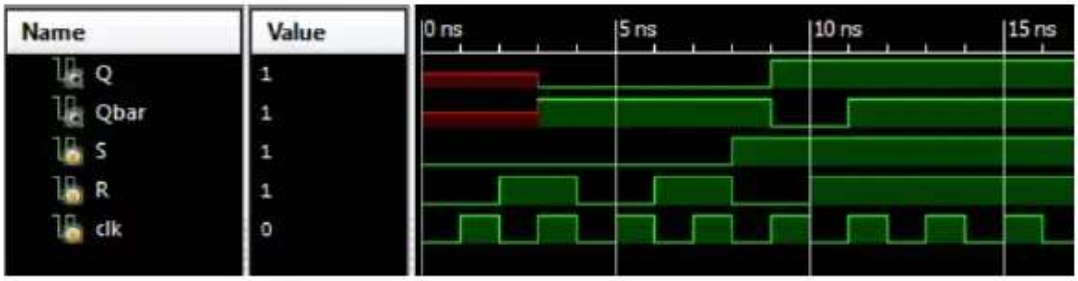


Figure 8.4 RTL Simulation waveform for SR Flipflop

Step 3: Get the result from the output window

At time 0:S = 0, R = 0 -> Q = 0, Qbar = xx

At time 1000:S = 0, R = 0 -> Q = 1, Qbar = xx

At time 2000:S = 0, R = 1 -> Q = 0, Qbar = xx

At time 3000:S = 0, R = 1 -> Q = 1, Qbar = 01

At time 4000:S = 0, R = 0 -> Q = 0, Qbar = 01

At time 5000:S = 0, R = 0 -> Q = 1, Qbar = 01

At time 6000:S = 0, R = 1 -> Q = 0, Qbar = 01

At time 7000:S = 0, R = 1 -> Q = 1, Qbar = 01

At time 8000:S = 1, R = 0 -> Q = 0, Qbar = 01

At time 9000:S = 1, R = 0 -> Q = 1, Qbar = 10

At time 10000:S = 1, R = 1 -> Q = 0, Qbar = 10

At time 11000:S = 1, R = 1 -> Q = 1, Qbar = 11

At time 12000:S = 1, R = 1 -> Q = 0, Qbar = 11

At time 13000:S = 1, R = 1 -> Q = 1, Qbar = 11

At time 14000:S = 1, R = 1 -> Q = 0, Qbar = 11

At time 15000:S = 1, R = 1 -> Q = 1, Qbar = 11

8.3.3 APPLICATIONS:

1. **Latches:** SR flip-flops are used to implement latches, which are basic memory elements. They store and retain information as long as power is applied or until reset. SR latches are used in data storage applications.
2. **Synchronization:** SR flip-flops are used in synchronization circuits to ensure that data is transferred from one part of a system to another only when a specific signal (e.g., a clock pulse) is received.
3. **Bistable Circuit:** An SR flip-flop can be used as a bistable circuit for toggling between two states. It can be used in signal processing and control circuits.

8.4 JK flipflop

8.4.1THEORY:

It has three inputs: J (set), K (reset), and Clock (C or CLK), and two outputs: Q and Q'. The behavior of a JK flip-flop is similar to that of an SR flip-flop except that it eliminates undefined output state (Q = x for S=1, R=1) and with the addition of a clock input.It has four possible operating modes, depending on the combination of J and K inputs, allowing for toggling, setting, resetting, and holding states. For J=1, K=1, output Q toggles from its previous output state.

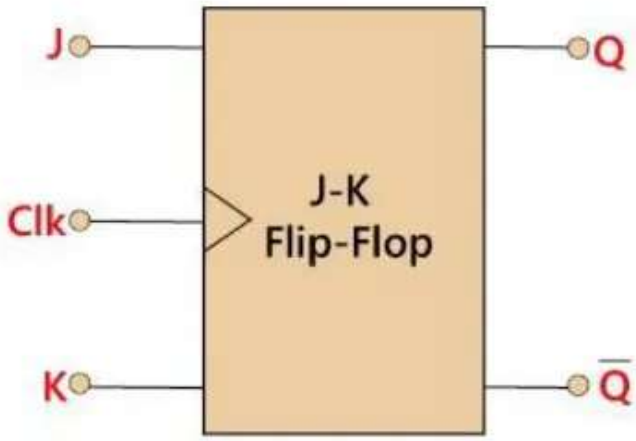


Figure 8.5 Block diagram of JK Flipflop

Table 8.2 Truth Table for JK Flipflop

Same as for SR Latch	Clock	Input		Output		Description
	Clk	J	K	Q	Q'	
	X	0	0	1	0	Memory no change
	X	0	0	0	1	
	$\overline{\downarrow}$	0	1	1	0	Reset Q>>0
	X	0	1	0	1	
	$\overline{\downarrow}$	1	0	0	1	Set Q>>1
	X	1	0	1	0	
Toggle action	$\overline{\downarrow}$	1	1	0	1	Toggle
	$\overline{\downarrow}$	1	1	1	0	

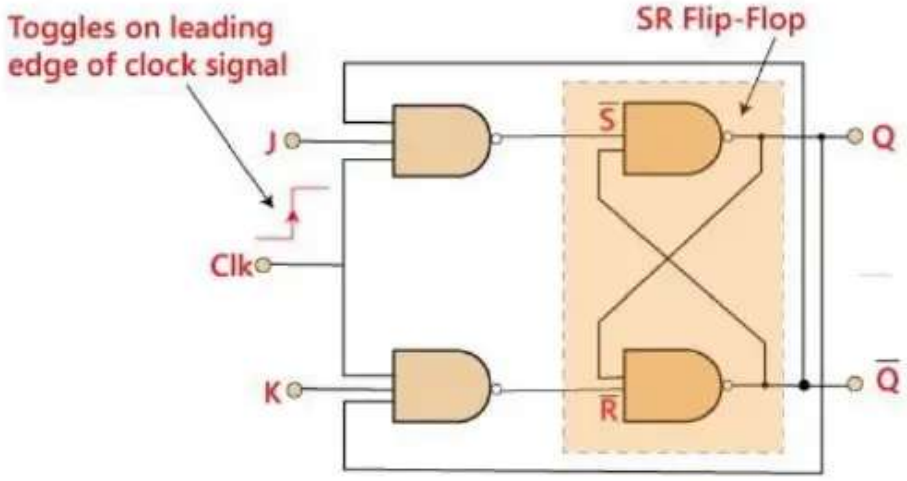
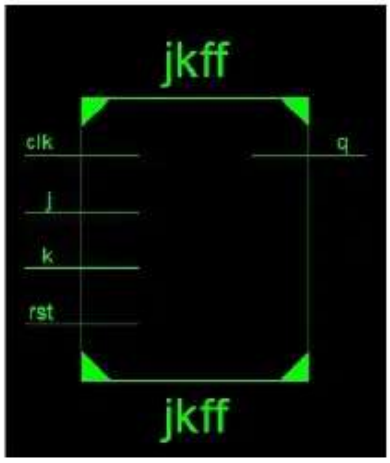


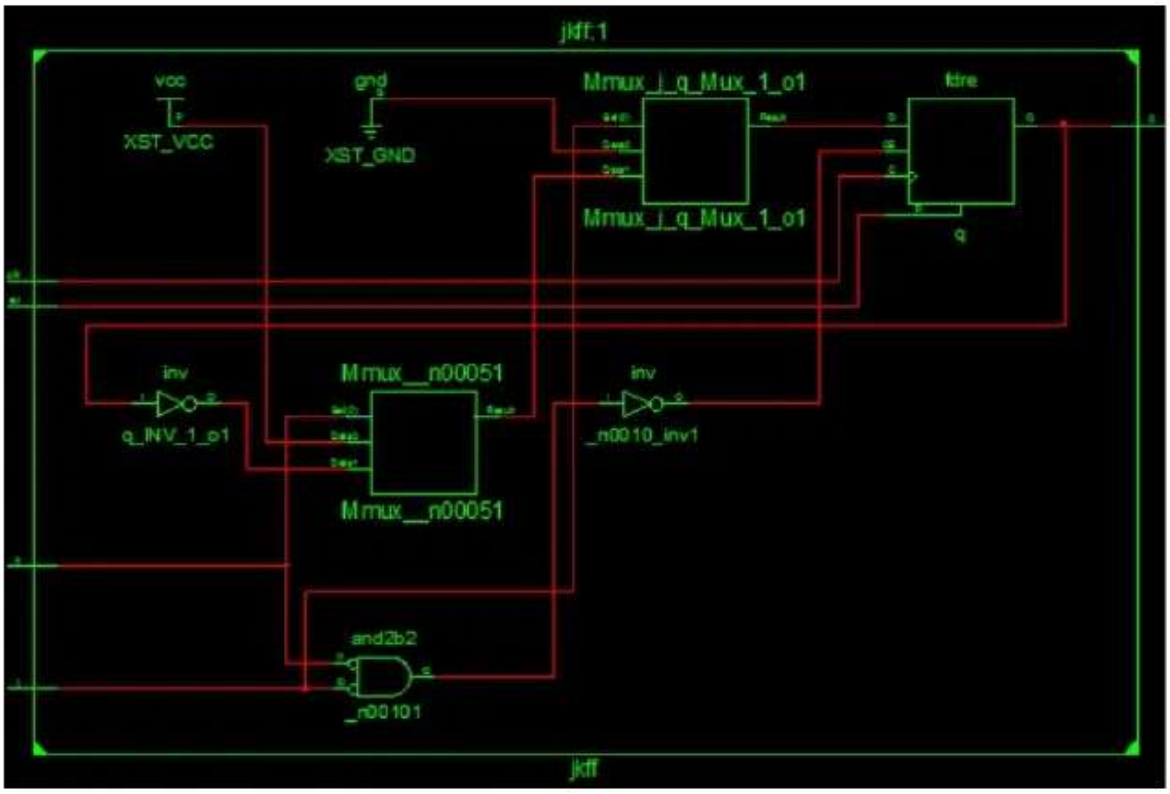
Figure 8.6 Logic circuit for JK Flipflop

8.4.2 PROCEDURE:

Step 1: Write the Verilog code for JK Flipflop and obtain the RTL schematic

```
// Code your design here
module jkff(j,k,clk,rst,q);
 input j,k,clk,rst;
 output reg q;
 always @(posedge clk)
  begin
   if(rst)
    q<=0;
   else
    begin
     case({j,k})
      2'b00:q<=q;
      2'b01:q<=0;
      2'b10:q<=1;
      2'b11:q<=(~q);
      default:q<=q;
      endcase
    end
  end
endmodule
```





8.7 RTL Schematic for JK Flipflop

Step 2: Write the testbench code for JK Flipflop and obtain the RTL simulation waveform

Testbench code:

```
module jkfftb();
 reg j,k,clk,rst;
 wire q;
 jkff p(j,k,clk,rst,q);
 initial
  begin
    $dumpfile("jkfftb.vcd");
    $dumpvars(0,jkfftb);
  end
 initial
  begin
    clk=1;
```

```
    forever #5 clk=~clk;
  end
initial
  begin
        $monitor("At time %0t:j = %b, k = %b, clk = %b,rst = %b,q = %0b",$time,j,k,clk,rst,q);
   #10 rst=1;
   #10 rst=0;
   #10 j=1;k=0;
    #10 j=1;k=1;
    #10 j=0;k=0;
    #10 j=0;k=1;
   #10000 $finish;
  end
endmodule
```

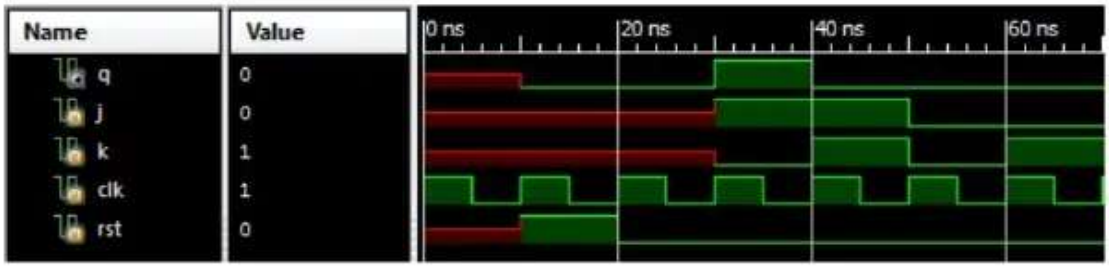


Figure 8.8 RTL Simulation waveform for JK Flipflop

Step 3: Get the result from the output window

At time 0:j = x, k = x, clk = 1,rst = x,q = x

At time 5000:j = x, k = x, clk = 0,rst = x,q = x

At time 10000:j = x, k = x, clk = 1,rst = 1,q = 0

At time 15000:j = x, k = x, clk = 0,rst = 1,q = 0

At time 20000:j = x, k = x, clk = 1,rst = 0,q = 0

At time 25000:j = x, k = x, clk = 0,rst = 0,q = 0

At time 30000:j = 1, k = 0, clk = 1,rst = 0,q = 1

At time 35000:j = 1, k = 0, clk = 0,rst = 0,q = 1

At time 40000:j = 1, k = 1, clk = 1,rst = 0,q = 0

At time 45000:j = 1, k = 1, clk = 0,rst = 0,q = 0

At time 50000:j = 0, k = 0, clk = 1,rst = 0,q = 0
At time 55000:j = 0, k = 0, clk = 0,rst = 0,q = 0
At time 60000:j = 0, k = 1, clk = 1,rst = 0,q = 0
At time 65000:j = 0, k = 1, clk = 0,rst = 0,q = 0
At time 70000:j = 0, k = 1, clk = 1,rst = 0,q = 0

8.4.3 APPLICATIONS:

1. **Counters:** JK flip-flops are often used in synchronous counters, including up-counters and down-counters. They are fundamental in digital systems for counting events, creating frequency dividers, and generating timing signals.
2. **State Machines:** JK flip-flops are used in state machines, which are critical components of control systems. They help implement sequential logic, where the behavior of a system depends on its current state and inputs.
3. **Frequency Division**: JK flip-flops can be used in frequency dividers and clock generators. By toggling the J and K inputs appropriately, they can divide the input frequency by various factors.

8.5 D FLIPFLOP

8.5.1 THEORY

A D flip-flop, also known as a Data flip-flop, is a fundamental digital circuit element used in digital electronics and sequential logic circuits. It is a bistable multivibrator, meaning it can store and represent a single bit of data. The primary function of a D flip-flop is to capture the data at its D (data) input and transfer it to its Q (output) on the rising or falling edge of a clock signal, allowing it to store and propagate data within digital systems. The state of the D flip-flop remains stable between clock edges, making it essential for various applications in data storage, synchronization, and state control in digital systems.

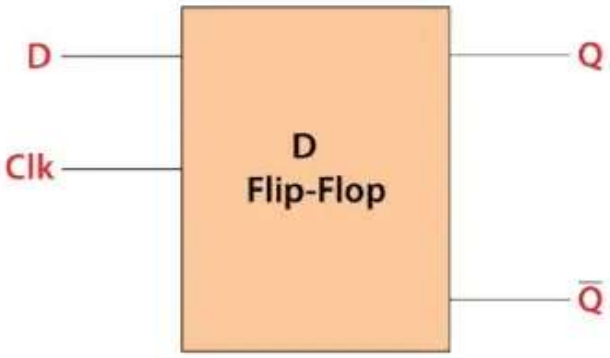


Figure 8.9 Block Diagram of D Flipflop

Table 8.3: Truth Table of D Flipflop

Clock	D	Q	Q'	Description
↓ » 0	X	Q	Q'	Memory no change
↑ » 1	0	0	1	Reset Q » 0
↑ » 1	1	1	0	Set Q » 1

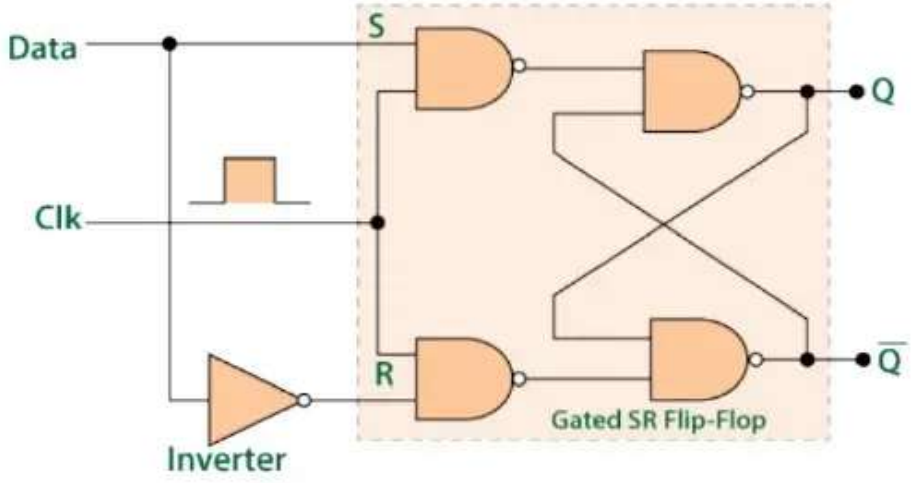


Figure 8.10 Logic circuit of D flipflop

8.5.2 PROCEDURE:

Step 1: Write the Verilog code for D Flipflop and obtain the RTL schematic

```
// Code your design here
module dff(d,clk,rst,q);
  input d,clk,rst;
```

```
 output reg q;
 always @(posedge clk)
   begin
     if(rst)
       q<=0;
     else
      q<=d;
   end
endmodule
```

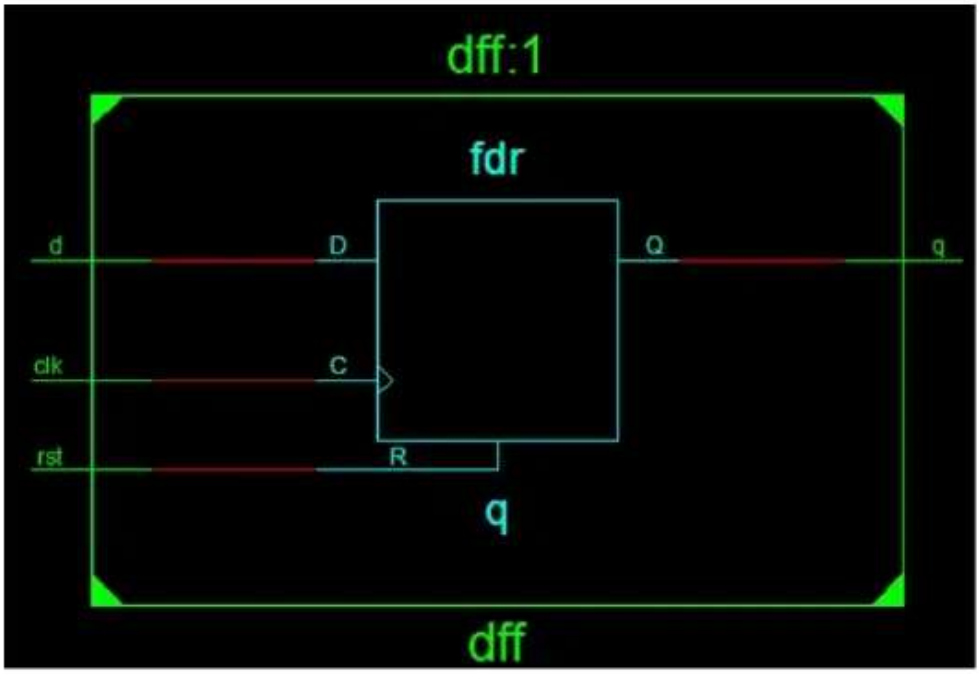


Figure 8.11 RTL Schematic of D Flipflop

Step 2: Write the testbench code for D Flipflop and obtain the RTL simulation waveform

```
module dfftb();
 reg d,clk,rst;
 wire q;
 dff p(d,clk,rst,q);
 initial
   begin
     $dumpfile("dfftb.vcd");
```

```
    $dumpvars(0,dfflb);
       #10000 $finish;
   end
initial
   begin
    clk=1;
     forever #5 clk=~clk;
   end
initial
   begin
          $monitor("At time %0t:d = %b, clk = %b, rst = %b, q = %0b",$time,d,clk,rst,q);
    #10 rst=1;
    #10 rst=0;
    #10 d=0;
    #10 d=1;
   end
endmodule
```

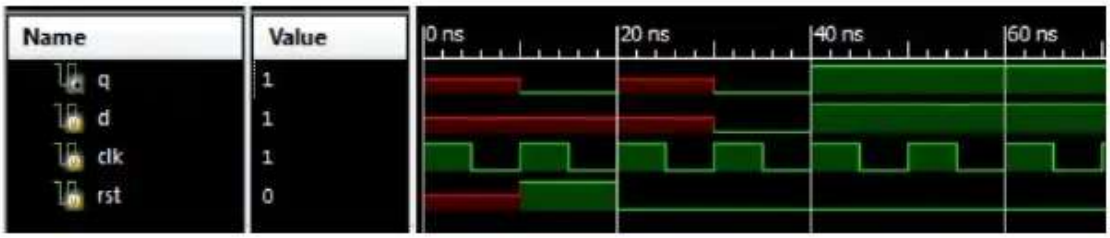


Figure 8.12 RTL Simulation waveform for D Flipflop

Step 3: Get the result from the output window

At time 0:d = x, clk = 1, rst = x, q = x

At time 5000:d = x, clk = 0, rst = x, q = x

At time 10000:d = x, clk = 1, rst = 1, q = 0

At time 15000:d = x, clk = 0, rst = 1, q = 0

At time 20000:d = x, clk = 1, rst = 0, q = x

At time 25000:d = x, clk = 0, rst = 0, q = x

At time 30000:d = 0, clk = 1, rst = 0, q = 0

At time 35000:d = 0, clk = 0, rst = 0, q = 0

At time 40000:d = 1, clk = 1, rst = 0, q = 1
At time 45000:d = 1, clk = 0, rst = 0, q = 1
At time 50000:d = 1, clk = 1, rst = 0, q = 1
At time 55000:d = 1, clk = 0, rst = 0, q = 1
At time 60000:d = 1, clk = 1, rst = 0, q = 1

8.5.3 APPLICATIONS:

1. **Data Storage:** D flip-flops are commonly used for data storage, particularly in registers and memory units. They hold data until it is either read or overwritten.
2. **Data Transfer:** D flip-flops are used to transfer data between different parts of a digital system or between clock domains. They are key components in data buses and data pipelines.
3. **Edge Detection:** D flip-flops are used in edge-detection circuits, which capture the state of a signal at the rising or falling edge of a clock signal. This is important in various digital communication and control systems.

8.6 EXPECTED VIVA QUESTIONS

SR Flip-Flop:

1. What is an SR flip-flop, and what are its primary characteristics?
2. Explain the truth table and operation of an SR flip-flop.
3. What are the possible states of an SR flip-flop?
4. How does the presence of an illegal state in an SR flip-flop affect its operation?
5. Describe a real-world application where an SR flip-flop is useful.
6. How can you use an SR flip-flop to create a simple toggle circuit?
7. Explain the concept of setup time and hold time in the context of an SR flip-flop.

JK Flip-Flop:

1. What are the features that distinguish a JK flip-flop from an SR flip-flop?
2. Walk me through the truth table and operation of a JK flip-flop.
3. How is a JK flip-flop used to create a toggle circuit?
4. In what applications is a JK flip-flop preferred over other types of flip-flops?

5. Can you discuss the significance of the J and K inputs in a JK flip-flop's operation?
6. Describe a scenario where a JK flip-flop is employed for edge detection.

D Flip-Flop:

1. What are the fundamental characteristics of a D flip-flop?
2. Explain the truth table and operation of a D flip-flop.
3. How does a D flip-flop store and transfer data?
4. Describe the role of the clock signal in a D flip-flop's operation.
5. In what applications is a D flip-flop typically used for data storage and transfer?
6. What is the advantage of using D flip-flops in registers and memory units?
7. Discuss the concept of edge-triggering in the context of D flip-flops.

Common Questions:

1. Can you compare and contrast the advantages and disadvantages of SR, JK, and D flip-flops in various applications?
2. What are the considerations for selecting the appropriate type of flip-flop for a given digital system design?
3. How do flip-flops contribute to sequential logic circuits and state machines?
4. Explain the concept of synchronous and asynchronous inputs in flip-flops.
5. What are the differences between master-slave and edge-triggered flip-flops?
6. How are flip-flops useful in addressing setup and hold time requirements in digital systems?
7. Describe a scenario where flip-flops are used for synchronization in a multi-clock domain system.
8. These questions should help you prepare for a viva examination or interview focused on SR, JK, and D flip-flops, covering their principles, operation, applications, and considerations. Be ready to explain the concepts in detail and provide practical examples where relevant.