RV Educational Institutions

RV Institute of Technology and Management®, Bengaluru

Department of Electronics and Communication Engineering

Digital System Design using Verilog (BEC302)

III Semester

2022 Scheme

Lab Manual

PRASHANT PATAVARDHAN

Professor of ECE, RVITM

EXPERIMENTS

PART-A

1. Simplify the given Boolean expressions and realize using Verilog program.
2. Realize Adder/Subtractor(Full/half)circuits using Verilog data flow description.
3. Realize 4-bit ALU using Verilog program.
4. Realize the following Code converters using Verilog Behavioral description.

   a) Gray to binary and vice versa b) Binary to excess3 and vice versa

5. Realize 8:1mux, 8:3encoder, and Priority encoder using Verilog Behavioral description.
6. Realize 1:8Demux, 3:8 decoder, and 2–bit Comparator using Verilog Behavioral description.
7. Realize using Verilog Behavioral description- Flip-flops:

   a) JK type b) SR type c) T type and d) D type.

8. Realize Counters-up/down (BCD and binary) using Verilog Behavioral description.

PART-B

Demonstration Experiments (For CIE only–not to be included for SEE)

Use FPGA/CPLD kits for down loading Verilog codes and check the output for interfacing experiments.

9. Verilog Program to interface a Stepper motor to the FPGA/CPLD and rotate the motor in the specified direction (by N steps).
10. Verilog programs to interface Switches and LEDs to the FPGA/CPLD and demonstrate its working.

Additional

11. Verilog program to interface a Relay or ADC to the FPGA/CPLD and demonstrate its working.
12. Verilog program to interface DAC to the FPGA/CPLD for Waveform generation.

Experiment No. 1

Simplify the given Boolean expressions and realize using Verilog program.

Theory:

Boolean expression simplification is a process used in digital logic design to reduce the complexity of Boolean expressions. This simplification is crucial for minimizing the number of logic gates required in a circuit, which can lead to increased reliability and reduced manufacturing costs. The simplification process involves applying various algebraic rules and identities from Boolean algebra to achieve a more efficient expression.

Key Concepts in Boolean Algebra

Basic Operations: The primary operations in Boolean algebra are:

AND (denoted as multiplication, e.g., A·B or AB)

OR (denoted as addition, e.g., A+B)

NOT (denoted as negation, e.g., $\bar{A}$)

Laws and Identities: Several fundamental laws govern Boolean algebra, including:

Idempotent Law: A+A=A and A·A=A

Domination Law: A+1=1 and A·0=0

Absorption Law: A+AB=A and A(A+B)=A

These laws allow for the manipulation of expressions to eliminate redundant terms and simplify the overall expression.

Methods of Simplification

Algebraic Manipulation: This involves applying the laws of Boolean algebra directly to the expression. For example, the expression A·(A+B) can be simplified to just A using the Absorption Law.

Karnaugh Maps (K-maps): A K-map is a visual tool that helps simplify Boolean expressions by grouping together adjacent cells that represent common factors. This method is particularly useful for expressions with two to four variables, allowing for a systematic approach to finding the simplest form.

Truth Tables: Constructing a truth table for a Boolean expression can help identify redundant terms and simplify the expression by focusing on the output values.

Benefits/Features of Simplification

Boolean expression simplification is a fundamental aspect of digital logic design, aimed at reducing the complexity of Boolean expressions. Here are some key features of this process:

Reduction of Complexity: The primary goal of simplification is to reduce the number of terms and operations in a Boolean expression. This leads to simpler logic circuits, which are easier to design, implement, and troubleshoot.

Use of Boolean Laws: Simplification relies heavily on various laws and identities of Boolean algebra, such as the Idempotent Law, Domination Law, and Absorption Law. These laws provide systematic methods for transforming complex expressions into simpler forms.

Visual Tools: Techniques like Karnaugh Maps (K-maps) are commonly used to simplify Boolean expressions visually. K-maps allow for the grouping of adjacent cells that represent common factors, making it easier to eliminate unnecessary variables and terms.

Truth Tables: Constructing truth tables is another method for simplification. By comparing the outputs of different expressions, one can identify redundant terms and ensure that the simplified expression maintains the same functionality as the original.

Minimized Circuit Design: Simplified Boolean expressions translate directly into more efficient circuit designs. Fewer gates and components lead to increased reliability and reduced manufacturing costs, which is particularly important in large-scale digital systems.

Application of De Morgan's Theorems: De Morgan's Theorems are essential in the simplification process, allowing for the transformation of expressions involving AND and OR operations into their complements. This is particularly useful when dealing with negated expressions.

Algorithmic Approaches: Advanced methods, such as the Quine-McCluskey algorithm, can be employed for simplifying Boolean expressions, especially when dealing with a large number of variables. These algorithms systematically reduce expressions to their simplest forms.

Practical Tools: There are various online calculators and software tools available that can automate the simplification process, providing detailed steps and visual representations of the logic circuits involved.

Cost Efficiency: Fewer components mean lower manufacturing costs and increased reliability of the circuit.

Let's simplify the given Boolean expression $\boldsymbol{F = X' Y Z + X Y' Z' + X Y Z + X Y Z'}$ and then implement it using a Verilog program.

Simplification of Boolean Expression

Given: $F = X' Y Z + X Y' Z' + X Y Z + X Y Z'$

Group and simplify the expression:

Group terms with common factors:

$$F = X' Y Z + X Y Z + X Y Z' + X Y' Z'$$

Factor common terms:

Factor YZ from the first and third terms:

$$F = Y Z (X' + X) + X (Y Z' + Y' Z')$$

Since $X' + X = 1$:

$$F = Y Z + X (Y Z' + Y' Z')$$

Simplify the remaining expression:

Factor Z' from the second group:

$$F = Y Z + X Z' (Y + Y')$$

Since $Y + Y' = 1$

$$F = Y Z + X Z'$$

The final simplified Boolean expression is:

$$\boldsymbol{F = Y Z + X Z'}$$

Verilog Code: boolean.v

```
module boolean_function(
  input wire X,
  input wire Y,
  input wire Z,
  output wire F
```

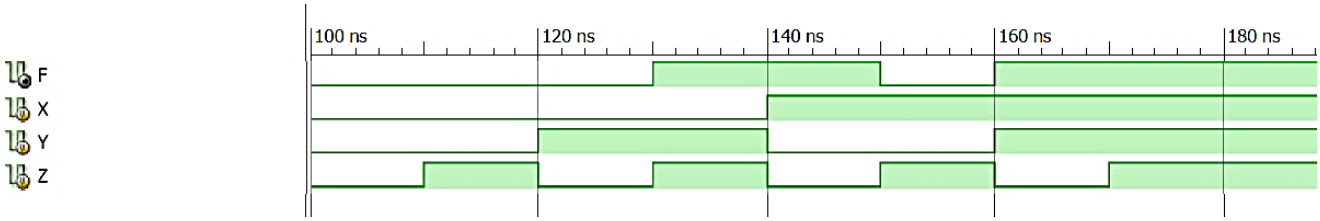
```
);
   // Implementing the simplified expression F = (Y & Z) | (X & ~Z)
   assign F = (Y & Z) | (X & ~Z);
endmodule
```

Test Bench: tb_boolean.v

```
module tb_boolean;
        // Inputs
        reg X;
        reg Y;
        reg Z;

        // Outputs
        wire F;

        // Instantiate the Unit Under Test (UUT)
        boolean_function uut (
                .X(X),
                .Y(Y),
                .Z(Z),
                .F(F)
        );

        // Testbench process
        initial begin
                // Initialize Inputs
                X = 0;
                Y = 0;
```

```
        Z = 0;

        // Wait 100 ns for global reset to finish
        #100;

        // Add stimulus here
        // Monitor outputs
    $monitor("At time %t, X = %b, Y = %b, Z = %b => F = %b", $time, X, Y, Z, F);

        // Apply all possible input combinations
        #10 X = 0; Y = 0; Z = 1;
        #10 X = 0; Y = 1; Z = 0;
        #10 X = 0; Y = 1; Z = 1;
        #10 X = 1; Y = 0; Z = 0;
        #10 X = 1; Y = 0; Z = 1;
        #10 X = 1; Y = 1; Z = 0;
        #10 X = 1; Y = 1; Z = 1;

        // End of simulation
        ;
    end
endmodule
```

Output Waveform:



Output Data:

At time 100000, X = 0, Y = 0, Z = 0 => F = 0

At time 110000, X = 0, Y = 0, Z = 1 => F = 0

At time 120000, X = 0, Y = 1, Z = 0 => F = 0

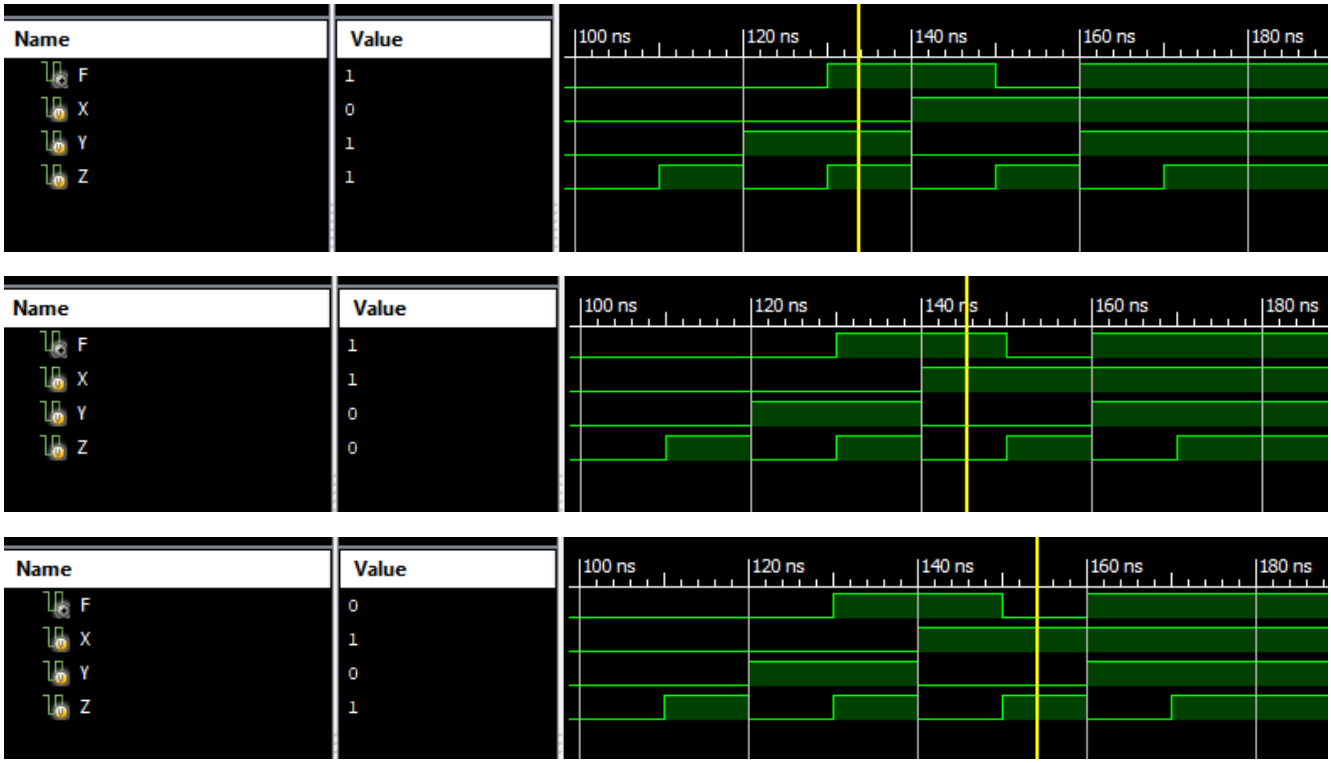
At time 130000, X = 0, Y = 1, Z = 1 => F = 1

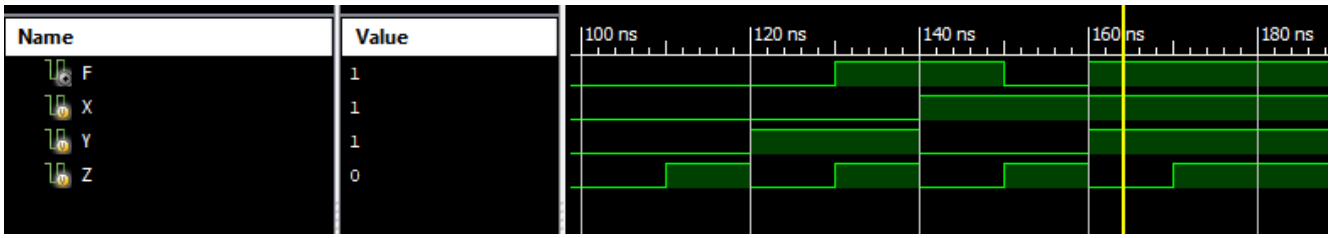
At time 140000, X = 1, Y = 0, Z = 0 => F = 1

At time 150000, X = 1, Y = 0, Z = 1 => F = 0

At time 160000, X = 1, Y = 1, Z = 0 => F = 1

At time 170000, X = 1, Y = 1, Z = 1 => F = 1




Name
Value
F
X
Y
Z
1
1
1
0
100 ns
120 ns
140 ns
160 ns
180 ns

Experiment No. 2

Realize Adder/Subtractor (Full/Half) circuits using Verilog data flow description.

Theory:

Half Adder:

A Half Adder is a fundamental digital circuit used for adding two single-bit binary numbers. It produces two outputs: the Sum (S) and the Carry (C). The Half Adder is the simplest form of an adder and can be constructed using just two types of logic gates: an XOR gate and an AND gate.

Truth Table

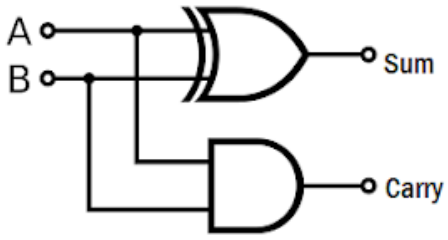
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Boolean Expression

Sum (S)- The Sum is true (1) when exactly one of the inputs is true. $S=A \oplus B$ (XOR operation)

Carry (C)- The Carry is true (1) when both inputs are true. $C=A \cdot B$ (AND operation)

Logic Circuit



Limitations

- While the Half Adder is useful for adding two single bits, it has limitations:

- It cannot handle carry input from previous additions, which is necessary for adding multi-bit binary numbers. For this purpose, a Full Adder is used, which can take an additional carry input.

Applications

Half Adders are commonly used in:

- Arithmetic Logic Units (ALUs): They perform basic arithmetic operations.
- Digital Circuits: They serve as building blocks for more complex adder circuits, such as Full Adders and multi-bit adders.

Full Adder:

A Full Adder is a digital circuit that performs the addition of three binary digits: two significant bits and a carry-in bit. It produces two outputs: the Sum (S) and the Carry-out (Cout). The Full Adder is essential in digital electronics, especially for constructing arithmetic circuits that handle multi-bit binary numbers.

Truth Table

A	B	C	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

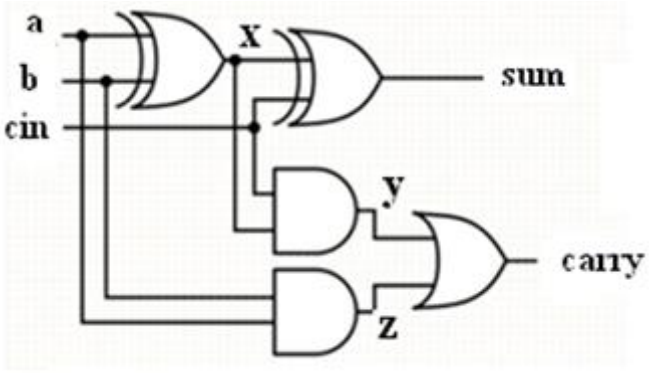
Boolean Expression

Sum (S): The Sum is true when an odd number of inputs are true. $S=A \oplus B \oplus C_{in}$ (XOR operation)

Carry-out (C_{out}): The Carry-out is true when at least two of the inputs are true.

$C_{out} = (A \cdot B) + (B \cdot C_{in}) + (A \cdot C_{in})$ / $C_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B))$ (AND and OR operations)

Logic Circuit



Applications

Full Adders are widely used in:

- Arithmetic Logic Units (ALUs): They perform arithmetic operations in processors.
- Multi-bit Adders: Full Adders can be cascaded to add binary numbers of more than one bit, allowing for the addition of larger binary numbers.

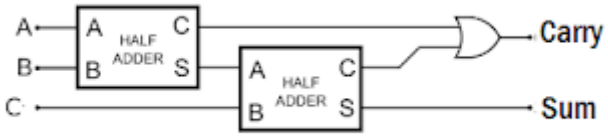
Full Adder using Half Adder

A Full Adder can be constructed using two Half Adders and one OR gate. This approach leverages the simpler Half Adder circuits to build the more complex Full Adder. This design allows the Full Adder to add three binary digits: two significant bits (A and B) and a carry-in bit (C_{in}).

Half Adder: A circuit that adds two single-bit binary numbers and produces a sum and a carry.

OR Gate: A logic gate that outputs true (1) if at least one of its inputs is true.

Logic Circuit



First Half Adder:

Sum1 (S_1) = A XOR B (This is the first partial sum)

Carry1 (C_1) = A AND B (This is the carry from the addition of A and B)

Second Half Adder:

Sum (S) = S_1 XOR C_{in} (This is the final sum output)

Carry2 (C_2) = S_1 AND C_{in} (This is the carry from the addition of Sum1 and C_{in})

Final Carry Output:

The final carry-out (C_{out}) is obtained by combining the two carry outputs:

C_{out} = C_1 OR C_2 (This indicates if there is a carry to the next higher bit)

Sum: $S=S_1 \oplus C_{in} = (A \oplus B) \oplus C_{in}$

Carry-out: $C_{out}=C_1+C_2 = (A \cdot B)+(S_1 \cdot C_{in})=(A \cdot B)+((A \oplus B) \cdot C_{in})$

Half Subtractor:

A Half Subtractor is a combinational logic circuit that performs the subtraction of two single-bit binary numbers. It has two inputs, typically referred to as A (the minuend) and B (the subtrahend), and produces two outputs: the Difference (D) and the Borrow (B). The Half Subtractor is a fundamental building block in digital electronics, particularly in arithmetic circuits.

Truth Table

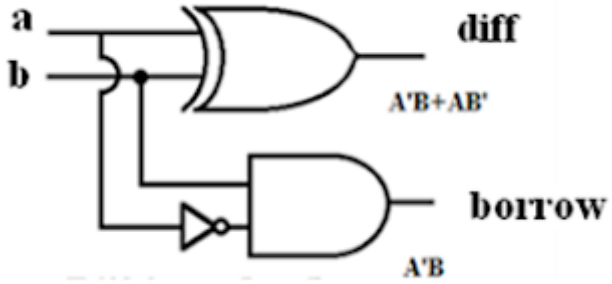
A	B	DIFFERENCE (D)	BORROW (B)
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Boolean Expression

Difference (D): The Difference is true (1) when the inputs A and B are different. $D=A \oplus B$ (XOR operation)

Borrow (B): The Borrow is true (1) when A is 0 and B is 1. $B=A' \cdot B$ (NOT A AND B)

Logic Circuit



Limitations

The Half Subtractor has limitations:

- It cannot handle a borrow input from a previous subtraction, which is necessary for subtracting multi-bit binary numbers. For this purpose, a Full Subtractor is used, which can take an additional borrow input.

Applications of a Half Subtractor

- Binary Subtraction: The Half Subtractor is used in circuits that perform binary subtraction, particularly in the first stage where no previous borrow exists.
- ALUs and Digital Systems: Like the Half Adder, the Half Subtractor is a fundamental building block in arithmetic logic units (ALUs) and digital processors.

Full Subtractor:

A Full Subtractor is a combinational logic circuit that performs the subtraction of two binary digits while considering a borrow from a previous subtraction. It has three inputs: the minuend (A), the subtrahend (B), and the borrow-in (B_{in}). The Full Subtractor produces two outputs: the Difference (D) and the Borrow-out (B_{out}).

Truth Table

A	B	Bin	Difference (D)	Borrow-out (Bout)
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0

1	0	1	0	1
1	1	0	0	0
1	1	1	1	1

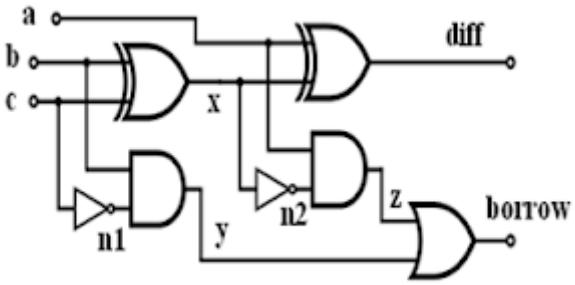
Boolean Expression

Difference (D): The Difference is true (1) when the number of true inputs (1s) is odd.
D=A⊕B⊕Bin (XOR operation)

Borrow-out (B_{out}): The Borrow-out is true (1) when B and/or B_{in} are greater than A.

Bout=A'·B+A'·B_{in}+B·B_{in} / $B_{out}=(A'\cdot B)+((A\oplus B)\cdot B_{in})$

Logic Circuit



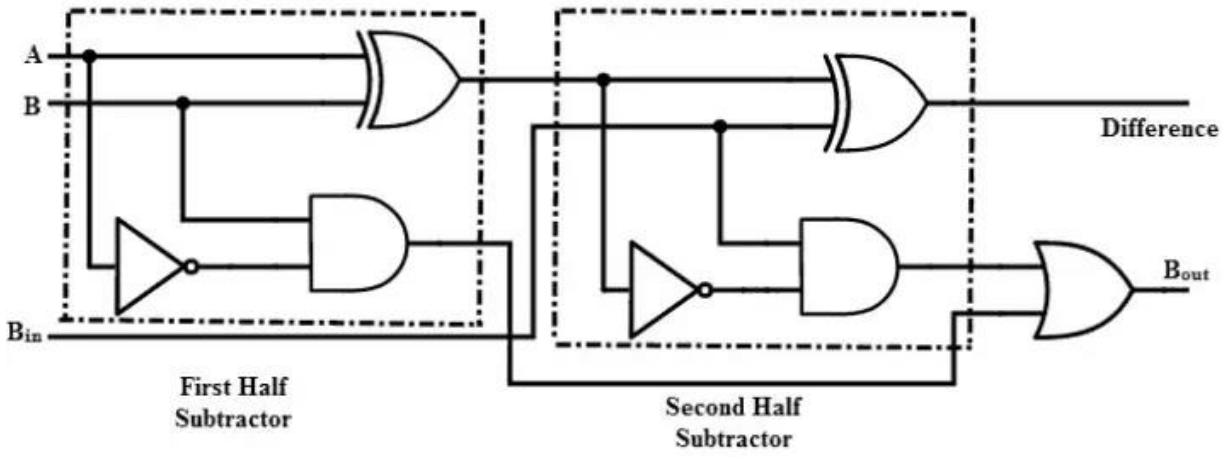
Applications of a Full Subtractor

- Multi-bit Binary Subtraction: Full Subtractors are used in series to subtract multi-bit binary numbers, with the Borrow-out of one stage connected to the Borrow-in of the next.
- Digital Circuits: They are used in arithmetic logic units (ALUs) for performing binary subtraction, which is fundamental in processors and digital systems.

Full Subtractor using Half Subtractor

A Full Subtractor can be constructed using two Half Subtractors and an OR gate. This design leverages the simpler Half Subtractor circuits to build the more complex Full Subtractor.

Logic Circuit



First Half Subtractor:

Inputs: A and B

Outputs: D1=A⊕B (Intermediate Difference) and B1=A'·B (Intermediate Borrow)

Second Half Subtractor:

Inputs: D1 (Difference from the first Half Subtractor) and Bin

Outputs: Difference (D): D=D1⊕Bin (Final Difference) and Borrow (B2): B2=D'1·Bin (Final Borrow)

OR Gate:

Inputs: B1 (Borrow from the first Half Subtractor) and B2 (Borrow from the second Half Subtractor).

Output: Bout=B1+B2 (Final Borrow-out)

Verilog Code:

Half Adder

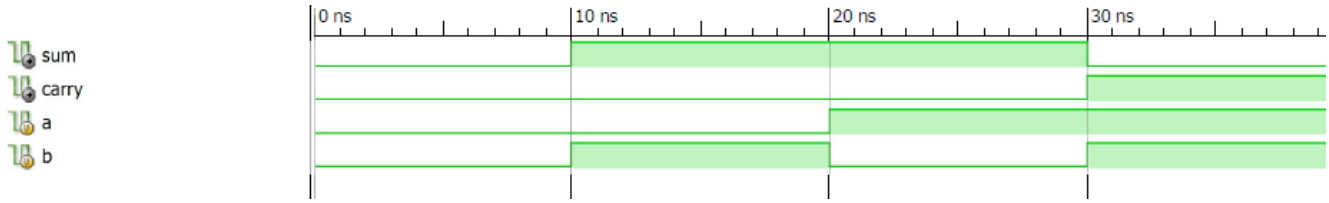
```
module half_adder(
    input wire A,    // First input
    input wire B,    // Second input
    output wire Sum, // Sum output
    output wire Carry // Carry output
);
```

```
    // Sum = A XOR B
    assign Sum = A ^ B;

    // Carry = A AND B
    assign Carry = A & B;

endmodule

module test_half_adder;

    // Inputs to the half adder
    reg A;
    reg B;

    // Outputs from the half adder
    wire Sum;
    wire Carry;

    // Instantiate the Half Adder module
    half_adder uut (
        .A(A),
        .B(B),
        .Sum(Sum),
        .Carry(Carry)
    );
```

```
// Test cases
initial begin
    // Display header
    $display("A B | Sum Carry");
    $display("--------------");

    // Test Case 1: A = 0, B = 0
    A = 0; B = 0;
    #10; // Wait for 10 time units
    $display("%b %b |  %b    %b", A, B, Sum, Carry);

    // Test Case 2: A = 0, B = 1
    A = 0; B = 1;
    #10;
    $display("%b %b |  %b    %b", A, B, Sum, Carry);

    // Test Case 3: A = 1, B = 0
    A = 1; B = 0;
    #10;
    $display("%b %b |  %b    %b", A, B, Sum, Carry);

    // Test Case 4: A = 1, B = 1
    A = 1; B = 1;
    #10;
    $display("%b %b |  %b    %b", A, B, Sum, Carry);

    // End the simulation
```

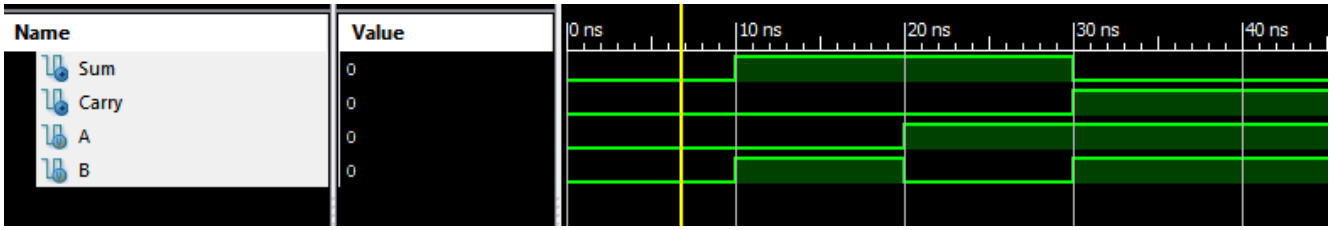
```
        $finish;
    end

endmodule
```

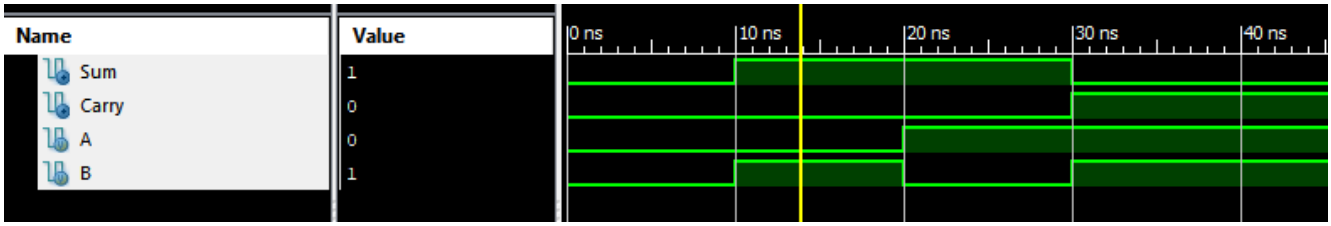
Output Waveform

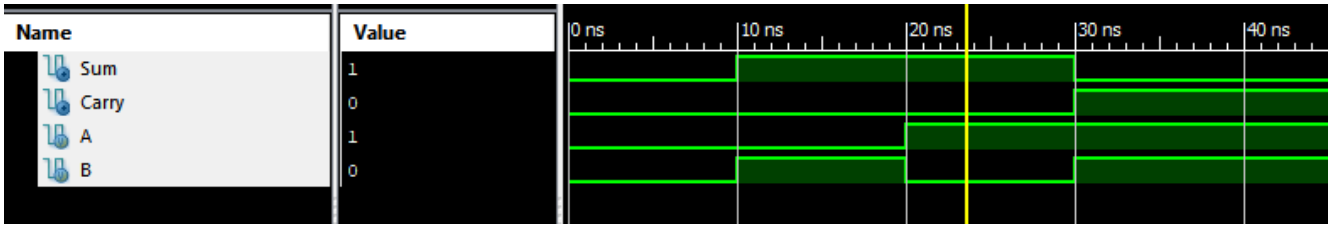


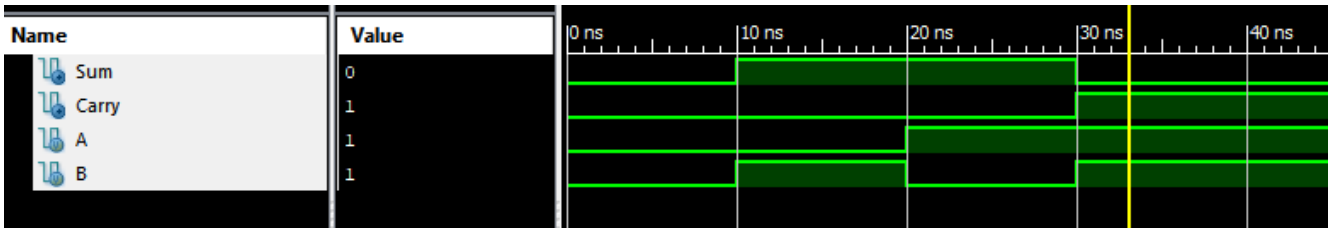
Output Data

A B | Sum Carry
---
0 0 | 0 0
0 1 | 1 0
1 0 | 1 0
1 1 | 0 1









Full Adder

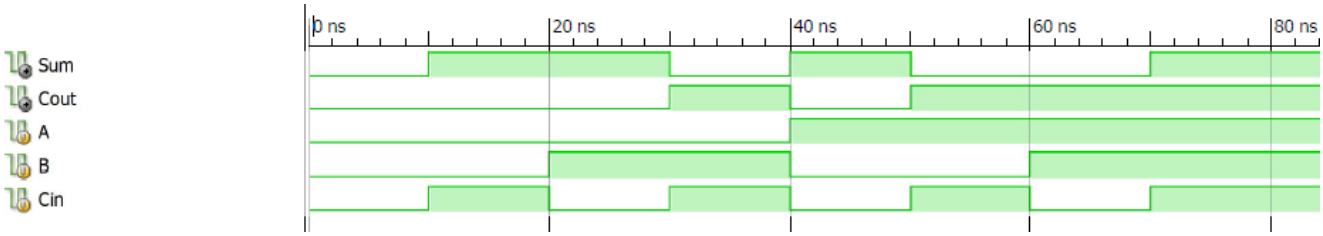
```
module full_adder(
   input wire A,    // First input bit
   input wire B,    // Second input bit
   input wire Cin,  // Carry-in bit
   output wire Sum, // Sum output
   output wire Cout // Carry-out output
);

   // Sum = A XOR B XOR Cin
   assign Sum = A ^ B ^ Cin;

   // Carry-out = (A AND B) OR (B AND Cin) OR (A AND Cin)
   assign Cout = (A & B) | (B & Cin) | (A & Cin);

endmodule

module test_full_adder;

   // Inputs to the full adder
   reg A;
   reg B;
   reg Cin;
```

```
// Outputs from the full adder
wire Sum;
wire Cout;

// Instantiate the Full Adder module
full_adder uut (
    .A(A),
    .B(B),
    .Cin(Cin),
    .Sum(Sum),
    .Cout(Cout)
);

// Test cases
initial begin
    // Display header
    $display("A B Cin | Sum Cout");
    $display("----------------");

    // Test Case 1: A = 0, B = 0, Cin = 0
    A = 0; B = 0; Cin = 0;
    #10; // Wait for 10 time units
    $display("%b %b  %b  |  %b   %b", A, B, Cin, Sum, Cout);

    // Test Case 2: A = 0, B = 0, Cin = 1
    A = 0; B = 0; Cin = 1;
    #10;
```

```
$display("%b %b  %b  |  %b   %b", A, B, Cin, Sum, Cout);

// Test Case 3: A = 0, B = 1, Cin = 0
A = 0; B = 1; Cin = 0;
#10;
$display("%b %b  %b  |  %b   %b", A, B, Cin, Sum, Cout);

// Test Case 4: A = 0, B = 1, Cin = 1
A = 0; B = 1; Cin = 1;
#10;
$display("%b %b  %b  |  %b   %b", A, B, Cin, Sum, Cout);

// Test Case 5: A = 1, B = 0, Cin = 0
A = 1; B = 0; Cin = 0;
#10;
$display("%b %b  %b  |  %b   %b", A, B, Cin, Sum, Cout);

// Test Case 6: A = 1, B = 0, Cin = 1
A = 1; B = 0; Cin = 1;
#10;
$display("%b %b  %b  |  %b   %b", A, B, Cin, Sum, Cout);

// Test Case 7: A = 1, B = 1, Cin = 0
A = 1; B = 1; Cin = 0;
#10;
$display("%b %b  %b  |  %b   %b", A, B, Cin, Sum, Cout);

// Test Case 8: A = 1, B = 1, Cin = 1
```

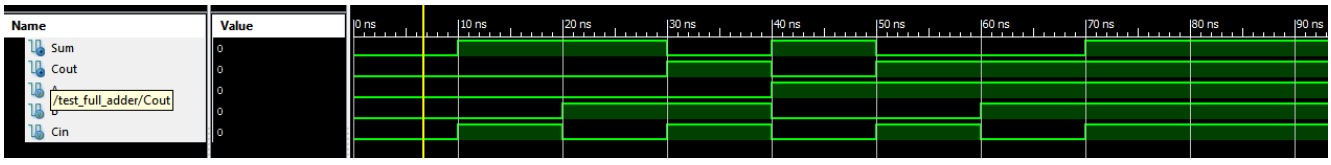
```
    A = 1; B = 1; Cin = 1;
    #10;
    $display("%b %b  %b  |  %b   %b", A, B, Cin, Sum, Cout);

    // End the simulation
    ;
  end

endmodule
```

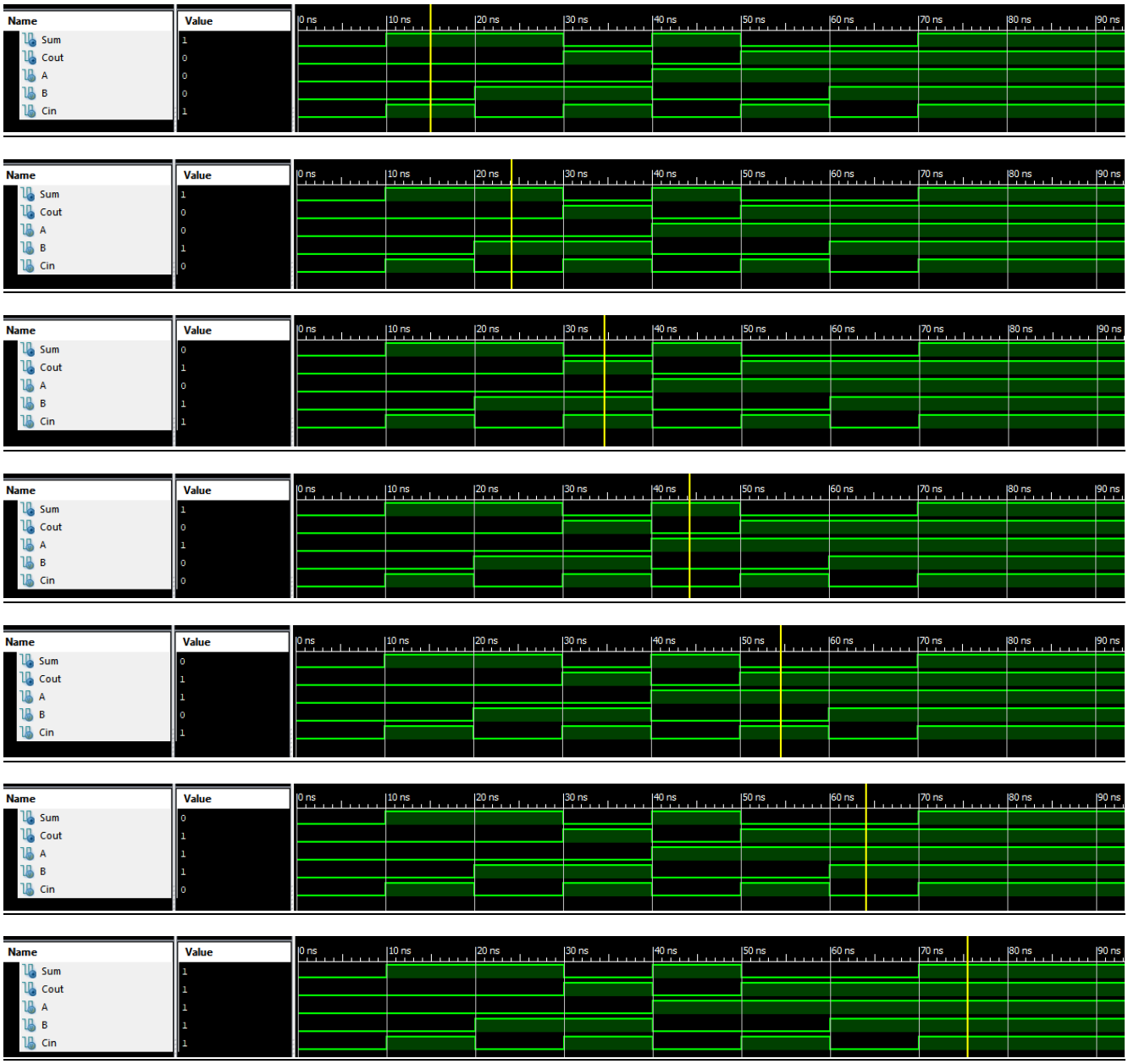
Output Waveform



Output Data

```
A B Cin | Sum Cout
-----------------
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
```





Full Adder using Half Adder

```
module half_adder(
   input wire A,    // First input bit
   input wire B,    // Second input bit
   output wire Sum, // Sum output
   output wire Carry // Carry output
```

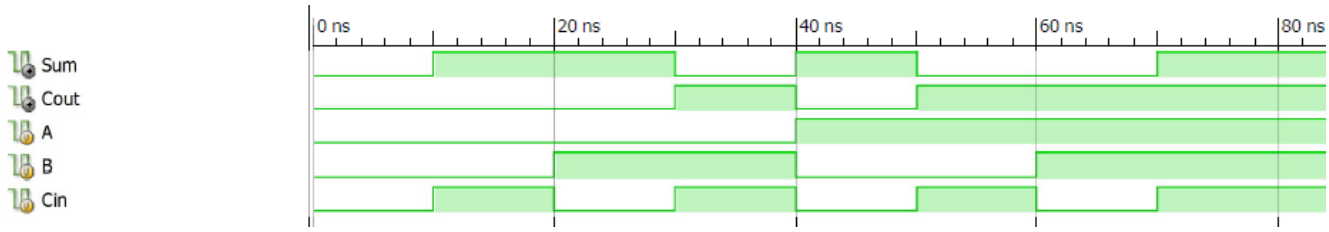
```
);

   // Sum = A XOR B
   assign Sum = A ^ B;

   // Carry = A AND B
   assign Carry = A & B;

endmodule

module full_adder(
   input wire A,    // First input bit
   input wire B,    // Second input bit
   input wire Cin,  // Carry-in bit
   output wire Sum, // Sum output
   output wire Cout // Carry-out output
);

   wire Sum1, Carry1, Carry2;

   // First Half Adder: Adds A and B
   half_adder HA1 (
      .A(A),
      .B(B),
      .Sum(Sum1),
      .Carry(Carry1)
   );
```

```
    // Second Half Adder: Adds Sum1 and Cin
    half_adder HA2 (
        .A(Sum1),
        .B(Cin),
        .Sum(Sum),
        .Carry(Carry2)
    );

    // OR gate for Carry-out
    assign Cout = Carry1 | Carry2;

endmodule

module test_full_adder;

    // Inputs to the full adder
    reg A;
    reg B;
    reg Cin;

    // Outputs from the full adder
    wire Sum;
    wire Cout;

    // Instantiate the Full Adder module
    full_adder uut (
        .A(A),
        .B(B),
```

```
    .Cin(Cin),
    .Sum(Sum),
    .Cout(Cout)
);

// Test cases
initial begin
    // Display header
    $display("A B Cin | Sum Cout");
    $display("-----------------");

    // Test Case 1: A = 0, B = 0, Cin = 0
    A = 0; B = 0; Cin = 0;
    #10; // Wait for 10 time units
    $display("%b %b  %b  |  %b   %b", A, B, Cin, Sum, Cout);

    // Test Case 2: A = 0, B = 0, Cin = 1
    A = 0; B = 0; Cin = 1;
    #10;
    $display("%b %b  %b  |  %b   %b", A, B, Cin, Sum, Cout);

    // Test Case 3: A = 0, B = 1, Cin = 0
    A = 0; B = 1; Cin = 0;
    #10;
    $display("%b %b  %b  |  %b   %b", A, B, Cin, Sum, Cout);

    // Test Case 4: A = 0, B = 1, Cin = 1
    A = 0; B = 1; Cin = 1;
```

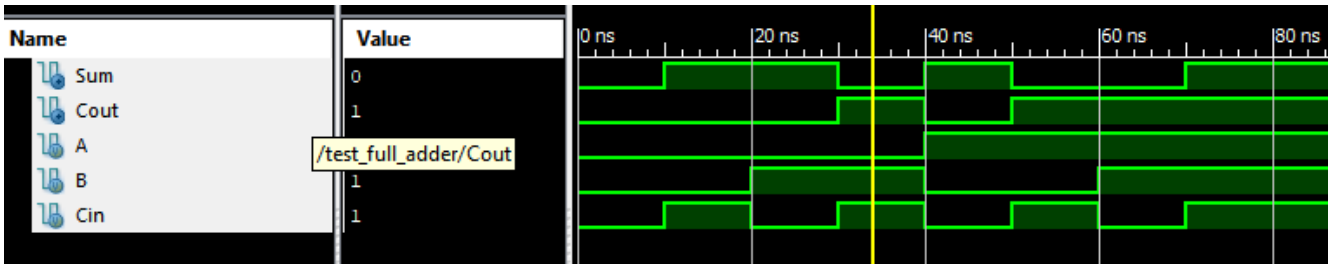
```
    #10;
    $display("%b %b  %b  |  %b   %b", A, B, Cin, Sum, Cout);

    // Test Case 5: A = 1, B = 0, Cin = 0
    A = 1; B = 0; Cin = 0;
    #10;
    $display("%b %b  %b  |  %b   %b", A, B, Cin, Sum, Cout);

    // Test Case 6: A = 1, B = 0, Cin = 1
    A = 1; B = 0; Cin = 1;
    #10;
    $display("%b %b  %b  |  %b   %b", A, B, Cin, Sum, Cout);

    // Test Case 7: A = 1, B = 1, Cin = 0
    A = 1; B = 1; Cin = 0;
    #10;
    $display("%b %b  %b  |  %b   %b", A, B, Cin, Sum, Cout);

    // Test Case 8: A = 1, B = 1, Cin = 1
    A = 1; B = 1; Cin = 1;
    #10;
    $display("%b %b  %b  |  %b   %b", A, B, Cin, Sum, Cout);

    // End the simulation
    ;
  end

endmodule
```

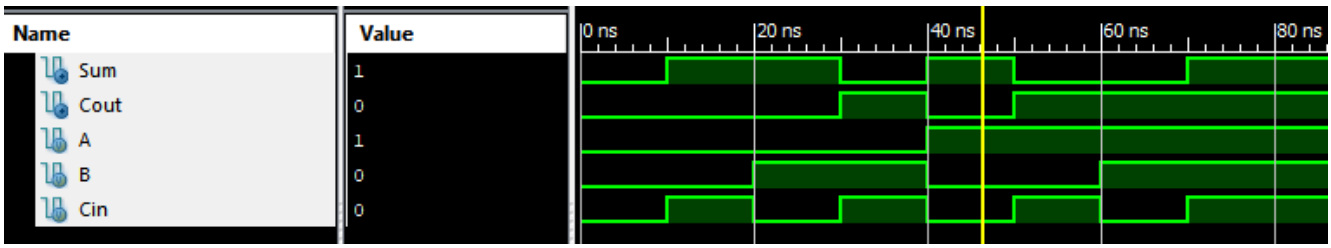
Output Waveform



Output Data

A B Cin | Sum Cout
---
0 0 0 | 0 0
0 0 1 | 1 0
0 1 0 | 1 0
0 1 1 | 0 1
1 0 0 | 1 0
1 0 1 | 0 1
1 1 0 | 0 1
1 1 1 | 1 1





Half Subtractor

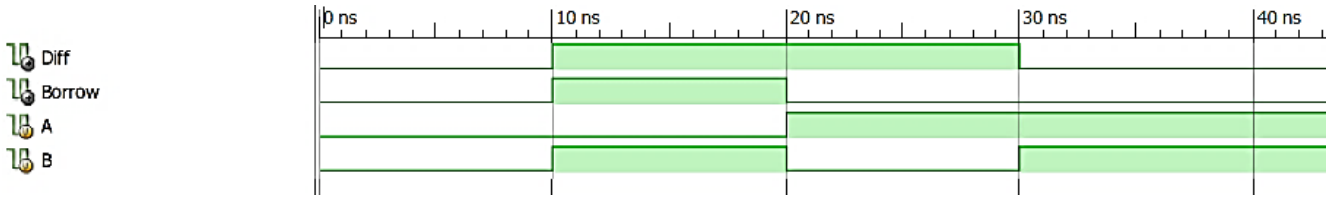
```
module half_subtractor(
  input wire A,    // Minuend (first input bit)
```

```
    input wire B,    // Subtrahend (second input bit)
    output wire Diff, // Difference output
    output wire Borrow // Borrow output
);

    // Difference = A XOR B
    assign Diff = A ^ B;

    // Borrow = NOT A AND B
    assign Borrow = (~A) & B;

endmodule

module test_half_subtractor;

    // Inputs to the half subtractor
    reg A;
    reg B;

    // Outputs from the half subtractor
    wire Diff;
    wire Borrow;

    // Instantiate the Half Subtractor module
    half_subtractor uut (
        .A(A),
        .B(B),
        .Diff(Diff),
```

```
    .Borrow(Borrow)
);

// Test cases
initial begin
    // Display header
    $display("A B | Diff Borrow");
    $display("-----------------");

    // Test Case 1: A = 0, B = 0
    A = 0; B = 0;
    #10; // Wait for 10 time units
    $display("%b %b |  %b    %b", A, B, Diff, Borrow);

    // Test Case 2: A = 0, B = 1
    A = 0; B = 1;
    #10;
    $display("%b %b |  %b    %b", A, B, Diff, Borrow);

    // Test Case 3: A = 1, B = 0
    A = 1; B = 0;
    #10;
    $display("%b %b |  %b    %b", A, B, Diff, Borrow);

    // Test Case 4: A = 1, B = 1
    A = 1; B = 1;
    #10;
    $display("%b %b |  %b    %b", A, B, Diff, Borrow);
```

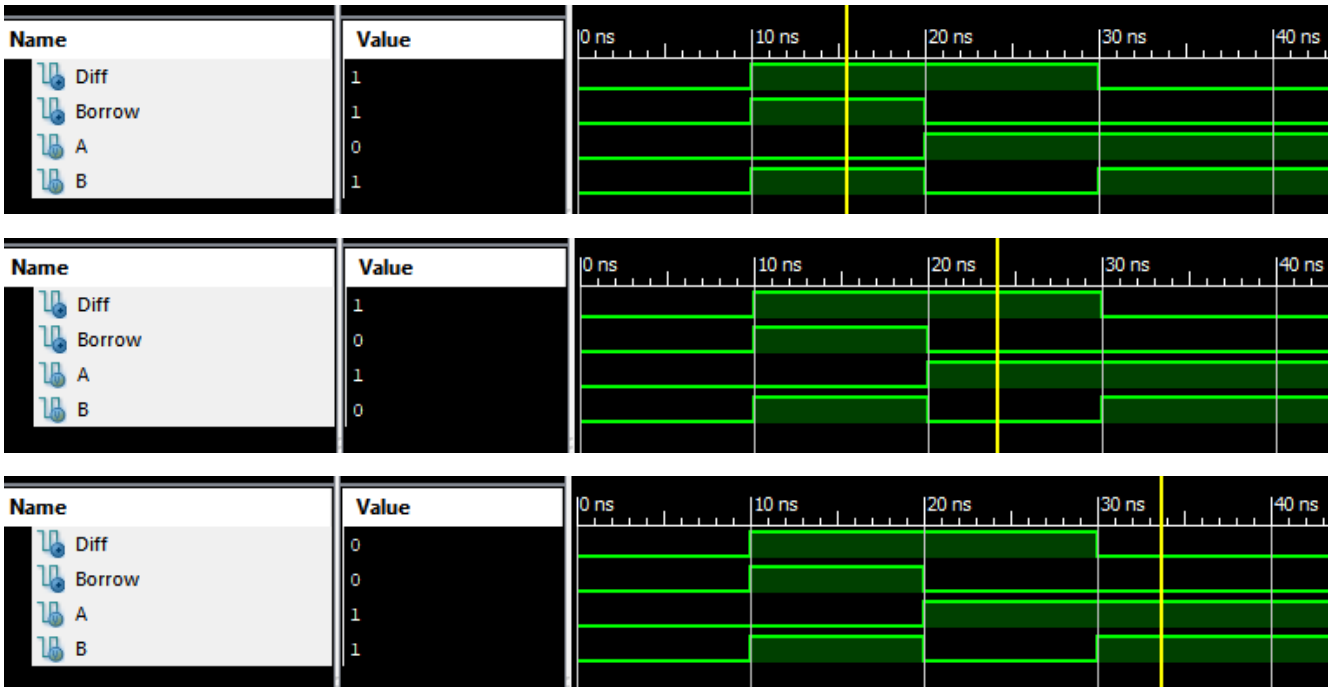
```
    // End the simulation
    ;
  end

endmodule
```

Output Waveform:



Output Data

A B | Diff Borrow
---
0 0 | 0 0
0 1 | 1 1
1 0 | 1 0
1 1 | 0 0



Full Subtractor

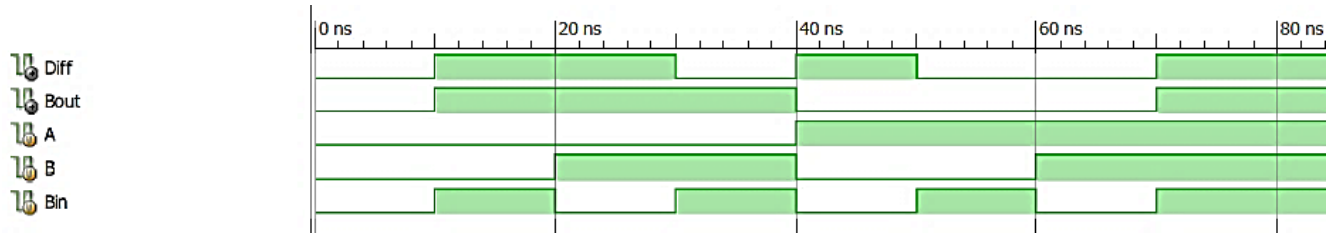
```
module full_subtractor(
   input wire A,      // Minuend (first input bit)
   input wire B,      // Subtrahend (second input bit)
   input wire Bin,    // Borrow-in bit
   output wire Diff,  // Difference output
   output wire Bout   // Borrow-out output
);

   // Difference = A XOR B XOR Bin
   assign Diff = A ^ B ^ Bin;

   // Borrow-out = (~A & B) | ((~A | B) & Bin)
   assign Bout = (~A & B) | ((~A | B) & Bin);

endmodule

module test_full_subtractor;

   // Inputs to the full subtractor
   reg A;
   reg B;
   reg Bin;

   // Outputs from the full subtractor
   wire Diff;
   wire Bout;
```

```
// Instantiate the Full Subtractor module
full_subtractor uut (
    .A(A),
    .B(B),
    .Bin(Bin),
    .Diff(Diff),
    .Bout(Bout)
);

// Test cases
initial begin
    // Display header
    $display("A B Bin | Diff Bout");
    $display("------------------");

    // Test Case 1: A = 0, B = 0, Bin = 0
    A = 0; B = 0; Bin = 0;
    #10; // Wait for 10 time units
    $display("%b %b  %b  |  %b    %b", A, B, Bin, Diff, Bout);

    // Test Case 2: A = 0, B = 0, Bin = 1
    A = 0; B = 0; Bin = 1;
    #10;
    $display("%b %b  %b  |  %b    %b", A, B, Bin, Diff, Bout);

    // Test Case 3: A = 0, B = 1, Bin = 0
    A = 0; B = 1; Bin = 0;
```

```
#10;
$display("%b %b  %b  |  %b    %b", A, B, Bin, Diff, Bout);

// Test Case 4: A = 0, B = 1, Bin = 1
A = 0; B = 1; Bin = 1;
#10;
$display("%b %b  %b  |  %b    %b", A, B, Bin, Diff, Bout);

// Test Case 5: A = 1, B = 0, Bin = 0
A = 1; B = 0; Bin = 0;
#10;
$display("%b %b  %b  |  %b    %b", A, B, Bin, Diff, Bout);

// Test Case 6: A = 1, B = 0, Bin = 1
A = 1; B = 0; Bin = 1;
#10;
$display("%b %b  %b  |  %b    %b", A, B, Bin, Diff, Bout);

// Test Case 7: A = 1, B = 1, Bin = 0
A = 1; B = 1; Bin = 0;
#10;
$display("%b %b  %b  |  %b    %b", A, B, Bin, Diff, Bout);

// Test Case 8: A = 1, B = 1, Bin = 1
A = 1; B = 1; Bin = 1;
#10;
$display("%b %b  %b  |  %b    %b", A, B, Bin, Diff, Bout);
```

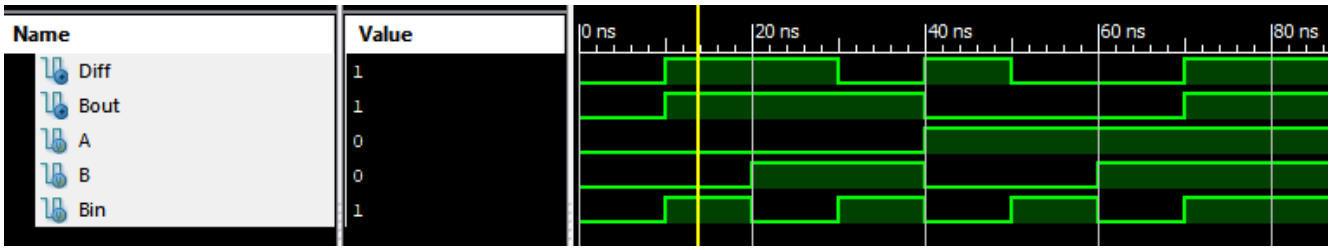
```
    // End the simulation
    ;
  end

endmodule
```

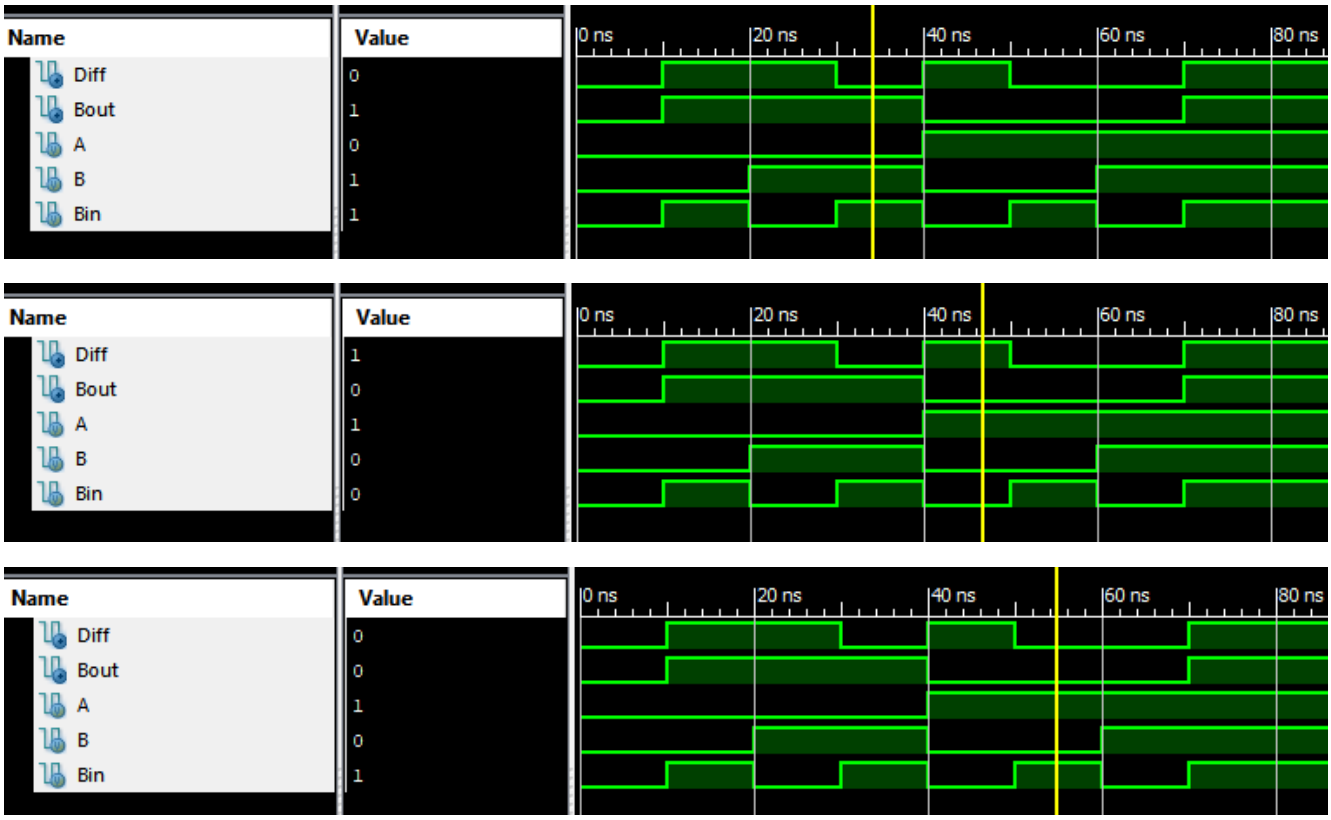
Output Waveform:



Output Data

A B Bin | Diff Bout
---
0 0 0 | 0 0
0 0 1 | 1 1
0 1 0 | 1 1
0 1 1 | 0 1
1 0 0 | 1 0
1 0 1 | 0 0
1 1 0 | 0 0
1 1 1 | 1 1





Full Subtractor using Half Subtractor

```
module half_subtractor(
   input wire A,      // Minuend
   input wire B,      // Subtrahend
   output wire Diff,  // Difference output
   output wire Borrow  // Borrow output
);

   assign Diff = A ^ B;
   assign Borrow = (~A) & B;

endmodule
```

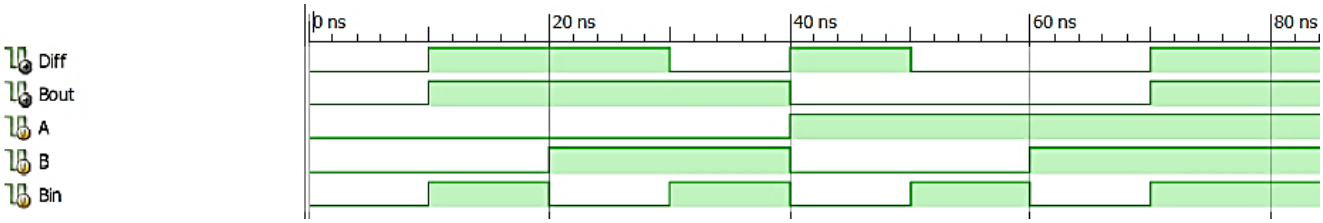
```
module full_subtractor(
   input wire A,      // Minuend
   input wire B,      // Subtrahend
   input wire Bin,    // Borrow-in
   output wire Diff,  // Difference output
   output wire Bout   // Borrow-out output
);

   // Internal signals
   wire Diff1;
   wire Borrow1;
   wire Borrow2;

   // First half subtractor
   half_subtractor hs1 (
      .A(A),
      .B(B),
      .Diff(Diff1),
      .Borrow(Borrow1)
   );

   // Second half subtractor
   half_subtractor hs2 (
      .A(Diff1),
      .B(Bin),
      .Diff(Diff),
      .Borrow(Borrow2)
```

```
    );

    // Final borrow-out
    assign Bout = Borrow1 | Borrow2;

endmodule

module test_full_subtractor;

    // Inputs to the full subtractor
    reg A;
    reg B;
    reg Bin;

    // Outputs from the full subtractor
    wire Diff;
    wire Bout;

    // Instantiate the Full Subtractor module
    full_subtractor uut (
        .A(A),
        .B(B),
        .Bin(Bin),
        .Diff(Diff),
        .Bout(Bout)
    );

    // Test cases
```

```
initial begin
    // Display header
    $display("A B Bin | Diff Bout");
    $display("-----------------");

    // Test Case 1: A = 0, B = 0, Bin = 0
    A = 0; B = 0; Bin = 0;
    #10; // Wait for 10 time units
    $display("%b %b  %b  |  %b    %b", A, B, Bin, Diff, Bout);

    // Test Case 2: A = 0, B = 0, Bin = 1
    A = 0; B = 0; Bin = 1;
    #10;
    $display("%b %b  %b  |  %b    %b", A, B, Bin, Diff, Bout);

    // Test Case 3: A = 0, B = 1, Bin = 0
    A = 0; B = 1; Bin = 0;
    #10;
    $display("%b %b  %b  |  %b    %b", A, B, Bin, Diff, Bout);

    // Test Case 4: A = 0, B = 1, Bin = 1
    A = 0; B = 1; Bin = 1;
    #10;
    $display("%b %b  %b  |  %b    %b", A, B, Bin, Diff, Bout);

    // Test Case 5: A = 1, B = 0, Bin = 0
    A = 1; B = 0; Bin = 0;
    #10;
```

```
    $display("%b %b  %b  |  %b    %b", A, B, Bin, Diff, Bout);

    // Test Case 6: A = 1, B = 0, Bin = 1
    A = 1; B = 0; Bin = 1;
    #10;
    $display("%b %b  %b  |  %b    %b", A, B, Bin, Diff, Bout);

    // Test Case 7: A = 1, B = 1, Bin = 0
    A = 1; B = 1; Bin = 0;
    #10;
    $display("%b %b  %b  |  %b    %b", A, B, Bin, Diff, Bout);

    // Test Case 8: A = 1, B = 1, Bin = 1
    A = 1; B = 1; Bin = 1;
    #10;
    $display("%b %b  %b  |  %b    %b", A, B, Bin, Diff, Bout);

    // End the simulation
    ;
  end

endmodule
```

Output Waveform:

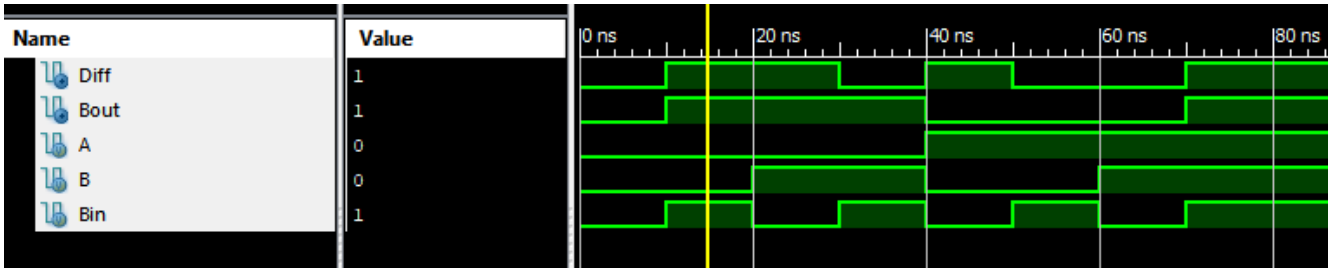


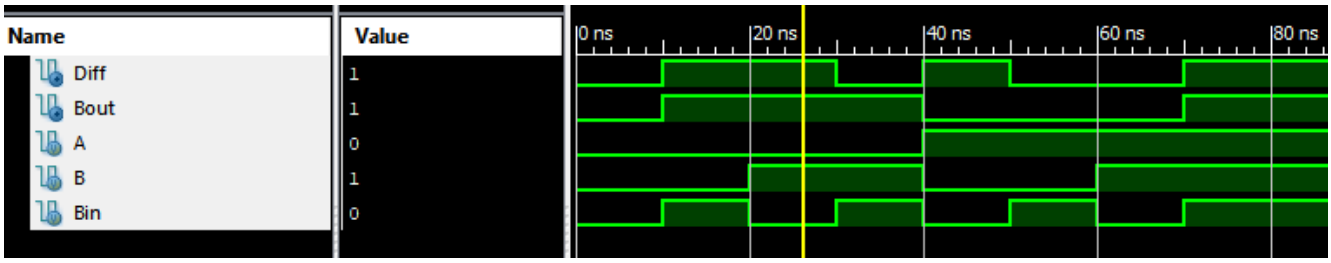
Output Data:

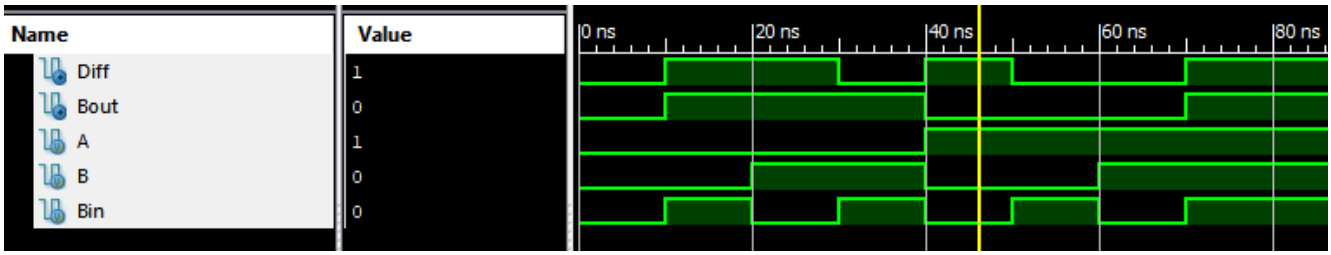
A B Bin | Diff Bout

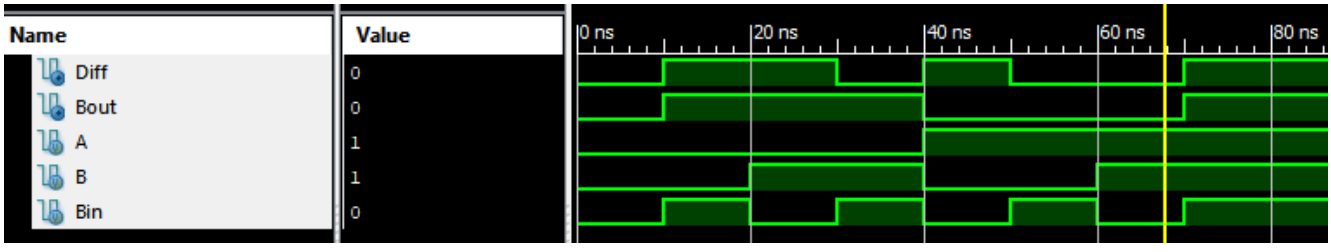
---

0 0 0 | 0 0
0 0 1 | 1 1
0 1 0 | 1 1
0 1 1 | 0 1
1 0 0 | 1 0
1 0 1 | 0 0
1 1 0 | 0 0
1 1 1 | 1 1









Experiment No. 3

Realize 4-bit ALU using Verilog program.

Theory:

Write related Theory as discussed.

Verilog Code:

```
module alu_4bit(
   input [3:0] A,      // 4-bit input A
   input [3:0] B,      // 4-bit input B
   input [2:0] ALU_Sel, // 3-bit control signal to select the ALU operation
   output reg [3:0] ALU_Out, // 4-bit output of the ALU
   output reg Carry_Out // Carry/Overflow flag
);

   always @(*) begin
      case(ALU_Sel)
         3'b000: {Carry_Out, ALU_Out} = A + B;     // Addition
         3'b001: {Carry_Out, ALU_Out} = A - B;     // Subtraction
         3'b010: ALU_Out = A & B;                  // AND
         3'b011: ALU_Out = A | B;                  // OR
         3'b100: ALU_Out = A ^ B;                  // XOR
         3'b101: ALU_Out = ~A;                     // NOT
         3'b110: ALU_Out = A << 1;                 // Left Shift
         3'b111: ALU_Out = A >> 1;                 // Right Shift
         default: ALU_Out = 4'b0000;               // Default case
      endcase
   end
```

```
endmodule
```

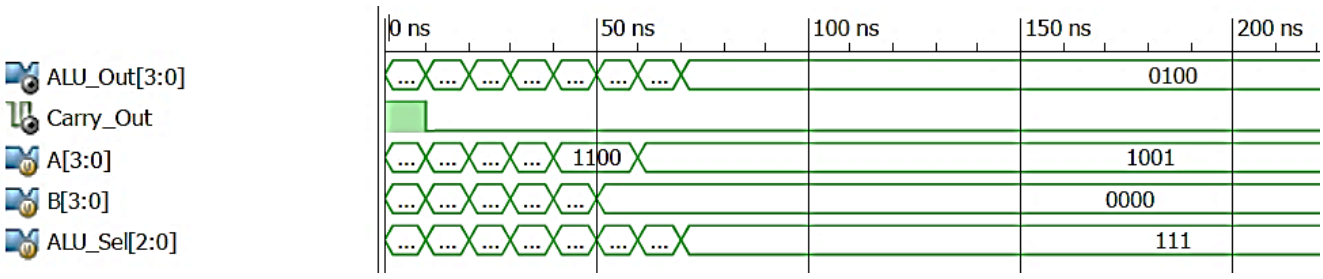
Test Bench:

```
module test_alu_4bit;

  // Inputs
  reg [3:0] A;
  reg [3:0] B;
  reg [2:0] ALU_Sel;

  // Outputs
  wire [3:0] ALU_Out;
  wire Carry_Out;

  // Instantiate the ALU module
  alu_4bit uut (
    .A(A),
    .B(B),
    .ALU_Sel(ALU_Sel),
    .ALU_Out(ALU_Out),
    .Carry_Out(Carry_Out)
  );

  // Test cases
  initial begin
    // Display header
    $display("A    B    ALU_Sel | ALU_Out Carry_Out");
    $display("-----------------------------------");
```

```
// Test Case 1: Addition (A = 4'b0011, B = 4'b0101, ALU_Sel = 3'b000)
A = 4'b1111; B = 4'b0101; ALU_Sel = 3'b000;
#10;
$display("%b %b  %b    |  %b     %b", A, B, ALU_Sel, ALU_Out, Carry_Out);

// Test Case 2: Subtraction (A = 4'b0110, B = 4'b0011, ALU_Sel = 3'b001)
A = 4'b0110; B = 4'b0011; ALU_Sel = 3'b001;
#10;
$display("%b %b  %b    |  %b     %b", A, B, ALU_Sel, ALU_Out, Carry_Out);

// Test Case 3: AND (A = 4'b1100, B = 4'b1010, ALU_Sel = 3'b010)
A = 4'b1100; B = 4'b1010; ALU_Sel = 3'b010;
#10;
$display("%b %b  %b    |  %b     %b", A, B, ALU_Sel, ALU_Out, Carry_Out);

// Test Case 4: OR (A = 4'b0011, B = 4'b0101, ALU_Sel = 3'b011)
A = 4'b0011; B = 4'b0101; ALU_Sel = 3'b011;
#10;
$display("%b %b  %b    |  %b     %b", A, B, ALU_Sel, ALU_Out, Carry_Out);

// Test Case 5: XOR (A = 4'b1100, B = 4'b1010, ALU_Sel = 3'b100)
A = 4'b1100; B = 4'b1010; ALU_Sel = 3'b100;
#10;
$display("%b %b  %b    |  %b     %b", A, B, ALU_Sel, ALU_Out, Carry_Out);

// Test Case 6: NOT (A = 4'b1100, ALU_Sel = 3'b101)
A = 4'b1100; B = 4'b0000; ALU_Sel = 3'b101; // B is not used in this case
```

```
    #10;
    $display("%b %b  %b    |  %b     %b", A, B, ALU_Sel, ALU_Out, Carry_Out);

    // Test Case 7: Left Shift (A = 4'b1001, ALU_Sel = 3'b110)
    A = 4'b1001; B = 4'b0000; ALU_Sel = 3'b110; // B is not used in this case
    #10;
    $display("%b %b  %b    |  %b     %b", A, B, ALU_Sel, ALU_Out, Carry_Out);

    // Test Case 8: Right Shift (A = 4'b1001, ALU_Sel = 3'b111)
    A = 4'b1001; B = 4'b0000; ALU_Sel = 3'b111; // B is not used in this case
    #10;
    $display("%b %b  %b    |  %b     %b", A, B, ALU_Sel, ALU_Out, Carry_Out);

    // End the simulation
    ;
  end

endmodule
```

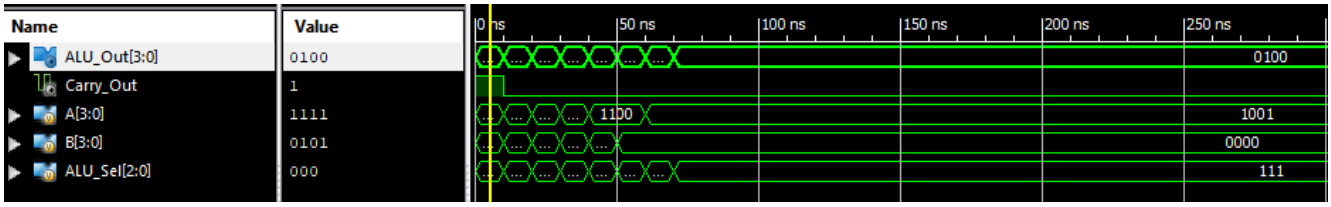
Output Waveform:

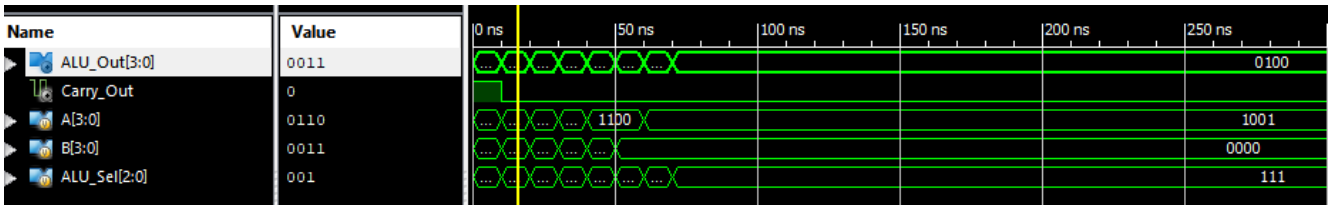


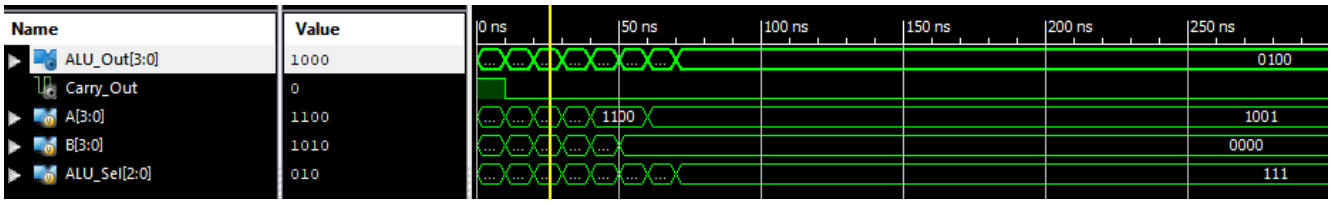
Output Data:

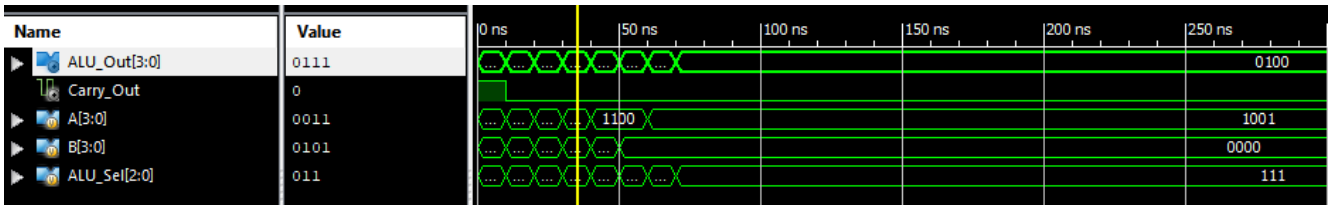
A B ALU_Sel | ALU_Out Carry_Out

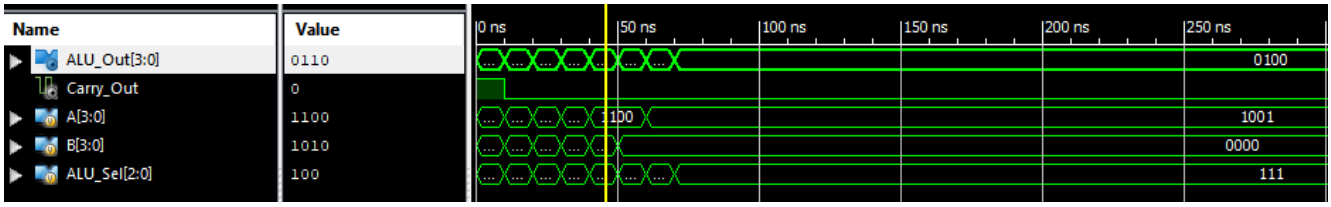
1111 0101 000 | 0100 1
0110 0011 001 | 0011 0
1100 1010 010 | 1000 0
0011 0101 011 | 0111 0
1100 1010 100 | 0110 0
1100 0000 101 | 0011 0
1001 0000 110 | 0010 0
1001 0000 111 | 0100 0

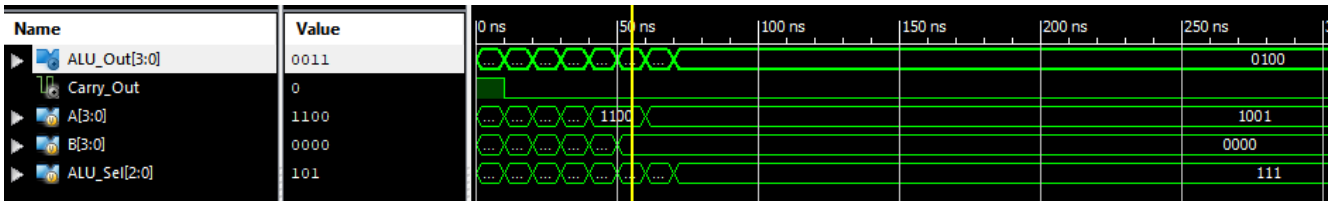


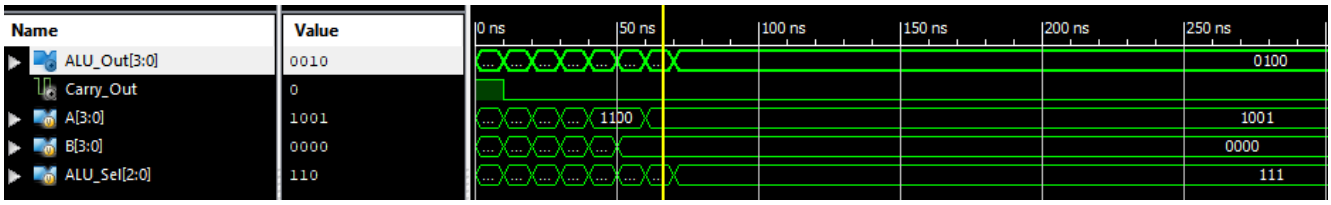


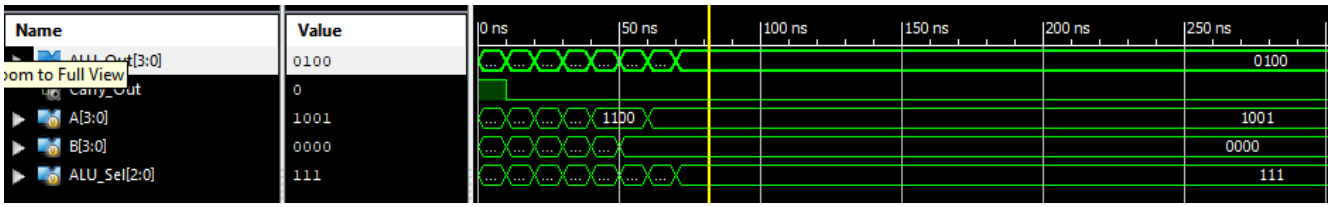







Name
Value
ALU_Out[3:0]
0011
Carry_Out
0
A[3:0]
1100
B[3:0]
0000
ALU_Sel[2:0]
101
0 ns
50 ns
100 ns
150 ns
200 ns
250 ns
0100
1100
1001
0000
111


Name
Value
ALU_Out[3:0]
0010
Carry_Out
0
A[3:0]
1001
B[3:0]
0000
ALU_Sel[2:0]
110
0 ns
50 ns
100 ns
150 ns
200 ns
250 ns
0100
1100
1001
0000
111


Name
Value
0100
oom to Full View
0
A[3:0]
1001
B[3:0]
0000
ALU_Sel[2:0]
111
0 ns
50 ns
100 ns
150 ns
200 ns
250 ns
0100
1100
1001
0000
111

Experiment No. 4

Realize the following Code converters using Verilog Behavioral description.

a) Gray to binary and vice versa b) Binary to excess3 and vice versa

Theory:

4-Bit Gray to Binary Converter

Verilog Code:

```
`timescale 1ns / 1ps
module gray_to_binary_4bit (
   input wire [3:0] gray,  // 4-bit Gray code input
   output reg [3:0] binary // 4-bit Binary code output
);

   always @(*) begin
      binary[3] = gray[3];              // MSB remains the same
      binary[2] = binary[3] ^ gray[2];      // XOR of previous binary bit and gray bit
      binary[1] = binary[2] ^ gray[1];      // Continue XOR process
      binary[0] = binary[1] ^ gray[0];      // LSB calculation
   end

endmodule
```

Test Bench:

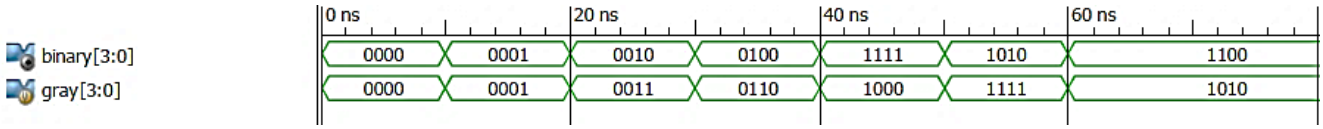
```
`timescale 1ns / 1ps
module test_gray_to_binary_4bit;
```

```
// Inputs
reg [3:0] gray;

// Outputs
wire [3:0] binary;

// Instantiate the Gray to Binary conversion module
gray_to_binary_4bit uut (
    .gray(gray),
    .binary(binary)
);

// Test cases
initial begin
    // Display header
    $display("Gray | Binary");
    $display("-------------");

    // Test Case 1: Gray code 0000
    gray = 4'b0000;
    #10;
    $display("%b | %b", gray, binary);

    // Test Case 2: Gray code 0001
    gray = 4'b0001;
    #10;
    $display("%b | %b", gray, binary);
```

```
    // Test Case 3: Gray code 0011
    gray = 4'b0011;
    #10;
    $display("%b | %b", gray, binary);

    // Test Case 4: Gray code 0110
    gray = 4'b0110;
    #10;
    $display("%b | %b", gray, binary);

    // Test Case 5: Gray code 1000
    gray = 4'b1000;
    #10;
    $display("%b | %b", gray, binary);

    // Test Case 6: Gray code 1111
    gray = 4'b1111;
    #10;
    $display("%b | %b", gray, binary);

    // Test Case 7: Gray code 1010
    gray = 4'b1010;
    #10;
    $display("%b | %b", gray, binary);

    // End the simulation
    ;
end
```

endmodule

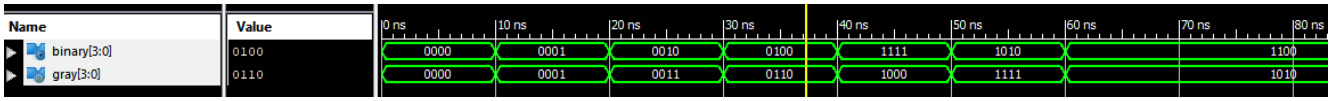
Output Waveform:



Output Data:

Gray | Binary

0000 | 0000
0001 | 0001
0011 | 0010
0110 | 0100
1000 | 1111
1111 | 1010
1010 | 1100



3-Bit Binary to Gray Code Converter

Verilog Code:

```
module binary_to_gray(
   input wire [2:0] binary,   // 3-bit binary input
   output wire [2:0] gray     // 3-bit gray code output
);

   // Gray code generation
   assign gray[2] = binary[2];             // MSB is the same
   assign gray[1] = binary[2] ^ binary[1];   // XOR of MSB and next bit
   assign gray[0] = binary[1] ^ binary[0];   // XOR of the next bit and LSB

endmodule
```

Test Bench:

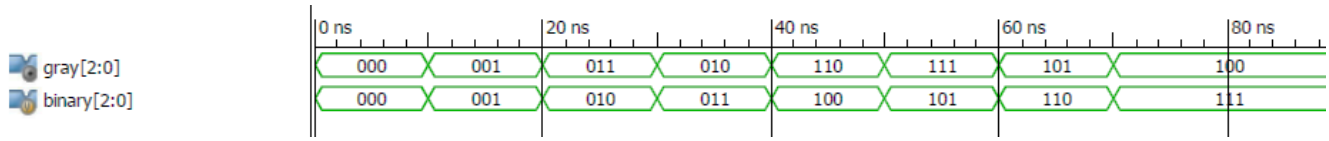
```
module tb_binary_to_gray;
  // Testbench signals
  reg [2:0] binary;       // 3-bit binary input
  wire [2:0] gray;        // 3-bit gray code output

  // Instantiate the converter module
  binary_to_gray uut (
    .binary(binary),
    .gray(gray)
  );

  // Test procedure
  initial begin
    // Monitor signals
    $monitor("Time=%0t, Binary=%b, Gray=%b", $time, binary, gray);

    // Test all possible 3-bit binary values
    binary = 3'b000; #10;
    binary = 3'b001; #10;
    binary = 3'b010; #10;
    binary = 3'b011; #10;
    binary = 3'b100; #10;
    binary = 3'b101; #10;
    binary = 3'b110; #10;
    binary = 3'b111; #10;

    // Finish the simulation
```

;

end

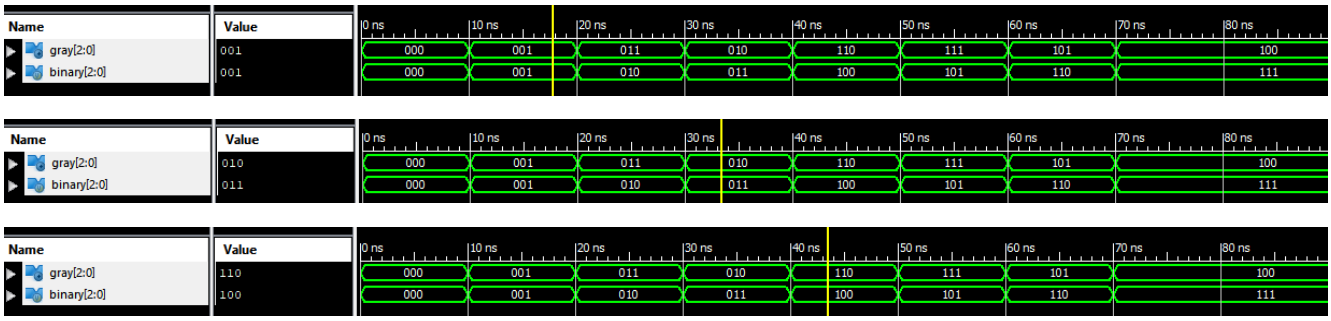
endmodule

Output Waveform:



Output Data:

Time=0, Binary=000, Gray=000
Time=10000, Binary=001, Gray=001
Time=20000, Binary=010, Gray=011
Time=30000, Binary=011, Gray=010
Time=40000, Binary=100, Gray=110
Time=50000, Binary=101, Gray=111
Time=60000, Binary=110, Gray=101
Time=70000, Binary=111, Gray=100



4-Bit Binary to Excess-3 Code Converter

Verilog Code:

```
module binary_to_excess3 (
   input wire [3:0] binary,  // 4-bit Binary code input
   output reg [3:0] excess3  // 4-bit Excess-3 code output
);
```

```
  always @(*) begin
    excess3 = binary + 4'b0011;  // Add 3 (0011) to the binary input
  end

endmodule
```

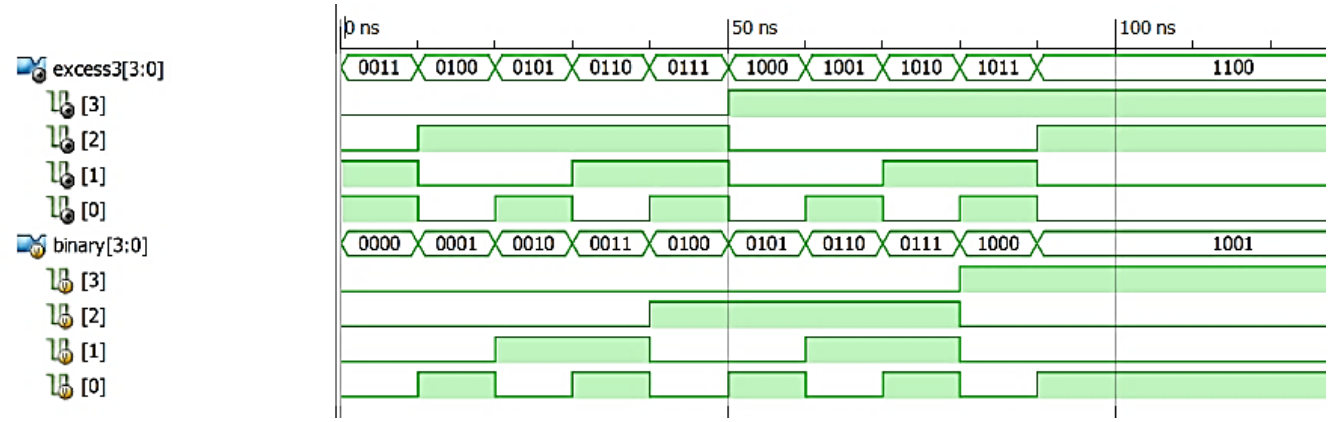
Test Bench:

```
module test_binary_to_excess3;

  // Inputs
  reg [3:0] binary;

  // Outputs
  wire [3:0] excess3;

  // Instantiate the Binary to Excess-3 conversion module
  binary_to_excess3 uut (
    .binary(binary),
    .excess3(excess3)
  );

  // Test cases
  initial begin
    // Display header
    $display("Binary | Excess-3");
    $display("----------------");

    // Test Case 1: Binary 0000 (Decimal 0)
```

```
binary = 4'b0000;
#10;
$display("%b    | %b", binary, excess3);

// Test Case 2: Binary 0001 (Decimal 1)
binary = 4'b0001;
#10;
$display("%b    | %b", binary, excess3);

// Test Case 3: Binary 0010 (Decimal 2)
binary = 4'b0010;
#10;
$display("%b    | %b", binary, excess3);

// Test Case 4: Binary 0011 (Decimal 3)
binary = 4'b0011;
#10;
$display("%b    | %b", binary, excess3);

// Test Case 5: Binary 0100 (Decimal 4)
binary = 4'b0100;
#10;
$display("%b    | %b", binary, excess3);

// Test Case 6: Binary 0101 (Decimal 5)
binary = 4'b0101;
#10;
$display("%b    | %b", binary, excess3);
```

```
    // Test Case 7: Binary 0110 (Decimal 6)
    binary = 4'b0110;
    #10;
    $display("%b    | %b", binary, excess3);

    // Test Case 8: Binary 0111 (Decimal 7)
    binary = 4'b0111;
    #10;
    $display("%b    | %b", binary, excess3);

    // Test Case 9: Binary 1000 (Decimal 8)
    binary = 4'b1000;
    #10;
    $display("%b    | %b", binary, excess3);

    // Test Case 10: Binary 1001 (Decimal 9)
    binary = 4'b1001;
    #10;
    $display("%b    | %b", binary, excess3);

    // End the simulation
    ;
  end

endmodule
```

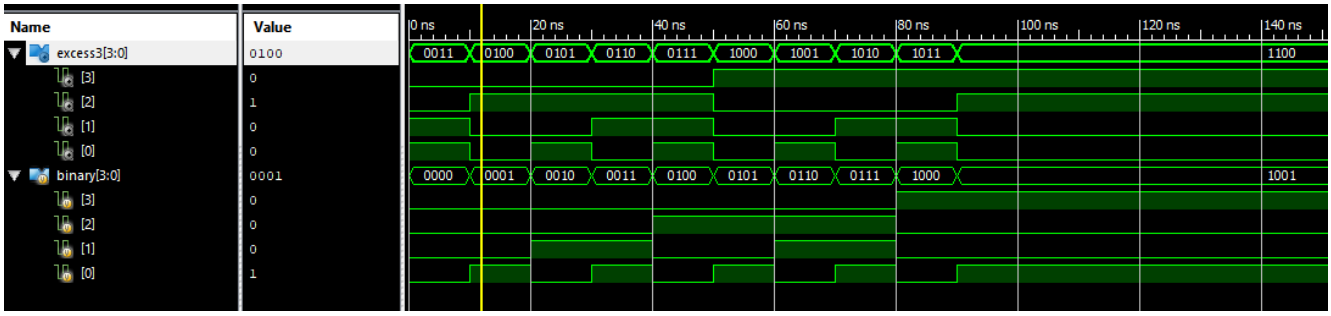
Output Waveform:



Output Data:

Binary | Excess-3

0000 | 0011
0001 | 0100
0010 | 0101
0011 | 0110
0100 | 0111
0101 | 1000
0110 | 1001
0111 | 1010
1000 | 1011
1001 | 1100



3-Bit Excess-3 to Binary Code Converter

Verilog Code:

```
module excess3_to_binary (
  input wire [2:0] excess3,  // 3-bit Excess-3 code input
  output reg [2:0] binary    // 3-bit Binary code output
```

```
);

  always @(*) begin
    if (excess3 >= 3'b011)  // Check if Excess-3 code is valid (should be 011 or greater)
      binary = excess3 - 3'b011;  // Subtract 3 (011) from Excess-3 code to get Binary
    else
      binary = 3'b000;  // Invalid Excess-3 codes are mapped to binary 000
  end

endmodule
```

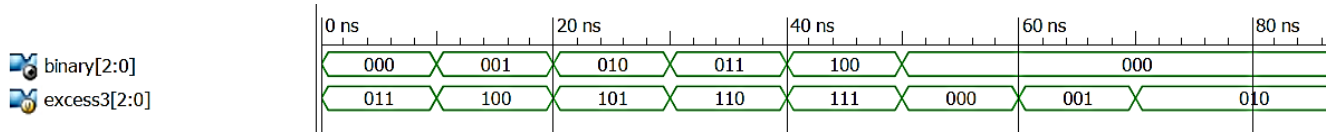
Test Bench:

```
module test_excess3_to_binary;

  // Inputs
  reg [2:0] excess3;

  // Outputs
  wire [2:0] binary;

  // Instantiate the Excess-3 to Binary conversion module
  excess3_to_binary uut (
    .excess3(excess3),
    .binary(binary)
  );

  // Test cases
  initial begin
    // Display header
```

```
$display("Excess-3 | Binary");
$display("----------------");

// Test Case 1: Excess-3 011 (Decimal 0)
excess3 = 3'b011;
#10;
$display("%b      | %b", excess3, binary);

// Test Case 2: Excess-3 100 (Decimal 1)
excess3 = 3'b100;
#10;
$display("%b      | %b", excess3, binary);

// Test Case 3: Excess-3 101 (Decimal 2)
excess3 = 3'b101;
#10;
$display("%b      | %b", excess3, binary);

// Test Case 4: Excess-3 110 (Decimal 3)
excess3 = 3'b110;
#10;
$display("%b      | %b", excess3, binary);

// Test Case 5: Excess-3 111 (Decimal 4)
excess3 = 3'b111;
#10;
$display("%b      | %b", excess3, binary);
```

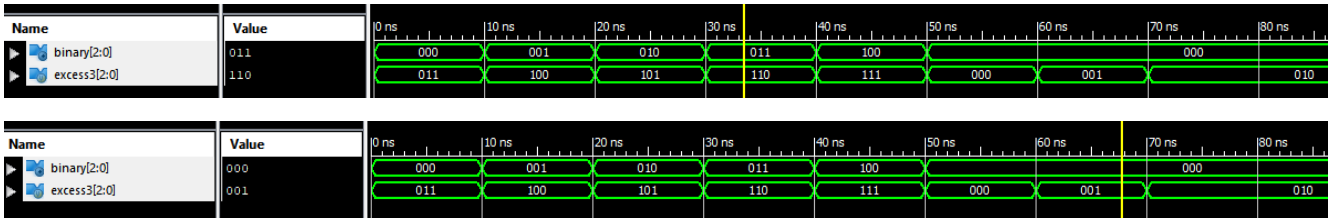
```
    // Test invalid cases
    $display("Invalid Excess-3 Codes:");
    excess3 = 3'b000; #10; $display("%b      | %b", excess3, binary);
    excess3 = 3'b001; #10; $display("%b      | %b", excess3, binary);
    excess3 = 3'b010; #10; $display("%b      | %b", excess3, binary);

    // End the simulation
    ;
  end

endmodule
```

Output Waveform:



Output Data:

```
Excess-3 | Binary
---------------
011 | 000
100 | 001
101 | 010
110 | 011
111 | 100
Invalid Excess-3 Codes:
000 | 000
001 | 000
010 | 000
```



Experiment No. 5

Realize 8:1mux, 8:3encoder, and Priority encoder using Verilog Behavioral description.

Theory:

8:1 Multiplexer (MUX)

Verilog Code:

```
module mux8to1(
   input wire [7:0] D,    // 8-bit data input (D0 to D7)
   input wire [2:0] S,    // 3-bit select input
   output reg Y           // Output
);

   always @(*) begin
      case (S)
         3'b000: Y = D[0];  // Select D0
         3'b001: Y = D[1];  // Select D1
         3'b010: Y = D[2];  // Select D2
         3'b011: Y = D[3];  // Select D3
         3'b100: Y = D[4];  // Select D4
         3'b101: Y = D[5];  // Select D5
         3'b110: Y = D[6];  // Select D6
         3'b111: Y = D[7];  // Select D7
         default: Y = 1'b0; // Default case
      endcase
   end

endmodule
```

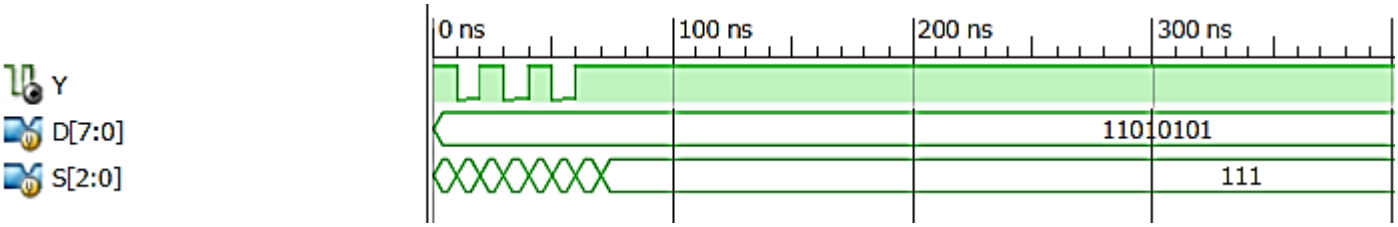
Test Bench:

```
module test_mux8to1;

  // Inputs
  reg [7:0] D;
  reg [2:0] S;

  // Output
  wire Y;

  // Instantiate the MUX module
  mux8to1 uut (
    .D(D),
    .S(S),
    .Y(Y)
  );

  // Test cases
  initial begin
    // Display header
    $display("S   | D       | Y");
    $display("--------------------");

    // Test Case 1: Select D0
    D = 8'b11010101; S = 3'b000;
    #10;
    $display("%b | %b | %b", S, D, Y);
```

```
// Test Case 2: Select D1
S = 3'b001;
#10;
$display("%b | %b | %b", S, D, Y);

// Test Case 3: Select D2
S = 3'b010;
#10;
$display("%b | %b | %b", S, D, Y);

// Test Case 4: Select D3
S = 3'b011;
#10;
$display("%b | %b | %b", S, D, Y);

// Test Case 5: Select D4
S = 3'b100;
#10;
$display("%b | %b | %b", S, D, Y);

// Test Case 6: Select D5
S = 3'b101;
#10;
$display("%b | %b | %b", S, D, Y);

// Test Case 7: Select D6
S = 3'b110;
```

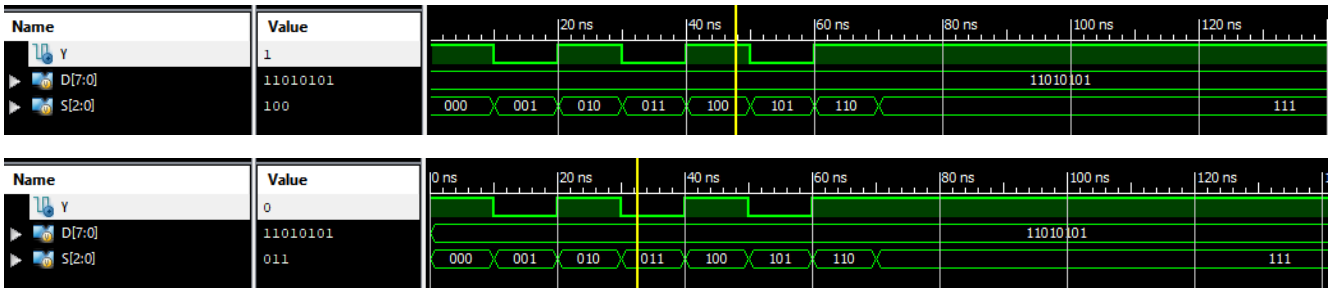
```
    #10;
    $display("%b | %b | %b", S, D, Y);

    // Test Case 8: Select D7
    S = 3'b111;
    #10;
    $display("%b | %b | %b", S, D, Y);

    // End the simulation
    ;
  end

endmodule
```

Output Waveform:



Output Data:

```
S | D | Y
------------------
000 | 11010101 | 1
001 | 11010101 | 0
010 | 11010101 | 1
011 | 11010101 | 0
100 | 11010101 | 1
101 | 11010101 | 0
110 | 11010101 | 1
111 | 11010101 | 1
```



8:3 Encoder

Verilog Program:

```
`timescale 1ns / 1ps
module encoder8to3(
   input wire [7:0] D,    // 8-bit input vector (D0 to D7)
   output reg [2:0] Y     // 3-bit output vector (Y2, Y1, Y0)
);

   always @(*) begin
      case (1'b1)  // When the input is active
         D[7]: Y = 3'b111;  // D7 is active
         D[6]: Y = 3'b110;  // D6 is active
         D[5]: Y = 3'b101;  // D5 is active
         D[4]: Y = 3'b100;  // D4 is active
         D[3]: Y = 3'b011;  // D3 is active
         D[2]: Y = 3'b010;  // D2 is active
         D[1]: Y = 3'b001;  // D1 is active
         D[0]: Y = 3'b000;  // D0 is active
         default: Y = 3'b000; // Default case (optional)
      endcase
   end
```

```
endmodule
```

Test Bench:

```
`timescale 1ns / 1ps
module test_encoder8to3;

  // Inputs
  reg [7:0] D;

  // Output
  wire [2:0] Y;

  // Instantiate the Encoder module
  encoder8to3 uut (
    .D(D),
    .Y(Y)
  );

  // Test cases
  initial begin
    // Display header
    $display("D       | Y");
    $display("---------|----");

    // Test Case 1: D0 is active
    D = 8'b00000001;
    #10;
    $display("%b | %b", D, Y);
```

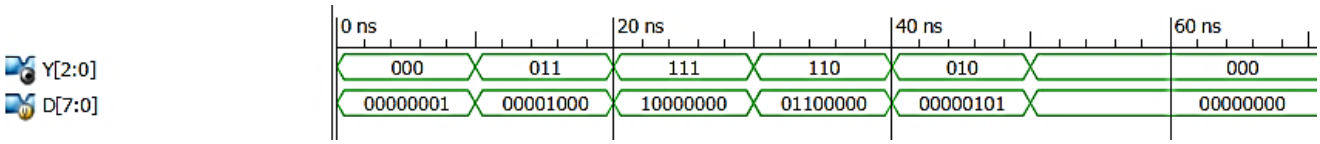
```
// Test Case 2: D3 is active
D = 8'b00001000;
#10;
$display("%b | %b", D, Y);

// Test Case 3: D7 is active
D = 8'b10000000;
#10;
$display("%b | %b", D, Y);

// Test Case 4: Multiple inputs active (D5 and D6)
D = 8'b01100000;
#10;
$display("%b | %b", D, Y);

// Test Case 5: Multiple inputs active (D0 and D2)
D = 8'b00000101;
#10;
$display("%b | %b", D, Y);

// Test Case 6: No inputs active
D = 8'b00000000;
#10;
$display("%b | %b", D, Y);

// End the simulation
//$finish;
```

;

end

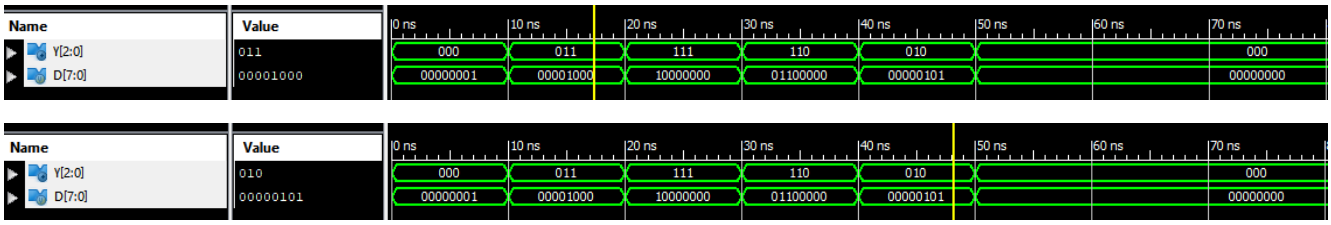
endmodule

Output Waveform:



Output Data:

D	Y
00000001	000
00001000	011
10000000	111
01100000	110
00000101	010
00000000	000



Priority Encoder

Verilog Program

`timescale 1ns / 1ps

module priority_encoder_4to2(

input wire [3:0] D, // 4-bit input vector (D0 to D3)

output reg [1:0] Y, // 2-bit output vector (Y1, Y0)

output reg valid // Valid output to indicate if any input is active

);

```
  always @(*) begin
    valid = 1'b1; // Assume a valid output unless none of the inputs are active
    case (1'b1)  // Check for the highest priority input
      D[3]: Y = 2'b11;  // D3 has the highest priority
      D[2]: Y = 2'b10;  // D2 has the next highest priority
      D[1]: Y = 2'b01;  // D1
      D[0]: Y = 2'b00;  // D0 has the lowest priority
      default: begin
        Y = 2'b00;    // Default output (if no inputs are active)
        valid = 1'b0; // No valid output
      end
    endcase
  end

endmodule
```

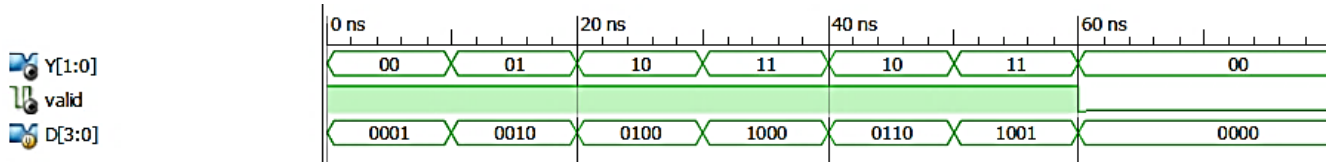
Test Bench

```
`timescale 1ns / 1ps
module test_priority_encoder_4to2;

  // Inputs
  reg [3:0] D;

  // Outputs
  wire [1:0] Y;
  wire valid;
```

```
// Instantiate the Encoder module
priority_encoder_4to2 uut (
    .D(D),
    .Y(Y),
    .valid(valid)
);

// Test cases
initial begin
    // Display header
    $display("D      | Y  | Valid");
    $display("---------------------");

    // Test Case 1: D0 is active
    D = 4'b0001;
    #10;
    $display("%b | %b | %b", D, Y, valid);

    // Test Case 2: D1 is active
    D = 4'b0010;
    #10;
    $display("%b | %b | %b", D, Y, valid);

    // Test Case 3: D2 is active
    D = 4'b0100;
    #10;
    $display("%b | %b | %b", D, Y, valid);
```

```
    // Test Case 4: D3 is active
    D = 4'b1000;
    #10;
    $display("%b | %b | %b", D, Y, valid);

    // Test Case 5: Multiple inputs active (D2 and D1)
    D = 4'b0110;
    #10;
    $display("%b | %b | %b", D, Y, valid);

    // Test Case 6: Multiple inputs active (D3 and D0)
    D = 4'b1001;
    #10;
    $display("%b | %b | %b", D, Y, valid);

    // Test Case 7: No inputs active
    D = 4'b0000;
    #10;
    $display("%b | %b | %b", D, Y, valid);

    // End the simulation
    //$finish;
                ;
  end

endmodule
```

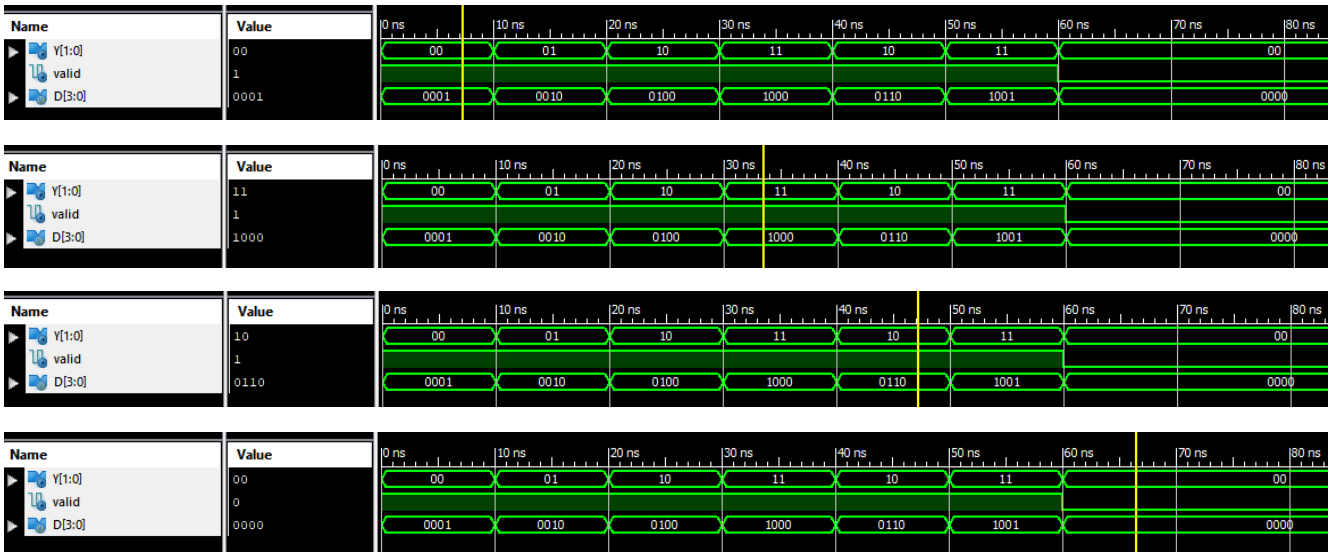
Output Waveform



Output Data

D | Y | Valid

0001 | 00 | 1
0010 | 01 | 1
0100 | 10 | 1
1000 | 11 | 1
0110 | 10 | 1
1001 | 11 | 1
0000 | 00 | 0



Experiment No. 6

Realize 1:8Demux, 3:8 decoder, and 2–bit Comparator using Verilog Behavioral description.

Theory:

1:8 Demultiplexer

Verilog Code:

```
module demux_1to8 (
   input wire D,          // Data input
   input wire [2:0] sel,   // 3-bit select input
   output reg [7:0] Y      // 8-bit output
);

   always @(*) begin
      // Initialize all outputs to 0
      Y = 8'b00000000;

      // Assign the input D to the appropriate output based on the select lines
      case (sel)
         3'b000: Y[0] = D;
         3'b001: Y[1] = D;
         3'b010: Y[2] = D;
         3'b011: Y[3] = D;
         3'b100: Y[4] = D;
         3'b101: Y[5] = D;
         3'b110: Y[6] = D;
         3'b111: Y[7] = D;
```

```
        default: Y = 8'bOOOOOOOO;
    endcase
  end

endmodule
```

Test Bench:

```
module test_demux_1to8;

  // Inputs
  reg D;
  reg [2:0] sel;

  // Outputs
  wire [7:0] Y;

  // Instantiate the Demux module
  demux_1to8 uut (
    .D(D),
    .sel(sel),
    .Y(Y)
  );

  // Test cases
  initial begin
    // Display header
    $display("D | sel | Y");
    $display("-------------------------");
```

```
// Test Case 1: D = 1, sel = 000 (Y0 should be active)
D = 1'b1;
sel = 3'b000;
#10;
$display("%b | %b | %b", D, sel, Y);

// Test Case 2: D = 1, sel = 001 (Y1 should be active)
D = 1'b1;
sel = 3'b001;
#10;
$display("%b | %b | %b", D, sel, Y);

// Test Case 3: D = 1, sel = 010 (Y2 should be active)
D = 1'b1;
sel = 3'b010;
#10;
$display("%b | %b | %b", D, sel, Y);

// Test Case 4: D = 1, sel = 011 (Y3 should be active)
D = 1'b1;
sel = 3'b011;
#10;
$display("%b | %b | %b", D, sel, Y);

// Test Case 5: D = 1, sel = 100 (Y4 should be active)
D = 1'b1;
sel = 3'b100;
```

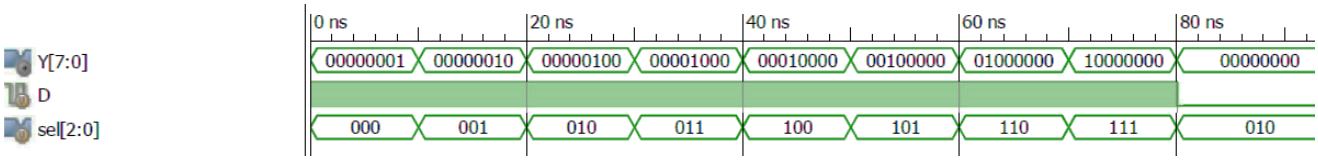
```
#10;
$display("%b | %b | %b", D, sel, Y);

// Test Case 6: D = 1, sel = 101 (Y5 should be active)
D = 1'b1;
sel = 3'b101;
#10;
$display("%b | %b | %b", D, sel, Y);

// Test Case 7: D = 1, sel = 110 (Y6 should be active)
D = 1'b1;
sel = 3'b110;
#10;
$display("%b | %b | %b", D, sel, Y);

// Test Case 8: D = 1, sel = 111 (Y7 should be active)
D = 1'b1;
sel = 3'b111;
#10;
$display("%b | %b | %b", D, sel, Y);

// Test Case 9: D = 0 (all outputs should be 0 regardless of sel)
D = 1'b0;
sel = 3'b010;
#10;
$display("%b | %b | %b", D, sel, Y);

// End the simulation
```

;

end

endmodule

Output Waveform:



Output Data:

D | sel | Y

---

1 | 000 | 00000001
1 | 001 | 00000010
1 | 010 | 00000100
1 | 011 | 00001000
1 | 100 | 00010000
1 | 101 | 00100000
1 | 110 | 01000000
1 | 111 | 10000000
0 | 010 | 00000000

3:8 Decoder

Verilog Code:

module decoder_3to8 (

input wire [2:0] in, // 3-bit input

```
    output reg [7:0] out   // 8-bit output
);

    // Always block that executes whenever the input changes
    always @(*) begin
        // Initialize all outputs to 0
        out = 8'b00000000;

        // Use a case statement to determine the active output
        case (in)
            3'b000: out[0] = 1'b1;  // When in = 000, out[0] = 1
            3'b001: out[1] = 1'b1;  // When in = 001, out[1] = 1
            3'b010: out[2] = 1'b1;  // When in = 010, out[2] = 1
            3'b011: out[3] = 1'b1;  // When in = 011, out[3] = 1
            3'b100: out[4] = 1'b1;  // When in = 100, out[4] = 1
            3'b101: out[5] = 1'b1;  // When in = 101, out[5] = 1
            3'b110: out[6] = 1'b1;  // When in = 110, out[6] = 1
            3'b111: out[7] = 1'b1;  // When in = 111, out[7] = 1
            default: out = 8'b00000000; // Default case, all outputs = 0
        endcase
    end

endmodule
```

Test Bench:

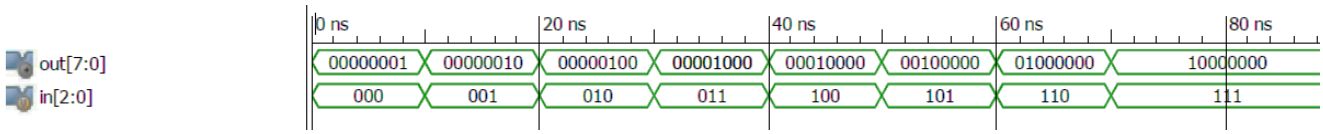
```
module test_decoder_3to8;

    // Inputs
    reg [2:0] in;
```

```
// Outputs
wire [7:0] out;

// Instantiate the 3:8 Decoder module
decoder_3to8 uut (
    .in(in),
    .out(out)
);

// Test cases
initial begin
    // Display header
    $display("in   | out");
    $display("-----------");

    // Test Case 1: in = 000 (out[0] should be active)
    in = 3'b000;
    #10;
    $display("%b | %b", in, out);

    // Test Case 2: in = 001 (out[1] should be active)
    in = 3'b001;
    #10;
    $display("%b | %b", in, out);

    // Test Case 3: in = 010 (out[2] should be active)
    in = 3'b010;
```

```
#10;
$display("%b | %b", in, out);

// Test Case 4: in = 011 (out[3] should be active)
in = 3'b011;
#10;
$display("%b | %b", in, out);

// Test Case 5: in = 100 (out[4] should be active)
in = 3'b100;
#10;
$display("%b | %b", in, out);

// Test Case 6: in = 101 (out[5] should be active)
in = 3'b101;
#10;
$display("%b | %b", in, out);

// Test Case 7: in = 110 (out[6] should be active)
in = 3'b110;
#10;
$display("%b | %b", in, out);

// Test Case 8: in = 111 (out[7] should be active)
in = 3'b111;
#10;
$display("%b | %b", in, out);
```

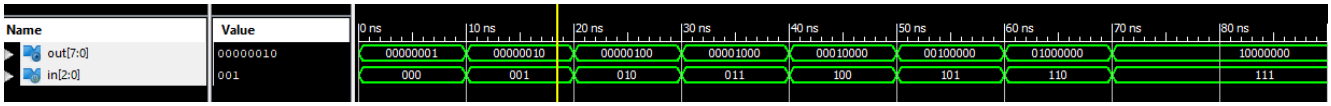
```
    // End the simulation
    ;
  end

endmodule
```

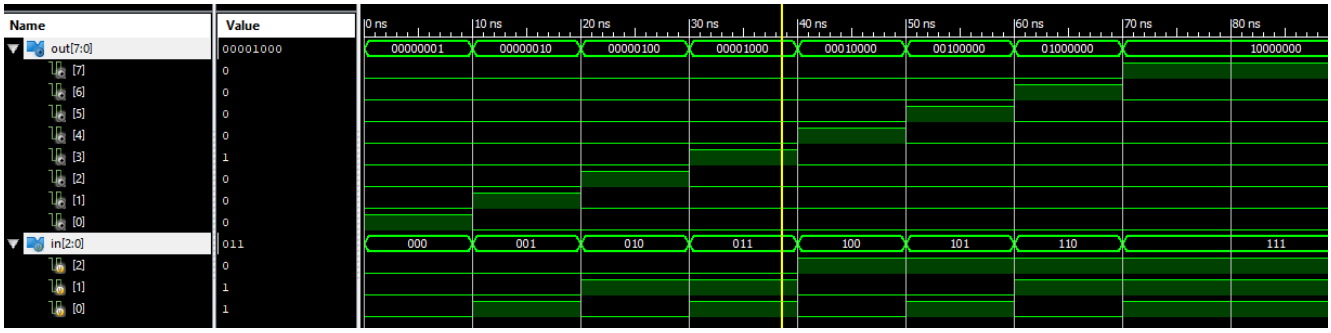
Output Waveform:



Output Data:

in | out
---
000 | 00000001
001 | 00000010
010 | 00000100
011 | 00001000
100 | 00010000
101 | 00100000
110 | 01000000
111 | 10000000





2-Bit Comparator

Verilog Code:

```
module comparator_2bit (
   input wire [1:0] A,  // 2-bit input A
   input wire [1:0] B,  // 2-bit input B
   output reg A_greater_B,  // A > B
   output reg A_less_B,     // A < B
   output reg A_equal_B     // A = B
);

   always @(*) begin
      // Initialize outputs
      A_greater_B = 0;
      A_less_B = 0;
      A_equal_B = 0;

      // Compare A and B
      if (A > B) begin
         A_greater_B = 1;
      end else if (A < B) begin
         A_less_B = 1;
      end else begin
         A_equal_B = 1;
      end
   end
```

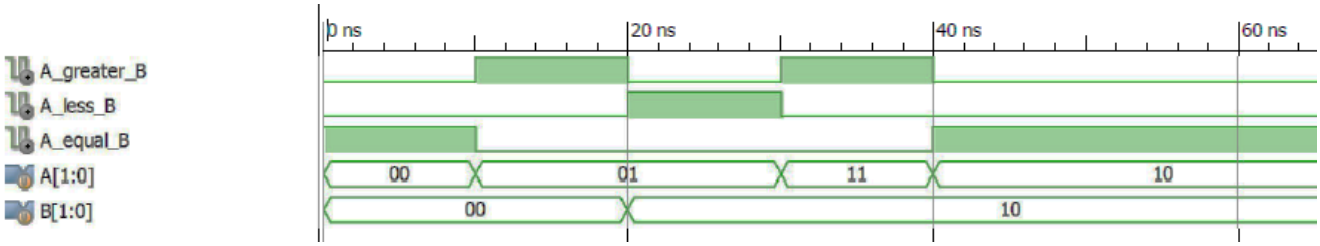
```
endmodule
```

Test Bench:

```
module test_comparator_2bit;

  // Inputs
  reg [1:0] A;
  reg [1:0] B;

  // Outputs
  wire A_greater_B;
  wire A_less_B;
  wire A_equal_B;

  // Instantiate the comparator module
  comparator_2bit uut (
    .A(A),
    .B(B),
    .A_greater_B(A_greater_B),
    .A_less_B(A_less_B),
    .A_equal_B(A_equal_B)
  );

  // Test cases
  initial begin
    // Monitor output changes
    $monitor("A = %b, B = %b -> A_greater_B = %b, A_less_B = %b, A_equal_B = %b", A, B, A_greater_B, A_less_B, A_equal_B);
```

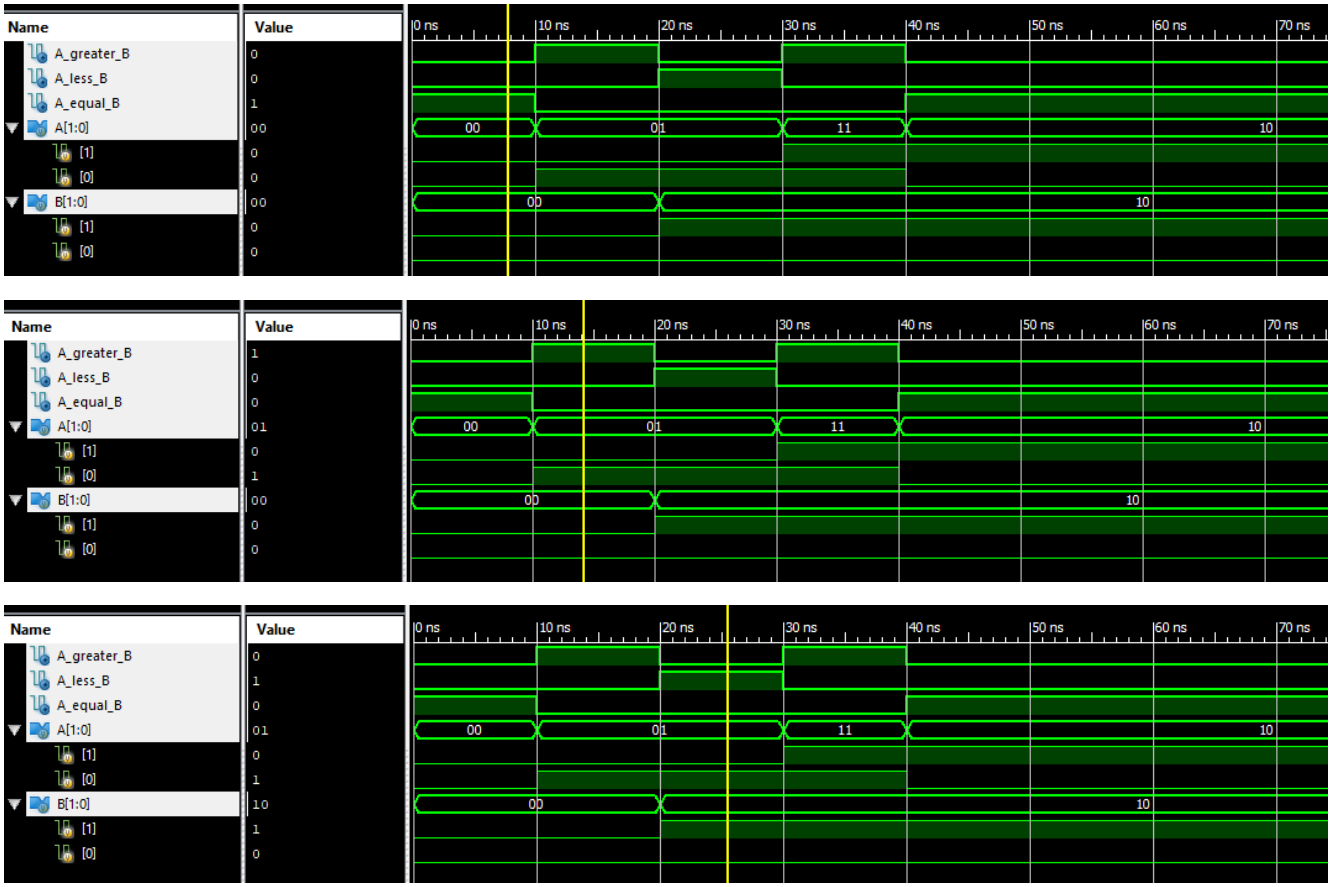
```
    // Test Case 1: A = 00, B = 00 (A = B)
    A = 2'b00; B = 2'b00;
    #10;

    // Test Case 2: A = 01, B = 00 (A > B)
    A = 2'b01; B = 2'b00;
    #10;

    // Test Case 3: A = 01, B = 10 (A < B)
    A = 2'b01; B = 2'b10;
    #10;

    // Test Case 4: A = 11, B = 10 (A > B)
    A = 2'b11; B = 2'b10;
    #10;

    // Test Case 5: A = 10, B = 10 (A = B)
    A = 2'b10; B = 2'b10;
    #10;

    // End simulation
    ;
  end

endmodule
```

Output Waveform:



Output Data:

A = 00, B = 00 -> A_greater_B = 0, A_less_B = 0, A_equal_B = 1
A = 01, B = 00 -> A_greater_B = 1, A_less_B = 0, A_equal_B = 0
A = 01, B = 10 -> A_greater_B = 0, A_less_B = 1, A_equal_B = 0
A = 11, B = 10 -> A_greater_B = 1, A_less_B = 0, A_equal_B = 0
A = 10, B = 10 -> A_greater_B = 0, A_less_B = 0, A_equal_B = 1



Experiment No. 7

Realize using Verilog Behavioral description - a) JK type, b) SR type, c) T type, and d) D type.

Theory:

Flip-Flops

SR Flip-Flop

Verilog Code:

```
module sr_flip_flop (
   input wire S,     // Set input
   input wire R,     // Reset input
   input wire clk,   // Clock input
   output reg Q,     // Output Q
   output reg Qn     // Output Q'
);

   always @(posedge clk) begin
      if (S == 1'b0 && R == 1'b0) begin
         // No change in the state
         Q <= Q;
         Qn <= Qn;
      end
      else if (S == 1'b0 && R == 1'b1) begin
         // Reset: Q = 0, Q' = 1
         Q <= 1'b0;
         Qn <= 1'b1;
      end
```

```
    else if (S == 1'b1 && R == 1'b0) begin
        // Set: Q = 1, Q' = 0
        Q <= 1'b1;
        Qn <= 1'b0;
    end
    else if (S == 1'b1 && R == 1'b1) begin
        // Undefined state, optional implementation
        Q <= 1'bx;   // 'x' indicates an unknown or invalid state
        Qn <= 1'bx;
    end
  end
endmodule
```

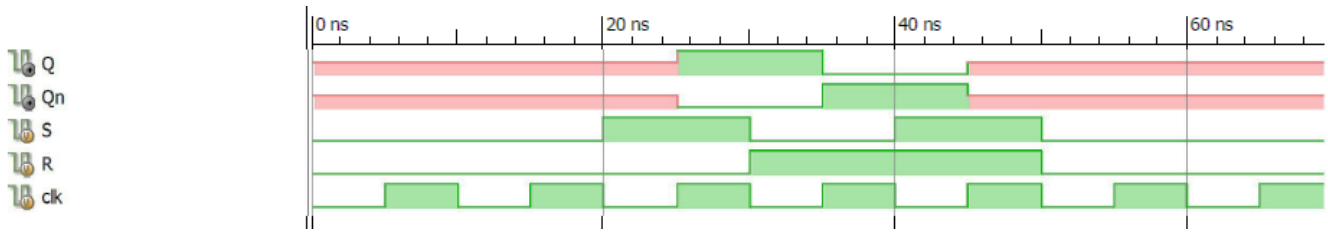
Test Bench:

```
module test_sr_flip_flop;

  // Inputs
  reg S;
  reg R;
  reg clk;

  // Outputs
  wire Q;
  wire Qn;

  // Instantiate the SR flip-flop module
  sr_flip_flop uut (
    .S(S),
    .R(R),
```

```
    .clk(clk),
    .Q(Q),
    .Qn(Qn)
);

// Generate clock signal
initial begin
    clk = 0;
    forever #5 clk = ~clk;  // 10 time units clock period
end

// Test cases
initial begin
    // Monitor output changes
    $monitor("At time %0t: S = %b, R = %b, Q = %b, Qn = %b", $time, S, R, Q, Qn);

    // Initialize inputs
    S = 0; R = 0;
    #10;  // Wait for 10 time units

    // Test Case 1: Set S = 0, R = 0 (No change)
    S = 0; R = 0;
    #10;

    // Test Case 2: Set S = 1, R = 0 (Set state)
    S = 1; R = 0;
    #10;
```

```
    // Test Case 3: Set S = 0, R = 1 (Reset state)
    S = 0; R = 1;
    #10;

    // Test Case 4: Set S = 1, R = 1 (Undefined state)
    S = 1; R = 1;
    #10;

    // Test Case 5: Return to S = 0, R = 0 (No change)
    S = 0; R = 0;
    #10;

    // End simulation
    ;
  end

endmodule
```

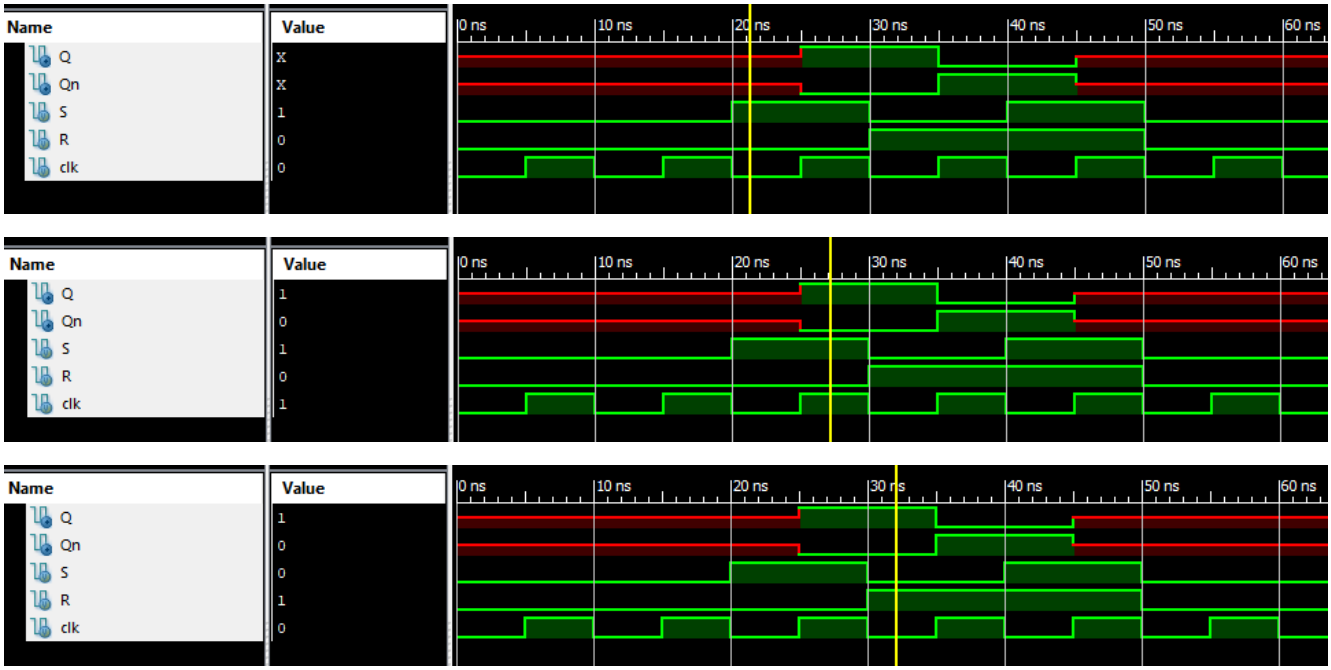
Output Waveform:



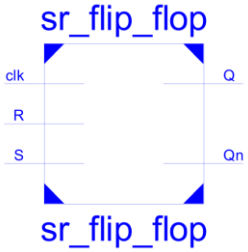
Output Data:

At time 0: S = 0, R = 0, Q = x, Qn = x
At time 20000: S = 1, R = 0, Q = x, Qn = x
At time 25000: S = 1, R = 0, Q = 1, Qn = 0
At time 30000: S = 0, R = 1, Q = 1, Qn = 0
At time 35000: S = 0, R = 1, Q = 0, Qn = 1
At time 40000: S = 1, R = 1, Q = 0, Qn = 1

At time 45000: S = 1, R = 1, Q = x, Qn = x
At time 50000: S = 0, R = 0, Q = x, Qn = x



SR Flip flop RTL Schematics:



D Flip-Flop:

Verilog Code:

```
module d_flip_flop (
   input wire D,    // Data input
   input wire clk,  // Clock input
   output reg Q,    // Output Q
   output wire Qn   // Output Q'
```

```
);

   // Invert the output Q to get Q'
   assign Qn = ~Q;

   // Always block to capture the data on the rising edge of the clock
   always @(posedge clk) begin
      Q <= D;  // On the rising edge of the clock, Q takes the value of D
   end

endmodule
```

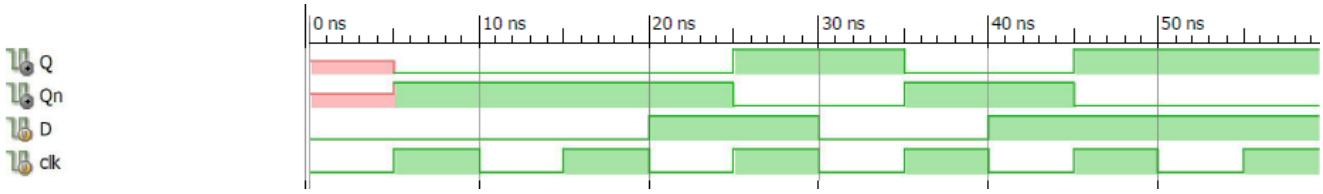
Test Bench:

```
module test_d_flip_flop;

   // Inputs
   reg D;
   reg clk;

   // Outputs
   wire Q;
   wire Qn;

   // Instantiate the D flip-flop module
   d_flip_flop uut (
      .D(D),
      .clk(clk),
      .Q(Q),
```

```
    .Qn(Qn)
);

// Generate clock signal
initial begin
    clk = 0;
    forever #5 clk = ~clk;  // Clock period of 10 time units
end

// Test cases
initial begin
    // Monitor output changes
    $monitor("At time %0t: D = %b, Q = %b, Qn = %b", $time, D, Q, Qn);

    // Initialize inputs
    D = 0;
    #10;  // Wait for 10 time units

    // Test Case 1: D = 0
    D = 0;
    #10;  // Wait for the clock edge

    // Test Case 2: D = 1
    D = 1;
    #10;  // Wait for the clock edge

    // Test Case 3: D = 0
    D = 0;
```

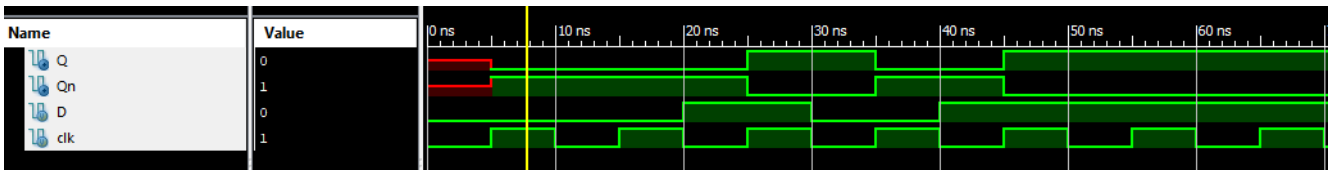
```
    #10;  // Wait for the clock edge

    // Test Case 4: D = 1
    D = 1;
    #10;  // Wait for the clock edge

    // End simulation
    ;
  end

endmodule
```

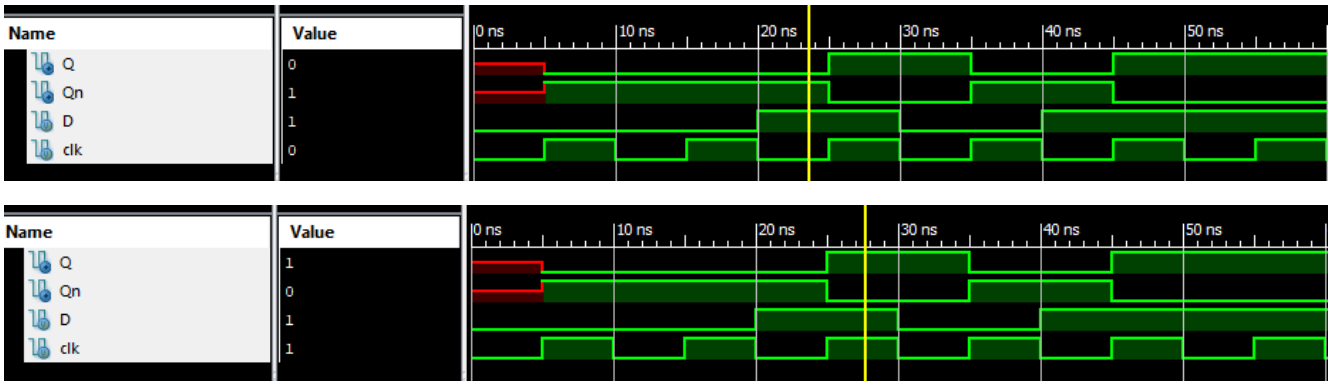
Output Waveform:



Output Data:

At time 0: D = 0, Q = x, Qn = x
At time 5000: D = 0, Q = 0, Qn = 1
At time 20000: D = 1, Q = 0, Qn = 1
At time 25000: D = 1, Q = 1, Qn = 0
At time 30000: D = 0, Q = 1, Qn = 0
At time 35000: D = 0, Q = 0, Qn = 1
At time 40000: D = 1, Q = 0, Qn = 1
At time 45000: D = 1, Q = 1, Qn = 0





JK Flip-Flop:

Verilog Code:

```
module jk_flip_flop (
   input wire J,     // J input
   input wire K,     // K input
   input wire clk,   // Clock input
   output reg Q,     // Output Q
   output wire Qn    // Output Q'
);

   // Invert the output Q to get Q'
   assign Qn = ~Q;

   // Always block to capture the data on the rising edge of the clock
   always @(posedge clk) begin
      if (J == 1'bo && K == 1'bo) begin
         // No Change
         Q <= Q;
      end
```

```
    else if (J == 1'b0 && K == 1'b1) begin
        // Reset
        Q <= 1'b0;
    end
    else if (J == 1'b1 && K == 1'b0) begin
        // Set
        Q <= 1'b1;
    end
    else if (J == 1'b1 && K == 1'b1) begin
        // Toggle
        Q <= ~Q;
    end
  end

endmodule
```

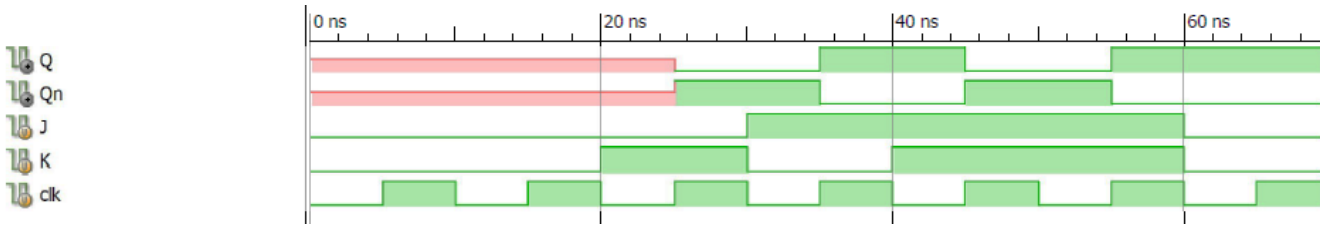
Test Bench:

```
module test_jk_flip_flop;

  // Inputs
  reg J;
  reg K;
  reg clk;

  // Outputs
  wire Q;
  wire Qn;

  // Instantiate the JK flip-flop module
```

```
jk_flip_flop uut (
    .J(J),
    .K(K),
    .clk(clk),
    .Q(Q),
    .Qn(Qn)
);

// Generate clock signal
initial begin
    clk = 0;
    forever #5 clk = ~clk;  // Clock period of 10 time units
end

// Test cases
initial begin
    // Monitor output changes
    $monitor("At time %0t: J = %b, K = %b, Q = %b, Qn = %b", $time, J, K, Q, Qn);

    // Initialize inputs
    J = 0; K = 0;
    #10;  // Wait for 10 time units

    // Test Case 1: J = 0, K = 0 (No Change)
    J = 0; K = 0;
    #10;

    // Test Case 2: J = 0, K = 1 (Reset)
```

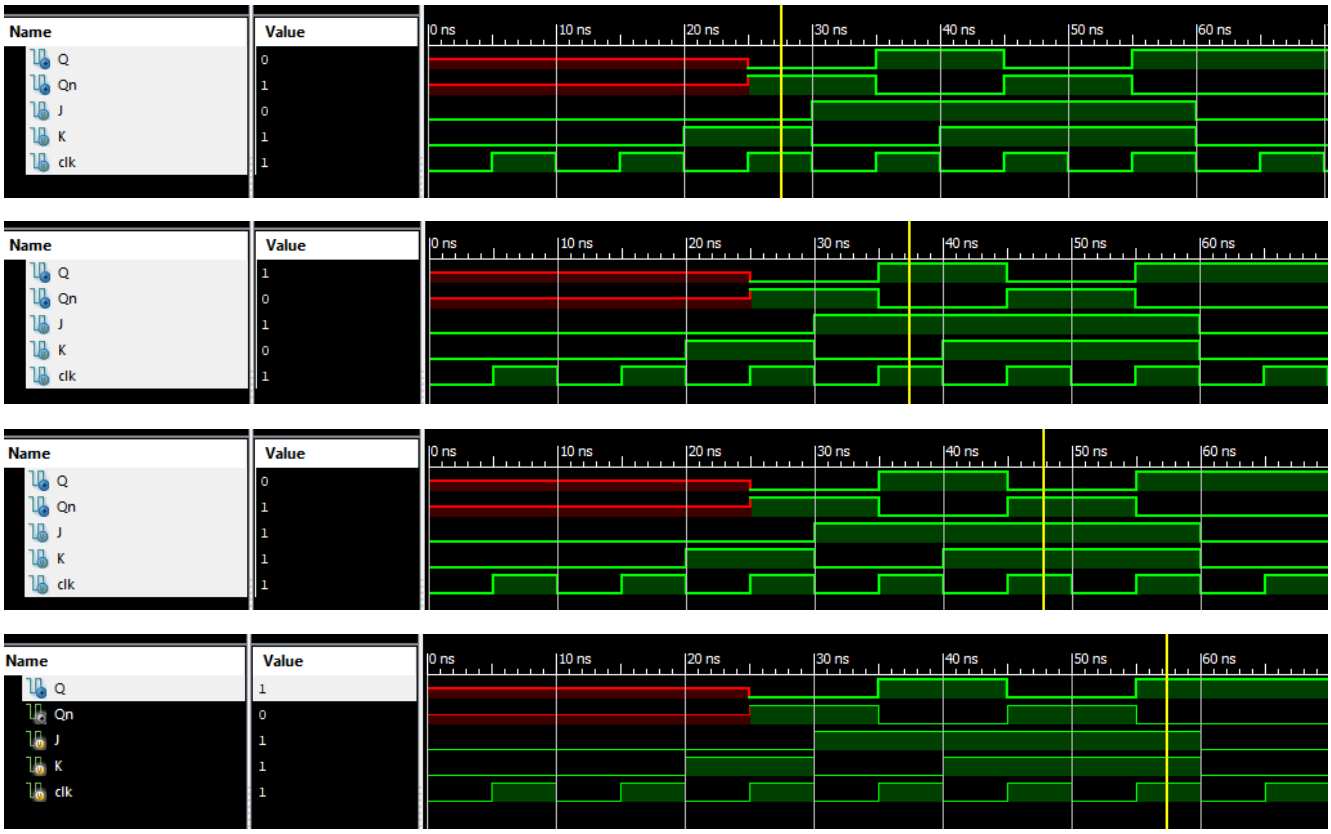
```
    J = 0; K = 1;
    #10;

    // Test Case 3: J = 1, K = 0 (Set)
    J = 1; K = 0;
    #10;

    // Test Case 4: J = 1, K = 1 (Toggle)
    J = 1; K = 1;
    #10;

    // Test Case 5: J = 1, K = 1 (Toggle again to observe toggling)
    J = 1; K = 1;
    #10;

    // Test Case 6: J = 0, K = 0 (No Change)
    J = 0; K = 0;
    #10;

    // End simulation
    ;
  end

endmodule
```

Output Waveform:



Output Data:

At time 0: J = 0, K = 0, Q = x, Qn = x
At time 20000: J = 0, K = 1, Q = x, Qn = x
At time 25000: J = 0, K = 1, Q = 0, Qn = 1
At time 30000: J = 1, K = 0, Q = 0, Qn = 1
At time 35000: J = 1, K = 0, Q = 1, Qn = 0
At time 40000: J = 1, K = 1, Q = 1, Qn = 0
At time 45000: J = 1, K = 1, Q = 0, Qn = 1
At time 55000: J = 1, K = 1, Q = 1, Qn = 0
At time 60000: J = 0, K = 0, Q = 1, Qn = 0



T Flip-Flop:

Verilog Code:

```
module t_flip_flop (
   input wire T,     // Toggle input
   input wire clk,   // Clock input
   output reg Q,     // Output Q
   output wire Qn    // Output Q'
);

   initial Q <= 1'b0;

   // Always block to capture the data on the rising edge of the clock
   always @(posedge clk) begin
      if (T==1'b1) begin
         Q <= ~Q;  // Toggle the state if T is high
      end
      // If T is low, the state of Q remains unchanged
   end

         // Invert the output Q to get Q'
   assign Qn = ~Q;
endmodule
```

Test Bench:

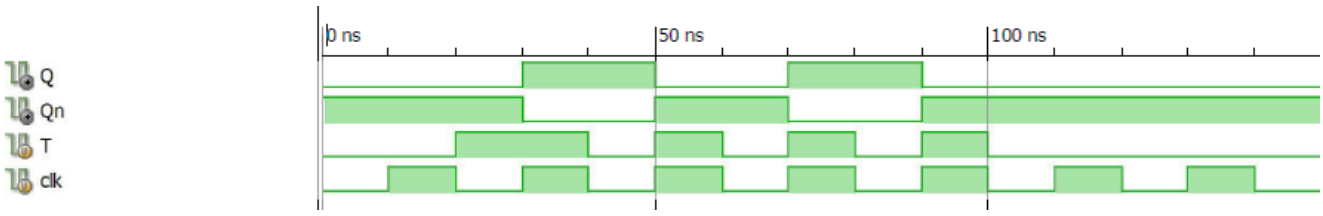
```
module test_t_flip_flop;

   // Inputs
   reg T;
   reg clk;
```

```
// Outputs
wire Q;
wire Qn;

// Instantiate the T flip-flop module
t_flip_flop uut (
    .T(T),
    .clk(clk),
    .Q(Q),
    .Qn(Qn)
);

// Generate clock signal
initial begin
    clk = 0;
    forever #10 clk = ~clk;  // Clock period of 10 time units
end

// Test cases
initial begin
    // Monitor output changes
    $monitor("At time %0t: T = %b, Q = %b, Qn = %b", $time, T, Q, Qn);

     // Initialize inputs
    T = 0;
    #10;  // Wait for 10 time units

    // Test Case 1: T = 0 (No Change)
```

```
T = 0;
#10;  // Wait for the clock edge

// Test Case 2: T = 1 (Toggle)
T = 1;
#10;  // Wait for the clock edge

// Test Case 3: T = 1 (Toggle again to observe toggling)
T = 1;
#10;  // Wait for the clock edge

// Test Case 4: T = 0 (No Change)
T = 0;
#10;  // Wait for the clock edge

// Test Case 5: T = 1 (Toggle)
T = 1;
#10;  // Wait for the clock edge

        // Test Case 6: T = 0 (No Change)
T = 0;
#10;  // Wait for the clock edge

        // Test Case 7: T = 1 (Toggle again to observe toggling)
T = 1;
#10;  // Wait for the clock edge

// Test Case 8: T = 0 (No Change)
```

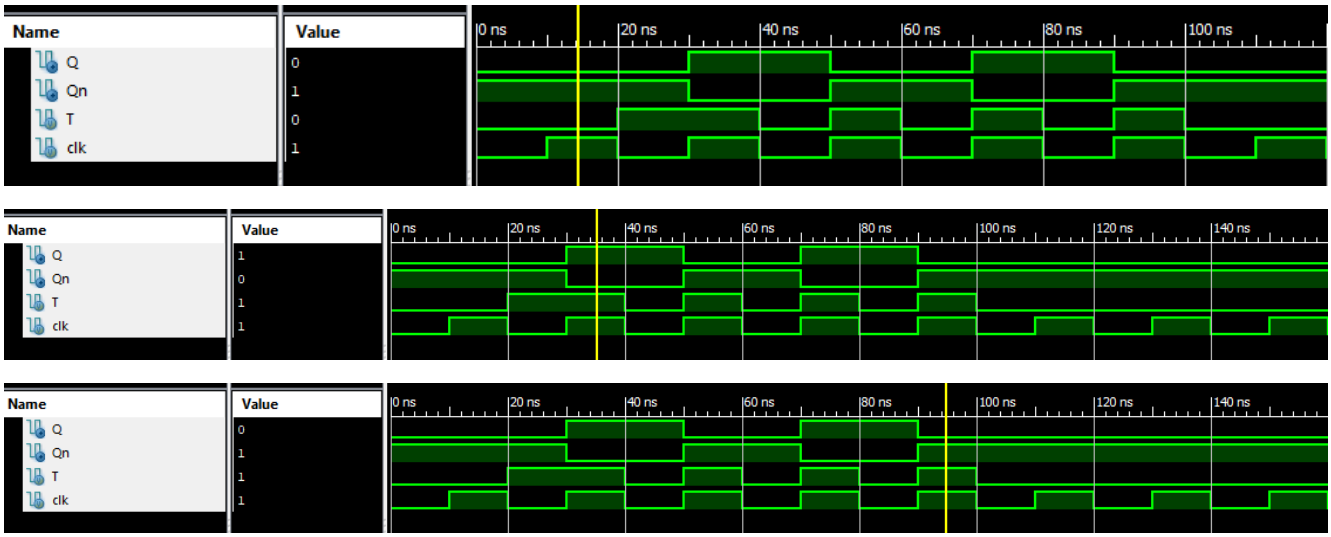
```
    T = 0;
    #10;  // Wait for the clock edge

    // Test Case 9: T = 1 (Toggle)
    T = 1;
    #10;  // Wait for the clock edge

            // Test Case 10: T = 0 (No Change)
    T = 0;
    #10;  // Wait for the clock edge
    // End simulation
    ;
  end

endmodule
```

Output Waveform:



Output Data:

At time 0: T = 0, Q = 0, Qn = 1
At time 20000: T = 1, Q = 0, Qn = 1
At time 30000: T = 1, Q = 1, Qn = 0
At time 40000: T = 0, Q = 1, Qn = 0
At time 50000: T = 1, Q = 0, Qn = 1
At time 60000: T = 0, Q = 0, Qn = 1
At time 70000: T = 1, Q = 1, Qn = 0
At time 80000: T = 0, Q = 1, Qn = 0
At time 90000: T = 1, Q = 0, Qn = 1
At time 100000: T = 0, Q = 0, Qn = 1


Name
Value
Q
Qn
T
clk
0
1
0
1
0 ns
20 ns
40 ns
60 ns
80 ns
100 ns
Name
Value
Q
Qn
T
clk
1
0
1
1
0 ns
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns
Name
Value
Q
Qn
T
clk
0
1
1
1
0 ns
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns

Experiment No. 8

Realize Counters-up/down (BCD and binary) using Verilog Behavioral description.

Theory:

A counter is a sequential circuit that goes through a prescribed sequence of states upon the application of input pulses. The input pulses called count pulses, may be clock pulses, or they may originate from an external source and may occur at prescribed intervals of time or at random.

4-Bit Synchronous Decade Counter:

BCD Counter

3-Bit Synchronous Binary Counter

4-Bit Synchronous Binary Counter

4-bit Synchronous BCD or Decade UP/DOWN Counter:

Verilog Code:

```
module bcd_up_down_counter (
   input wire clk,          // Clock signal
   input wire reset,        // Reset signal
   input wire up_down,      // Control signal: 1 for up, 0 for down
   output reg [3:0] count   // 4-bit BCD output
);

   always @(posedge clk or posedge reset) begin
      if (reset) begin
         count <= 4'b0000;  // Reset counter to 0000
      end else begin
         if (up_down) begin
```

```
            // Up Counter
            if (count == 4'b1001) begin
                count <= 4'b0000;  // Reset to 0000 after 9
            end else begin
                count <= count + 1;
            end
        end else begin
            // Down Counter
            if (count == 4'b0000) begin
                count <= 4'b1001;  // Reset to 1001 after 0
            end else begin
                count <= count - 1;
            end
        end
    end
  end
endmodule
```

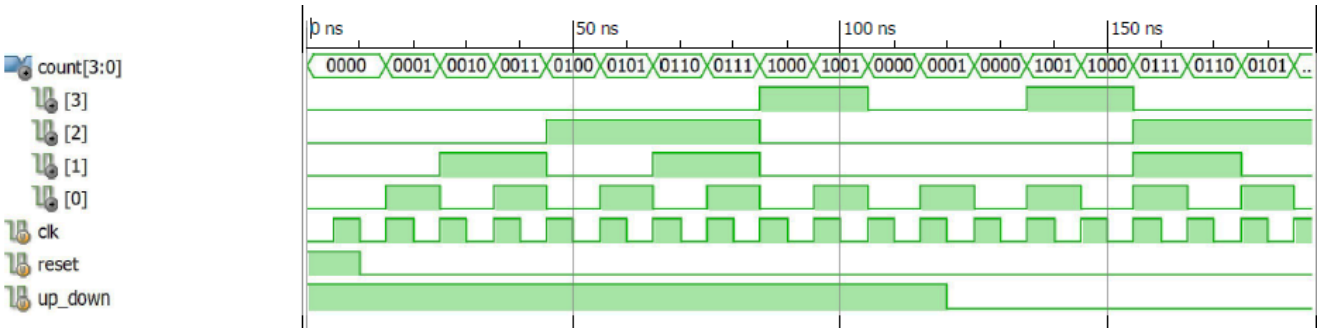
Test Bench:

```
module tb_bcd_up_down_counter;
  // Testbench signals
  reg clk;
  reg reset;
  reg up_down;
  wire [3:0] count;

  // Instantiate the counter
  bcd_up_down_counter uut (
    .clk(clk),
```

```
    .reset(reset),
    .up_down(up_down),
    .count(count)
);

// Clock generation
initial begin
    clk = 0;
    forever #5 clk = ~clk; // Clock period = 10 units
end

// Test procedure
initial begin
    // Monitor signals
    $monitor("Time=%0t, reset=%b, up_down=%b, count=%b", $time, reset, up_down,
count);

    // Test case 1: Reset the counter
    reset = 1; up_down = 1; #10;
    reset = 0; #10;

    // Test case 2: Count up from 0 to 9
    up_down = 1; #100;

    // Test case 3: Count down from 9 to 0
    up_down = 0; #100;

    // Test case 4: Apply reset during counting
```

```
    reset = 1; #10;
    reset = 0; #20;

    // Finish the simulation
    ;
  end
endmodule
```

Output Waveform:



Output Data:

Time=0, reset=1, up_down=1, count=0000
Time=10000, reset=0, up_down=1, count=0000
Time=15000, reset=0, up_down=1, count=0001
Time=25000, reset=0, up_down=1, count=0010
Time=35000, reset=0, up_down=1, count=0011
Time=45000, reset=0, up_down=1, count=0100
Time=55000, reset=0, up_down=1, count=0101
Time=65000, reset=0, up_down=1, count=0110
Time=75000, reset=0, up_down=1, count=0111
Time=85000, reset=0, up_down=1, count=1000
Time=95000, reset=0, up_down=1, count=1001
Time=105000, reset=0, up_down=1, count=0000
Time=115000, reset=0, up_down=1, count=0001
Time=120000, reset=0, up_down=0, count=0001
Time=125000, reset=0, up_down=0, count=0000
Time=135000, reset=0, up_down=0, count=1001
Time=145000, reset=0, up_down=0, count=1000
Time=155000, reset=0, up_down=0, count=0111
Time=165000, reset=0, up_down=0, count=0110
Time=175000, reset=0, up_down=0, count=0101
Time=185000, reset=0, up_down=0, count=0100

```
Time=195000, reset=0, up_down=0, count=0011
Time=205000, reset=0, up_down=0, count=0010
Time=215000, reset=0, up_down=0, count=0001
Time=220000, reset=1, up_down=0, count=0000
Time=230000, reset=0, up_down=0, count=0000
```

4-bit Synchronous Binary UP/DOWN Counter:

Verilog Code:

```
module binary_up_down_counter (
   input wire clk,          // Clock signal
   input wire reset,        // Reset signal
   input wire up_down,      // Control signal: 1 for up, 0 for down
   output reg [3:0] count   // 4-bit binary output
);

   always @(posedge clk or posedge reset) begin
      if (reset) begin
         count <= 4'b0000;  // Reset counter to 0000
      end else begin
         if (up_down) begin
            count <= count + 1;  // Up counting
         end else begin
            count <= count - 1;  // Down counting
         end
      end
   end
endmodule
```

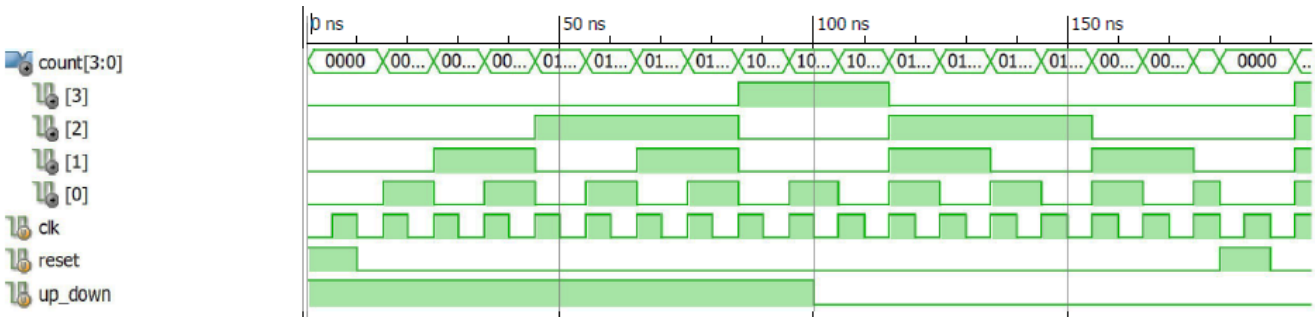
Test Bench:

```
module tb_binary_up_down_counter;
   // Testbench signals
```

```
reg clk;
reg reset;
reg up_down;
wire [3:0] count;

// Instantiate the counter
binary_up_down_counter uut (
    .clk(clk),
    .reset(reset),
    .up_down(up_down),
    .count(count)
);

// Clock generation
initial begin
    clk = 0;
    forever #5 clk = ~clk; // Clock period = 10 units
end

// Test procedure
initial begin
    // Monitor signals
    $monitor("Time=%0t, reset=%b, up_down=%b, count=%b", $time, reset, up_down,
count);

    // Test case 1: Reset the counter
    reset = 1; up_down = 1; #10;
    reset = 0; #10;
```

```
    // Test case 2: Count up from 0 to 15
    up_down = 1; #80;

    // Test case 3: Count down from 15 to 0
    up_down = 0; #80;

    // Test case 4: Apply reset during counting
    reset = 1; #10;
    reset = 0; #20;

    // Finish the simulation
    ;
  end
endmodule
```

Output Waveform:



Output Data:

Time=0, reset=1, up_down=1, count=0000
Time=10000, reset=0, up_down=1, count=0000
Time=15000, reset=0, up_down=1, count=0001
Time=25000, reset=0, up_down=1, count=0010
Time=35000, reset=0, up_down=1, count=0011
Time=45000, reset=0, up_down=1, count=0100
Time=55000, reset=0, up_down=1, count=0101
Time=65000, reset=0, up_down=1, count=0110
Time=75000, reset=0, up_down=1, count=0111

Time=85000, reset=0, up_down=1, count=1000
Time=95000, reset=0, up_down=1, count=1001
Time=100000, reset=0, up_down=0, count=1001
Time=105000, reset=0, up_down=0, count=1000
Time=115000, reset=0, up_down=0, count=0111
Time=125000, reset=0, up_down=0, count=0110
Time=135000, reset=0, up_down=0, count=0101
Time=145000, reset=0, up_down=0, count=0100
Time=155000, reset=0, up_down=0, count=0011
Time=165000, reset=0, up_down=0, count=0010
Time=175000, reset=0, up_down=0, count=0001
Time=180000, reset=1, up_down=0, count=0000
Time=190000, reset=0, up_down=0, count=0000
Time=195000, reset=0, up_down=0, count=1111
Time=205000, reset=0, up_down=0, count=1110
Time=215000, reset=0, up_down=0, count=1101
Time=225000, reset=0, up_down=0, count=1100
Time=235000, reset=0, up_down=0, count=1011
Time=245000, reset=0, up_down=0, count=1010
Time=255000, reset=0, up_down=0, count=1001
Time=265000, reset=0, up_down=0, count=1000
Time=275000, reset=0, up_down=0, count=0111
Time=285000, reset=0, up_down=0, count=0110
Time=295000, reset=0, up_down=0, count=0101
Time=305000, reset=0, up_down=0, count=0100
Time=315000, reset=0, up_down=0, count=0011

How to download and install Xilinx ISE Design Suite 14.7? || Install Xilinx on Windows ||
https://www.youtube.com/watch?v=_UYb7LhhhsY