

## Contents

Introduction to Verilog .....	2
FPGA.....	3
Design Flow of FPGA.....	4
Procedure to Design a Digital system using Xilinx.....	4
Experiment 1: DESIGN OF BASIC GATES.....	6
Experiment 2: Design of Half Adder.....	11
Experiment 3: Design of 4:1MUX .....	13
Experiment 4: 4 Bit adder using 1-bit adder .....	18
Experiment 5: 3:8 Decoder .....	15
Experiment 6: 8:3 Encoder.....	20
Experiment 7: SR Flipflops.....	22
Experiment 8: D-FLIPFLOP .....	24
Experiment 9: JK-FLIPFLOP.....	26
Experiment 10: Synchronous Up/Down Counter.....	28
Experiment 11: 32-Bit ALU.....	30

## INTRODUCTION:

**Verilog** is a hardware description language (HDL) used to model electronic systems. It is primarily used in the design and simulation of digital circuits, such as FPGAs, ASICs, and other hardware systems. Verilog allows designers to describe the structure and behavior of digital circuits at various levels of abstraction, from high-level algorithmic descriptions to low-level gate-level implementations.

### Types of Verilog Code:

#### 1. Behavioral Verilog:

- Describes what the circuit should do without specifying how it is implemented.
- Focuses on the functionality of the system (e.g., if, case, loops).
- Example:

```
always@(posedgeclk)begin if
    (reset)
        count<=0;
    else
        count<=count+1; end
```

#### 2. Dataflow Verilog:

- Describes the flow of data between registers and combinational logic.
- Uses continuous assignments(assign) to model the logic.
- Example:

```
assign sum = a + b;
assignproduct=a*b;
```

#### 3. Structural Verilog:

- Describes the physical structure of the circuit, such as gates and modules.
- It connects primitive gates or previously defined modules.

- Example:

andand1(out,a,b); or

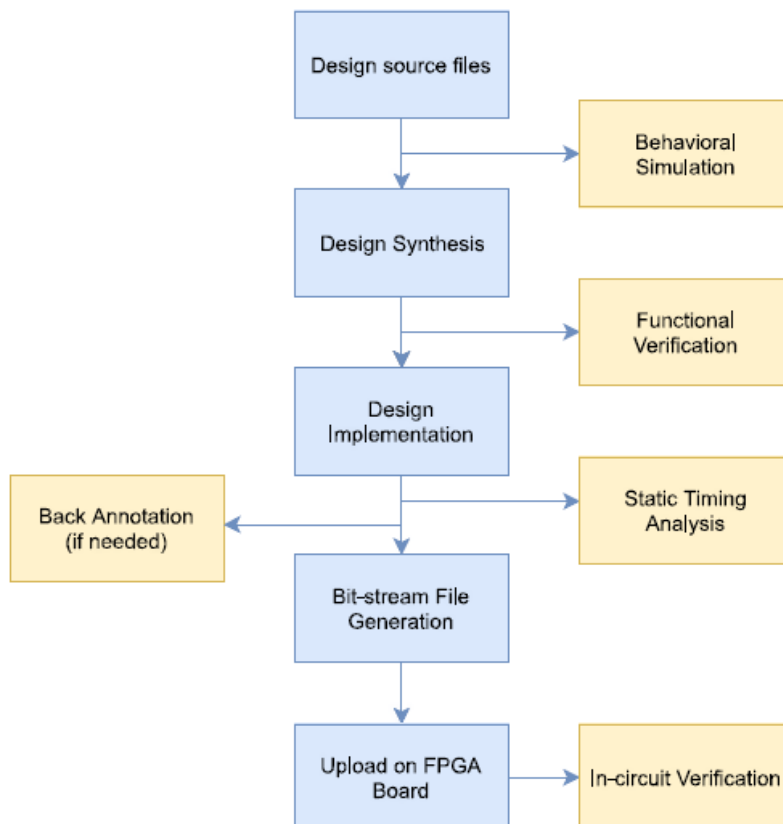
or1 (out, a, b);

## FPGA:

A **Field-Programmable Gate Array (FPGA)** is a type of programmable hardware device that allows users to configure its logic and interconnections after manufacturing. Unlike fixed-function chips such as ASICs (Application-Specific Integrated Circuits), FPGAs are highly flexible and can be reprogrammed multiple times to implement various digital circuits and systems.

### FPGA Design Flow

Designing a system on an FPGA involves a series of steps that transform a high-level design specification into a hardware implementation. Below is the standard FPGA design flow:



1. **Design Source Files:**

- Prepare HDL files using Verilog, VHDL, or other hardware description languages.

2. **Behavioral Simulation:**

- Simulate the design source files to verify their functional correctness at a high level.

3. **Design Synthesis:**

- Convert the HDL code into a gate-level netlist to map the design to FPGA components.

4. **Functional Verification:**

- Validate the synthesized netlist for functional correctness against the design source files.

5. **Design Implementation:**

- Perform mapping, placing, and routing of the design to match FPGA resources.

6. **Static Timing Analysis:**

- Analyze the timing characteristics to ensure that the design meets timing constraints like setup and hold times.

7. **Bit-stream File Generation:**

- Generate a configuration file (bitstream) that can be used to program the FPGA.

8. **Upload on FPGA Board:**

- Program the FPGA device using the generated bitstream file.
- Verify the design on the actual FPGA board to ensure proper operation in real-world conditions.

## **Procedure**

1. Launch Vivado on your system.
2. Click on create new project & provide the project name and directory where you want to save your project.
3. Choose RTL project & make sure to click on the checkbox *"Do not specify sources at this time."*
4. Select device with the following specifications:

- **Product category:** All
- **Family:** Artix-7
- **Sub-family:** Artix-7
- **Package:** xc7a100tcs324

- **Speed grade:** -1
- **Temp grade:** All remaining
- **Part:** xc7a100tcs324-1
- **SI Division:** All remaining

5. **To add Verilog source code:**

- In the flow navigator, go to Project Manager and click on *Add Sources*.
  - Select *Add or create design sources*.
  - Click on the + icon then Create file. Choose file type as Verilog, and give the file a name. Then click Finish.
  - Define the I/O port after that.
6. In the opened editor, write your Verilog code.
7. To create a test bench:
- In the flow navigator, click on *Add Sources* again and select *Add or create simulation sources*.
  - Create a new Verilog file for the test bench.
  - Write the test bench code.
8. Save both Verilog as well as test bench code, and check if there is any error or not. Debug the errors if they arise.
9. In the flow navigator, click on *Run Simulation* & select Run Behavioral Simulation.
10. The waveform window will show the result of your test cases. Verify the waveform.

# EXPERIMENT 1: BASIC GATES

## AIM:

Write a Verilog code to implement all the basic gates.

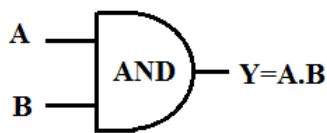
## THEORY:

### 1. AND GATE:

An **AND gate** is a basic digital logic gate that implements logical conjunction—it gives an output of **1 (True)** only when **all** of its inputs are **1 (True)**. If any input is **0 (False)**, the output will be **0 (False)**.

For a 2-input AND gate, the Boolean expression is:

$Y = A \cdot B$  Truth table:

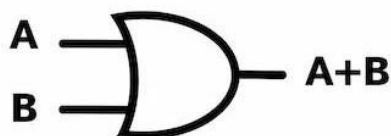


Inputs		Output
A	B	$Y = A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

### 2. OR GATE:

An OR gate is a basic digital logic gate that implements logical disjunction. It produces an output of 1 (True) if at least one of its inputs is 1 (True). If all inputs are 0 (False), the output will be 0 (False).

For a 2-input OR gate, the Boolean expression is :  $Y = A + B$

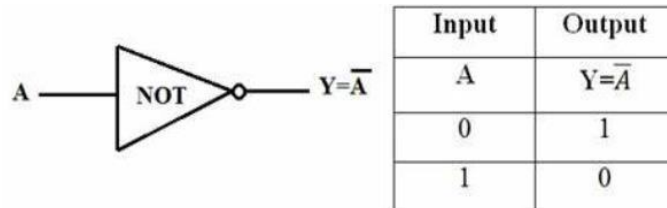


2 input OR gate		
A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

### 3. NOT GATE:

A NOT gate is a basic digital logic gate that inverts the input signal. It is also called an inverter because it produces an output that is the opposite (or negation) of its input. If the input is 1(True), the output will be 0 (False), and if the input is 0 (False), the output will be 1 (True).

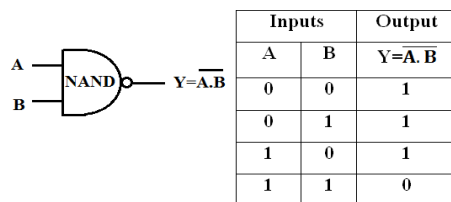
Or a NOT gate, the Boolean expression is simply the negation of the input:



### 4. NAND GATE:

The NAND gate (NOT-AND gate) is a fundamental digital logic gate that performs the inverse of the AND gate operation. It produces an output of 1 (True) unless all of its inputs are 1(True). If all inputs are 1, the output will be 0 (False)

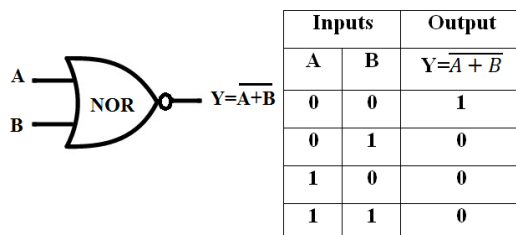
For a 2-input NAND gate, the Boolean expression is:  $Y = A' \cdot B'$



### 5. NOR GATE:

A NOR gate is a fundamental digital logic gate that performs the NOT operation on the output of an OR gate. It produces an output of 1(True) only when all of its inputs are 0(False). If any input is 1(True), the output will be 0 (False). In essence, it is the combination of the OR gate and a NOT gate.

For a 2-input NOR gate, the Boolean expression is:  $Y = A' + B'$

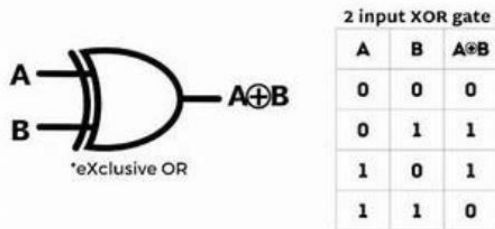


## 6. XOR GATE:

An XOR gate (Exclusive OR gate) is a digital logic gate that outputs **1 (True)** when the number of **1 inputs** is **odd**. Specifically, an XOR gate produces a **1** when exactly one of its inputs is **1**, and it outputs **0** when both inputs are the same (either both are **0** or both are **1**).

It is called "exclusive OR" because it returns **True** only when the inputs are different.

For a **2-input XOR gate**, the Boolean expression is:  $Y = A \oplus B = A \cdot \bar{B} + \bar{A} \cdot B$ .

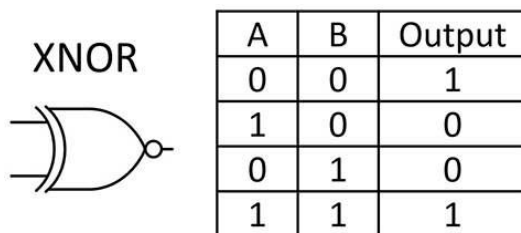


## 7. XNOR Gate

An XNOR gate (Exclusive NOR gate) is a digital logic gate that serves as the inverse of an XOR gate (Exclusive OR gate). It performs a logical operation that outputs **True (1)** when the number of **True inputs** is **even**, and **False (0)** when the number of **True inputs** is **odd**.

For a **2-input XNOR gate**, the Boolean expression is:

$$Y = \bar{A}\bar{B} + AB$$





## CODE:

```
// Module for Basic Logic Gates
module basic_gates(
    input A,    // First Input
    input B,    // Second Input
    output AND_OUT, OR_OUT, NOT_OUT_A, NAND_OUT, NOR_OUT, XOR_OUT, XNOR_OUT
);

// Logic Gate Implementations
assign AND_OUT = A & B;    // AND Gate
assign OR_OUT  = A | B;    // OR Gate
assign NOT_OUT_A = ~A;    // NOT Gate (for input A)
assign NAND_OUT = ~(A & B); // NAND Gate
assign NOR_OUT  = ~(A | B); // NOR Gate
assign XOR_OUT  = A ^ B;   // XOR Gate
assign XNOR_OUT = ~(A ^ B); // XNOR Gate

endmodule
```

## TEST BENCH:

```
// Testbench for Basic Gates
module tb_basic_gates;

// Test Inputs and Outputs
reg A, B;
wire AND_OUT, OR_OUT, NOT_OUT_A, NAND_OUT, NOR_OUT, XOR_OUT, XNOR_OUT;

// Instantiate the Basic Gates Module
basic_gates uut
(.A(A),.B(B),.AND_OUT(AND_OUT),.OR_OUT(OR_OUT),.NOT_OUT_A(NOT_OUT_A),.NAND_OUT(NAND_OUT),.NOR_OUT(NOR_OUT),.XOR_OUT(XOR_OUT),.XNOR_OUT(XNOR_OUT));

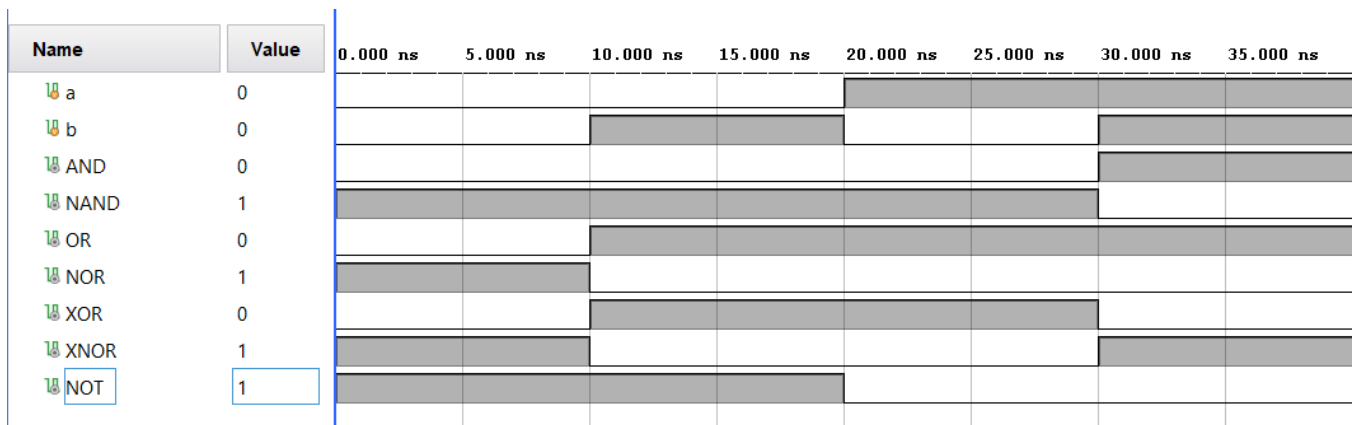
// Initial block for test cases
initial begin
    // Monitor outputs
    $monitor("A=%b, B=%b | AND=%b, OR=%b, NOT_A=%b, NAND=%b, NOR=%b, XOR=%b, XNOR=%b",
        A, B, AND_OUT, OR_OUT, NOT_OUT_A, NAND_OUT, NOR_OUT, XOR_OUT,
```

```

XNOR_OUT);
    // Apply Test Vectors
    A = 0; B = 0; #10;
    A = 0; B = 1; #10;
    A = 1; B = 0; #10;
    A = 1; B = 1; #10;
    $finish; // End Simulation
end
endmodule

```

## OUTPUT:



## EXPERIMENT 2:

### HALF ADDER

#### AIM:

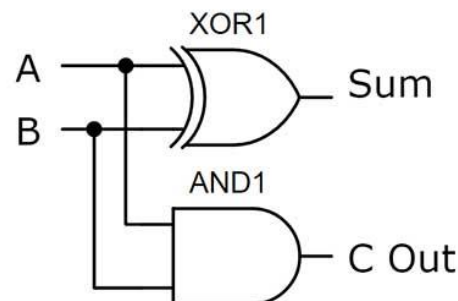
Write a Verilog code to implement Half Adder

#### THEORY:

A half adder is a basic combinational logic circuit used to add two single-bit binary numbers and produce two outputs: sum and carry. The sum is obtained using the XOR operation of the inputs, represented as  $S = A \oplus B$ , while the carry is generated using the AND operation, represented as  $C = A \cdot B$ . The circuit outputs a sum of 1 when exactly one of the inputs is 1, and a carry of 1 when both inputs are 1. The truth table of the half adder defines its operation with two inputs and two outputs. However, the half adder cannot handle a carry input from previous stages, limiting it to single-bit addition, while multi-bit addition requires a full adder.

Half Adder

A	B	C Out	Sum
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0



#### SOURCE CODE:

```
module HalfAdder(  
    input A,      // First input bit  
    input B,      // Second input bit  
    output Sum,   // Sum output  
    output Carry  // Carry output  
);  
    // Logic for sum and carry  
    assign Sum = A ^ B; // XOR for sum  
    assign Carry = A & B; // AND for carry  
endmodule
```

#### TESTBENCH:

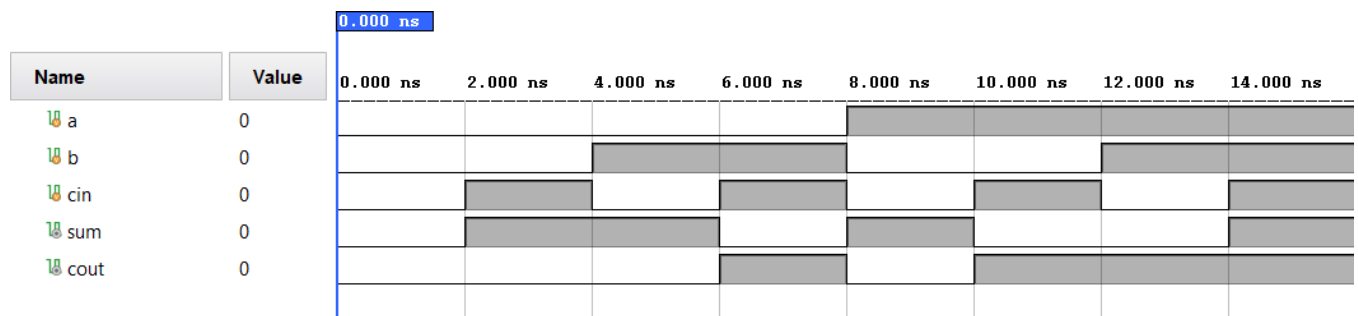
```
module tb_HalfAdder;  
    reg A, B;      // Inputs as reg types for driving values  
    wire Sum, Carry; // Outputs as wire types to observe results
```

```

HalfAdder uut(
  .A(A),
  .B(B),
  .Sum(Sum),
  .Carry(Carry)
);
initial begin
  $monitor("At time %0t, A = %b, B = %b, Sum = %b, Carry = %b", $time, A, B, Sum, Carry);
  A = 0; B = 0;
  #10 A = 0; B = 1;
  #10 A = 1; B = 0;
  #10 A = 1; B = 1;
  $finish;
end
endmodule

```

## OUTPUT:



## EXPERIMENT 3:

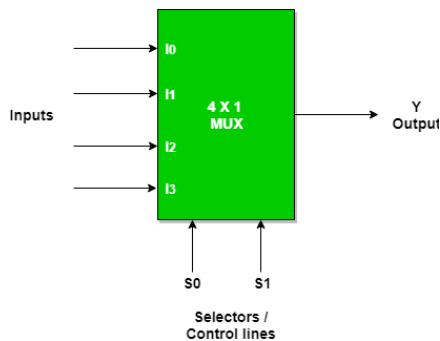
### 4:1 MUX

#### AIM:

Write a Verilog code to implement 4:1 MUX.

#### THEORY:

A 4:1 Multiplexer (MUX) is a combinational logic circuit that selects one of four input signals and forwards the selected signal to a single output line. It is often used for data routing, signal switching, or in applications where multiple inputs need to be processed or transmitted but only one output is required.

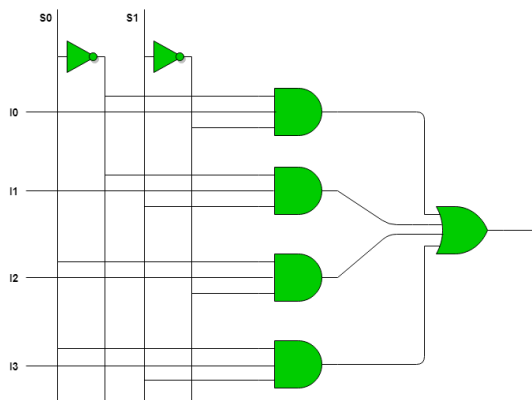


Truth Table

S0	S1	Y
0	0	I0
0	1	I1
1	0	I2
1	1	I3

So, final equation,

$$Y = S0'.S1'.I0 + S0'.S1.I1 + S0.S1'.I2 + S0.S1.I3$$



#### SOURCE CODE:

```
module mux4_1 (
    input wire [3:0] d, // 4 data inputs
    input wire [1:0] sel, // 2-bit select signal
    output reg y // Output
);
```

```

always @(*) begin
    case (sel)
        2'b00: y = d[0];
        2'b01: y = d[1];
        2'b10: y = d[2];
        2'b11: y = d[3];
        default: y = 1'b0; // Default case
    endcase
end
endmodule

```

## TESTBENCH:

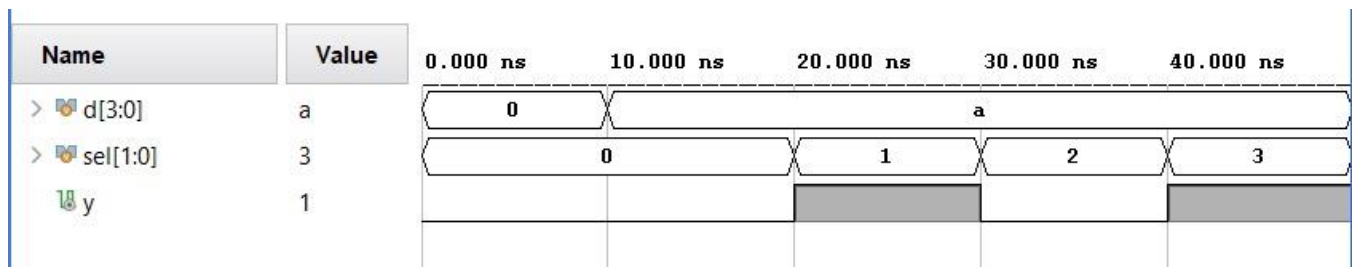
```

module tb_mux4_1;
    reg [3:0] d;
    reg [1:0] sel;
    wire y;

    mux4_1 uut (.d(d),.sel(sel),.y(y));
    initial begin
        $monitor("At time %0t: sel = %b, d = %b -> y = %b", $time, sel, d, y);
        d = 4'b1010; sel = 2'b00;
        #10 d = 4'b1010; sel = 2'b01;
        #10 d = 4'b1010; sel = 2'b10;
        #10 d = 4'b1010; sel = 2'b11;
        $finish;
    end
endmodule

```

## OUTPUT



## EXPERIMENT 4:

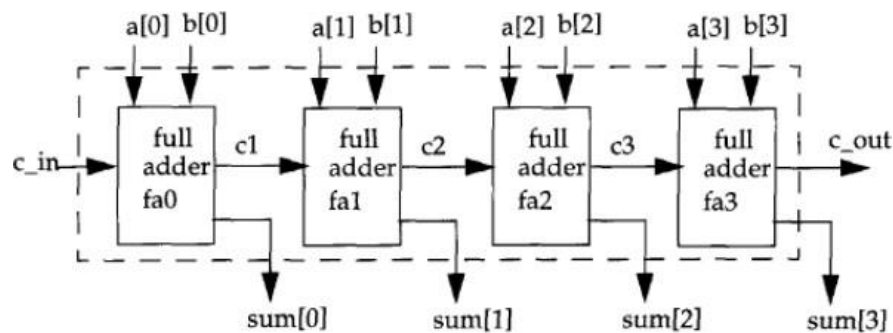
### 4 BIT ADDER USING 1-BITADDER

#### AIM:

Write a Verilog code to implement 4 Bit Full adder using 1-Bit Full Adder.

#### THEORY:

A 4-bit adder is designed to perform the addition of two 4-bit numbers by processing each bit individually, starting from the least significant bit (LSB) to the most significant bit (MSB). The process begins by adding the corresponding LSBs of the two input numbers, along with any carry-in ( $C_{in}$ ) from the previous lower bit. The result of the addition gives the sum bit for that position, and the carry-out ( $C_{out}$ ) is generated, which is then passed to the next higher bit's addition. This carry propagation continues from the LSB to the MSB, ensuring that each bit of the sum is calculated in sequence. The final carry-out from the MSB represents the carry for the overall operation, indicating whether the result exceeds 4 bits. This type of adder is often called a ripple-carry adder because the carry is "rippled" through each successive bit addition.



#### SOURCE CODE:

##### 1. 1-Bit Full Adder

```
module FullAdder (  
    input A,      // Single bit input A  
    input B,      // Single bit input B  
    input Cin,    // Carry input  
    output Sum,   // Single bit sum output  
    output Cout   // Carry output  
);  
    // Full adder logic  
    assign Sum = A ^ B ^ Cin;
```

```

    assign Cout = (A & B) | (B & Cin) | (A & Cin);
endmodule

```

## 2. 4-Bit Full Adder:

```

module Adder4bit (
    input [3:0] A, // 4-bit input A
    input [3:0] B, // 4-bit input B
    input Cin,    // Carry input
    output [3:0] Sum, // 4-bit sum output
    output Cout   // Final carry output
);
    // Internal wires to carry carry-out between stages
    wire C1, C2, C3;

    // Instantiate four 1-bit Full Adders
    FullAdder FA0 (.A(A[0]), .B(B[0]), .Cin(Cin), .Sum(Sum[0]), .Cout(C1));
    FullAdder FA1 (.A(A[1]), .B(B[1]), .Cin(C1), .Sum(Sum[1]), .Cout(C2));
    FullAdder FA2 (.A(A[2]), .B(B[2]), .Cin(C2), .Sum(Sum[2]), .Cout(C3));
    FullAdder FA3 (.A(A[3]), .B(B[3]), .Cin(C3), .Sum(Sum[3]), .Cout(Cout));
endmodule

```

## **TESTBENCH:**

```

`timescale 1ns / 1ps

```

```

module tb_Adder4bit;

    // Testbench variables
    reg [3:0] A;    // 4-bit input A
    reg [3:0] B;    // 4-bit input B
    reg Cin;        // Carry input
    wire [3:0] Sum; // 4-bit sum output
    wire Cout;      // Final carry output

    // Instantiate the 4-bit Full Adder
    Adder4bit uut (
        .A(A),
        .B(B),
        .Cin(Cin),
        .Sum(Sum),
        .Cout(Cout)
    );

```



```

// Test cases
initial begin
    $display("Time\tA\tB\tCin\tSum\tCout");

    // Apply test inputs
    A = 4'b0001; B = 4'b0010; Cin = 0; #10; // Test case 1
    A = 4'b0101; B = 4'b0011; Cin = 0; #10; // Test case 2
    A = 4'b1111; B = 4'b1111; Cin = 0; #10; // Test case 3
    A = 4'b1010; B = 4'b0101; Cin = 1; #10; // Test case 4
    A = 4'b0000; B = 4'b0000; Cin = 1; #10; // Test case 5
    // End simulation
$finish;
end
endmodule

```

## OUTPUT:

Name	Value	0.000 ns	10.000 ns	20.000 ns	30.000 ns	40.000 ns
> A[3:0]	0	1	5	f	a	0
> B[3:0]	0	2	3	f	5	0
Cin	1					
> Sum[3:0]	1	3	8	e	0	1
Cout	0					

## EXPERIMENT 5:

### 3:8 DECODER

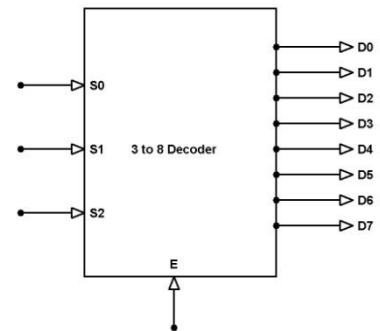
#### AIM:

Write a Verilog code to implement 3:8 Decoder.

#### THEORY:

A 3-to-8 decoder is a digital circuit that accepts 3 binary input signals and decodes them into one of 8 possible output lines. It is a specific type of an n-to-m decoder, where n=3 (the number of input bits) and m=8 (the number of output lines). This means the 3-to-8 decoder can uniquely identify and activate one of the 8 output lines based on the binary value of the 3 input bits.

A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0



#### SOURCE CODE:

```
`timescale 1ns / 1ps
module decoder_3to8 (
    input [2:0] A,    // 3-bit input
    output reg [7:0] D // 8-bit output);
    always @(*) begin
        // Set all outputs to 0 initially
        D = 8'b00000000;
        case (A)
            3'b000: D[0] = 1'b1;
            3'b001: D[1] = 1'b1;
            3'b010: D[2] = 1'b1;
            3'b011: D[3] = 1'b1;
            3'b100: D[4] = 1'b1;
            3'b101: D[5] = 1'b1;
            3'b110: D[6] = 1'b1;
            3'b111: D[7] = 1'b1;
            default: D = 8'b00000000; // Default case for safety
        endcase
    end
end
```

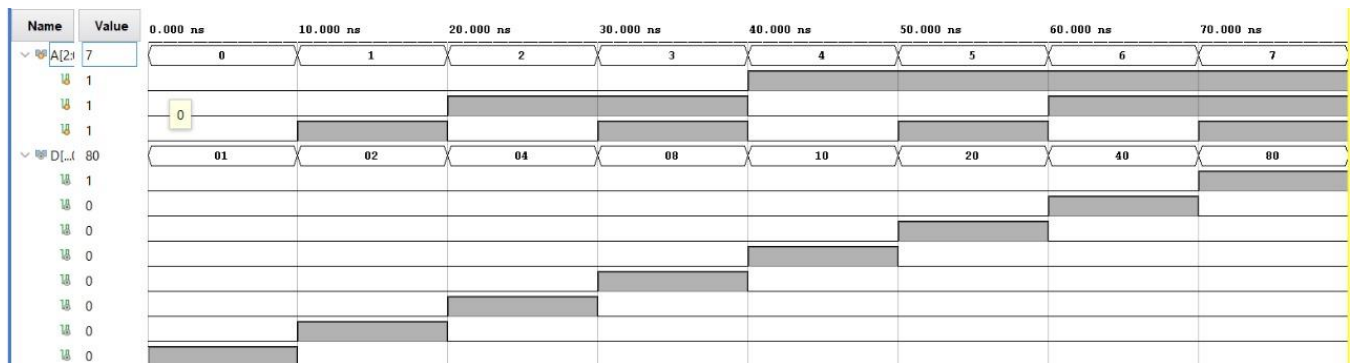
## TESTBENCH:

```

`timescale 1ns / 1ps
module tb_decoder_3to8;
    reg [2:0] A;
    wire [7:0] D;
    decoder_3to8 uut (
        .A(A),
        .D(D)
    );
    initial begin
        $display("Time\tA\tD");
        A = 3'b000; #10;
        A = 3'b001; #10;
        A = 3'b010; #10;
        A = 3'b011; #10;
        A = 3'b100; #10;
        A = 3'b101; #10;
        A = 3'b110; #10;
        A = 3'b111; #10;
        $finish;
    end
endmodule

```

## OUTPUT:



## EXPERIMENT 6:

### 8:3 ENCODER

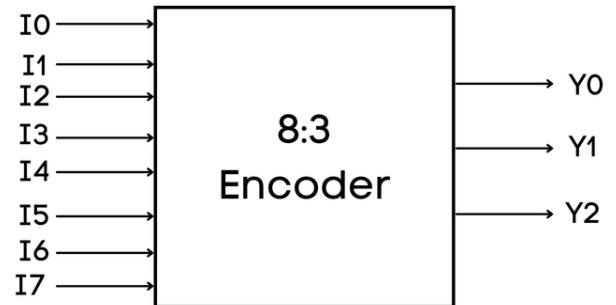
#### AIM:

Write a Verilog code to implement 8:3 Encoder.

#### THEORY:

An **8-to-3 encoder** is a digital circuit that takes 8 input signals and converts them into a 3-bit binary code. The circuit assigns a unique 3-bit binary output for each of the 8 input lines based on the active input, where the output represents the index of the highest-order active input. If multiple inputs are active simultaneously, the encoder typically prioritizes the highest-numbered active input. This encoding reduces the number of output lines from 8 to 3, thus enabling more compact representation of multiple input signals in binary form. The 8-to-3 encoder is often used in digital communication, control systems, and other applications where efficient data representation or signal multiplexing is necessary.

INPUTS								OUTPUTS		
Y <sub>7</sub>	Y <sub>6</sub>	Y <sub>5</sub>	Y <sub>4</sub>	Y <sub>3</sub>	Y <sub>2</sub>	Y <sub>1</sub>	Y <sub>0</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1



#### SOURCE CODE:

```
`timescale 1ns / 1ps
module encoder_8to3 (
    input [7:0] D,    // 8-bit input
    output reg [2:0] Y // 3-bit encoded output
);
    always @(*) begin
        casez (D)
            8'b10000000: Y = 3'b111; // Highest priority D[7]
            8'b01000000: Y = 3'b110; // D[6]
            8'b00100000: Y = 3'b101; // D[5]
            8'b00010000: Y = 3'b100; // D[4]
            8'b00001000: Y = 3'b011; // D[3]
            8'b00000100: Y = 3'b010; // D[2]
```

```

        8'b00000010: Y = 3'b001; // D[1]
        8'b00000001: Y = 3'b000; // Lowest priority D[0]
        default:    Y = 3'b000; // Default case if all inputs are 0
    endcase
end
endmodule

```

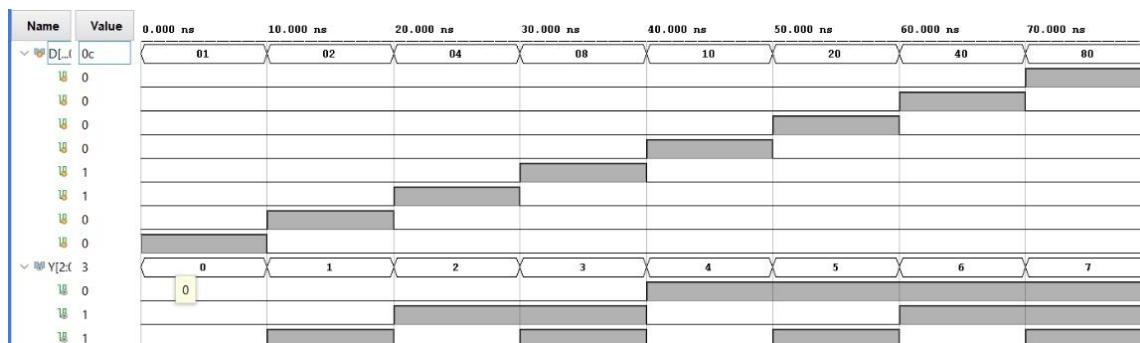
## TESTBENCH:

```

`timescale 1ns / 1ps
module encoder_tb();
    reg [7:0] D;
    wire [2:0] Y;
    encoder_8to3 uut (
        .D(D),
        .Y(Y) );
    initial begin
        $display("Time\tD\t\tY");
        D = 8'b00000001; #10; // Input 0
        D = 8'b00000010; #10; // Input 1
        D = 8'b00000100; #10; // Input 2
        D = 8'b00001000; #10; // Input 3
        D = 8'b00010000; #10; // Input 4
        D = 8'b00100000; #10; // Input 5
        D = 8'b01000000; #10; // Input 6
        D = 8'b10000000; #10; // Input 7
        D = 8'b00001100; #10; // Multiple inputs active
        $display("%4t\t%b\t%b", $time, D, Y);
        // End simulation
        $finish;
    end
endmodule

```

## OUTPUT:



## EXPERIMENT 7:

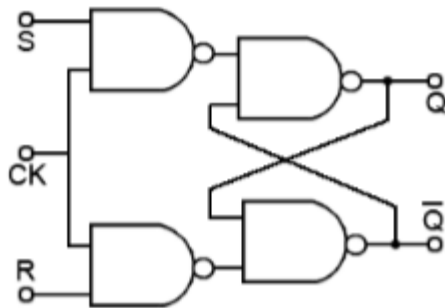
### SR FLIPFLOP

#### AIM:

Write a Verilog code to implement SR Flipflop

#### THEORY:

An **SR (Set-Reset) Flip-Flop** is a basic digital storage element used to store a single bit of data. It has two inputs, labeled **S (Set)** and **R (Reset)**, and two outputs, typically **Q** and **Q'** (the inverse of Q). The flip-flop operates based on the values of the Set and Reset inputs. When **S=1** and **R=0**, the output **Q** is set to 1, and **Q'** becomes 0. When **S=0** and **R=1**, the output **Q** is reset to 0, and **Q'** becomes 1. If both **S=0** and **R=0**, the outputs hold their previous state, essentially making the flip-flop a memory element. The condition **S=1** and **R=1** is typically not allowed as it causes an invalid or undefined state. SR flip-flops are commonly used in building sequential circuits for data storage and state control.



TRUTH TABLE

S	R	$Q_N$	$Q_{N+1}$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	-
1	1	1	-

#### SOURCE CODE:

```
`timescale 1ns / 1ps
module sr_flip_flop (
    input S,      // Set input
    input R,      // Reset input
    input clk,    // Clock input
    output reg Q, // Output
    output reg Qn // Complementary Output
);
always @(posedge clk) begin
    case ({S, R})
        2'b00: ; // No change
        2'b01: begin
            Q = 1'b0;
            Qn = 1'b1;
        end
        2'b10: begin
            Q = 1'b1;
            Qn = 1'b0;
        end
        2'b11: ; // Invalid state
    endcase
end
```

```

        2'b10: begin
            Q = 1'b1;
            Qn = 1'b0;
        end
        2'b11: begin
            Q = 1'bx; // Invalid state
            Qn = 1'bx;
        end
    endcase
end
endmodule

```

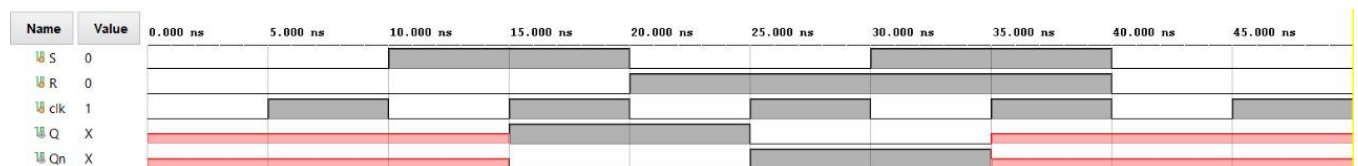
## TESTBENCH:

```

`timescale 1ns / 1ps
module tb_sr_flip_flop;
    reg S;
    reg R;
    reg clk;
    wire Q;
    wire Qn;
    sr_flip_flop uut (.S(S),.R(R),.clk(clk),.Q(Q),.Qn(Qn));
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // Toggle clk every 5ns
    end
    initial begin
        $display("Time\tS\tR\tQ\tQn");
        S = 0; R = 0; #10
        S = 1; R = 0; #10;
        S = 0; R = 1; #10;
        S = 1; R = 1; #10;
        S = 0; R = 0; #10;
        $finish;
    end
endmodule

```

## OUTPUT:



## EXPERIMENT 8:

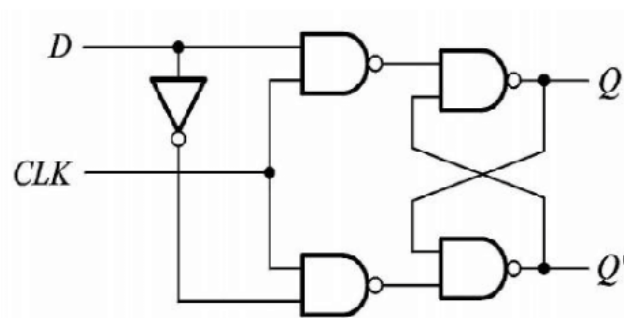
### D FLIPFLOP

#### AIM:

Write a Verilog code to implement D Flipflop

#### THEORY:

A **D Flip-Flop** is a digital memory element that stores a single bit of data. It has a data input (**D**), a clock input (**CLK**), and two outputs: **Q** and its complement **Q'**. The flip-flop captures the value of the **D** input at the rising or falling edge of the clock signal, and the output **Q** reflects this value. When the clock signal transitions, the **D** input value is transferred to the output **Q**, and **Q'** becomes the inverse. The **D Flip-Flop** ensures that the output always follows the **D** input at the clock edge and maintains this value until the next clock cycle. This makes the **D Flip-Flop** essential for data synchronization and storage in sequential circuits, providing stable state storage and eliminating potential problems from unintended data changes, often used in shift registers, counters, and memory units.



Q	D	Q(t+1)
0	0	0
0	1	1
1	0	0
1	1	1

#### SOURCE CODE:





```
`timescale 1ns / 1ps
module d_flip_flop (
    input D,      // Data input
    input clk,    // Clock input
    output reg Q, // Output
    output reg Qn // Complementary Output
);
    always @(posedge clk) begin
        Q <= D;    // Assign the value of D to Q on the clock's rising edge
        Qn <= ~D;  // Complementary value of Q
    end
endmodule
```



## TESTBENCH:

```
`timescale 1ns / 1ps
module tb_d_flip_flop;
    reg D;
    reg clk;
    wire Q;
    wire Qn;
    d_flip_flop uut (.D(D),.clk(clk),.Q(Q),.Qn(Qn));
    initial begin
        clk = 0;
        forever #10 clk = ~clk; // Toggle clk every 10ns
    end
    initial begin
        $display("Time\tD\tQ\tQn");
        D = 0; #10;
        D = 1; #10;
        D = 0; #10;
        D = 1; #10;
        D = 1; #20;
    end
end
endmodule
```

## OUTPUT:

Name	Value	0.000 ns	10.000 ns	20.000 ns	30.000 ns	40.000 ns	50.000 ns
 D	1						
 clk	1						
 Q	1						
 Qn	0						

## EXPERIMENT 9:

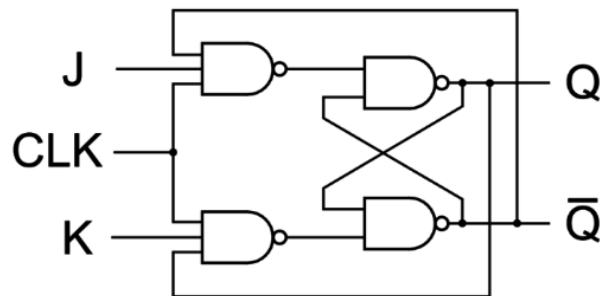
### JK FLIPFLOP

#### AIM:

Write a Verilog code to implement JK Flipflop

#### THEORY:

A **JK Flip-Flop** is a versatile digital storage element used for sequential logic, having inputs **J**, **K**, a clock input (**CLK**), and two outputs (**Q** and **Q'**). It addresses the undefined state problem of an SR flip-flop when both inputs are active. The JK flip-flop operates as follows: when **J=0** and **K=0**, it retains the previous state; when **J=1** and **K=0**, it sets the output **Q=1**; when **J=0** and **K=1**, it resets the output **Q=0**; and when **J=1** and **K=1**, it toggles the output, switching **Q** to its complement. This toggle functionality makes the JK flip-flop widely used in counters and control circuits. The state transitions occur on the triggering edge of the clock signal, ensuring reliable synchronization in digital systems.



Clock	J	K	Q <sub>n+1</sub>	State
0	X	X	Q <sub>n</sub>	
1	0	0	Q <sub>n</sub>	Hold
1	0	1	0	Reset
1	1	1	1	Set
1	1	1	Q <sub>n</sub>	Toggle

#### SOURCE CODE:

```
`timescale 1ns / 1ps
module jk_flip_flop (
    input J,      // J input
    input K,      // K input
    input clk,    // Clock input
    output reg Q, // Output
    output reg Qn // Complementary Output
);
always @(posedge clk) begin
    case ({J, K})
        2'b00: ; // No change
        2'b01: begin
            Q = 1'b0; // Reset
            Qn = 1'b1;
        end
        2'b10: begin
            Q = 1'b1; // Set
            Qn = 1'b0;
        end
        2'b11: begin
            Q = ~Q; // Toggle
            Qn = ~Qn;
        end
    endcase
end
```

```

        2'b10: begin
            Q = 1'b1; // Set
            Qn = 1'b0;
        end
        2'b11: begin
            Q = ~Q; // Toggle
            Qn = ~Qn;
        end
    endcase
end
endmodule

```

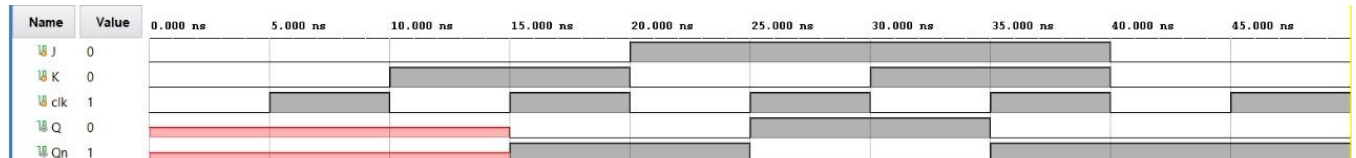
## TESTBENCH:

```

`timescale 1ns / 1ps
module tb_jk_flip_flop;
    reg J;
    reg K;
    reg clk;
    wire Q;
    wire Qn;
    jk_flip_flop uut (.J(J),.K(K),.clk(clk),.Q(Q),.Qn(Qn));
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // Toggle clk every 5ns
    end
    initial begin
        $display("Time\tJ\tK\tQ\tQn");
        J = 0; K = 0; #10;
        J = 0; K = 1; #10;
        J = 1; K = 0; #10;
        J = 1; K = 1; #10;
        J = 0; K = 0; #10;
        $finish;
    end
endmodule

```

## OUTPUT:



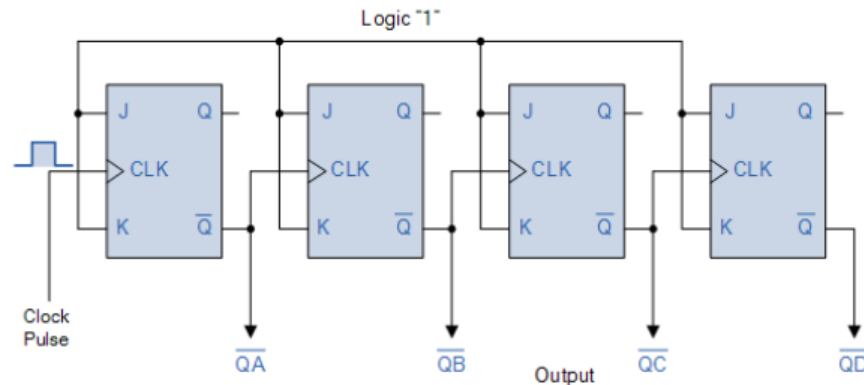
## EXPERIMENT 10: SYNCHRONOUS UP/DOWN COUNTER

### AIM:

Write a Verilog code to implement Synchronous Up/Down Counter.

### THEORY:

A **Synchronous Up/Down Counter** is a sequential circuit that can count both upwards and downwards in a binary sequence, based on a control signal, and all its flip-flops are triggered simultaneously by the same clock signal. The direction of counting is determined by an additional input control signal, typically labeled as **up/down** or **ud**. When the control signal is set to count up, the counter increases its count value with each clock pulse, and when set to count down, it decrements the count value. Since it is synchronous, all flip-flops in the counter change state simultaneously at the clock edge, ensuring better timing consistency compared to asynchronous counters. These counters are widely used in digital systems for applications like timers, event counters, and frequency division, due to their precise control and predictable operation.



### SOURCE CODE:

```
module counter(clk, reset, up_down, load, data, count);
    input clk, reset, load, up_down;
    input [3:0] data;
    output reg [3:0] count;
    always @(posedge clk) begin
        if (reset) // Set counter to zero
            count <= 0;
        else if (load) // Load the counter with data value
            count <= data;
        else if (up_down) // Count up
            count <= count + 1;
        else // Count down
            count <= count - 1;
    end
endmodule
```

```

end
endmodule

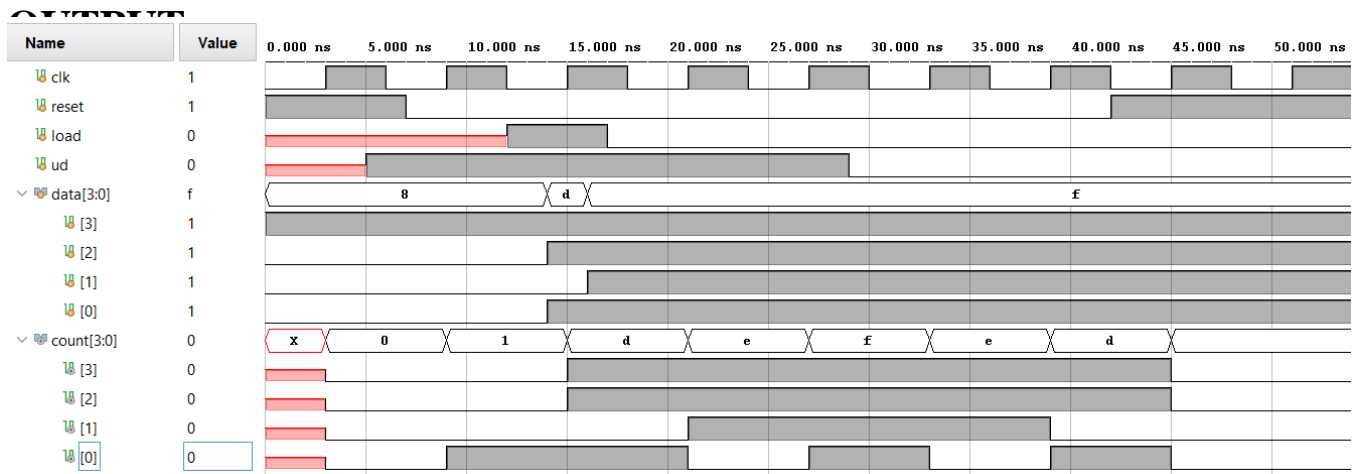
```

## TESTBENCH:

```

module counter_tb;
  reg clk, reset, load, ud;
  reg [3:0] data;
  wire [3:0] count;
  counter ct_1(.up_down(ud), .clk(clk), .reset(reset), .load(load), .data(data), .count(count));
  initial begin
    clk = 1'b0;
    repeat (30) #3 clk = ~clk;
  end
  initial begin
    reset = 1'b1;
    #7 reset = 1'b0;
    #35 reset = 1'b1;
  end
  initial begin
    #12 load = 1'b1;
    #5 load = 1'b0;
  end
  initial begin
    #5 ud = 1'b1;
    #24 ud = 1'b0;
  end
  initial begin
    data = 4'b1000;
    #14 data = 4'b1101;
    #2 data = 4'b1111;
  end
  initial begin
    $monitor("time=%0d, reset=%b, load=%b, ud=%b, data=%d, count=%d",
      $time, reset, load, ud, data, count);
  end
end
endmodule

```



## EXPERIMENT 11:

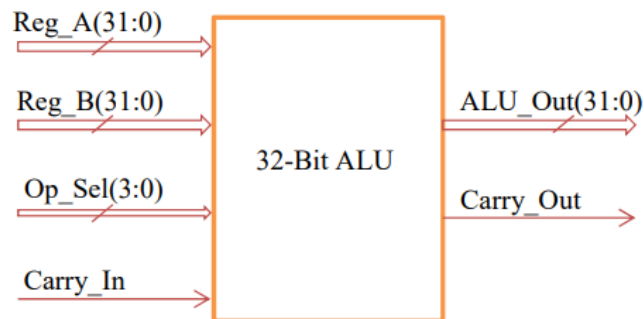
### 32 BIT ALU

#### AIM:

Write a Verilog code to implement 32 Bit ALU.

#### THEORY:

A 32-bit ALU (Arithmetic Logic Unit) is a digital circuit that performs arithmetic and logical operations on 32-bit binary numbers. It is a fundamental component in processors, handling operations like addition, subtraction, AND, OR, XOR, shifts, and comparisons. The 32-bit ALU takes two 32-bit input operands, processes them according to the operation specified by a control signal, and outputs a 32-bit result. It also provides flags such as zero, carry, overflow, and negative, which are used for condition checking in various applications. The ALU's design involves combinational logic circuits, with multiplexers, adders, and logic gates to perform its tasks efficiently. In modern processors, the ALU is optimized for speed, power consumption, and integration with other parts of the CPU for overall performance.



#### SOURCE CODE:

```
module ALU32(A, B, Op, En, Y);
    input En;
    input [3:0] Op;
    input [31:0] A, B;
    output reg [32:0] Y;

    always @(En, Op, A, B)
    begin
        if (En == 0)
            Y = 33'bz;
        else
            begin
                case (Op)
                    4'b0000: Y = A + B;
                    4'b0001: Y = A - B;
```

```

        4'b0010: Y = ~A;
        4'b0011: Y = A & B;
        4'b0100: Y = A | B;
        4'b0101: Y = ~(A & B);
        4'b0110: Y = A ^ B;
        default: Y = 33'bz;
    endcase
end
end
endmodule

```

## TESTBENCH:

```

`timescale 1ns/1ns
module ALU32_tb;
    reg En;
    reg [3:0] Op;
    reg [31:0] A, B;
    wire [32:0] Y;

    ALU32 A1(A, B, Op, En, Y);

    initial
    begin
        #100 A = 32'b0010101010101010101010101010101;
        B = 32'b0000000000000000000011111111111111;
        Op = 4'b0011;
        En = 1'b1;

        #100 A = 32'b0001111000011110000111100001111;
        B = 32'b00110011001100110011001100110011;
        Op = 4'b0101;
        En = 1'b1;
    end
endmodule

```

## OUTPUT:

