

## INDEX



<b>Ex No:</b>	<b>Design of basic combinational circuits using HDL. Simulate it using Xilinx/Altera Software and implement by Xilinx/Altera FPGA</b>
<b>Date:</b>	

### **AIM:**

To design and implement the Combinational circuits using Verilog.

### **SOFTWARE AND HARDWARE REQUIREMENTS:**

Simulation tool : Xilinx 8.1

FPGA Kit: Universal Development Board (UDB) Personal Computer (PC)

### **THEORY:**

#### **HALF ADDER:**

A half adder is a fundamental digital circuit used in computer arithmetic to add two binary numbers. It has two inputs and two outputs. Here's a short explanation of a half adder:

- Inputs: A half adder takes two binary inputs, typically labeled as 'A' and 'B'. These inputs can only be either 0 or 1, representing the binary digits.
- Outputs: A half adder has two outputs: - Sum (S): This output represents the result of adding the two binary inputs 'A' and 'B'. It's calculated as  $S = A \text{ XOR } B$  (XOR stands for exclusive OR). - Carry (C): This output represents the carry generated when adding 'A' and 'B'. It's calculated as  $C = A \text{ AND } B$ , where AND is a logical AND operation. In summary, a half adder can add two binary digits and produce both the sum and the carry. However, it can't account for any carry from previous additions, making it a basic building block for more complex adder circuits like the full adder, which can handle multiple bits and carry inputs.

#### **FULL ADDER:**

A full adder is a more complex digital circuit used in computer arithmetic to add binary numbers. It takes three binary inputs and has two outputs. Here's a short explanation of a full adder:

- Inputs: A full adder takes three binary inputs: A, B, and Cin (carry-in). A and B are the binary numbers to be added, while Cin represents the carry from a previous addition.
- Outputs: A full adder has two outputs: - Sum (S): This output represents the result of adding A, B, and Cin. It's calculated as  $S = (A \text{ XOR } B) \text{ XOR } \text{Cin}$ , where XOR is the exclusive OR operation. - Carry-out (Cout): This output represents the carry generated when adding A, B, and Cin. It's calculated as  $\text{Cout} = (A \text{ AND } B) \text{ OR } (\text{Cin} \text{ AND } (A \text{ XOR } B))$ , where AND is the logical AND operation, and OR is the logical OR operation. In summary, a full adder can

add three binary digits (A,B, and Cin) and produce both the sum and the carry-out. It can handle not only the addition of A and B but also account for any carry from previous additions,making it a key component in binary arithmetic circuits. Full adders are used to construct multi-bit binary adders and other complex arithmetic operations in digital electronics.

### **HALF SUBTRACTOR:**

A half subtractor is a fundamental digital circuit used in computer arithmetic to subtract binary numbers. It has two inputs and two outputs. Here's a short explanation of a half subtractor:

- Inputs: A half subtractor takes two binary inputs: - 'A': The minuend, which is the number from which you want to subtract. - 'B': The subtrahend, which is the number to be subtracted from the minuend.
- Outputs: A half subtractor has two outputs: - 'Difference (D)': This output represents the result of subtracting B from A. It's calculated as  $D = A \text{ XOR } B$ , where XOR is the exclusive OR operation. - 'Borrow (Bout)': This output indicates whether a borrow is required for the subtraction. It's calculated as  $Bout = A \text{ AND } (\text{NOT } B)$ , where AND is the logical AND operation, and NOT is the logical NOT operation.In summary, a half subtractor can subtract two binary digits (A and B) and produce both the difference (D) and the borrow (Bout).
- However, it does not account for any borrow from previous subtractions, making it a basic building block for more complex subtractor circuits like the full subtractor, which can handle multiple bits and borrow inputs.Full subtractors are used to perform binary subtraction for multi-bit numbers and are essential components in digital arithmetic circuits.

### **FULL SUBTRACTOR:**

A full subtractor is a digital circuit used for subtracting binary numbers. It takes three binary inputs and has two outputs. Here's an explanation of a full subtractor:

- Inputs: A full subtractor takes three binary inputs: - A: The minuend, which is the number from which you want to subtract. - B: The subtrahend, which is the number to be subtracted from the minuend. - Borrow-in (Bin): This input represents the borrow from a previous subtraction or is set to 0 if there's no previous borrow.

- Outputs: A full subtractor has two outputs:
  - Difference (D): This output represents the result of subtracting B from A, considering the borrow from Bin. It's calculated as  $D = (A \text{ XOR } B) \text{ XOR } \text{Bin}$ , where XOR is the exclusive OR operation.
  - Borrow-out (Bout): This output indicates whether a borrow is required for the subtraction. It's calculated as  $\text{Bout} = (\text{NOT } A \text{ AND } B) \text{ OR } (\text{NOT } A \text{ AND } \text{Bin}) \text{ OR } (B \text{ AND } \text{Bin})$ , where AND is the logical AND operation, OR is the logical OR operation, and NOT is the logical NOT operation.
- Functionality: A full subtractor can subtract three binary digits (A, B, and Bin) and produce both the difference (D) and the borrow-out (Bout). It accounts for the borrow from previous subtractions, making it suitable for multi-bit binary subtraction.
- Applications: Full subtractors are used to construct multi-bit binary subtractors and other complex arithmetic operations in digital electronics. They are vital in microprocessors, arithmetic logic units (ALUs), and many other digital systems for performing subtraction operations on binary numbers.

In summary, a full subtractor is a versatile building block in digital electronics, allowing for the subtraction of binary numbers while efficiently managing borrow operations to handle multi-bit arithmetic.

## MULTIPLEXER:

4x1 multiplexer (MUX) is a digital circuit that is used to select one of four input data lines and route it to a single output line based on the control inputs. Here's an explanation of a 4x1 MUX:

- Inputs: A 4x1 MUX has four data input lines, typically labeled D0, D1, D2, and D3. These are the data sources that you want to select from.
- Control Inputs: It has two control inputs: Select lines (S1 and S0): These control inputs determine which of the four data inputs gets transmitted to the output. They can be used to select one of the four data lines. The number of select lines (S1, S0) determines the number of input lines ( $2^n$  for n select lines).
- Inputs: A 4x1 MUX has four data input lines, typically labeled D0, D1, D2, and D3. These are the data sources that you want to select from.

Control Inputs: It has two control inputs: Select lines (S1 and S0): These control inputs determine which of the four data inputs gets transmitted to the output. They can be used to select one of the four data lines. The number of select lines (S1, S0) determines the number of input lines ( $2^n$  for n select lines).

- The output Y is equal to the data input (D0, D1, D2, or D3) that corresponds to the binary value formed by the control inputs (S1, S0).

When  $(S_1, S_0) = (0, 0)$ , the output  $Y = D_0$ . When  $(S_1, S_0) = (0, 1)$ , the output  $Y = D_1$ . When  $(S_1, S_0) = (1, 0)$ , the output  $Y = D_2$ . When  $(S_1, S_0) = (1, 1)$ , the output  $Y = D_3$ .

4x1 MUXes are used in digital design for various purposes, including data routing, data selection, and control logic. They are building blocks for creating more complex digital circuits and are widely used in applications like memory access, data processing, and multiplexing/demultiplexing data streams.

### **DEMUTIPLEXER:**

A demultiplexer (DEMUX) is a digital circuit that takes a single input and routes it to one of four output lines based on control inputs. Here's an explanation of a DEMUX:

1. Input: A DEMUX has a single input line, often labeled as  $D$ . This is the data source that you want to distribute to one of the four output lines.
2. Control Inputs: It has two control inputs: - Select lines ( $S_1$  and  $S_0$ ): These control inputs determine which of the four output lines the input data is routed to. They can be used to select one of the four output lines. The number of select lines ( $S_1, S_0$ ) determines the number of output lines ( $2^n$  for  $n$  select lines).
3. Outputs: The DEMUX has four output lines, typically labeled as  $Y_0, Y_1, Y_2$ , and  $Y_3$ . The selected input data is sent to one of these output lines based on the control inputs ( $S_1, S_0$ ).
4. Functionality: The DEMUX operates based on the control inputs ( $S_1$  and  $S_0$ ). The input data ( $D$ ) is routed to one of the output lines ( $Y_0, Y_1, Y_2$ , or  $Y_3$ ). The output  $Y_x$  (where  $x$  is 0, 1, 2, or 3) is equal to the input data ( $D$ ) when the control inputs ( $S_1, S_0$ ) correspond to the binary value  $x$ .

When  $(S_1, S_0) = (0, 0)$ , the output  $Y_0$  receives the input data.

When  $(S_1, S_0) = (0, 1)$ , the output  $Y_1$  receives the input data.

When  $(S_1, S_0) = (1, 0)$ , the output  $Y_2$  receives the input data.

When  $(S_1, S_0) = (1, 1)$ , the output  $Y_3$  receives the input data.

DEMUXes are used in digital design for various purposes, including data distribution, data routing, and control logic. They are building blocks for creating more complex digital circuits and are widely used in applications where a single data source needs to be distributed to one of multiple destinations.

### **ENCODER:**

A 4x2 line encoder is a digital combinational circuit that takes four input lines and encodes them into a 2-bit binary output based on the active input

line. Here's an explanation of a 4x2 line encoder:

- Inputs: A 4x2 line encoder has four input lines, typically labeled as I<sub>0</sub>, I<sub>1</sub>, I<sub>2</sub>, and I<sub>3</sub>. These input lines represent different data sources, and only one of them can be active at a time.
- Outputs: It has two output lines, typically labeled as E<sub>0</sub> and E<sub>1</sub>. These are the 2-bit binary output lines that encode the active input line.
- Functionality: The 4x2 line encoder determines which of the four input lines (I<sub>0</sub>, I<sub>1</sub>, I<sub>2</sub>, I<sub>3</sub>) is active (has a logical '1') and encodes it into a 2-bit binary output. The output lines (E<sub>0</sub> and E<sub>1</sub>) will represent the binary code corresponding to the active input line.

When I<sub>0</sub> is active (I<sub>0</sub> = 1), the output is E<sub>0</sub>E<sub>1</sub> = 00.

When I<sub>1</sub> is active (I<sub>1</sub> = 1), the output is E<sub>0</sub>E<sub>1</sub> = 01.

When I<sub>2</sub> is active (I<sub>2</sub> = 1), the output is E<sub>0</sub>E<sub>1</sub> = 10.

When I<sub>3</sub> is active (I<sub>3</sub> = 1), the output is E<sub>0</sub>E<sub>1</sub> = 11.

The 4x2 line encoder is used in digital systems to prioritize and encode multiple input lines, often used for applications such as data selection, signal prioritization, and control logic. It allows for the efficient encoding of active inputs into a binary code for further processing in digital circuits.

## DECODER:

A 2x4 line decoder is a digital combinational circuit that takes a 2-bit binary input and decodes it to select one of four output lines. Here's an explanation of a 2x4 line decoder:

1. Inputs: A 2x4 line decoder has two input lines, typically labeled as A<sub>1</sub> and A<sub>0</sub>. These two input lines represent a 2-bit binary code. In this configuration, there are four possible input combinations (00, 01, 10, 11).
2. Outputs: It has four output lines, usually labeled as Y<sub>0</sub>, Y<sub>1</sub>, Y<sub>2</sub>, and Y<sub>3</sub>. Each output corresponds to one of the possible input combinations.

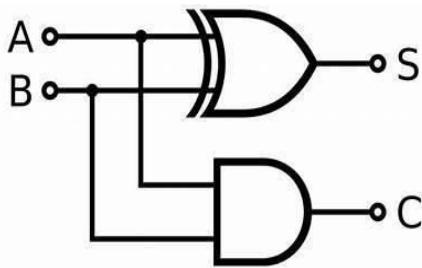
Functionality: The 2x4 line decoder takes the 2-bit binary input (A<sub>1</sub>, A<sub>0</sub>) and activates one of its four output lines based on the input value. The output line that corresponds to the binary value represented by the inputs is set to a logical '1', while the other output lines are set to '0'. This means only one output line is active at a time, depending on the input value.

When (A<sub>1</sub>, A<sub>0</sub>) = (0, 0), the output Y<sub>0</sub> is '1', and Y<sub>1</sub>, Y<sub>2</sub>, and Y<sub>3</sub> are '0'.  
When (A<sub>1</sub>, A<sub>0</sub>) = (0, 1), the output Y<sub>1</sub> is '1', and Y<sub>0</sub>, Y<sub>2</sub>, and Y<sub>3</sub> are '0'.  
When (A<sub>1</sub>, A<sub>0</sub>) = (1, 0), the output Y<sub>2</sub> is '1', and Y<sub>0</sub>, Y<sub>1</sub>, and Y<sub>3</sub> are '0'.  
When (A<sub>1</sub>, A<sub>0</sub>) = (1, 1), the output Y<sub>3</sub> is '1', and Y<sub>0</sub>, Y<sub>1</sub>, and Y<sub>2</sub> are '0'.

2x4 line decoders are used in digital design for various applications, including memory addressing, data routing, and control logic. They help select one of the

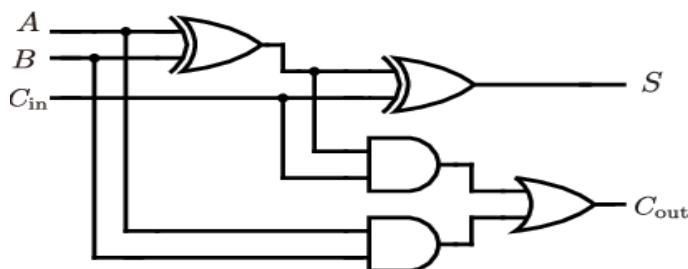
multiple output lines based on the binary input, allowing for efficient data distribution and control in digital systems.

#### HALF ADDER CIRCUIT AND TRUTH TABLE:



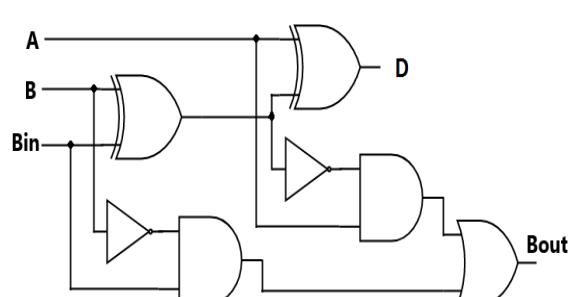
Inputs		Outputs	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

#### FULL ADDER CIRCUIT AND TRUTH TABLE:



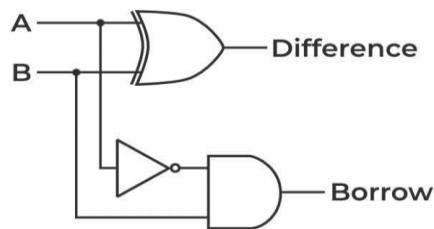
Inputs			Outputs	
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

#### FULL SUBTRACTOR CIRCUIT AND TRUTH TABLE:



Inputs			Outputs	
A	B	Borrow <sub>in</sub>	Diff	Borrow
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

### HALF SUBTRACTOR CIRCUIT AND TRUTH TABLE:

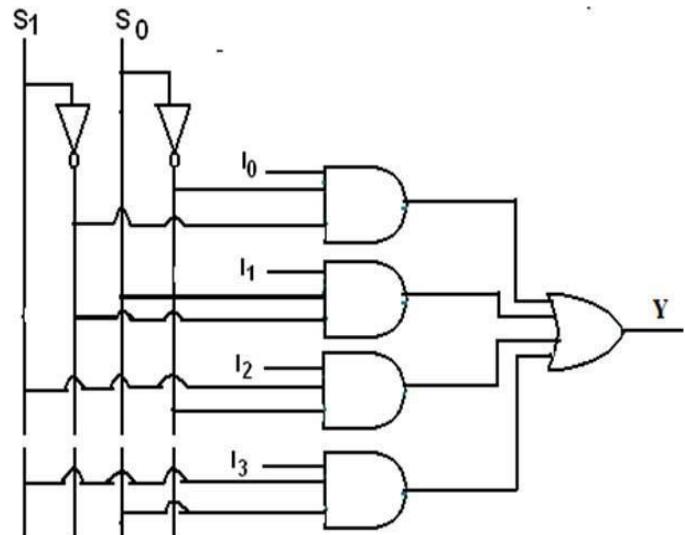


Inputs		Outputs	
A	B	Diff	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

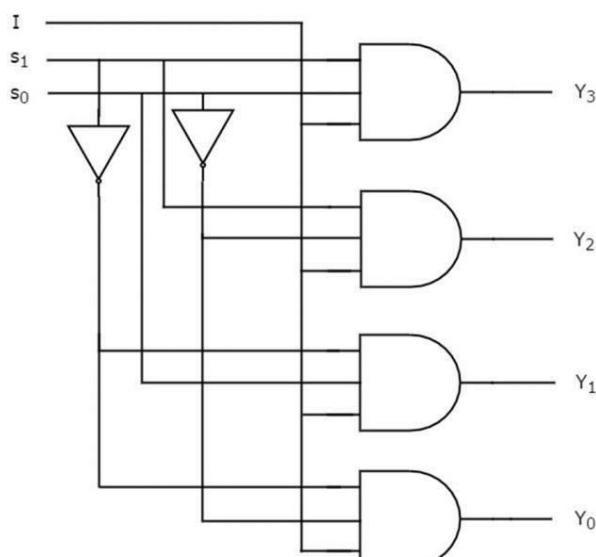
### MULTIPLEXER CIRCUIT AND TRUTH TABLE:

Input	S1	S0	Y
I <sub>0</sub>	0	0	I <sub>0</sub>
I <sub>1</sub>	0	1	I <sub>1</sub>
I <sub>2</sub>	1	0	I <sub>2</sub>
I <sub>3</sub>	1	1	I <sub>3</sub>

$$Y = S_1 S_0 I_3 + S_1 \bar{S}_0 I_2 + \bar{S}_1 S_0 I_1 + \bar{S}_1 \bar{S}_0 I_0$$

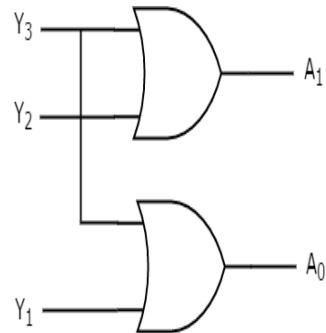


### DEMULTIPLEXER CIRCUIT AND TRUTH TABLE:



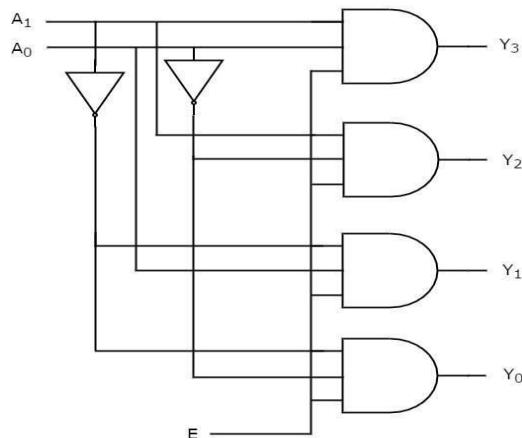
INPUTS		Output			
S <sub>1</sub>	S <sub>0</sub>	Y <sub>3</sub>	Y <sub>2</sub>	Y <sub>1</sub>	Y <sub>0</sub>
0	0	0	0	0	A
0	1	0	0	A	0
1	0	0	A	0	0
1	1	A	0	0	0

## ENCODER CIRCUIT AND TRUTH TABLE:



INPUTS				OUTPUTS	
$Y_3$	$Y_2$	$Y_1$	$Y_0$	$A_1$	$A_0$
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

## DECODER CIRCUIT AND TRUTH TABLE:



Enable	INPUTS			OUTPUTS			
	$E$	$A_1$	$A_0$	$Y_3$	$Y_2$	$Y_1$	$Y_0$
0	X	X	X	0	0	0	0
1	0	0	0	0	0	0	1
1	0	1	0	0	0	1	0
1	1	0	0	0	1	0	0
1	1	1	0	1	0	0	0

## PROCEDURE:

- Create a new project in Xilinx ISE project navigator.
- Open project->new
- Create a Verilog module.
- Type the file name->save.
- Open the program.
- Select the simulator under “source for” in the design window.
- Compile the design and check the syntax under behavior simulate behavioral model.
- Run the simulation.
- Output graph will be displayed

## HALF ADDER :

### Program:

```

module half(a,b,sum,carry);
input a,b;
output sum,carry;
assign sum=a^b;
assign carry=a&b;
endmodule

```

**FULL ADDER:****Program:**

```
module full(a,b,c,sum,carry);
input a,b,c;
output sum,carry;
assign sum=a^b^c;
assign carry=(a&b)|(b&c)|(c&a);
endmodule
```

**HALF SUBTRACTOR:****Program:**

```
module halfsub(
input a,
input b,
output diff,
output borrow
);
wire c;
not(c,a);
xor(diff,a,b);
and(borrow,c,b);
endmodule
```

**FULL SUBTRACTOR:****Program:**

```
module fullsub(
input a,b,c,
output diff,borrow
);
wire w1,w2,w3,w4;
not(w1,a);
and(w2,w1,b),(w3,b,c),(w4,w1,c);
or(borrow,w2,w3,w4);
xor(diff,a,b,c);
endmodule
```

**MULTIPLEXER:****Program:**

```
module mul(a,b,c,d,s0,s1,y);
input a,b,c,d,s0,s1;
output y;
wire w,w1,w2,w3,w4,w5;
not(w,s0),(w1,s1);
and(w2,a,w,w1);
and(w3,b,w,s1);
and(w4,c,s0,w1);
and(w5,d,s0,s1);
```

```
or(y,w2,w3,w4,w5);
endmodule
```

### **DEMUTIPLEXER:**

#### **Program:**

```
module DEMUX(
input s0,
input s1,
input c1,
output y0,
output y1,
output y2,
output y3
);
wire w,w1;
not(w,s0),(w1,s1);
and(y0,w,w1,c1);
and(y1,w,s1,c1);
and(y2,s0,w1,c1);
and(y3,s0,s1,c1);
endmodule
```

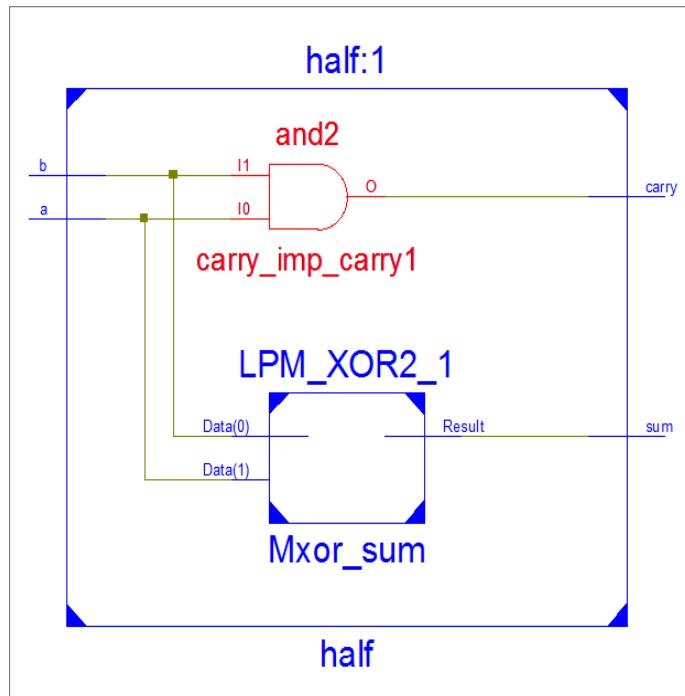
### **ENCODER:**

#### **Program:**

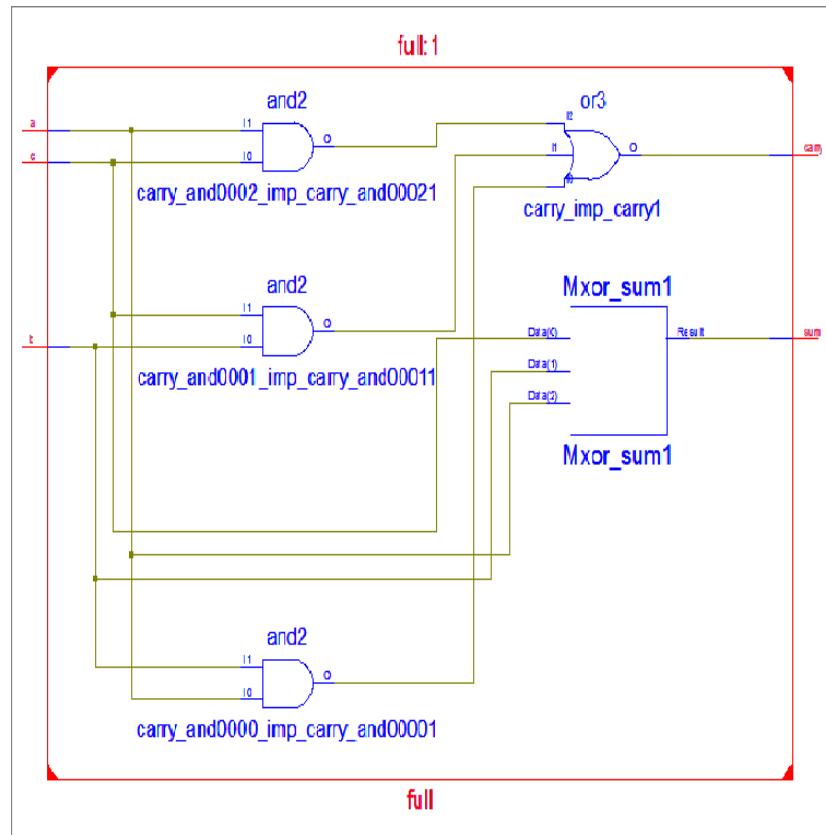
```
module Encoderrrr(
input y0,
input y1,
input y2,
input y3,
output a0,
output a1
);
or(a0,y2,y3);
or(a1,y1,y0);
endmodule
```

**DECODER:****Program:**

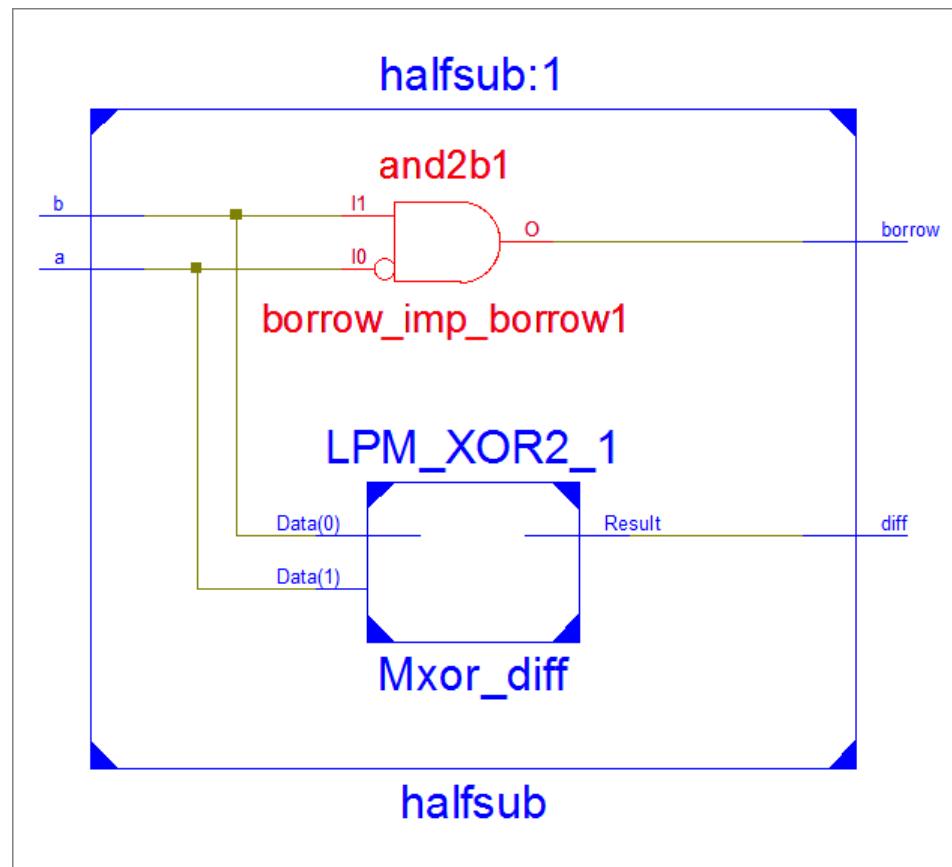
```
module decoder(  
    input s0,  
    input s1,  
    output y0,  
    output y1,  
    output y2,  
    output y3  
)  
    wire w,w1;  
    not(w,s0),(w1,s1);  
    and(y0,w,w1);  
    and(y1,w,s1);  
    and(y2,s0,w1);  
    and(y3,s0,s1);  
endmodule
```

**HALF ADDER RTL SCHEMATIC:**

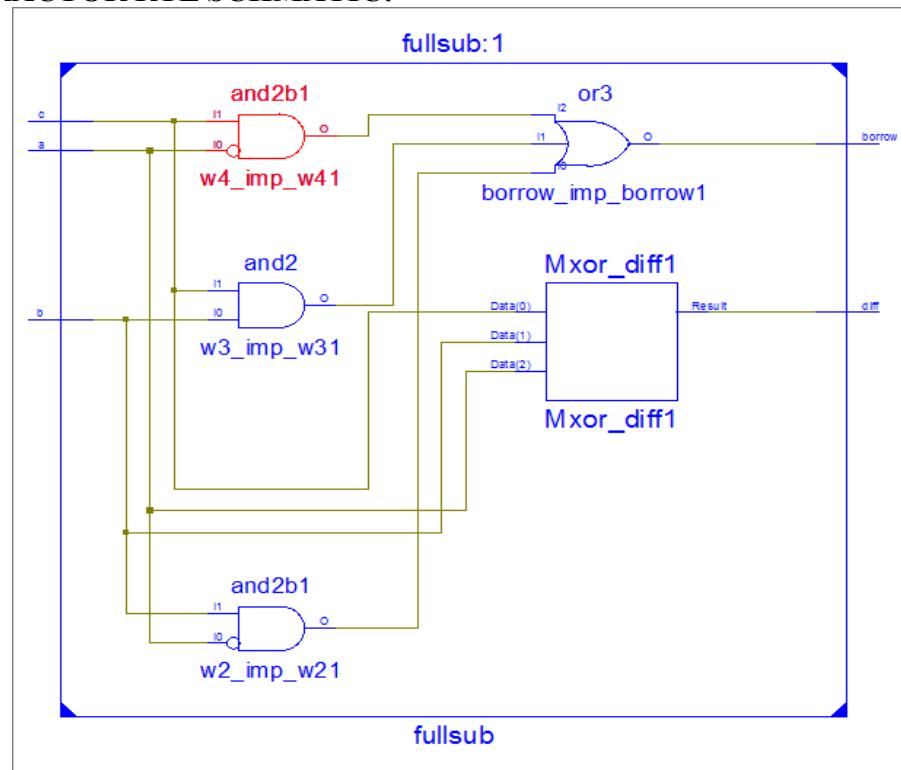
### FULL ADDER RTL SCHEMATIC:



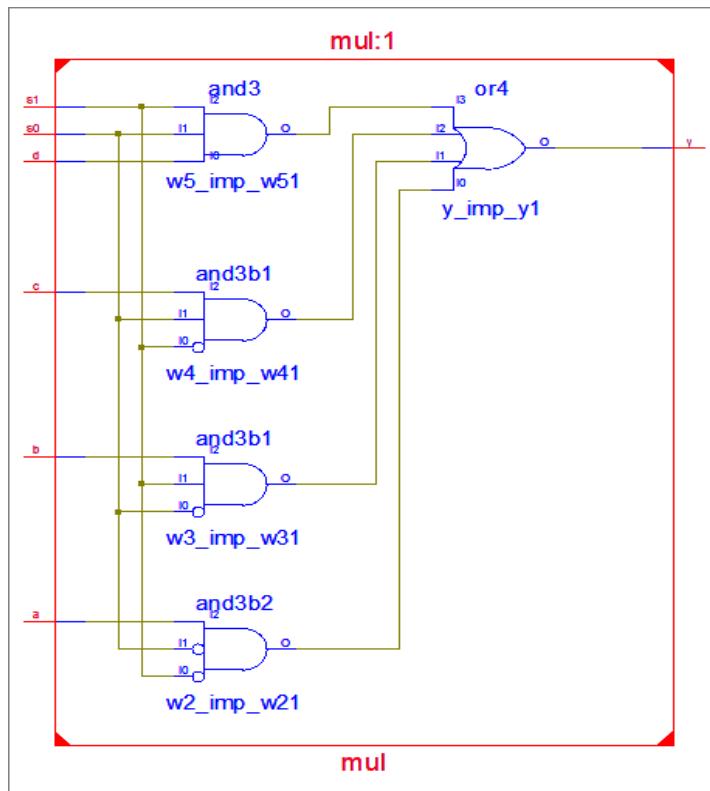
### HALF SUBTRACTOR RTL SCHEMATIC:



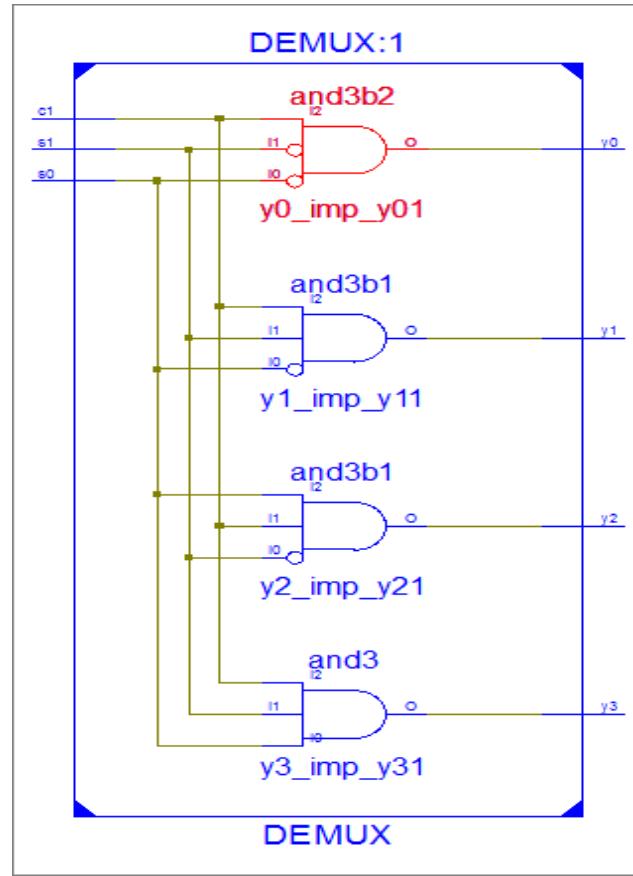
## FULL SUBTRACTOR RTL SCHMATIC:



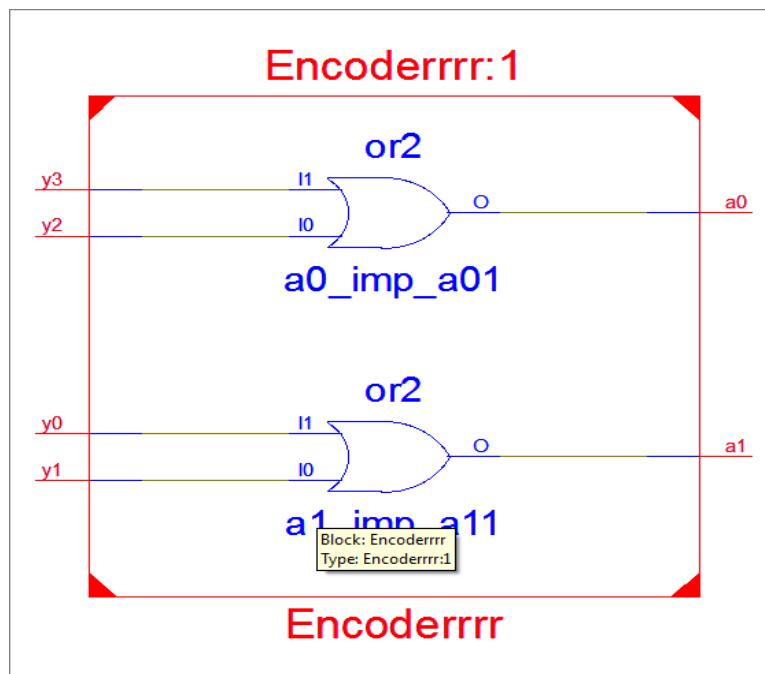
## MULTIPLEXER RTL SCHEMATIC:



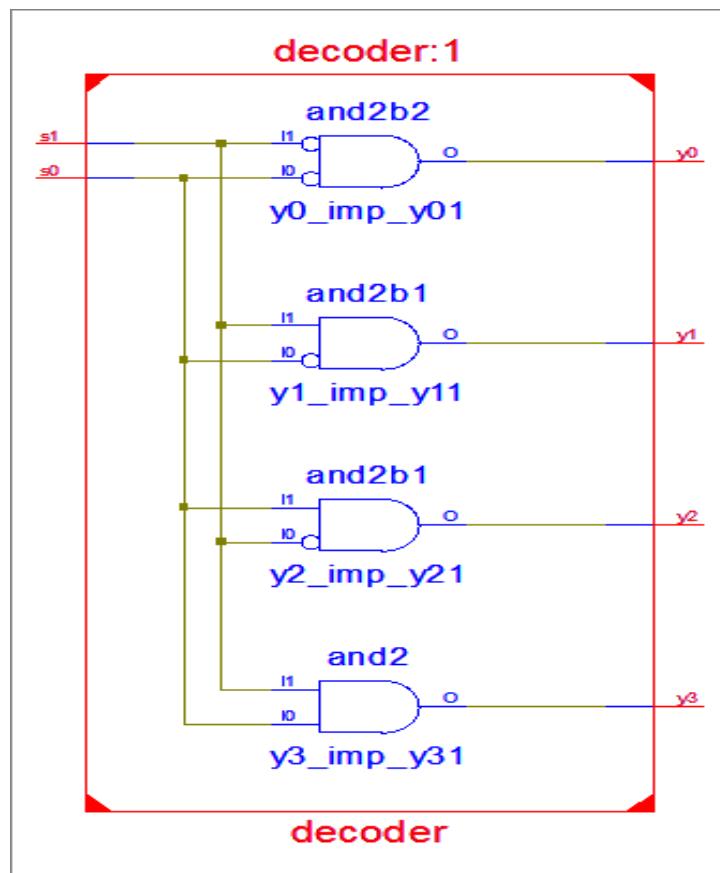
### DEMUTIPLEXER RTL SCHEMATIC:



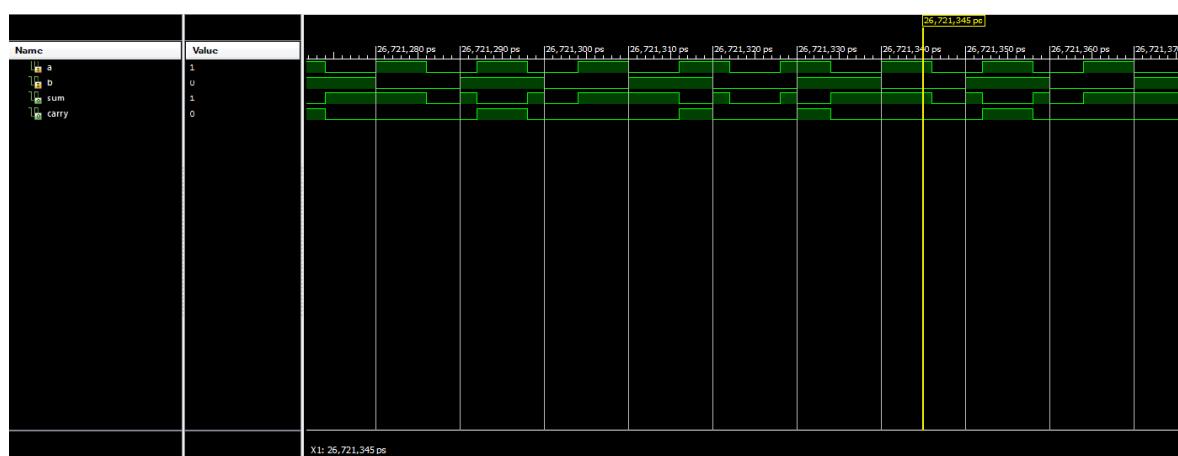
### ENCODER RTL SCHEMATIC:



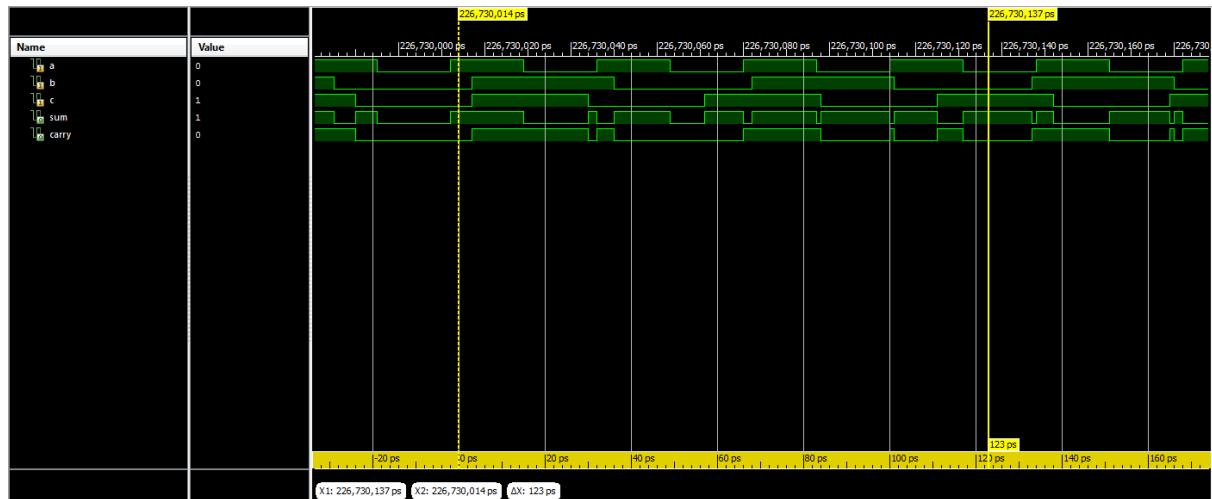
## DECODER RTL SCHEMATIC:



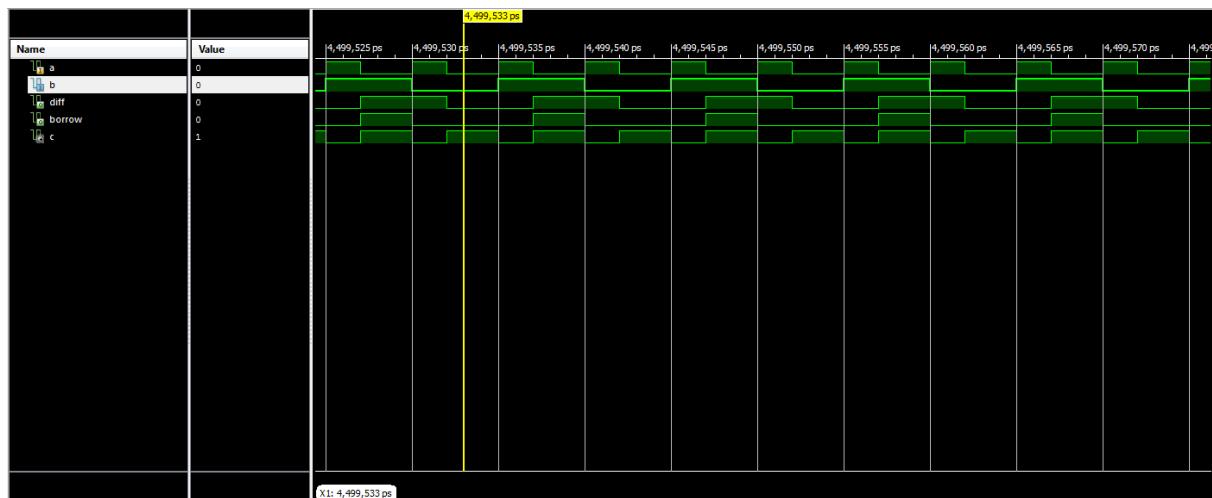
## HALF ADDER OUTPUT:



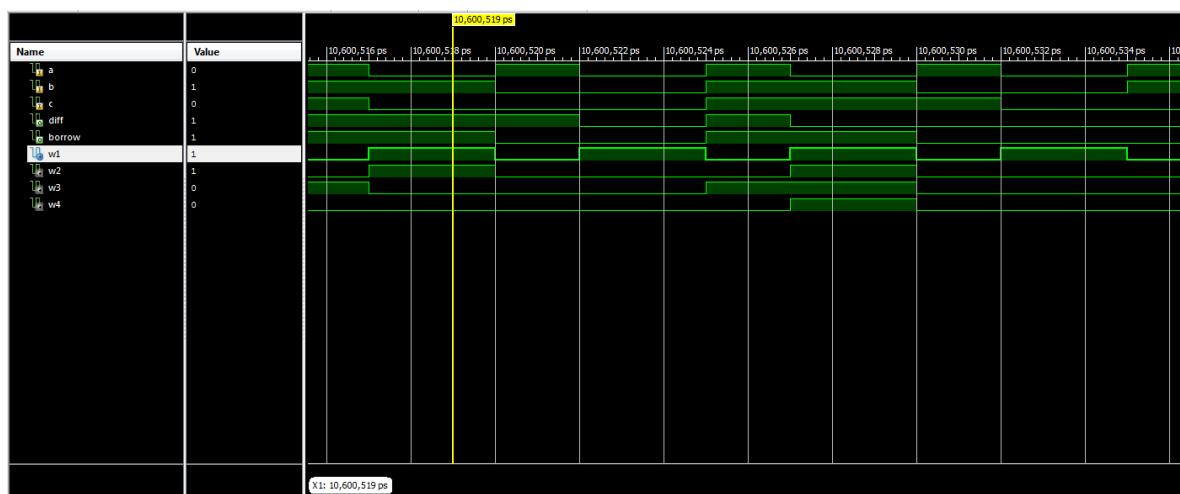
## FULL ADDER OUTPUT:



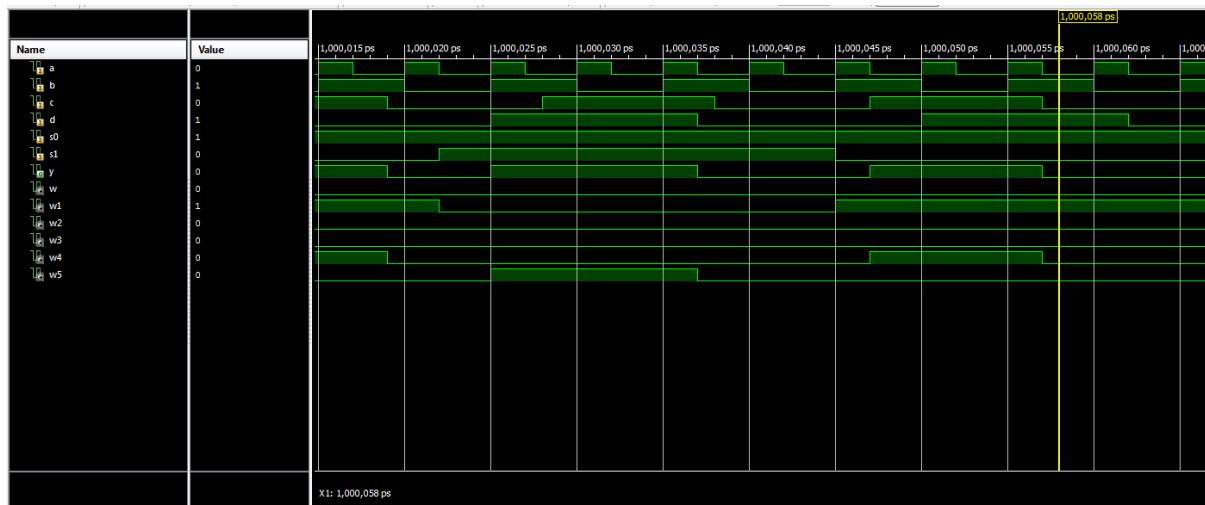
## HALF SUBTRACTOR OUTPUT:



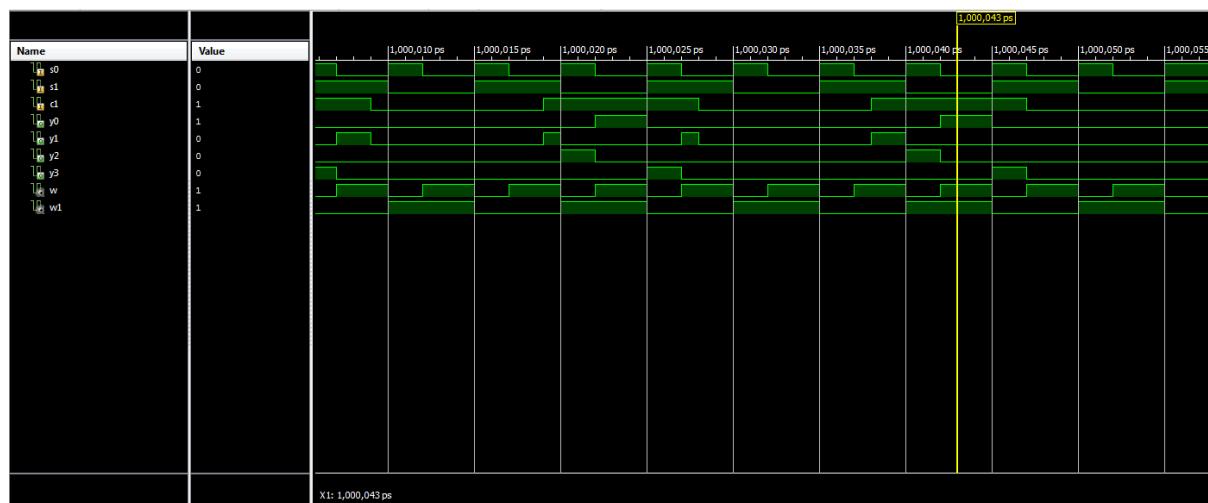
## FULL SUBTRACTOR OUTPUT:



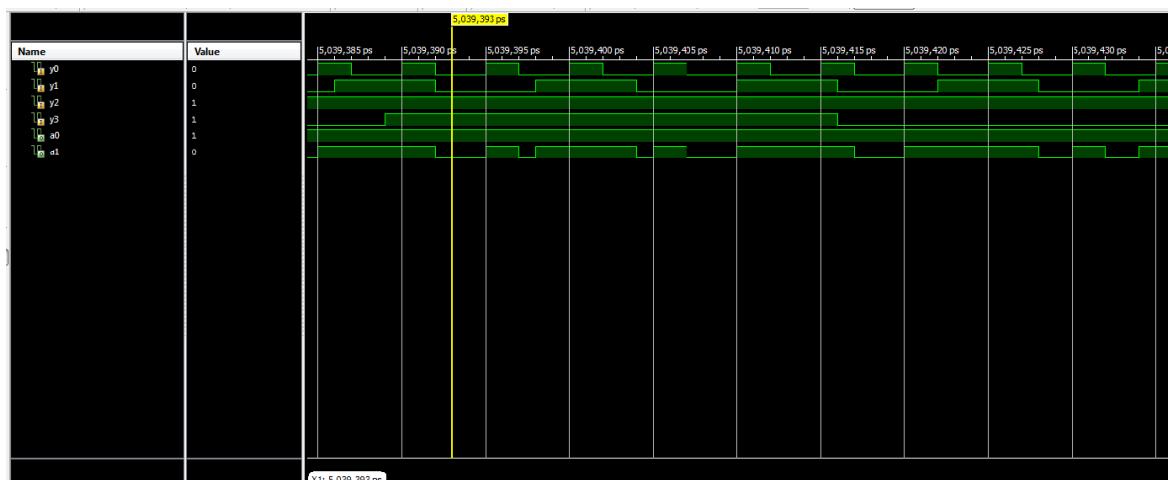
## MULTIPLEXER OUTPUT:



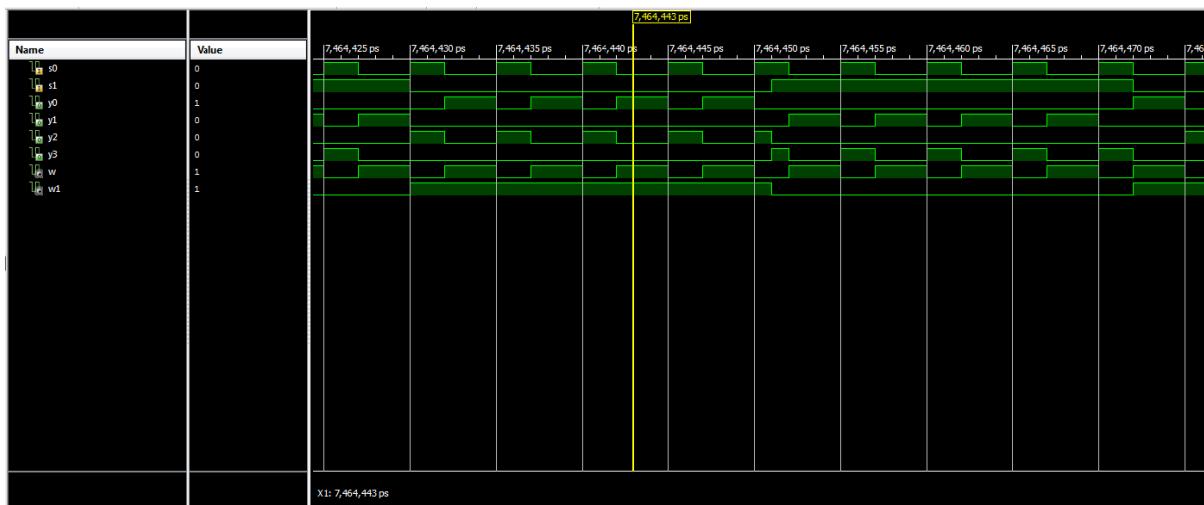
## DEMUTIPLEXER OUTPUT:



## ENCODER OUTPUT:



## DECODER OUTPUT:



## RESULT:

Hence the combinational circuits was implemented using Verilog.

<b>Ex No:</b>	Design of basic Sequential (Flip Flops) circuits using HDL. Simulate it using Xilinx/Altera Software and implement by Xilinx/Altera FPGA
<b>Date:</b>	

### **AIM:**

To design and implement the Sequential (Flip Flops) circuits using Verilog.

### **SOFTWARE AND HARDWARE REQUIREMENTS:**

Simulation tool : Xilinx 8.1

FPGA Kit: Universal Development Board (UDB) Personal Computer (PC)

### **THEORY:**

#### **JK FLIP-FLOP**

Due to the undefined state in the SR flip flop, another flip flop is required in electronics. The JK flip flop is an improvement on the SR flip flop where S=R=1 is not a problem. The input condition of J=K=1, gives an output inverting the output state. However, the outputs are the same when one tests the circuit practically. In simple words, If J and K data input are different (i.e. high and low) then the output Q takes the value of J at the next clock edge. If J and K are both low then no change occurs. If J and K are both high at the clock edge then the output will toggle from one state to the other. JK Flip Flop can function as Set or Reset Flip flop.

#### **D FLIPFLOP**

D flip flop is an electronic devices that is known as “delay flip flop” or “data flip flop” which is used to store single bit of data. D flip flops are synchronous or asynchronous. The clock single required for the synchronous version of D flip flops but not for the asynchronous one. The D flip flop has two inputs, data and clock input which controls the flip flop. When clock input is high, the data is transferred to the output of the flip flop and when the clock input is low, the output of the flip flop is held in its previous state.

#### **T FLIPFLOP**

A T flip-flop is like a JK flip-flop. These are basically single-input versions of JK flip-flops. This modified form of the JK is obtained by connecting inputs J and K together. It has only one input along with the clock input. These flip-flops are called T flip-flops because of their ability to complement their state i.e. Toggle, hence they are named Toggle flip-flops.

#### **SR FLIPFLOP**

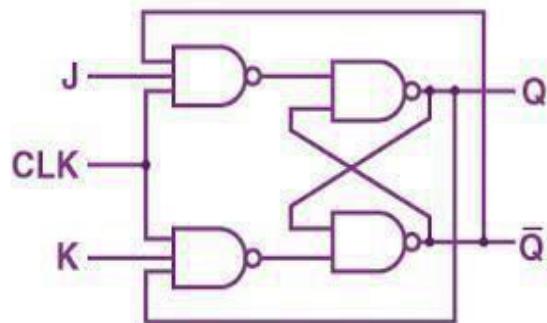
The SR flip flop is a 1-bit memory bistable device having two inputs, i.e., SET and RESET. The SET input 'S' set the device or produce the output 1, and the RESET input 'R' reset the device or produce the output 0. The SET and RESET inputs are labeled as S and R, respectively. The SR flip flop stands for "Set-Reset" flip flop. The reset input is used to get back the flip flop to its original state from the current state with an output 'Q'. This output depends on the set and reset conditions, which is either at the logic level "0" or "1".

### **PROCEDURE:**

- Create a new project in Xilinx ISE project navigator.
- Open project->new
- Create a Verilog module.
- Type the file name->save.
- Open the program.
- Select the simulator under “source for” in the design window.
- Compile the design and check the syntax under behavior simulate behavioral model.
- Run the simulation.
- Output graph will be displayed.

### **CIRCUIT DIAGRAM:**

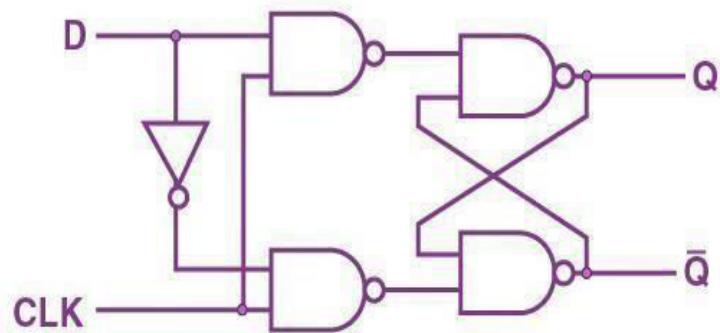
#### **JK FLIPFLOP**



**Truth Table**

J	K	$Q_N$	$Q_{N+1}$
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	1	0

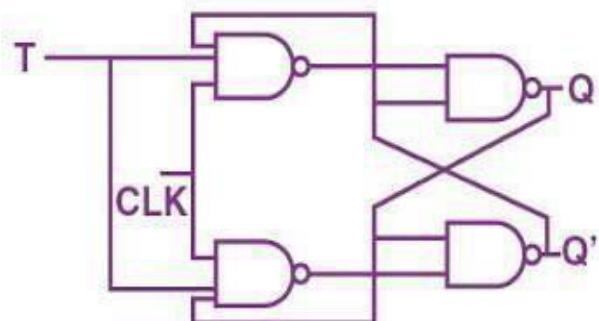
## D FLIPFLOP



Truth Table

Q	D	$Q_{(t+1)}$
0	0	0
0	1	1
1	0	0
1	1	1

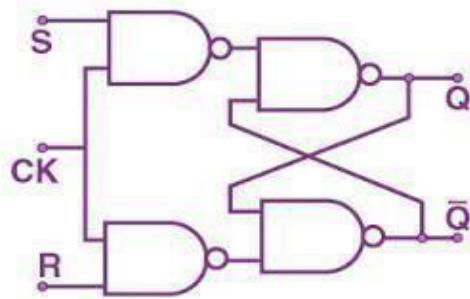
## T FLIPFLOP



Truth Table

T	$Q_N$	$Q_{N+1}$
0	0	0
0	1	1
1	0	1
1	1	0

## SR FLIPFLOP



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	-

## JK FLIPFLOP

### Program:

```
module jkff(input clk,rst_n,input j,k,output reg q,output q_bar);
always@ (posedge clk)
begin
if(!rst_n) q<=0;
else
begin
case({j,k})
2'b00:
q<=q;
2'b01:
q<=1'b0;
2'b10:
q<=1'b1;
2'b11:
q<=q;
endcase
end
end
```

```
assign q_bar=~q;  
endmodule
```

## D FLIPFLOP

### Program:

```
module D_flipflop (  
input clk, rst_n,  
input d,  
output reg q  
);  
always@(posedge clk) begin  
if(!rst_n) q <= 0;  
else q <= d;  
end  
endmodule
```

## T FLIPFLOP

### Program:

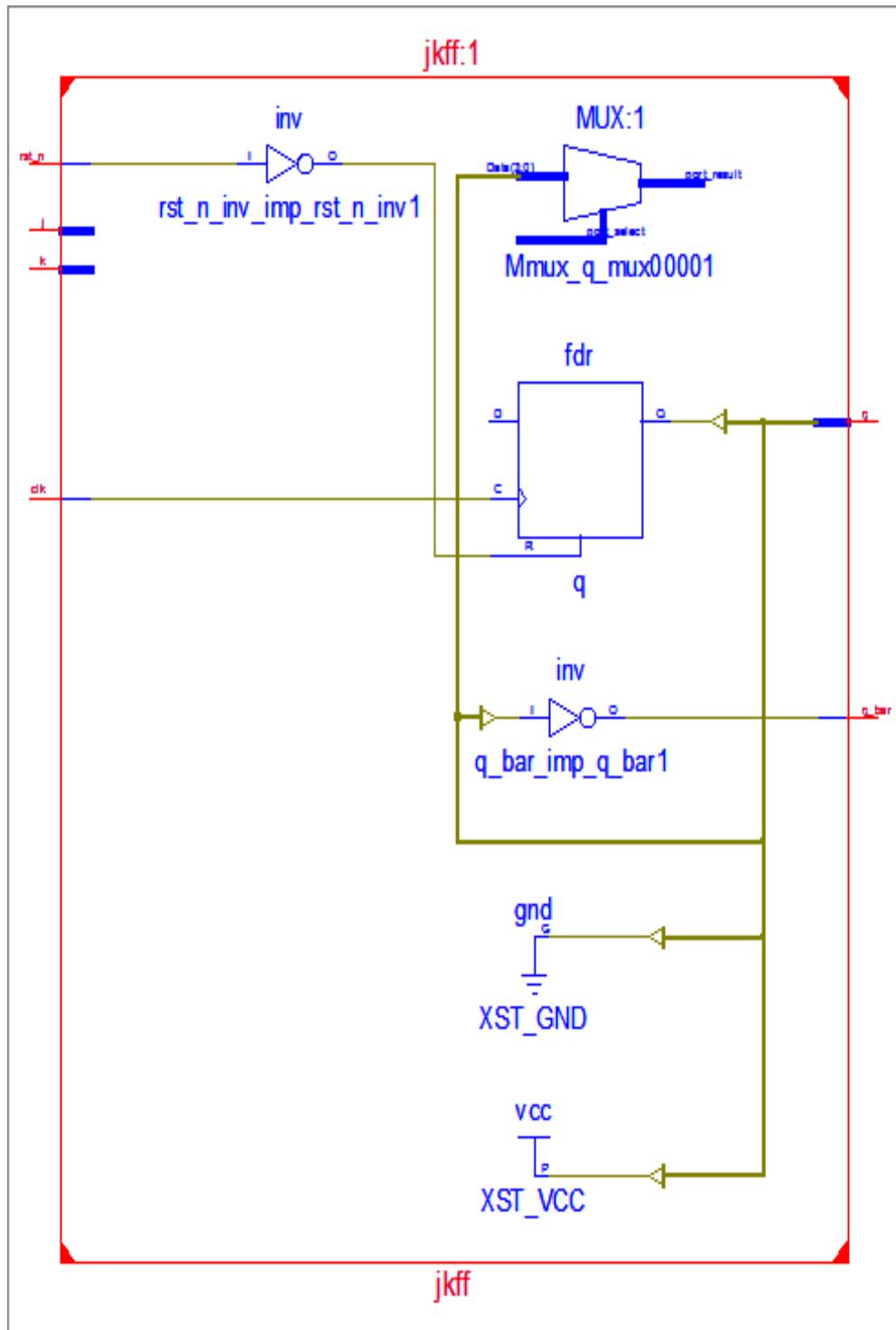
```
module tflip(input t,q,clk, output reg q1);  
always @ (posedge clk)  
assign q1=(t&(~q))|((~t)&q);  
endmodule
```

## SR FLIPFLOP

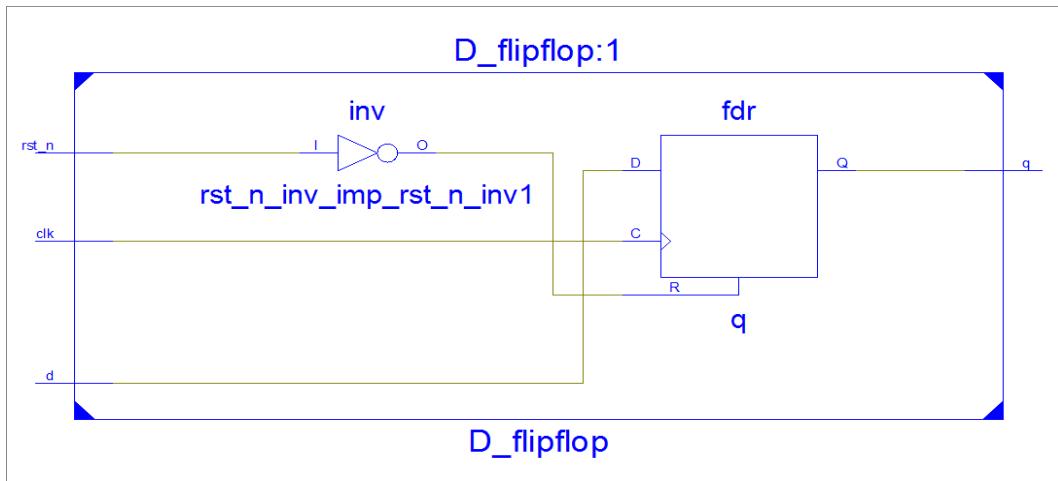
### Program:

```
module SRflipflops(clk,s,r,q);  
input clk,s,r;  
output reg q;  
always@ (posedge clk)  
begin  
case({s,r})  
2'b00:  
q<=q;  
2'b01:  
q<=0;  
2'b10:  
q<=1;  
2'b11:  
q<=1'bx;  
default:  
q<=q;  
endcase  
end  
endmodule
```

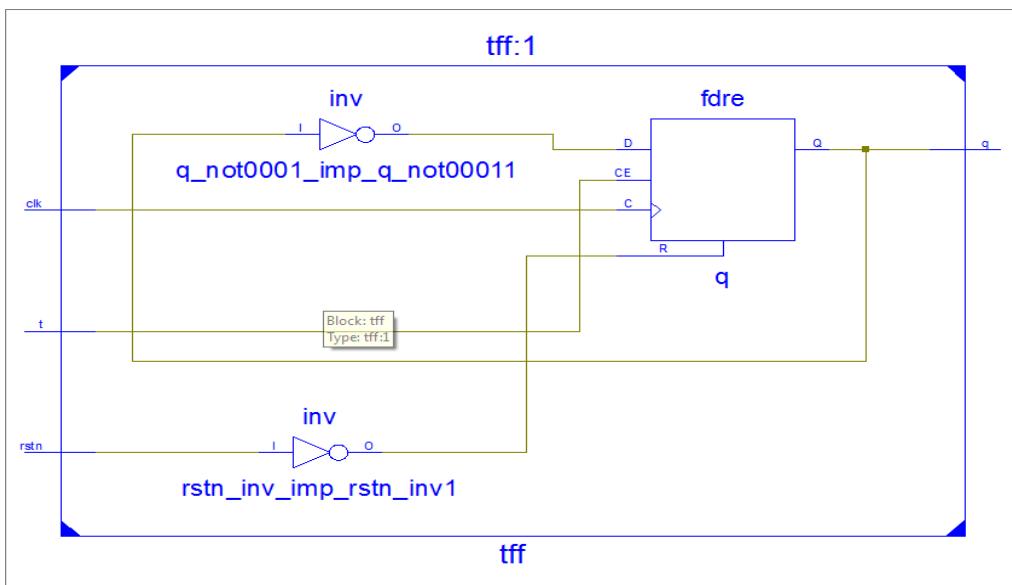
## JK FLIPFLOP RTL SCHEMATIC:



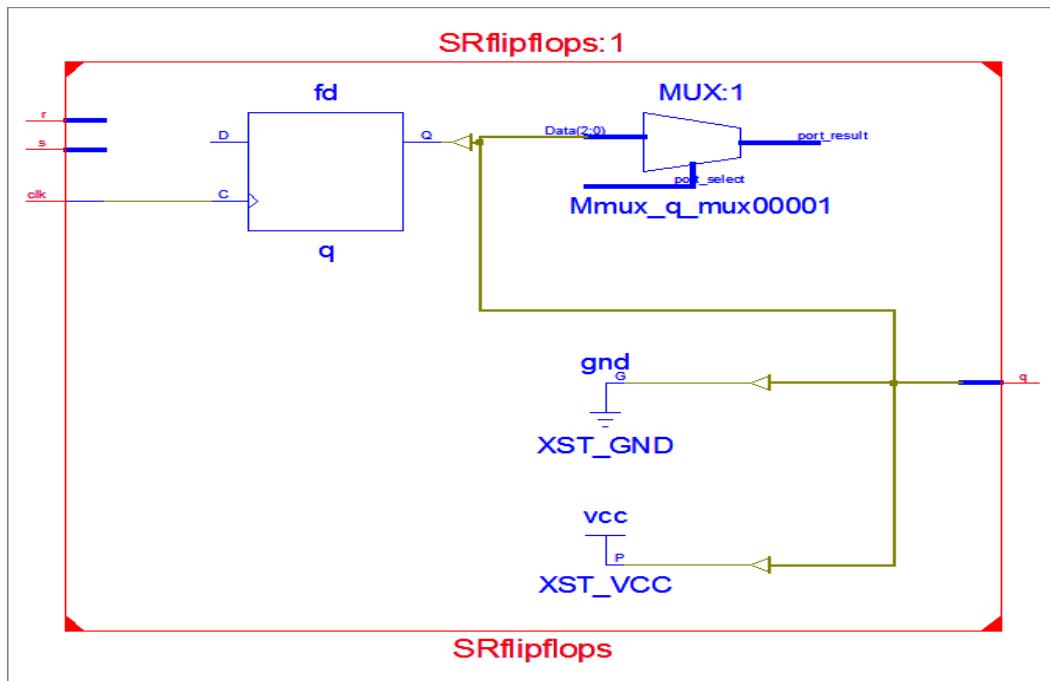
## D FLIPFLOP RTL SCHEMATIC:



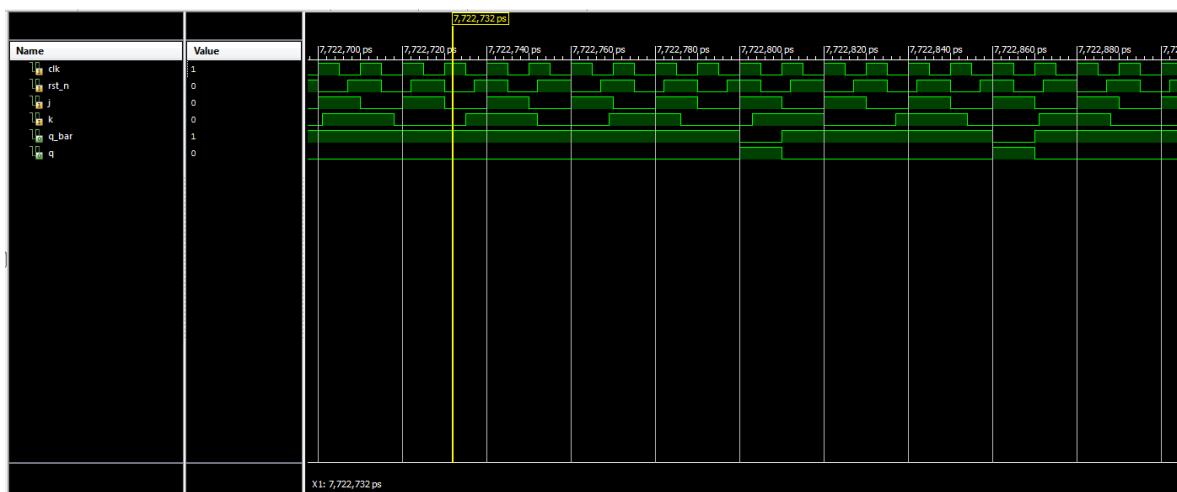
## T FLIPFLOP RTL SCHEMATIC:



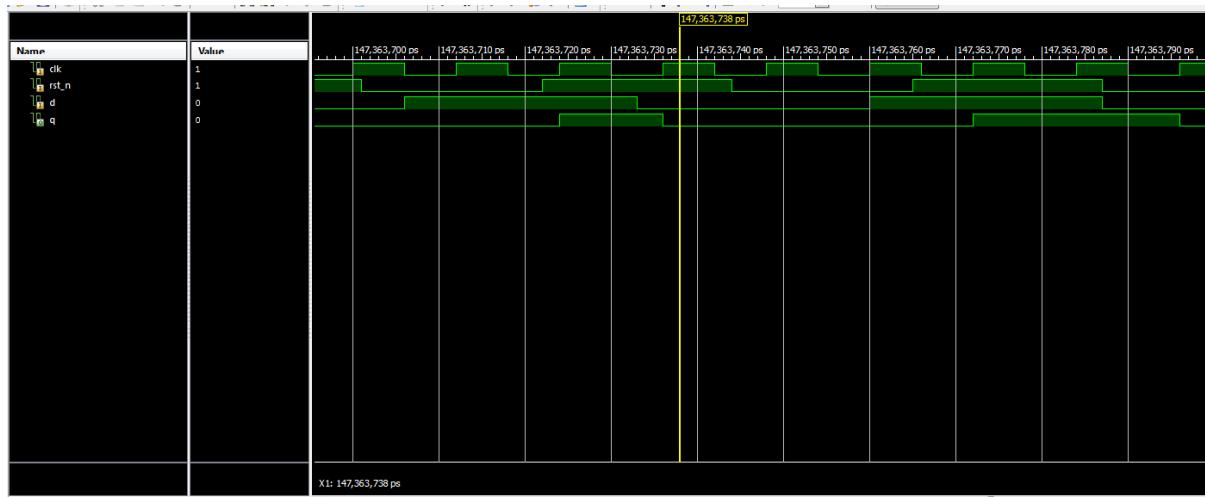
## SR FLIPFLOP RTL SCHEMATIC:



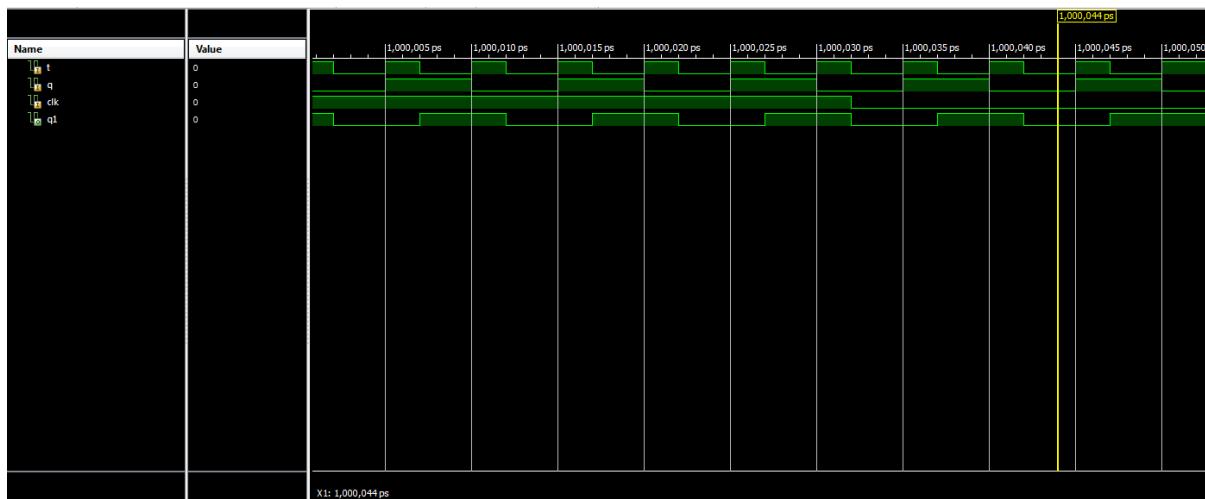
## JK FLIPFLOP OUTPUT:



#### D FLIPFLOP OUTPUT:



#### T FLIPFLOP OUTPUT:



#### SR FLIPFLOP OUTPUT:



**RESULT:**

Hence the sequential (Flip Flops) circuits was implemented using Verilog.

<b>Ex No:</b>	<b>Design an Adder (min 8 bit) using HDL. Simulate it using Xilinx/Altera Software and implement by Xilinx/Altera FPGA</b>
<b>Date:</b>	

### **AIM:**

To design and implement Min 8 bit Adder the using Verilog.

### **SOFTWARE AND HARDWARE REQUIREMENTS:**

Simulation tool : Xilinx 8.1

FPGA Kit: Universal Development Board (UDB) Personal Computer (PC)

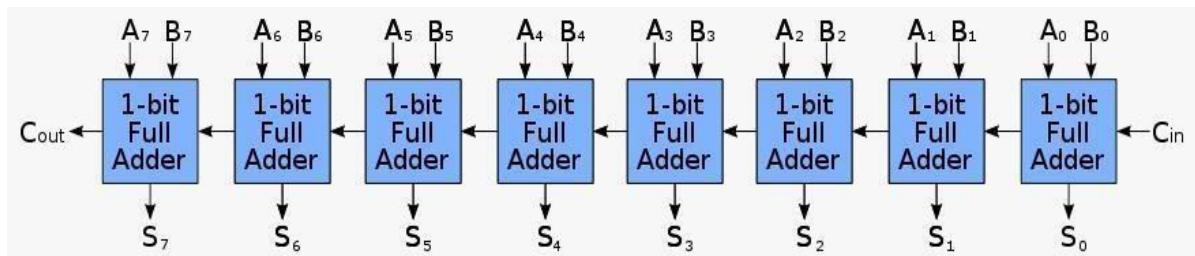
### **THEORY:**

The adder component is an 8-bit ripple carry adder; real ALUS would normally feature a 'carry-look-ahead adder', allowing for high- speed operation. However for this example the much simpler ripple carry adder is adequate, as the operation is totally manual. The adder component is illustrated in Fig. 5.8.5 and consists of eight full-adder circuits with additional logic consisting of an XOR gate to detect overflow errors, and an 8-input NOR gate to detect a zero result. Negative result are indicated by sampling the most significant bit of the 'sum' output, and a 'carry' is indicated by sampling the carry output of the most significant full adder. Four D type flip-flop are used as 'flag' outputs to indicate the current state of the ALU after each operation.

### **PROCEDURE:**

- Create a new project in Xilinx ISE project navigator.
- Open project->new
- Create a Verilog module.
- Type the file name->save.
- Open the program.
- Select the simulator under “source for” in the design window.
- Compile the design and check the syntax under behavior simulate behavioral model.
- Run the simulation.
- Output graph will be displayed.

## CIRCUIT DIAGRAM:



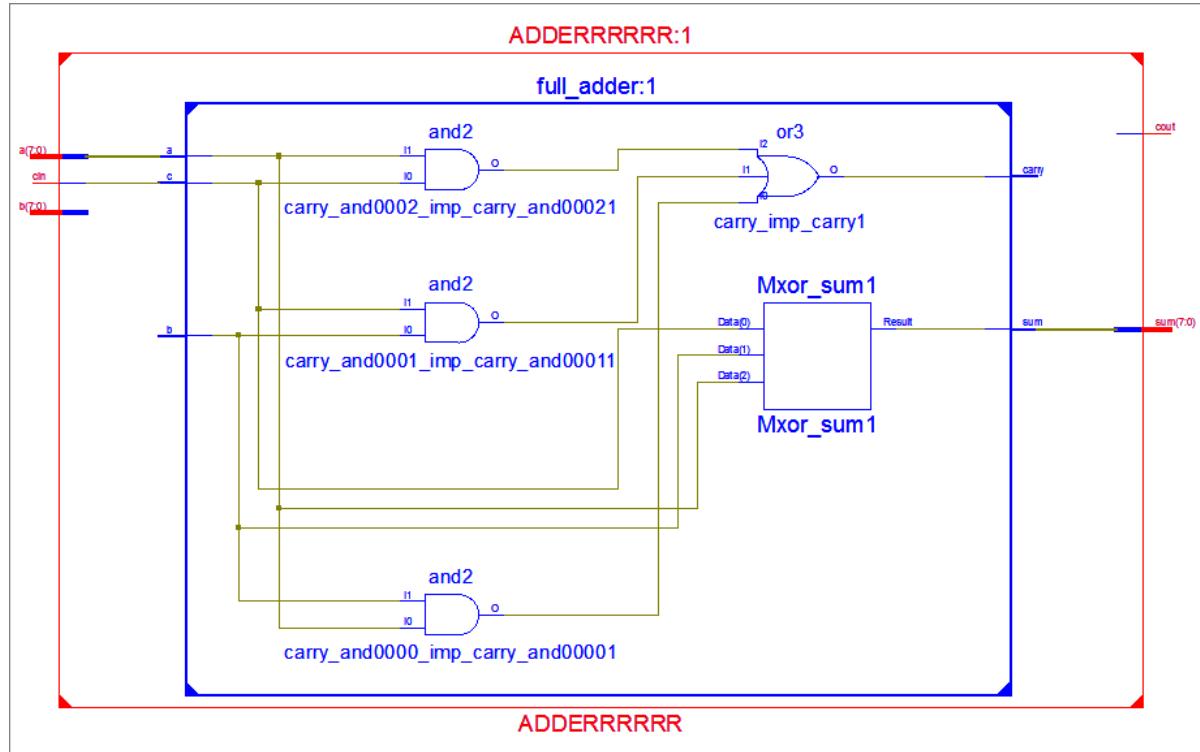
## PROGRAM:

```

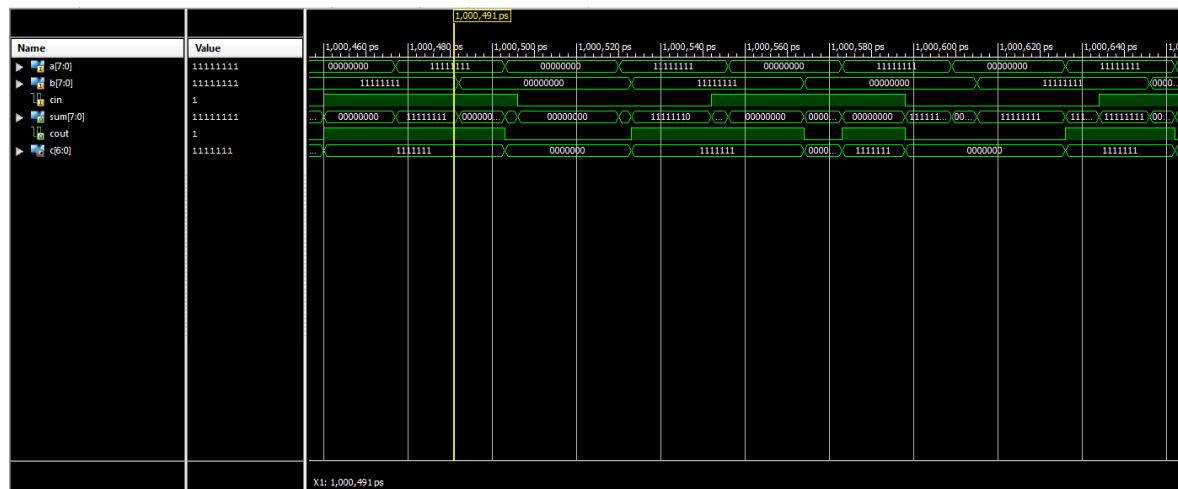
module ADDERRRRRR(a,b,cin,sum,cout);
input [7:0]a,b;
input cin;
output [7:0]sum;
output cout;
wire [6:0]c;
full_adder a1(a[0],b[1],cin,sum[0],c[0]);
full_adder a2(a[1],b[1],c[0],sum[1],c[1]);
full_adder a3(a[2],b[2],c[1],sum[2],c[2]);
full_adder a4(a[3],b[3],c[2],sum[3],c[3]);
full_adder a5(a[4],b[4],c[3],sum[4],c[4]);
full_adder a6(a[5],b[5],c[4],sum[5],c[5]);
full_adder a7(a[6],b[6],c[5],sum[6],c[6]);
full_adder a8(a[7],b[7],c[6],sum[7],cout);
endmodule
module half_adder(a,b,sum,carry);
input a,b;
output sum,carry;
assign sum=a^b;
assign carry=a&b;
endmodule
module full_adder(a,b,c,sum,carry);
input a,b,c;
output sum,carry;
assign sum=a^b^c;
assign carry=(a&b)|(b&c)|(c&a);
endmodule

```

## RTL SCHEMATIC:



## OUTPUT:



**RESULT:**

Hence the Min 8 bit Adder was implemented using Verilog.

<b>Ex No:</b>	<b>Design an Multiplier (min 8 bit) using HDL. Simulate it using Xilinx/Altera Software and implement by Xilinx/Altera FPGA</b>
<b>Date:</b>	

### **AIM:**

To design and implement Min 8 bit Multiplier the using Verilog.

### **SOFTWARE AND HARDWARE REQUIREMENTS:**

Simulation tool : Xilinx 8.1

FPGA Kit: Universal Development Board (UDB) Personal Computer (PC)

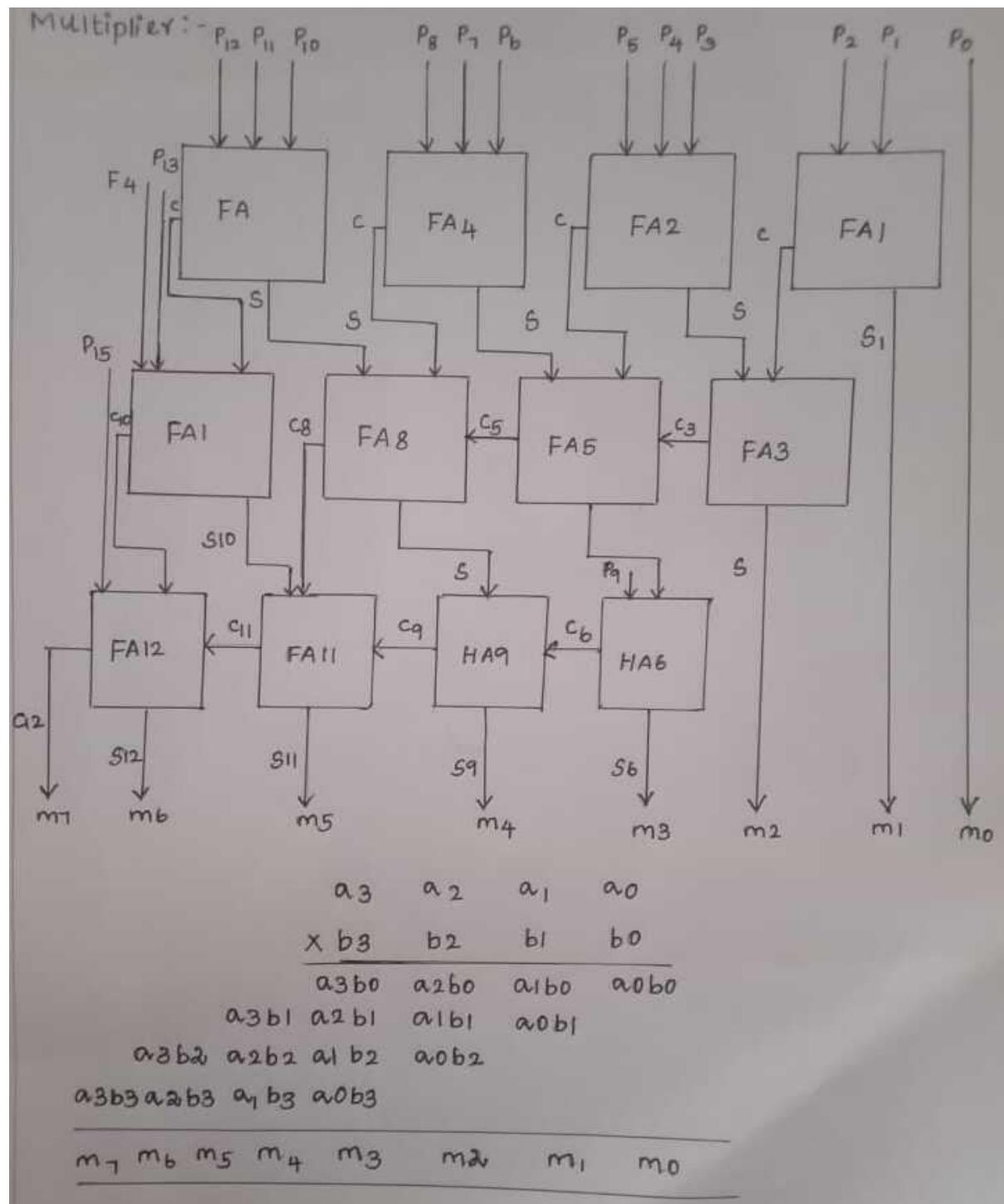
### **THEORY:**

A multiplier is simply a factor that amplifies or increase the base value of something else. A multiplier of 2x, for instance, would double the base figure. A multiplier of 0.5x, on the other hand, would actually reduce the base figure by half. Many different multipliers exist in finance and economics.

### **PROCEDURE:**

- Create a new project in Xilinx ISE project navigator.
- Open project->new
- Create a Verilog module.
- Type the file name->save.
- Open the program.
- Select the simulator under “source for” in the design window.
- Compile the design and check the syntax under behavior simulate behavioral model.
- Run the simulation.
- Output graph will be displayed.

### CIRCUIT DIAGRAM:

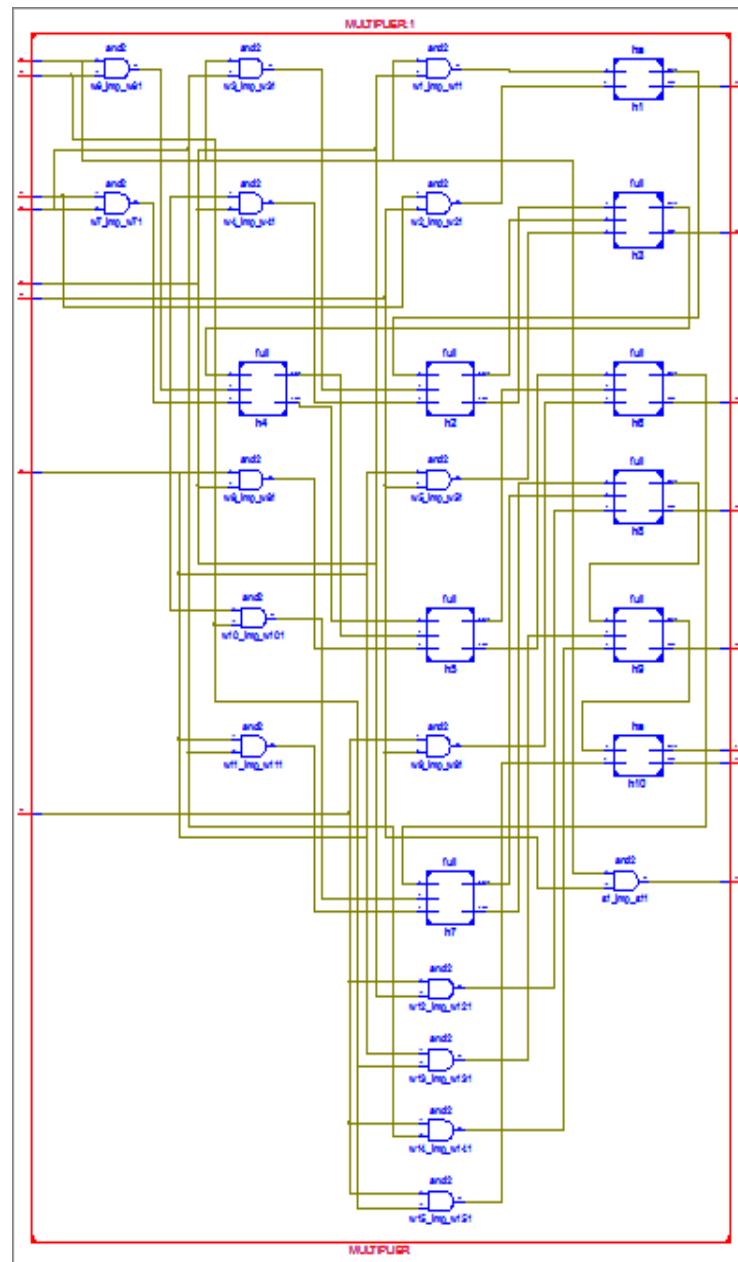


**PROGRAM:**

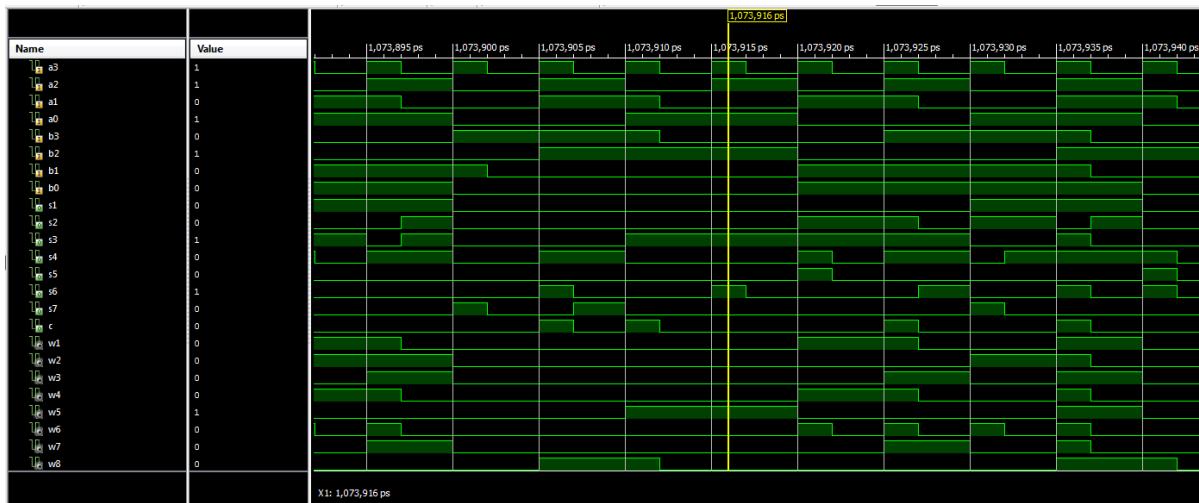
```
module MULTIPLIER(  
    input a3,  
    input a2,  
    input a1,  
    input a0,  
    input b3,  
    input b2,  
    input b1,  
    input b0,  
    output s1,  
    output s2,  
    output s3,  
    output s4,  
    output s5,  
    output s6,  
    output s7,  
    output c  
);  
wire  
w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,w15,u1,u2,u3,u4,u5,u6,u7,u8  
;  
and(s1,a0,b0),(w1,a1,b0),(w2,a0,b1),(w3,a2,b0),(w4,a1,b1),(w5,a0,b2),(w6,a3,b0),(w  
7,  
a2,b1),(w8,a1,b2),(w9,a0,b3),(w10,a3,b1),(w11,a2,b2),(w12,a1,b3),(w13,a3,b2),(w14  
,a2,b3),(w15,a3,b3);  
ha h1(w1,w2,s2,c1);  
full h2(c1,w3,w4,u1,u2);  
full h3(u1,u2,w5,s3,c2);  
full h4(c2,w6,w7,u3,u4);  
full h5(u3,u4,w8,u5,u6);  
full h6(u5,u6,w9,s4,c3);  
full h7(c3,w10,w11,u7,u8);  
full h8(u7,u8,w12,s5,c4);  
full h9(c4,w13,w14,s6,c5);  
ha h10(c5,w15,s7,c);  
endmodule  
module full(x,y,z,sum,carry);  
input x,y,z;
```

```
output sum,carry;  
assign sum=x^y^z;  
assign carry=(x&y)|(y&z)|(z&x);  
endmodule  
module ha(a,b,sum,carry);  
input a,b;  
output sum,carry;  
assign sum=a^b;  
assign carry=(a&b);  
endmodule
```

## RTL SCHEMATIC:



## OUTPUT:



## RESULT:

Hence the Min 8 bit Multiplier was implemented using Verilog

<b>Ex No:</b>	<b>Design and implement Universal Shift Register Using HDL. Simulate it using Xilinx/Altera Software</b>
<b>Date:</b>	

### **AIM:**

To design and implement Universal Shift Register using Verilog.

### **SOFTWARE AND HARDWARE REQUIREMENTS:**

Simulation tool : Xilinx 8.1

FPGA Kit: Universal Development Board (UDB) Personal Computer (PC)

### **THEORY:**

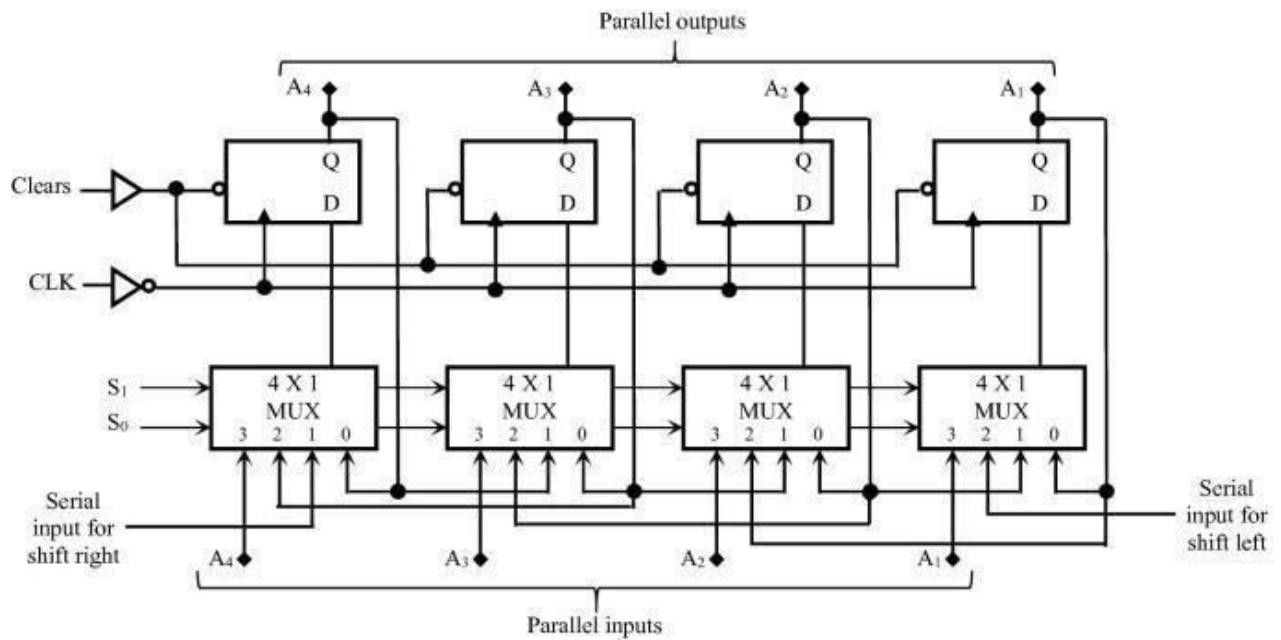
#### **Universal Shift Register:**

A register that can store the data and /shifts the data towards the right and left along with the parallel load capability is known as a universal shift register. It can be used to perform input/output operations in both serial and parallel modes. Unidirectional shift registers and bidirectional shift registers are combined together to get the design of the universal shift register. It is also known as a parallel-in-parallel-out shift register or shift register with the parallel load. Universal shift registers are capable of performing 3 operations as listed below. Parallel load operation – stores the data in parallel as well as the data in parallel Shift left operation – stores the data and transfers the data shifting towards left in the serial path Shift right operation – stores the data and transfers the data by shifting towards right in the serial path. Hence, Universal shift registers can perform input/output operations with both serial and parallel loads..

### **PROCEDURE:**

- Create a new project in Xilinx ISE project navigator.
- Open project->new
- Create a Verilog module.
- Type the file name->save.
- Open the program.
- Select the simulator under “source for” in the design window.
- Compile the design and check the syntax under behavior simulate behavioral model.
- Run the simulation.
- Output graph will be displayed.

## CIRCUIT DIAGRAM:



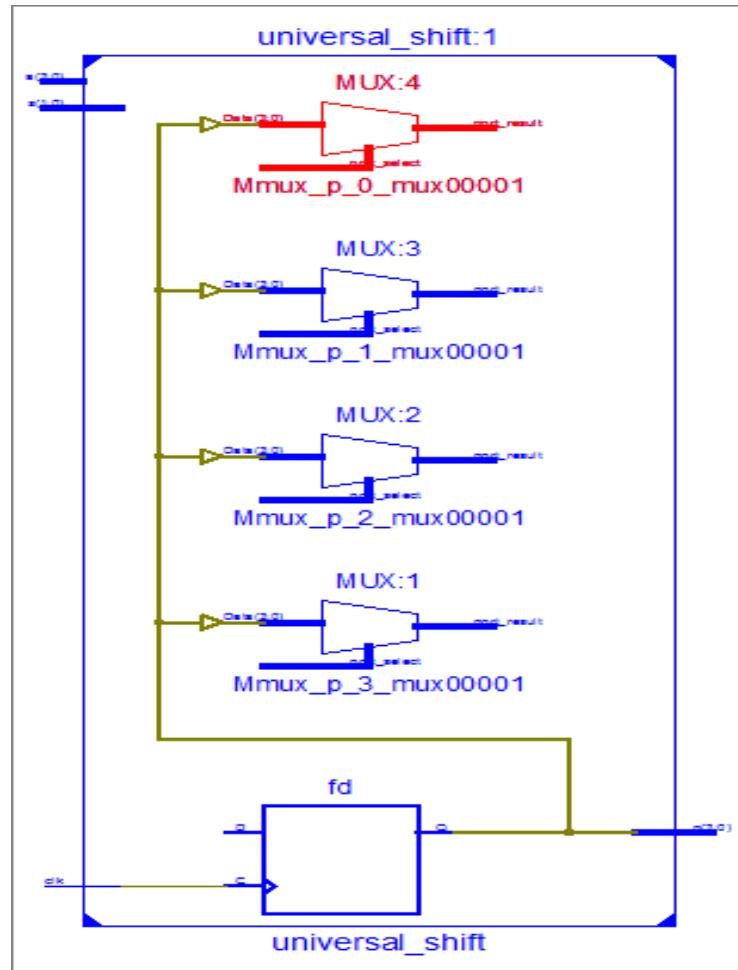
## TRUTH TABLE:

S <sub>1</sub>	S <sub>0</sub>	Register operation
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

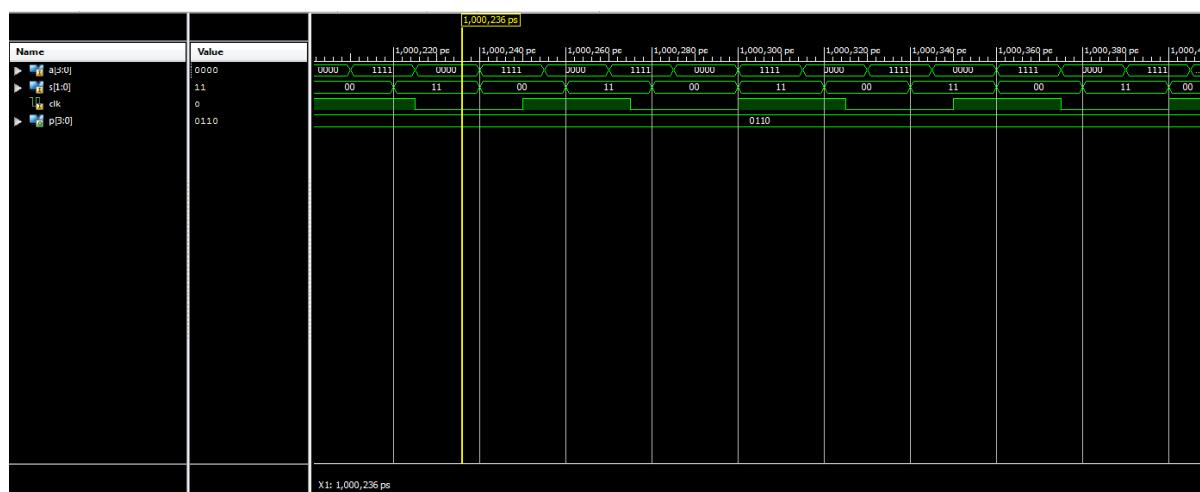
**PROGRAM:**

```
module universal_shift(a,s,clk,p);
input [3:0]a;
input [1:0]s;
input clk;
output reg [3:0]p;
initial p<=4'b0110;
always@ (posedge clk)
begin
case(s)
2'b00:
begin
p[3]<=p[3];
p[2]<=p[2];
p[1]<=p[1];
p[0]<=p[0];
end
2'b01:
begin
p[3]<=p[0];
p[2]<=p[3];
p[1]<=p[2];
p[0]<=p[1];
end
2'b10:
begin
p[0]<=p[3];
p[1]<=p[0];
p[2]<=p[1];
p[3]<=p[2];
end
2'b11:
begin
p[0]<=p[0];
p[1]<=p[1];
p[2]<=p[2];
p[3]<=p[3];
end endcase
end
endmodule
```

## RTL SCHEMATIC:



## OUTPUT:



**RESULT:**

Hence the Universal Shift Register was implemented using Verilog.

<b>Ex No:</b>	<b>Design Memories using HDL. Simulate it using Xilinx/Altera Software and implement by Xilinx/Altera FPGA</b>
<b>Date:</b>	

### **AIM:**

To design and implement Memories using Verilog.

### **SOFTWARE AND HARDWARE REQUIREMENTS:**

Simulation tool : Xilinx 8.1

FPGA Kit: Universal Development Board (UDB) Personal Computer (PC)

### **THEORY:**

#### **MEMORY:**

A memory is neither a sequential circuit (since we require sequential circuits to be clocked, and memories are not clocked), nor a combinatorial circuit, since its output values depend on past values. In general, a memory has  $m$  inputs that are called the *address inputs* that are used to select exactly one out of  $2m$  words, each one consisting of  $n$  bits. Furthermore, it has  $n$  connectors that are *bidirectional* that are called the *data lines*. These data lines are used both as *inputs* in order to store information in a word selected by the address inputs, and as *outputs* in order to recall a previously stored value. Such a solution reduces the number of required connectors by a factor two. Finally, it has an input called *enable* (see the section on tri-state logic for an explanation) that controls whether the data lines have defined states or not, and an input called *r/w* that determines the direction of the data lines.

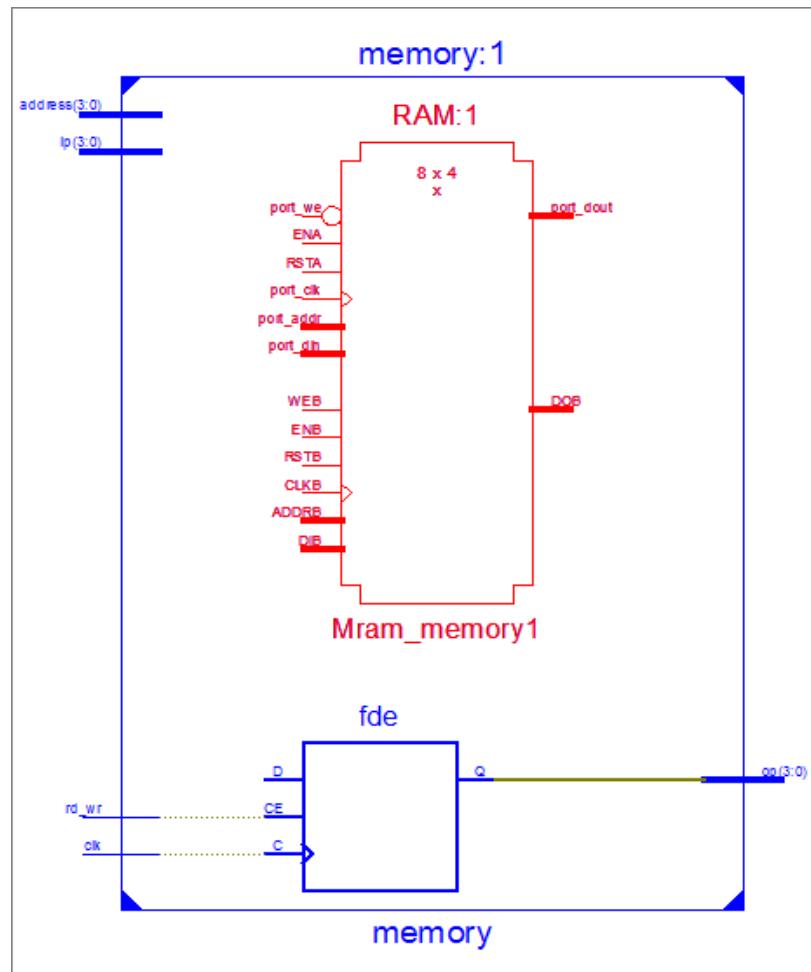
### **PROCEDURE:**

- Create a new project in Xilinx ISE project navigator.
- Open project->new
- Create a Verilog module.
- Type the file name->save.
- Open the program.
- Select the simulator under “source for” in the design window.
- Compile the design and check the syntax under behavior simulate behavioral model.
- Run the simulation.
- Output graph will be displayed.

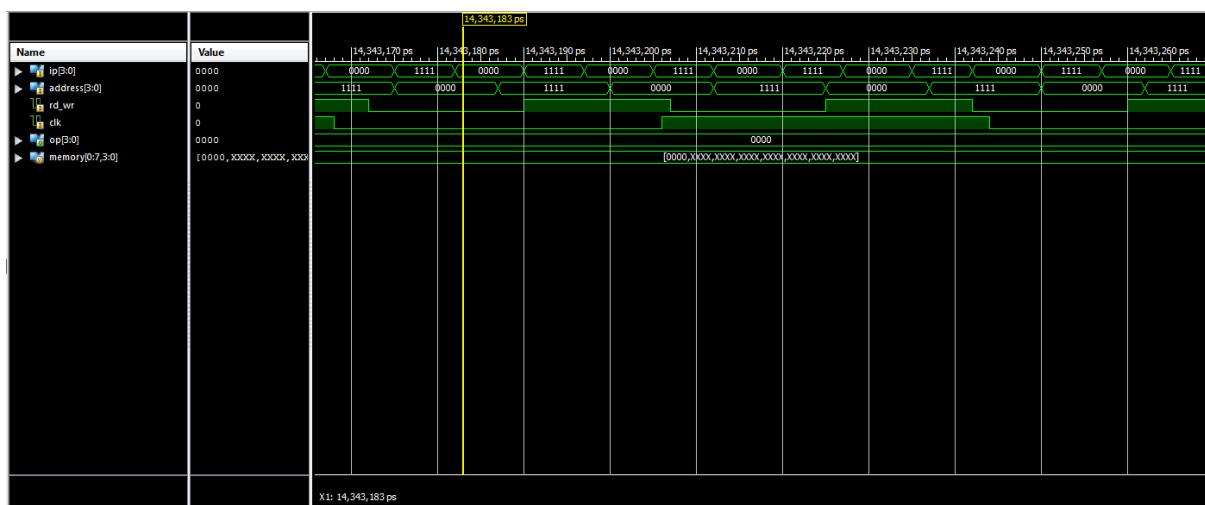
## PROGRAM:

```
module memory(op,ip,rd_wr,clk,address);
output reg [3:0]op;
input [3:0]ip;
input [3:0]address;
input rd_wr,clk;
reg [3:0] memory [0:7];
always@ (posedge clk)
begin
if(rd_wr)
op=memory[address];
else
begin
memory[address]=ip;
end
end
endmodule
```

## RTL SCHEMATIC:



## OUTPUT:



## RESULT:

Hence the Memories was implemented using Verilog.

<b>Ex No:</b>	<b>Design Finite State Machine (Moore/Mealy) using HDL. Simulate it using Xilinx/Altera Software and implement by Xilinx/Altera FPGA</b>
<b>Date:</b>	

### **AIM:**

To design and implement Finite State Machine (Moore/Mealy) using Verilog.

### **SOFTWARE AND HARDWARE REQUIREMENTS:**

Simulation tool : Xilinx 8.1

FPGA Kit: Universal Development Board (UDB) Personal Computer (PC)

### **THEORY:**

#### **STATE MACHINE**

A finite-state machine (FSM) or finite-state automaton (FSA, plural: automata), finite automaton, or simply a state machine, is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time.

**Moore Machines:** Moore machines are finite state machines with output value and its output depends only on present state.

In the Moore machine, the output is represented with each input state separated by / (slash). The length of output for a Moore machine is greater than input by 1.

**Mealy Machines:** Mealy machines are also finite state machines with output value and its output depends on present state and current input symbol.

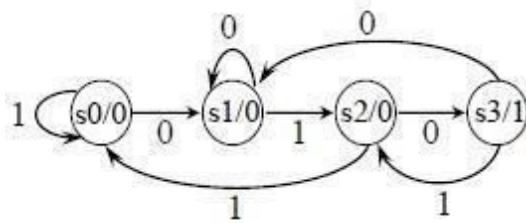
In the mealy machine, the output is represented with each input symbol for each state separated by / (Slash). The length of output for a mealy machine is equal to the length of input.

### **PROCEDURE:**

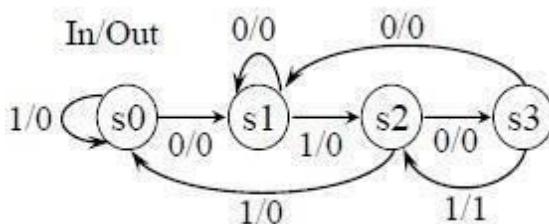
- Create a new project in Xilinx ISE project navigator.
- Open project->new
- Create a Verilog module.
- Type the file name->save.
- Open the program.
- Select the simulator under “source for” in the design window.
- Compile the design and check the syntax under behavior simulate behavioral model.
- Run the simulation.
- Output graph will be displayed.

## CIRCUIT DIAGRAM:

### Moore Machines:



### Mealy Machines:



## MOORE MODEL

### Program:

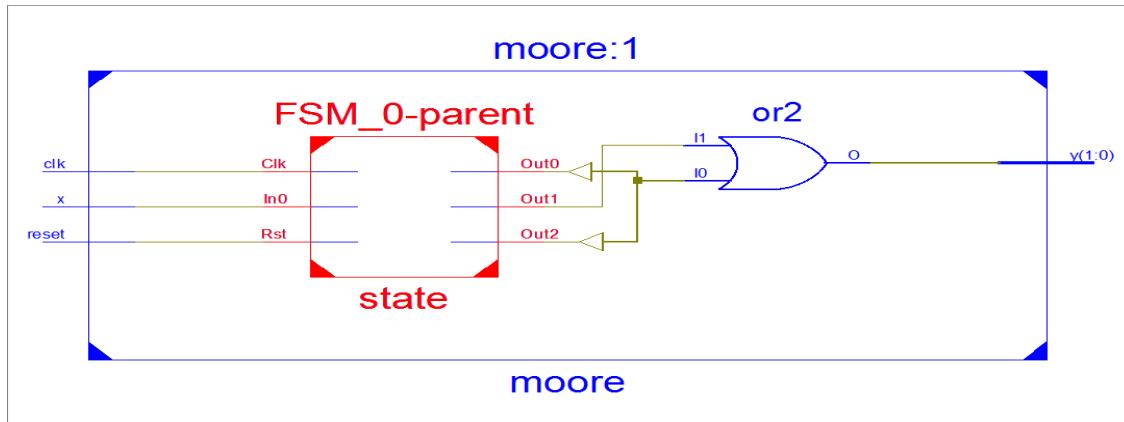
```
module moore(y,x,clk,reset);
input x,clk,reset;
output[1:0]y;
reg[1:0]state;
parameter S0=2'b00,S1=2'b01,S2=2'b10,S3=2'b11;
always@(posedge clk)
if(reset==0)
state<=S0;
else case(state)S0:
if(x)state<=S0;
else state<=S1;
S1:
if(x)state<=S2;
else state<=S3;
S2:
if(x)state<=S2;
else state<=S3;
S3:
if(x)state<=S3;
else state<=S0;
endcase
assign y=state;
endmodule
```

## MEALY MODEL

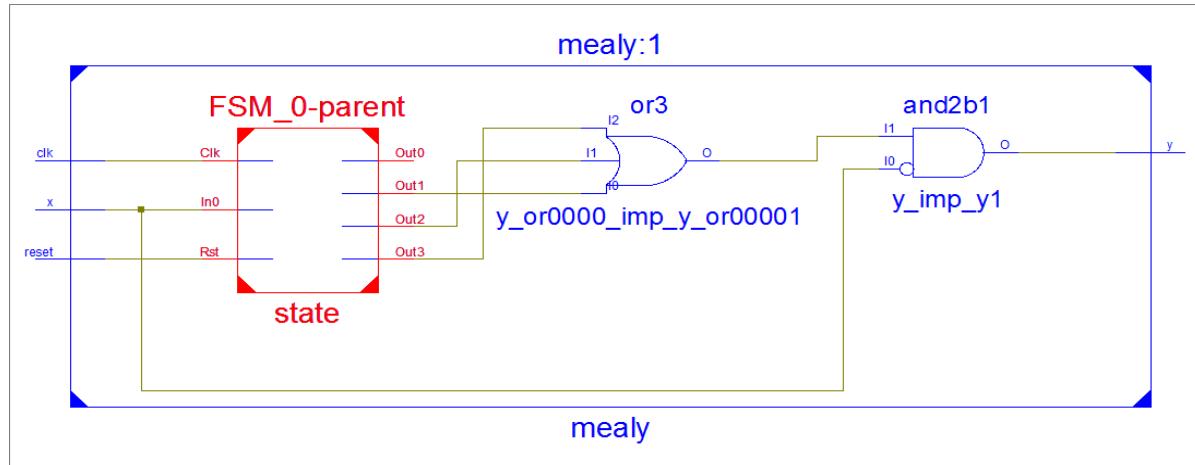
### Program:

```
module mealy(y,x,clk,reset);
input x,clk,reset;
output reg y;
reg [1:0] state,next_state;
parameter S0=2'b00,S1=2'b01,S2=2'b10,S3=2'b11;
always@ (posedge clk)
if(reset==0) state<=S0;
else
state<=next_state;
always@ (state,x)
case (state)
S0:
if(x) next_state=S1;
else
next_state=S0;
S1:
if(x) next_state=S3;
else
next_state=S0;
S2:
if(x) next_state=S2;
else
next_state=S0;
S3:
if(x) next_state=S2;
else
next_state=S0;
V endcase
always@(state,x)
case(state)
S0:y=0;
S1,S2,S3:
y=~x;
endcase
endmodule
```

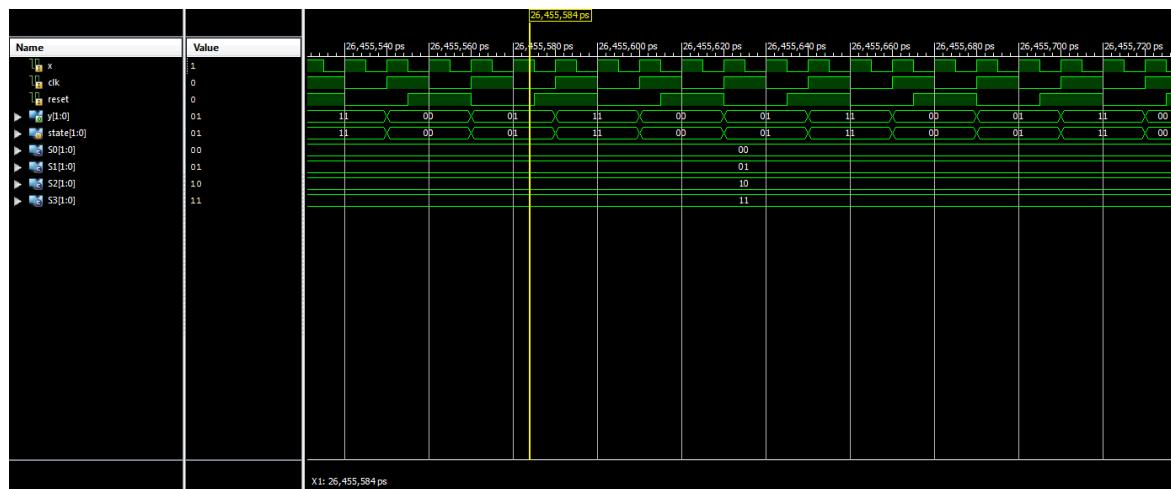
## RTL SCHEMATIC: MOORE MODEL:



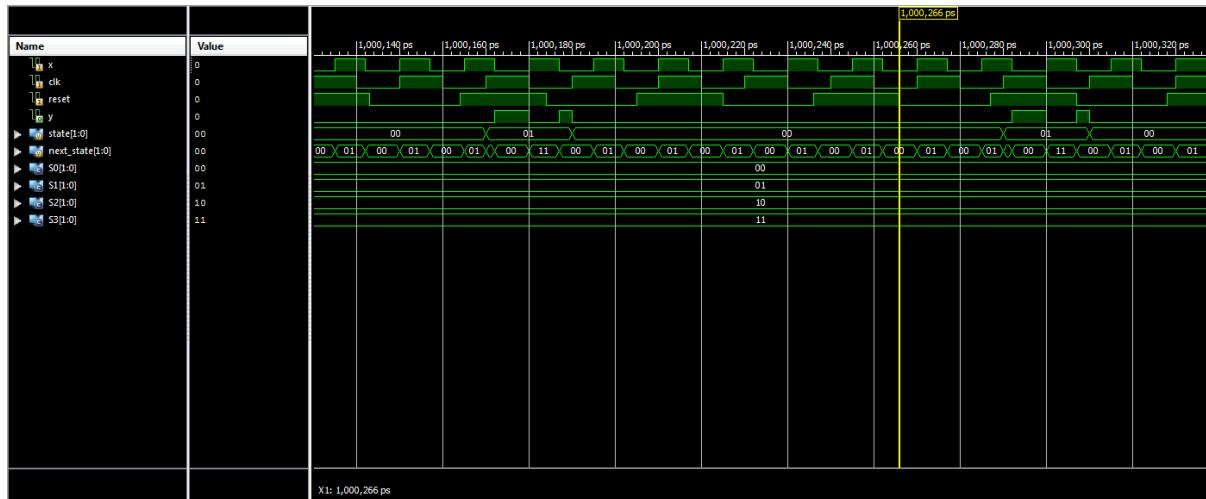
## MEALY MODEL:



## MOORE MODEL OUTPUT:



## MEALY MODEL OUTPUT:



## RESULT:

Hence the Finite State Machine (Moore/Mealy) was implemented using Verilog.

<b>Ex No:</b>	<b>Design 3-bit Synchronous up/down counter using HDL. Simulate it using Xilinx/Altera Software and implement by Xilinx/Altera FPGA</b>
<b>Date:</b>	

### **AIM:**

To design and implement 3-bit Synchronous Up/Down Counter using Verilog.

### **SOFTWARE AND HARDWARE REQUIREMENTS:**

Simulation tool : Xilinx 8.1

FPGA Kit: Universal Development Board (UDB) Personal Computer (PC)

### **THEORY:**

#### **3-bit Synchronous Counter:**

A counter is a device which can count any particular event on the basis of how many times the particular event(s) is occurred. In a digital logic system or computers, this counter can count and store the number of time any particular event or process have occurred, depending on a clock signal. Most common type of counter is sequential digital logic circuit with a single clock input and multiple outputs. The outputs represent binary or binary coded decimal numbers. Each clock pulse either increases the number or decreases the number.

Synchronous generally refers to something which is coordinated with others based on time. Synchronous signals occur at same clock rate and all the clocks follow the same reference clock.

In Asynchronous Counter, we have seen that the output of that counter is directly connected to the input of next subsequent counter and making a chain system, and due to this chain system propagation delay appears during counting stage and create counting delays. In synchronous counter, the clock input across all the flip-flops use the same source and creates the same clock signal at the same time. So, a counter which is using the same clock signal from the same source at the same time is called synchronous counter.

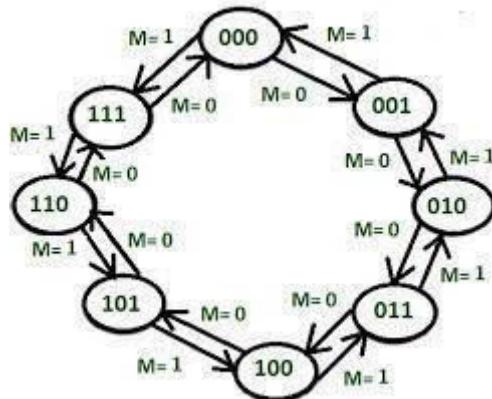
A 3-bit Synchronous up counter start to count from 0 (000 in binary) and increment or count upwards to 7 (111 in binary) and then start new counting cycle by getting reset. Its operating frequency is much higher than the same range Asynchronous counter. Also, there is no propagation delay in the synchronous counter just because all flip-flops or counter stage is in parallel clock source and the clock triggers all counters at the same time.

### **PROCEDURE:**

- Create a new project in Xilinx ISE project navigator.
- Open project->new
- Create a Verilog module.
- Type the file name->save.
- Open the program.

- Select the simulator under “source for” in the design window.
- Compile the design and check the syntax under behavior simulate behavioral model.
- Run the simulation.
- Output graph will be displayed.

### CIRCUIT DIAGRAM:



### PROGRAM:

```

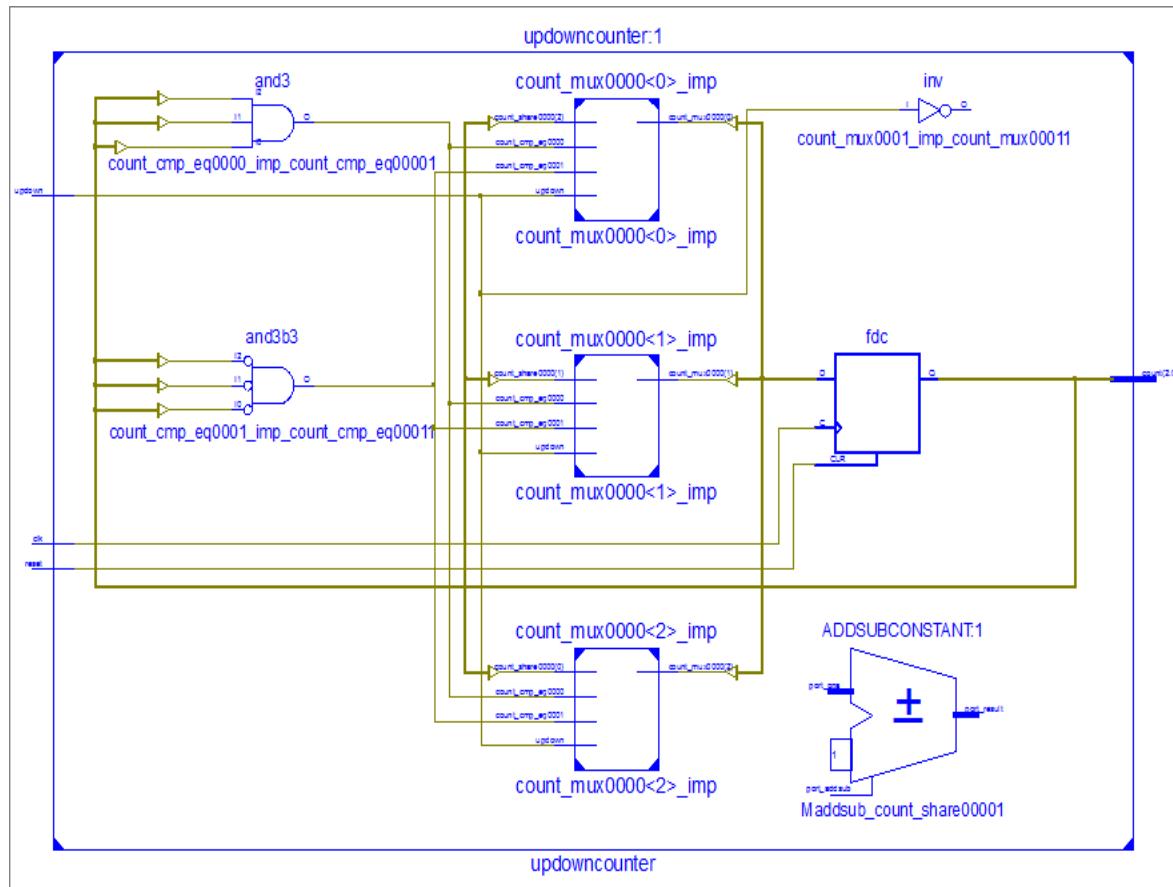
module updowncounter(
clk,
reset,
updown,
count
);
input clk,reset,updown;
output[2:0]count;
reg[2:0]count=0;
always@ (posedge clk or posedge reset)
begin
if(reset==1)
count<=0;
else
if(updown==1)
if(count==7)
count<=0;
else
count<=count+1;
else
if(count==0)
count<=7;
else
  
```

```

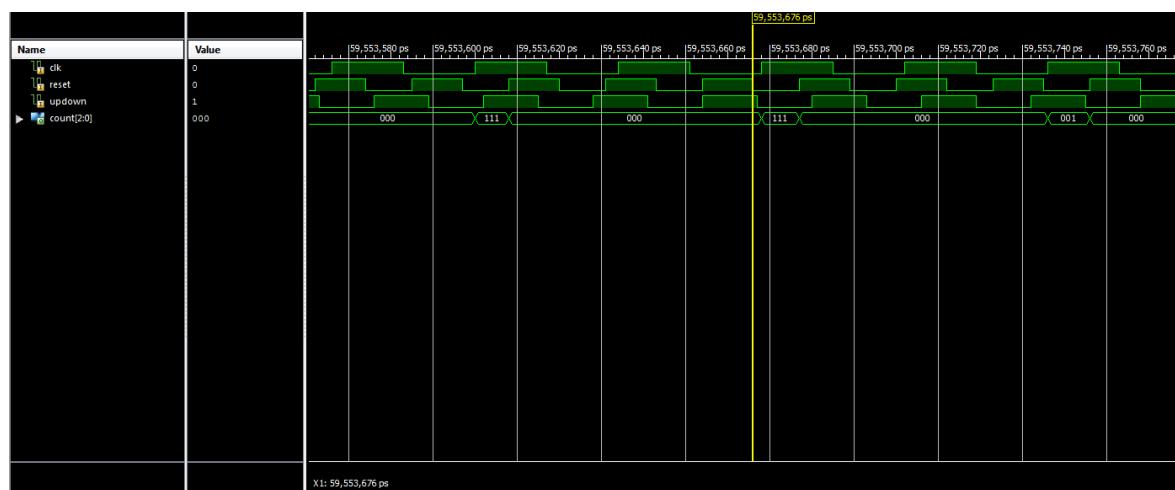
count<=count-1;
end
endmodule

```

## RTL SCHEMATIC:



## OUTPUT:



**RESULT:**

Hence the 3-bit Synchronous Up/Down Counter was implemented using Verilog.

<b>Ex No:</b>	<b>Design 4-bit Asynchronous up/down counter using HDL. Simulate it using Xilinx/Altera Software and implement by Xilinx/Altera FPGA</b>
<b>Date:</b>	

### **AIM:**

To design and implement 4-bit Asynchronous up/down Counter using Verilog.

### **SOFTWARE AND HARDWARE REQUIREMENTS:**

Simulation tool : Xilinx 8.1

FPGA Kit: Universal Development Board (UDB) Personal Computer (PC)

### **THEORY:**

#### **4-bit Synchronous Counter:**

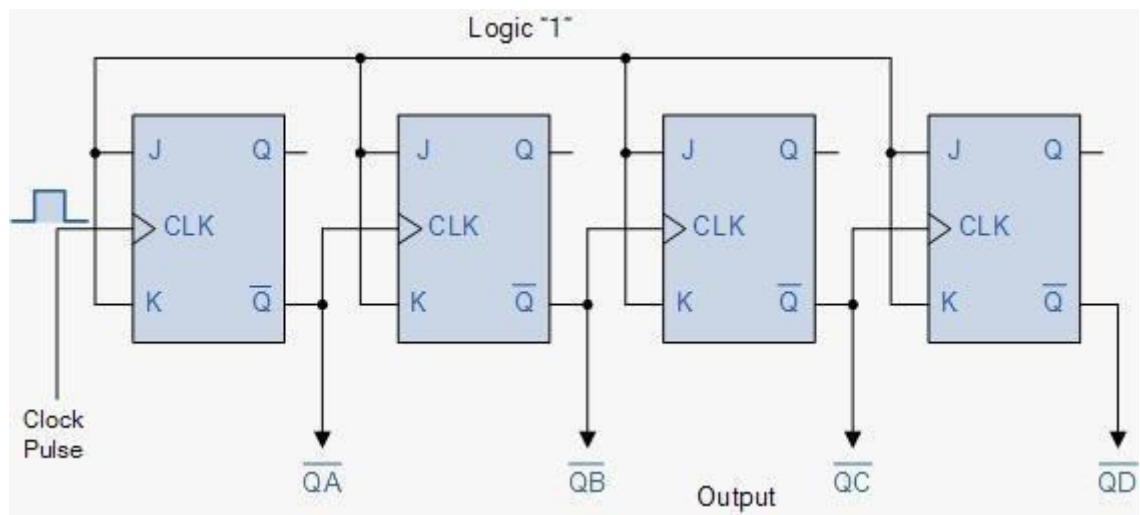
A counter is a device which can count any particular event on the basis of how many times the particular event(s) is occurred. In a digital logic system or computers, this counter can count and store the number of time any particular event or process have occurred, depending on a clock signal. Most common type of counter is sequential digital logic circuit with a single clock input and multiple outputs. The outputs represent binary or binary coded decimal numbers. Each clock pulse either increases the number or decreases the number.

Synchronous generally refers to something which is coordinated with others based on time. Synchronous signals occur at same clock rate and all the clocks follow the same reference clock.

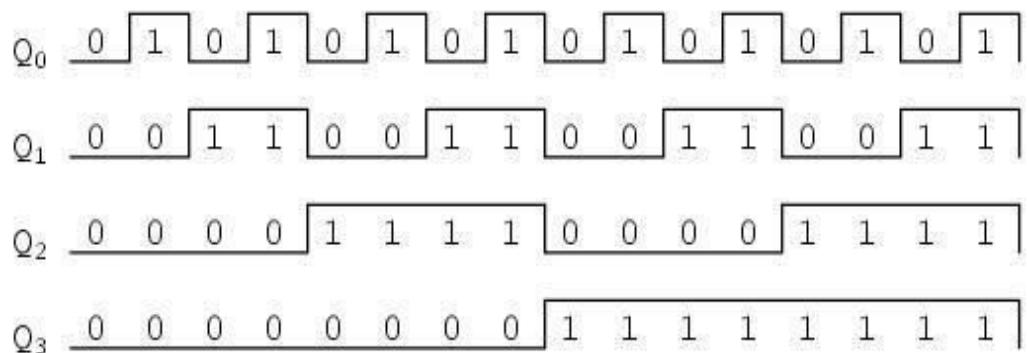
In Asynchronous Counter, we have seen that the output of that counter is directly connected to the input of next subsequent counter and making a chain system, and due to this chain system propagation delay appears during counting stage and create counting delays. In synchronous counter, the clock input across all the flip-flops use the same source and creates the same clock signal at the same time. So, a counter which is using the same clock signal from the same source at the same time is called Synchronous counter.

A 4-bit Synchronous up counter start to count from 0 (0000 in binary) and increment or count upwards to 15 (1111 in binary) and then start new counting cycle by getting reset. Its operating frequency is much higher than the same range Asynchronous counter. Also, there is no propagation delay in the synchronous counter just because all flip-flops or counter stage is in parallel clock source and the clock triggers all counters at the same time.

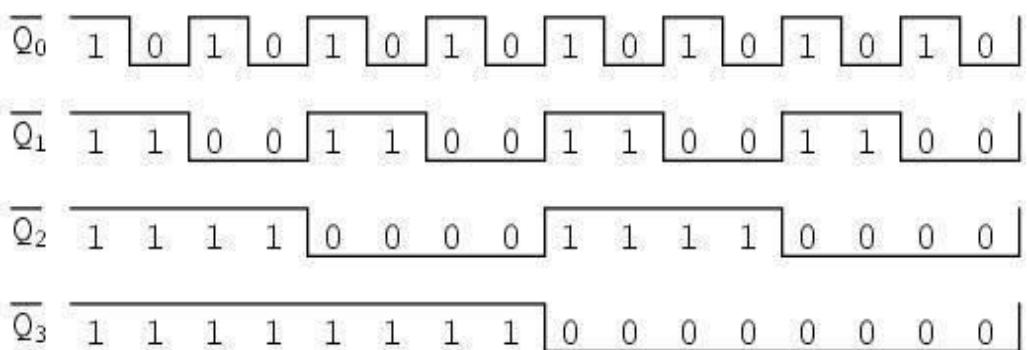
**CIRCUIT DIAGRAM:**



*"Up" count sequence*

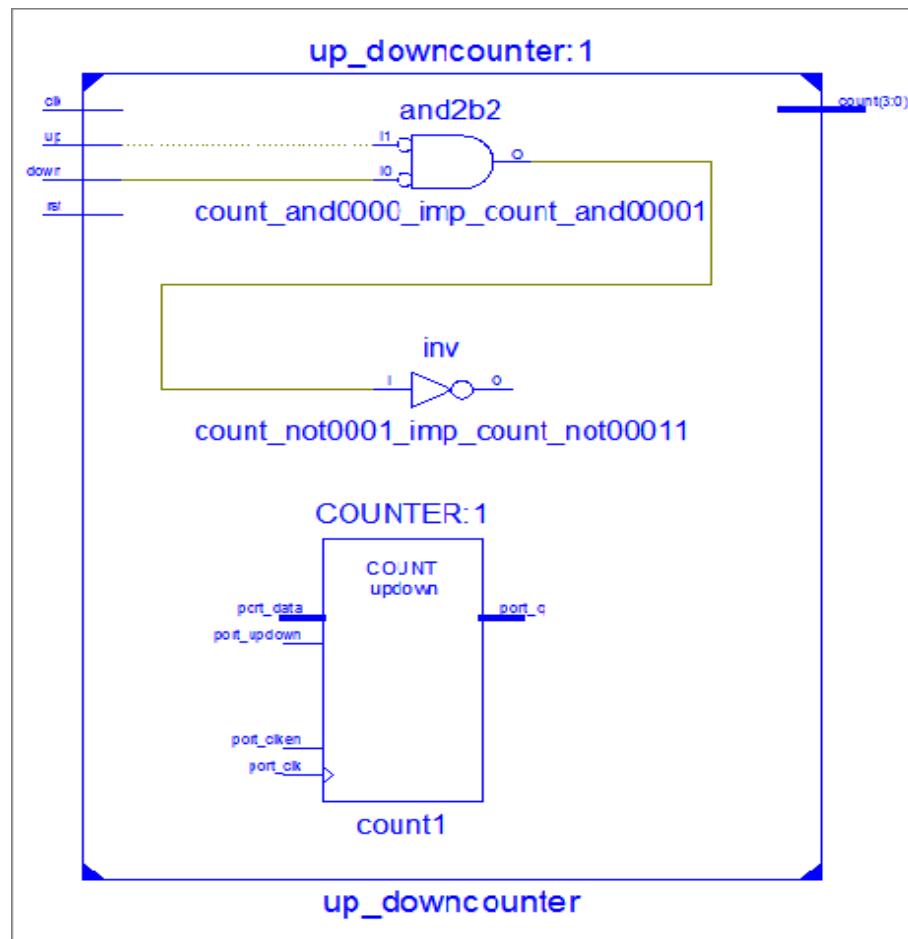


*"Down" count sequence*

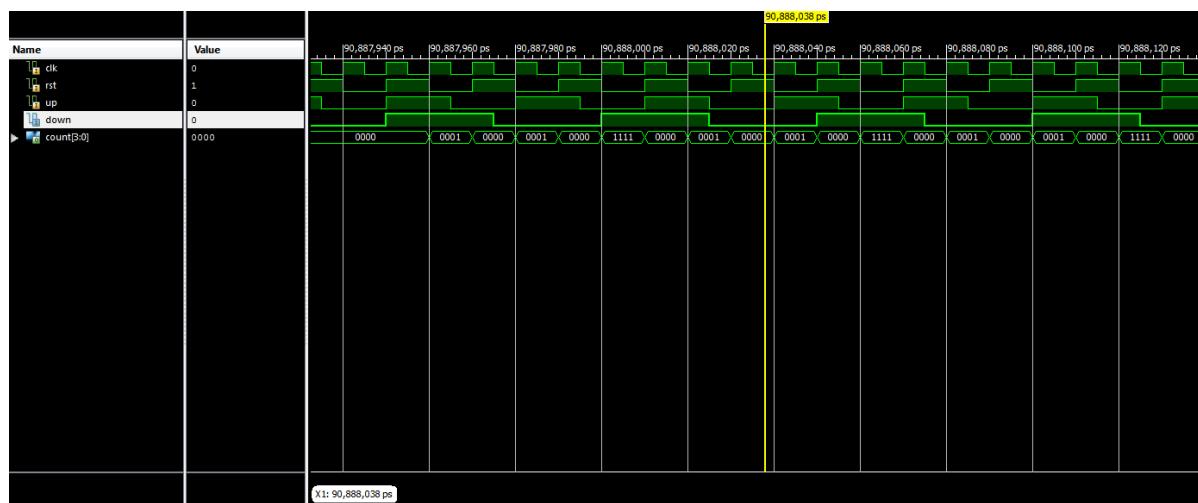


**PROGRAM:**

```
module up_downcounter()
input wire clk,
input wire rst,
input wire up,
input wire down,
output reg [3:0] count);
always@ (posedge clk or posedge rst)begin
if (rst)begin
count<=4'b0000;
end else begin
if(up)begin
count<=count+1;
end else if(down) begin
count<=count-1;
end
end
end
endmodule
```

**RTL SCHEMATIC:**

## OUTPUT:



## RESULT:

Hence the 4-bit Asynchronous up/down Counter was implemented using Verilog.

**Ex No:**

**Design and Simulate a CMOS Basic Gates & Amp, Flip-Flops. Generate Manual/Automatic Layout**

**Date:**

**AIM:**

To design and simulate a cmos basic gates using Cadence application.

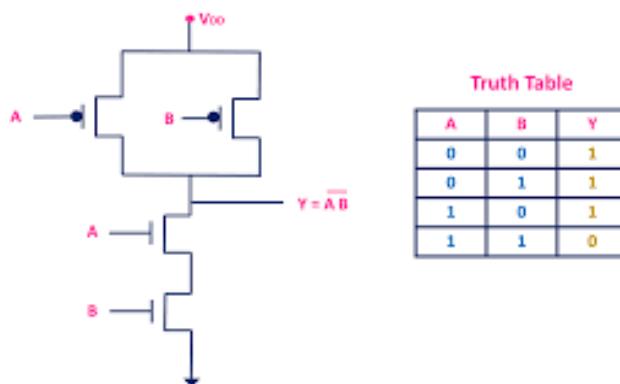
**SOFTWARE REQUIRED:**

- Pc loaded with linux os
- Cadence software

**THEORY:**

**CMOS And Gate:**

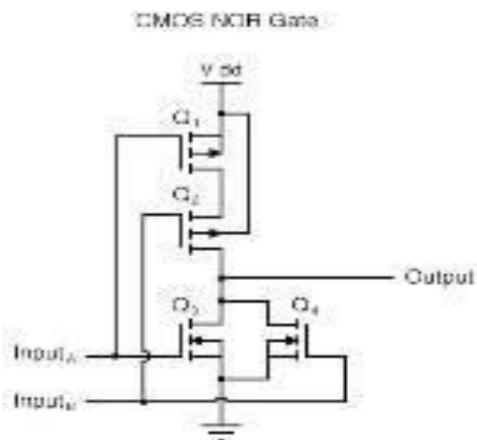
As with the TTL NAND gate, the CMOS NAND gate circuit may be used as the starting point for the creation of an AND gate. All that needs to be added is another stage of transistors to invert the output signal.



**CMOS NOR GATES:**

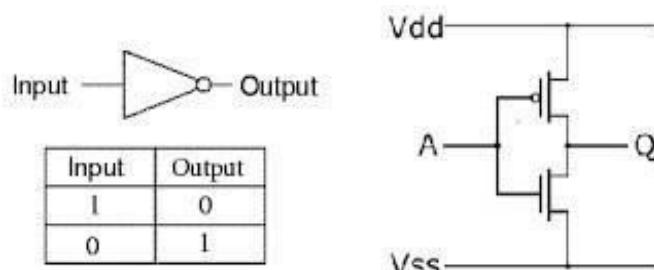
A CMOS NOR gate circuit uses four MOSFETs just like the NAND gate, except that its transistors are differently arranged. Instead of two paralleled sourcing (upper) transistors connected to Vdd and two series connected sinking (lower) transistors connected to ground, the NOR gate uses two series-connected sourcing transistors and two parallel-connected sinking transistors like this. As with the NAND gate, transistors Q1 and Q3 work as a complementary pair, as do transistors Q2 and Q4. Each pair is controlled by a single input signal. If either input A or input B are "high" (1), at least one of the lower transistors (Q3 or Q4) will be saturated, thus making the output "low" (0).

Only in the event of both inputs being "low" (0) will both lower transistors be in cutoff mode and both upper transistors be saturated, the conditions necessary for the output to go "high" (1). This behavior, of course, defines the NOR logic function.



### CMOS INVERTER:

CMOS inverter definition is a device that is used to generate logic functions is known as CMOS inverter and is the essential component in all integrated circuits. A CMOS inverter is a FET (field effect transistor), composed of a metal gate that lies on top of oxygen's insulating layer on top of a semiconductor. These inverters are used in most electronic devices which are accountable for generating data n small circuits.



### PROCEDURE:

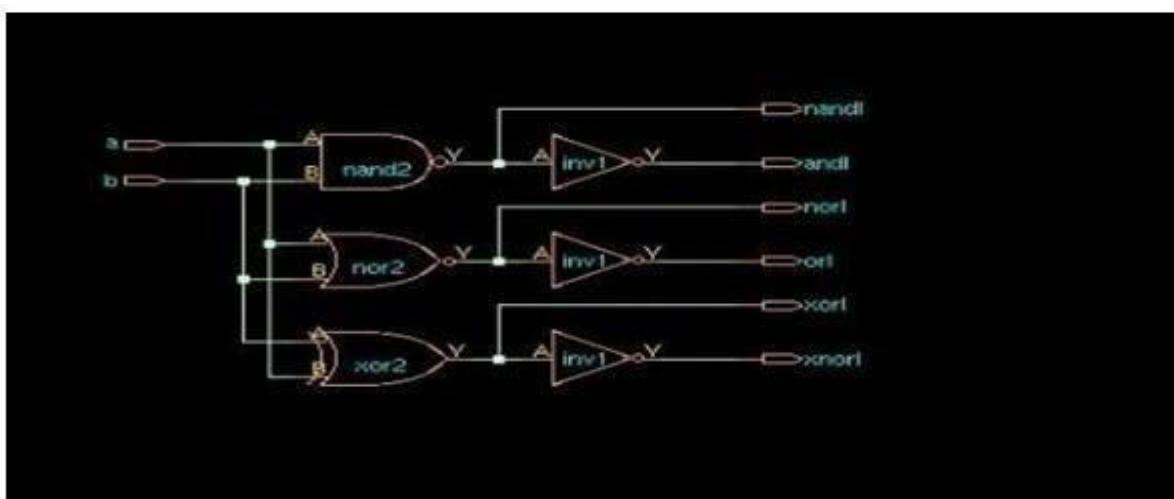
- Open the new terminal window and type the command given below.
- Mount -a
- Cd cadence\_db
- Csh
- Source cshrc
- Cd cadence\_ms\_labs\_614
- Virtuoso
- Now two windows (what's new window and virtuoso window) are open.
- Close the what's new window.
- In the virtuoso window, go to file -> new -> library.
- Now new library window is open.
- Type the file name.
- Activate attach to existing.
- Click ok.

- Now attach library window is open.
- Choose gpdk 180.
- Click ok.
- Again, in the virtuoso window, go to file->new->cell view->cell
- name (type the cell name)->ok.
- The schematic window is open.

### **PROGRAM:**

```
module gates(
input a,b;
output o_and, o_nand, o_or, o_nor, o_not, o_xor, o_xnor
);
assign o_and=a&b;
assign o_nand=--(a&b);
assign o_or=alb;
assign o_nor=(alb);
assign o_not=--a;
assign o_xor=a^b;
assign o_xnor=--(a^b);
endmodule
```

### **SCHEMATIC DIAGRAM:**



## OUTPUT:



## RESULT:

Hence the CMOS basic gates was designed and simulated using Cadence application tool.

<b>Ex No:</b>	<b>Design and Simulate a 4-bit synchronous counter using a Flip-Flops.</b>
<b>Date:</b>	<b>Generate Manual/Automatic Layout</b>

### **AIM:**

To design and simulate a 4-bit synchronous counter using a Flip-Flops using Cadence application.

### **SOFTWARE REQUIRED:**

- Pc loaded with linux os
- Cadence software

### **THEORY:**

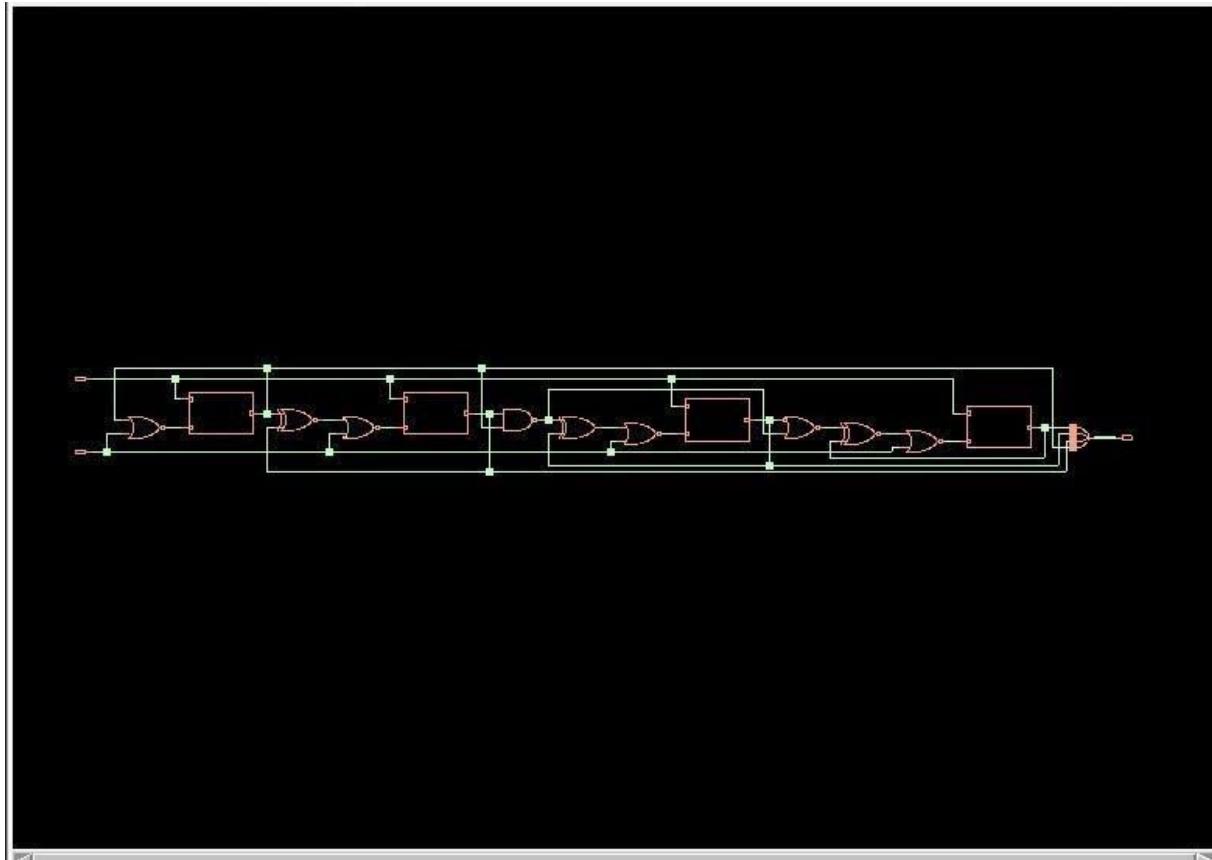
A counter is a sequential circuit that moves through a predefined sequence of states upon applying of clock pulses. The sequence of states may follow the binary number sequence or an arbitrary manner (no sequence). The simplest example of a counter is the binary counter which follows the binary number sequence. An n-bit binary counter contains n flip-flops and can count binary numbers from 0 to  $(2^n - 1)$  (up counter which is incremental, if it counts decremental, it is then called a down counter).

### **PROCEDURE:**

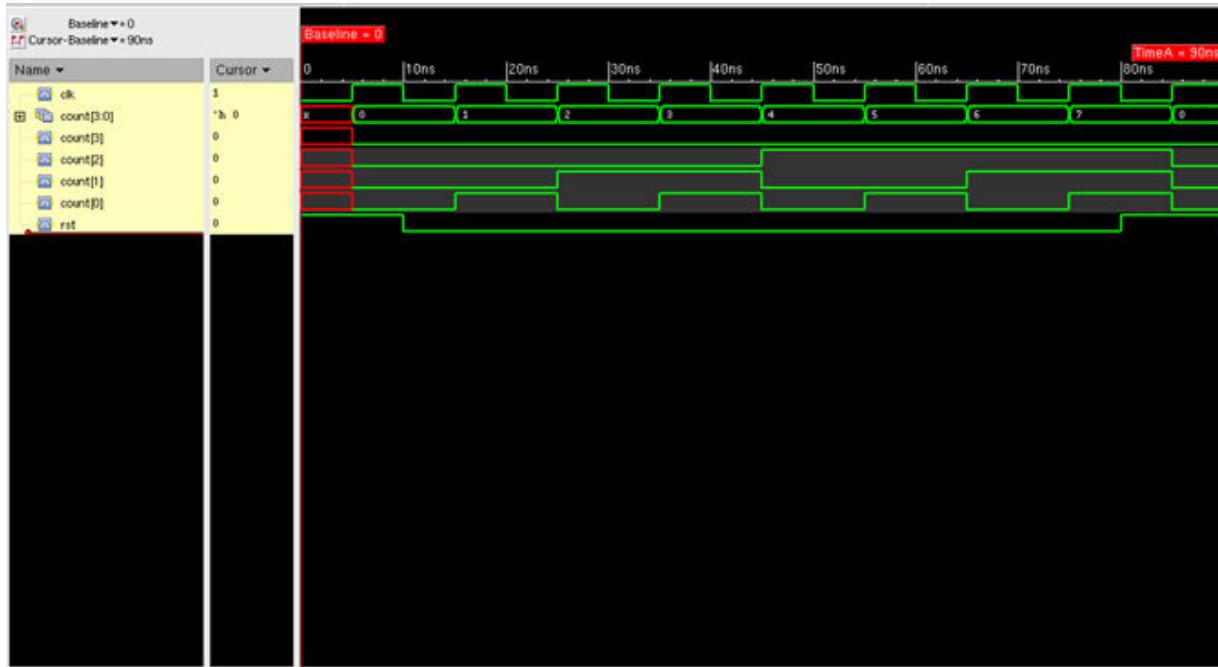
- Open the new terminal window and type the command given below.
- Mount -a
- Cd cadence\_db
- Csh
- Source cshrc
- Cd cadence\_ms\_labs\_614
- Virtuoso
- Now two windows (what's new window and virtuoso window) are open.
- Close the what's new window.
- In the virtuoso window, go to file -> new -> library.
- Now new library window is open.
- Type the file name.
- Activate attach to existing.
- Click ok.
- Now attach library window is open.
- Choose gpdk 180.
- Click ok.
- Again, in the virtuoso window, go to file->new->cell view->cell
- name (type the cell name)->ok.
- The schematic window is open.

**PROGRAM:**

```
module counter(clk,rst,count);
input clk;
input rst;
output [3:0]count;
reg [3:0] count;
always s@ (posedge clk) begin
if(rst) count<=4'b0000;
else
count<=count+4,b0001;
end
endmodule
```

**SCHEMATIC DIAGRAM:**

## OUTPUT:



## RESULT:

Hence the 4-bit synchronous counter using D flip flop is designed and simulated using Cadence application.

<b>Ex No:</b>	<b>Design and Simulate a CMOS Inverting Amplifier.</b>
<b>Date:</b>	

### **AIM:**

To design and simulate a CMOS Inverting Amplifier using Cadence application.

### **SOFTWARE REQUIRED:**

- Pc loaded with linux os
- Cadence software

### **THEORY:**

#### **CMOS INVERTER AS AN AMPLIFIER:**

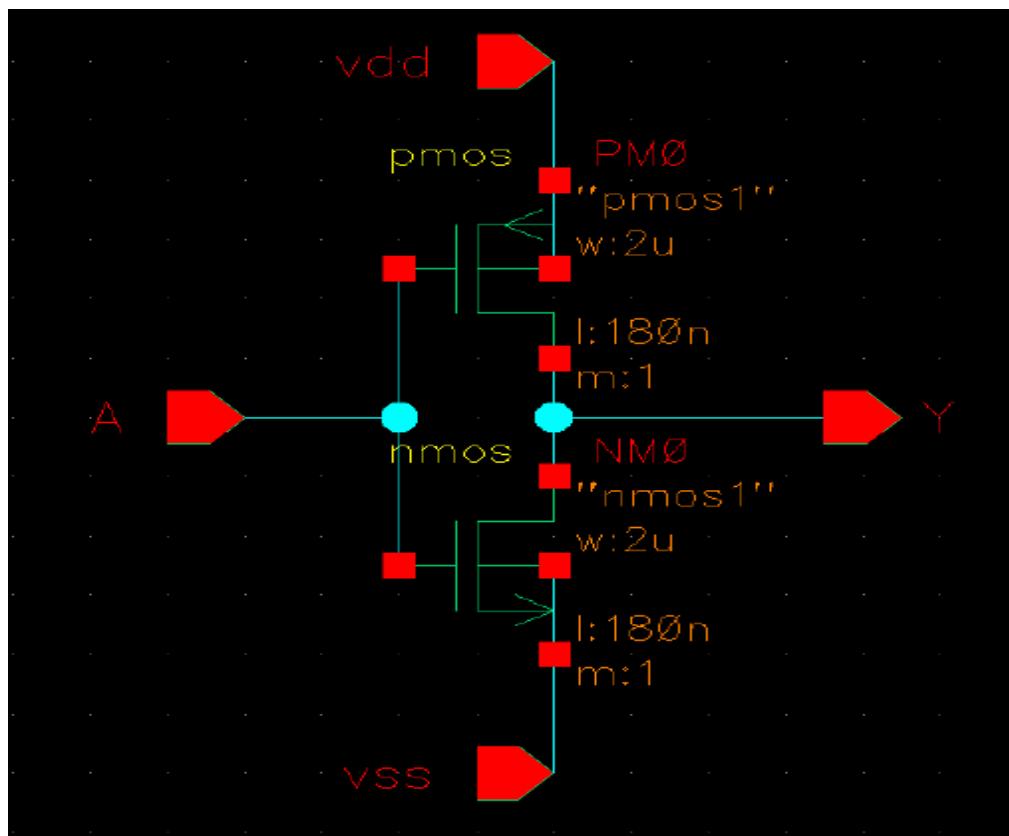
The inverter is the basic gain stage of CMOS analog circuits. In this the inverter uses the common source configuration with active resistor as a load or a current source as a load. The various configurations of CMOS inverter amplifier are

- 1) active load inverter
- 2) Current source load inverter
- 3) Push-pull inverter.

### **PROCEDURE:**

- Open the new terminal window and type the command given below.
- Mount -a
- Cd cadence\_db
- Csh
- Source cshrc
- Cd cadence\_ms\_labs\_614
- Virtuoso
- Now two windows (what's new window and virtuoso window) are open.
- Close the what's new window.
- In the virtuoso window, go to file -> new -> library.
- Now new library window is open.
- Type the file name.
- Activate attach to existing.
- Click ok.
- Now attach library window is open.
- Choose gpdk 180.
- Click ok.
- Again, in the virtuoso window, go to file->new->cell view->cell name (type the cell name)->ok.
- The schematic window is open.

## CIRCUIT DIAGRAM:



## SELECTING THE COMPONENTS:

- In schematic window ->create ->instance->library->browse->gpdk 180->cellview ->device to be selected->close->hide.
- For the pin configuration create->pin.
- After selecting the components needed for the design create the circuit diagram by connecting the components.
- Then click check and save (“schematic with no error” message display in virtuoso window).

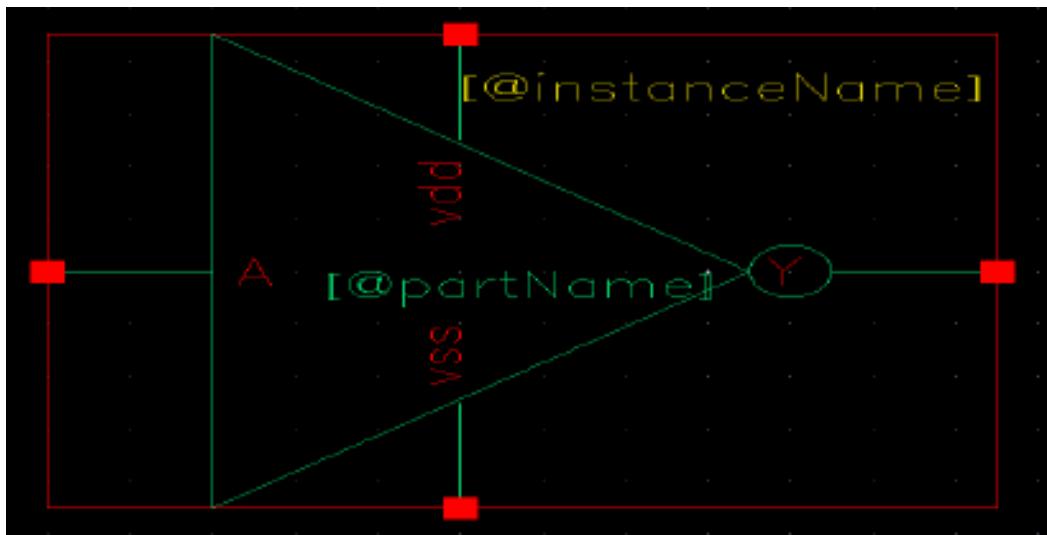
## Generation Of The Block Diagram:

- Create the cell view.
- Create->cell view->from cell view->ok.
- In the symbol generation window, assign vin for left, vout for right, vdd for top and vss for bottom pins.
- Then click ok.
- Symbol editor window will open.

## Create Polygon:

- To change the rectangular sized window into required shape.
- Delete the rectangular lines by using delete icon.

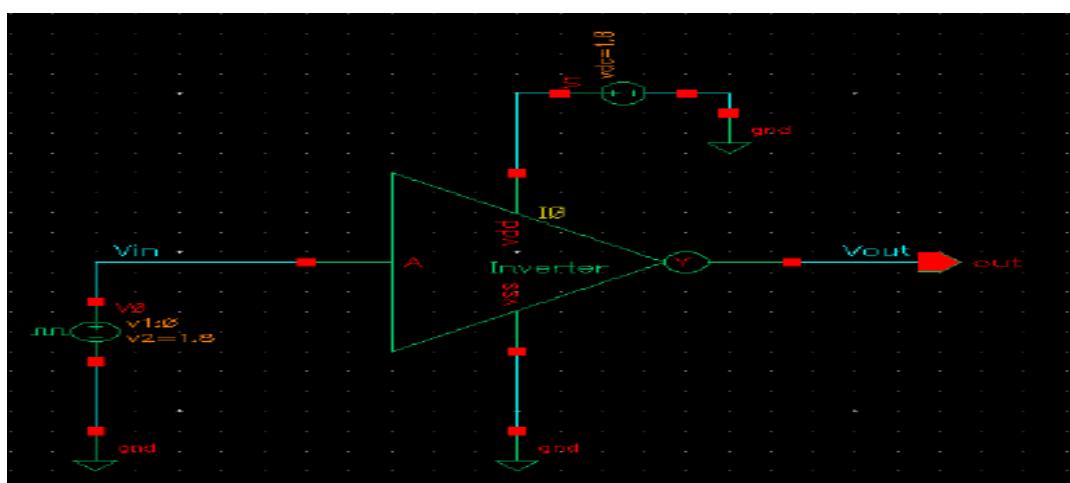
- Then create->shape->polygon->draw the required shape.
- Save and close the window.



#### Creation of the inverter test circuit:

- Again, in virtuoso window go to file ->new->cell view
- In new file window->cellbar->file name \_test
- For the selection of pins go to analog library and select the required pins.
- Then draw the symbol diagram and click check and save it.

Library name	Cellview name	Properties/Comments
myDesignLib	Inverter	Symbol
analogLib	vpulse	v1=0, v2=1.8, td=0 tr=tf=1ns, ton=10n, T=20n
analogLib	vdc, gnd	vdc=1.8



**SIMULATION:**

- In the simulation window go to launch->ADE L

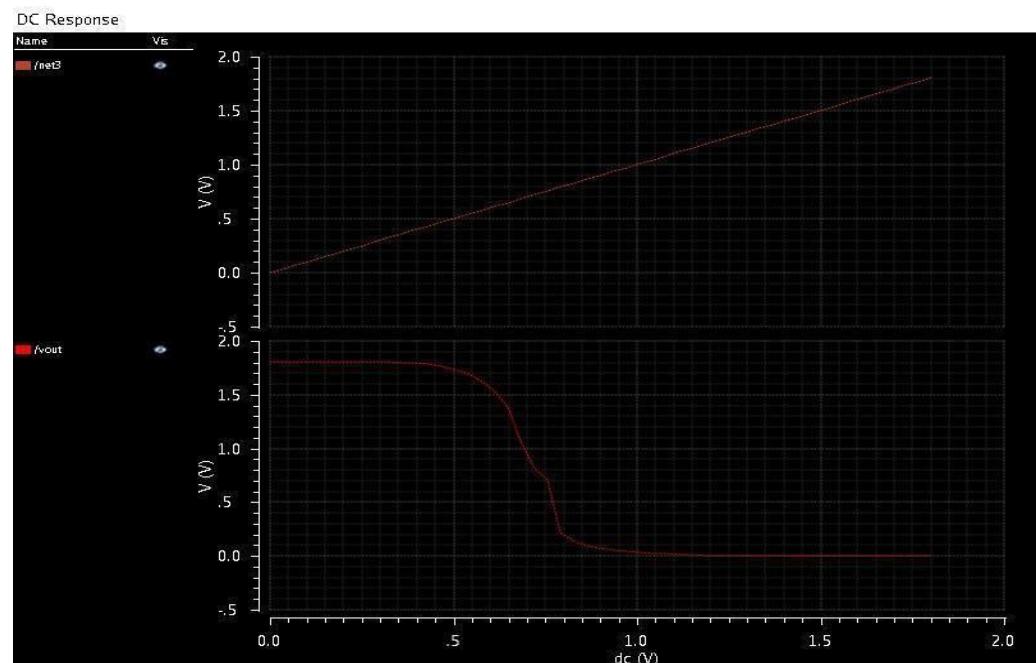
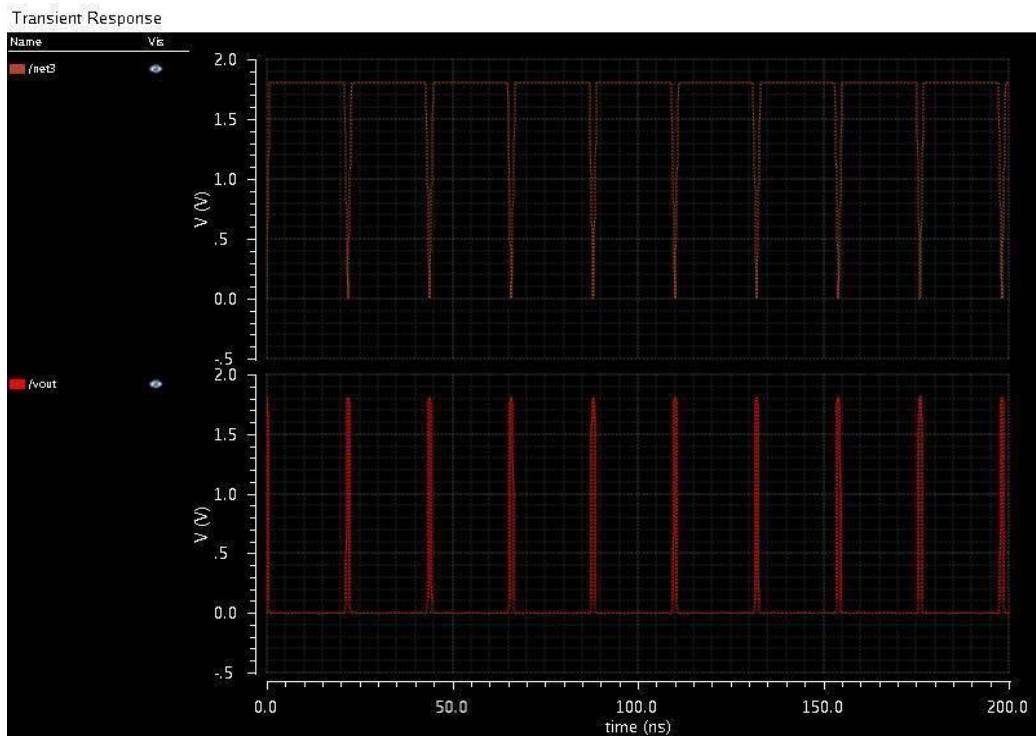
**Choosing a simulator:**

- In the simulation window (ADE), execute Setup ->simulator/directory/host ->spectre ->ok.

**Setting the model libraries:**

- In the simulation window (ade), Execute setup – model libraries.  
Your Model Library Setup Window Should Now Look Like The Below Figure.
- In the ade l window, analyses->choose.
- To setup for transient analysis
  1. In the analysis section select tran
  2. Set the stop time as 200n
  3. Click at the moderate or enabled button at the bottom, and then click apply
- In Ade I window select output->to be plotted->select on schematic (select the pin which you want to see in the waveform window).
- After selection go to simulation->netlist and run.
- The inverted output will be shown in waveform window

## OUTPUT:



## RESULT:

Thus, the simple inverter has been designed and its output and input characteristics has been simulated.

<b>Ex No:</b>	<b>Design and Simulate basic common source amplifiers</b>
<b>Date:</b>	

### **AIM:**

To design the common source amplifier by using cadence Application.

### **SOFTWARE REQUIRED:**

- Pc loaded with linux os
- Cadence software

### **THEORY:**

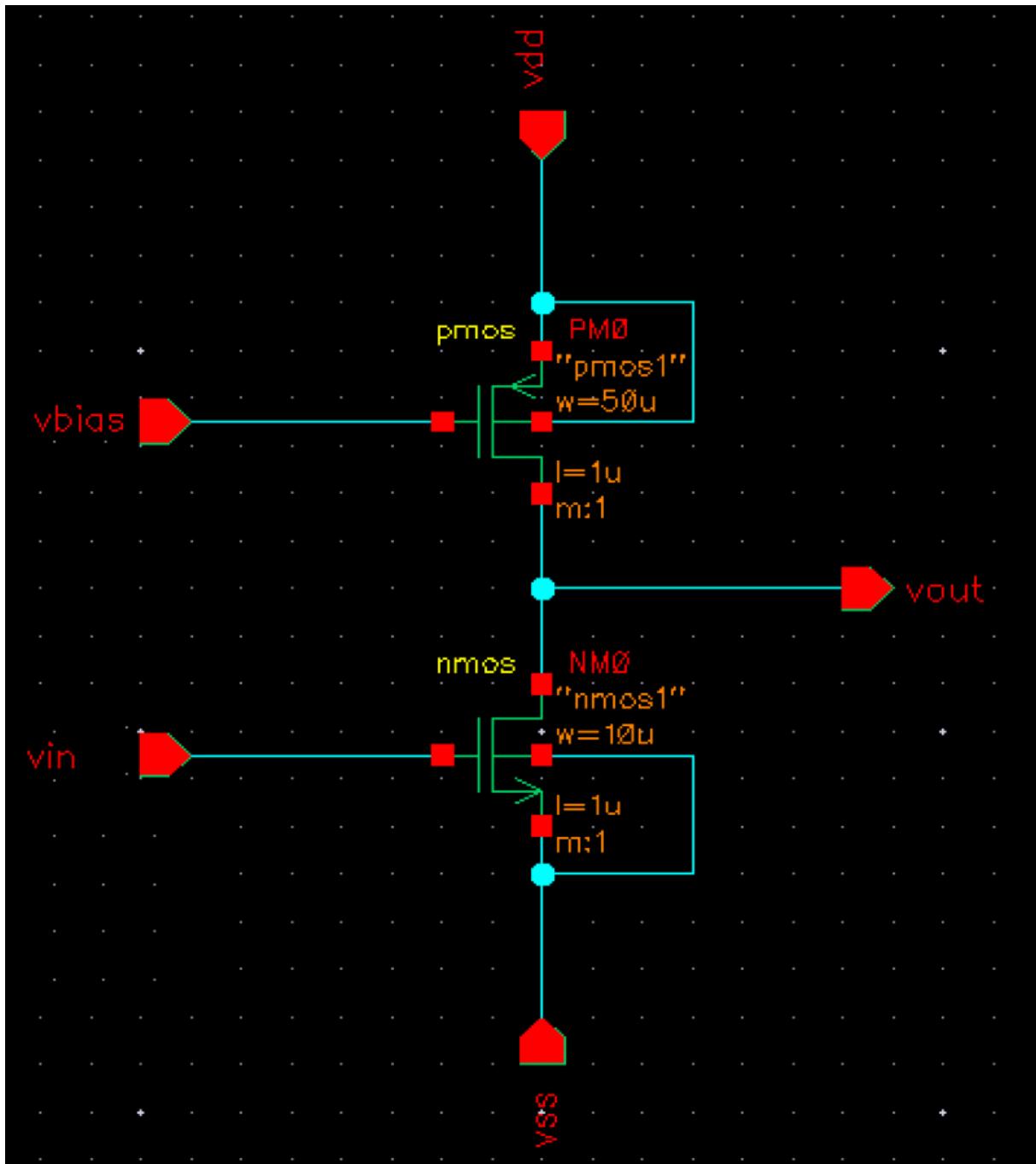
#### **Common source:**

A common-source amplifier is one of three basic single-stage field-effect transistor (FET) amplifier topologies, typically used as a voltage or transconductance amplifier. The easiest way to tell if a FET is common source, common drain, or common gate is to examine where the signal enters and leaves. The remaining terminal is what is known as "common". In this example, the signal enters the gate, and exits the drain. The only terminal remaining is the source. This is a common-source FET circuit.

### **PROCEDURE:**

- Open the new terminal window and type the command given below.
- Mount -a
- Cd cadence\_db
- Csh
- Source cshrc
- Cd cadence\_ms\_labs\_614
- Virtuoso
- Now two windows (what's new window and virtuoso window) are open.
- Close the what's new window.
- In the virtuoso window, go to file -> new -> library.
- Now new library window is open.
- Type the file name.
- Activate attach to existing.
- Click ok.
- Now attach library window is open.
- Choose gpdk 180.
- Click ok.
- Again, in the virtuoso window, go to file->new->cell view->cell name (type the cell name)->ok.
- The schematic window is open.

### CIRCUIT DIAGRAM:



### SELECTING THE COMPONENTS:

- In schematic window ->create ->instance->library->browse->gdk 180 ->cell view ->device to be selected->close.

Library name	Cell name	Properties/comments
Gpdk180	Pmos	Model name =pmos1; W=50u; l=1u
Gpdk180	Nmos	Model name=nmos1; W=10u; l=1u

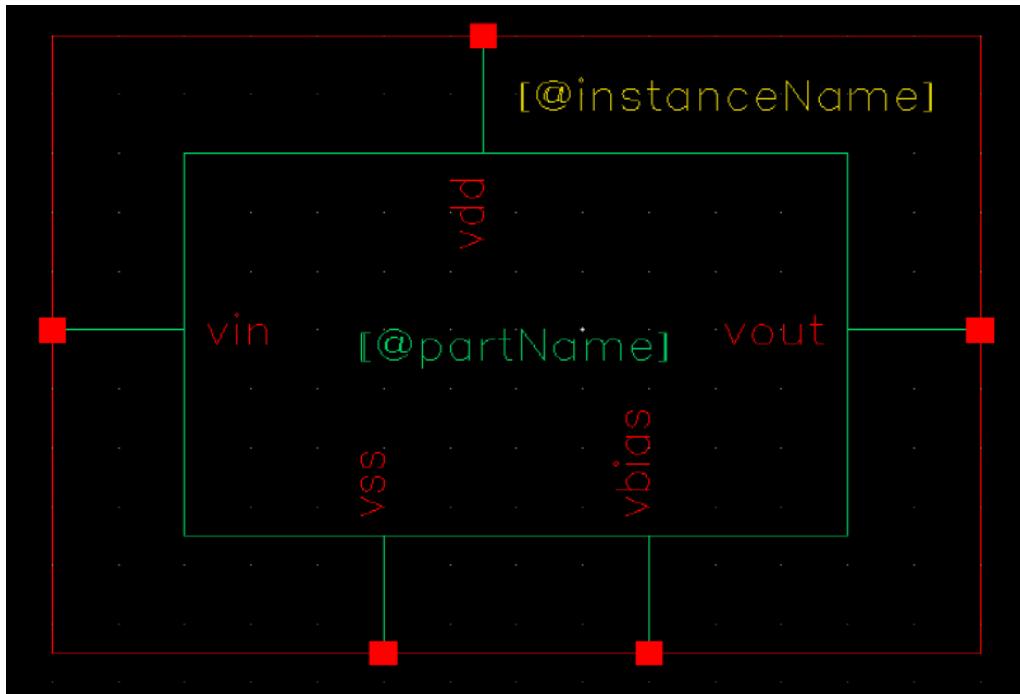
- For the Pin Configuration Create->Pin

Pin names	Directions
Vin, vbias	Input
Vout	Output
Vdd, vss	Input

- After selecting the components needed for the design create the circuit diagram by connecting the components.
- Then click check and save (“schematic with no error” message display in virtuoso window).

#### Generation of the block diagram:

- Create the cell view.
- Create->cell view->from cell view->ok.
- The cell view from cell view form appears. With the edit options function active, you can control the appearance of the symbol to generate.
- Verify that the from view name field is set to schematic, and the to view name field is set to symbol, with the tool/data type set as schematic symbol.
- Click ok in the cell view from cell view form. The symbol generation form appears.
- Modify the pin specifications.
- Click ok in the symbol generation options form.
- A new window displays an automatically created common source amplifier symbol.
- Modifying automatically generated symbol.
- Execute create->selection box. In the add selection box form, click automatic. A new red selection box is automatically added.
- After creating symbol, click on the save icon in the symbol editor window to save the symbol. In the symbol editor, execute file->close to the close the symbol view window.



#### CREATING THE COMMON SOURCE AMPLIFIER TEST CELL VIEW:

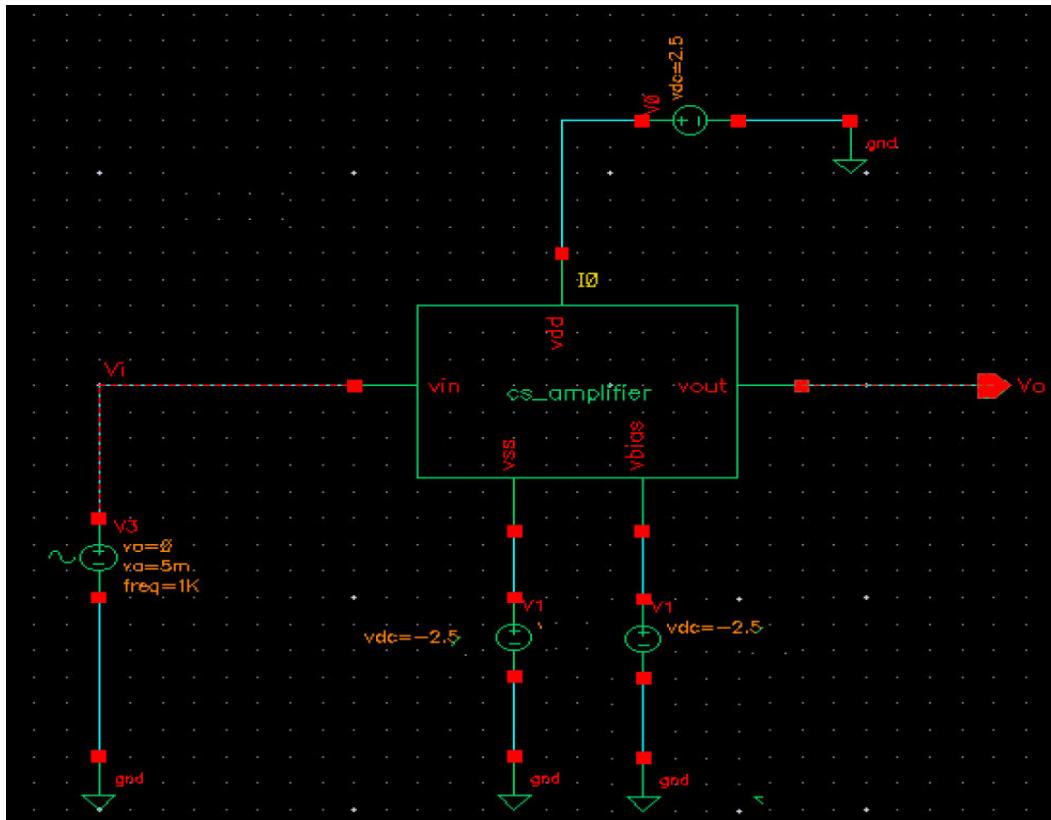
- In the ciw or library manager, execute file-> new->cellview
- Setup the create new file as cs\_amplifier\_test.
- Click ok when done.
- A blank schematic window for the cs\_amplifier\_test design appears.

#### BUILDING THE CS\_AMPLIFIER\_TEST CIRCUIT:

- Using the component list and properties/comments in this table,
- Build the cs\_amplifier\_test schematic

Library name	Cell view name	Properties/comments
Mydesignlib	Cs_amplifier	Symbol
Analoglib	Vsin	Define pulse specification as ac magnitude=1; dc voltage=0; offset voltage=0; amplitude=5m; frequency=1k
Analoglib	Vdd,vss,gnd	vdd=2.5; vss=-2.5 vbias=2.5

- Click the wire (narrow)icon and wire your schematic.
- Tip: you can also press the w key, or execute create->wire(narrow).
- Click on the check and save icon to save the design.
- The schematic should look like this.
- Leave your cs\_amplifier \_test schematic window open for the next section.
- Launch->ade l->design->model library.



### CHOOSING ANALYSES:

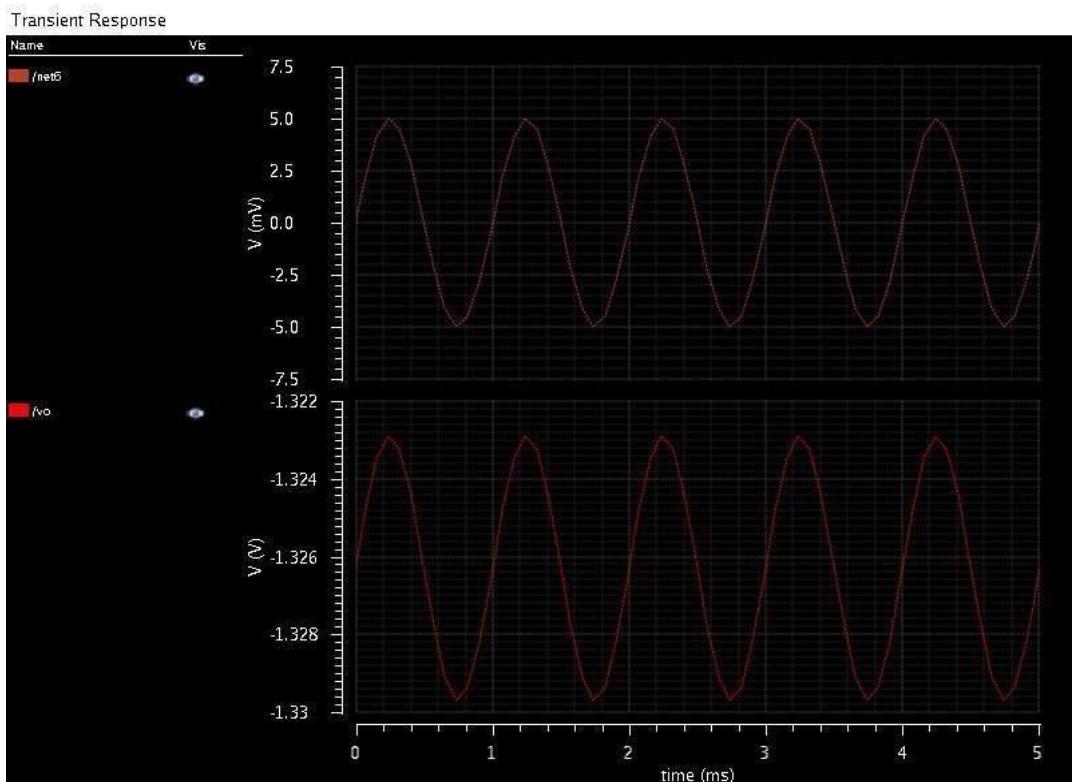
- In the simulation window, click the choose->analyses icon.
- You can also execute analyses->choose.
- The choosing analysis form appears. This is dynamic form, the bottom of the form changes based on the selection above.
- To setup for transient analysis
- In the analysis section select tran
- Set the stop time as 5m
- Click at the moderate or enabled button at the bottom, and then click apply.
- To set up for dc analyses:
- In the analyses section, select dc. In the dc analyses section, turn on save dc operating point.
- Turn on the component parameter.
- Double click the select component, which takes you to the schematic window.
- Type the parameter name as "dc".
- Select input signal vsin dc analysis
- In the analysis form, select start and stop voltages as -5 to 5 respectively.
- Check the enable button and then click apply.
- To set up for ac analyses form is shown in the previous page.
- In the analyses section, select ac
- In the ac analyses section, turn on frequency.

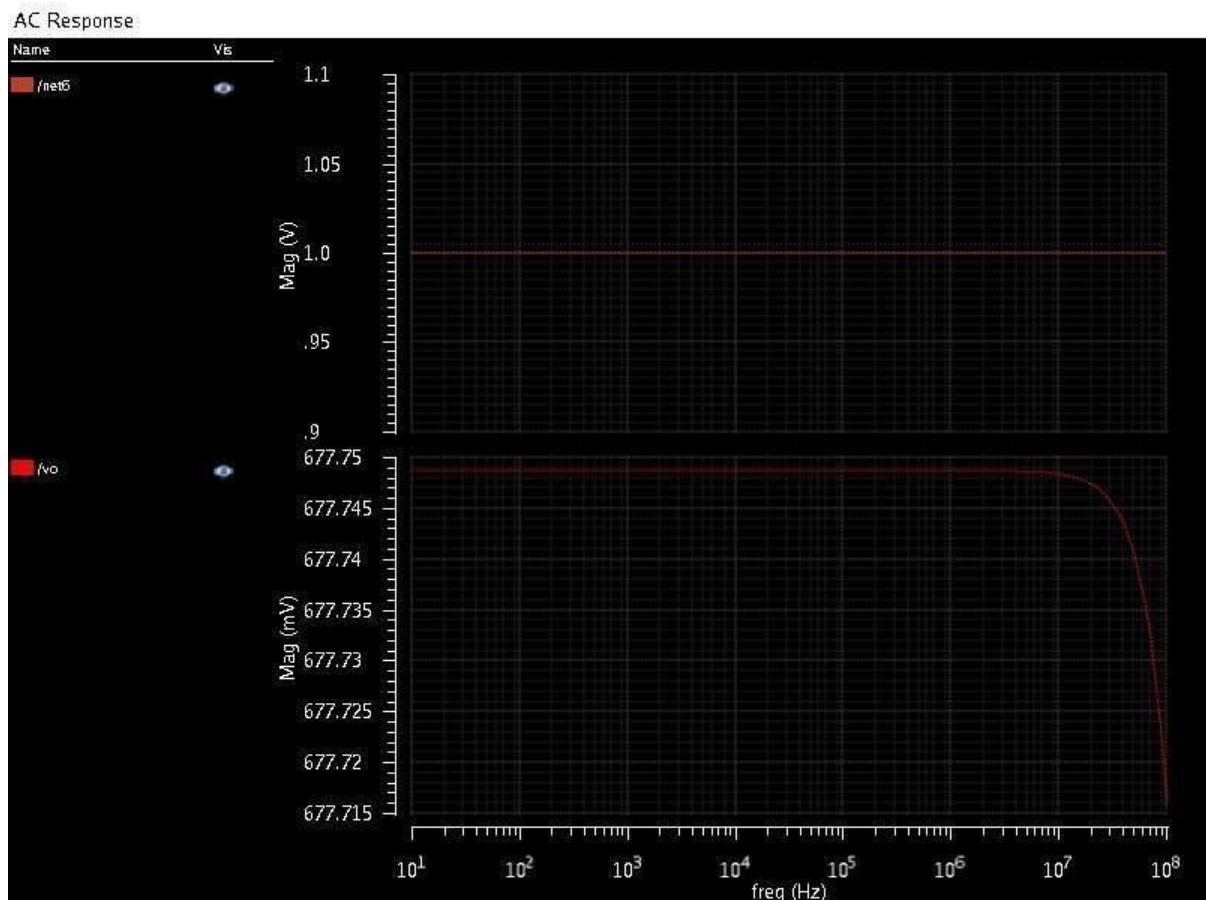
- In the sweep range section select start and stop frequencies as 10 to 100m
- Select points per decade as 20
- Check the enable button and then click apply.
- Click ok in the choosing analyses form

### **SELECTING OUTPUTS FOR PLOTTING:**

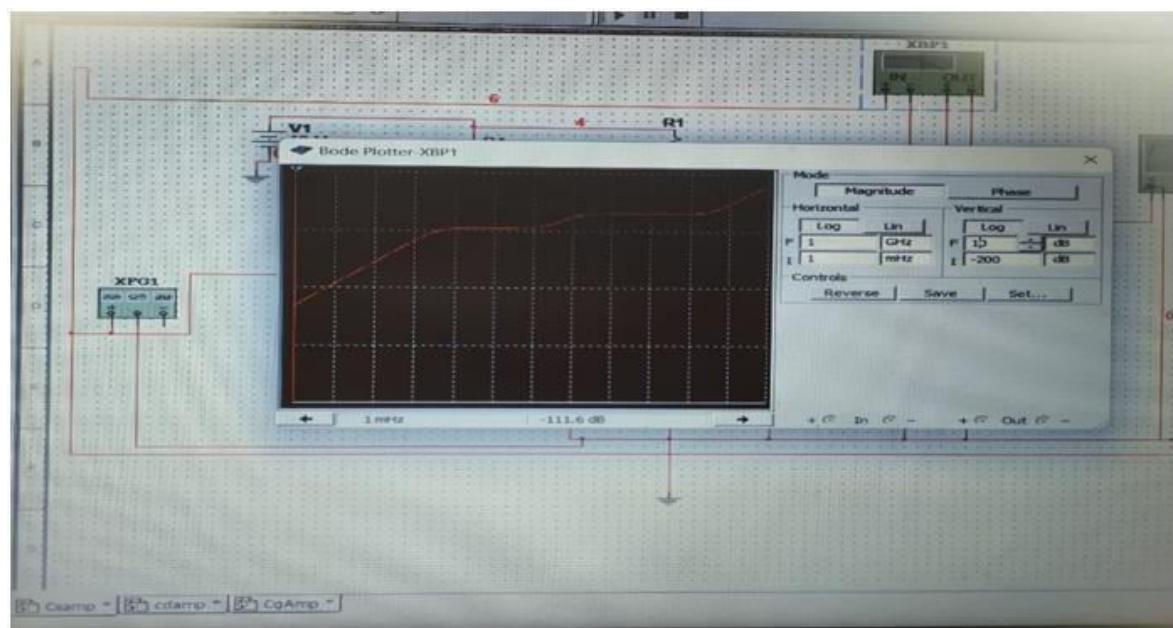
- Execute outputs->to be plotted->select on schematic in the simulation window.
- Follow the prompt at the bottom of the schematic window, click on output net v0, input net vin of the cs\_amplifier. Press esc with the cursor in the schematic after selecting node. Next go to adel window results->direct plot->ac db20 and it direct to schematic window, click the output nets from the schematic and press escape. The output waveform will appear.
- Tools->calculator.

Select the waveform, the waveform detail is added in the calculator, now select average in the function panel, click evaluate the buffer and stack, now the gain value is display on the calculator window.





## OUTPUT :



**RESULT:**

Thus, the common source amplifier has been designed and the simulation of its input and output characteristics has been done successfully.

<b>Ex No:</b>	<b>Design and Simulate a basic common gate amplifiers</b>
<b>Date:</b>	

### **AIM:**

To design and simulate a basic common gate amplifiers using Cadence application.

### **SOFTWARE REQUIRED:**

- Pc loaded with linux os
- Cadence software

### **THEORY:**

A common-gate amplifier is one of three basic single-stage field-effect transistor (FET) amplifier topologies, typically used as a current buffer or voltage amplifier. In this circuit the source terminal of the transistor serves as the input, the drain is the output and the gate is connected to ground, or "common," hence its name. The analogous bipolar junction transistor circuit is the common-base amplifier.

### **PROCEDURE:**

- Open the new terminal window and type the command given below.
- Mount -a
- Cd cadence\_db
- Csh
- Source cshrc
- Cd cadence\_ms\_labs\_614
- Virtuoso
- Now two windows (what's new window and virtuoso window) are open.
- Close the what's new window.
- In the virtuoso window, go to file -> new -> library.
- Now new library window is open.
- Type the file name.
- Activate attach to existing.
- Click ok.
- Now attach library window is open.
- Choose gpdk 180.
- Click ok.
- Again, in the virtuoso window, go to file->new->cell view->cell
- name (type the cell name)->ok.
- The schematic window is open.

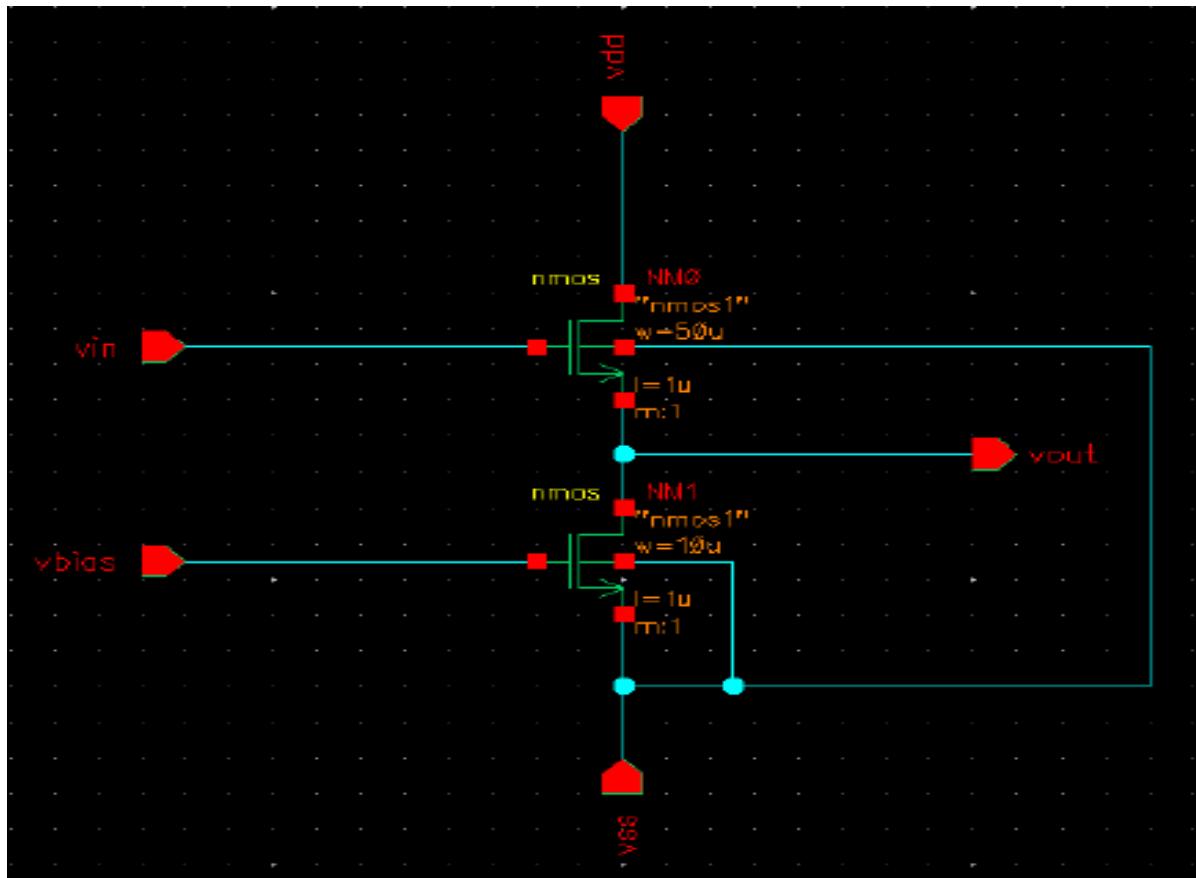
## SELECTING THE COMPONENTS:

In schematic window ->create ->instance->library->browse->gpdk 180->cell view ->device to be selected->close

Library name	Cell name	Properties/comments
Gpdk180	Nmos Nmos	Model name =nmos1; W=50u; l=1u
Gpdk180	Nmos Nmos	Model name=nmos1; W=10u; l=1u

(type the properties as mentioned above)

Then click hide.



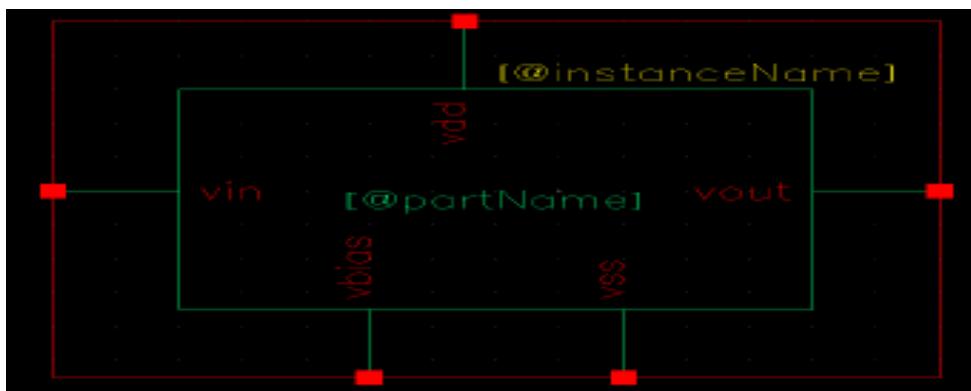
- For the pin configuration create->pin.

Pin names	Directions
Vin vbias	Input
Vout	Output
Vdd,vss	Input

- After selecting the components needed for the design create the circuit diagram by connecting the components.
- Then click check and save (“schematic with no error” message display in virtuoso window).

#### GENERATION OF THE BLOCK DIAGRAM:

- Create the cell view.
- Create->cell view->from cell view->ok.
- The cell view from cell view form appears. With the edit options function active, you can control the appearance of the symbol to generate.
- Verify that the from view name field is set to schematic, and the to view name field is set to symbol, with the tool/data type set as schematic symbol.
- Click ok in the cell view from cell view form. The symbol generation form appears.
- Modify the pin specifications.
- Click ok in the symbol generation options form.
- A new window displays an automatically created common drain amplifier symbol.
- Modifying automatically generated symbol.
- Execute create->selection box. In the add selection box form, click automatic.a new red selection box is automatically added.
- After creating symbol, click on the save icon in the symbol editor window to save the symbol. In the symbol editor, execute file->close to the close the symbol view window.



## CREATING THE COMMON DRAIN AMPLIFIER TEST CELLVIEW:

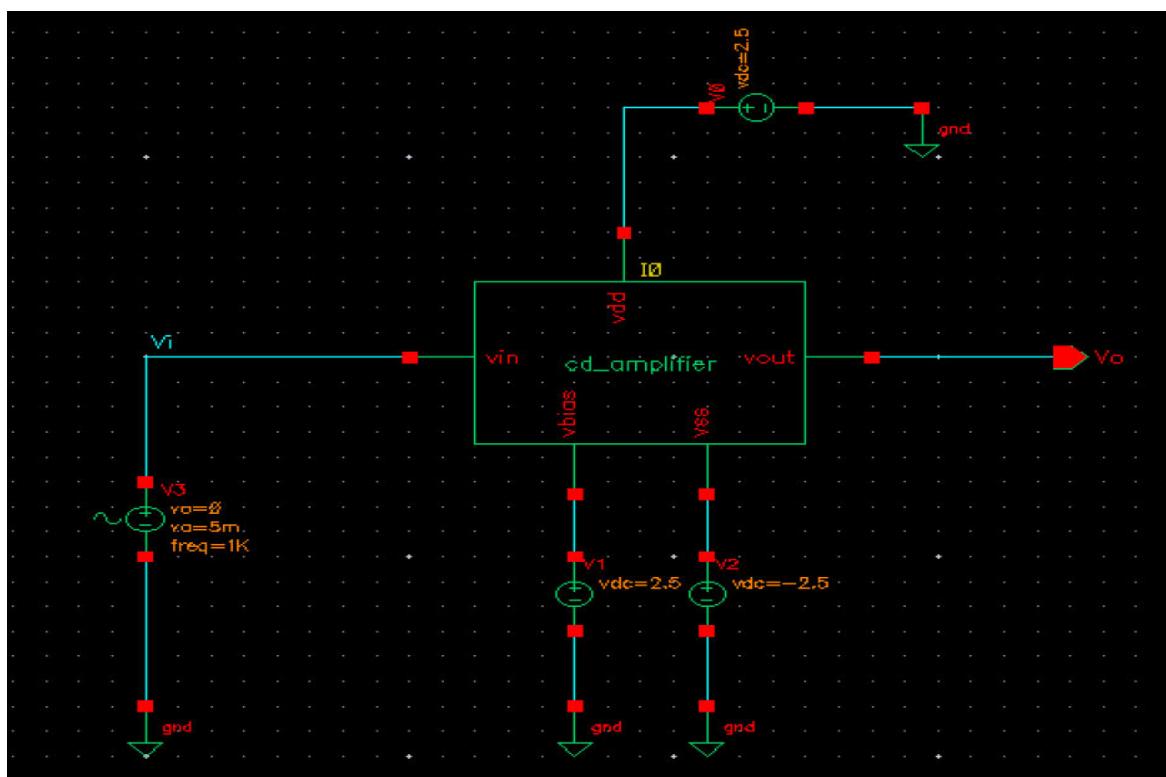
- In the ciw or library manager, execute file->new->cellview
- Setup the create new file as cd amplifier test.
- Click ok when done. A blank schematic window for the ed amplifier test design appears.

## BUILDING THE CD AMPLIFIER TEST CIRCUIT:

- Using the component list and properties/comments in this table, build the cs amplifier test schematic

Library name	Cell view name	Properties/comments
Mydesignlib	Cd_amplifier	Symbol
Analoglib	Vsin	Define pulse specification as ac magnitude=1; dcvoltag=0; offset voltage=0; amplitude=5m; frequency=1k
Analoglib	Vdd.vss.gnd	vdd=2.5; vss=-2.5; vbias=2.5

- Click the wire (narrow) icon and wire your schematic.
- Tip: you can also press the w key, or execute create->wire(narrow).
- Click on the check and save icon to save the design.
- The schematic should look like this.



- Leave your cd\_amplifier \_test schematic window open for the next section.
- Launch->ade l->design->model library.

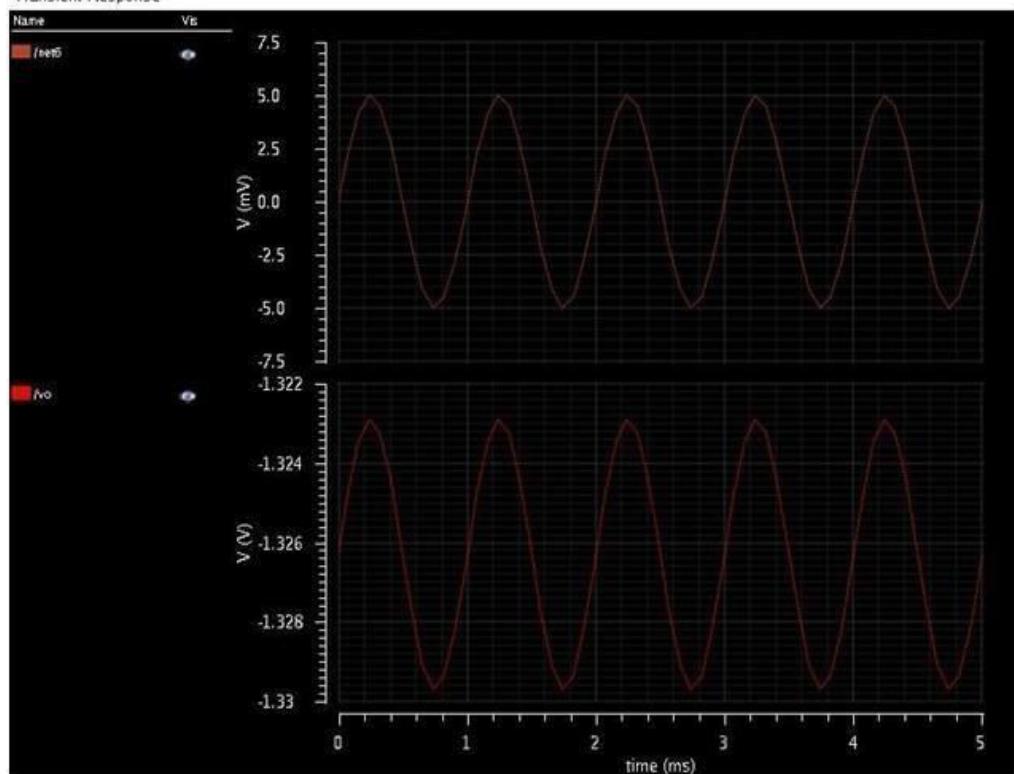
### **CHOOSING ANALYSES:**

- In the simulation window, click the choose->analyses icon.
- You can also execute analyses->choose.
- The choosing analysis form appears. This is dynamic form, the bottom of the form changes based on the selection above.
- To setup for transient analysis
- In the analysis section select tran
- Set the stop time as 5m
- Click at the moderate or enabled button at the bottom, and then click apply.
- To set up for dc analyses:
- In the analyses section, select dc.
- In the dc analyses section, turn on save dc operating point.
- Turn on the component parameter.
- Double click the select component, which takes you to the schematic window.
- Type the parameter name as “dc”.
- Select input signal vsin dc analysis.
- In the analysis form, select start and stop voltages as -5 to 5 respectively.
- Check the enable button and then click apply.
- To set up for ac analyses form is shown in the previous page.
- In the analyses section, select ac.
- In the ac analyses section, turn on frequency.
- In the sweep range section select start and stop frequencies as 10 to 100m
- Select points per decade as 20.
- Check the enable button and then click apply.
- Click ok in the choosing analyses form

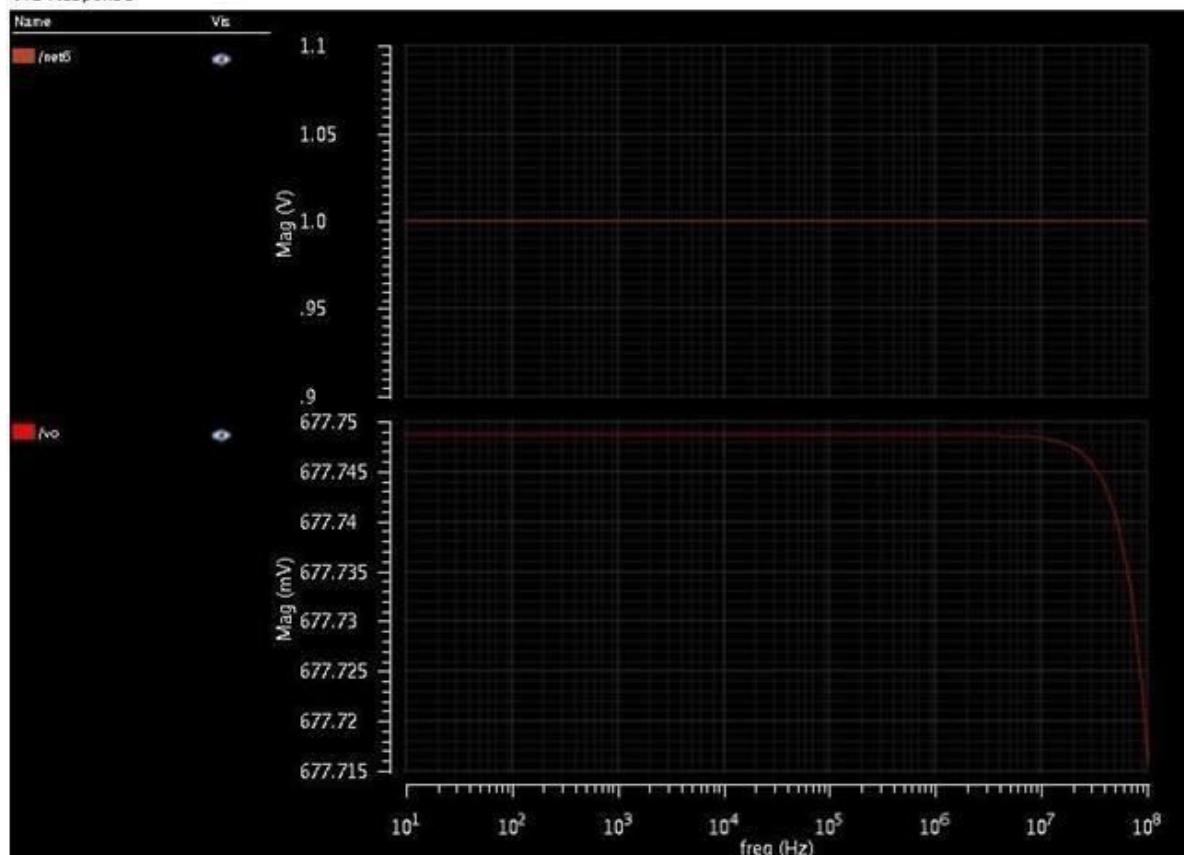
### **SELECTING OUTPUTS FOR PLOTTING:**

- Execute outputs->to be plotted->select on schematic in the simulation window.
- Follow the prompt at the bottom of the schematic window, click on output net v0, input net vin of the cd amplifier. Press esc with the cursor in the schematic after selecting node.
- Next go to adel window results->direct plot->ac db20 and it direct to schematic www window, click the output nets from the schematic and press escape. The output waveform will appear.
- Tools->calculator.
- Select the waveform, the waveform detail is added in the calculator, now select average in the function panel, click evaluate the buffer and stack, now the gain value is displayed on the calculator window

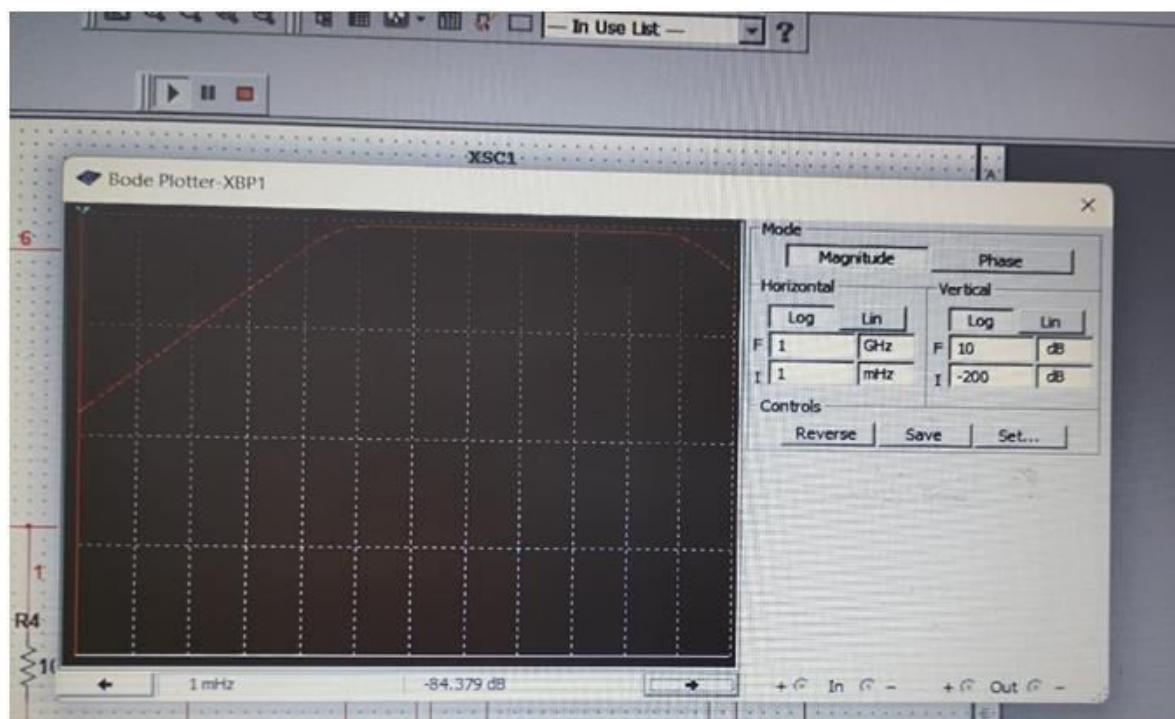
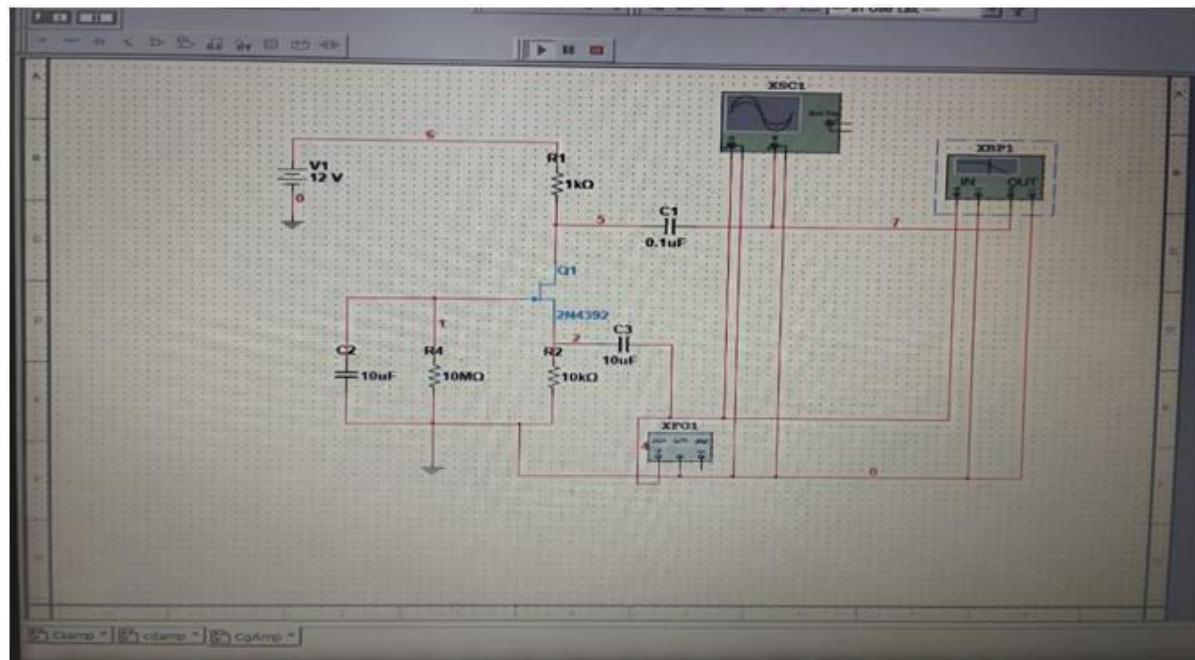
### Transient Response



### AC Response



## OUTPUT:



**RESULT:**

Thus, the common gate amplifier has been designed and simulation of its input and output characteristics has been done successfully.

<b>Ex No:</b>	<b>Design and Simulate a common drain amplifiers</b>
<b>Date:</b>	

### **AIM:**

To design and simulate a common drain amplifiers using Cadence application.

### **SOFTWARE REQUIRED:**

- Pc loaded with linux os
- Cadence software

### **THEORY:**

A common-drain amplifier, also known as a source follower, is one of three basic single-stage field effect transistor (FET) amplifier topologies, typically used as a voltage buffer. In this circuit (NMOS) the gate terminal of the transistor serves as the input, the source is the output, and the drain is *common* to both (input and output), hence its name. The analogous bipolar junction transistor circuit is the common-collector amplifier. This circuit is also commonly called a "stabilizer." In addition, this circuit is used to transform impedances.

### **PROCEDURE:**

- Open the new terminal window and type the command given below.
- Mount -a
- Cd cadence\_db
- Csh
- Source cshrc
- Cd cadence\_ms\_labs\_614
- Virtuoso
- Now two windows (what's new window and virtuoso window) are open.
- Close the what's new window.
- In the virtuoso window, go to file -> new -> library.
- Now new library window is open.
- Type the file name.
- Activate attach to existing.
- Click ok.
- Now attach library window is open.
- Choose gpdk 180.
- Click ok.
- Again, in the virtuoso window, go to file->new->cell view->cell name (type the cell name)->ok.
- The schematic window is open.

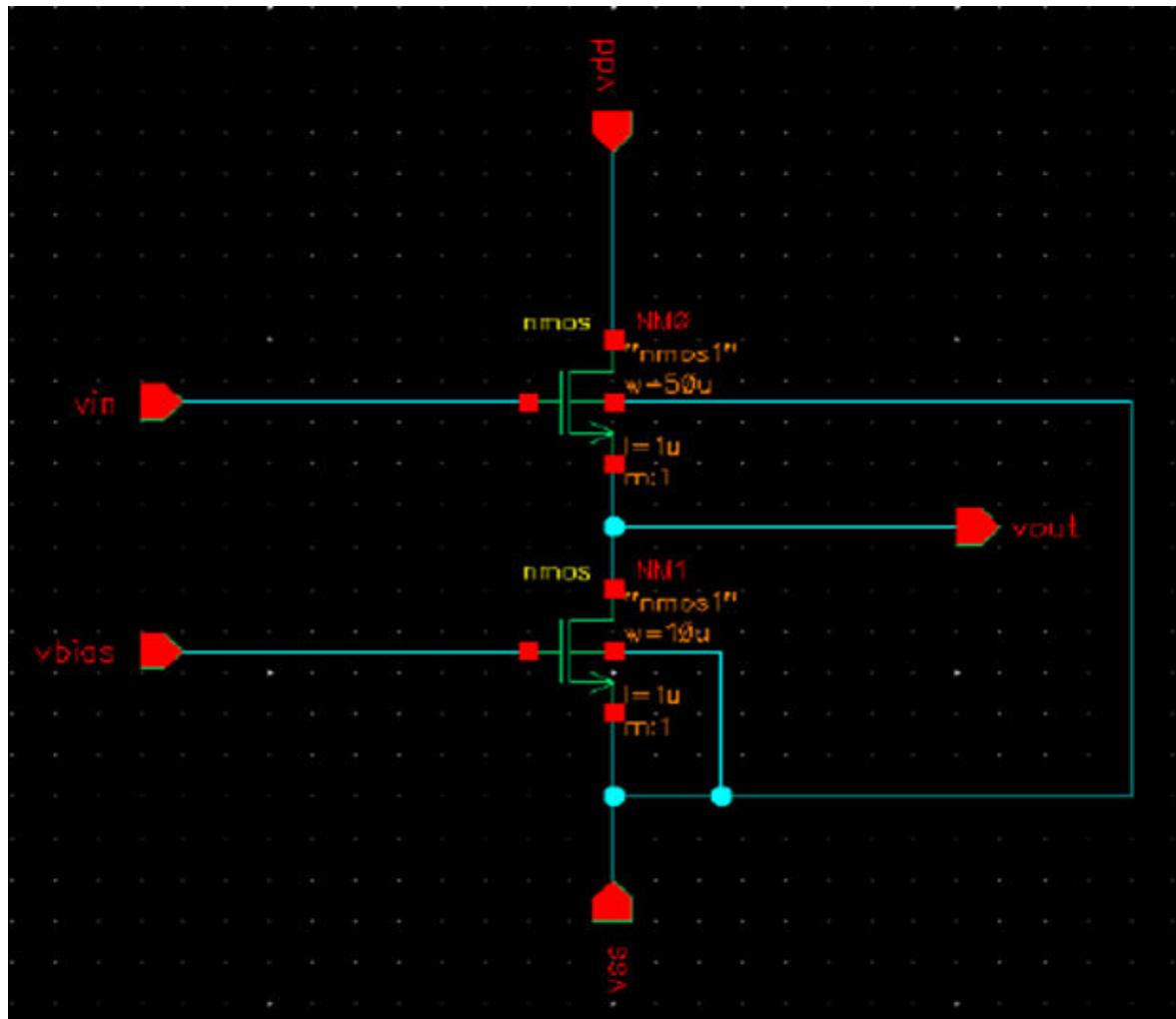
## SELECTING THE COMPONENTS:

- In schematic window ->create ->instance->library->browse->gpdk 180->cell view ->device to be selected->close

Library name	Cell name	Properties/comments
Gpdk180	Nmos	Modelname =nmos1; W=50u; l=1u
Gpdk180	Nmos	Modelname=nmos1; W=10u; l=1u

(type the properties as mentioned above)

then click hide.



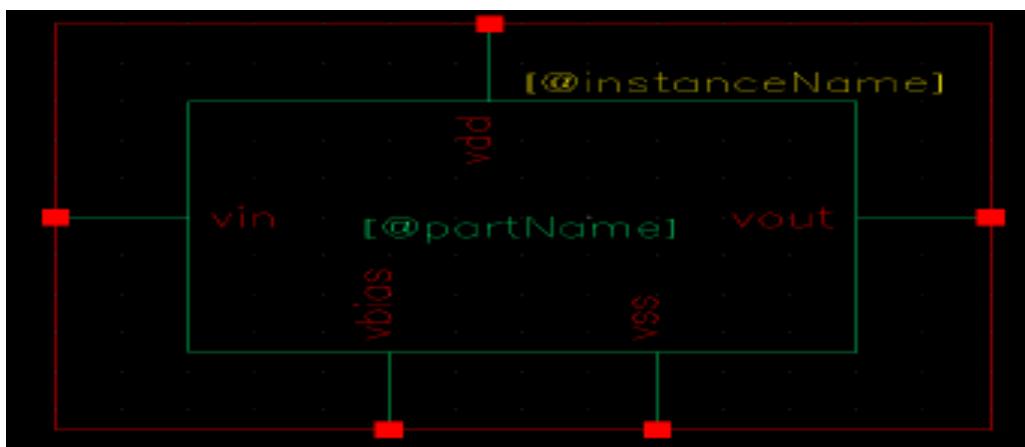
- For the pin configuration create->pin.

Pin names	Directions
Vin <u>vbias</u>	Input
Vout	Output
Vdd,vss	Input

- After selecting the components needed for the design create the circuit diagram by connecting the components.
- Then click check and save (“schematic with no error” message display in virtuoso window).

#### GENERATION OF THE BLOCK DIAGRAM:

- Create the cell view.
- Create->cell view->from cell view->ok.
- The cell view from cell view form appears. With the edit options function active, you can control the appearance of the symbol to generate.
- Verify that the from view name field is set to schematic, and the to view name field is set to symbol, with the tool/data type set as schematic symbol.
- Click ok in the cell view from cell view form. The symbol generation form appears.
- Modify the pin specifications.
- Click ok in the symbol generation options form.
- A new window displays an automatically created common drain amplifier symbol.
- Modifying automatically generated symbol.
- Execute create->selection box. In the add selection box form, click automatic.a new red selection box is automatically added.
- After creating symbol, click on the save icon in the symbol editor window to save the symbol. In the symbol editor, execute file->close to close the symbol view window.



## CREATING THE COMMON DRAIN AMPLIFIER TEST CELLVIEW:

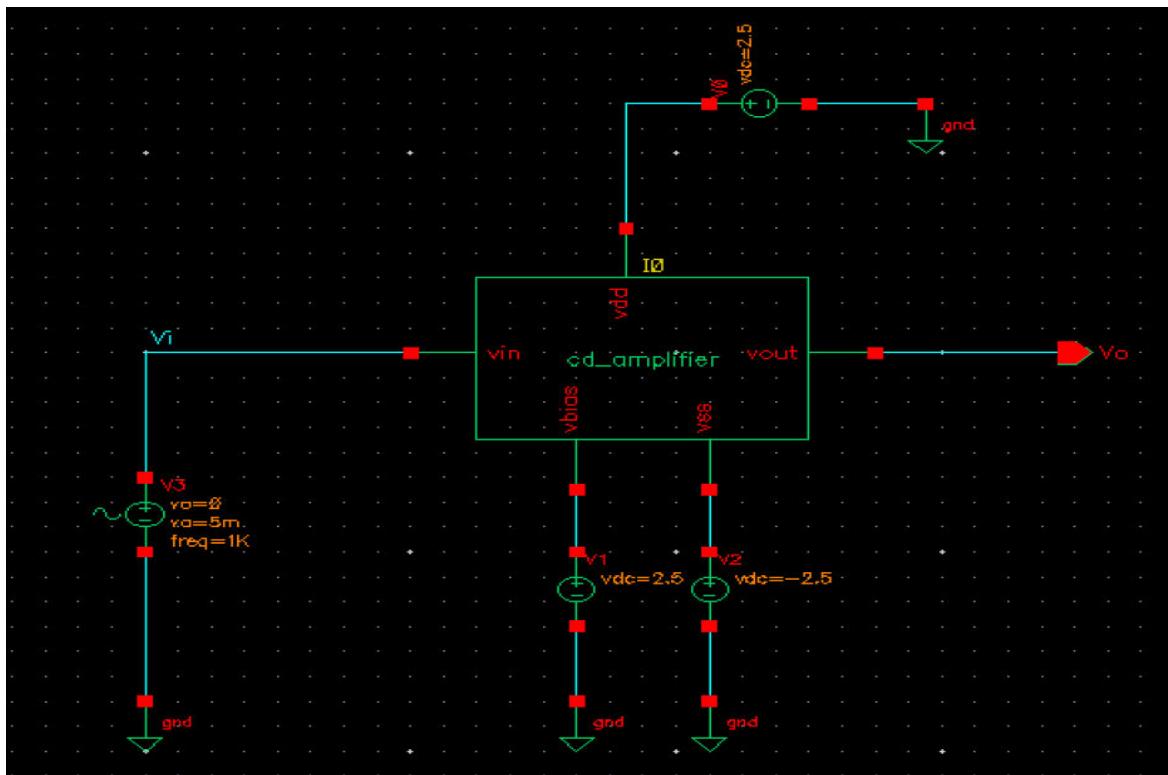
- In the ciw or library manager, execute file->new->cellview
- Setup the create new file as cd amplifier test.
- Click ok when done. A blank schematic window for the ed amplifier test design appears.

## BUILDING THE CD AMPLIFIER TEST CIRCUIT:

Using the component list and properties/comments in this table, build the cs amplifier test schematic

Library name	Cell view name	Properties/comments
Mydesignlib	Cd_amplifier	Symbol
Analoglib	Vsin	Define pulse specification as ac magnitude=1; dc voltage=0; offset voltage=0; amplitude=5m; frequency=1k
Analoglib	Vdd.vss.gnd	vdd=2.5; vss=-2.5; vbias=2.5

- Click the wire (narrow) icon and wire your schematic.
- Tip: you can also press the w key, or execute create->wire(narrow).
- Click on the check and save icon to save the design.
- The schematic should look like this.



- Leave your cd\_amplifier \_test schematic window open for the next section.
- Launch->ade l->design->model library.

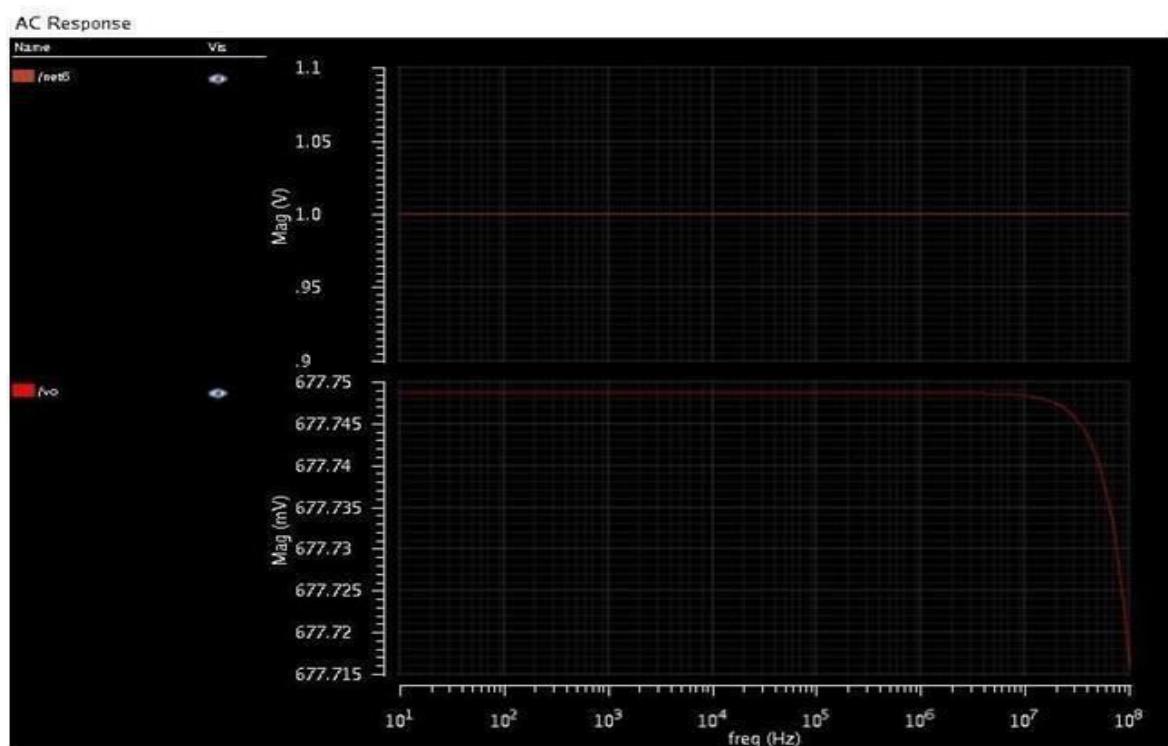
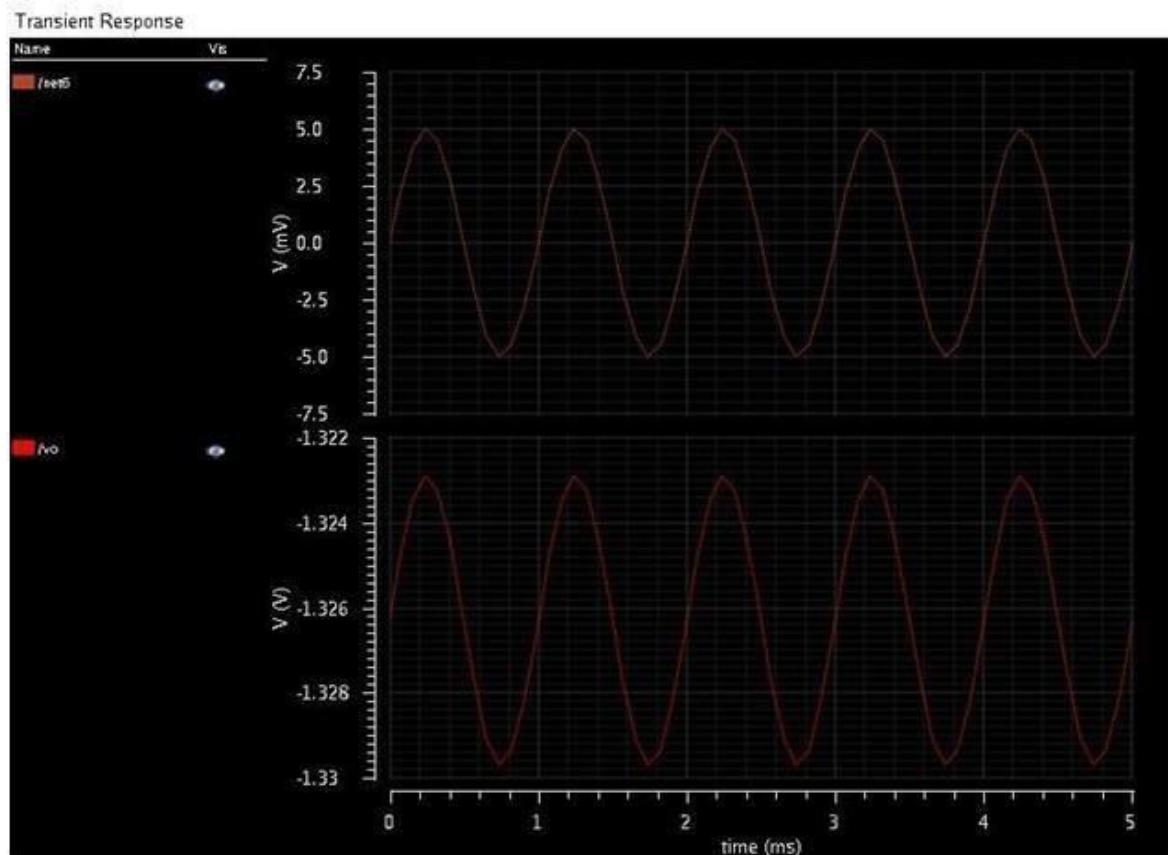
### **CHOOSING ANALYSES:**

- In the simulation window, click the choose->analyses icon.
- You can also execute analyses->choose.
- The choosing analysis form appears. This is dynamic form, the bottom of the form changes based on the selection above.
- To set up for transient analysis
- In the analysis section select tran
- Set the stop time as 5m
- Click at the moderate or enabled button at the bottom, and then click apply.
- To set up for dc analyses:
- In the analyses section, select dc.
- In the dc analyses section, turn on save dc operating point.
- Turn on the component parameter.
- Double click the select component, which takes you to the schematic window.
- Type the parameter name as “dc”.
- Select input signal vsin dc analysis.
- In the analysis form, select start and stop voltages as -5 to 5 respectively.
- Check the enable button and then click apply.
- To set up for ac analyses form is shown in the previous page.
- In the analyses section, select ac.
- In the ac analyses section, turn on frequency.
- In the sweep range section select start and stop frequencies as 10 to 100m
- Select points per decade as 20.
- Check the enable button and then click apply.
- Click ok in the choosing analyses form

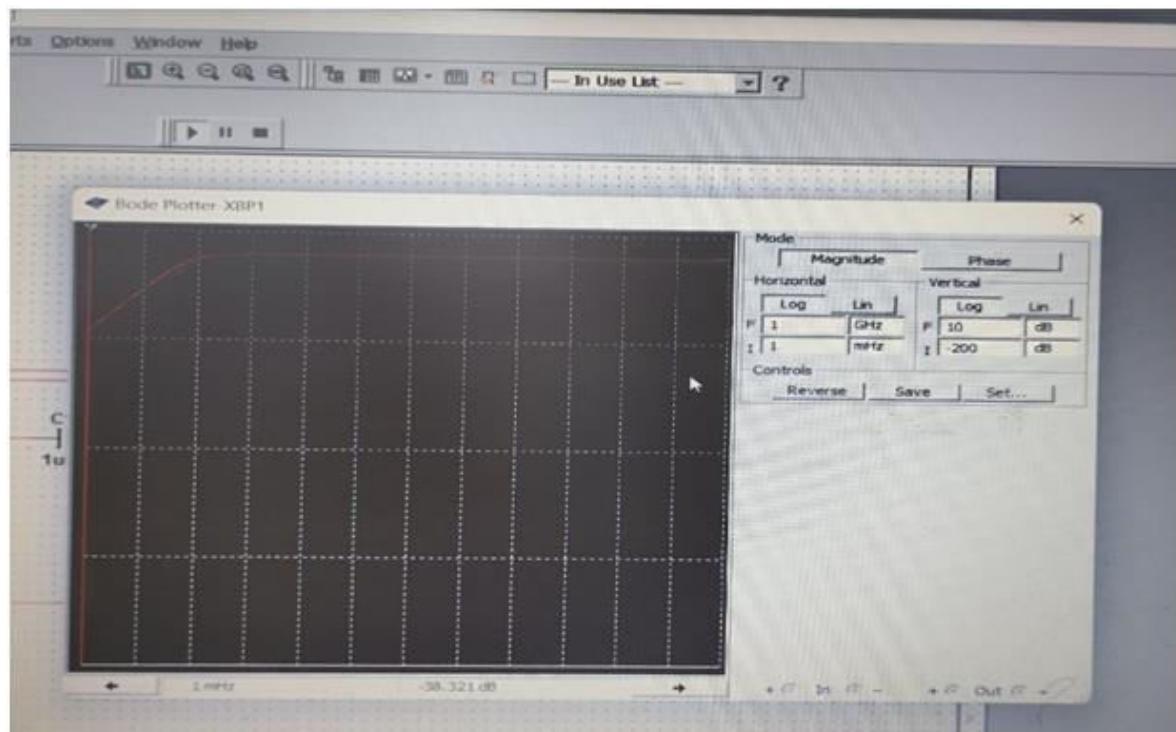
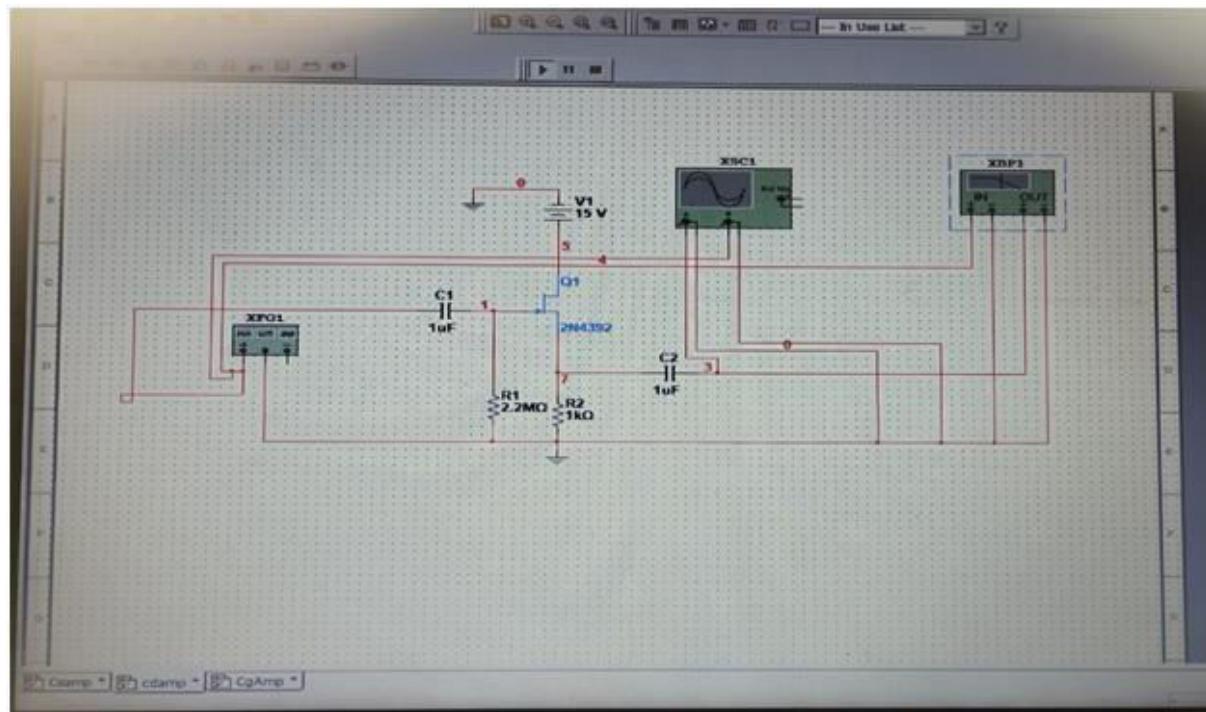
### **SELECTING OUTPUTS FOR PLOTTING:**

- Execute outputs->to be plotted->select on schematic in the simulation window.
- Follow the prompt at the bottom of the schematic window, click on output net v0, input net vin of the ed amplifier. Press esc with the cursor in the schematic after selecting node.
- Next go to adel window results->direct plot->ac db20 and it direct to schematic window, click the output nets from the schematic and press escape. The output waveform will appear.
- Tools->calculator.

- Select the waveform, the waveform detail is added in the calculator, now select average in the function panel, click evaluate the buffer and stack, now the gain value is display on the calculator window.



## OUTPUT:



**RESULT:**

Thus, the common drain amplifier has been designed and simulation of its input and output characteristics has been done successfully.

<b>Ex No:</b>	<b>Design and Simulate simple 5 transistor differential amplifier</b>
<b>Date:</b>	

### **AIM:**

To design and simulate simple 5 transistor differential amplifier by using cadence application.

### **SOFTWARE REQUIRED:**

- Pc loaded with linux os
- Cadence software

### **THEORY:**

Simulation can be defined as, simulating a chip design through software programs that use models to replicate how a device will perform in terms of timing and results.

**DIFFERENTIAL AMPLIFIER** Amplifies the difference between the input signals.

**INPUTS:** Differential input:  $V_{id} = V_{i1} - V_{i2}$  Common input:  $V_{ic} = (V_{i1} + V_{i2})/2$  **OUTPUTS:**

Differential output:  $V_{od} = V_{o1} - V_{o2}$  Common output:  $V_{oc} = (V_{o1} + V_{o2})/2$  The common mode rejection ratio (CMRR) of a differential amplifier (or other device) is a metric used to quantify the ability of the device to reject common-mode signals, i.e. those that appear simultaneously and in-phase on both inputs. An ideal differential amplifier would have infinite CMRR, however this is not achievable in practice. A high CMRR is required when a differential signal must be amplified in the presence of a possibly large common-mode input, such as strong electromagnetic interference (EMI).

**Differential Gain:**  $A_d$ . **Common Mode Gain:**  $A_c$ . **Common Mode Rejection Ratio (CMRR)\*:**  $|A_d|/|A_c|$ .

\* CMRR is usually given in dB:  $CMRR(dB) = 20 \log (|A_d|/|A_c|)$ . Advantages •

Manipulating differential signals • High input impedance • Not sensitive to temperature •

Fabrication is easier • Provides immunity to external noise • A 6 db increase in dynamic range which is a clear advantage for low voltage systems • Reduces second order harmonics Disadvantages • Lower gain • Complexity • Need for negative voltage source for proper bias Applications • Analog systems • DC amplifiers • Audio amplifiers - speakers and microphone circuits in cellphones • Servocontrol systems • Analog computers.

### **PROCEDURE:**

- Open the new terminal window and type the command given below.
- Mount -a
- Cd cadence\_db

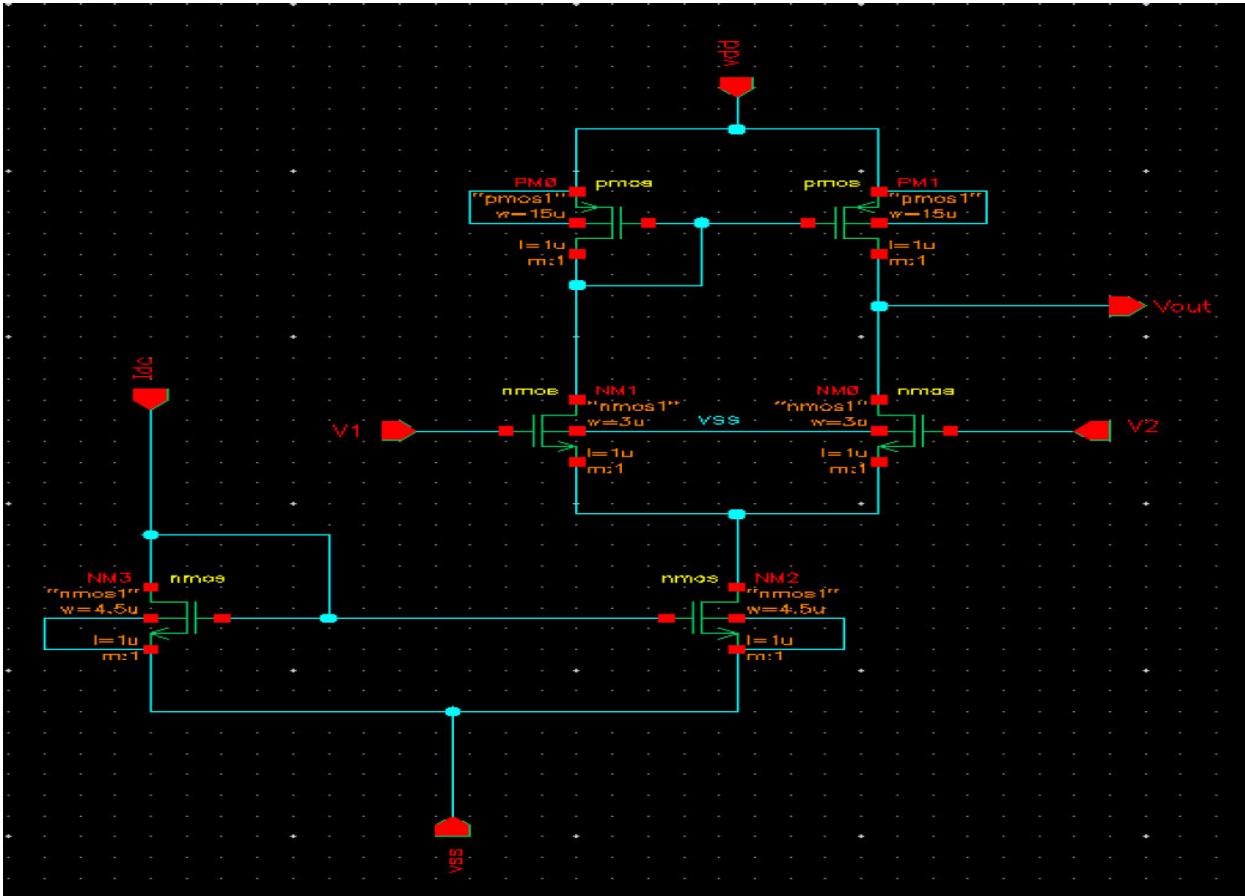
- Csh
- Source cshrc
- Cd cadence\_ms\_labs\_614
- Virtuoso
- Now two windows (what's new window and virtuoso window) are open.
- Close the what's new window.
- In the virtuoso window, go to file -> new -> library.
- Now new library window is open.
- Type the file name.
- Activate attach to existing.
- Click ok.
- Now attach library window is open.
- Choose gdk 180.
- Click ok.
- Again, in the virtuoso window, go to file->new->cell view->cell
- name (type the cell name)->ok.
- The schematic window is open.

#### **CIRCUIT DIAGRAM (COMMON MODE) SELECTING THE COMPONENTS:**

In schematic window ->create ->instance->library->browse ->gdk 180->cell view ->device to be selected->close

<b>Library name</b>	<b>Cell name</b>	<b>Properties/comments</b>
Gdk 180	Nmos	Model name =nmos 1(nmo, nm1); W=3u; l=1u
Gdk 180	Nmos	Model name=nmos1 (nm2, nm3); W=4.5u; l=1u
Gdk 180	Pmos	Model name=pmos 1(pm0, pm1); W=15u; l=1u

(type the properties as mentioned above)



- For the pin configuration, create->pin.

Pin names	Directions
Idc, v1, v2	Input
Vout	Output
Vdd,vss	Input

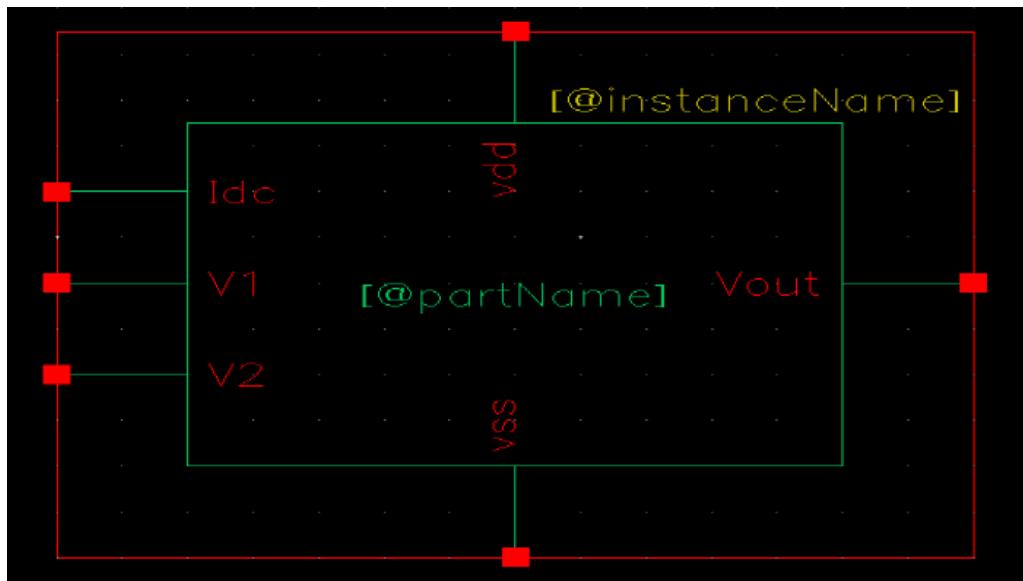
- After selecting the components needed for the design create the circuit diagram by connecting the components.
- Then click check and save (“schematic with no error” message display in virtuoso window).

### GENERATION OF THE BLOCK DIAGRAM:

- Create the cell view.
- Create->cell view->from cell view->ok.
- The cell view from cell view form appears. With the edit options

function active, you can control the appearance of the symbol to generate.

- Verify that the from view name field is set to schematic, and the to view name field is set to symbol, with the tool/data type set as schematic symbol.
- Click ok in the cell view from cell view form. The symbol generation form appears.
- Modify the pin specifications.
- Click ok in the symbol generation options form.
- A new window displays an automatically created differential amplifier symbol.
- Modifying automatically generated symbol.
- Execute create->selection box. In the add selection box form, click automatic.a new red selection box is automatically added.
- After creating symbol, click on the save icon in the symbol editor window to save the symbol. In the symboleditor,execute file->close to the close the symbol view window.



### CREATING THE DIFFERENTIAL AMPLIFIER TEST CELL VIEW:

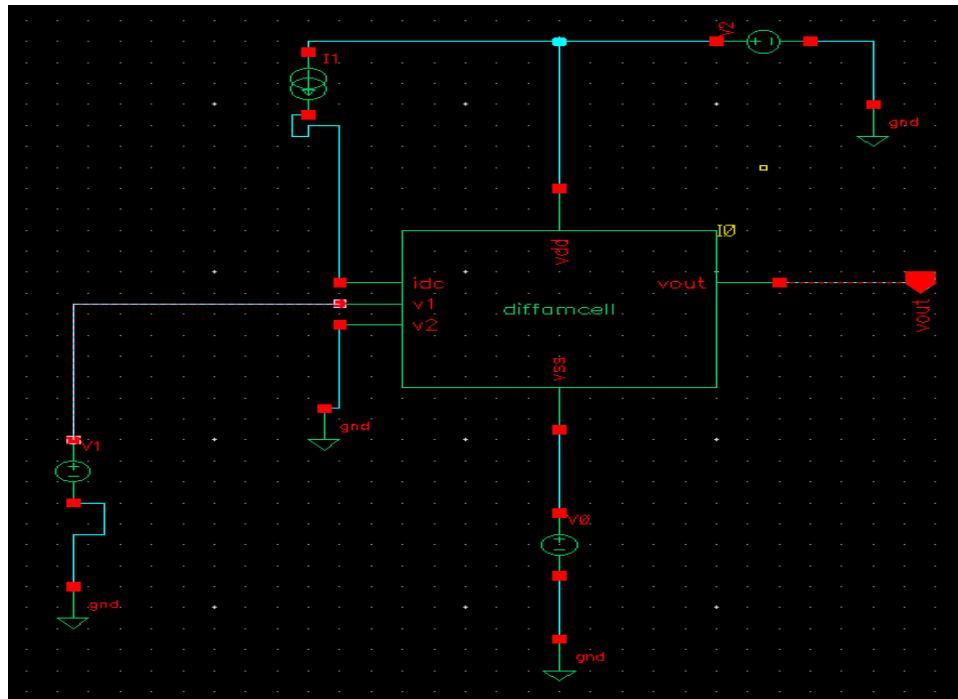
- In the CIW or library manager, execute file->new->cellview
- Setup the create new file as diff-amplifier\_test.
- Click ok when done. A blank schematic window for the diff\_amplifier\_test design appears.

## BUILDING THE DIFF\_AMPLIFIER\_TEST CIRCUIT:

- Using the component list and properties/comments in this table, Build the diff\_amplifier\_test schematic

Library name	Cell view name	Properties/comments
Mydesignlib	Diff_amplifier	Symbol
Analoglib	Vsin	Define specification as ac magnitude =1; Amplitude=5m; frequency=1k
Analoglib	Vdd, Vss, Gnd	Vdd=2.5; Vss=-2.5
Analoglib	Idc	Dc Current=30u

- Click the wire (narrow) icon and wire your schematic.
- Tip: you can also press the w key, or execute create->wire(narrow).
- Click on the check and save icon to save the design.
- The schematic should look like this
- Leave your diff\_amplifier \_test schematic window open for the next section.
- Lunch->ade l->design->model library.

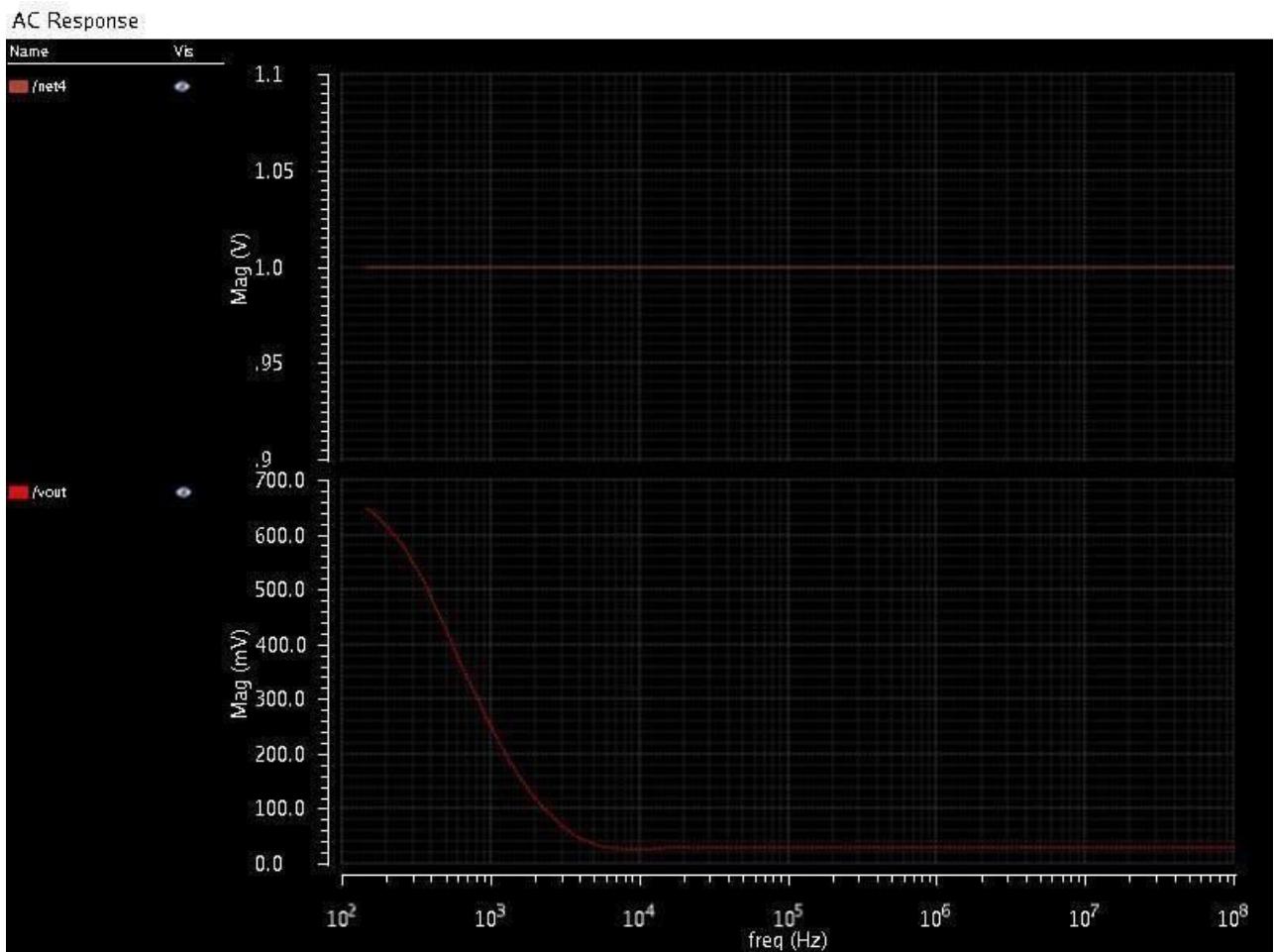
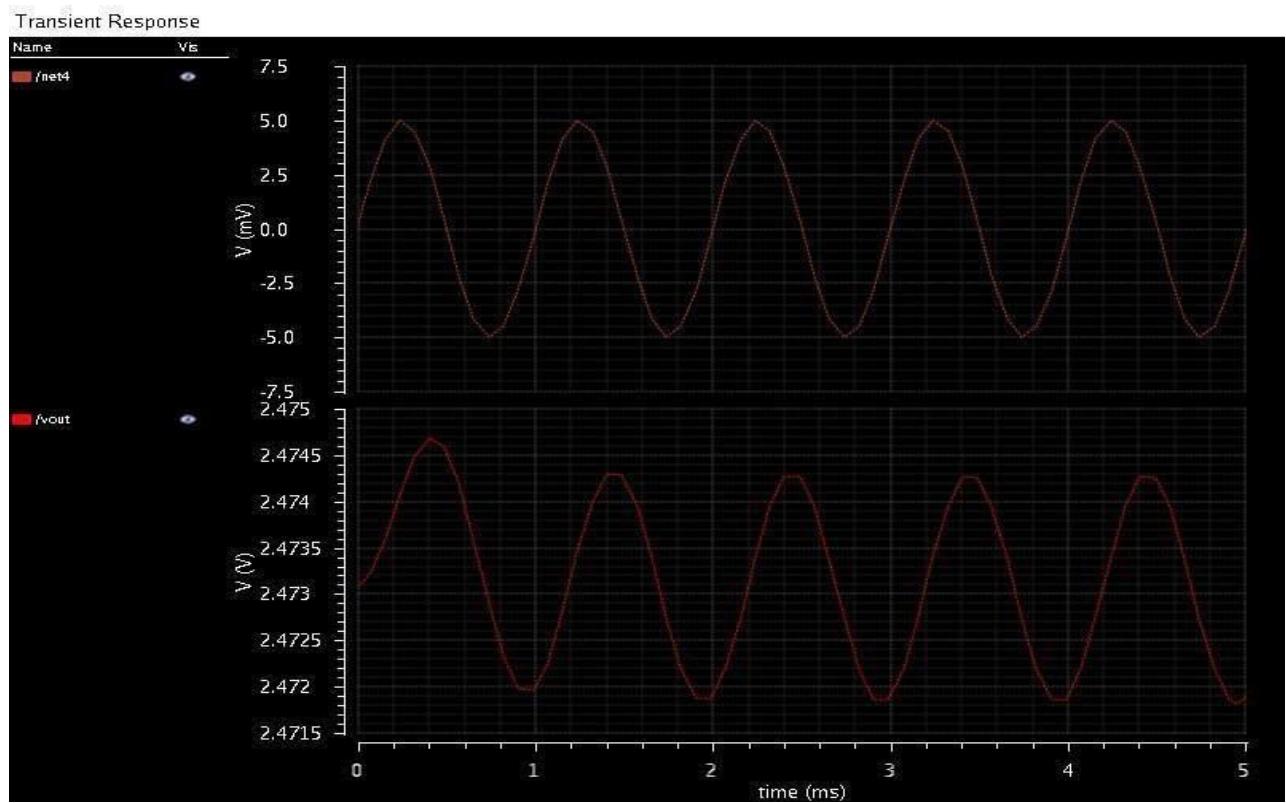


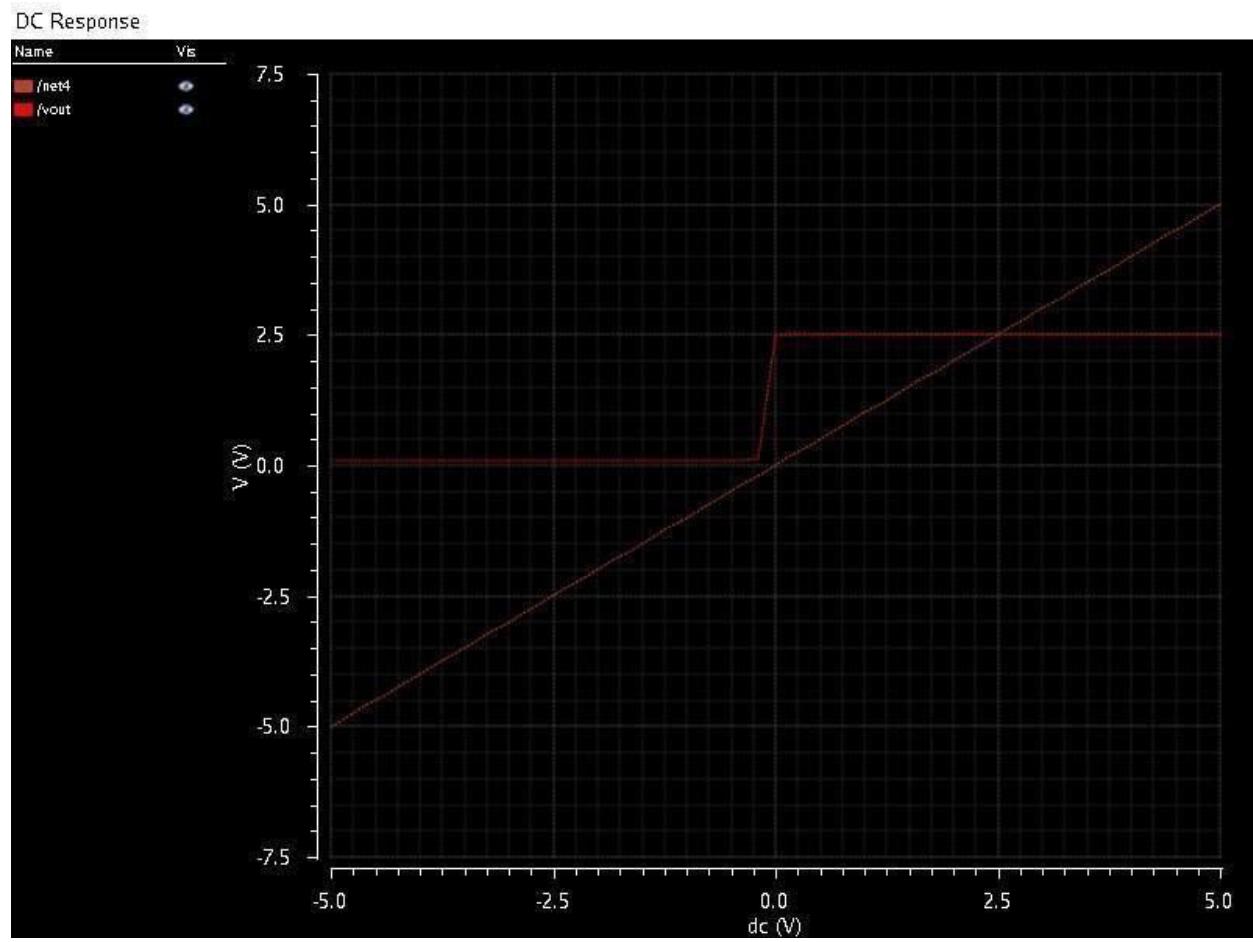
## **CHOOSING ANALYSES:**

- In the simulation window, click the choose->analyses icon.
- You can also execute analyses->choose.
- The choosing analysis form appears. This is dynamic form, the bottom of the form changes based on the selection above.
- To setup for transient analysis
- In the analysis section select tran
- Set the stop time as 5m
- Click at the moderate or enabled button at the bottom, and then click apply.
- To set up for dc analyses:
- In the analyses section, select dc.
- In the dc analyses section, turn on save dc operating point.
- Turn on the component parameter.
- Double click the select component, which takes you to the schematic window.
- Type the parameter name as “dc”.
- Select input signal vsin dc analysis
- In the analysis form, select start and stop voltages as -5 to 5 respectively.
- Check the enable button and then click apply.
  
- To set up for ac analyses form is shown in the previous page.
- In the analyses section, select ac
- In the ac analyses section, turn on frequency.
- In the sweep range section select start and stop frequencies as 150 to 100m
- Select points per decade as 20
- Check the enable button and then click apply.
- Click ok in the choosing analyses form

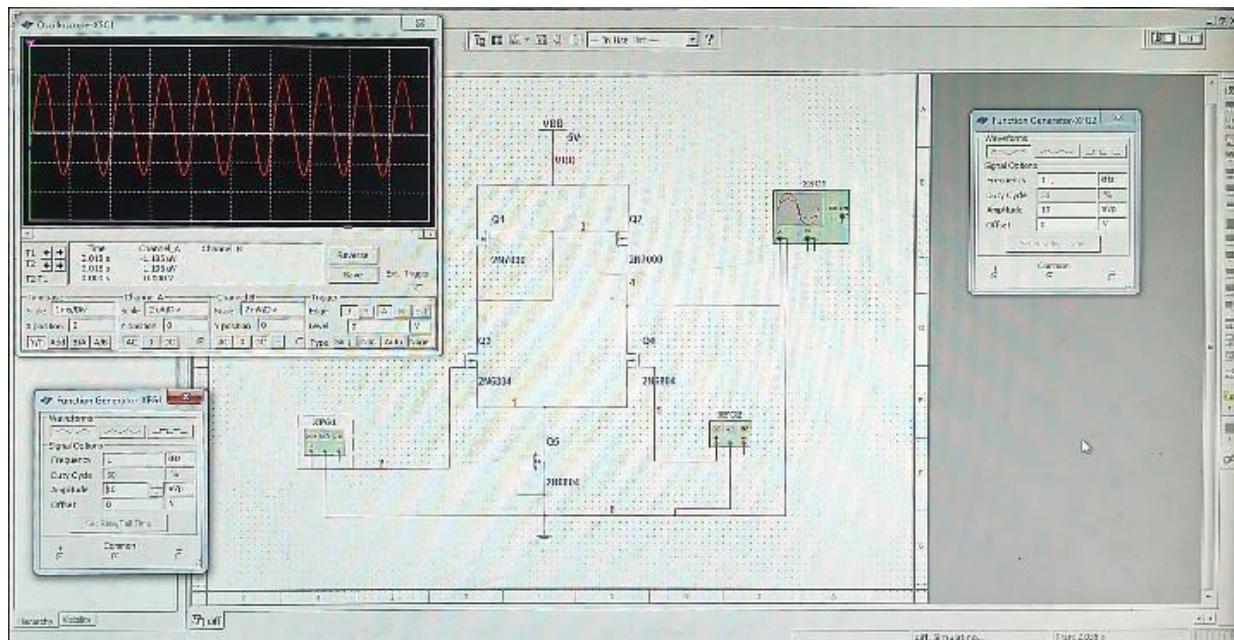
## **SELECTING OUTPUTS FOR PLOTTING:**

- Execute outputs->to be plotted->select on schematic in the simulation window.
- Follow the prompt at the bottom of the schematic window, click on output net v0, input net vin of the diff\_amplifier. Press esc with the cursor in the schematic after selecting node.
- Next go to adel window results->direct plot->ac db20 and it direct to schematic window, click the output nets from the schematic and press escape. The output waveform will appear.
- Tools->calculator.
- Select the waveform, the waveform detail is added in the calculator, now select average in the function panel, click evaluate the buffer and stack, now the differential gain value is display on the calculator window.





**OUTPUT :**



**RESULT:**

Thus, the differential amplifier has been designed and the gain and CMRR has been calculated.