Lab Report On VLSI Design

Submitted by:

Rounak Goswami

Reg. No.:-**20215105 --EC-3**

B.Tech (VI SEM)

VLSI Design Lab

SUBJECT CODE (EC-16203)

Session: 2023-2024

Submitted to:

Prof. Sanjeev Rai and
Dr. V.Karupannan

DEPARTMENT OF **ELECTRONICS AND COMMUNICATION ENGINEERING**

MOTILAL NEHRU NATIONAL INSTITUTE OF TECHNOLGY
ALLAHABAD PRAYAGRAJ-211004, INDIA

05/02/24

EXPERIMENT - 1

- **AIM:-** To design and simulate AND, OR and NOT gates using gate level, data flow and behavioural modelling

- **SOFTWARE USED:-** AMD Vivado 2023.2

- **THEORY:-**

OR GATE-

The OR Gate Output attains high (1) state whenever one of the inputs are high, otherwise it remains low (0).

$$Y = A + B$$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

AND GATE-

The AND Gate Output remains low (0) state whenever one of the inputs are low, otherwise it attains high (1) state.

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

NOT GATE-

The NOT Gate Output attains high (1) state whenever the input is low, and vice versa

A	Y
0	1
0	0

- **VERILOG CODES:-**

✓ **DATA FLOW MODELLING:-**

1) **AND GATE:**

```
module andGate(input A,input B,output Y);
assign Y = A & B;
endmodule
```

2) **OR GATE:**

```
module orGate(input A,input B,output Y);
assign Y = A ^ B;
endmodule
```

3) **NOT GATE:**

```
module notGate(input A,output Y);
assign Y = ~A;
endmodule
```

✓ **GATE LEVEL MODELLING:-**

1) **AND GATE:**

```
module andGate(input A,input B,output Y);
and(Y,A,B);
endmodule
```

2) OR GATE:

```
module orGate(input A,input B,output Y);
or(Y,A,B);
endmodule
```

3) NOT GATE:

```
module notGate(input A,output Y);
not(Y,A);
endmodule
```

✓ **BEHAVIOURAL MODELLING:-**

1) AND GATE:

```
module andGate(input A,input B,output Y);
initial
begin
Y = 0;
end
always @(A or B)
case({A,B})
0: begin Y=0; end
1: begin Y=0; end
2: begin Y=0; end
3: begin Y=1; end
endcase
endmodule
```

2) OR GATE:

```
module orGate(input A,input B,output Y);
initial
begin
Y = 0;
end
always @(A or B)
case({A,B})
0: begin Y=0; end
1: begin Y=1; end
2: begin Y=1; end
3: begin Y=1; end
endcase
endmodule
```

3) NOT GATE:

```
module notGate(input A,output Y);
initial
begin
Y = 0;
end
always @(A or B)
case({A})
0: begin Y=1; end
1: begin Y=0; end
endcase
endmodule
```

- **TESTBENCH:-**

1) AND GATE:-

```
module testBench;
reg A,B;
wire Y;
andGate uut(.A(A),.B(B),.Y(Y));
initial
        begin
        A = 0; B = 0; #100;
        A = 0; B = 1; #100;
        A = 1; B = 0; #100;
        A = 1; B = 1; #100;
        end
endmodule
```

2) <u>OR GATE:-</u>

```
module testBench;
reg A,B;
wire Y;
orGate uut(.A(A),.B(B),.Y(Y));
initial
        begin
        A = 0; B = 0; #100;
        A = 0; B = 1; #100;
        A = 1; B = 0; #100;
        A = 1; B = 1; #100;
        end
endmodule
```

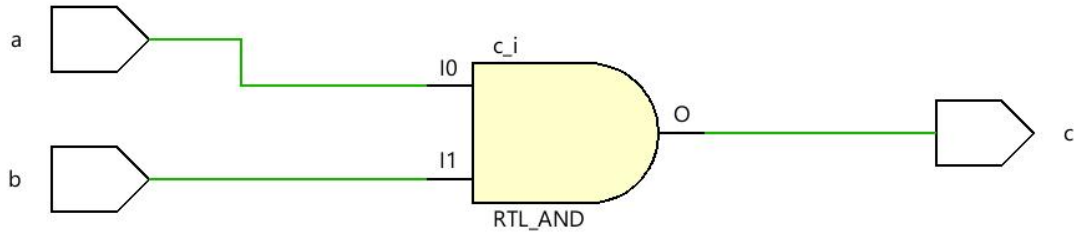
3) <u>NOT GATE:-</u>

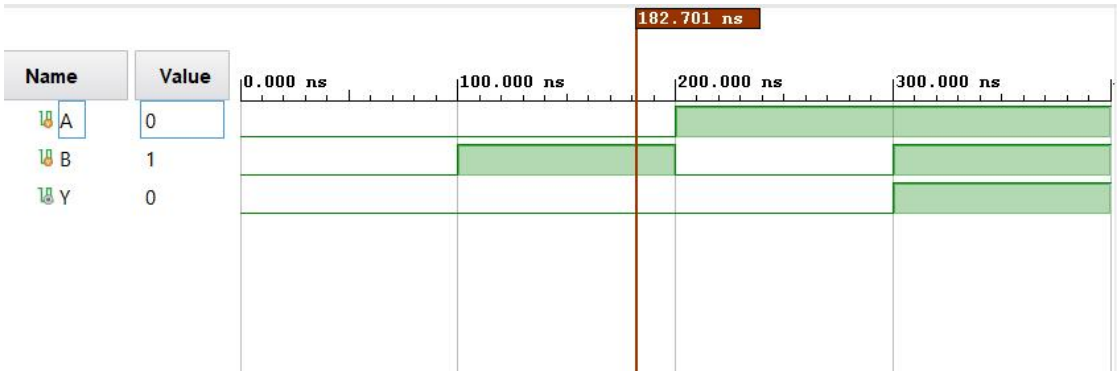
```
module testBench;
reg A;
wire Y;
notGate uut(.A(A),.Y(Y));
```

```
initial
        begin
        A = 0; #100;
        A = 1; #100;
        end
endmodule
```

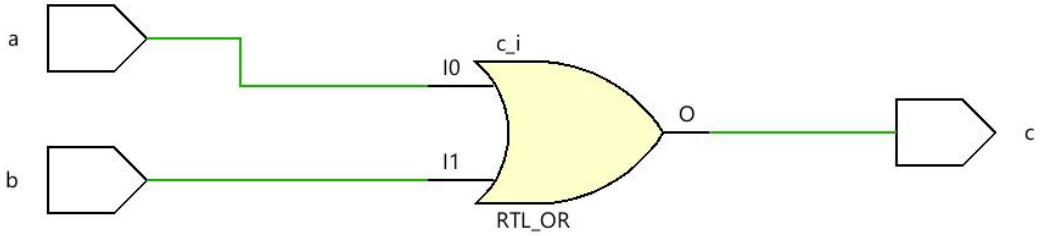
- **OUTPUT WAVEFORMS:-**

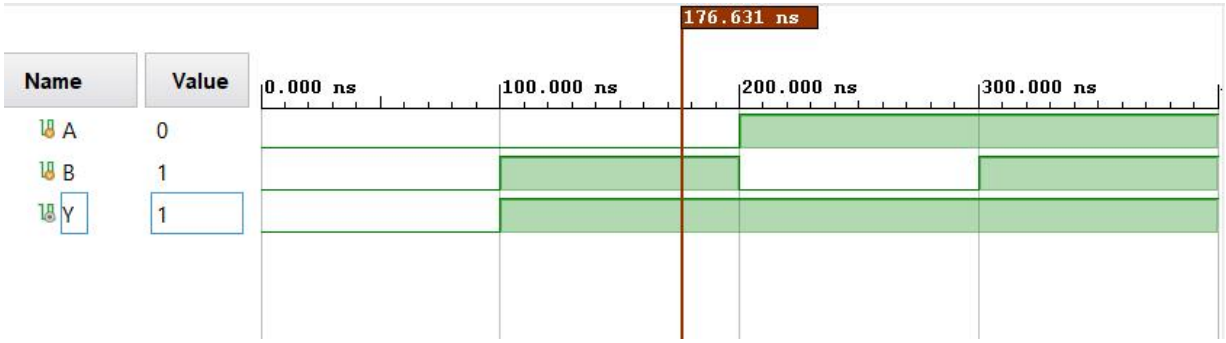
1) AND GATE:-



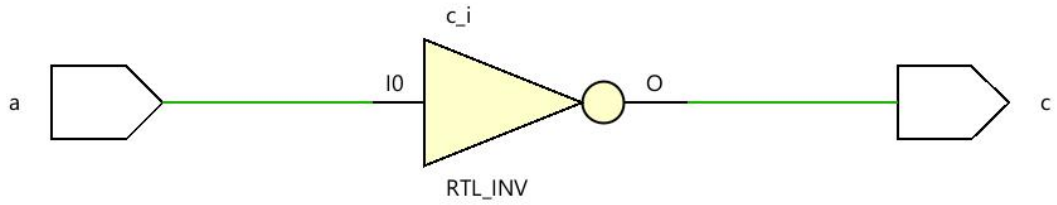


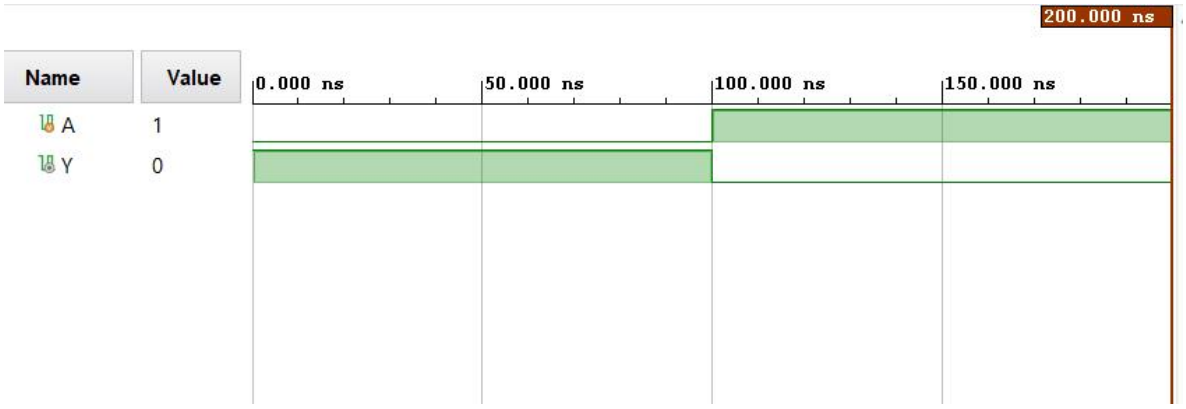
2) OR GATE:-





3) NOT GATE:-





● RESULT :-

All the basic gates, i.e., AND,OR and NOT Gates have been implemented in gate level, data flow and behavioural modelling.

● APPLICATIONS:-

1) Logic Gates are the fundamental blocks in microcontrollers and microprocessors.

2) Logic Gates are the building blocks of any complex digital system design.

12/02/24

EXPERIMENT - 2

- **AIM:-** To design and simulate Half Adder and Full Adder using Gate Level Modelling

- **SOFTWARE USED:-** AMD Vivado 2023.2

- **THEORY:-**

Half Adder is a combinational arithmetic circuit that adds two numbers and produces a sum bit (sum) and a carry bit (carry) as an output. If A and B are the input bits, then sum bit (sum) is the XOR of A and B and carry bit (carry) will be AND of A and B. From this it is clear that a half adder circuit can be easily constructed using one XOR Gate and one AND Gate.

A	B	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

sum = A (xor) B
carry = A (and) B

Full Adder is the adder that adds three inputs and produces two outputs. The first two inputs are A and B and the third input is an input carry as C. The output carry is designated as (carry) and the sum output is designated as (sum).

A	B	C	sum	carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1

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

sum = A (xor) B (xor) C
Carry = A.B + B.C + A.C

- **VERILOG CODES:-**

1) HALF ADDER:-

```
module halfAdder(
        input a,
        input b,
        output s,
        output c
        );

xor(s,a,b);
and(c,a,b);
endmodule
```

2) FULL ADDER:-

```
module fullAdder(
        input a,
        input b,
        input c,
        input sum,
        input carry
        );

xor(sum,a,b,c);
and(x,a,b);
```

```
and(y,b,c);
and(z,a,c);
or(carry,x,y,z);
endmodule
```

- **TESTBENCH:-**

1) HALF ADDER:-

```
module halfAddertestbench;
        reg a,b;
        wire s,c;

        halfAdder uut(.a(a),.b(b),.s(s),.c(c));

        initial begin
        a = 0; b = 0;
        #100
        a = 0; b = 1;
        #100
        a = 1; b = 0;
        #100
        a = 1; b = 1;
        #100
        $finish;
        end
endmodule
```

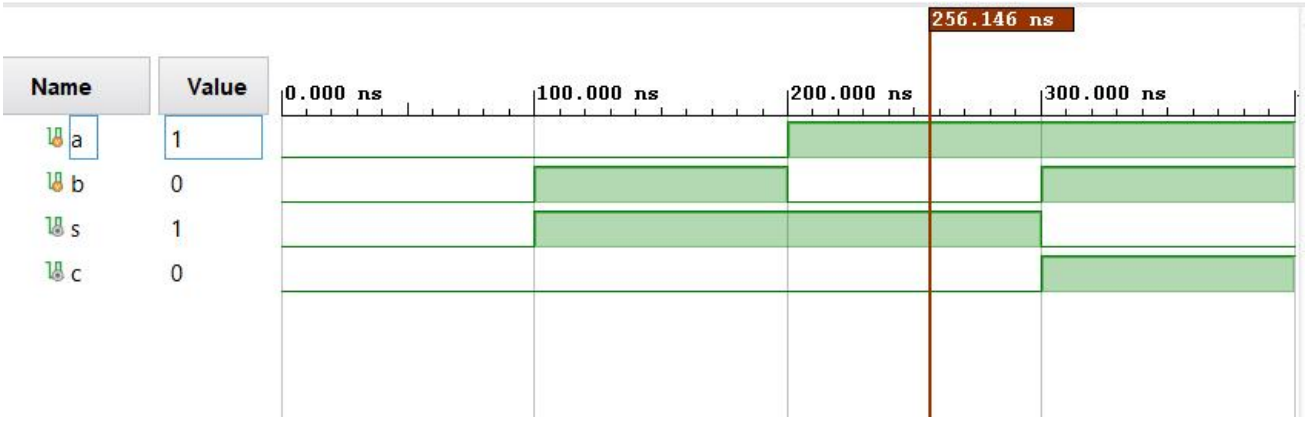
2) FULL ADDER:-

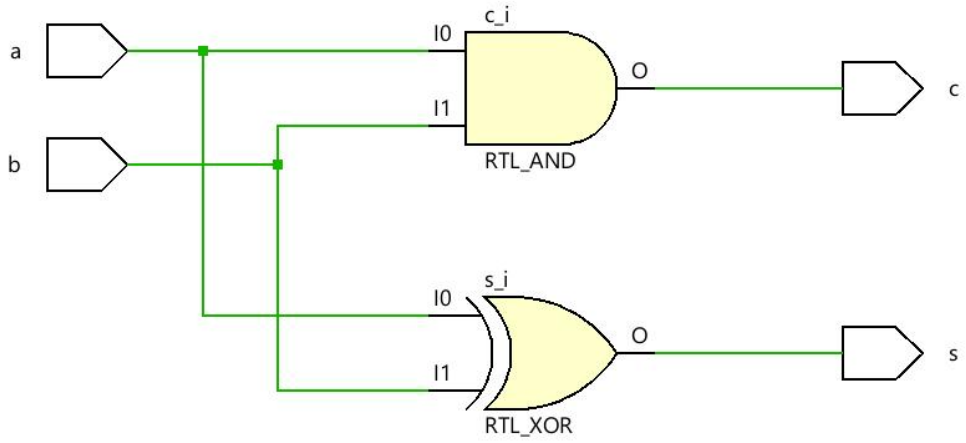
```
module fullAddertestbench;
        reg a,b,c;
```

```
        wire sum,carry;

        fullAdder uut(.a(a),.b(b),.c(c),.sum(sum),.carry(carry));

        initial begin
        a = 0; b = 0; c = 0;
        #100
        a = 0; b = 0; c = 1;
        #100
        a = 0; b = 1; c = 0;
        #100
        a = 0; b = 1; c = 1;
        #100
        a = 1; b = 0; c = 0;
        #100
        a = 1; b = 0; c = 1;
        #100
        a = 1; b = 1; c = 0;
        #100
        a = 1; b = 1; c = 1;
        #100
        $finish;
        end
endmodule
```

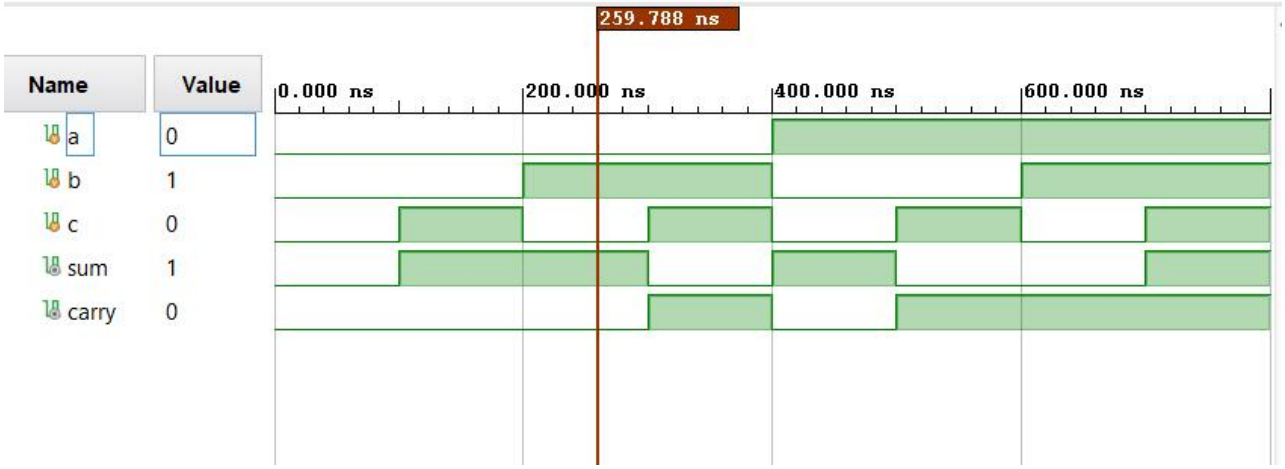
- **OUTPUT WAVEFORMS:-**

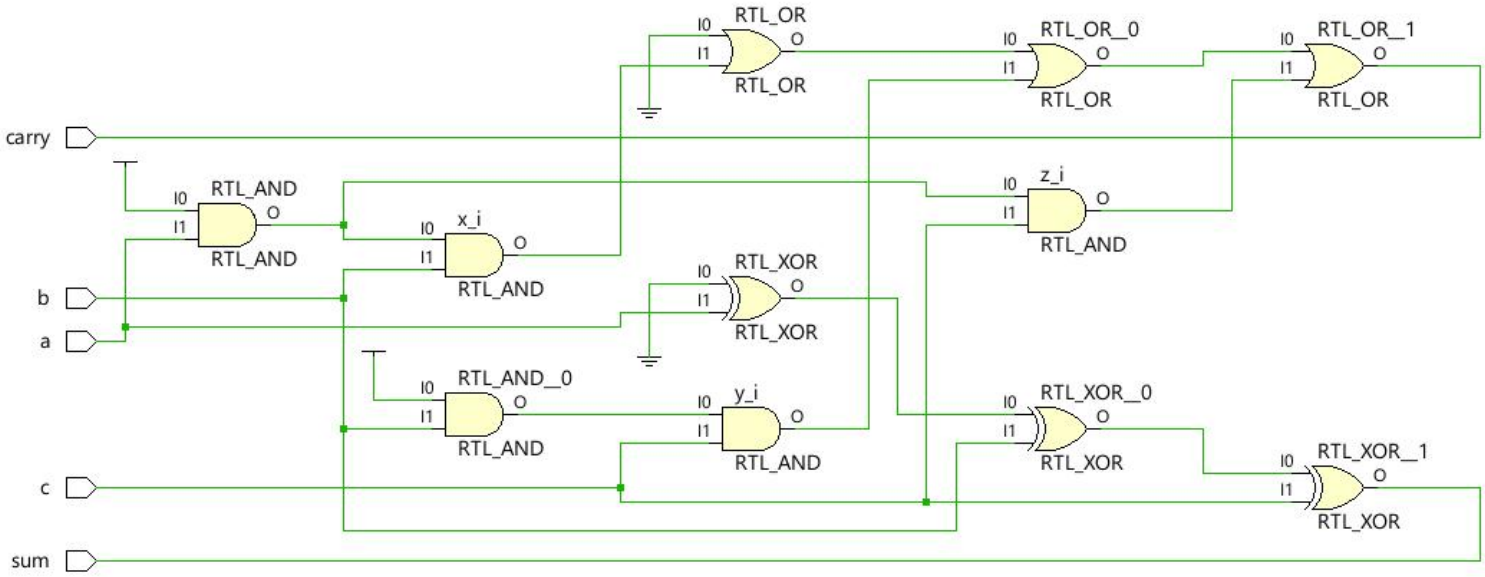
1) HALF ADDER:-





2) FULL ADDER:-





- **RESULT:**

Half Adder and Full Adder circuits has been designed and simulated using gate-level modelling technique and the output waveforms have been observed.

12/02/24

EXPERIMENT - 3

- **AIM:-** To design and simulate Half Subtractor and Full Subtractor using Gate Level Modelling

- **SOFTWARE USED:-** AMD Vivado 2023.2

- **THEORY:-**

Half Subtractor is a combinational arithmetic circuit that subtracts two numbers and produces a diff bit (D) and a borrow bit (Bout) as an output. If A and B are the input bits, then diff bit (D) is the XOR of A and B and borrow bit (Bout) will be AND of ~A and B. From this it is clear that a half adder circuit can be easily constructed using one XOR Gate, one NOT GATE and one AND Gate.

A	B	D	Bout
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

D = A (xor) B
Bout = ~A (and) B

Full Subtractor is the subtractor that performs subtraction of two bits, one is minuend and the other is subtrahend, taking into account the borrow of the previous adjacent lower minuend bit. The three inputs A,B and C, denote the minuend , subtrahend, and previous borrow, respectively. The two outputs are represented as difference(D) and borrow (Bout) bits.

A	B	C	D	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

D = A (xor) B (xor) C
Bout = ~A.B + B.C + ~A.C

- **VERILOG CODES:-**

1) HALF SUBTRACTOR:-

```
module halfSubtractor(
        input a,
        input b,
        output diff,
        output borrow
        );

xor(diff,a,b);
not(notA,a);
and(borrow,notA,b);
endmodule
```

2) FULL SUBTRACTOR:-

```
module fullSubtractor(
        input a,
        input b,
```

```
        input c,
        output diff,
        output borrow
        );

xor(diff,a,b,c);
not(notA,a);
and(x,notA,b);
and(y,b,c);
and(z,notA,c);
or(borrow,x,y,z);
Endmodule
```

- **TESTBENCH:-**

1) HALF SUBTRACTOR:-

```
module testbench;
        reg a,b;
        wire s,c;

        halfSubtractor uut(.a(a),.b(b),.diff(s),.borrow(c));

        initial begin
        a = 0; b = 0;
        #100
        a = 0; b = 1;
        #100
        a = 1; b = 0;
        #100
        a = 1; b = 1;
        #100
        $finish;
        end
```

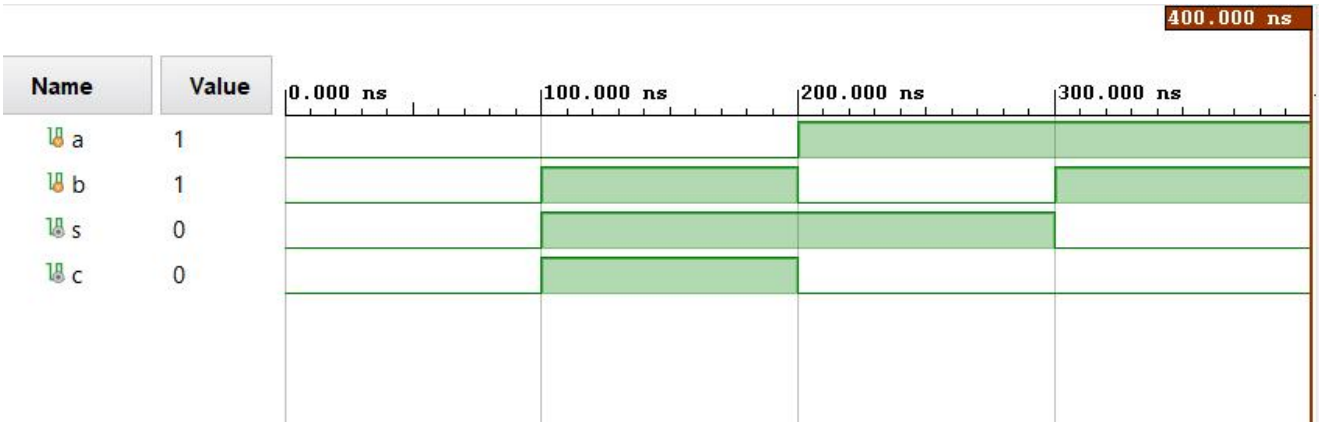
```
endmodule
```

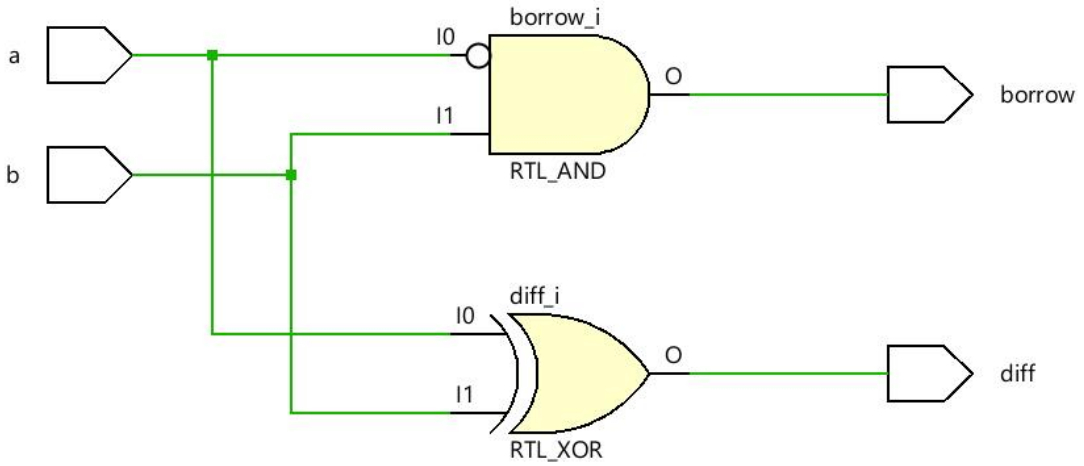
2) **FULL SUBTRACTOR:-**

```
module testbench;
	reg a,b,c;
	wire diff,borrow;

	fullSubtractor
uut(.a(a),.b(b),.c(c),.diff(diff),.borrow(borrow));

	initial begin
	a = 0; b = 0; c = 0;
	#100
	a = 0; b = 0; c = 1;
	#100
	a = 0; b = 1; c = 0;
	#100
	a = 0; b = 1; c = 1;
	#100
	a = 1; b = 0; c = 0;
	#100
	a = 1; b = 0; c = 1;
	#100
	a = 1; b = 1; c = 0;
	#100
	a = 1; b = 1; c = 1;
	#100
	$finish;
	end
endmodule
```

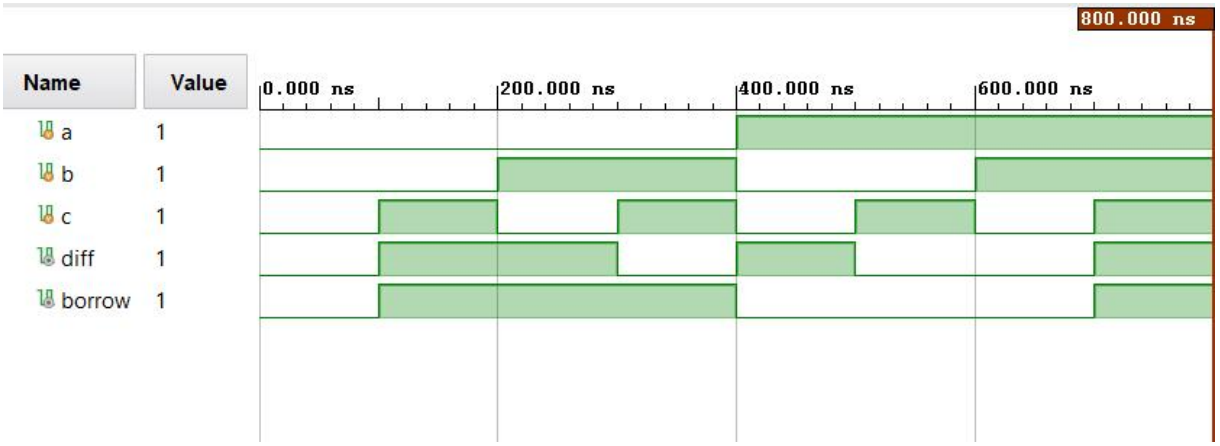
- **OUTPUT WAVEFORMS:-**

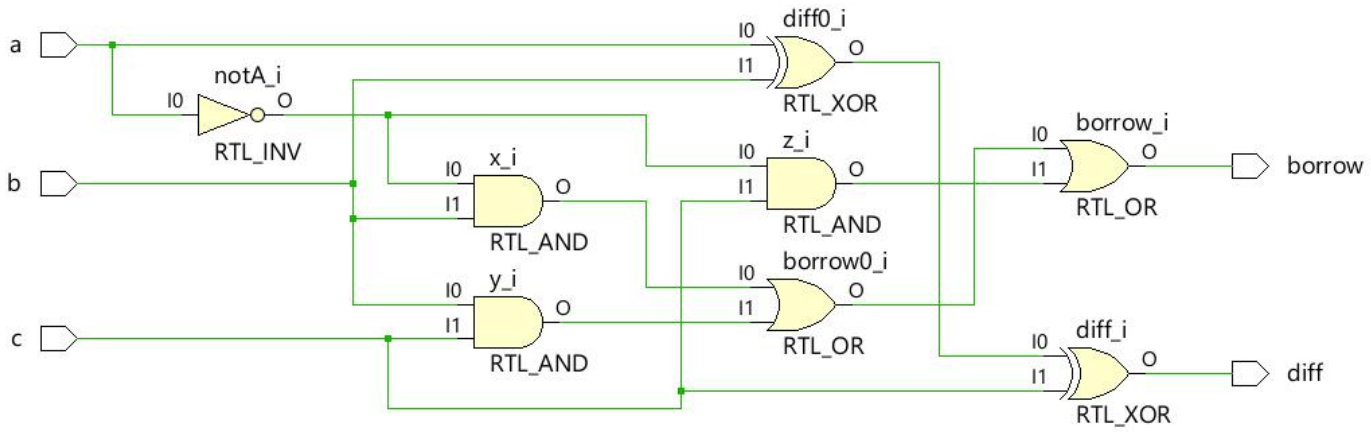
1) HALF SUBTRACTOR:-





2) FULL SUBTRACTOR:-





- **RESULT:**

Half Subtractor and Full Subtractor circuits has been designed and simulated using gate-level modelling technique and the output waveforms have been observed.

EXPERIMENT-4

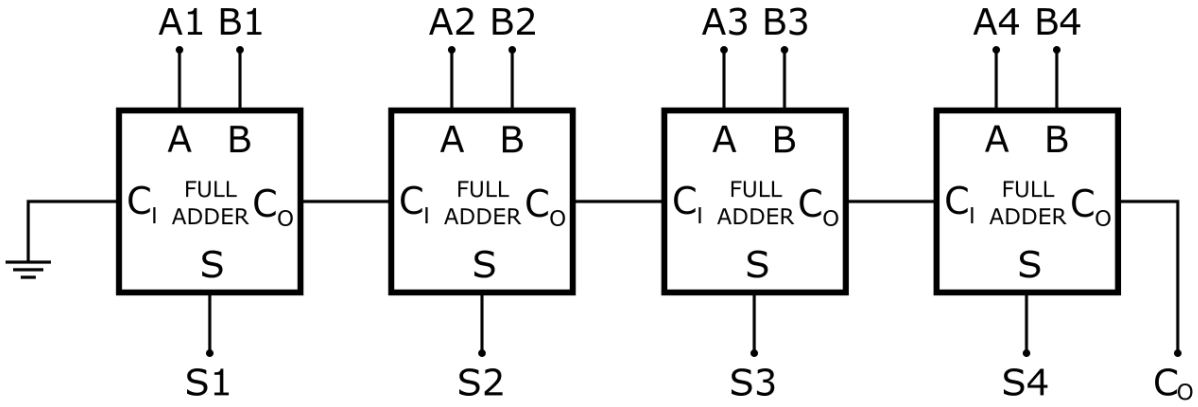
- **AIM:-** To design and simulate 4-bit Parallel Adder and Subtractor using Full Adder and Full Subtractor

- **SOFTWARE USED:-** AMD Vivado 2023.2

- **THEORY:-**

1) 4-Bit Parallel Adder:-

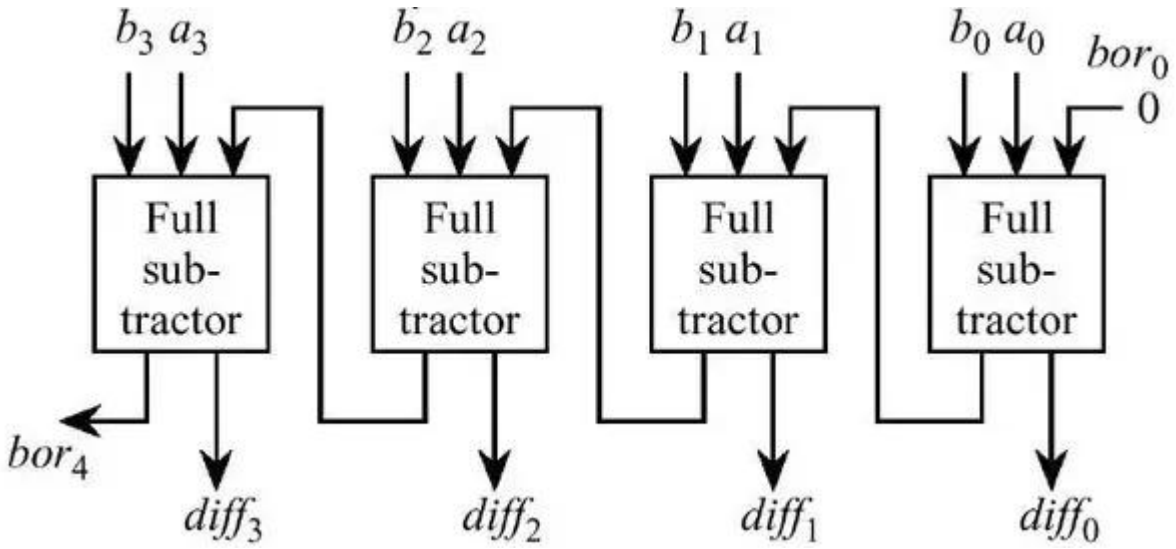
A 4-Bit Parallel Adder takes two 4-bit numbers as input along with a carry bit, and gives the result as a 4-bit number along with a final generated carry bit. It utilizes 4 Full-Adder Blocks and successively adds on the carry generated from the LSB bits towards the MSB using full adders.



A = (A4A3A2A1), B = (B4B3B2B1), S = (S4S3S2S1)

2) 4-Bit Parallel Subtractor :-

A 4-Bit Parallel Subtractor is exactly similar to the 4-Bit Parallel Adder, with the only difference, that instead of the full adder blocks, it utilizes the corresponding full subtractor blocks, and successively adds the borrow generated from the LSB bits towards the MSB using the full subtractors.



A = (a3a2a1a0), B = (b3b2b1b0), D = (diff3diff2diff1diff0)

- **VERILOG CODES:-**

1) 4-Bit Parallel Adder:-

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

module parallelAdder(
        input[3:0] a,
        input[3:0] b,
        input c,
        output[3:0] sum,
        output[3:0] carry
);
```

```
fullAdder fa1(a[0],b[0],c,sum[0],carry[0]);
fullAdder fa2(a[1],b[1],carry[0],sum[1],carry[1]);
fullAdder fa3(a[2],b[2],carry[1],sum[2],carry[2]);
fullAdder fa4(a[3],b[3],carry[2],sum[3],carry[3]);
endmodule
```

2) 4-bit Parallel Subtractor:-

```
module fullSubtractor(
        input a,
        input b,
        input c,
        output diff,
        output borrow
        );
assign diff = a ^ b ^ c;
assign borrow = (~a&b) | (~a&c) | (b&c);
endmodule

module parallelSubtractor(
        input[3:0] a,
        input[3:0] b,
        input c,
        output[3:0] diff,
        output[3:0] borrow
);

fullSubtractor fs1(a[0],b[0],c,diff[0],borrow[0]);
fullSubtractor fs2(a[1],b[1],borrow[0],diff[1],borrow[1]);
fullSubtractor fs3(a[2],b[2],borrow[1],diff[2],borrow[2]);
fullSubtractor fs4(a[3],b[3],borrow[2],diff[3],borrow[3]);
endmodule
```

- **TESTBENCH:-**

1) 4-bit Parallel Adder:-

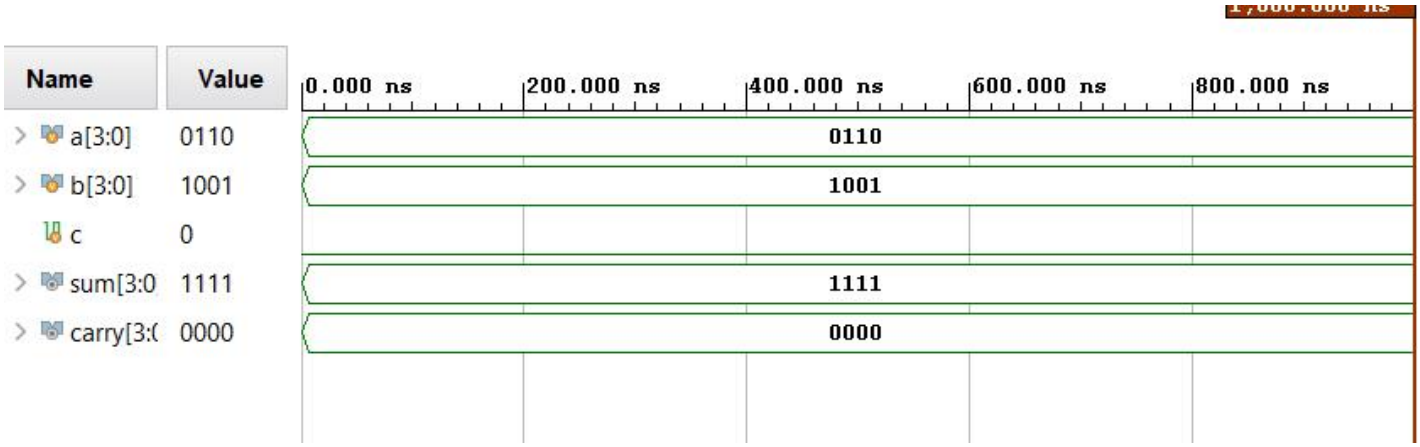
```
module testbench;
reg [3:0]a;
reg [3:0]b;
reg c;
wire [3:0]sum;
wire [3:0]carry;

parallelAdder
uut(.a(a),.b(b),.c(c),.sum(sum),.carry(carry));
initial begin
a[0] = 0;a[1] = 1;a[2] = 1;a[3] = 0;
b[0] = 1;b[1] = 0;b[2] = 0;b[3] = 1;
c = 0;
end
endmodule
```

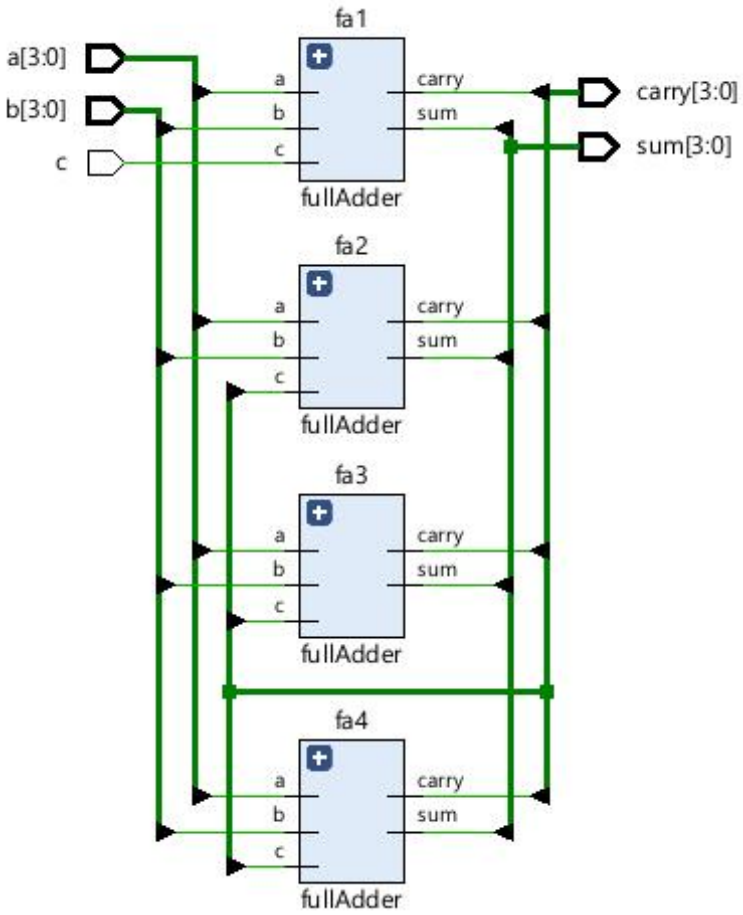
2) 4-bit Full Subtractor:-

```
module testbench;
reg [3:0]a;
reg [3:0]b;
reg c;
wire [3:0]diff;
wire [3:0]borrow;

parallelSubtractor
uut(.a(a),.b(b),.c(c),.diff(diff),.borrow(borrow));
initial begin
a[0] = 0;a[1] = 1;a[2] = 1;a[3] = 0;
b[0] = 1;b[1] = 0;b[2] = 0;b[3] = 1;
c = 0;
end
endmodule
```

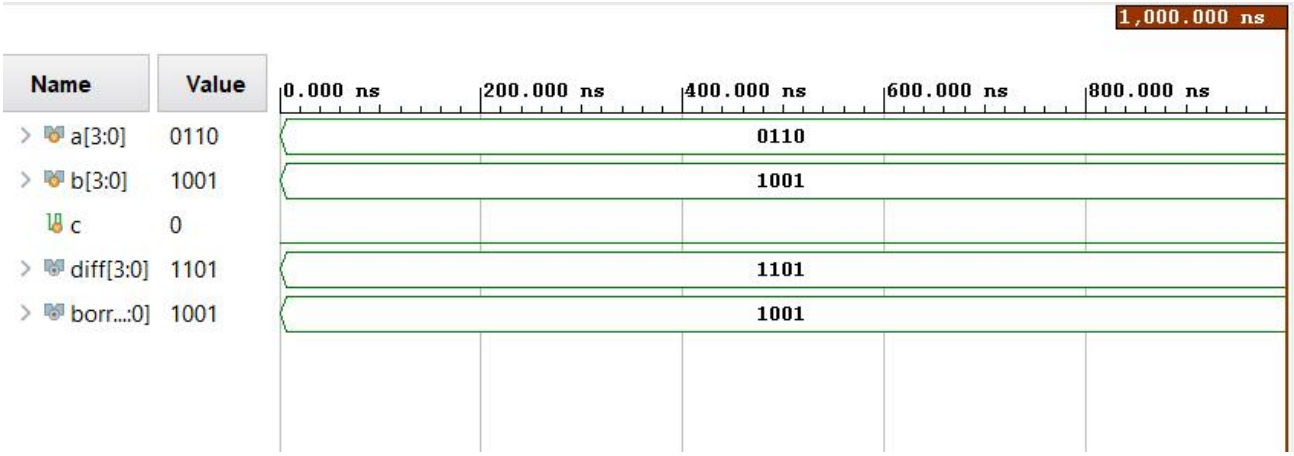
- **OUTPUT WAVEFORMS:-**

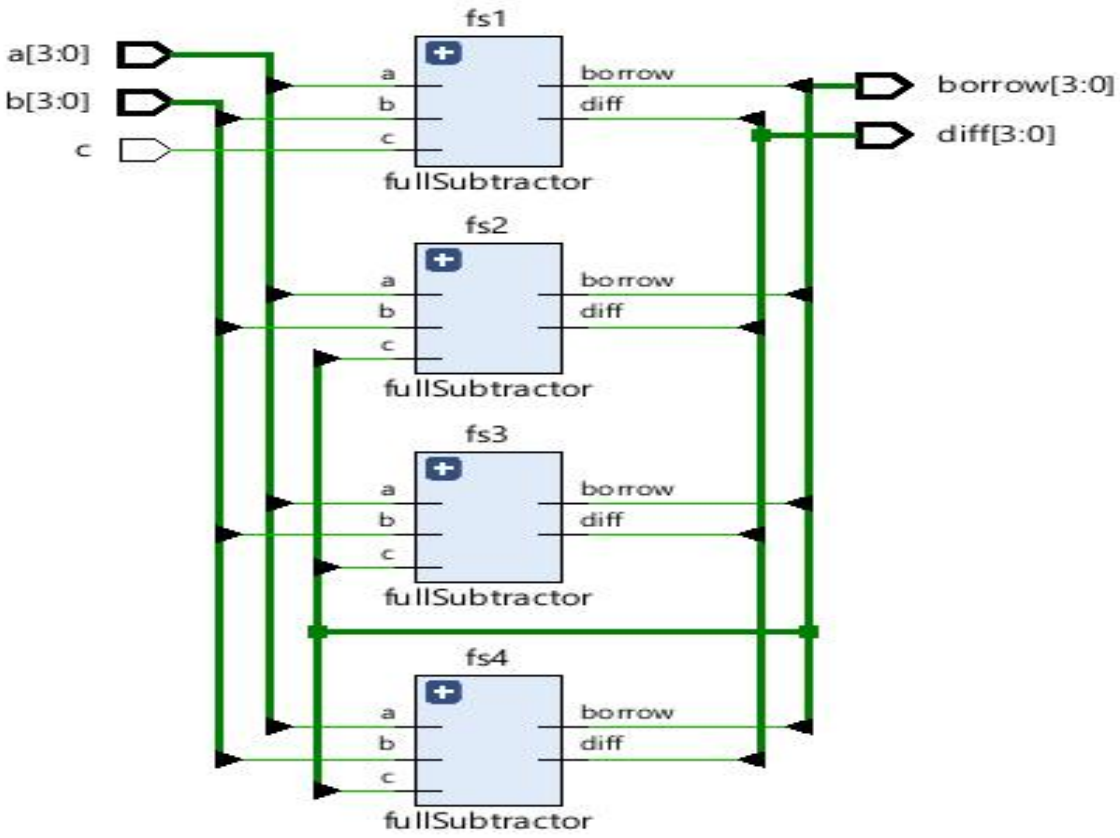
1) 4-bit Parallel Adder:-





2) 4-bit Parallel Subtractor:-





- **RESULT:-** 4-bit Parallel Adder and Subtractor have been implement using Gate Level Modelling, and the corresponding waveform and RTL Schematic Diagram have been observed.

01/04/24 **EXPERIMENT- 5**

- **AIM:-** To design and simulate T-Flip Flop, D-Flip Flop and SR-Flip Flop

- **SOFTWARE USED:-** AMD Vivado 2023.2

- **THEORY:-**

1) Truth Table of T- Flip Flop :-

T	Q_{n+1}
0	Q_n(No Change)
1	$\overline{Q_n}$(Toggles)

2) Truth Table of D- Flip Flop :-

CLK	D	Q(n+1)	State
–	0	0	RESET
–	1	1	SET

3) Truth Table of SR- Flip Flop :-

S	R	Q_{n+1}
0	0	Q_n(No Change)
0	1	0
1	0	1
1	1	x

4) Truth Table of JK- Flip Flop :-

J	K	Q_{n+1}
0	0	Q_n(No Change)
0	1	0
1	0	1
1	1	$\overline{Q_n}$(Toggles)

● VERILOG CODES:-

1) T-Flip Flop:-

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

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

2) D-Flip Flop:-

```
module dff(input d,input clk,input rstn,output reg q);
   always @(posedge clk)
      if(!rstn)
         q <= 0;
    else
         q <= d;
endmodule
```

3) SR Flip Flop :-

```
module srFlipFlop(input s,input r,input clk,input rstn,output reg q,output qbar);

   always @(posedge clk)// Synchronous reset
      begin
         if(!rstn)
            q <= 0;
         else
            begin
               case({s,r})
                  2'b00 : q <= q; // No change
                  2'b01 : q <= 1'b0; // Reset
                  2'b10 : q <= 1'b1; // Set
                  2'b11 : q <= 1'bx; // Invalid
               endcase
            end
      end
   assign qbar = ~q;
endmodule
```

4) JK Flip Flop :-

```
module jkflipFlop(input j,input k,input clk,output reg q);
   always @(posedge clk)
      begin
         case({j,k})
            2'b00 : q <= q; // No change
            2'b01 : q <= 0; // Reset
            2'b10 : q <= 1; // Set
            2'b11 : q <= ~q; // Complement
         endcase
```

```
        end
endmodule
```

- **TESTBENCH:-**

1) T- Flip Flop:-

```
module tb;
    reg clk;
    reg rstn;
    reg t;
//    reg q;
    reg [4:0] dly; // Declaration of delay block
    tff u0(.clk(clk),.rstn(rstn),.t(t),.q(q));
    always #5 clk = ~clk;

    initial begin
//        $dumpfile("dump.vcd"); $dumpvars;
        {rstn,clk,t} <= 0;

        $monitor ("T=%0t rstn=%0b t=%0d q=%0d", $time,
rstn, t, q);
        repeat(2) @(posedge clk);
        rstn <= 1;

        for(integer i = 0 ; i < 20 ; i = i + 1) begin
            dly = $random;
            #dly t <= $random;
        end
        #20 $finish;
    end
endmodule
```

2) D-Flip Flop :-

```
module tb;
    reg clk;
    reg d;
    reg rstn;
    reg[2:0] delay;

    dff u0(.d(d),.rstn(rstn),.clk(clk),.q(q));

    always #5 clk = ~clk;

    initial begin
//        $dumpfile("dump.vcd"); $dumpvars;
        {clk, rstn, d} <= 0;

        #15 d <= 1;
        #10 rstn <= 1;
        for(integer i = 0 ; i < 5 ; i = i + 1) begin
            delay = $random;
            #delay d <= i;
        end
        #10 $finish;
    end
endmodule
```

3) <u>SR Flip Flop :-</u>

```
module tb();
    reg s;
    reg r;
    reg clk;
    reg rstn;
    wire q,qbar;
```

```
    srFlipFlop
dff(.s(s), .r(r), .clk(clk), .rstn(rstn), .q(q), .qbar(qbar));

    always #5 clk = ~clk;
    initial begin
        {clk, rstn} <= 0;

        #5 rstn = 1;

        drive(2'b00);
        drive(2'b01);
        drive(2'b10);
        drive(2'b11);
        #5;
        $finish;
    end

    task drive(input [1:0] ip);
        begin
                @(posedge clk);
                {s,r} = ip;
        end
    endtask
endmodule
```

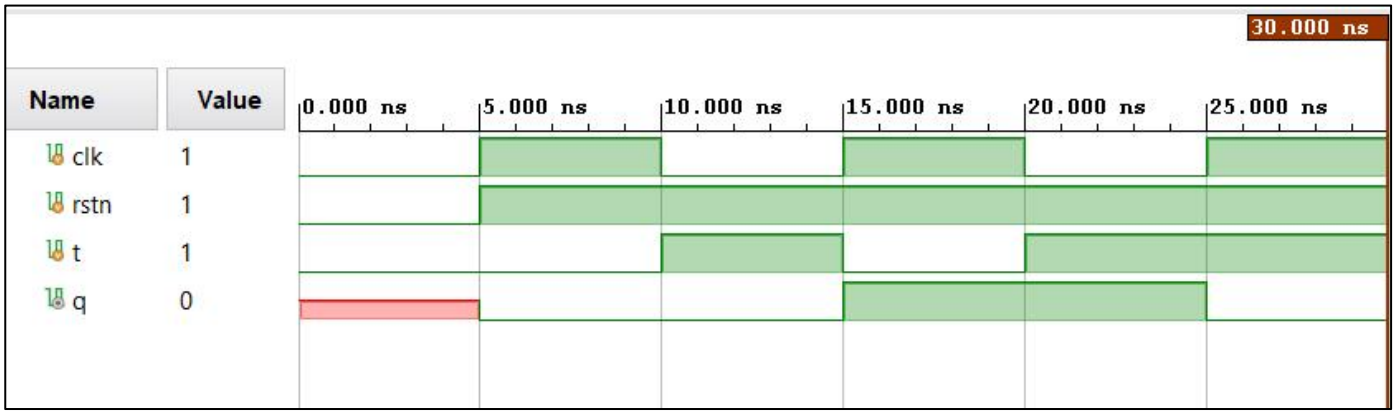
4) <u>JK Flip-Flop:-</u>

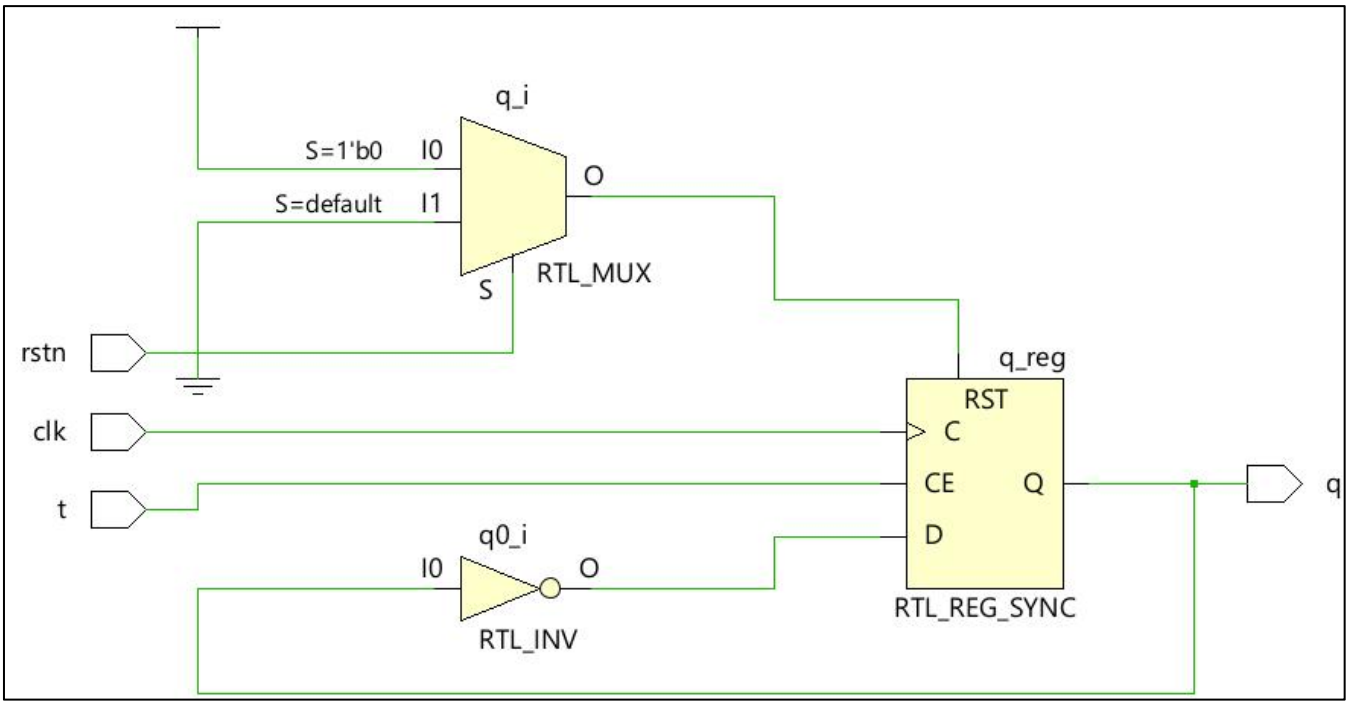
```
module tb;

    reg j,k,clk;
    always #5 clk = ~clk;
```

```
    jkflipFlop u0(.j(j),.k(k),.clk(clk),.q(q));

    initial
        begin
            $dumpfile("dump.vcd"); $dumpvars;
            {j,k,clk} <= 0;

            #5 j <= 0; k <= 1;
            #10 j <= 1; k <= 0;
            #15 j <= 1; k <= 1;

            #20 $finish;
        end
endmodule
```

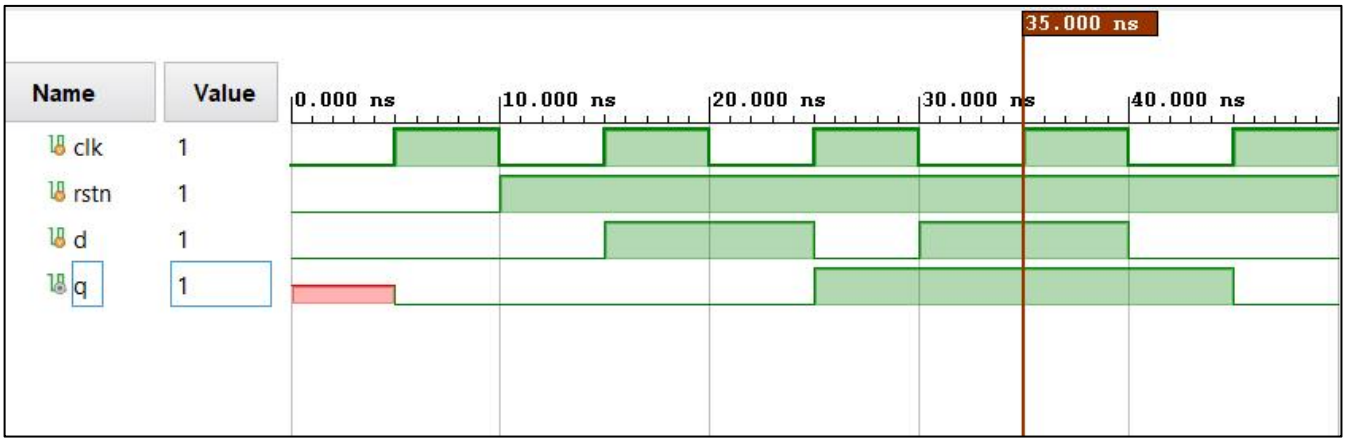
- **OUTPUT WAVEFORMS:-**

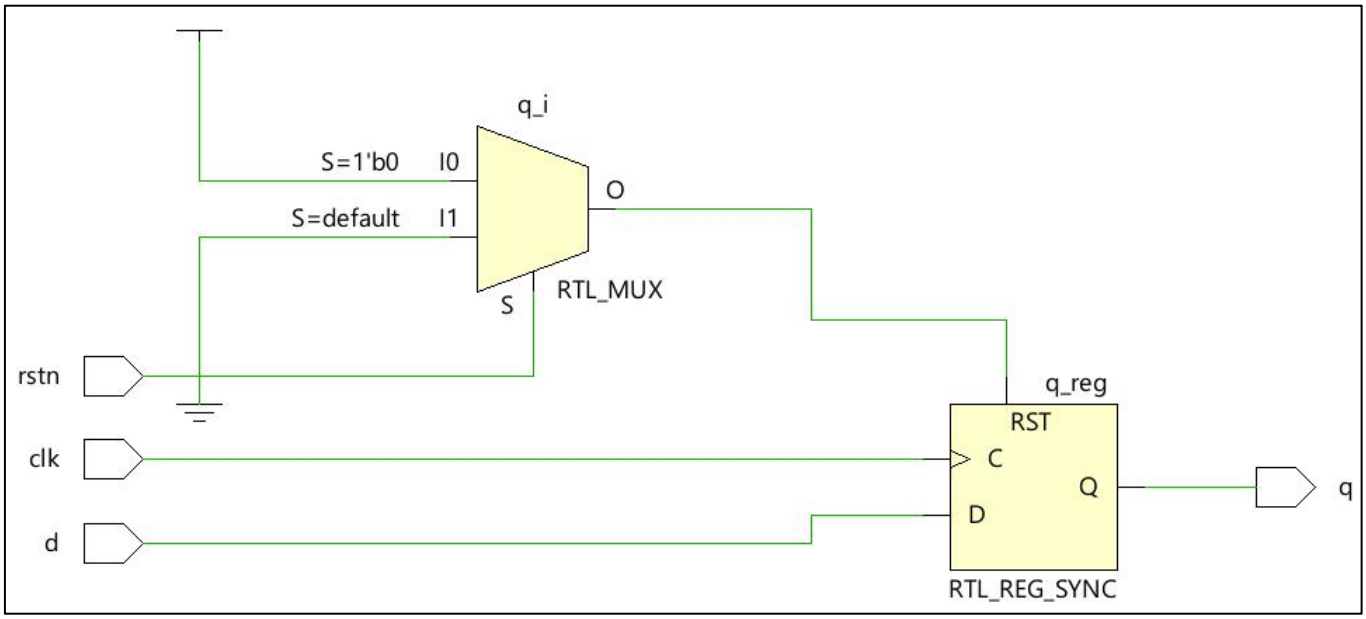
1) T-Flip Flop:-



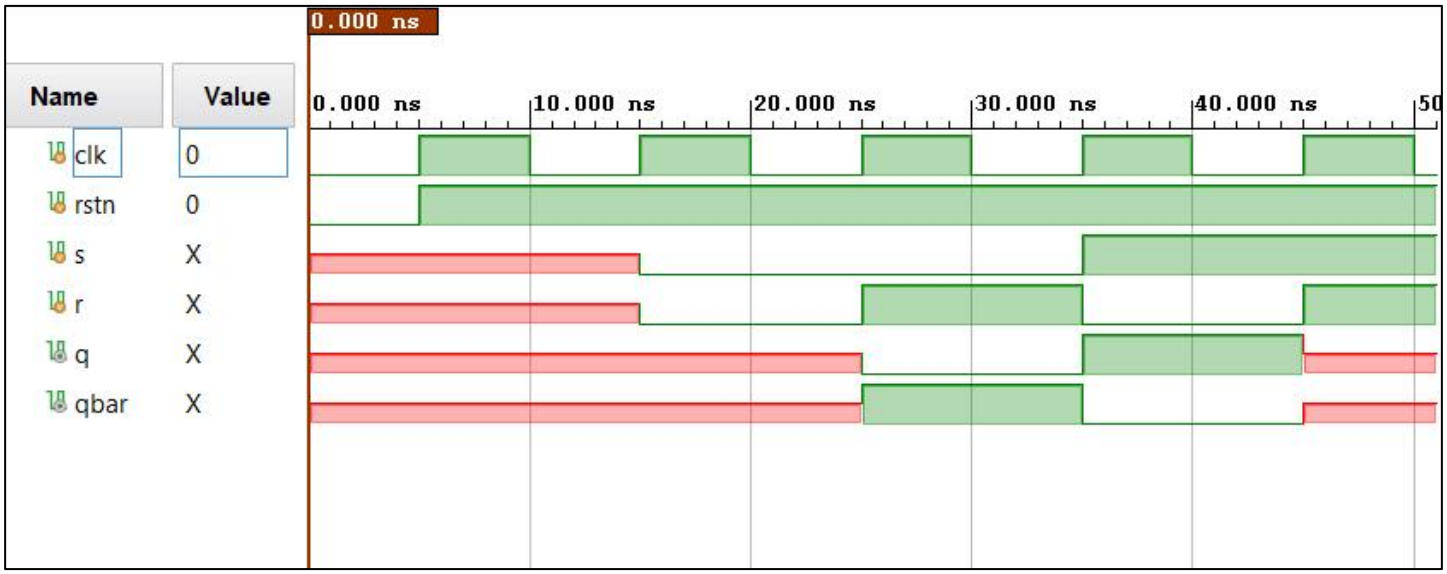


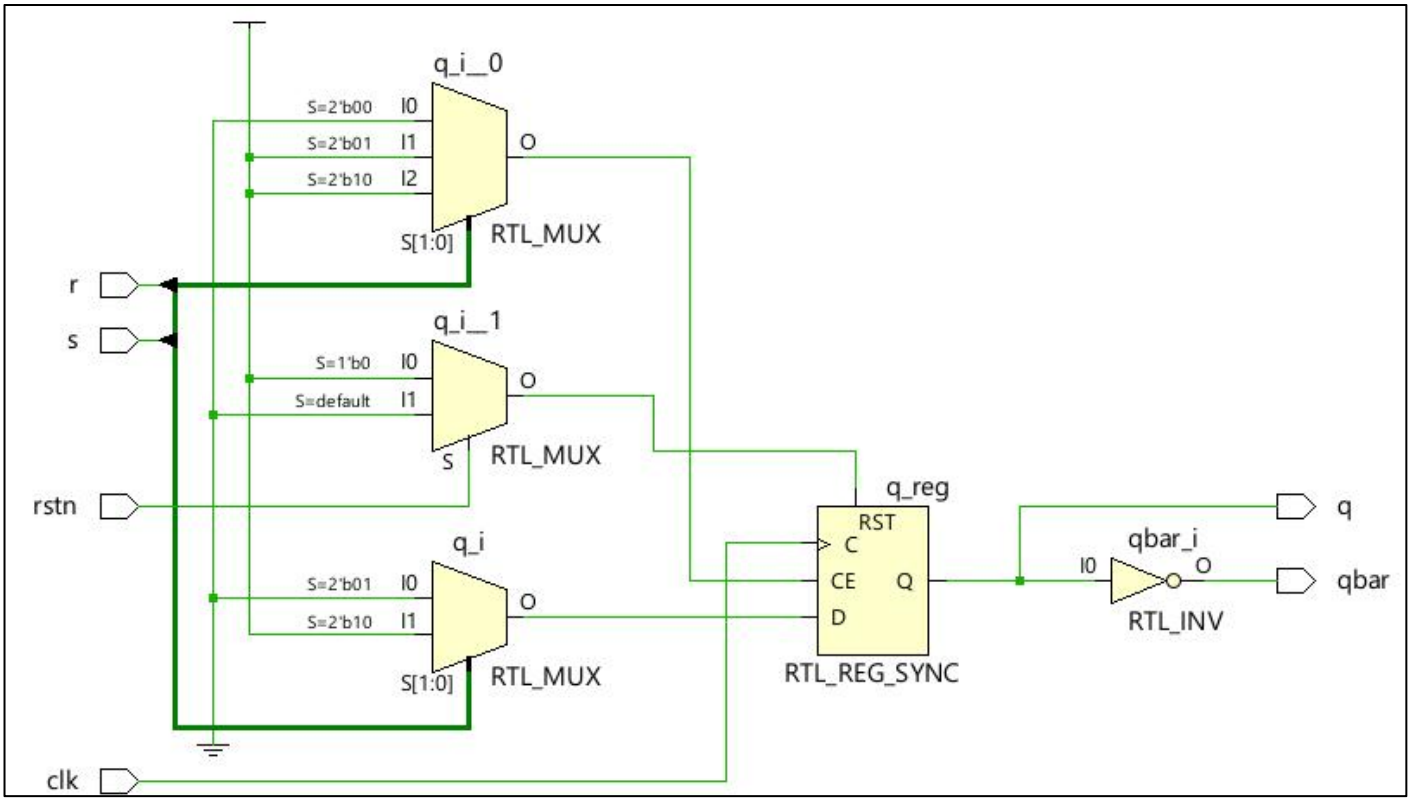
2) D-Flip Flop :-


35.000 ns
Name
Value
0.000 ns
10.000 ns
20.000 ns
30.000 ns
40.000 ns
clk
1
rstn
1
d
1
q
1

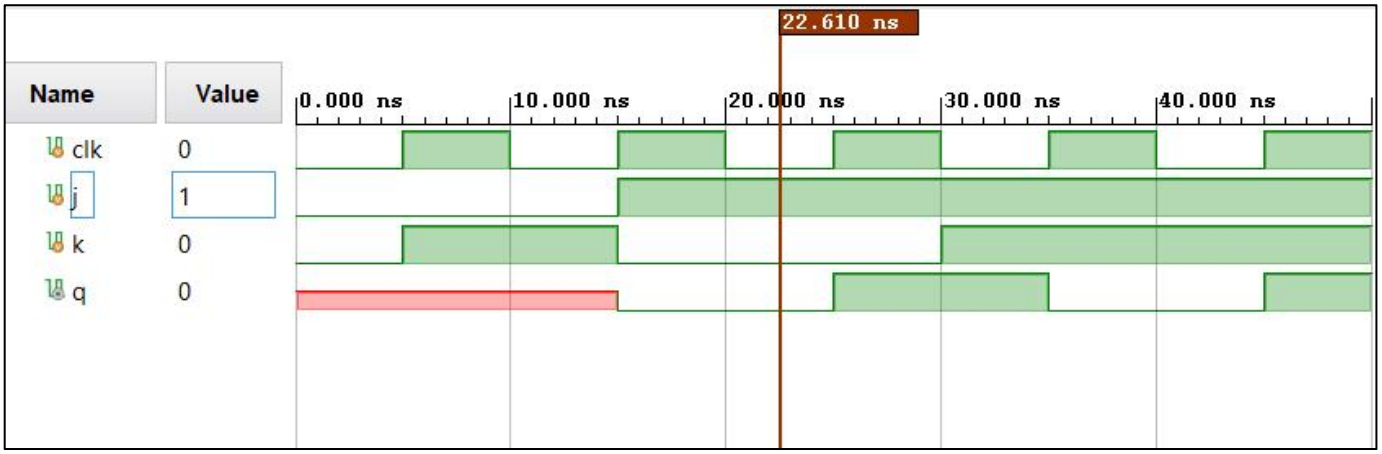

q_i
S=1'b0
I0
S=default
I1
O
S
RTL_MUX
rstn
clk
d
q_reg
RST
C
D
Q
q
RTL_REG_SYNC

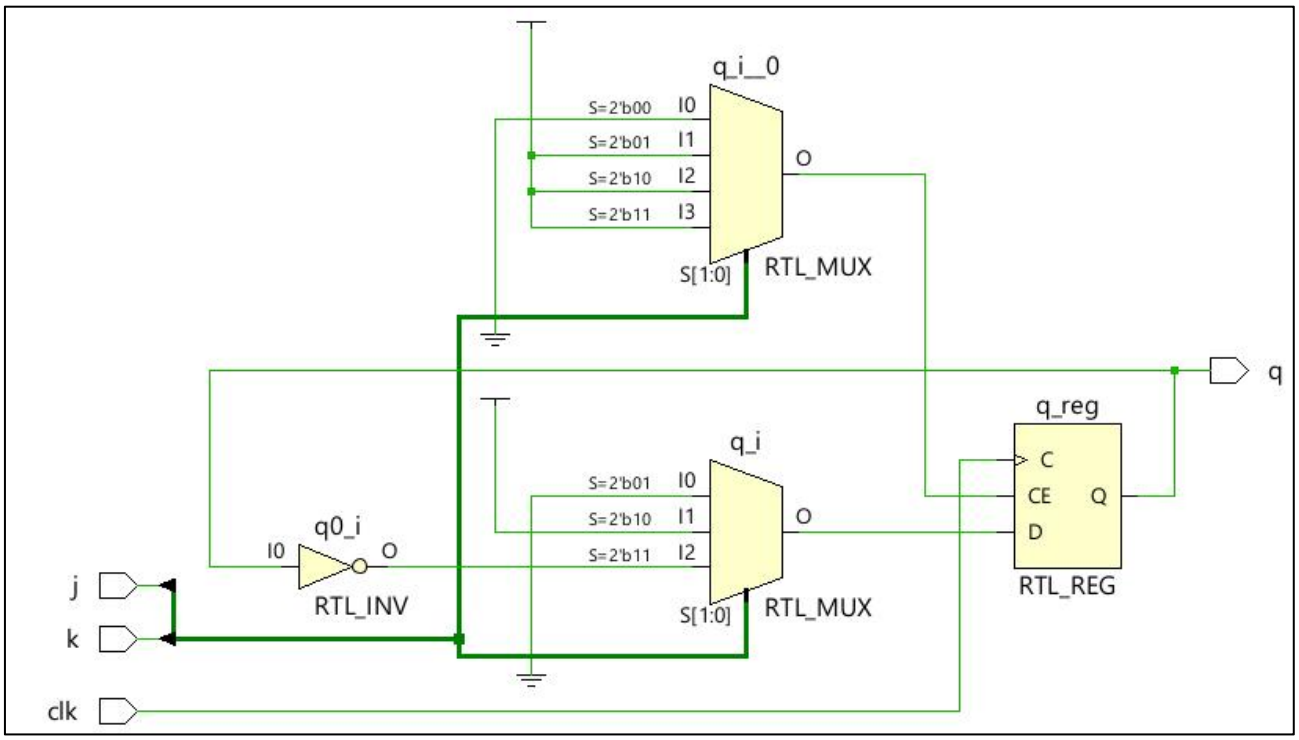
3) SR Flip Flop :-





4) JK Flip Flop:-





- **RESULT :-**T-Flip Flop, D-Flip Flop, SR-Flip Flop and JK-Flip Flop have been implemented in Verilog and the corresponding waveforms and the RTL Schematic are observed.

01/04/24

EXPERIMENT - 6

- **AIM :-** To design and simulate the Array Multiplier.

- **SOFTWARE USED :-** AMD Vivado 2023.2

- **THEORY :-**

Array multiplier is similar to how we perform multiplication with pen and paper i.e. finding a partial product and adding them together. It is simple architecture for implementation.

Disadvantages:

1) Delay in computation and high power consumption
2) Physical implementation takes a large area.

Advantages:

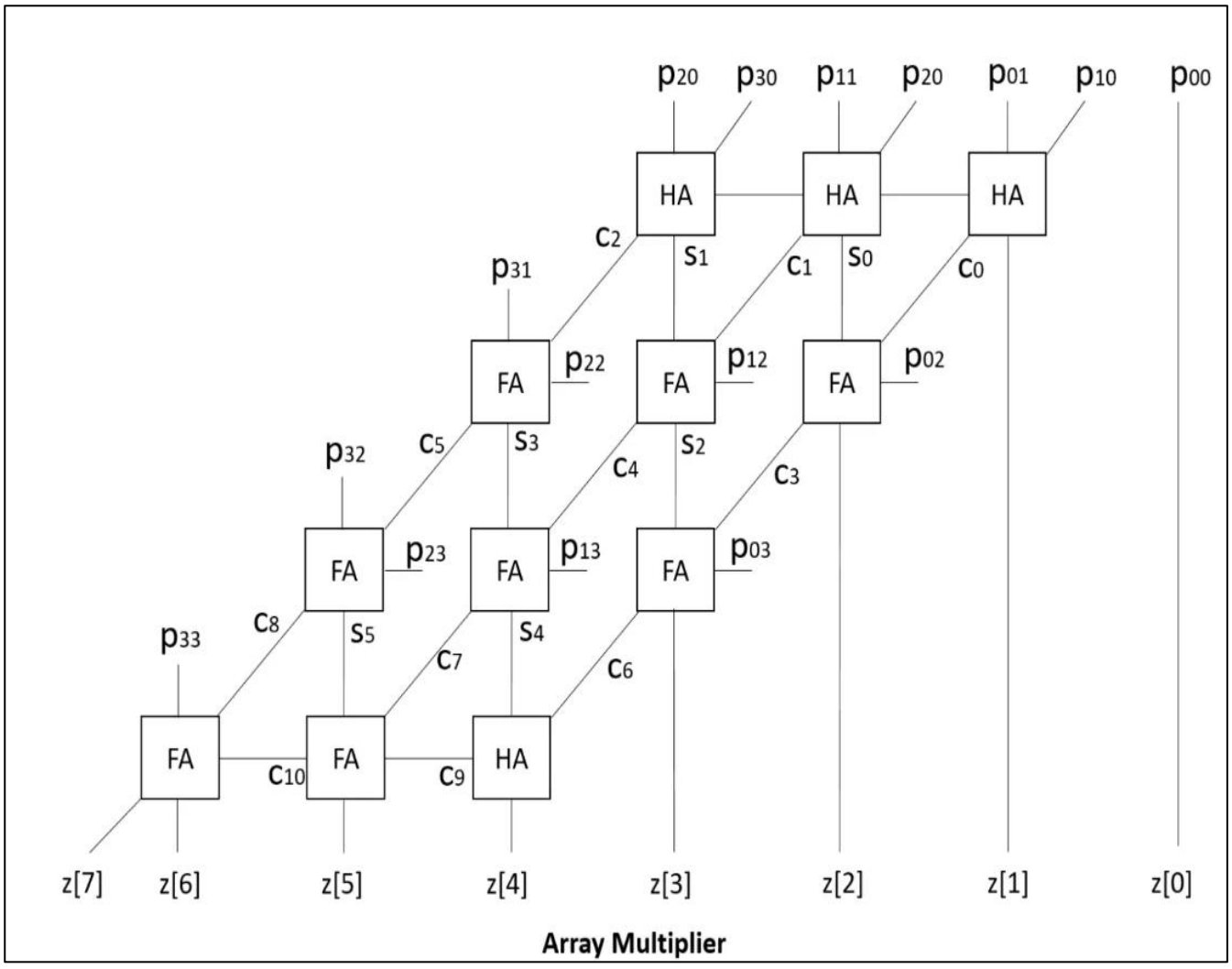
1) Simple implementation that uses an add-shift algorithm.
2) Easy to route and scalable.

Let's understand its design implementation with a 4 x 4 unsigned array multiplier.

Here,
A – Multiplicand, B – Multiplier, p00 – a0b0, p10 – a1b0, p20 – a2b0 and so on.

$$
\begin{array}{cccccccc}
 & & & a_3 & a_2 & a_1 & a_0 & \\
 & & x & b_3 & b_2 & b_1 & b_0 & \\
\hline
 & & & p_{30} & p_{20} & p_{10} & p_{00} & \\
 & & p_{31} & p_{21} & p_{11} & p_{01} & x & \\
 & p_{32} & p_{22} & p_{12} & p_{02} & x & x & \\
p_{33} & p_{23} & p_{13} & p_{03} & x & x & x & \\
\hline
z_7 & z_6 & z_5 & z_4 & z_3 & z_2 & z_1 & z_0
\end{array}
$$

Following is the block diagram of the same.



Array Multiplier

- **VERILOG CODE :-**

```
module halfAdder(input a,input b,output s0,output c0);
  assign s0 = a ^ b;
  assign c0 = a & b;
endmodule

module fullAdder(input a,input b,input cin,output
s0,output c0);
  assign s0 = a ^ b ^ cin;
  assign c0 = (a & b) | (b & cin) | (a & cin);
endmodule


module arrayMultiplier(input [3:0] A,B, output[7:0] z);
  reg signed p[4][4];
  wire [10:0] c;
  wire [5:0] s;

  genvar g;

  generate
    for(g = 0; g<4; g++) begin
      and a0(p[g][0], A[g], B[0]);
      and a1(p[g][1], A[g], B[1]);
      and a2(p[g][2], A[g], B[2]);
      and a3(p[g][3], A[g], B[3]);
    end
  endgenerate
  assign z[0] = p[0][0];
```

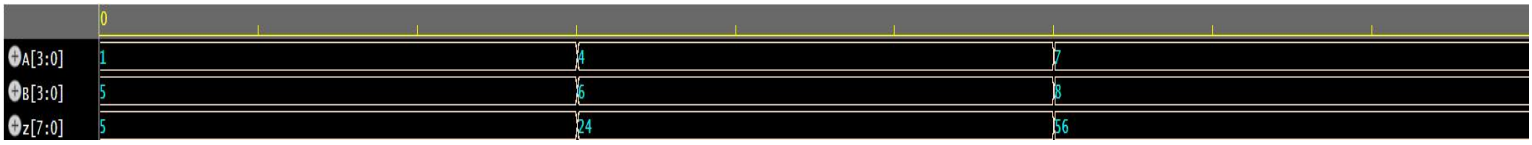
```
  //row 0
 halfAdder h0(p[0][1], p[1][0], z[1], c[0]);
 halfAdder h1(p[1][1], p[2][0], s[0], c[1]);
 halfAdder h2(p[2][1], p[3][0], s[1], c[2]);

 //row1
 fullAdder f0(p[0][2], c[0], s[0], z[2], c[3]);
 fullAdder f1(p[1][2], c[1], s[1], s[2], c[4]);
 fullAdder f2(p[2][2], c[2], p[3][1], s[3], c[5]);

 //row2
 fullAdder f3(p[0][3], c[3], s[2], z[3], c[6]);
 fullAdder f4(p[1][3], c[4], s[3], s[4], c[7]);
 fullAdder f5(p[2][3], c[5], p[3][2], s[5], c[8]);

 //row3
 halfAdder h3(c[6], s[4], z[4], c[9]);
 fullAdder f6(c[9], c[7], s[5], z[5], c[10]);
 fullAdder f7(c[10], c[8], p[3][3], z[6], z[7]);
endmodule
```

- **TESTBENCH :-**

```
module tb;
 reg [3:0] A,B;
 wire[7:0] z;

 arrayMultiplier am(A,B,z);

 initial
  begin
   $dumpfile("dump.vcd"); $dumpvars;
```

```
    $monitor("A = %0d: B = %0d -->  P(dec) = %0d", A,
B,z);
     A = 1; B = 5; #3;
     A = 4; B = 6; #3;
     A = 7; B = 8; #3;
     $finish;
   end
endmodule
```

- **OUTPUT WAVEFORM :-**




```
A = 1: B = 5 -->  P(dec) = 5
A = 4: B = 6 -->  P(dec) = 24
A = 7: B = 8 -->  P(dec) = 56
```

- **RESULT :-**

Array Multiplier has been implemented through Verilog Code and the results were verified.

08/04/24 **EXPERIMENT - 7**

- **AIM:-** To design and simulate Mod-13 Asynchronous Counter using JK Flip Flop and test its working in Verilog

- **SOFTWARE USED:-** AMD Vivado 2023.2

- **THEORY:-**

Counters are sequential logic devices that follow a predetermined sequence of counting states triggered by an external clock (CLK) signal. The number of states or counting sequences through which a particular counter advances before returning to its original first state is called the modulus (MOD). In other words, the modulus (or modulo) is the number of states the counter counts and is the dividing number of the counter.

Modulus Counters, or MOD counters, are defined based on the number of states that the counter will sequence before returning to its original value.

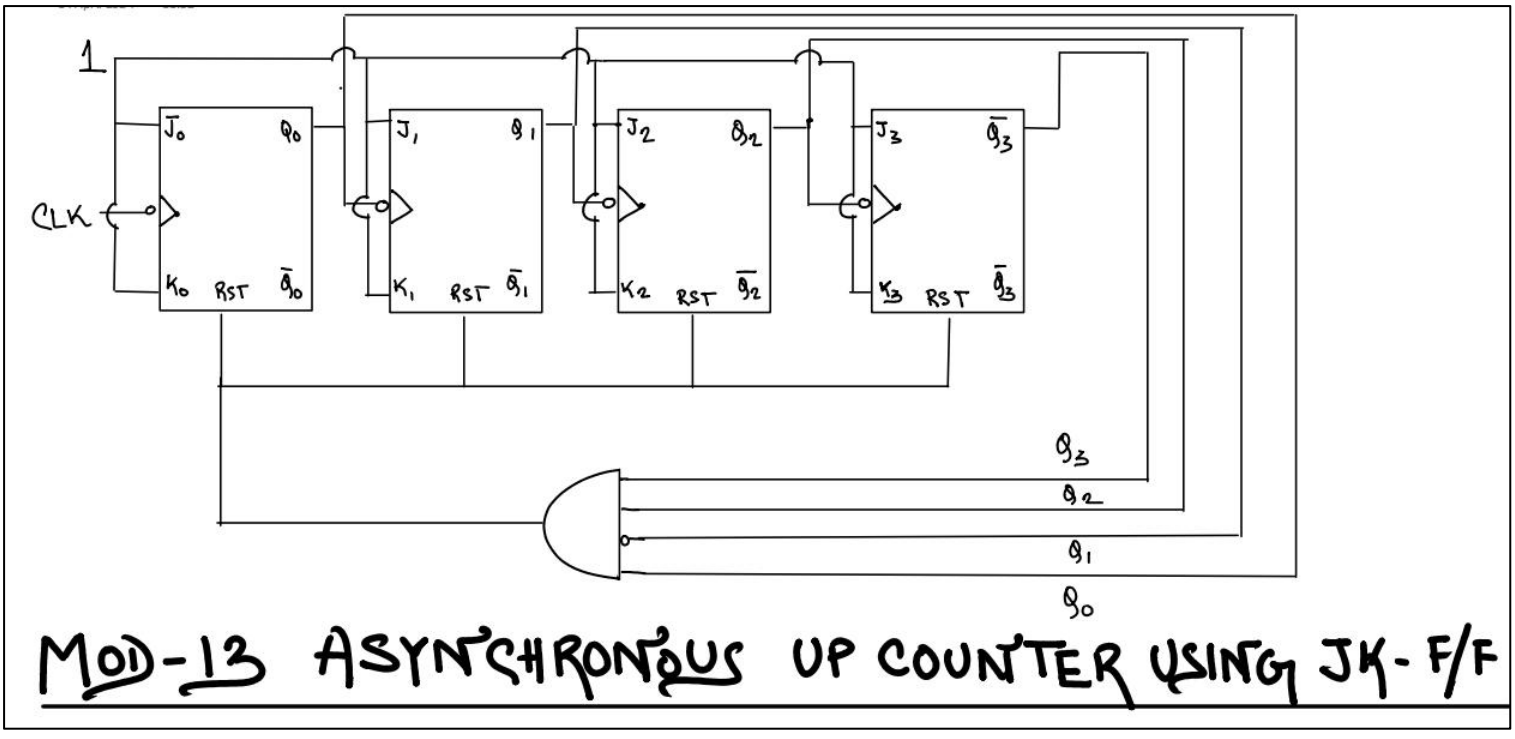
Therefore, a "Mod-N" counter will require the "N" number of flip-flops connected to count a single data bit while providing 2n different output states (n is the number of bits). Note that N is always a whole integer value.

Then we can see that MOD counters have a modulus value that is an integral power of 2, that is, 2, 4, 8, 16 and so on to produce an n-bit counter depending on the number of flip-flops used, and how they are connected, determining the type and modulus of the counter.

Specifically, to design a Mod-13 counter using JK Flip Flops, we need 4 Flip Flops, since 13 can be written as 1101 in base 2

which requires 4 bits and hence 4 flip flops to store all the states. Also, since we are designing an asynchronous counter we need to ensure that the output of the i^{th} stage goes as the clock input to the $(i+1)^{th}$ stage and also ensure that all the inputs to the JK Flip Flops are set as 1, i.e. the toggling state of the Flip Flop so as to get the desired result.

In our case, we are using a negative edge-triggering clock and if we give the input of Q to the next stage of the clock, then we can design an Asynchronous Up Counter which in a generic way counts up to 2^n,where n is the total number of Flip-Flops used. To reduce the number of states and count upto 13 only, we need to ensure that we use a combinational logic circuit forming out of the output of every stage and give that specific input to the RST pin of all the Flip-Flops to finally achieve a Mod-13 Asynchronous Up Counter as follows :-



- **VERILOG CODE:-**

```
module jkFlipFlop(input j,k,clk,rstn, output reg q);
 initial
   begin
```

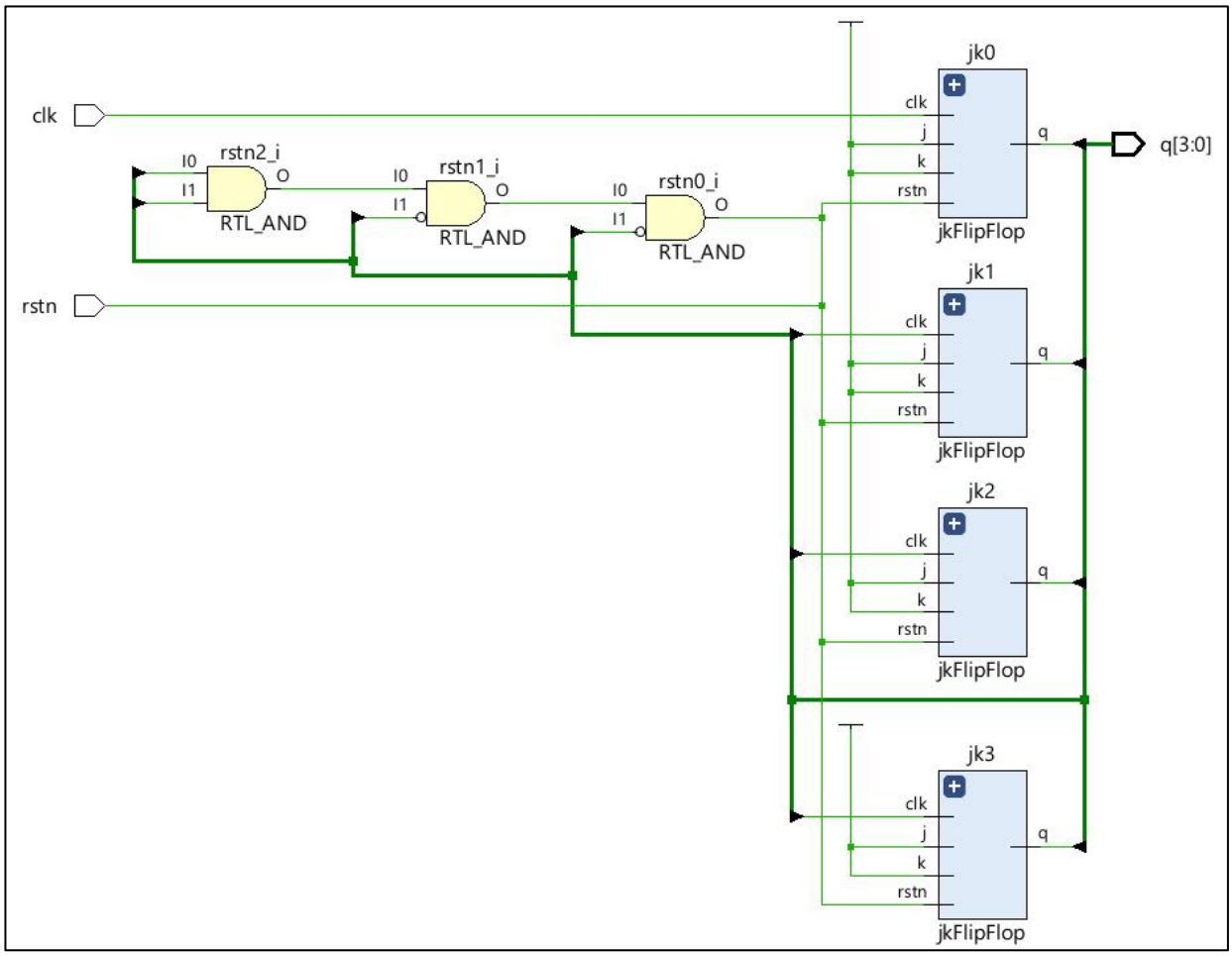
```
    q <= 1'b0;
   end

  always @(negedge clk or negedge rstn)
   begin
    if(!rstn)
     begin
      q <= 0;
     end
    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
endmodule

module asyncCounter(input clk,rstn, output [3:0] q);
  // Up-Counter
  not (x,q[1]);
  not (y,q[0]);
  and(rstn,q[3],q[2],x,y);

  jkFlipFlop jk0(1,1,clk,rstn,q[0]);
  jkFlipFlop jk1(1,1,q[0],rstn,q[1]);
  jkFlipFlop jk2(1,1,q[1],rstn,q[2]);
  jkFlipFlop jk3(1,1,q[2],rstn,q[3]);
endmodule
```

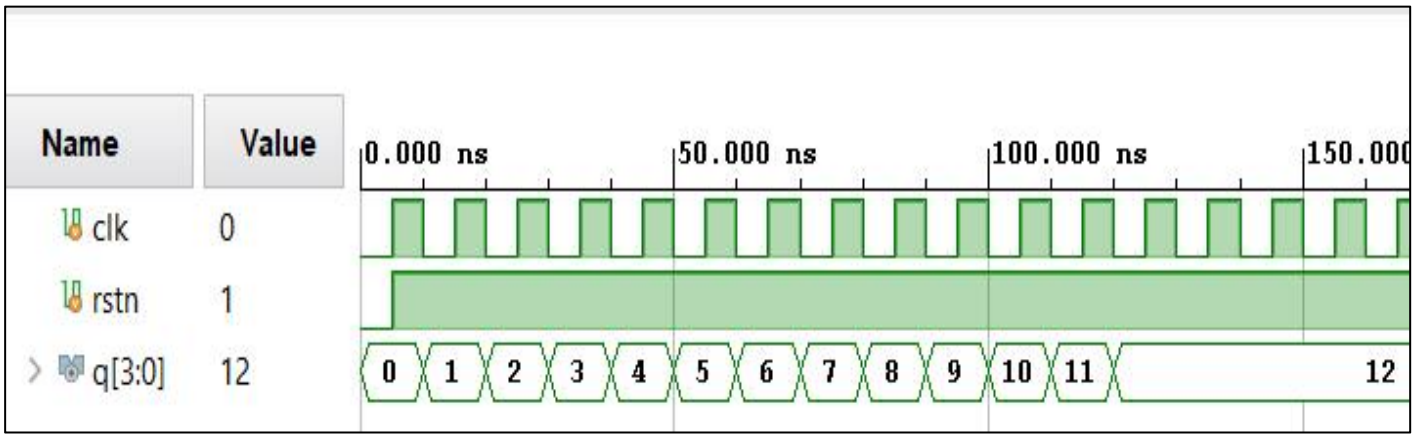
● TESTBENCH :-

```
module tb;
```

```
reg clk,rstn;
wire[3:0] q;

asyncCounter uut(clk,rstn,q);
always #5 clk = ~clk;

initial
  begin
    clk = 0; rstn = 0;
    #5 rstn = 1;
    #200 $finish;
   end
endmodule
```

- **OUTPUT WAVEFORM:-**





- **RESULT :-**

Mod-13 Asynchronous up counter has been successfully implemented and the result has been verified and output has been observed.

15/04/2024

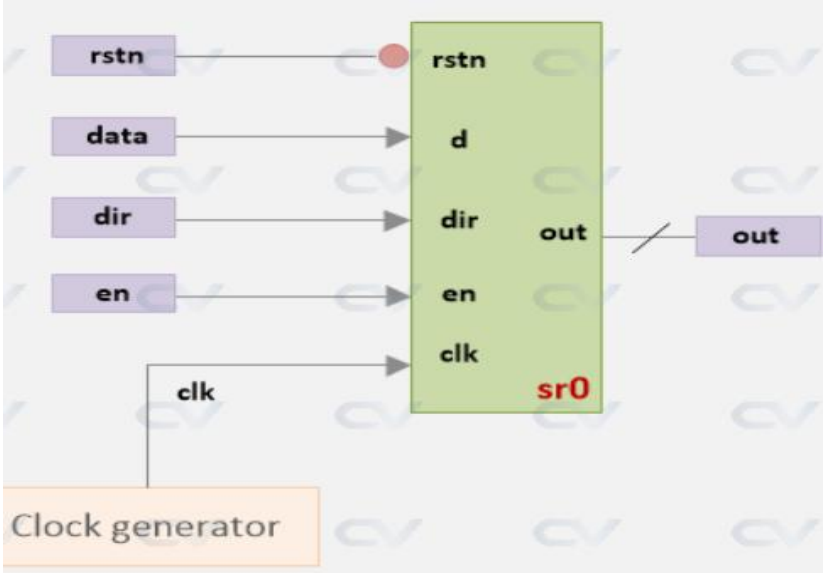
EXPERIMENT - 8

- **AIM:-** To design and simulate a 4-bit Universal Shift Register using D-Flip Flop and test its working
- **SOFTWARE USED :-** AMD Vivado 2023.2
- **THEORY:-**

A shift register is a type of digital circuit that can store and shift data in a sequential manner. It is a cascade of flip-flops, where the output of one flip-flop is connected to the input of the next flip-flop. The data is shifted from one flip-flop to the next on every clock cycle, allowing the circuit to delay or store the data.

A typical shift register has a single bit input, and a n-bit output governed by the parameter *MSB* used in the verilog code. Depending on this *MSB,* we can design a 4-bit, 8-bit, 16-bit and so on, universal shift registers. This shift register have the following key features :-

- Can be enabled or disabled using the *enable* pin of the design
- Can shift to the left or to the right using the *direction* pin of the design
- If *rstn* is pulled low, it will reset the shift register and output will become 0
- Input data values to the shift register can be controlled by the *d* pin



● VERILOG CODE :-

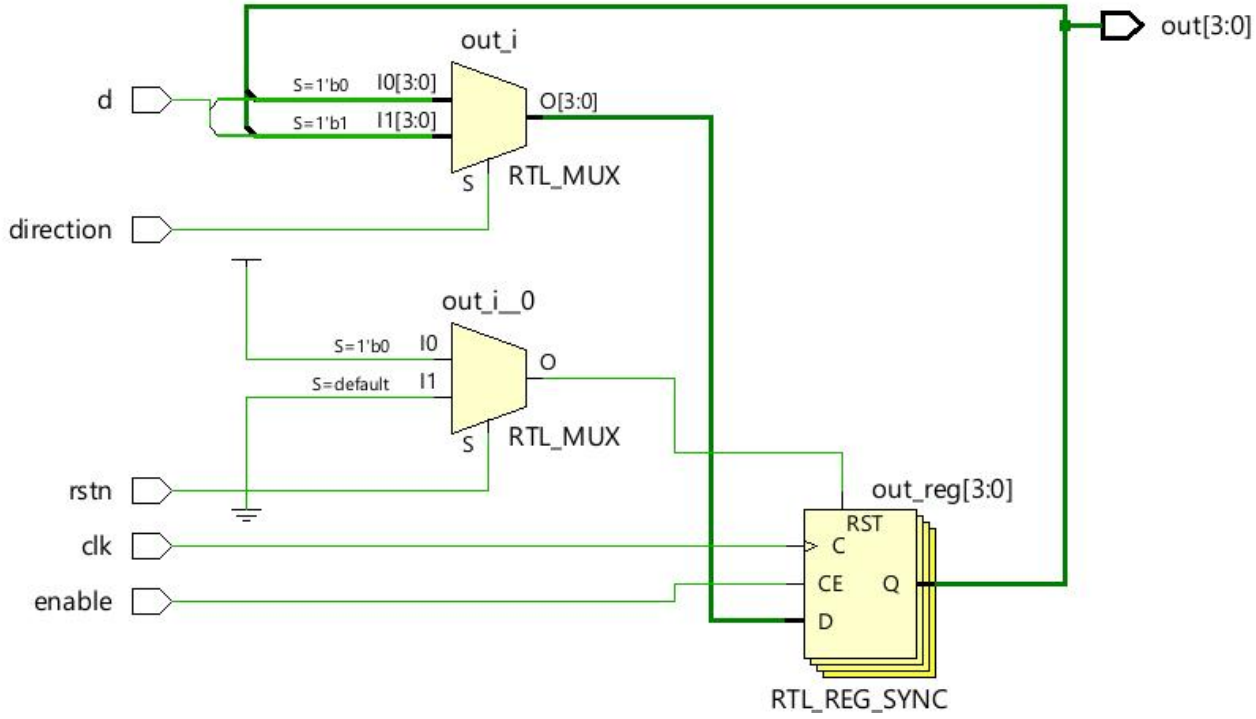
```
module shiftRegister #(parameter MSB = 4) (input
d,clk,rstn,enable,direction,output reg [MSB-1:0] out);

 always @(posedge clk)
  if(!rstn)
    out <= 0;
  else
    begin
     if(enable)
       case(direction)
         0 : out <= {out[MSB-2:0],d};
         1 : out <= {d,out[MSB-1:1]};
       endcase
     else
       out <= out;
    end
endmodule
```

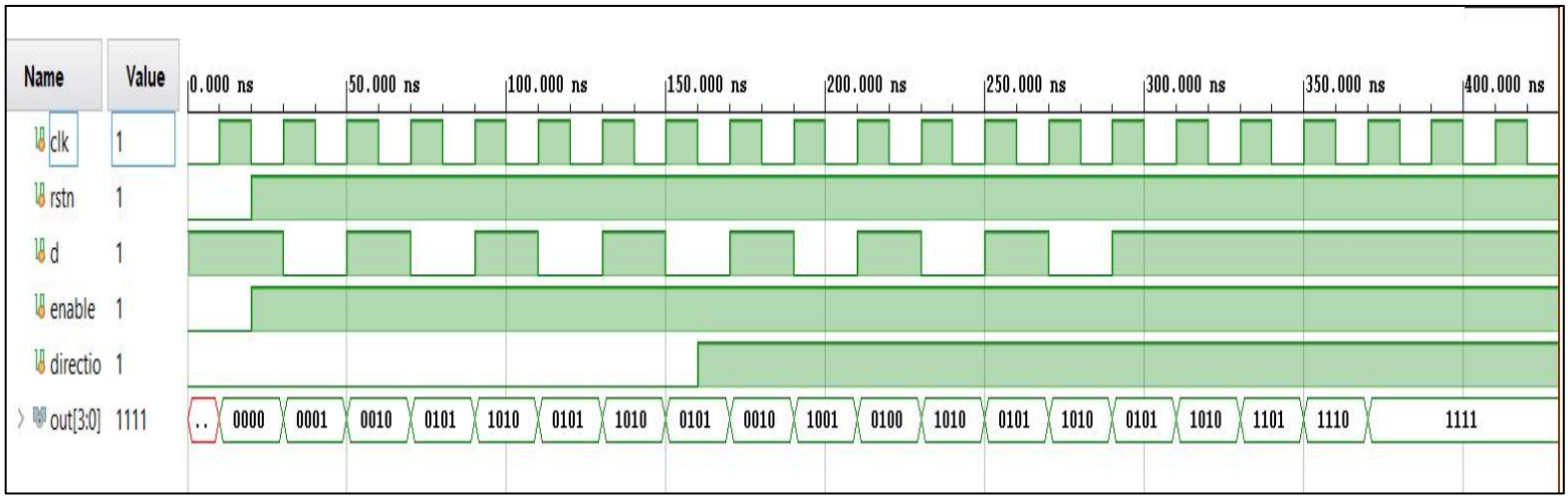
● TESTBENCH :-

```
module tb;
 parameter MSB = 4;
 reg clk,rstn,d,enable,direction;
 wire[MSB-1:0] out;

 shiftRegister #(MSB) uut(d,clk,rstn,enable,direction,out);

 always #10 clk = ~clk;

 initial
  begin
    $dumpfile("dump.vcd"); $dumpvars;

    clk <= 0; d <= 'h1; rstn <= 0; enable <= 0; direction <= 0;
    rstn <= 0;
```

```
    #20 rstn <= 1; enable = 1;

    repeat(7) @(posedge clk)
      d <= ~d;

    #10 direction <= 1;

    repeat(7) @(posedge clk)
      d <= ~d;

    repeat(7) @(posedge clk);

    $finish;
  end
endmodule
```

- **OUTPUT WAVEFORMS :-**





- **RESULT :-**

Hence, the 4-bit universal shift register have been implemented and the shifting characteristics has been observed and verified.

15/04/2024

EXPERIMENT - 9

- **AIM :-** To design and simulate the basic gates, i.e., NOT, AND and OR Gates using CMOS Simulation and find the delay difference between the input and the output signals

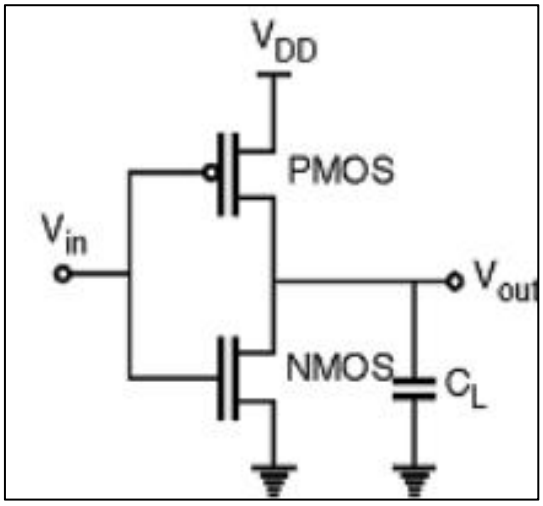
- **SOFTWARE USED :-** Symica DE Free Edition

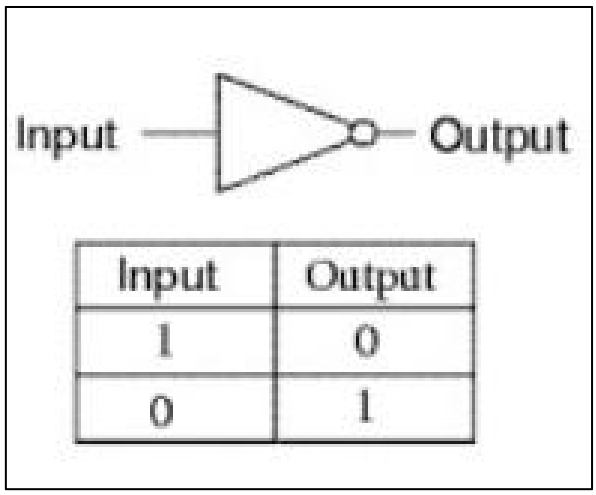
- **THEORY :-**

CMOS logic uses both NMOS and PMOS transistors. The PMOS transistors are used as pull-up network and NMOS transistors are used as pull-down network. And because of that, the static power consumption of the CMOS based logic gates and logic circuit is very low compared to the logic gates which is designed using only either NMOS or PMOS transistors. Moreover, CMOS based logic gates has higher noise margin compared to NMOS and PMOS based logic gates.

1) **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. The logic element like an inverter reverses the applied input signal. In digital logic circuits, binary arithmetic & switching or logic function's mathematical manipulation are best performed through the symbols 0 & 1. If the input logic is zero (0) then the output will be high (1) whereas, if the input logic is one (1), then the output will be low (0). The general CMOS inverter structure is the combination of both the PMOS & NMOS transistors where the pMOS is arranged at the top & nMOS is arranged at the bottom. It is very significant to observe that the CMOS device does not have any resistors, so it will be more power-efficient.





Input	Output
1	0
0	1

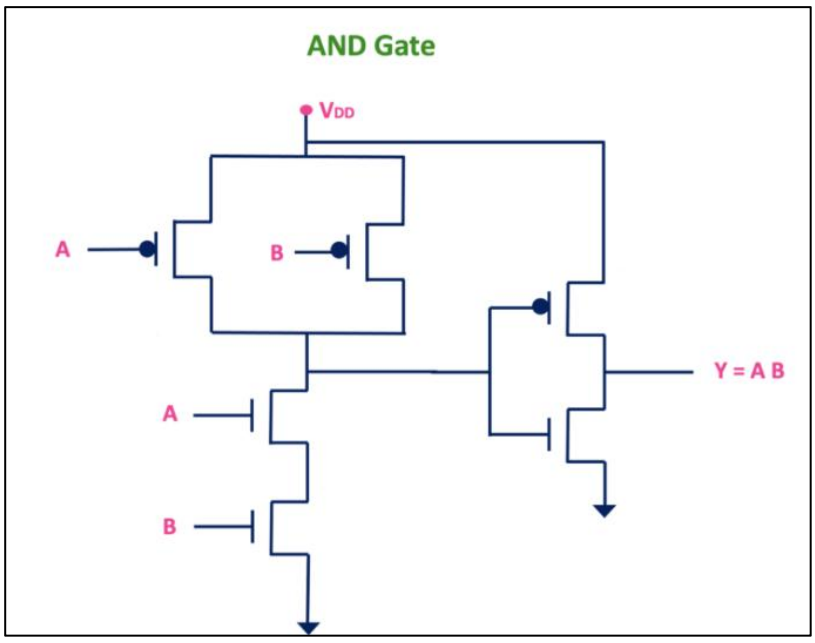
2) **CMOS AND GATE** :-

For two input NAND gate, if A and B are the inputs then its output Y = (A.B)'.

In NMOS network when we have AND operation between the two variables, then two NMOS transistors will get connected in series. And the output will be complement of it.

The PMOS network is dual of the NMOS network. In the NMOS network, if two transistors are connected in series then in the PMOS network, the two PMOS transistors will get connected in parallel.

By connecting the inverter at the output of the NAND gate, we can implement AND gate. The CMOS AND gate is shown below.

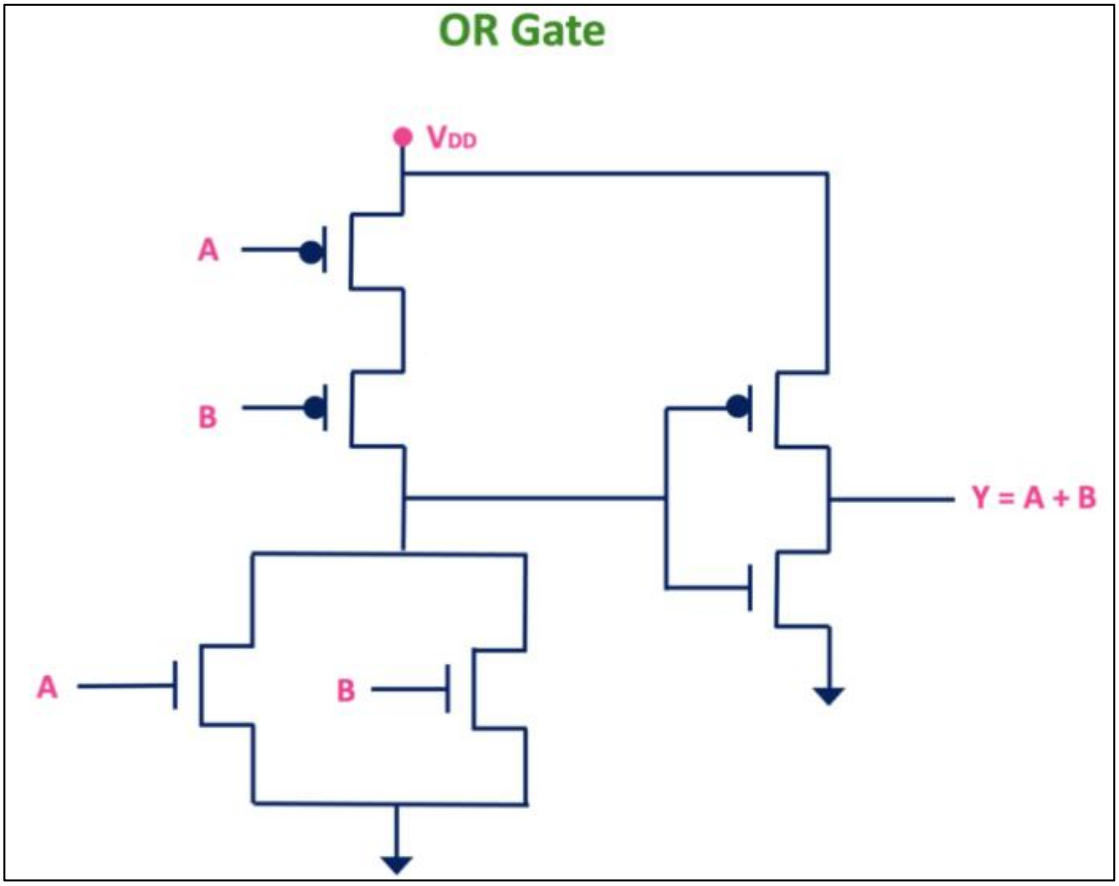


3) **CMOS OR GATE** :-
For two input NOR gate, if A and B are the inputs then its output Y = (A+B)'.

In the NMOS network, whenever there is an OR operation between the two variables then two NMOS transistors will get connected in parallel. And the output will be complement of it.

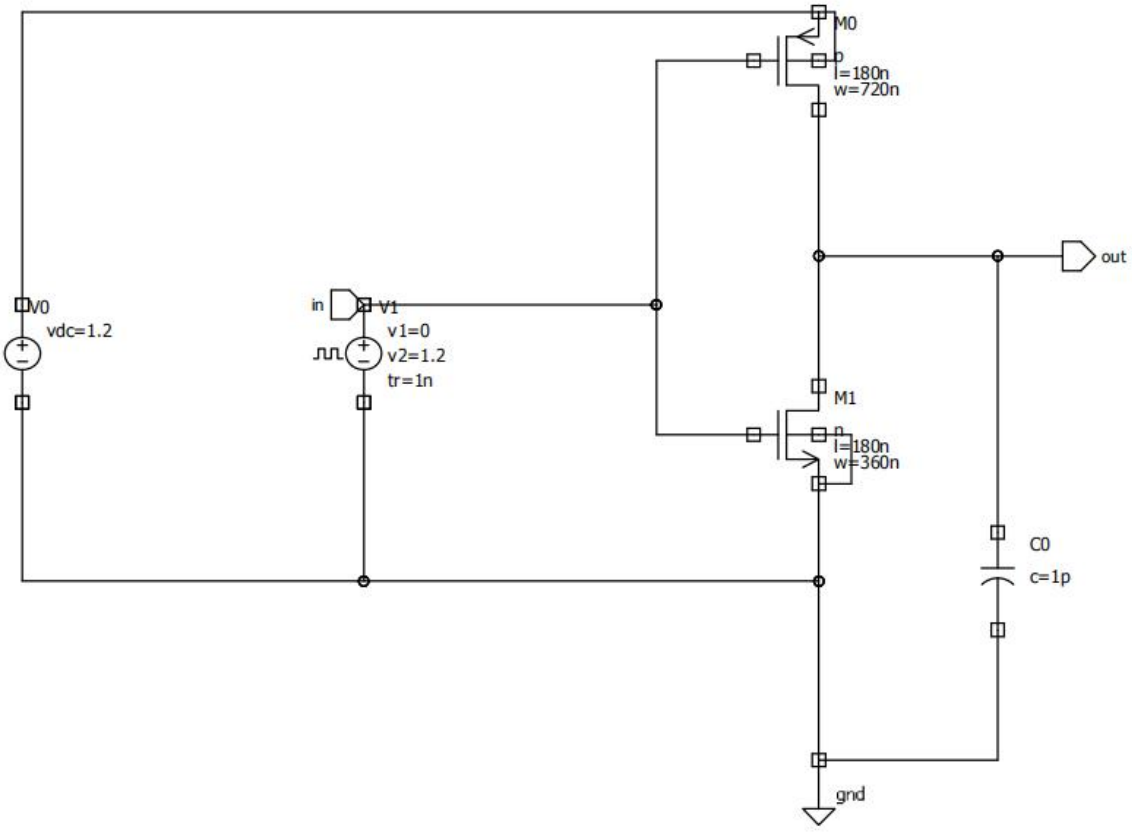
The PMOS network will be the dual of the NMOS network. Therefore, in the PMOS network, the two PMOS transistors will get connected in series.

To implement the OR gate, just add the inverter at the output of the NOR gate. The CMOS OR gate is shown below.

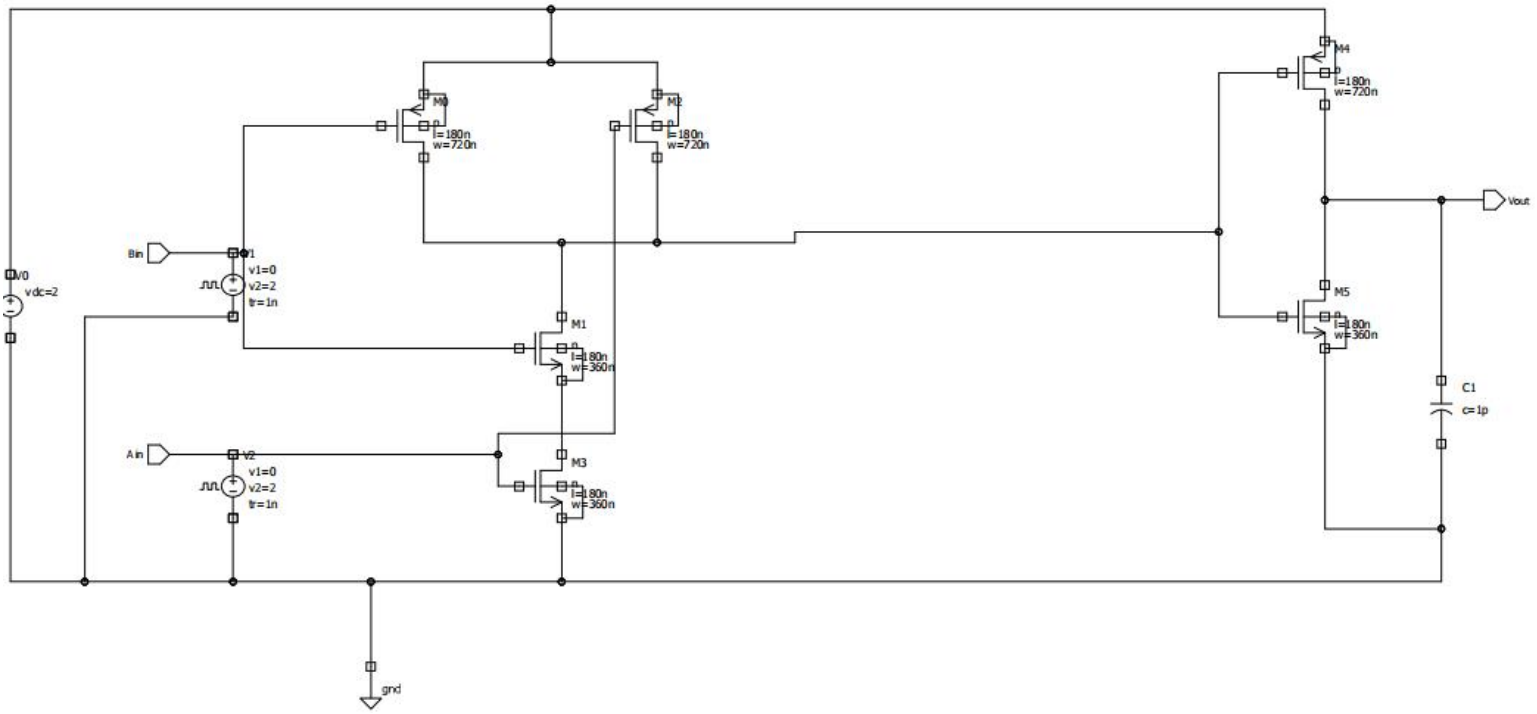


- **SYMICA CELL VIEW :-**

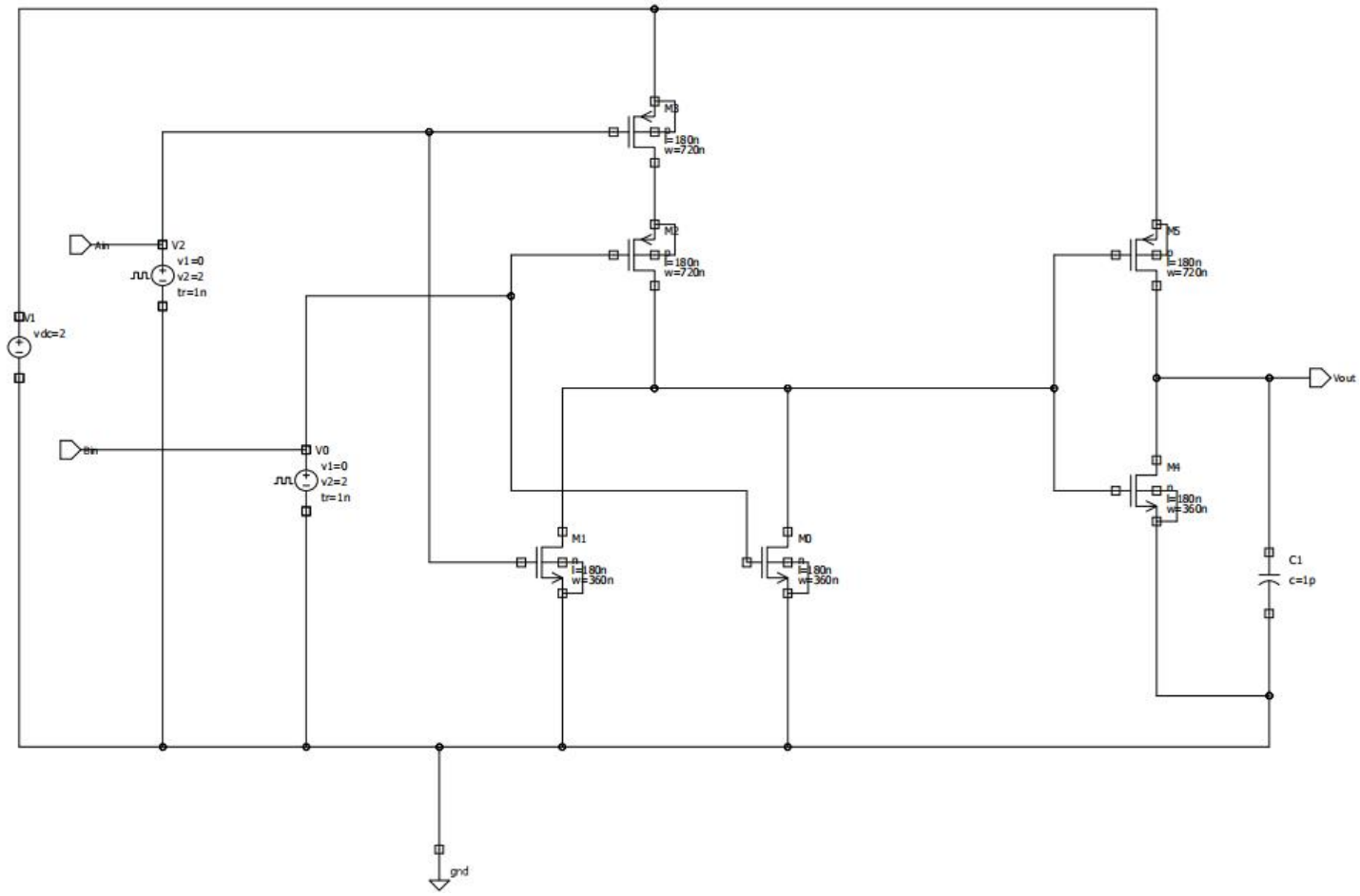
1) **CMOS Inverter** :-



2) **CMOS AND Gate** :-

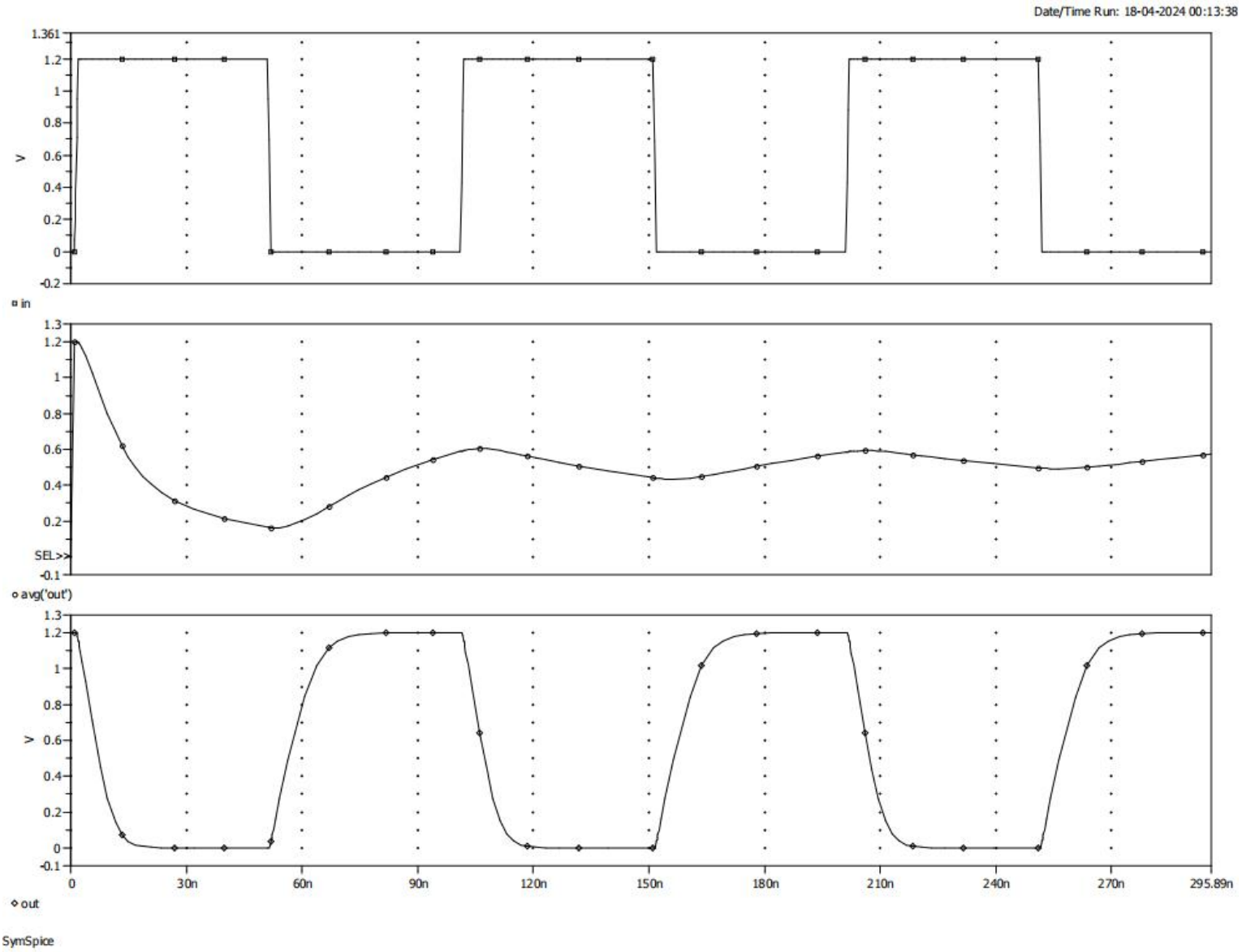


3) **CMOS OR Gate** :-



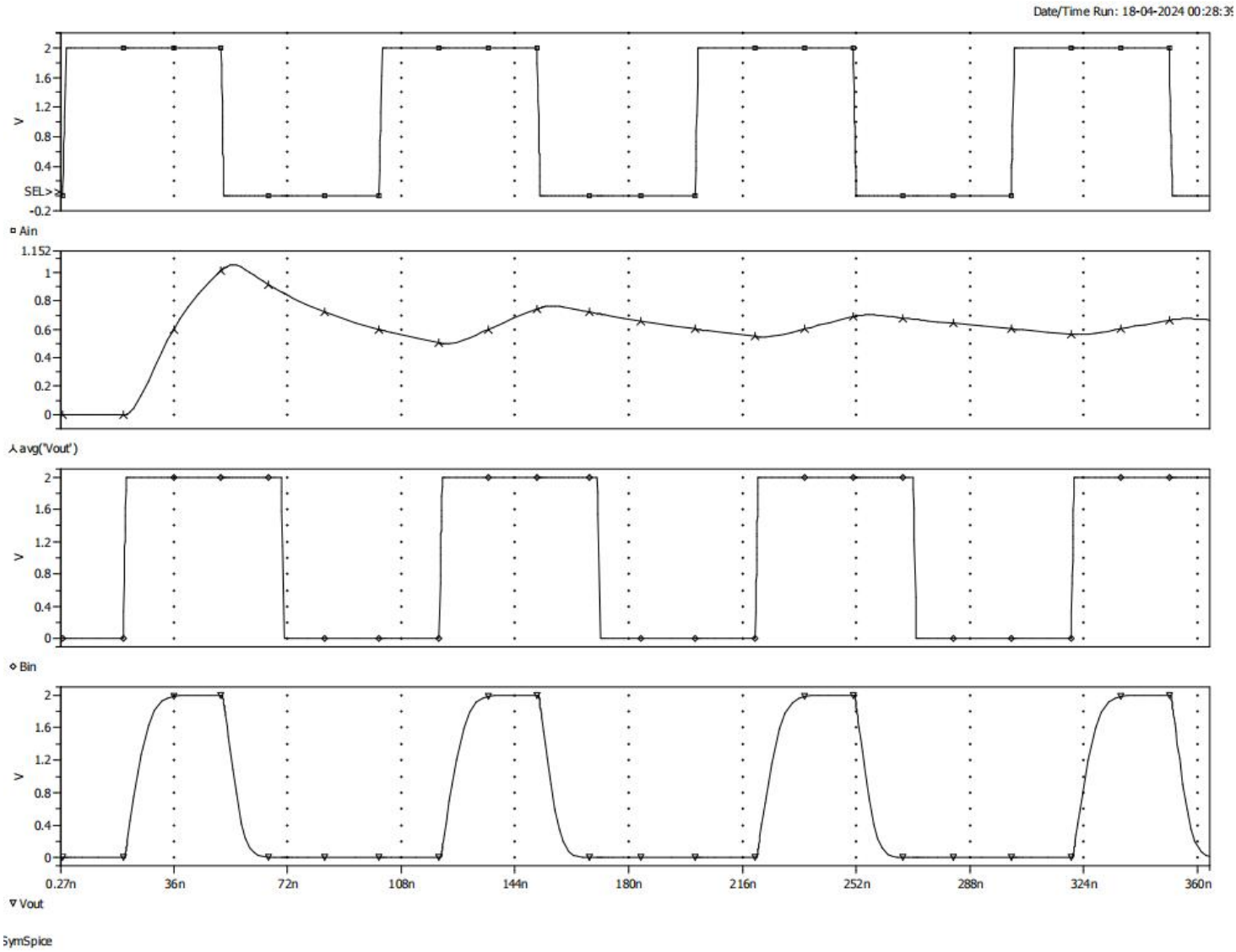
- **SIMULATION RESULTS:-**

1) **CMOS Inverter** :-



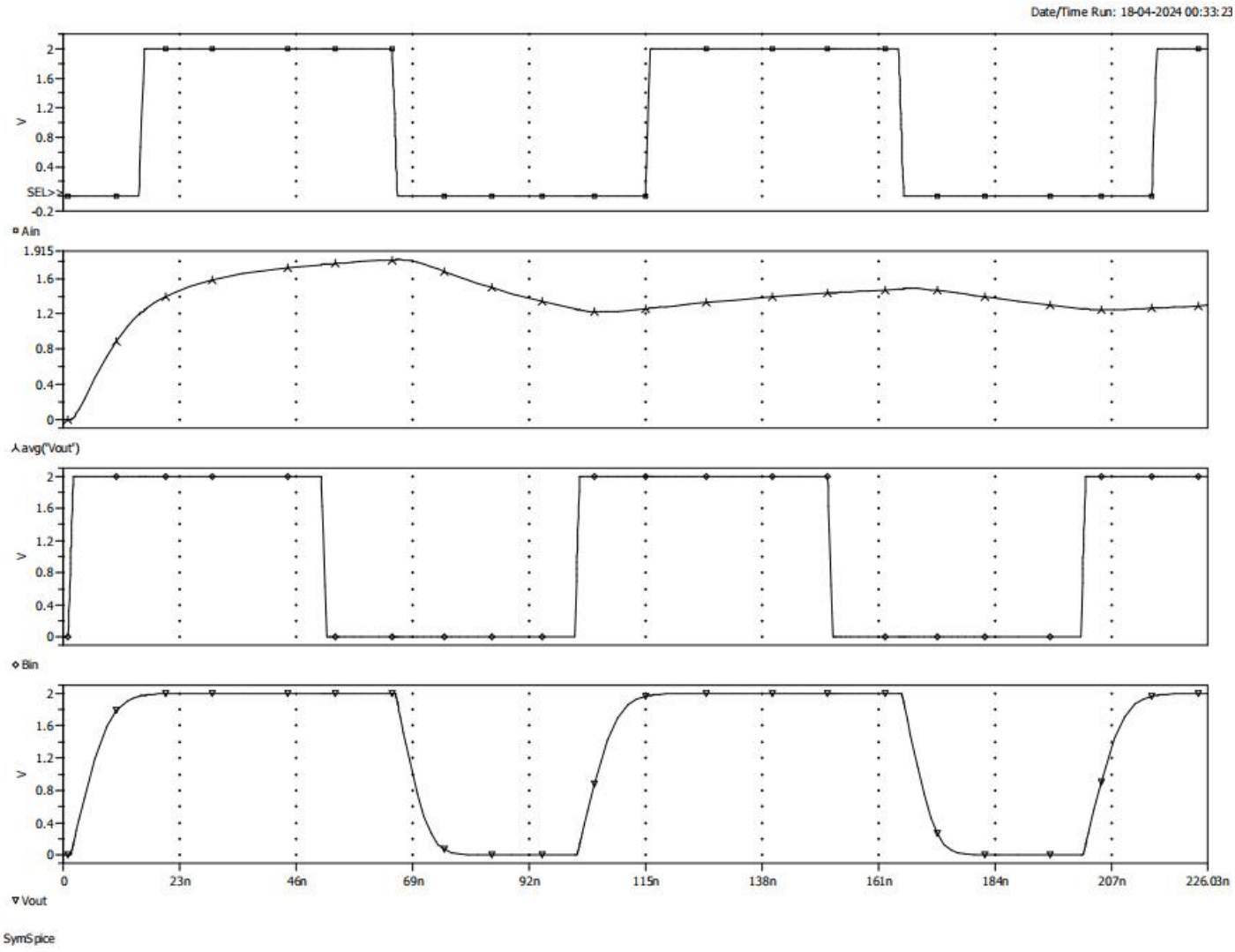
Delay between Input Signal and Output Signal = 1.39012e-009 s

2) **CMOS AND Gate** :-



Delay Difference between Ain and Vout = 2.30845e-008 s
Delay Difference between Bin and Vout = 4.08446e-009 s

3) **CMOS OR Gate** :-



Delay Difference between Ain and Vout = 5.35965e-008 s
Delay Difference between Bin and Vout = 3.96678e-009 s

- **RESULT :-** The basic logic gates , i.e., NOT, AND and OR Gates have been implemented using CMOS Technology and the same has been simulated in *Symica* and corresponding delay differences are calculated between input and the output signal.

15/04/2024

EXPERIMENT - 10

- **AIM :-** To design and simulate the universal gates, i.e., NAND and NOR Gates and find the delay differences between the input and the output signals.

- **SOFTWARE USED :-** Symica DE Free Edition

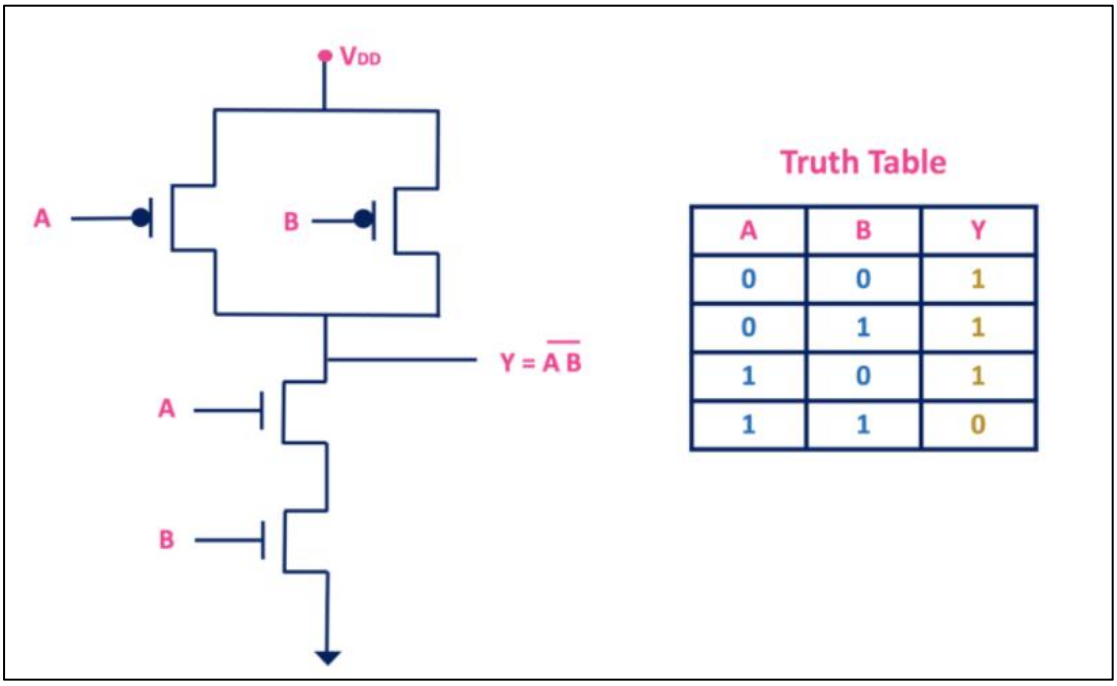
- **THEORY :-**

1) CMOS NAND Gate :-

For two input NAND gate, if A and B are the inputs then its output Y = (A.B)'.

In NMOS network when we have AND operation between the two variables, then two NMOS transistors will get connected in series. And the output will be complement of it.

The PMOS network is dual of the NMOS network. In the NMOS network, if two transistors are connected in series then in the PMOS network, the two PMOS transistors will get connected in parallel.



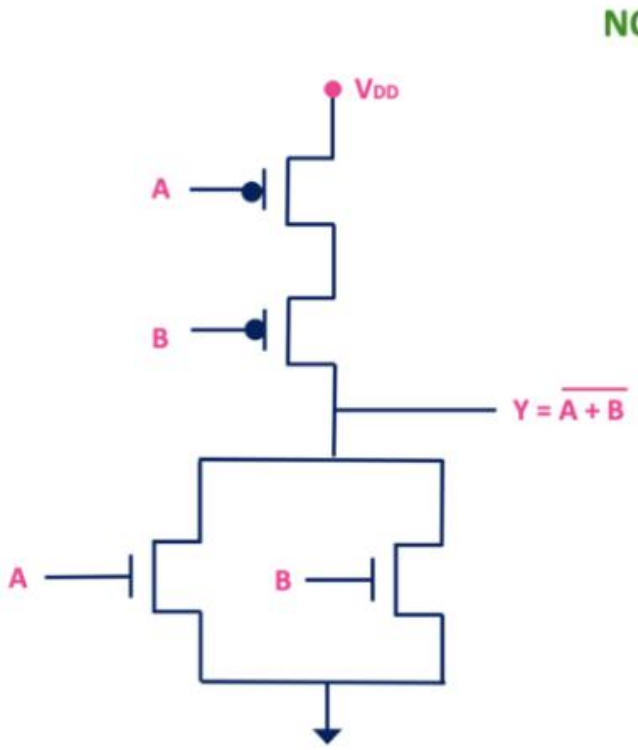
Truth Table

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

2) CMOS NOR Gate :-
For two input NOR gate, if A and B are the inputs then its output Y = (A+B)'.

In the NMOS network, whenever there is an OR operation between the two variables then two NMOS transistors will get connected in parallel. And the output will be complement of it.

The PMOS network will be the dual of the NMOS network. Therefore, in the PMOS network, the two PMOS transistors will get connected in series.

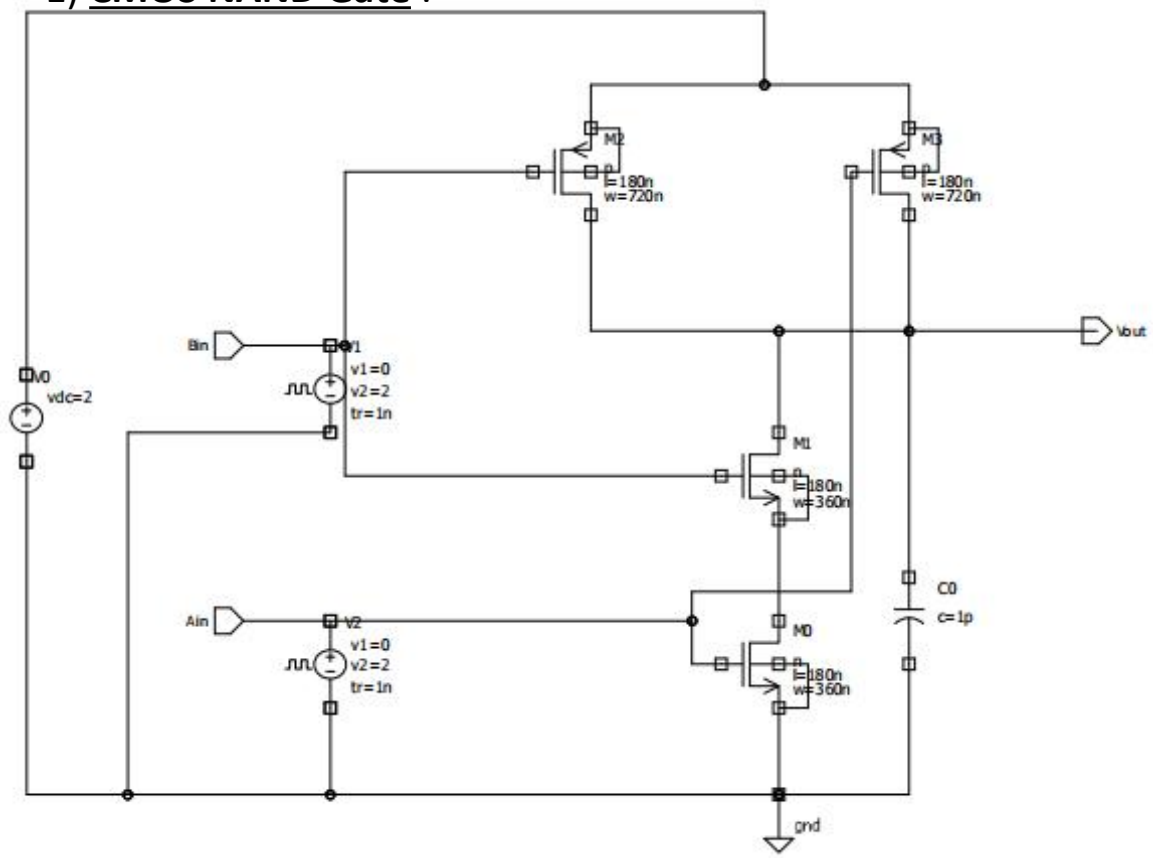


Truth Table

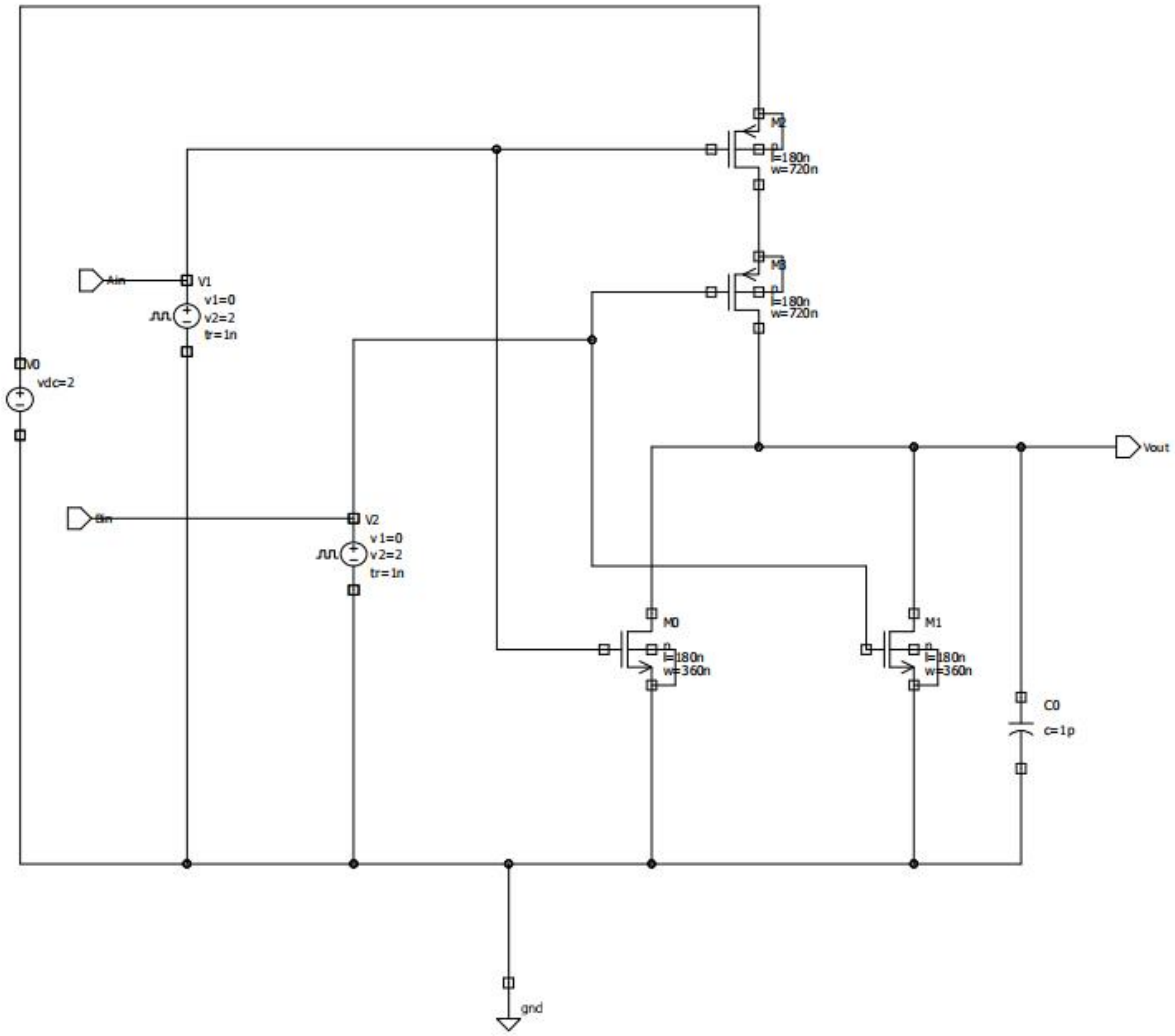
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

● SYMICA CIRCUIT DIAGRAMS :-

1) **CMOS NAND Gate** :-

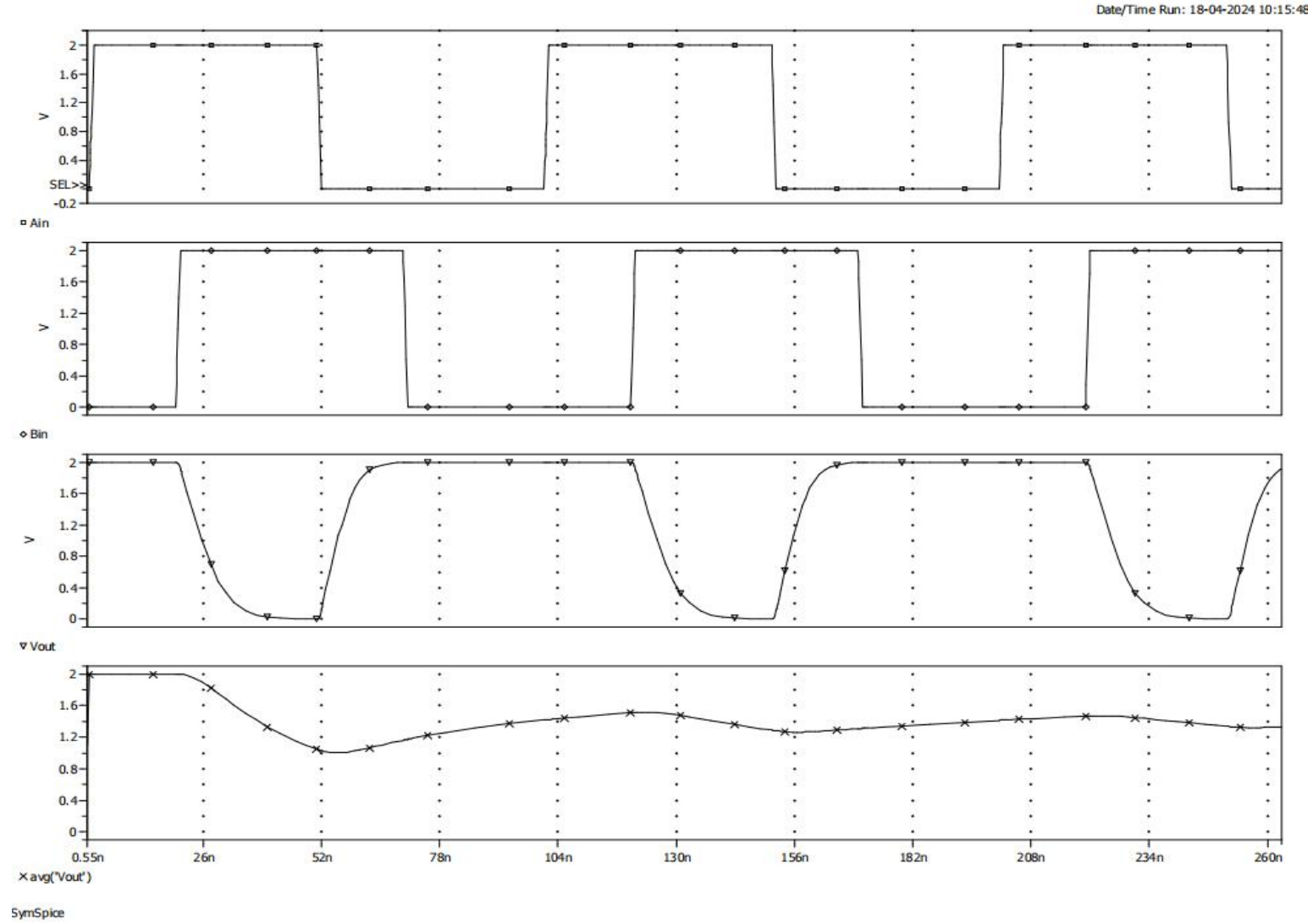


2) **CMOS NOR Gate** :-



● SIMULATION RESULTS :-

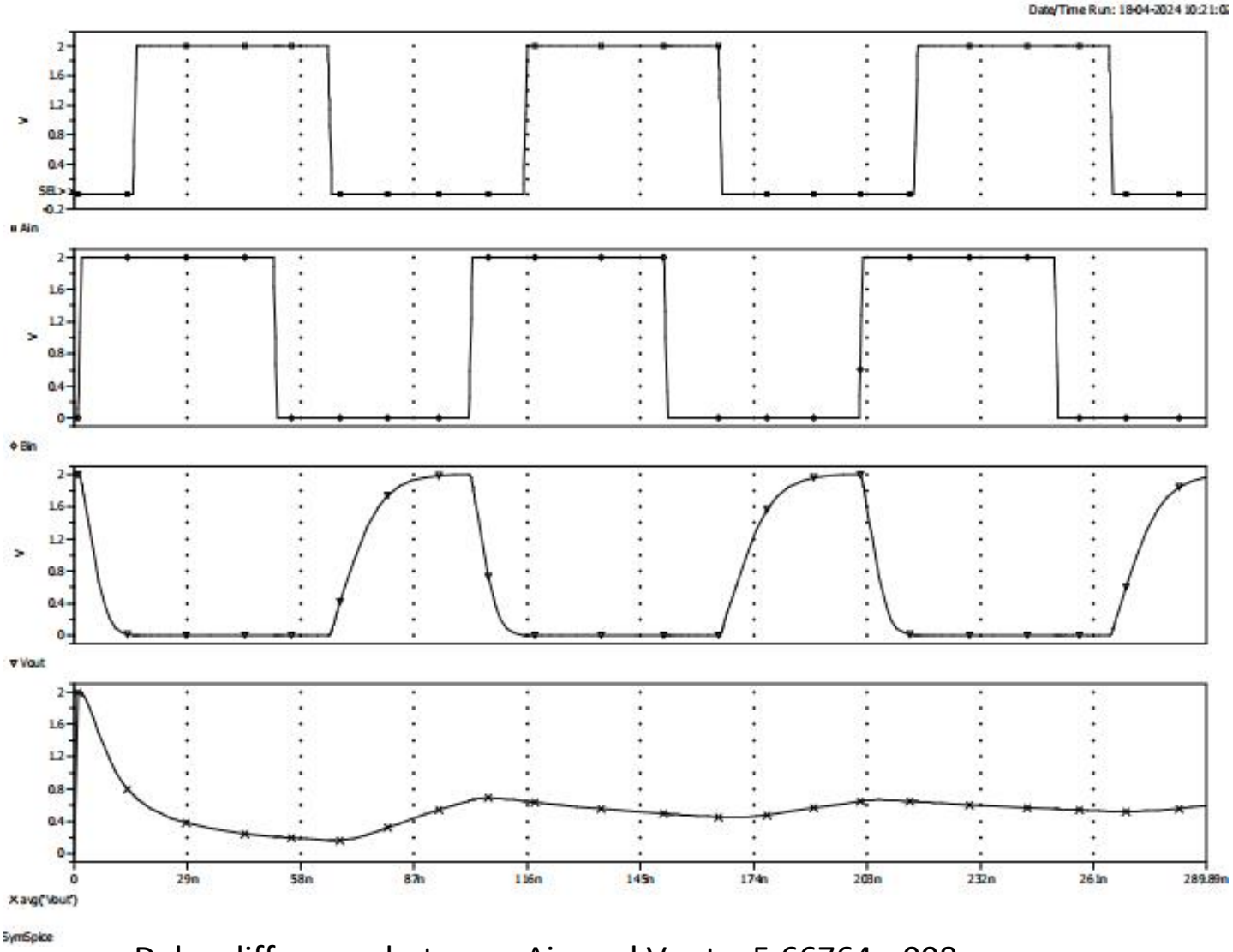
1) **CMOS NAND Gate** :-



Delay difference between Ain and Vout = 2.436765e-008

Delay difference between Bin and Vout = 5.36748e-008

2) CMOS NOR Gate :-



Delay difference between Ain and Vout = 5.66764e-008
Delay difference between Bin and Vout = 3.54742e-009

- **<u>RESULT :-</u>** The Universal Logic Gates,i.e., NAND and NOR Gates were implemented using CMOS Technology and the simulations were observed and delay differences between input and output were calculated.

22/04/2024

EXPERIMENT - 11

- **AIM :-** To design and simulate a 2x1 MUX in CMOS Technology and verify its working

- **SOFTWARE USED :-** Symica DE Free Edition

- **THEORY :-**

Multiplexer (MUX) is a data selector which sends single input data at the output based on select line input.

A 2:1 MUX has 2 inputs (A and B), 1 output (Y), and 1 select line (S). Output Y will be A or B based on 0 or 1 input at the select line (S). If the select line is “0” output Y will be A and if the select line is “1” then output Y will be B.

2:1 MUX using CMOS will be designed using 2 parts: PMOS (pull-up lattice) and NMOS (pull-down lattice). PMOS circuit is connected to supply voltage VDD and NMOS circuit is connected to ground GND.

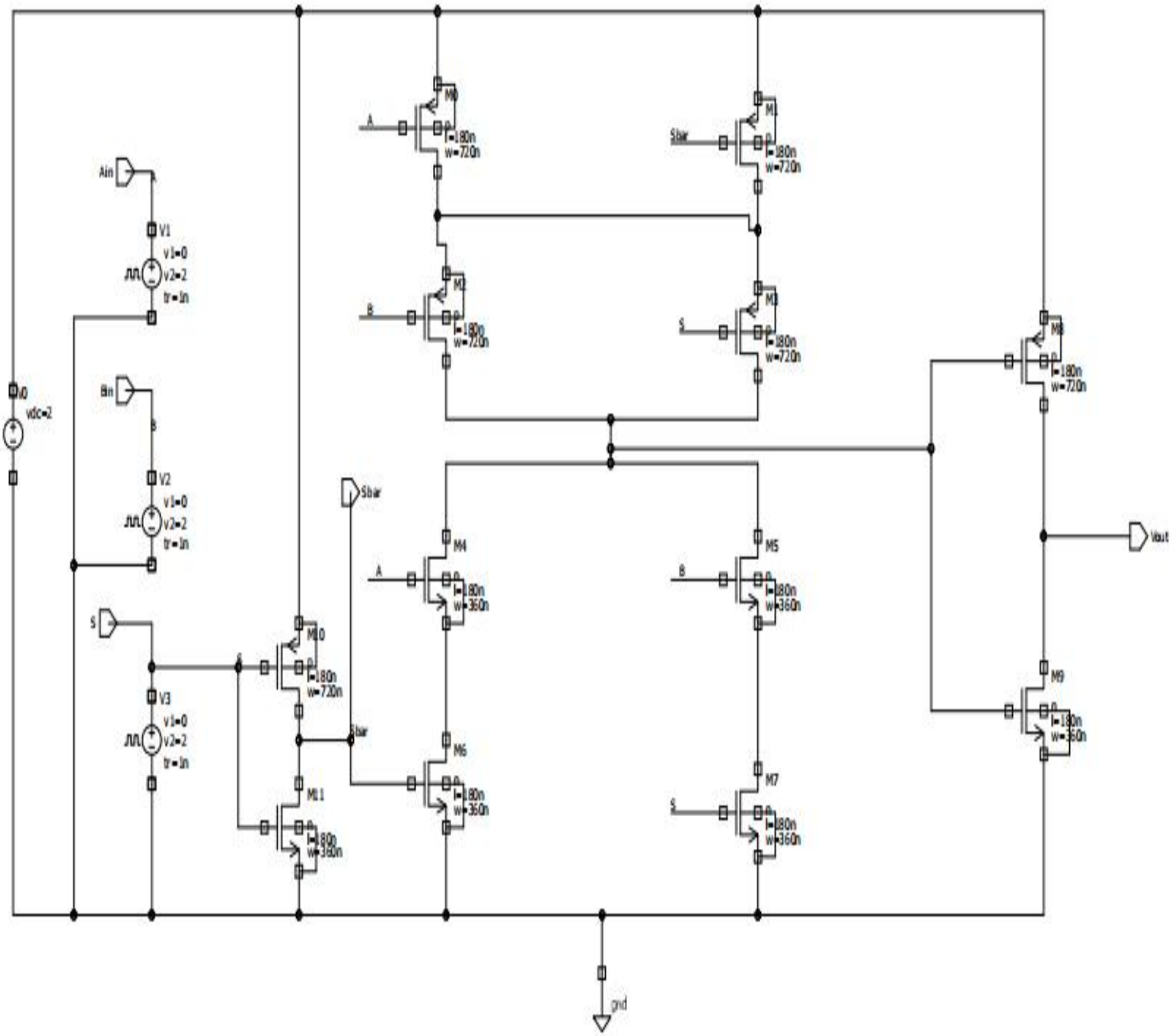
The equation for output Y will be Y=A.Sbar + B.S . According to circuit design rules, ASbar and BS will be connected in parallel in PMOS lattice and it will be connected in series in NMOS lattice. We know that the output of CMOS is always inverted so we have to connect the CMOS inverter circuit at the output.

In the Circuit Waveform, we will verify the above implementation using clock pulse. Output Y will have the same clock pulse sequence as A when S will be “0” and it will have the same clock pulse sequence as B when S will be “1”.

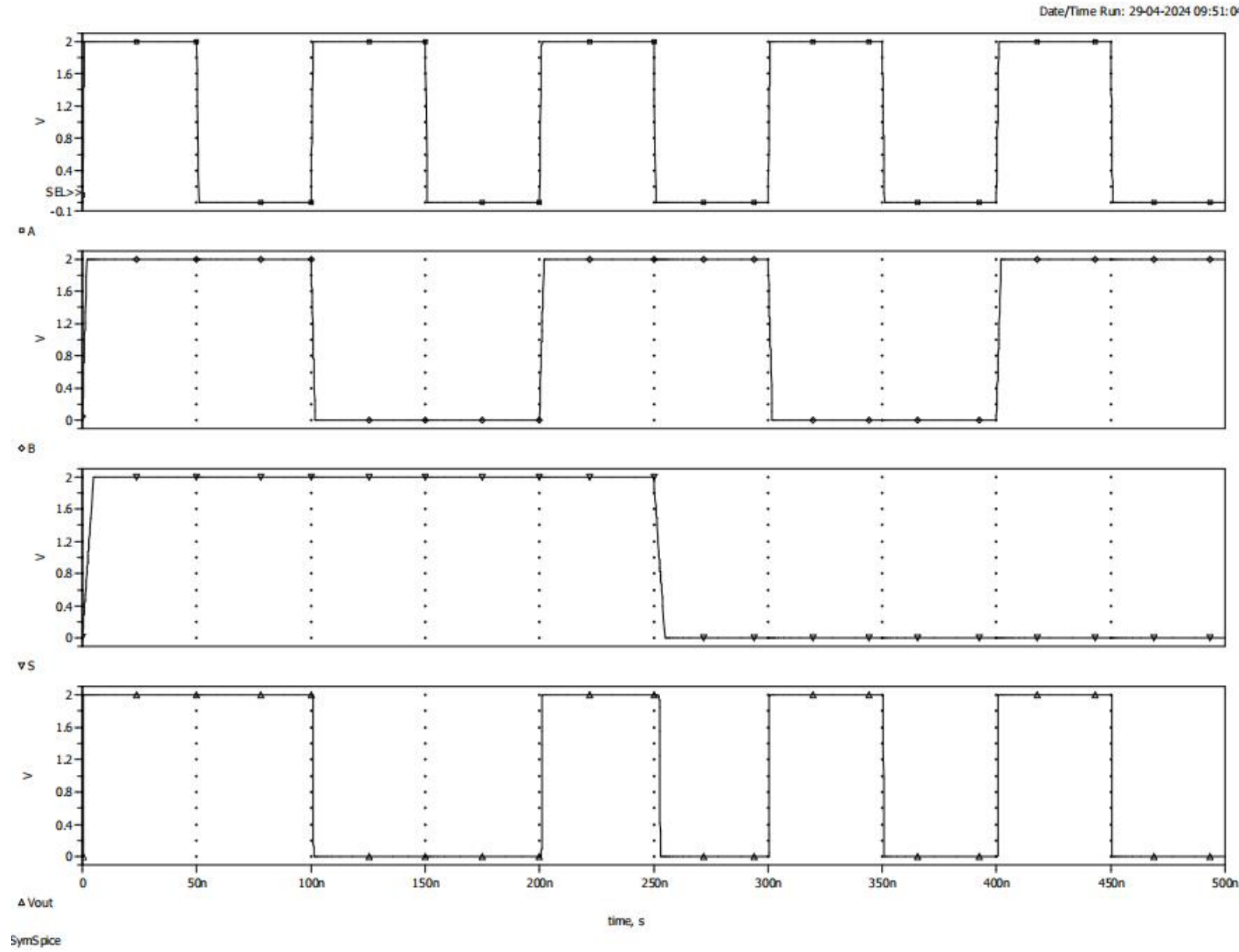
Truth Table for 2x1 MUX :-

S	A	B	Y
0	0	x	0
0	1	x	1
1	x	0	0
1	x	1	1

- **SYMICA CIRCUIT DIAGRAMS :-**



- **SIMULATION RESULTS :-**



- **RESULT :-** 2x1 MUX using CMOS Technology was implemented using CMOS Technology and the result was verified by observing the waveform.