60 DAYS OF PRAYER

BY

PUNCHAVALA MITHUN CHAKRAVARTHI

Mithun Chakravarthi Bungatavala Verify now

Attended Vellore Institute of Technology

Vellore, Tamil Nadu, India · Contact info

1,143 followers · 500+ connections

Vellore Institute of Technology

Contact info:

chakravarthibmithun@gmail.com

Mithun Chakravarthi Bungatavala | LinkedIn

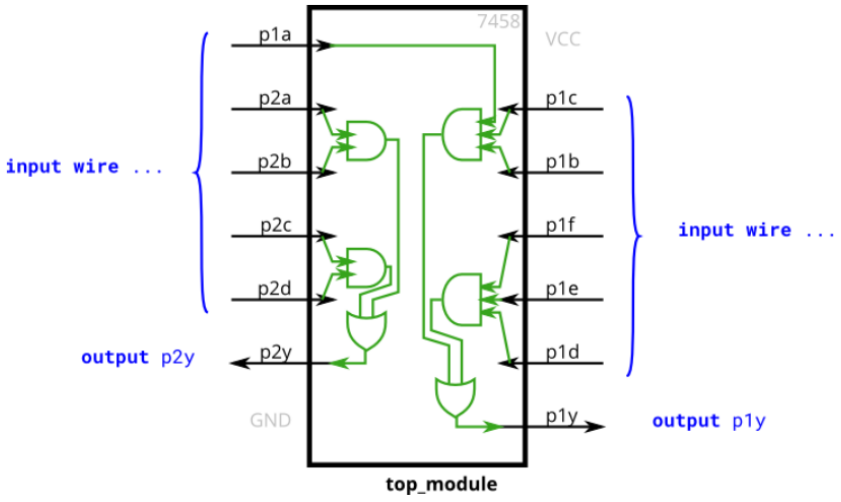
DAY 1

7458

wire_decl Previous Next vector0

The 7458 is a chip with four AND gates and two OR gates. This problem is slightly more complex than 7420.

Create a module with the same functionality as the 7458 chip. It has 10 inputs and 2 outputs. You may choose to use an `assign` statement to drive each of the output wires, or you may choose to declare (four) wires for use as intermediate signals, where each internal wire is driven by the output of one of the AND gates. For extra practice, try it both ways.

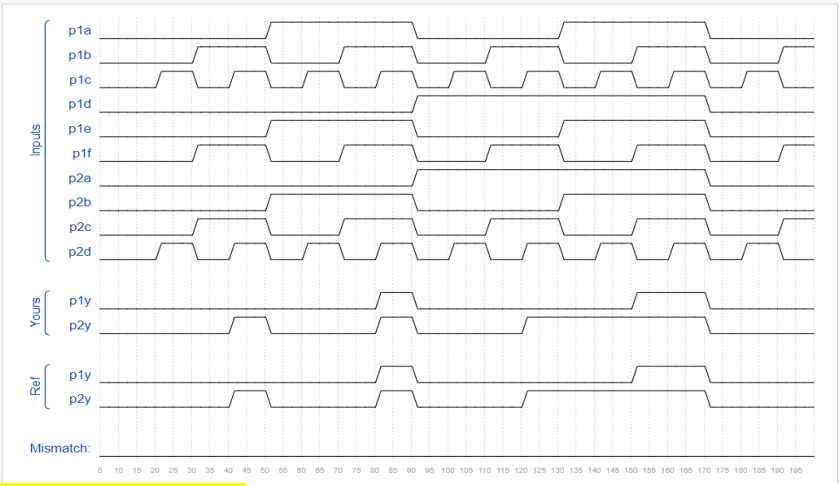


Code for the circuit:

```
module top_module (
    input p1a, p1b, p1c, p1d, p1e, p1f,
    output p1y,
    input p2a, p2b, p2c, p2d,
    output p2y );
    assign p2y=(p2a&p2b)|(p2c&p2d);
    assign p1y=(p1a&p1b&p1c)|(p1f&p1e&p1d);

endmodule
```

Timing diagrams for selected test cases

These are timing diagrams from some of the test cases we used. They may help you debug your circuit. The diagrams show inputs to the circuit, outputs from your circuit, and the expected reference outputs. The "Mismatch" trace shows which cycles your outputs don't match the reference outputs (0 = correct, 1 = incorrect).



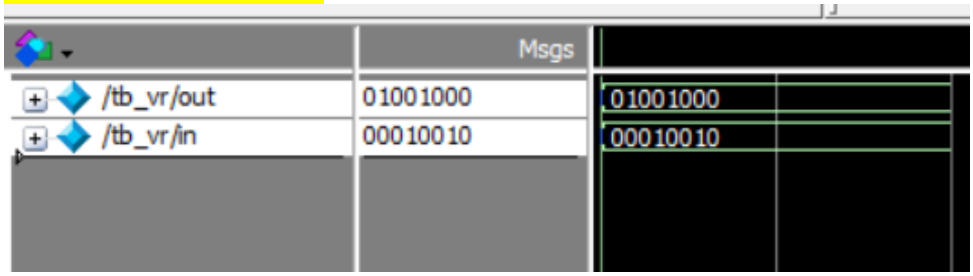
b) Reversing a vector

M:/DSD/50-Days RTL/Vector_Reversal.v (/tb_vr) - Default *

Ln#

```
1   //Reversing a vector of 8 bit using
2   //generate variable//
3   module Vector_Reversal (
4       input [7:0] in,
5       output [7:0] out
6   );
7       generate
8           genvar i;
9           for (i=0; i<8; i = i+1) begin
10              assign out[i] = in[8-i-1];
11          end
12      endgenerate
13  endmodule
14
15  //TEST-BENCH//
16
17  module tb_vr();
18  wire [7:0]out;
19  reg [7:0]in;
20  Vector_Reversal v1(in,out);
21  initial begin
22  in=01001000;
23  #10 $stop;
24  end
25  endmodule
```

SIMULATION WAVEFORM:



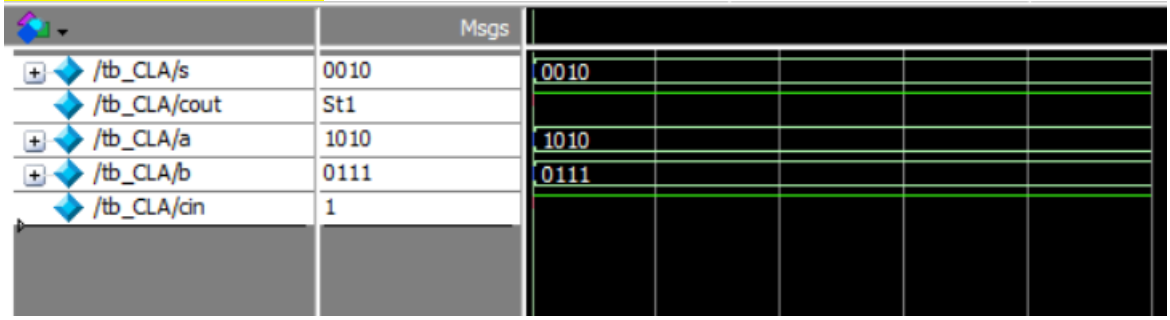
DAY 2

IMPLEMENTATION OF CARRY LOOK AHEAD ADDER

VERILOG CODE WITH TESTBENCH:

```
1  //DAY2 -CARRY LOOK AHEAD ADDER//
2  //WE KNOW FOR FULL ADDER SUM=A^B//
3  // OUTPUT CARRY= AB+Cin(A^B)//
4  //HERE A&B IS CARRY GENERATOR//
5  //AND A^B IS CALLED CARRY PROPOGATOR//
6  module cla(input [3:0] a, input [3:0] b, input cin, output [3:0] s, output cout);
7   wire [3:0] g, p, c;
8   assign g = a & b;
9   assign p = a ^ b;
10  assign c[0] = cin;
11  assign c[1] = g[0] | (p[0] & c[0]);
12  assign c[2] = g[1] | (p[1] & c[1]);
13  assign c[3] = g[2] | (p[2] & c[2]);
14  assign cout =g[3] | (p[3] & c[3]);
15  assign s = p ^ c;
16 endmodule
17
18  //TESTBENCH//
19 module tb_CLA();
20  wire [3:0]s;
21  wire cout;
22  reg [3:0]a,b;
23  reg cin;
24  cla c1(a,b,cin,s,cout);
25 initial begin
26  a=4'b1010;
27  b=4'b0111;
28  cin=1'b1;
29  #10 $stop;
30  end
31  endmodule
```

SIMULATION WAVEFORM:



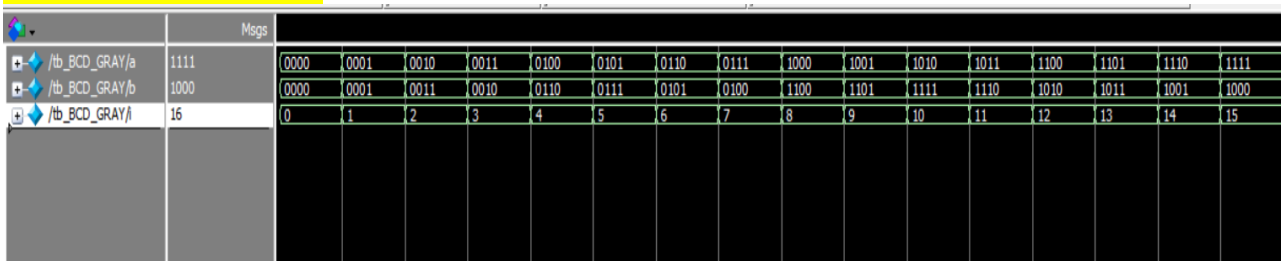
DAY 3

1.IMPLEMENTATION OF BINARY TO GARY CONVERTER

VERILOG CODE WITH TESTBENCH

```
Ln#
 1  //DAY 3 //
 2  //BINARY TO GRAY CONVERTER//
 3
 4  module BCD_GRAY(a,b);
 5  input [3:0]a;
 6  output [3:0]b;
 7  generate
 8  genvar i;
 9  assign b[3]=a[3];
10  for(i=0;i<3;i=i+1) begin
11  assign b[i]=a[i]^a[i+1];
12  end
13  endgenerate
14  endmodule
15
16  //TESTBENCH//
17
18  module tb_BCD_GRAY();
19  wire [3:0]b;
20  reg [3:0]a;
21  integer i;
22  BCD_GRAY b1(a,b);
23  initial
24  begin
25  for(i=0;i<16;i=i+1)
26  begin
27  a=i; #5;
28  end
29  #200 $stop;
30  end
31  endmodule
```

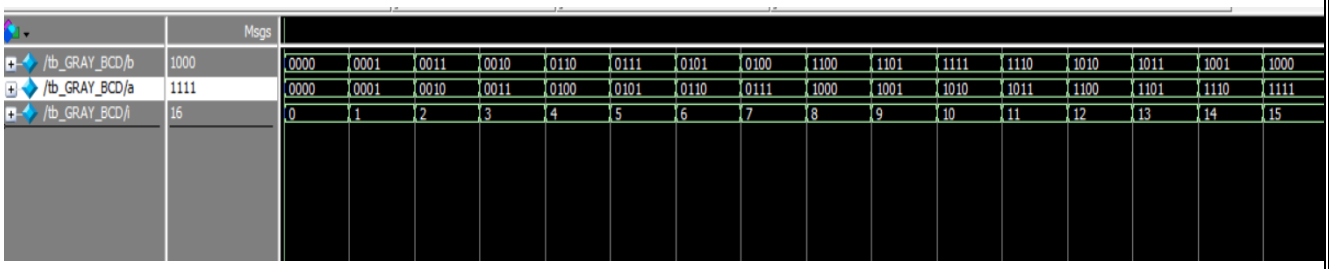
SIMULATION WAVEFORM:



2. IMPLEMENTATION OF GRAY TO BINARY CONVERTER VERILOG CODE WITH TESTBENCH

```
Ln#
 1  //DAY 3 //
 2  //GRAY TO BINARY CONVERTER//
 3
 4  module GRAY_BCD(a,b);
 5  input [3:0]a;
 6  output [3:0]b;
 7  generate
 8  genvar i;
 9  assign b[3]=a[3];
10  for(i=2;i>=0;i=i+1) begin
11  assign b[i]=a[i]^b[i+1];
12  end
13  endgenerate
14  endmodule
15
16  //TESTBENCH//
17
18  module tb_GRAY_BCD();
19  wire [3:0]b;
20  reg [3:0]a;
21  integer i;
22  BCD_GRAY b1(a,b);
23  initial
24  begin
25  for(i=0;i<16;i=i+1)
26  begin
27  a=i; #5;
28  end
29  #80 $stop;
30  end
31  endmodule
32
```

SIMULATION WAVEFORM



DAY-4

DESIGN A CIRCUIT FOR 2'S COMPLEMENTOR USING MULTIPLEXERS AND ADDERS

Day-4 Design a circuit for 2's complement using multiplexer and Adder's.

a) 2's complement ?

Let us consider a number $a = 1010$

2's complement = 1's complement + 1'b1

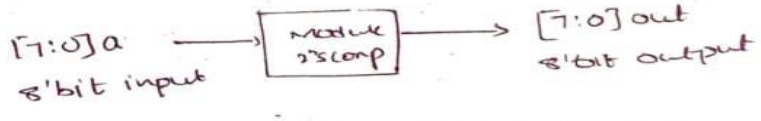
1's complement of $a = 0101$

$$2\text{'s comp} = 1\text{'s comp} + 1'b1$$

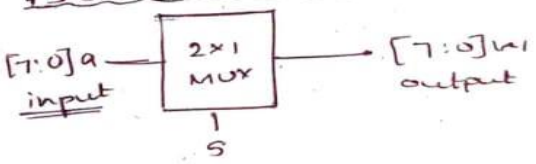
$$= 0101 + 1 = 0110$$

∴ 2's complement of $1010 = 0110$

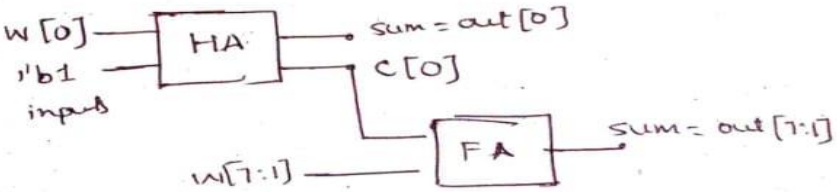
b) Designing of circuit



0. 1's complement using multiplexer



1.



So final output will be [7:0] out from Half Adder and Full Adder.

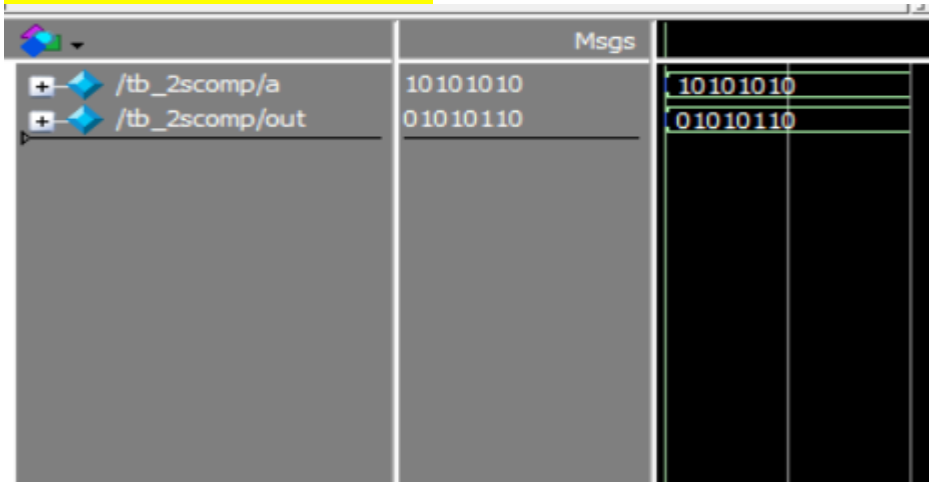
VERILOG CODE FOR HALF ADDER,FULL ADDER,MULTIPLEXER

```
1   //HALF ADDER VERILOG CODE//
2   module Half_Adder(input a,b,output s,c);
3   assign s=a^b;
4   assign c=a&b;
5   endmodule
6
7   //FULL ADDER VERILOG CODE//
8   module Full_Adder(input a,b,cin,output s,cout);
9   assign s=a^b^cin;
10  assign cout=(a&b)|(cin&(a^b));
11  endmodule
12
13  //MULTIPLEXER VERILOG CODE//
14  module mux(input i0,i1,s,output reg out);
15  always@(*)
16  begin
17  case(s)
18  1'b0:assign out=i0;
19  1'b1:assign out=i1;
20  endcase
21  end
22  endmodule
```

INSTANTIATING THESE BLOCKS ACCORDING TO LOGIC DISCUSSED BEFORE

```
// 1st BLOCK //
// THIS BLOCK converts given binary number to 1'S COMPLEMENT //
module complement_2s(a,out);
input [7:0]a;
output [7:0]out;
wire [7:0]w;
genvar j;
generate
begin
for(j=0;j<8;j=j+1)
begin
mux m(1'b1,1'b0,a[j],w[j]);
end
end
endgenerate

//THIS BLOCK IS FOR 2'S COMPLEMENT//
wire [7:0]c;
// HALF ADDER IS USED TO ADD THE LEAST SIGNIFICANT BIT WITH 1'B1 //
Half_Adder h1(w[0],1'b1,out[0],c[0]);
genvar i;
generate
begin
for(i=1;i<8;i=i+1)
begin
//FULL ADDER IS USED TO ADD THE OTHER PART OF INPUT WITH THE CARRY FROM HALF ADDER//
Full_Adder F1(w[i],1'b0,c[i-1],out[i],c[i]);
end
end
endgenerate
endmodule


//TEST BENCH//
module tb_2scomp();
reg [7:0]a;
wire [7:0]out;
complement_2s c1(a,out);
initial begin
a=8'b10101010;
#10 $stop;
end
endmodule
```

OUTPUT SIMULATION WAVEFORM:



DAY-5

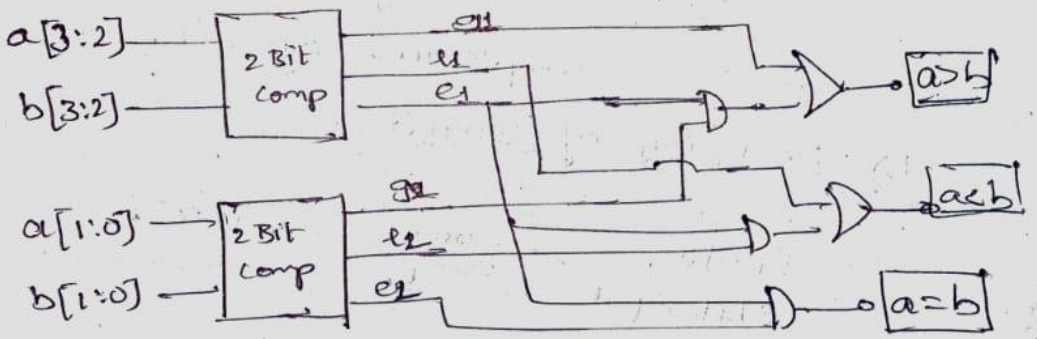
a. DESIGN A 4 BIT COMPARATOR USING 2 BIT COMPARATOR

Day-5

Design a 4 Bit comparator by 2 Bit Comparator

a) 2 Bit Comparator

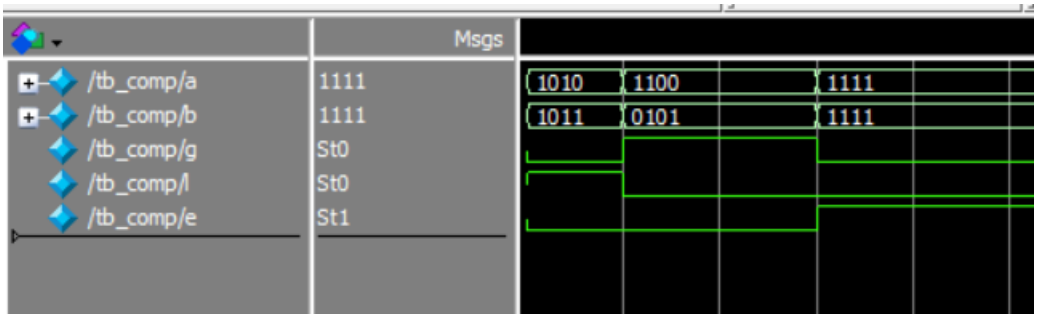
b) 4 Bit Comparator



VERILOG CODE THE THE DESIGN:

```
1  //DAY 5//
2  //DESIGN A 4 BIT COMPARATOR USING 2 BIT COMPARATOR//
3  //CODE FOR 2 BIT COMPARATOR//
4  module Comp_2bit(a,b,g,l,e);
5  input [1:0]a,b;
6  output g,l,e;
7  assign g=(a>b)?1:0;
8  assign l=(a<b)?1:0;
9  assign e=(a==b)?1:0;
10 endmodule
11
12 //CODE FOR 4 BIT COMPARATOR USING 2 BIT COMPARATOR//
13 module Comp_4bit(a,b,g,l,e);
14 input [3:0]a,b;
15 output g,l,e;
16 wire g1,g2,l1,l2,e1,e2;
17 wire w1,w2,w3;
18 Comp_2bit c1(a[3:2],b[3:2],g1,l1,e1);
19 Comp_2bit c2(a[1:0],b[1:0],g2,l2,e2);
20 assign g=(e1&g2)|g1;
21 assign l=(e1&l2)|l1;
22 assign e=(e1&e2);
23 endmodule
24
25 //TESTBENCH FOR THE DESIGN//
26 module tb_comp();
27 wire g,l,e;
28 reg [3:0]a,b;
29 Comp_4bit c1(a,b,g,l,e);
30 initial begin
31 a=4'b1010;b=4'b1011;
32 #10 a=4'b1100;b=4'b0101;
33 #20 $stop;
34 end
35 endmodule
```

SIMULATION WAVEFORM:



DAY6

PRIORITY ENCODER

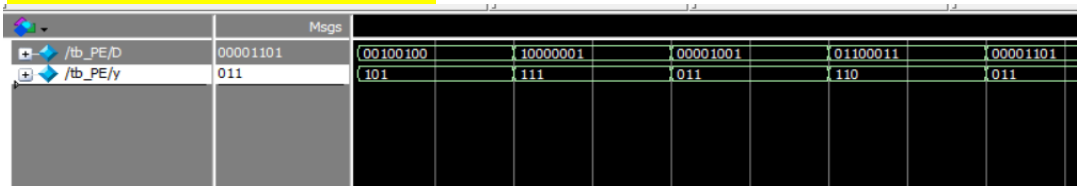
TRUTH TABLE

	Inputs								Output
Decimal Number	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	Binary ($Y_2Y_1Y_0$)
0	1	0	0	0	0	0	0	0	000
1	X	1	0	0	0	0	0	0	001
2	X	X	1	0	0	0	0	0	010
3	X	X	X	1	0	0	0	0	011
4	X	X	X	X	1	0	0	0	100
5	X	X	X	X	X	1	0	0	101
6	X	X	X	X	X	X	1	0	110
7	X	X	X	X	X	X	X	1	111

Verilog code

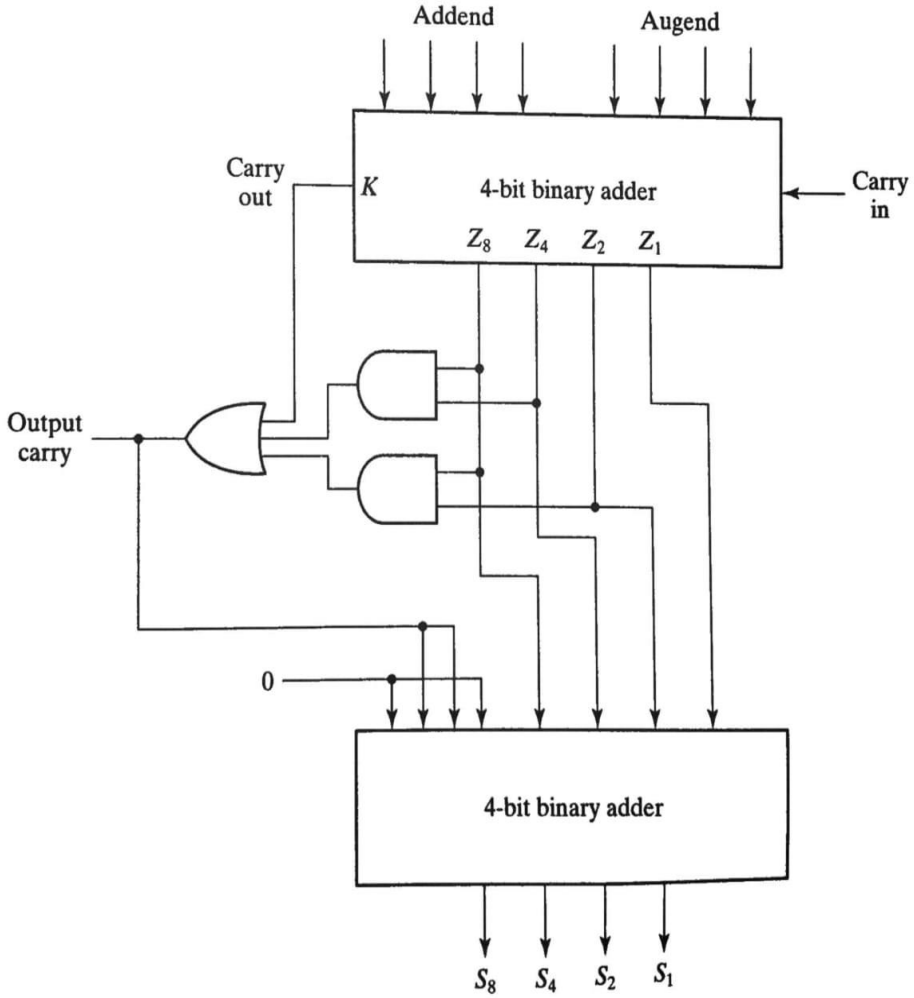
```
1   //DAY=6 //
2   //PRIORITY ENCODER//
3   module priority_encoder(
4     input [7:0] D,
5     output reg [2:0] y);
6
7     always@(D) begin
8       casex(D)
9           8'b0000_0001: y = 3'b000;
10          8'b0000_001x: y = 3'b001;
11          8'b0000_01xx: y = 3'b010;
12          8'b0000_1xxx: y = 3'b011;
13          8'b0001_xxxx: y = 3'b100;
14          8'b001x_xxxx: y = 3'b101;
15          8'b01xx_xxxx: y = 3'b110;
16          8'b1xxx_xxxx: y = 3'b111;
17
18
19
20        default: $display("Invalid data received");
21      endcase
22    end
23  endmodule
24
25  module tb_PE;
26    reg [7:0] D;
27    wire [2:0] y;
28
29    priority_encoder pri_enc(D, y);
30
31    initial begin
32      $monitor("D = %b -> y = %0b", D, y);
33      repeat(5) begin
34        D=$random; #10;
35      end
36    end
37  endmodule
```

OUTPUT SIMULATION WAVEFORM:



Day-7

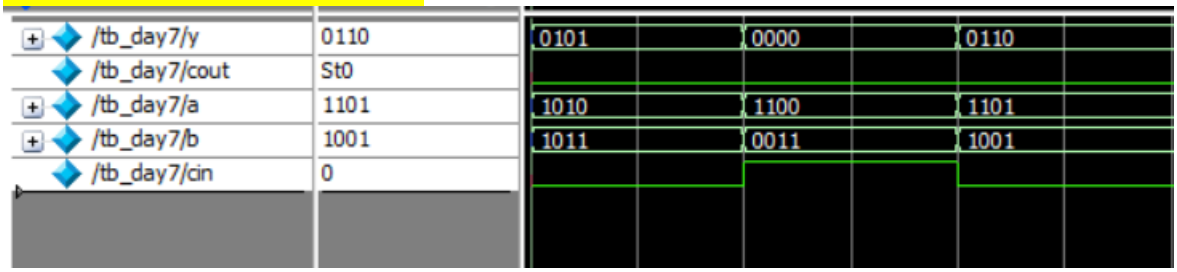
1.WRITE A VERILOG CODE FOR THE CIRCUIT



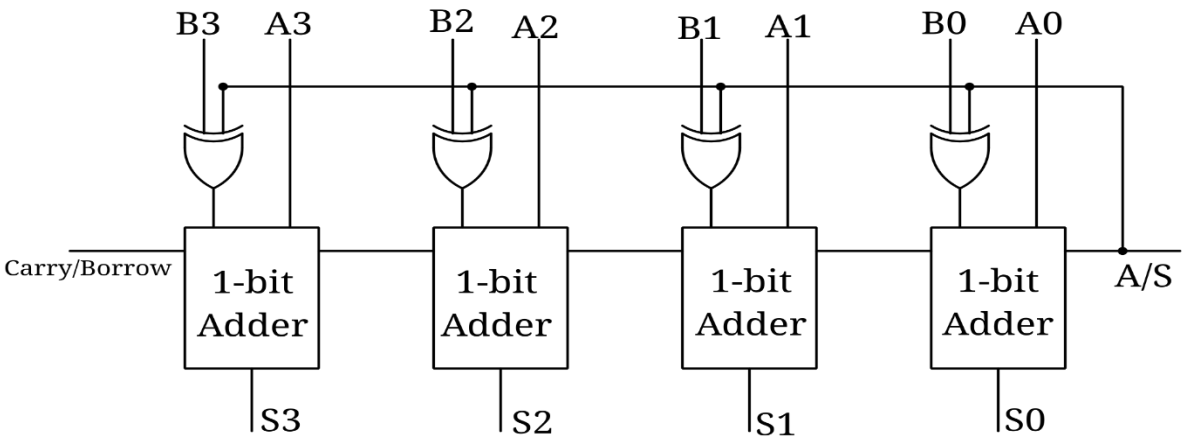
VERILOG CODE WITH TESTBENCH

```
1  //DAY 7 DESIGNING THE CIRCUIT//
2  //THE GIVEN BLOCK DIAGRAM CORRESPONDS TO BCD ADDERR//
3  //LET US FIRST CREATE A FULL ADDER//
4  module FA(a,b,cin,cout,y);
5  input [3:0]a,b;
6  input cin;
7  output [3:0]y;
8  output cout;
9  assign {cout,y}=a+b+cin;
10 endmodule
11
12 // INSTANTIATING THE CREATED FULL ADER ACCORDING TO DESIGN//
13 module topmodule(a,b,cin,cout,y);
14 input [3:0]a,b;
15 input cin;
16 output [3:0]y;
17 output cout;
18 wire [3:0]o;
19 wire w1,w2;
20 FA f1(a,b,cin,w1,o);
21 assign w2=(w1|(o[3]&o[2])|(o[3]&o[1]));
22 FA f2({0,w2,w2,0},o,1'b0,cout,y);
23 endmodule
24
25 //TESTBENCH//
26 module tb_day7();
27 wire [3:0]y;
28 wire cout;
29 reg [3:0]a,b;
30 reg cin;
31 topmodule t1(a,b,cin,cout,y);
32 initial begin
33 a=4'b1010;b=4'b1011;cin=1'b0;
34 #10 a=4'b1100;b=4'b0011;cin=1'b1;
35 #10 a=4'b1101;b=4'b1001;cin=1'b0;
36 #35 $stop;
37 end
38 endmodule
```

OUTPUT SIMULATION WAVEFORM



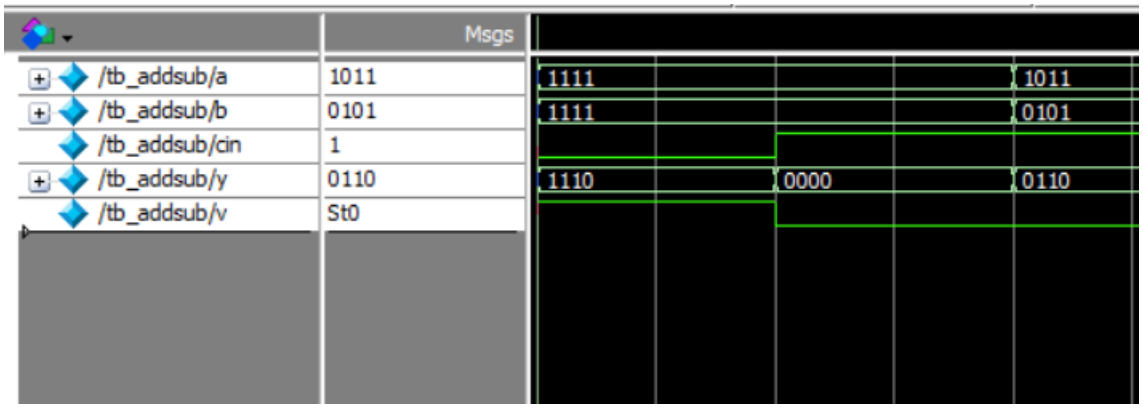
2. Adder Subtractor using a control input



VERILOG CODE WITH TESTBENCH

```
//DAY7 ADDER SUBTRACTOR CIRCUIT WITH CONTROL INPUT//
module Add_Sub(a,b,cin,y,v);
input [3:0]a,b;
input cin;
output reg [3:0]y;
output reg v;
always@(*)
begin
case(cin)
1'b0: {v,y}=a+b;
1'b1:{y}=a+(~b)+1'b1;
endcase
end
endmodule
//TESTBENCH//
module tb_addsub();
reg [3:0]a,b;
reg cin;
wire [3:0]y;
wire v;
Add_Sub a1(a,b,cin,y,v);
initial begin
cin=1'b0;a=4'b1111;b=4'b1111;
#10 cin=1'b1;
#10 a=4'b1011;b=4'b0101;
#30 $stop;
end
endmodule
```

OUTPUT SIMULATION WAVEFORM:



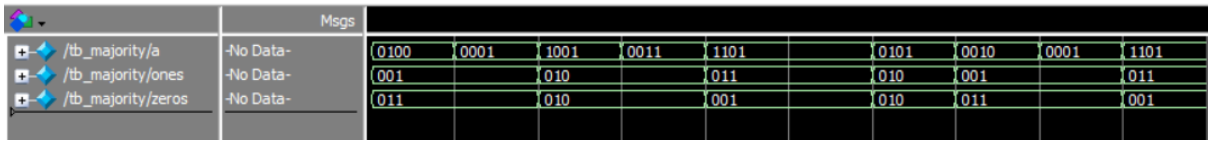
DAY-8

ZEROS COUNTER,ONES COUNTER

VERILOG CODE WITH TESTBENCH

```
//DAY-8//
//THIS CIRUCIUT COUNTS NUMBER OF 0'S AND 1'S//
//AND OUTPUT IS HIGH IF NUMBER OF ZEROES> ONES//
module ZeroCountDetector (input [3:0]a,
output reg [2:0] ones,zeros);
  integer i;
  // Count the number of zeros and ones
  always @* begin
   ones=0;
   for(i=0;i<4;i=i+1)
   if(a[i]==1'b1)
   ones=ones+1'b1;
   end
   always@* begin
   zeros=0;
   for(i=0;i<4;i=i+1)
   if(a[i]==1'b0)
   zeros=zeros+1'b1;
   end
endmodule

//TESTBENCH//
module tb_majority();
 reg [3:0] a;
  wire [2:0] ones,zeros;
  ZeroCountDetector z1(a,ones,zeros);
  initial begin
    $monitor("a= %b,ones = %2d,zeros=%2d",a,ones,zeros);
    repeat(10) begin
      a=$random; #10;
    end
  end
endmodule
```

OUTPUT SIMULATION WAVEFORM:



```
# a= 0100 , ones =  1,zeros= 3
# a= 0001 , ones =  1,zeros= 3
# a= 1001 , ones =  2,zeros= 2
# a= 0011 , ones =  2,zeros= 2
# a= 1101 , ones =  3,zeros= 1
# a= 0101 , ones =  2,zeros= 2
# a= 0010 , ones =  1,zeros= 3
# a= 0001 , ones =  1,zeros= 3
# a= 1101 , ones =  3,zeros= 1
```

DAY-9

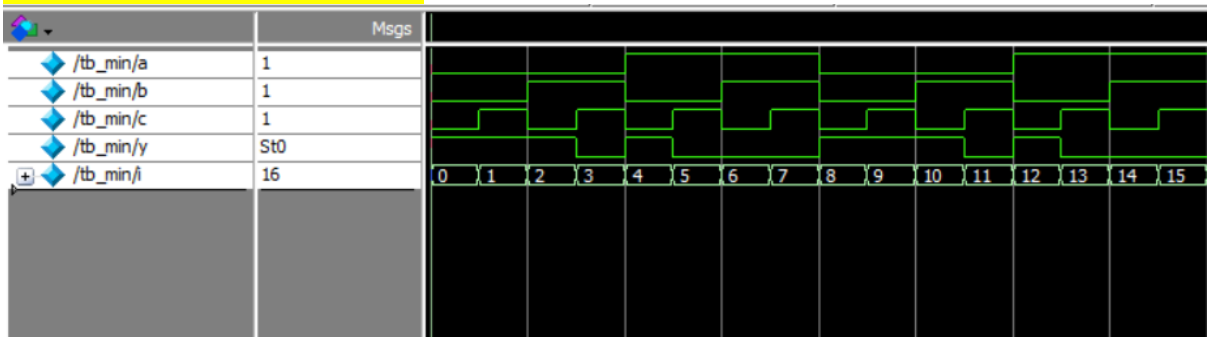
IMPLEMENTATION OF MINORITY DETECTOR

VERILOG CODE WITH TESTBENCH

```
//DAY-9//
//A Minority detector is a digital circuit that detects
//if number of 0's in input is more than number of 1's

module minority_detector(input [3:0]a,output y);
wire w1,w2,w3;
assign w1=~a|~b;
assign w2=~c|~b;
assign w3=~c|~a;
assign y=(w1&w2&w3);
endmodule

//TESTBENCH FOR DESIGN
module tb_min();
reg a,b,c;
wire y;
integer i;
minority_detector m1(a,b,c,y);
initial begin
for(i=0;i<16;i=i+1)
begin
{a,b,c}=i; #5;
$monitor("a=%b,b=%b,c=%b,y=%b",a,b,c,y);
end
#45 $stop;
end
endmodule
```

```
# a=0,b=0,c=1,y=1
# a=0,b=1,c=0,y=1
# a=0,b=1,c=1,y=0
# a=1,b=0,c=0,y=1
# a=1,b=0,c=1,y=0
# a=1,b=1,c=0,y=0
# a=1,b=1,c=1,y=0
# a=0,b=0,c=0,y=1
# a=0,b=0,c=1,y=1
# a=0,b=1,c=0,y=1
# a=0,b=1,c=1,y=0
# a=1,b=0,c=0,y=1
# a=1,b=0,c=1,y=0
# a=1,b=1,c=0,y=0
# a=1,b=1,c=1,y=0
# a=1,b=1,c=1,y=0
```

OUTPUT SIMULATION WAVEFORM

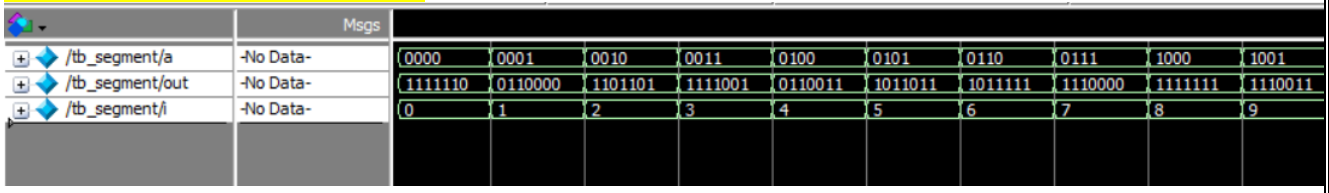


DAY 10 SEVEN SEGMENT DISPLAY
VERILOG CODE WITH TESTBENCH

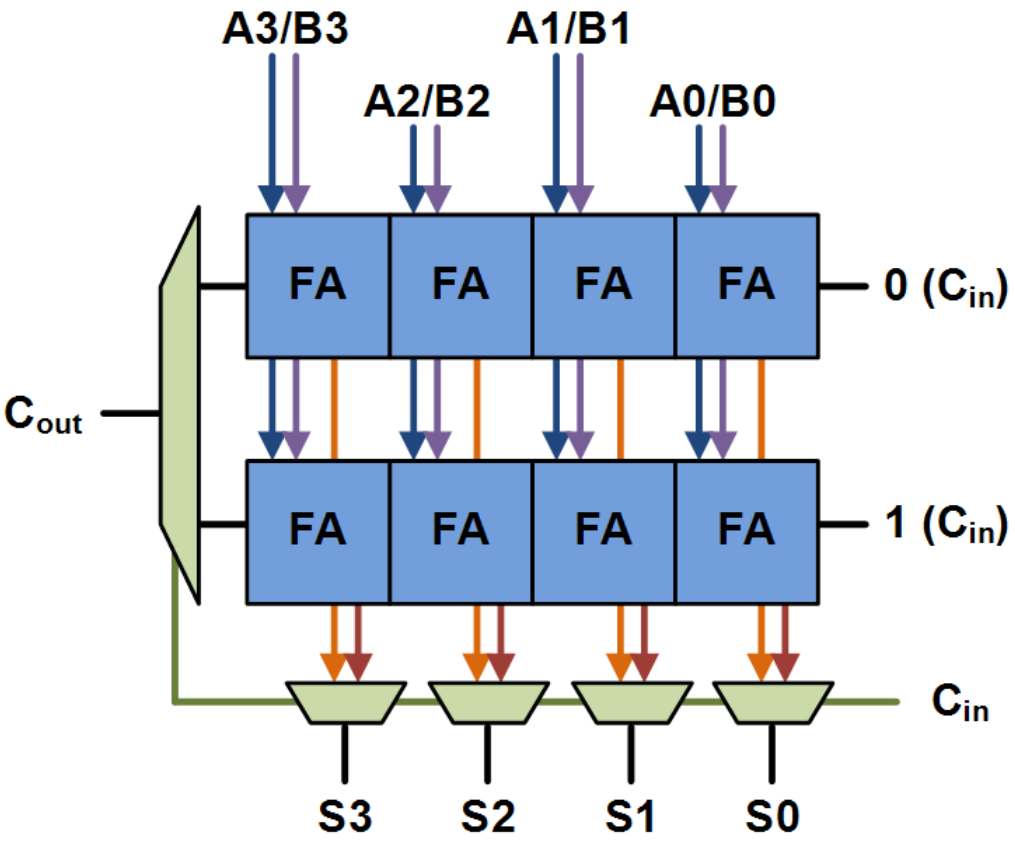
```
//DAY 10 SEVEN SEGMENT DISPALY//
module Seven_Segment(a,out);
input [3:0]a;
output reg [6:0]out;
always@(a)
begin
case(a)
0:out=7'b1111110;
1:out=7'b0110000;
2:out=7'b1101101;
3:out=7'b1111001;
4:out=7'b0110011;
5:out=7'b1011011;
6:out=7'b1011111;
7:out=7'b1110000;
8:out=7'b1111111;
9:out=7'b1110011;
default:out=7'b0000000;
endcase
end
endmodule
//TEST BENCH//
module tb_segment();
reg [3:0]a;
wire [6:0]out;
integer i;
Seven_Segment s1(a,out);
initial begin
for(i=0;i<10;i=i+1)
begin
a=i; #5;
$monitor("a=%d,out=%b",a,out);
end
#45 $stop;
end
endmodule
```

```
VSIM 15> run
# a= 1,out=0110000
# a= 2,out=1101101
# a= 3,out=1111001
# a= 4,out=0110011
# a= 5,out=1011011
# a= 6,out=1011111
# a= 7,out=1110000
# a= 8,out=1111111
# a= 9,out=1110011
# a= 9,out=1110011
```

OUTPUT SIMULATION WAVEFORM



DAY-11 CARRY SELECT ADDER



VERILOG CODE WITH TEST BENCH

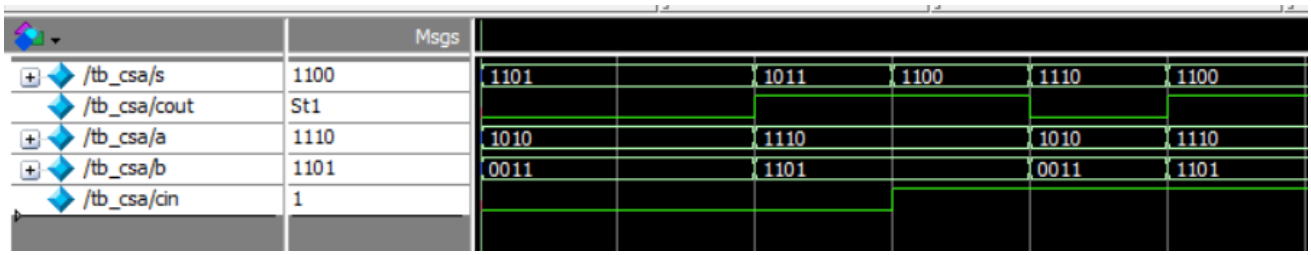
```
//DAY-11 CARRY SELECT ADDER//
module FA(a,b,cin,sum,cout);
input  a,b,cin;
output sum,cout;
assign sum=a^b^cin;
assign cout=(a&b)|(cin&(a^b));
endmodule

module mux21(i0,i1,s,y);
input i0,i1,s;
output reg y;
assign y=(~s&i0)|(s&i1);
endmodule

module CSA(a,b,cin,s,cout);
input [3:0]a,b;
input cin;
output [3:0]s;
output cout;
wire [3:0]w,x,y,z;
FA f1(a[0],b[0],1'b0,w[0],x[0]);
FA f2(a[1],b[1],x[0],w[1],x[1]);
FA f3(a[2],b[2],x[1],w[2],x[2]);
FA f4(a[3],b[3],x[2],w[3],x[3]);

FA f5(a[0],b[0],1'b1,y[0],z[0]);
FA f6(a[1],b[1],z[0],y[1],z[1]);
FA f7(a[2],b[2],z[1],y[2],z[2]);
FA f8(a[3],b[3],z[2],y[3],z[3]);

mux m1(w[0],y[0],cin,s[0]);
mux m2(w[1],y[1],cin,s[1]);
mux m3(w[2],y[2],cin,s[2]);
mux m4(w[3],y[3],cin,s[3]);
mux m5(x[3],z[3],cin,cout);
endmodule
```

```
//TESTBENCH//
module tb_csa();
wire [3:0]s;
wire cout;
reg [3:0]a,b;
reg cin;
CSA c1(a,b,cin,s,cout);
initial begin
cin=1'b0;a=4'b1010;b=1011;
#10 a=4'b1010;b=1011;
#10 a=4'b1110;b=1101;
#10 cin=1'b1;
#10 a=4'b1010;b=1011;
#10 a=4'b1110;b=1101;
end
initial begin
$monitor("a=%b,b=%b,cin=%b,s=%b,cout=%b",a,b,cin,s,cout);
#55 $stop;
end
endmodule
```

TRANSCRIPT WINDOW

```
VSIM 7> run
# a=1010,b=0011,cin=0,s=1101,cout=0
# a=1110,b=1101,cin=0,s=1011,cout=1
# a=1110,b=1101,cin=1,s=1100,cout=1
# a=1010,b=0011,cin=1,s=1110,cout=0
# a=1110,b=1101,cin=1,s=1100,cout=1
```

OUTPUT SIMULATION WAVEFORM



DAY-12

4 BIT BINARY MULTIPILER

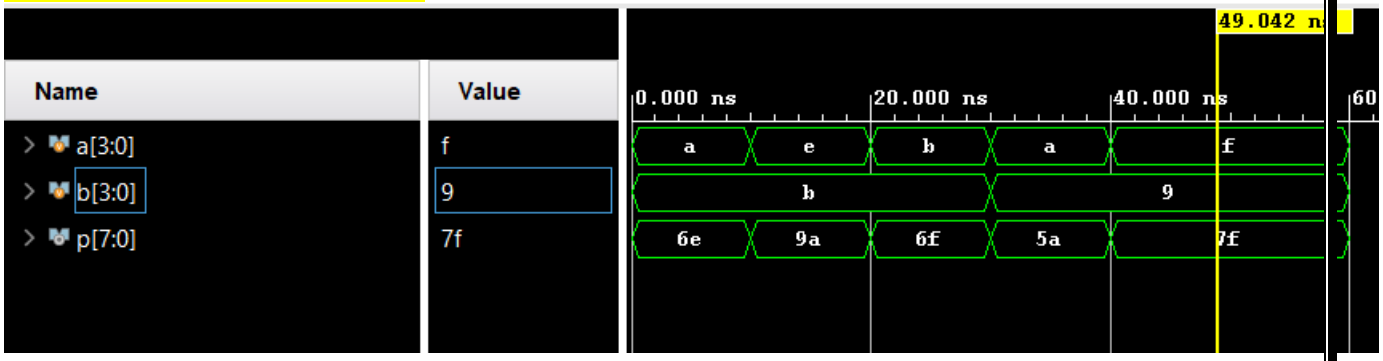
VERILOG CODE

```
//DAY 12 4 BIT BINARY MULTIPLIER//
module multiplier(
    input [3:0] a,b,
    output [7:0] p
    );

    wire [3:0]mo;
    wire [4:0]m1;
    wire [5:0]m2;
    wire [6:0]m3;
    wire [7:0]s1,s2,s3;
    assign m0={4{a[0]}}&b[3:0];
    assign m1={4{a[1]}}&b[3:0];
    assign m2={4{a[2]}}&b[3:0];
    assign m3={4{a[3]}}&b[3:0];
    assign s1=m0+(m1<<1);
    assign s2=s1+(m2<<2);
    assign s3=s2+(m3<<3);
    assign p=s3;
endmodule
```

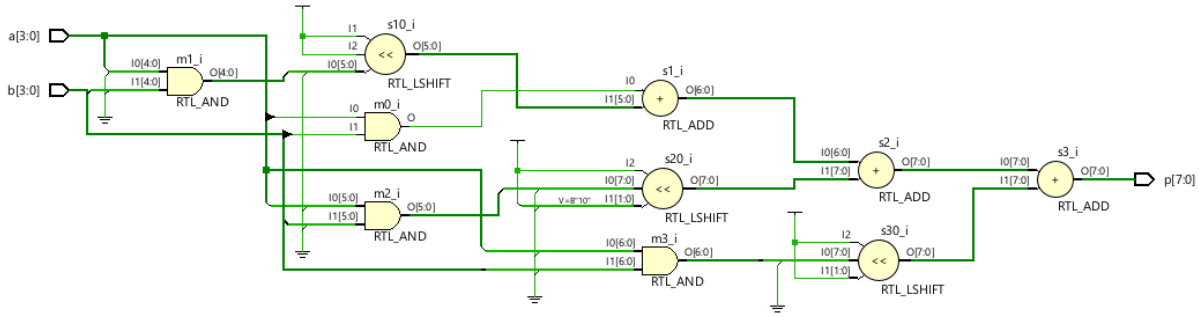
TEST BENCH

```
`timescale 1ns / 1ps
module multiplier_tb();
    reg [3:0] a,b;
    wire [7:0] p;
    multiplier dut(a,b,p);
    initial begin
    a=4'b1010;b=4'b1011;
    #10 a=4'b1110;b=4'b1011;
    #10 a=4'b1011;b=4'b1011;
    #10 a=4'b1010;b=4'b1001;
    #10 a=4'b1111;b=4'b1001;
    end
    initial begin
    #60;
    $stop;
    end
endmodule
```

OUTPUT SIMULATION WAVEFORM



SCHEMATIC CIRCUIT



DAY 13 4 BIT BARREL SHIFTER

A. 4X1 MULTIPLEXER

```
//DAY 13 4X1 MULTIPLEXER//
module mux41(
    input [3:0] i,
    input [1:0] s,
    output reg y
    );
    always@(s)
    begin
    case(s)
    2'b00:y=i[0];
    2'b01:y=i[1];
    2'b10:y=i[2];
    2'b11:y=i[3];
    endcase
    end
endmodule
```

B. BARREL SHIFTER USING MULTIPLEXER

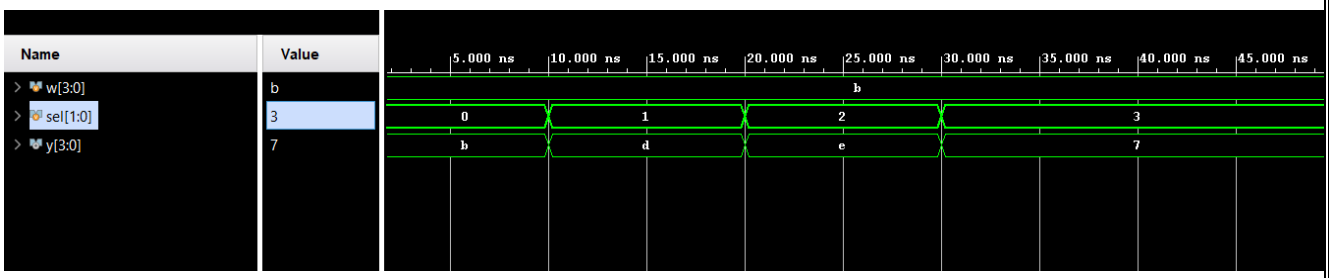
```
//BARREL SHIFTER BY INSTANTIATIN G MULTIPLEXERS//
module barrel_shifter(
     input [3:0] w,
     input [1:0]sel,
     output [3:0] y
     );

     mux41 m1({w[2],w[1],w[0],w[3]}, sel, y[3] );
     mux41 m2({w[1],w[0],w[3],w[2]}, sel, y[2] );
     mux41 m3({w[0],w[3],w[2],w[1]}, sel, y[1] );
     mux41 m4({w[3],w[2],w[1],w[0]}, sel, y[0] );

endmodule
```

C. TEST BENCH

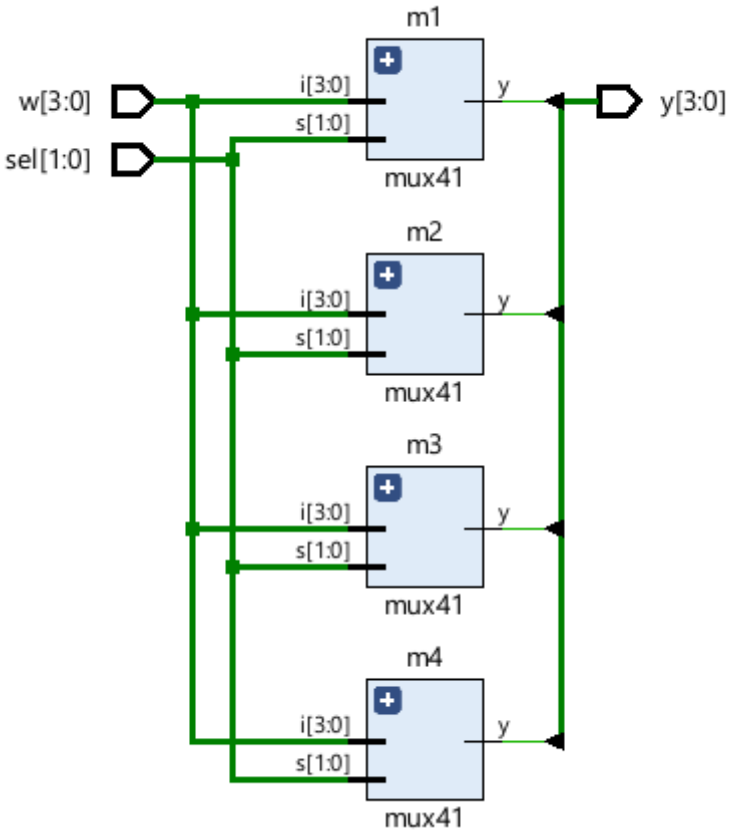
```
//TEST BENCH//
module tb_Barrel_shifter( );
    reg [3:0]w;
    reg [1:0]sel;
    wire [3:0]y;
    barrel_shifter b1( w,sel, y);
    initial begin
    w=4'b1011;sel=2'b00;
    #10 sel=2'b01;
    #10 sel=2'b10;
    #10 sel=2'b11;
    end
    initial begin
    $monitor(" w=%b,sel=%b,y=%b",w,sel,y);
    end
    initial begin
    #50 $stop;
    end
endmodule
```

D. CONSOLE WINDOW

```
# run 1000ns
 w=1011,sel=00,y=1011
 w=1011,sel=01,y=1101
 w=1011,sel=10,y=1110
 w=1011,sel=11,y=0111
$stop called at time : 50 ns
```



RTL SCHEMATIC OF BARREL SHIFTER



DAY 14 DESIGNING OF ALU WITH 16 OPERATIONS

VERILOG CODE

```
module ALU(
    input [7:0] a,b,
    input [3:0] sel,
    output reg  [7:0] result,
    output reg cout
    );
    always@(sel,a,b)
begin
case(sel)
0: {cout,result}=a+b;
1:  result =    a-b;
2:  result =    a*b;
3:  result =    a/b;
4:  result =    a<<1 ;
5:  result =    a>>1;
6:  result =    {a[6:0],a[7]};
7:  result =    {a[0],a[7:1]};
8:  result =    a&b ;
9:  result =    a^b ;
10: result =   ~(a^b)    ;
11: result =    a&b     ;
12: result =   ~(a&b)     ;
13: result =    a|b    ;
14: result =   ~(a|b)   ;
15: result =   (a==b)?1:0;
default: result=a+b;
endcase
end
endmodule
```

TEST BENCH:

```
`timescale 1ns / 1ps
module tb_alu( );
 reg[7:0] a,b;
 reg[3:0] sel;
 wire[7:0]result;
 integer i;
 ALU test_unit(a,b,sel,result);
    initial begin
      a = 8'h0A;
      b = 4'h02;
      sel = 4'h0;
      for (i=0;i<=15;i=i+1)
      begin
       sel = sel + 8'h01;
       #10;
      end end
      initial
      begin
       $monitor("a= %b,b= %b,sel=%d,result=%b",a,b,sel,result);
      end
      initial begin
      #200 $stop;
      end
endmodule
```

CONSOLE RESULT:

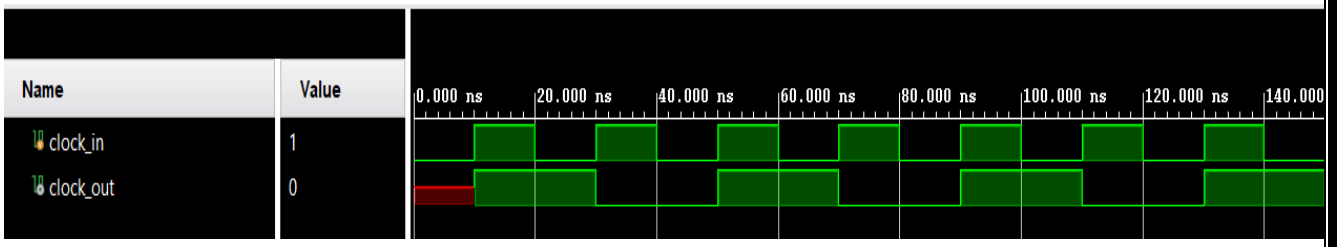
```
a= 00001010,b= 00000010,sel= 1,result=00001000
a= 00001010,b= 00000010,sel= 2,result=00010100
a= 00001010,b= 00000010,sel= 3,result=00000101
a= 00001010,b= 00000010,sel= 4,result=00010100
a= 00001010,b= 00000010,sel= 5,result=00000101
a= 00001010,b= 00000010,sel= 6,result=00010100
a= 00001010,b= 00000010,sel= 7,result=00000101
a= 00001010,b= 00000010,sel= 8,result=00000010
a= 00001010,b= 00000010,sel= 9,result=00001000
a= 00001010,b= 00000010,sel=10,result=11110111
a= 00001010,b= 00000010,sel=11,result=00000010
a= 00001010,b= 00000010,sel=12,result=11111101
a= 00001010,b= 00000010,sel=13,result=00001010
a= 00001010,b= 00000010,sel=14,result=11110101
a= 00001010,b= 00000010,sel=15,result=00000000
a= 00001010,b= 00000010,sel= 0,result=00001100
```

DAY 15
CLOCK DIVIDER USING A COUNTER
VERILOG CODE WITH TEST BENCH

```
module clock__divider(
input clock_in,
output reg clock_out );

reg[27:0] counter=28'd0;
parameter DIVISOR = 28'd2;
always @(posedge clock_in)
begin
 counter <= counter + 28'd1;

 if(counter>=(DIVISOR-1))
  counter <= 28'd0;
 clock_out <= (counter<DIVISOR/2)?1'b1:1'b0;
end
endmodule
```

```
`timescale 1ns / 1ps

module tb__clock_divider();
reg clock_in;
wire  clock_out;
clock__divider c1( clock_in,  clock_out );
initial begin
    clock_in=0;
    end
    always
    #10 clock_in=~clock_in;
    initial
    #200 $stop;
endmodule
```

OUTPUT SIMULATION WAVEFORM



NOTE:
TO DIVIDER CLOCK BY SOME OTHER DIVISOR . JUST CHANGE PARAMETER VALUE.

DAY 16

CREATE A CIRCUIT THAT DIVIDES CLOCK FREQUNECY BY ODD NUMBER (HERE I AM DOING IT WITH 3)

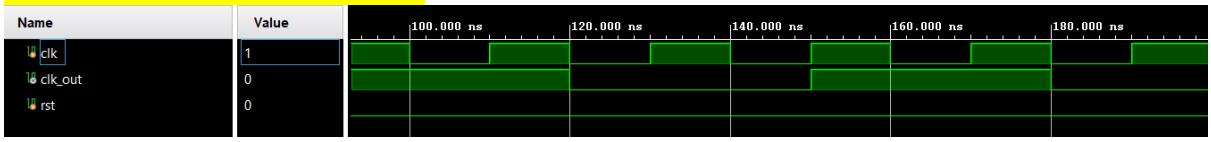
VERILOG CODE :

```
`timescale 1ns / 1ps
//DAY 16 ODD FREQUNECY DIVIDER//
module odd_freq_divider(
    input clk,rst,
    output clk_out);

    reg q0,q1,q2;
    wire d0;
    always@(posedge rst or posedge clk)

    if(rst) begin
    q0<=1'b0;
    q1<=1'b0;
    end

    else begin
    q0<=d0;
    q1<=q0; end

    always @(posedge rst or negedge clk)
    if(rst)
    q2<=1'b0;
    else
    q2<=q1;

    assign d0=~q0&~q1;
    assign clk_out=q1|q2;

endmodule
```

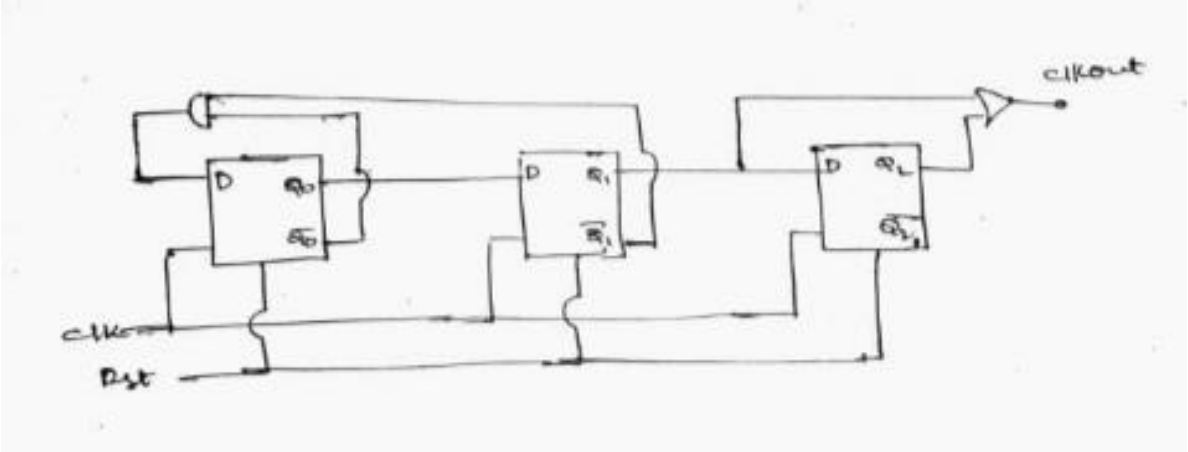
TESTBENCH:

```
`timescale 1ns / 1ps

module tb_odd_divider( );
    wire clk_out;
    reg clk,rst;
    odd_freq_divider o1( clk,rst,clk_out);

    always
    #10 clk=~clk;

    initial begin
    clk=1'b0;rst=1'b1;
    #5 rst=1'b0;
    end
    initial
    #200 $stop;

endmodule
```

OUTPUT SIMULATION WAVEFORM:



CIRCUIT:



DAY 16 -B

Q. CREATE A CLOCK WITH FREQUNCY 50 MHz , 16 MHz, 8 MHz.

To do this we should be clear about timescale derivative.

a. **50 MHz means it has a time period of 20 ns (i.e. Half time period 10ns)**
b. **16 MHz means it has a time period of 62.5 ns (i.e. Half time period 31.25ns)**
c. **8 MHz means it has a time period of 125 ns (i.e. Half time period 62.5ns)**
d. **So we need a precision of 2 bits**
   So our time scale derivative should be

   `timescale 1ns/1ps

   Which means 10^-9/10^-12 =10^3 i.e. we have a precision of 3 bits.

   i.e. Till a precision of 3 bits after decimal we can specify our value.

VERILOG CODE :

```
`timescale 1ns / 1ps
//DAY 16 PART 2//

module timescale_example();
    reg clk50=0;
    reg clk16=0;
    reg clk8=0;
    always #10 clk50=~clk50;
    always #31.25 clk16=~clk16;
    always #62.5 clk8=~clk8;
    initial begin
    #200 $stop;
    end
endmodule
```

OUTPUT SIMULATION WAVEFROM

DAY 17

A. IMPLEMENTATION OF SR,D,T,JK FLIP FLOPS

1. SR (SET-RESET FLIP FLOP)

VERILOG CODE WITH TEST BENCH:

```
module srff(output reg q,qbar,input s,r,clk,clr);
always @(posedge clk)
if(clr==1'b1)
begin
q<=1'b0;qbar<=1'b1;end
else
case({s,r})
2'b00:begin q<=q;qbar<=qbar;end
2'b01:begin q<=1'b0;qbar<=1'b1;end
2'b10:begin q<=1'b1;qbar<=1'b0;end
2'b11:begin q<=1'bx;qbar<=1'bx;end
default :{q,qbar}<=2'bxx;
endcase
endmodule

module tb_sr();
wire q,qbar;
reg s,r,clk,clr;
srff s1(q,qbar,s,r,clk,clr);
initial begin
s=0;
r=0;
clk=1'b1;end
always #2 {s,r}={s,r}+1'b1;
always #1 clk=~clk;
initial #10 clr=1'b0;
initial #50 $stop;
endmodule
```

2. D (DATA FLIP FLOP)

VERILOG CODE WITH TEST BENCH:

```
2   module d_ff(clk,clear,d,q,qbar);
3     input clk,clear,d;
4     output  reg q,qbar;
5     always @ (posedge clk or negedge clear)
6   if(clear==1'b0)
7   begin q<=1'b0;qbar<=1'b1;end
8   else
9   begin q<=d;qbar<=~d;end
10  endmodule
11
12
13  module dff_tb();
14  reg d;
15  reg clk;
16  reg clear;
17  wire q;
18  wire qbar;
19  d_ff d1(.q(q),.qbar(qbar),.d(d),.clk(clk),.clear(clear));
20  initial begin
21  d=1'b0;
22  clk=1'b0;
23  clear=1'b1;
24  #10 clear=1'b0;
25  #7 clear=1'b1;
26
27  end
28  always #2 d=d+1'b1;
29
30
31  always #5 clk=~clk;
32  initial #100
33  $stop;
34   endmodule
```

3. T FLIP FLOP

VERILOG CODE WITH TEST BENCH:

```
module tff(output reg q,qbar,input t,clk,clr);
always @(posedge clk)
if(clr==1'b1)
begin
q<=1'b0; qbar<=1'b1;end
else if (t==1'b0)
begin
q<=q;qbar<=qbar;end
else
begin
q<=qbar;qbar<=q;
end
endmodule


module tff_tb();
reg t,clk,clr;
wire q,qbar;

tff t1(q,qbar,t,clk,clr);
initial begin
t=0;
clk=0;
clr=1;
end

always #1 clk=~clk;
always #2 t=~t;
initial #10 clr=1'b0;
initial #30 $stop;
endmodule
```

4. J K (JACK KILBY) FLIP FLOP

VERLIOG CODE WITH TEST BENCH:

```
module jk_ff(output reg q,qbar,input j,k,clk,clr);
always@(posedge clk)
if (clr==1'b1)
begin
q<=1'b0;qbar<=1'b1;end
else
case({j,k})
2'b00: begin q<=q;qbar<=qbar;end

2'b01: begin q<=1'b0;qbar<=1'b1;end

2'b10: begin q<=1'b1;qbar<=1'b0;end

2'b11: begin q<=~q;qbar<=~qbar;end

default:{q,qbar}<=2'bxx;
endcase
endmodule

module tb_jk();
wire q,qbar;
reg j,k,clk,clr;
jk_ff j1(q,qbar,j,k,clk,clr);
initial begin
j=0;k=0;clk=0;clr=1;
end
always #1 clk=~clk;
always #2 {j,k}={j,k}+1'b1;
initial #10 clr=1'b0;
initial #100 $stop;
endmodule
```

DAY 18
CONVERSION OF FLIP FLOPS

1. D to JK flip flop

Verilog code for D_ ff

```
module dff(input d,clk,clr
,output reg q,qbar);
    always @(posedge clk)
    begin
    if(clr)
    begin
    q<=1'b0;qbar<=1'b1;
    end
    else
    begin
    q<=d;qbar<=~d;
    end
    end
endmodule
```

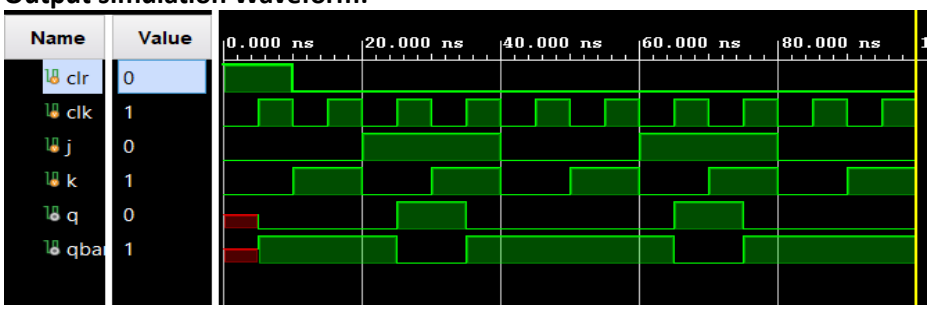
Conversion of D to JK

```
module D_to_JK(
input j,k,clk,clr
,output q,qbar );
wire w;
assign w=((j&qbar)|(~k&q));
 dff d1(w ,clk,clr, q,qbar);
endmodule
```

Test Bench

```
`timescale 1ns / 1ps
module tb_djk();
reg j,k,clk,clr;
wire q,qbar;
D_to_JK uut( j,k,clk,clr,q,qbar );
initial begin
clr=1'b1;{j,k}=2'b0;clk=1'b0;
#10 clr=1'b0;
end
always #10 {j,k}={j,k}+1'b1;
always #5 clk=~clk;
initial #100 $stop;
endmodule
```

Output simulation Waveform:



2. CONVERSION OF SR TO TFLIP FLOP

VERILOG CODE FOR SR FLIP FLOP

```
`timescale 1ns / 1ps
module SR_FF(input s,r,clk,clr
,output reg q,qbar );
always @(posedge clk)
begin
if(clr)
begin q<=1'b0;qbar<=1'b1;end
else
begin
case({s,r})
2'b00:begin q<=q;qbar<=qbar;end
2'b01:begin q<=1'b0;qbar<=1'b1;end
2'b10:begin q<=1'b1;qbar<=1'b0;end
2'b11:begin q<=1'bx;qbar<=1'bx;end
default:{q,qbar}<=2'bxx;
endcase
end
end
endmodule
```

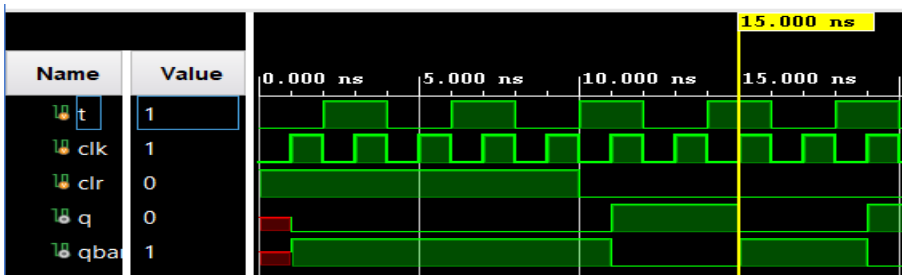
VERILOG CODE FOR CONVERSION OF SR TO T FLIP FLOP:

```
`timescale 1ns / 1ps
module SR_to_T(
input t,clk,clr,output q,qbar);
wire w;
assign w=t&qbar;
assign w1=t&q;
SR_FF b(w,w1,clk,clr, q,qbar );
endmodule
```

TEST BENCH:

```
`timescale 1ns / 1ps
module tb_srt();
    reg t,clk,clr;
    wire q,qbar;
SR_to_T s(t,clk,clr, q,qbar);
initial begin
t=0;clk=0;clr=1'b1;
#10 clr=1'b0;
end
always #1 clk=~clk;
always #2 t=~t;
initial #100 $stop;
endmodule
```

OUTPUT SIMULATION WAVEFORM



CONVERSION OF ALL FLIP FLOPS
YOU CAN CONVERT FLIP SLOPS FROM ONE TO OTHER BY USING THESE AS INPUTS.
THESE CAN BE DERIVEDD FROM WRITINF TRUTH ABLE AND EXCITATION TABLES.

(1) D to JK ⟶ $D = J\bar{Q}(t) + \bar{K}Q(t)$

(2) JK to D ⟶ $J = D$, $K = \bar{D}$

(3) T to D ⟶ $T = D\bar{Q} + \bar{D}Q$

(4) D to T ⟶ $D = T\bar{Q} + Q\bar{T}$

(5) SR to JK ⟶ $S = J\bar{Q}$, $R = KQ$

(6) JK to SR ⟶ $J = S$, $K = R$

(7) SR to D ⟶ $S = D$, $R = \bar{D}$

(8) D to SR ⟶ $D = S + \bar{R}Q$

(9) T to SR ⟶ $T = RQ + S\bar{Q}$

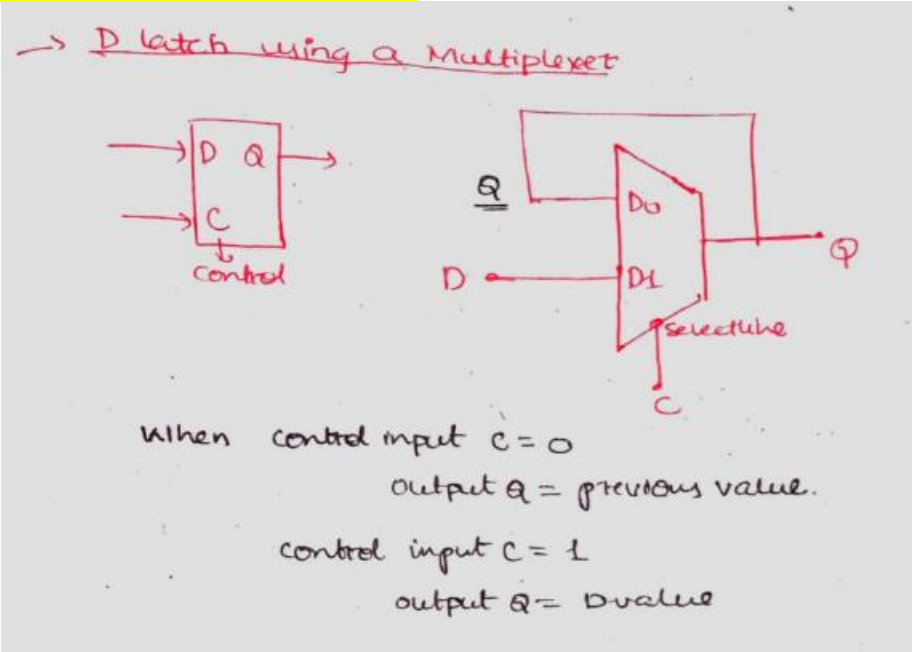
(10) SR to T ⟶ $S = T\bar{Q}$, $R = QT$

(11) JK to T ⟶ $J = T$, $K = T$

(12) T to JK ⟶ $T = KQ + J\bar{Q}$

DAY-19

A. D_LATCH FROM 2X1 MULTIPLEXER



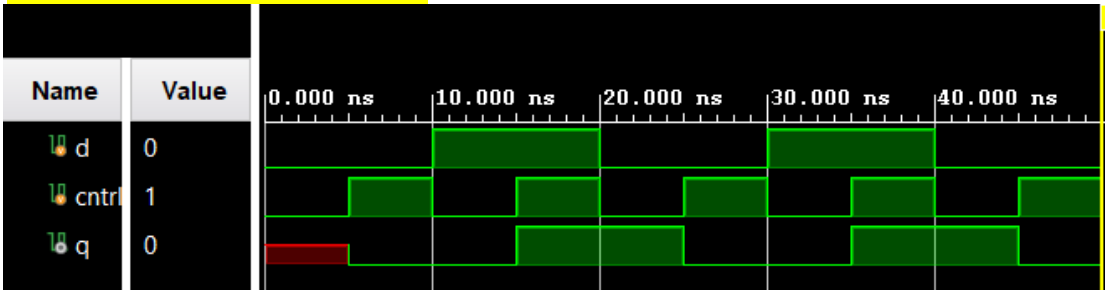
VERILOG CODE

```
//DAY 19//
module mux2_1(input a,b,s,output reg y);
always@(*)
begin
y=(~s&a)|(s&b);end
endmodule

module D_LATCH_MUX(
d,cntrl,q);
input d,cntrl;
inout q;
mux2_1 m1(q,d,cntrl,q);
endmodule
```

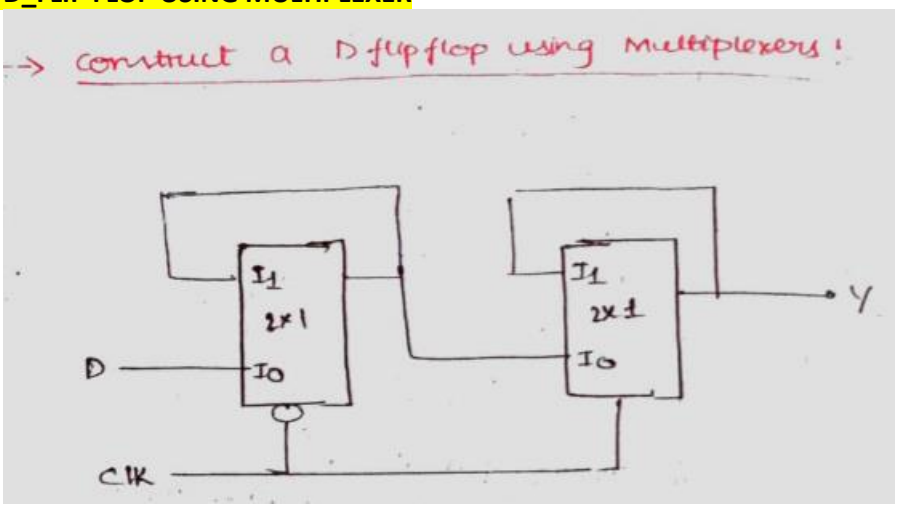
TEST BENCH

```
`timescale 1ns / 1ps

module tb_dlatch( );
reg d,cntrl;
wire q;
D_LATCH_MUX d1(d,cntrl,q);
initial begin
cntrl=0;d=0;
end
always #5 cntrl=~cntrl;
always #10 d=~d;
initial #50 $stop;
endmodule
```

OUTPUT SIMULATION WAVEFORM



b. D_FLIP FLOP USING MULTIPLEXER



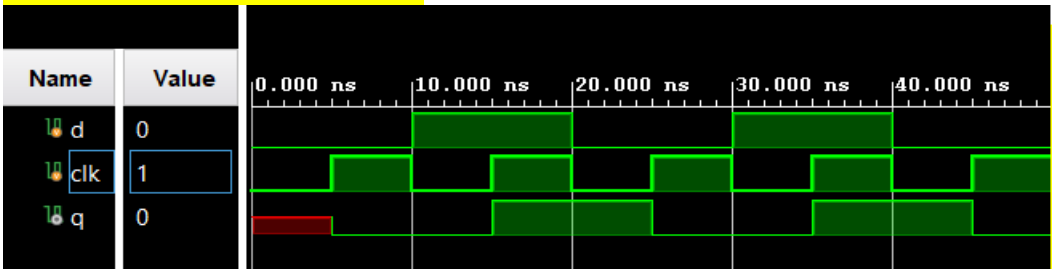
VERILOG CODE

```
`timescale 1ns / 1ps

//DAY 19//
module mux2_1(input a,b,s,output reg y);
always@(*)
begin
y=(~s&a)|(s&b);end
endmodule

module D_FF_MUX(
d,clk,q);
input d,clk;
inout q;
wire w;
mux2_1 m1(w,d,~clk,w);
mux2_1 m2(q,w,clk,q);
endmodule
```

TEST BENCH

```
`timescale 1ns/1ps
module tb_dff( );
reg d,clk;
wire q;
D_FF_MUX d1(d,clk,q);
initial begin
clk=0;d=0;
end
always #5 clk=~clk;
always #10 d=~d;
initial #50 $stop;
endmodule
```

OUTPUT SIMULATION WAVEFORM



DAY-20

Using DFFs and minimum no. of 2×1 Mux, implement the following XYZ flip-flop.

X	Y	Z	Q(t+1)
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	Q(t)
1	0	1	Q(t)`
1	1	0	Q(t)`
1	1	1	Q(t)

From the above truth table

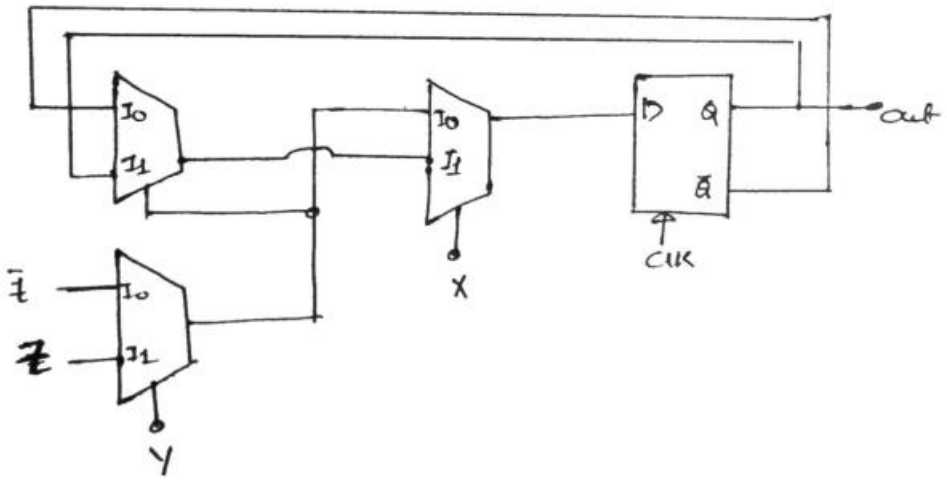
a) If $x=0$, $Q(t+1) = y \oplus t$ xnor → ⊕

b) If $x=1$, $y=t$

Then $Q(t+1) = Q(t)$

b) If $x=1$, $y \neq t$

Then $Q(t+1) = Q(t)'$



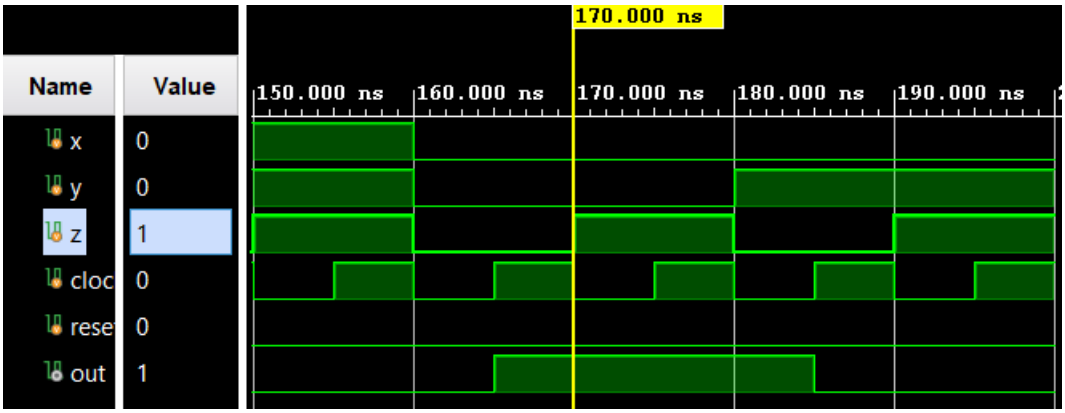
VERILOG CODE WITH TEST BENCH

```
`timescale 1ns / 1ps
//2X1 MUX//
module mux21(input i0,i1,s,output reg o);
always @(*)
begin
o=(~s&i0)|(s&i1);
end
endmodule
//DFF//
module dff(input d,clk,rst,output reg q);
always@(posedge clk)
begin
if(rst) begin
q<=1'b0;
end
else begin
q<=d;
end
end
endmodule
//TOP MODULE//
module top_module(
input x,y,z,clock,reset,output reg out);
    wire w1,w2,w3,w4,q,q1;
    mux21 m1(~z,z,y,w1);
    mux21 m2(~q,q,w1,w2);
    mux21 m3(w1,w2,x,w3);
    dff d1(w3,clock,reset,q);
     always @(posedge clock )
     begin
       out <= q;
    end
endmodule
```

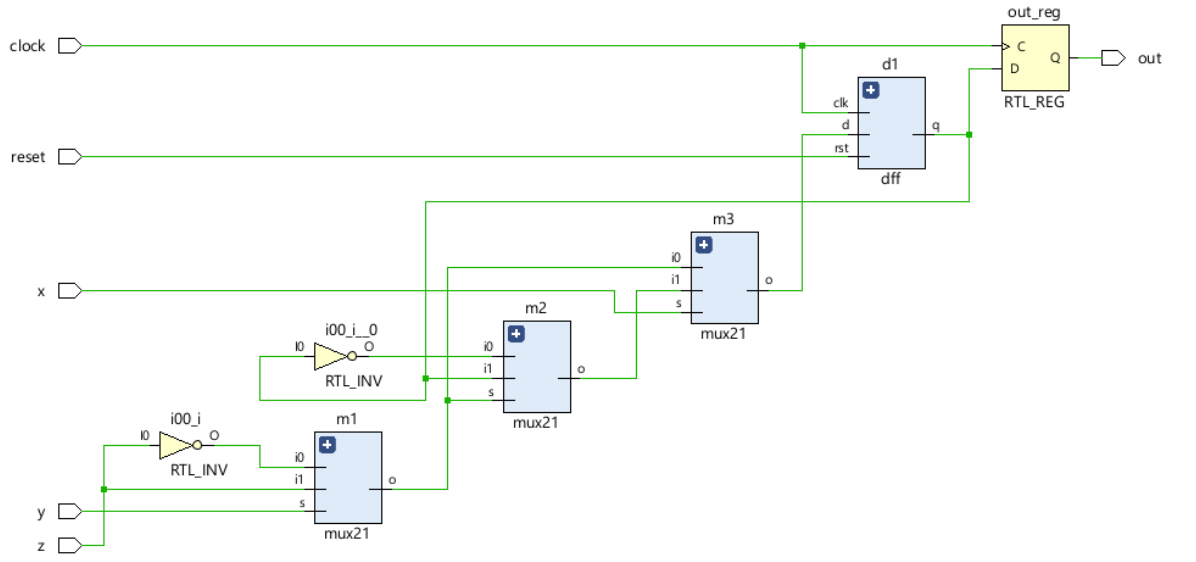
TESTBENCH

```
`timescale 1ns / 1ps

module tb_q();
reg x,y,z,clock,reset;
wire out;
top_module tt(x,y,z,clock,reset,out);
initial begin
reset=1'b1;{x,y,z}=3'b0;clock=1'b0;
#10 reset=1'b0;
end
always #10 {x,y,z}={x,y,z}+1'b1;
always #5 clock=~clock;
initial #200 $stop;

endmodule
```

OUTPUT SIMULATION WAVEFORM



RTL SCHEMATIC



DAY 21

IMPLEMENTATION OF SISO ,SIPO,PISO,PIPO IN BEHAVIOUR MODELLING

1. SERIAL IN SERIAL OUT SHIFT REGISTER

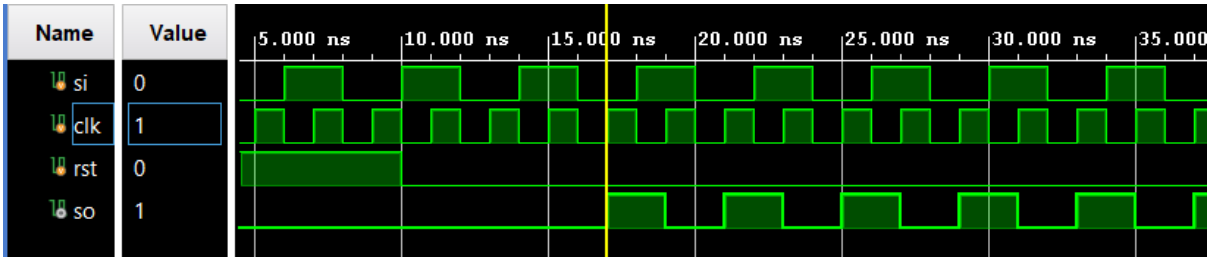
VERILOG CODE

```
module siso(si,so,clk,rst);
input si,clk,rst;
output so;
reg[3:0]q;
always@(posedge clk )
begin
if(rst)
q<=4'b0;
else begin
q<=q>>1;
q[3]<=si;
end
end
assign so=q[0];
endmodule
```

TEST BENCH

```
`timescale 1ns / 1ps

module tb_siso();
reg si,clk,rst;
wire so;
siso s1(si,so,clk,rst);
initial begin
clk=0;rst=1;si=0;
#10 rst=0;
end
always #1 clk=~clk;
always #2 si=~si;
initial #100 $stop;
endmodule
```

OUTPUT SIMULATION WAVEFORM



2. SERIAL IN PARALLEL OUT SHIFT REGISTER

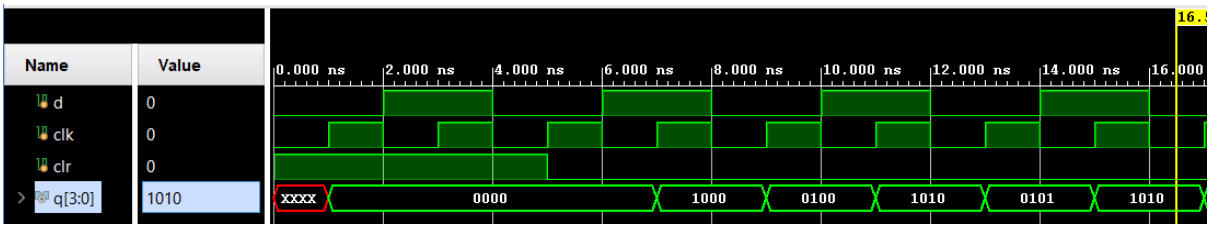
VERILOG CODE

```
`timescale 1ns / 1ps

module sipo(input d,clk,clr,
output reg [3:0]q );
    reg [3:0]temp;
    always @(posedge clk)
    if(clr==1'b1)
    q<=4'b0000;
    else begin
    temp=q>>1;
    q={d,temp[2:0]};
    end
endmodule
```

TEST BENCH

```
`timescale 1ns / 1ps

module tb_sipo();
    reg d,clk,clr;
    wire [3:0] q;
    sipo s1(d,clk,clr,q);
    initial begin
    clr=1;clk=0;d=0;
    #5 clr=1'b0;
    end
    always #1 clk=~clk;
    always #2 d=~d;
    initial #50 $stop;

endmodule
```

OUTPUT SIMULATION WAVEFORM



3. PARALLEL IN PARALLEL OUT SHIFT REGISTER

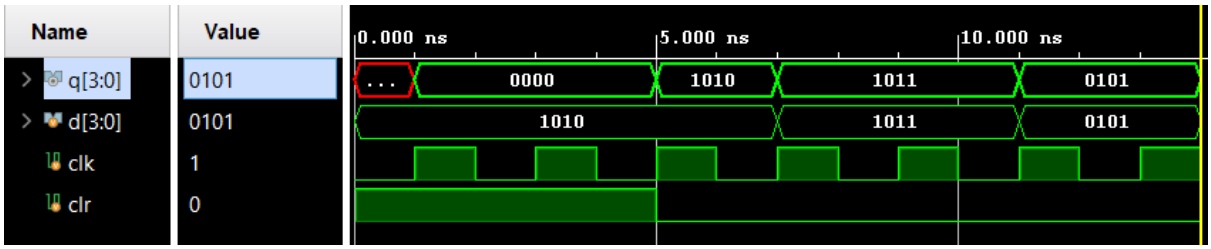
VERILOG CODE

```
module PIPO(q,clk,clr,d );
output reg [3:0]q;
input clk,clr;
input [3:0]d;
always@(posedge clk)
if (clr==1'b1)
q=4'b0000;
else
q=d;
endmodule
```

TEST BENCH

```
`timescale 1ns / 1ps

module tb_pipo();
wire [3:0]q;
reg [3:0]d ;
reg clk,clr;
PIPO p1(q,clk,clr,d );
initial begin
clr=1'b1;clk=1'b0;d=4'b1010;
#5 clr=1'b0;
#2 d=4'b1011;
#2 d=4'b1011;
#2 d=4'b0101;
end
always #1 clk=~clk;
initial #14 $stop;
endmodule
```

OUTPUT SIMULATION WAVEFORM



4. PARALLEL IN SERIAL OUT SHIFT REGISTER

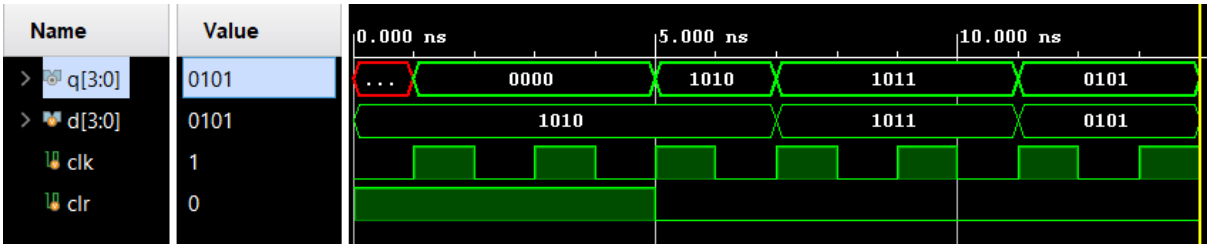
VERILOG CODE

```
module piso(v,clk,sel,clr,d);
input [3:0]d;
input clk,sel,clr;
output reg v;
reg [3:0]q;
always @(posedge clk)
begin
if(clr==1'b1)
q<=4'b0000;
else if(sel==0)
q<=d;
else
begin
v<=q[0];
q<=q>>1;
end
end
endmodule
```

TESTBENCH

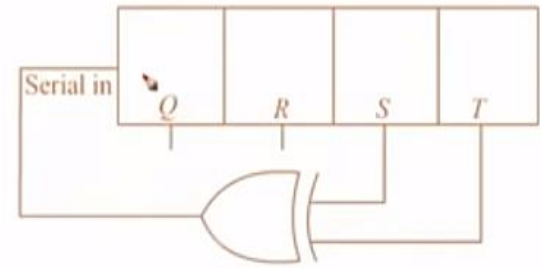
```
`timescale 1ns / 1ps
module tb_piSo();
wire v;
reg sel,clk,clr;
reg [3:0]d;
piso p1(v,clk,sel,clr,d);
initial begin
clk=0;
clr=1;
#2 clr=0;
sel=0;
d=4'b0100;
#2 sel=1;
#10 sel=0;
d=4'b1110;
#2 sel=1;
end
always #1 clk=~clk;
initial #25 $stop;
endmodule
```

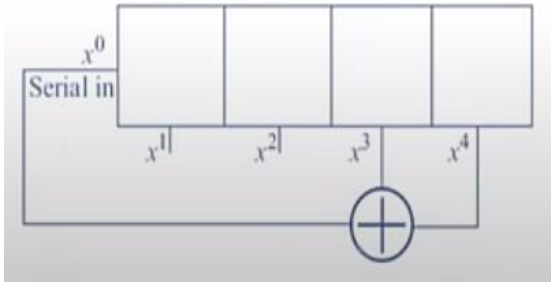
OUTPUT SIMULATION WAVEFORM



DAY 22

LINEAR FEEDBACK SHIFT REGISTER





TRUTH TABLE FOR 4 BIT RANDOM SEQUNECE GENERATOR:

Q	R	S	T	Serial in $= S \oplus T$	Clock cycle
1	1	1	1	0	1
0	1	1	1	0	2
0	0	1	1	0	3
0	0	0	1	1	4
1	0	0	0	0	5
0	1	0	0	0	6
0	0	1	0	1	7
1	0	0	1	1	8
1	1	0	0	0	9
0	1	1	0	1	10
1	0	1	1	0	11
0	1	0	1	1	12
1	0	1	0	1	13
1	1	0	1	1	14
1	1	1	0	1	15
1	1	1	1	0	16 (repeats)

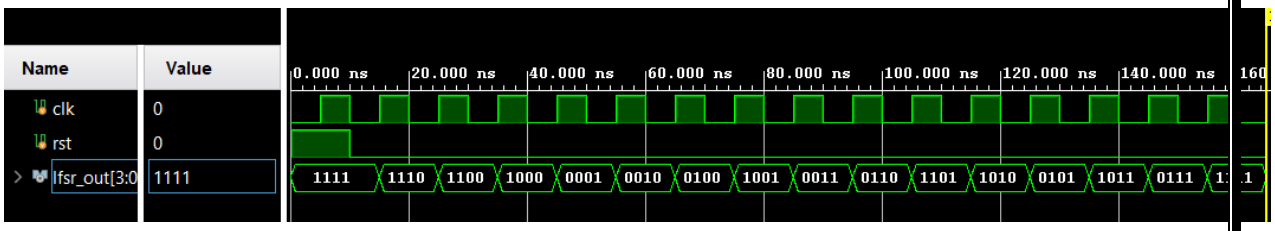
VERILOG CODE

```
module random_lfsr(clk,rst,lfsr_out);
input clk,rst;
output reg [3:0] lfsr_out;
always@(posedge clk or posedge rst)
begin
if(rst)
lfsr_out<=4'hf;
else
lfsr_out<={lfsr_out[2:0],(lfsr_out[3]^lfsr_out[2])};
end
endmodule
```

TESTBENCH:

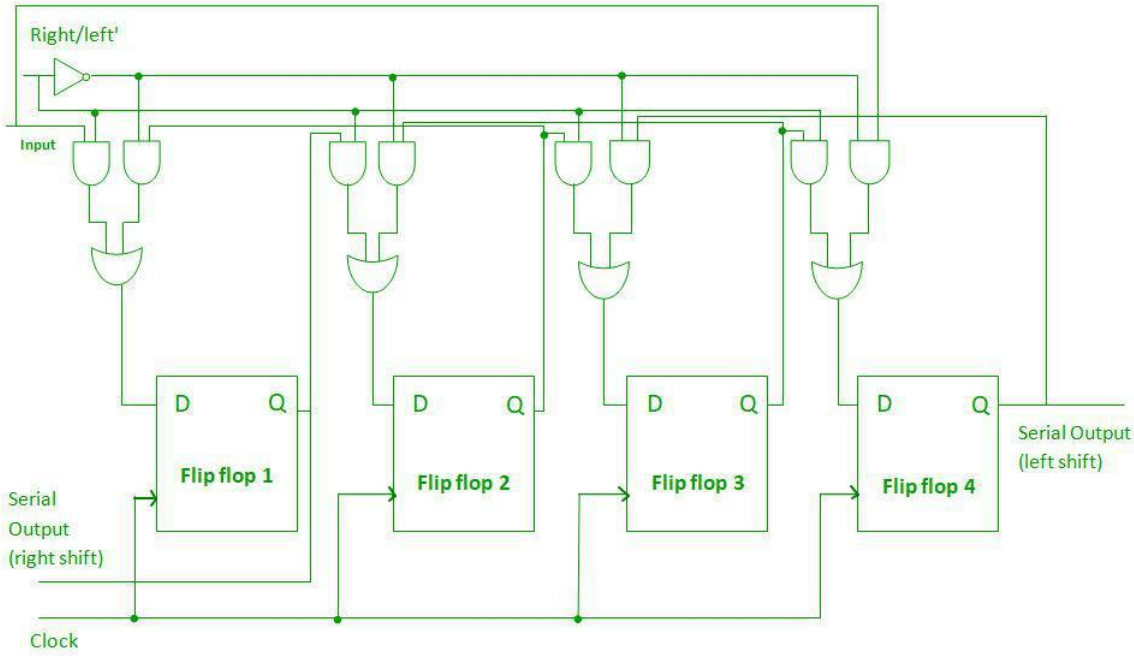
```
`timescale 1ns / 1ps
module tb_lfsr();
reg  clk,rst;
wire [3:0] lfsr_out;
random_lfsr r1(clk,rst,lfsr_out);
initial begin
clk=0;rst=1;
#10 rst=0;
end
always #5 clk=~clk;
initial #165 $stop;
endmodule
```

OUTPUT SIMULATION WAVEFORM:



DAY 23

BI DIRECTIONAL SHIFT REGISTER



This shift register shifts the contents of the flip flop to

a. **Right if mode=1**
b. **Left if mode=0**

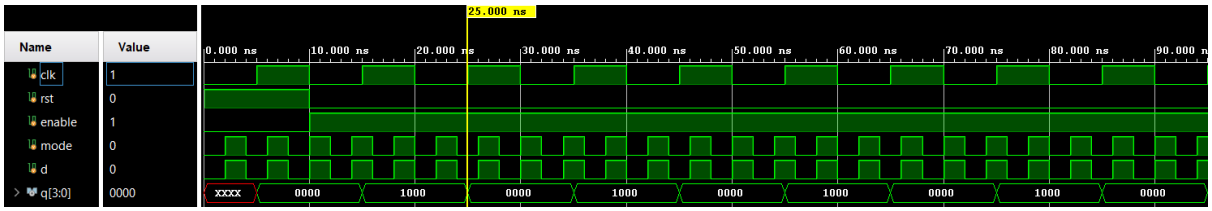
VERILOG CODE:

```
module Bi_Direct_SR(
input clk,rst,enable,mode,d,
output reg[3:0]q );
    always @(posedge clk)
    if(rst)
    q<=0;
    else
    begin
    if(enable)
    case(mode)
    0:q<={q[2:0],d};
    1:q<={d,q[3:1]};
    endcase
    else
    q<=q;
    end
endmodule
```

TEST BENCH:

```
`timescale 1ns / 1ps

module tb_bi_directional_sr();
reg clk,rst,enable,mode;
reg d;
wire[3:0]q;
Bi_Direct_SR b1(clk,rst,enable,mode,d,q);
initial begin
clk=0;rst=1;enable=0;mode=0;d=0;
#10 rst=0;enable=1;
end
always #2 d=~d;
always #5 clk=~clk;
always #2 mode=~mode;
initial #100 $stop;
endmodule
```

OUTPUT SIMULATION WAVEFORM:



DAY 24
UNIVERSAL SHIFT REGISTER

A Universal shift register is a register which has both the right shift and left shift with parallel load capabilities. Universal shift registers are used as memory elements in computers. A Unidirectional shift register is capable of shifting in only one direction. A bidirectional shift register is capable of shifting in both the directions. The Universal shift register is a combination design of bidirectional shift register and a unidirectional shift register with parallel load provision.

WHEN MODE IS

S1	S0	OPERATION
0	0	NO CHANGE
0	1	RIGHT SHIFT
1	0	LEFT SHIFT
1	1	LOAD

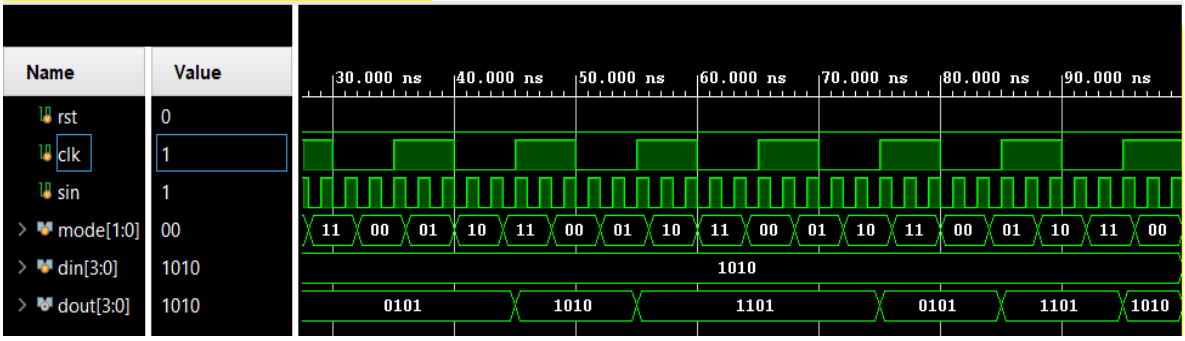
VERILOG CODE

```
//DAY 24// UNIVERSAL SHIFT REGISTER//
module universal_SR(
clk,din,rst,mode,sin,dout);
    input clk,rst,sin;
    input [1:0]mode;
    input [3:0]din;
    output reg [3:0]dout;
    always @(posedge clk,posedge rst)
    begin
    if(rst)
    dout<=4'b0000;
    else begin
    case(mode)
    2'b00:dout<=dout;//no change
    2'b01:dout<={sin,din[3:1]};//right shift
    2'b10:dout<={din[2:0],sin};//left shift
    2'b11:dout<=din;//load data
    endcase
    end
    end
endmodule
```

TEST BENCH:

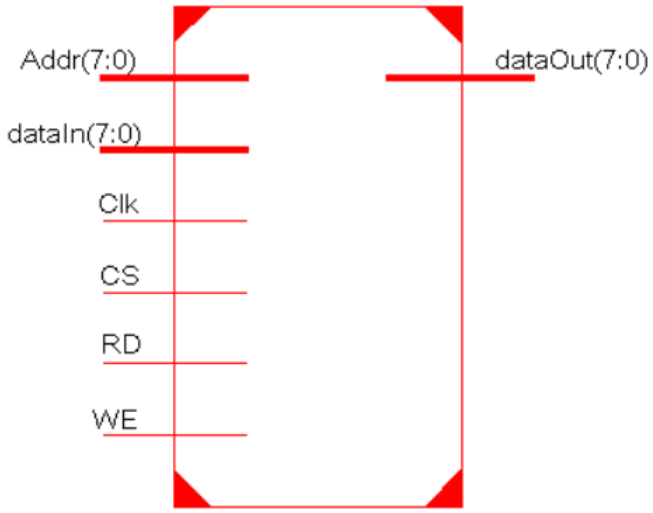
```
`timescale 1ns / 1ps
module tb_universal_sr();
    reg clk,rst,sin;
    reg [1:0]mode;
    reg [3:0]din;
    wire [3:0]dout;
universal_SR u1(
clk,din,rst,mode,sin,dout);
initial begin
rst=1'b1;clk=1'b0;din=4'b1010;sin=1'b0;mode=2'b00;
#10 rst=1'b0;
end
always #5 clk=~clk;
always #4 mode=mode+1;
always #1 sin=~sin;
initial #100 $stop;
endmodule
```

OUTPUT SIMULATION WAVEFORM:



DAY 25

SINGLE PORT RAM



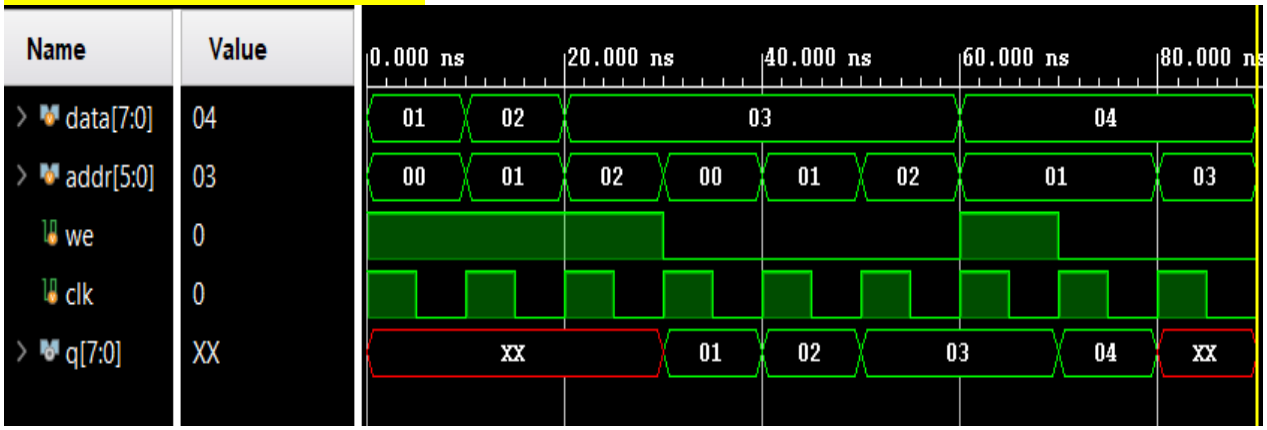
VERILOG CODE:

```
`timescale 1ns / 1ps

module single_port_ram(
  input [7:0] data, //input data
  input [5:0] addr, //address
  input we, //write enable
  input clk, //clk
  output [7:0] q //output data
);

  reg [7:0] ram [63:0]; //8*64 bit ram
  reg [5:0] addr_reg; //address register

  always @ (posedge clk)
    begin
      if(we)
        ram[addr] <= data;
      else
        addr_reg <= addr;
    end

  assign q = ram[addr_reg];

endmodule
```

TEST BENCH

```
`timescale 1ns / 1ps
module tb_single_port_ram;
  reg [7:0] data; //input data
  reg [5:0] addr; //address
  reg we; //write enable
  reg clk; //clk
  wire [7:0] q; //output data
  single_port_ram spr1(
    .data(data),.addr(addr),.we(we),.clk(clk),.q(q));
  initial
    begin
      clk=1'b1;
      forever #5 clk = ~clk;
    end
  initial
    begin
      data = 8'h01;
      addr = 5'd0;
      we = 1'b1;
      #10;
      data = 8'h02;
      addr = 5'd1;
      #10;
      data = 8'h03;
      addr = 5'd2;
#10;  addr = 5'd0;
      we = 1'b0;
      #10;addr = 5'd1;
      #10;addr = 5'd2;
      #10; data = 8'h04;
      addr = 5'd1;
      we = 1'b1;
      #10;addr = 5'd1;
      we = 1'b0;
      #10; addr = 5'd3;
      #10;end
  initial
    #90 $stop;
endmodule
```

OUTPUT SIMULATION WAVEFORM



NORMAL PACTICE:

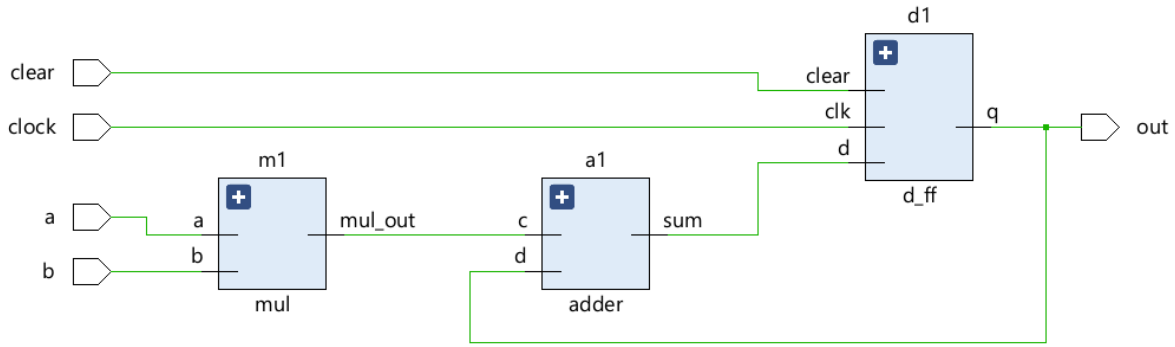
```
module mul(
input a,b,
output mul_out);
assign mul_out=a*b;
endmodule
```

```
`timescale 1ns / 1ps

module adder(
input c,d,output sum,carry
    );
    assign sum=c^d;
    assign carry=c&d;
endmodule
```

```
`timescale 1ns / 1ps

module d_ff(
input d,clk,clear,output reg q
    );
    always @(posedge clk )
    if(clear==1) begin
    q<=1'b0;
    end
    else
    begin
    q<=d; end
endmodule
```

```
module top(input a,b,clock,clear,output out);
wire mul_out,sum,carry,q;
mul m1 (a,b,mul_out);
adder a1(mul_out,q,sum,carry);
d_ff d1(sum,clock,clear,q);
assign out=q;
endmodule
```

```
`timescale 1ns / 1ps

module tb_top( );
reg a,b,clock,clear;
wire out;
top t1(a,b,clock,clear,out);
initial begin
a=1'b1;b=1'b1;
clock=0;clear=1;
#10 clear=0;
end
always #5 clock=~clock;
endmodule
```



UP_DOWN COUNTER

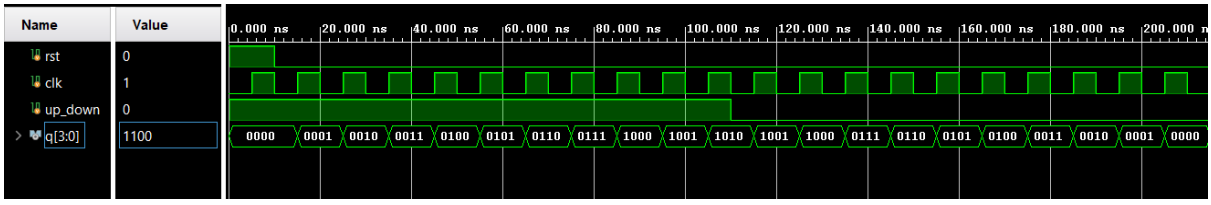
VERLOG CODE

```
`timescale 1ns / 1ps
module Up_Counter(q,up_down,clk,rst);
output reg [3:0]q;
input up_down,clk,rst;
always @(posedge clk or posedge rst)
begin
if(rst)
q<=4'b0;
else if(up_down)
q<=q+1;
else
q<=q-1;
end
endmodule
```

TEST BENCH

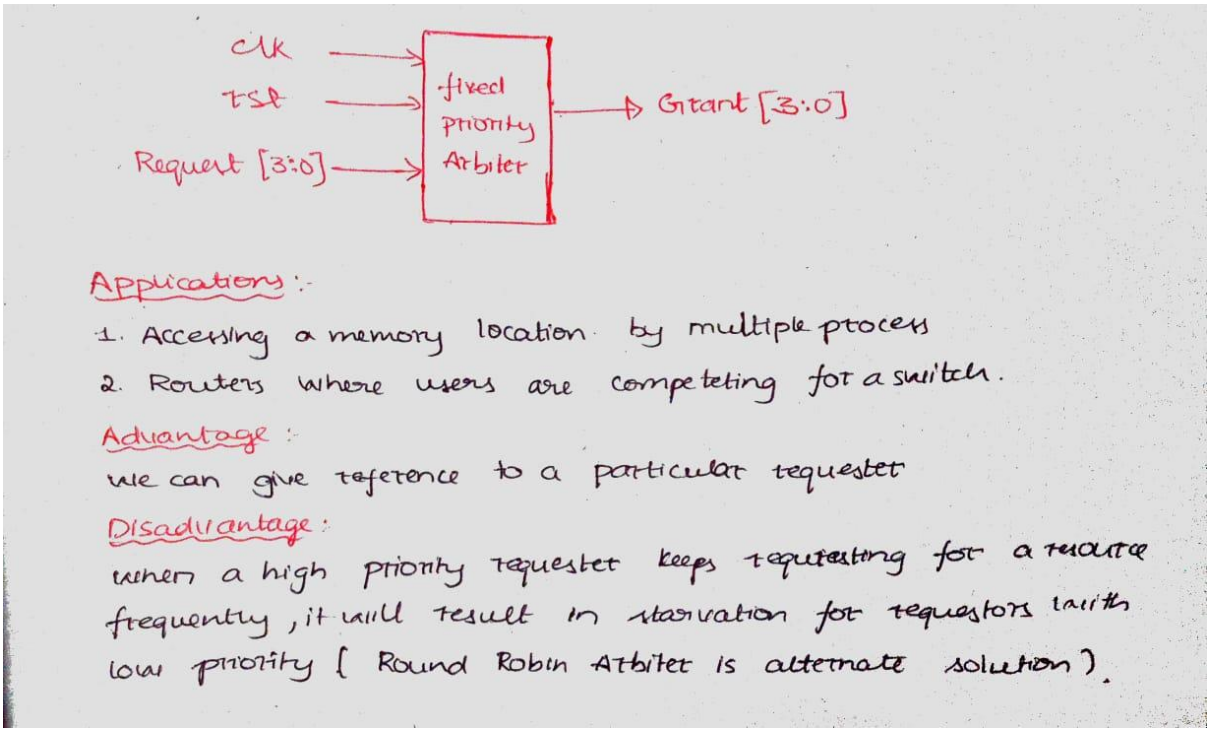
```
`timescale 1ns / 1ps

module tb_updown();
wire [3:0]q;
reg up_down,clk,rst;
Up_Counter u1(q,up_down,clk,rst);
always
#5 clk=~clk;
initial begin
clk=0;up_down=1;rst=1;
#10 rst=0;
#100 up_down=0;
#200 $stop;
end
endmodule
```

OUTPUT SIMULATION WAVEFORM



DAY-26

FIXED POINT ARBITER



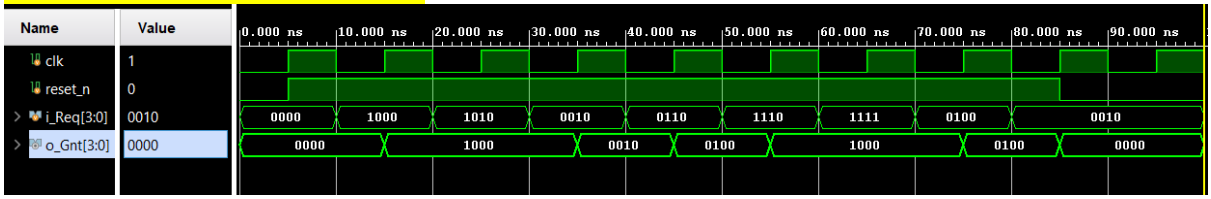
VERILOG CODE

```
module Fixed_Priority_Arbite(
input clk,reset_n,
input [3:0]i_Req,
output reg [3:0] o_Gnt
);
always @ (posedge clk or negedge reset_n)
begin
   if(!reset_n)
   o_Gnt <= 4'b0000;
   else if(i_Req[3])
   o_Gnt <= 4'b1000;
   else if(i_Req[2])
   o_Gnt <= 4'b0100;
   else if(i_Req[1])
   o_Gnt <= 4'b0010;
   else if(i_Req[0])
   o_Gnt <= 4'b0001;
   else
   o_Gnt <= 4'b0000;
    end
endmodule
```

TEST BENCH

```
`timescale 1ns / 1ps

module Fixed_Priority_Arbite_TB;
  reg          clk;
  reg          reset_n;
  reg [3:0]    i_Req;
  wire [3:0]   o_Gnt;
  Fixed_Priority_Arbite DUT(
     clk,reset_n,i_Req,o_Gnt);
  always #5 clk = ~clk;
  initial begin
    clk = 0;
    reset_n = 0;
    i_Req = 4'b0;
    #5 reset_n = 1;
    @(negedge clk) i_Req = 4'b1000;
    @(negedge clk) i_Req = 4'b1010;
    @(negedge clk) i_Req = 4'b0010;
    @(negedge clk) i_Req = 4'b0110;
    @(negedge clk) i_Req = 4'b1110;
    @(negedge clk) i_Req = 4'b1111;
    @(negedge clk) i_Req = 4'b0100;
    @(negedge clk) i_Req = 4'b0010;
    #5 reset_n = 0;
    end
    initial
    #100 $stop;
endmodule
```

OUTPUT SIMULATION WAVEFORM



DAY-27
ROUND ROBIN ARBITER

VERILOG CODE:

```
`timescale 1ns / 1ps
module Round_Robin_Arbiter(
input clk,rst,[3:0]req,
output reg [3:0]grant
    );
    reg [2:0]present_state;
    reg [2:0]next_state;
    parameter [2:0]ideal=3'b000;
    parameter [2:0]s0=3'b001;
    parameter [2:0]s1=3'b010;
    parameter [2:0]s2=3'b011;
    parameter [2:0]s3=3'b100;

   //combinational logic

    always@(posedge clk or negedge rst)
    begin
    if(!rst)
    present_state<=ideal;
    else
    present_state<=next_state;
    end

   //sequential logic
   always@(*)
   begin
   case(present_state)
   ideal:begin
         if(req[0])
         begin next_state=s0;end
         else if(req[1])
         begin next_state=s1;end
         else if(req[2])
         begin next_state=s2; end
         else if(req[3])
         begin next_state=s3;end
         else
         begin next_state=ideal;end
         end
```

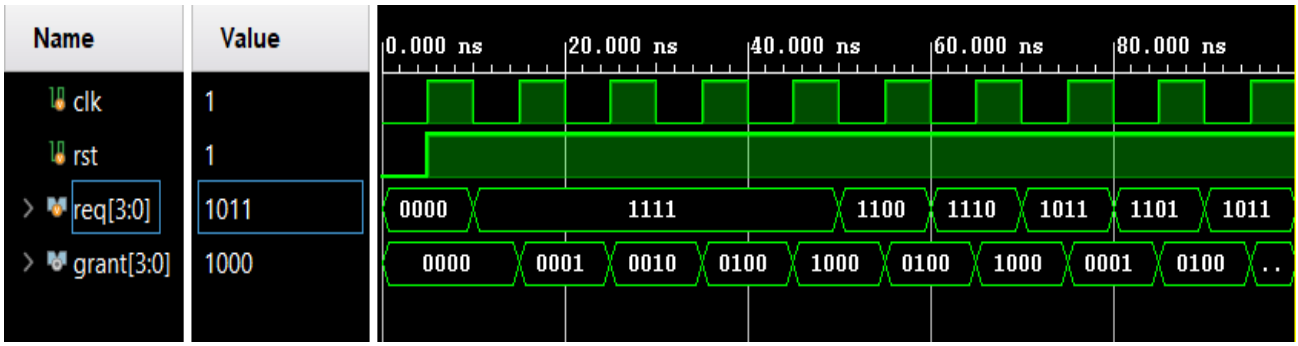
```
s0:begin
  if(req[1])
  begin next_state=s1;end
  else if(req[2])
  begin next_state=s2;end
  else if(req[3])
  begin next_state=s3; end
  else if(req[0])
  begin next_state=s0;  end
  else
  begin  next_state=ideal; end
  end

s1:begin
  if(req[2])
  begin next_state=s2;end
  else if(req[3])
  begin next_state=s3;end
  else if(req[0])
  begin next_state=s0; end
  else if(req[1])
  begin next_state=s1; end
  else
  begin next_state=ideal; end
  end

s2:begin
  if(req[3])
  begin next_state=s3; end
  else if(req[0])
  begin  next_state=s0; end
  else if(req[1])
  begin  next_state=s1; end
  else if(req[2])
  begin  next_state=s2; end
  else
  begin next_state=ideal; end
  end
```

```
      s3:begin
         if(req[0])
         begin next_state=s0; end
         else if(req[1])
         begin next_state=s1; end
         else if(req[2])
         begin next_state=s2; end
         else if(req[3])
         begin next_state=s3; end
         else
         begin next_state=ideal;end
         end
         endcase
         end
    //output logic//
    always@(*)
    begin
    case(present_state)
    s0:begin grant=4'b0001;end
    s1:begin grant=4'b0010;end
    s2:begin grant=4'b0100;end
    s3:begin grant=4'b1000;end
    default:begin grant=4'b0000;end
    endcase
    end
endmodule
```

TEST BENCH

```
`timescale 1ns / 1ps
module tb_RRA_fixed_time_slices;
reg clk,rst;
reg [3:0]req;
wire [3:0]grant;
Round_Robin_Arbiter dut(clk,rst,req,grant);
always #5 clk=~clk;
initial begin
clk=0; rst=0; req=4'b0;
#5 rst=1;
@(negedge clk)req=4'b1111;
@(negedge clk)req=4'b1111;
@(negedge clk)req=4'b1111;
@(negedge clk)req=4'b1111;
@(negedge clk)req=4'b1100;
@(negedge clk)req=4'b1110;
@(negedge clk)req=4'b1011;
@(negedge clk)req=4'b1101;
@(negedge clk)req=4'b1011;
@(negedge clk)req=4'b0111;
#5 rst=0;end
initial #100 $stop;
endmodule
```

OUTPUT SIMULATION WAVEFORM:



DAY-28
UP_DOWN COUNTER

VERILOG CODE:

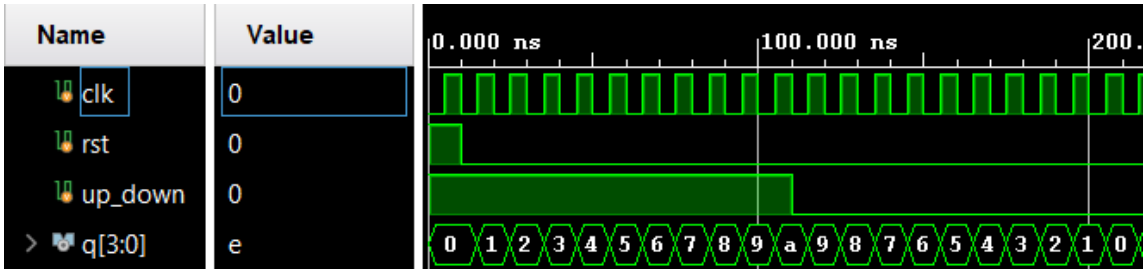
Counter acts as Up_counter if (up_down=1)
Down_counter if (up_down=0)

```
module up_down_counter(
input clk,rst,up_down,
output reg [3:0]q
    );
    always @(posedge clk or posedge rst)
    begin
    if(rst)
    q<=4'b0;
    else if(up_down)
    q<=q+1;
    else
    q<=q-1;
    end
endmodule
```

TEST BENCH

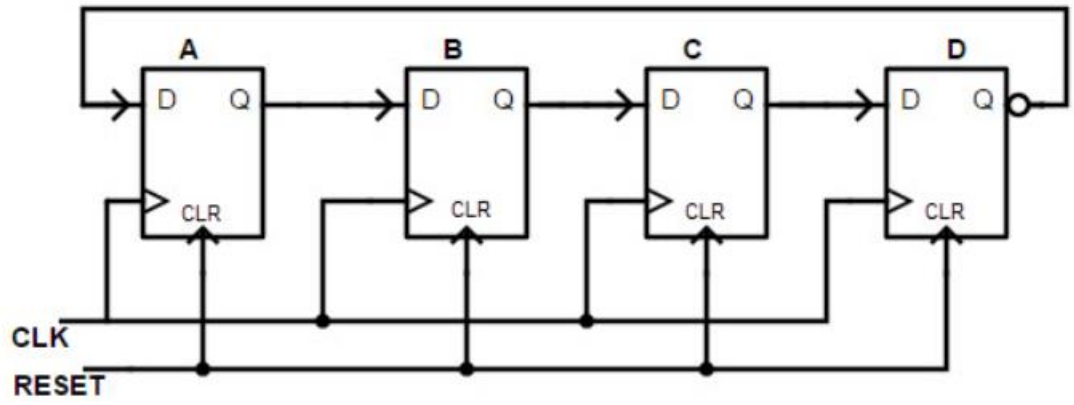
```
`timescale 1ns / 1ps

module tb_up_down_counter;
reg clk,rst,up_down;
wire [3:0]q;
up_down_counter dut(clk,rst,up_down,q);
always #5 clk=~clk;
initial begin
rst=1;clk=0;up_down=1;
#10 rst=0;
#100 up_down=0;
#200 $stop;
end
endmodule
```

OUTPUT SIMULATION WAVEFORM



DAY-29

RING COUNTER



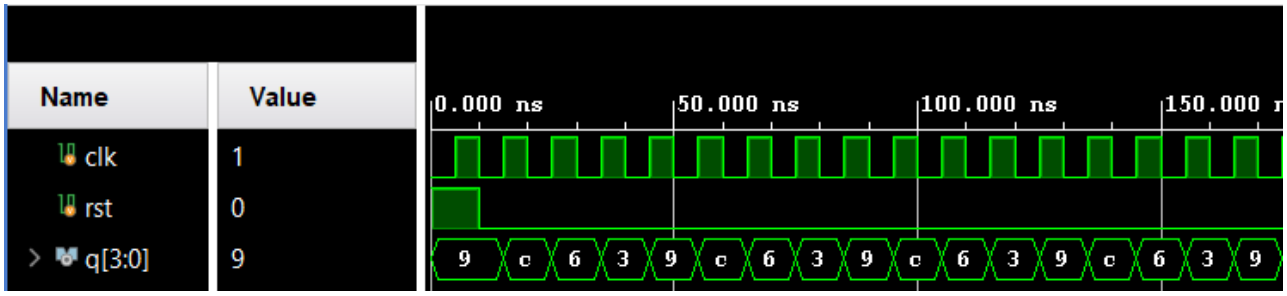
VERILLOG CODE:

```
module Ring_Counter(
input clk,rst,output reg [3:0]q);
    always@(posedge clk or posedge rst)
    begin
    if(rst)
    q<=4'b1001; //initial value loaded to flip fop
    else
    q<={q[0],q[3:1]};
    end
endmodule
```

TEST BENCH:

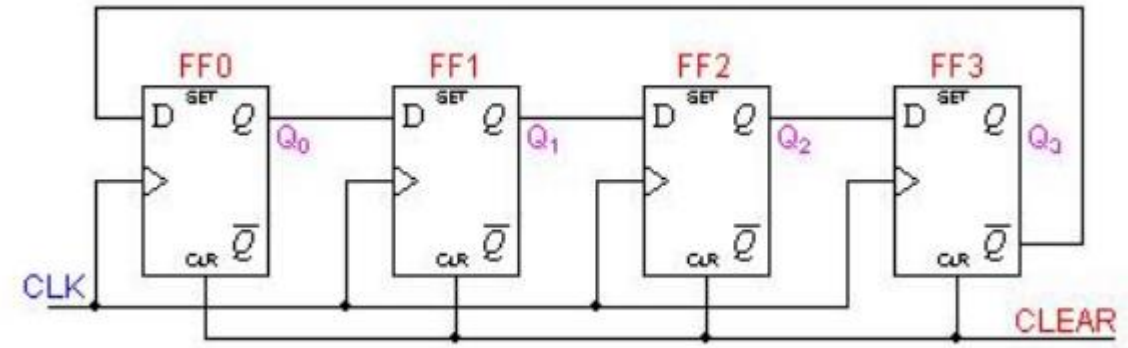
```
`timescale 1ns / 1ps
module tb_ring_counter();
reg clk,rst;
wire [3:0]q;
Ring_Counter dut(clk,rst,q);
always #5 clk=~clk;
initial begin
clk=0;rst=1;
#10 rst=0;
#200 $stop;
end
endmodule
```

OUTPUT SIMULATION WAVEFORM:



DAY-30

JOHNSON COUNTER



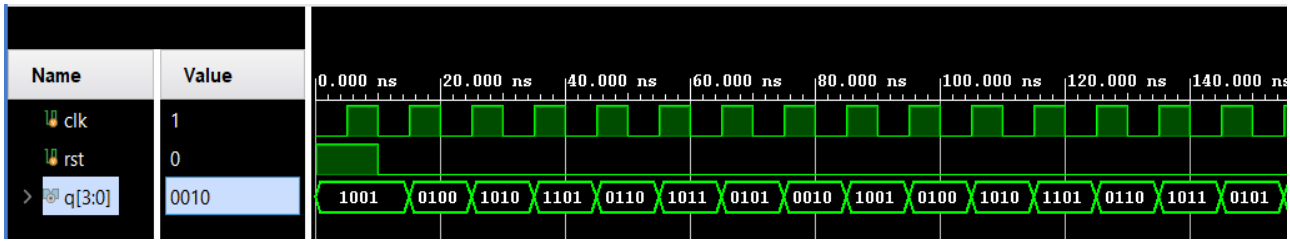
VERILOG CODE:

```
`timescale 1ns / 1ps
module johnson_counter(
input clk,rst,output reg [3:0]q
);
    always@(posedge clk or posedge rst)
    begin
    if(rst)
    q<=4'b1001; //initial value loaded to flip fop
    else
    q<={~q[0],q[3:1]};
    end
endmodule
```

TEST BENCH:

```
`timescale 1ns / 1ps

module tb_johnson_counter();
reg clk,rst;
wire [3:0]q;
johnson_counter dut(clk,rst,q);
always #5 clk=~clk;
initial begin
clk=0;rst=1;
#10 rst=0;
#200 $stop;
end
endmodule
```

OUTPUT SIMULATION WAVEFORM:



Difference Between Ring Counter and Johnson Counter

The difference between the ring counter and johnson counter is, the inverter output of the last flip-flop is connected back as the input to the first flip-flop.

In-ring counter, the no.of input clock pulses given to the flip-flops are equal to the no.of stages. That means the MOD of the n-bit ring counter is 'n'.

In johnson's counter, the no.of input clock pulses divide by a factor of twice equal to the no.of stages. That means the MOD of the n-bit johnson counter is '2n'.

Advantages

- The johnson counter counts the no.of stages twice equal to the no.of clock pulses given to the flip-flops.
- It counts the events in a continuous closed loop within the circuit.
- It can be designed by using D and JK flip-flops
- It can be used as a self-decoding circuit.

Disadvantages

- It cannot be used to count the binary sequence
- It doesn't utilize all the stages equal to the no.of stages in the counter.
- It needs only half the no.of flip-flops on half the no.of timing signals
- It is used in any timing sequence.

Applications

- Johnson counters are used as frequency dividers and pattern recognizers.
- It is used as a synchronous decade counter and divider circuit
- It can be used to create complicated finite state machines in hardware logic design.
- The 3-bit johnson counter is used as a 3-phase square wave generator to produce 120 degrees phase shift
- The frequency of the clock signal is divided by varying their feedback.

DAY-31

BOOTH MULTIPLIER

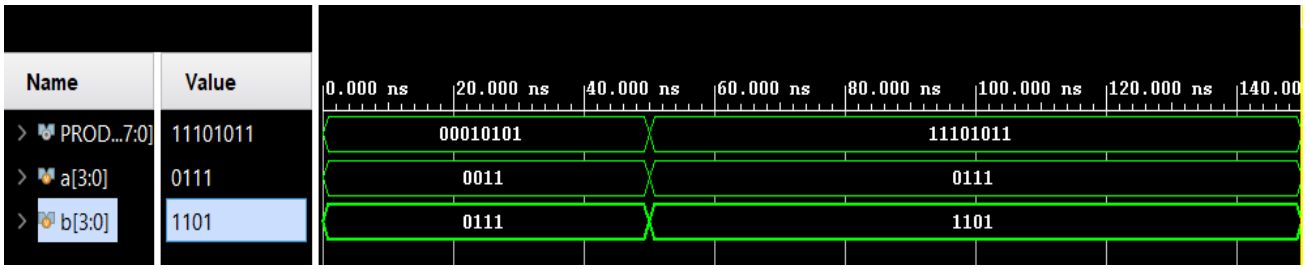
VERILOG CODE

```
`timescale 1ns / 1ps
module Booth_Mutltiplier(PRODUCT, A, B);
output reg [7:0] PRODUCT;
input [3:0] A, B;
reg [1:0] temp;
integer i;
reg e;
reg [3:0] B1;
always @(A,B)
begin
PRODUCT = 8'd0;
e = 1'b0;
B1 = -B;
for (i=0; i<4; i=i+1)
 begin
temp = { A[i], e };
 case(temp)
2'd2 : PRODUCT[7:4] = PRODUCT[7:4] + B1;
 2'd1 : PRODUCT[7:4] = PRODUCT[7:4] + B;
 endcase
PRODUCT = PRODUCT >> 1;
PRODUCT[7] = PRODUCT[6];
e=A[i];
end
end
endmodule
```

TEST BENCH

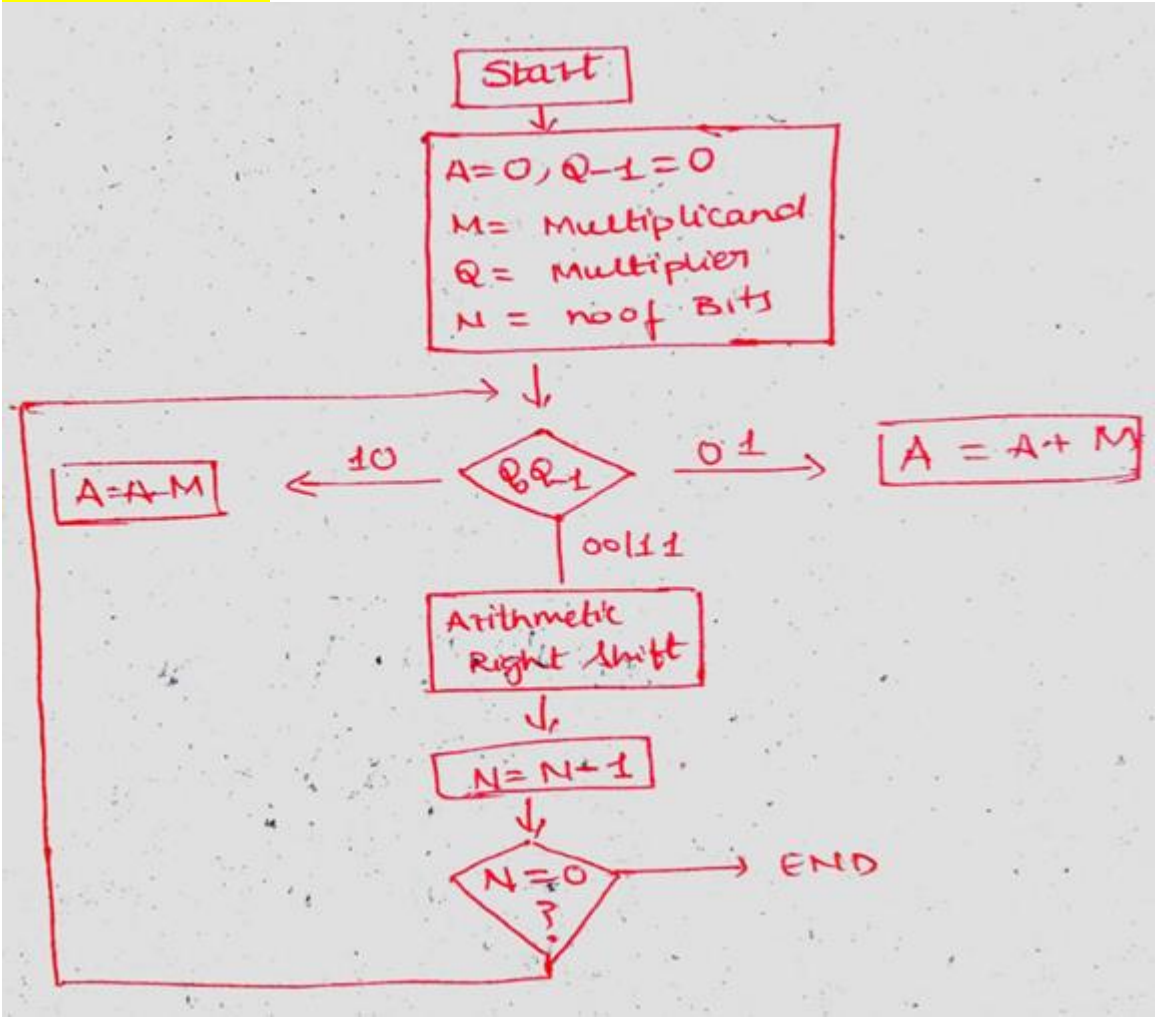
```
`timescale 1ns / 1ps
module tb_Booth_Multiplier();
wire [7:0] PRODUCT;
reg [3:0] a,b;
Booth_Mutltiplier dut(PRODUCT, a,b);
initial begin
a=4'b0011;b=4'b0111;
#50
a=4'b0111;b=4'b1101;
#100 $stop;
end
endmodule
```

OUTPUT:



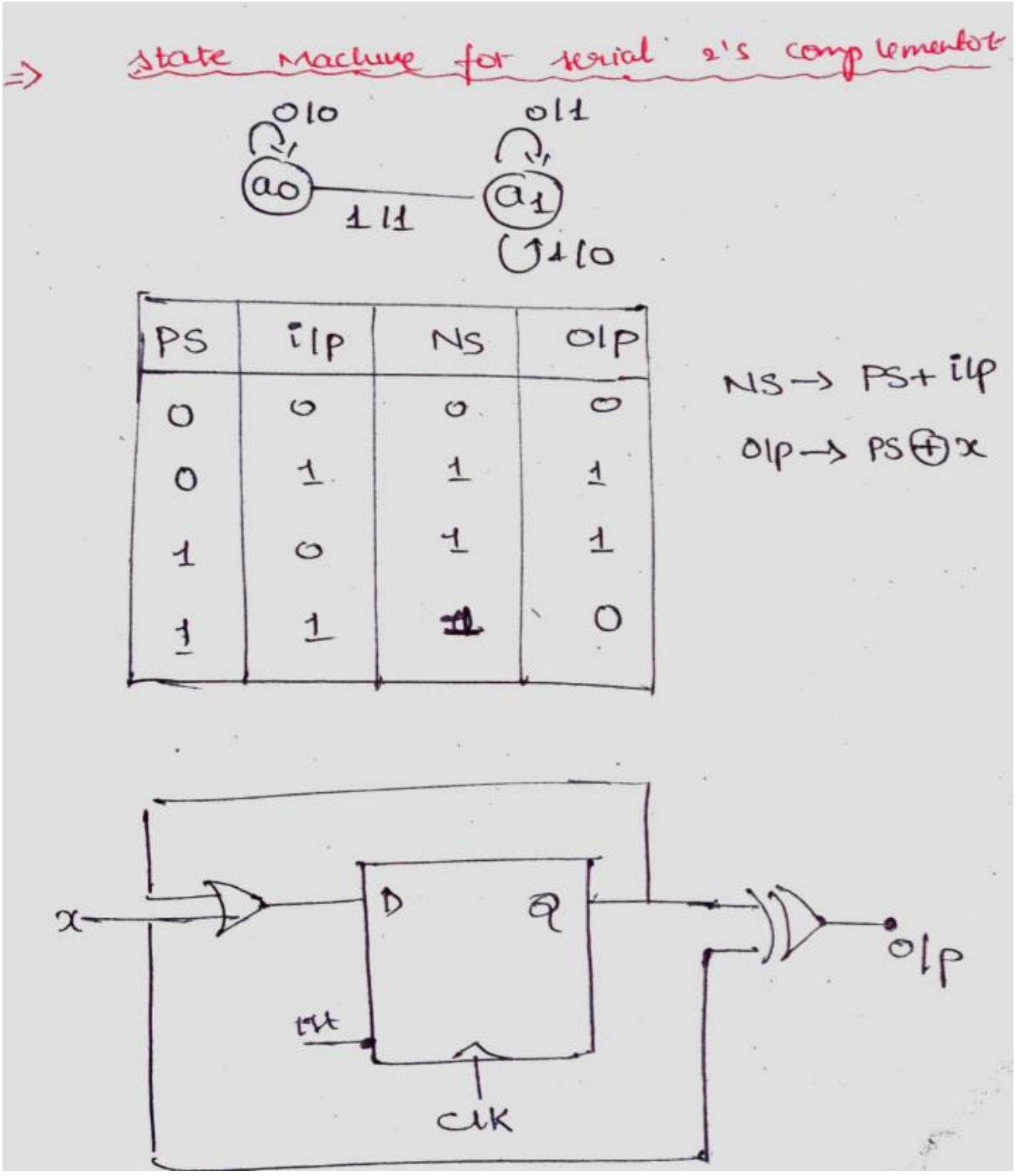
A. **7*3 = 21 (0001_0101)**
B. **7*-3= -21 (1110_1011)**

BOOTH'S ALGORITHM



DAY-32

Give the state machine for serial 2's complementor and design the same using D-Flip Flop.



Verilog code:

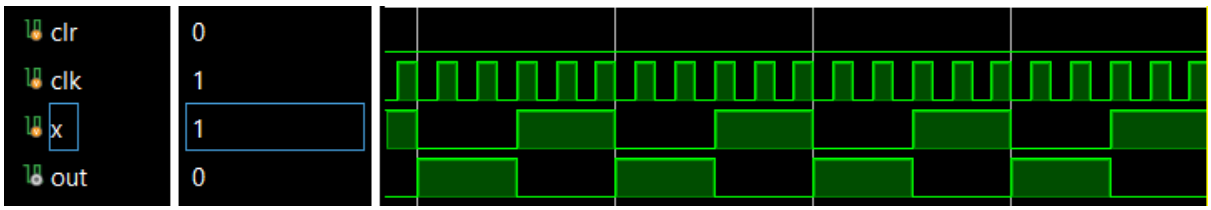
```
`timescale 1ns / 1ps
module d_ff(
input clk,rst,d,
output reg q);
    always @(posedge clk or posedge rst)
    begin
    if(rst)q<=0;
    else begin
    q<=d; end
    end
endmodule
```

```
module serial_twos_complementor(
input clk,clr,x,
output out);
wire q;
d_ff dut(clk,clr,q|x,q);
assign out=x^q;
endmodule
```

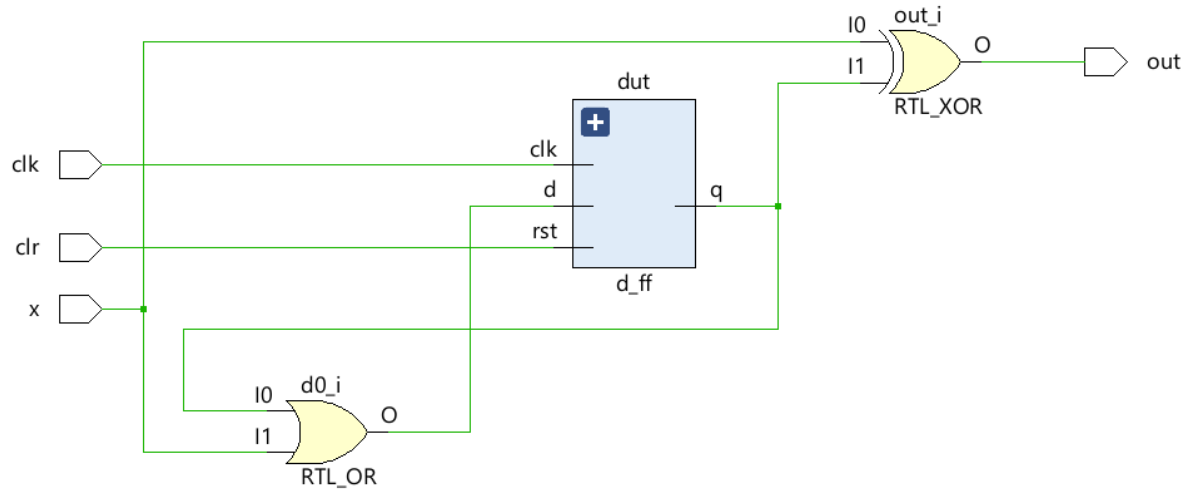
Test Bench:

```
module tb_serial_twos_complementor();
reg clk,clr,x;
wire out;
serial_twos_complementor s1(clk,clr,x,out);
initial begin
clr=1;clk=0;x=0;
#5 clr=0;
end
always #5 x=~x;
always #1 clk=~clk;
initial #50 $stop;

endmodule
```

Output Simulation Waveform:



Schematic:



Day-33

A CN flip-flop has four operations: no change, clear to 0, no change, complement, when inputs C and N are 00, 01, 10, and 11, respectively.
(a) Tabulate the characteristic table. **(b) * Derive the characteristic equation.**
(c) Show how the CN flip-flop with a D flip-flop and 2X1 Multiplexer.

→ Change - No Change Flip-Flop:

C	N	Next state Q(N+1)	state
0	0	Q(N)	nochange
0	1	0	Reset
1	0	Q(N)	nochange
1	1	Q'(N)	Toggle

Truth Table

C	N	Qn	Qn+1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

K-Map

Q(n+1)

C \ N Qn	00	01	11	10
0	0	1	0	0
1	0	1	0	1

$Q(n+1) = CN\bar{Q}_n + \bar{N}Q_n$

Implementation of Change No Change Flip Flop using 2 x 1 multiplexers.

Verilog code of 2x1 Mux:

```
module mux2X1(a,b,s,y);
input a,b,s;
output reg y;
always @(a or b or s)
begin
case(s)
0: y=a;
1:y=b;
default: y=1'b0;
endcase
end
endmodule
```

Verilog code of D-Flip Flop:

```
`timescale 1ns / 1ps
module d_ff(d,clk,reset,q);
input d,clk,reset;
output reg q;
always @(posedge clk)
begin
if(reset)
q=0;
else
q=d;
end
endmodule
```

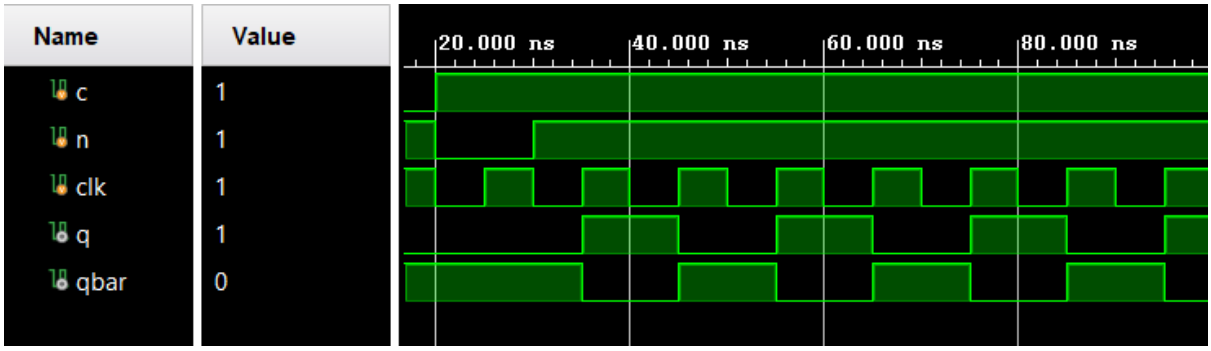
Verilog code of Change- No Change flip flop:

```
module cn_flipflop(c,n,clk,q,qbar);
input c,n,clk;
output q,qbar;
wire cn,n_bar,d_wire;
mux2X1 mux1(1'b0,c,n,cn);
mux2X1 mux2(1'b1,1'b0,n,n_bar);
mux2X1 mux3(cn,n_bar,q,d_wire);
d_ff dff1( .d(d_wire),.clk(clk),.reset(),.q(q));
assign qbar=~q;
endmodule
```

Test Bench:

```
`timescale 1ns / 1ps
module cn_flipflop_tb;
    reg c;
    reg n;
    reg clk;
    wire q,qbar;
    cn_flipflop uut (.c(c),.n(n),.clk(clk), .q(q),.qbar(qbar));
    initial begin
        c = 0;
        n = 0;
        clk = 0;
        #10 c=0;n=1;
        #10 c=1;n=0;
        #10 c=1;n=1;
    end
    always #5 clk=~clk;
    initial
    begin $monitor("C=%b | N=%b | Q=%b | Qbar=%b",c,n,q,qbar);
    #100 $finish;
    end
endmodule
```

Output Simulation Waveform:



Day -34
Implementation of Greatest Common Divisor in behaviour level.

Verilog code:

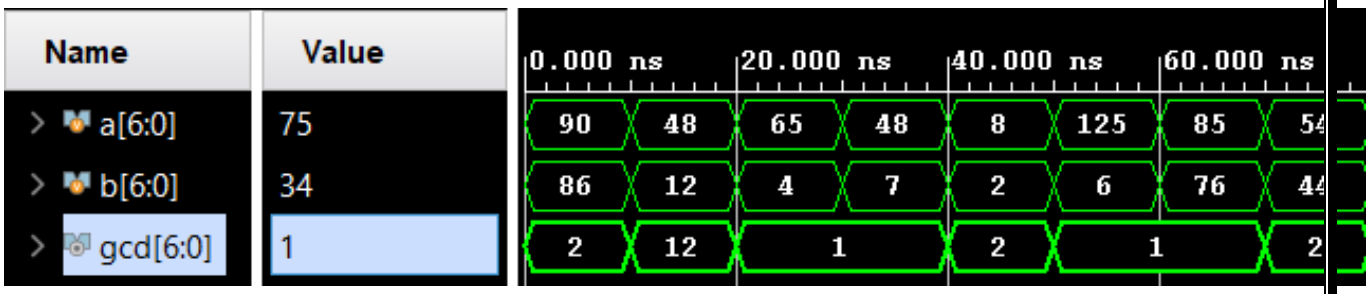
```
`timescale 1ns / 1ps
module gcd_behaviour(
input [6:0]a,b,
output reg [6:0]gcd);
    reg [6:0]ain,bin;
    always@(*)
    begin
    ain=a;bin=b;
    while(ain!=bin)
    begin
    if(ain<bin)
    bin=bin-ain;
    else
    ain=ain-bin;
    end
    gcd=ain;
    end
endmodule
```

Test Bench:

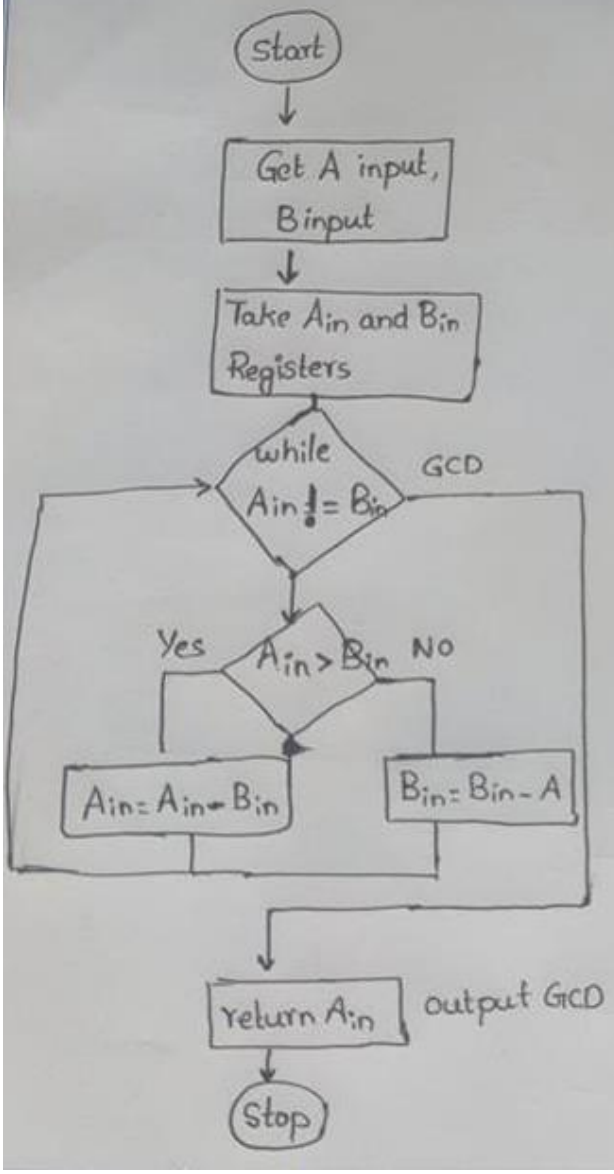
```
`timescale 1ns / 1ps
module tb_gcd();
reg [6:0]a,b;
wire [6:0]gcd;
gcd_behaviour dut(a,b,gcd);
initial begin
        a=90;b=86;
        #10 a=48;b=12;
        #10 a=65;b=4;
        #10 a=48;b=7;
        #10 a=8;b=2;
        #10 a=125;b=6;
        #10 a=85;b=76;
        #10 a=54;b=44;
        #10 a=95;b=32;
        #10 a=109;b=91;
        #10 a=75;b=34;
    end
      initial
        begin
        $monitor("a= %d | b=%d | gcd=%d", a,b,gcd);
        #200 $finish;
        end
endmodule
```

Output simulation waveform:

```
a=  90 | b= 86 | gcd=  2
a=  48 | b= 12 | gcd= 12
a=  65 | b=  4 | gcd=  1
a=  48 | b=  7 | gcd=  1
a=   8 | b=  2 | gcd=  2
a= 125 | b=  6 | gcd=  1
a=  85 | b= 76 | gcd=  1
a=  54 | b= 44 | gcd=  2
a=  95 | b= 32 | gcd=  1
a= 109 | b= 91 | gcd=  1
a=  75 | b= 34 | gcd=  1
```



Greatest Common Divisor



Drawback of This Circuit:

- As we are using Behavioural modelling to develop GCD verilog cod while loop is used as a looping statement which is not Synthesizeable
- As it is a combinational circuit, more iterations are not possible.
- In while loop, we don't know how many iterations that's why it is not Synthesizeable S
- It can't be implemented on Hardware (FPGA)
- Hence, GCD via Datapath and Controller Unit can be implemented to overcome the above drawback.

DAY-35

Write a Verilog module to illustrate the equivalence between @(a or b) and @(a | b).

- In Verilog, @( ) is used to trigger a simulation event based on the values of signals given inside it.

- The @(a or b) syntax uses the logical operator or to specify that the simulation event will be triggered when either signal a or signal b changes value. Similarly, the @(a | b) syntax uses the bitwise OR operator |, but in Verilog, the bitwise OR operation is the same as the logical OR operation for single-bit signals, so it's equivalent to @(a or b).

VERILOG MODULE:

```
`timescale 1ns / 1ps
module Example_Event;
reg a, b;
integer time_counter;

always @(a or b) begin
$display("@(a or b) event triggered at time %0t", $time);
time_counter = time_counter + 1;
end

always @(a | b) begin
$display("@(a | b) event triggered at time %0t", $time);
time_counter = time_counter + 1;
end

initial begin
a = 0; b = 0;
time_counter = 0;
// Simulate changing values of a and b
repeat (5) begin
#10;
a = !a;
#15;
b = !b;
end
// Display the total number of triggered events
$display("Total triggered events: %d", time_counter);
$finish;
end
endmodule
```

CONSOLE WINDOW:

```
@(a or b) event triggered at time 0
@(a | b) event triggered at time 0
@(a or b) event triggered at time 10000
@(a | b) event triggered at time 10000
@(a or b) event triggered at time 25000
@(a or b) event triggered at time 35000
@(a or b) event triggered at time 50000
@(a | b) event triggered at time 50000
@(a or b) event triggered at time 60000
@(a | b) event triggered at time 60000
@(a or b) event triggered at time 75000
@(a or b) event triggered at time 85000
@(a or b) event triggered at time 100000
@(a | b) event triggered at time 100000
@(a or b) event triggered at time 110000
@(a | b) event triggered at time 110000
Total triggered events:          16
```

DAY-36

Write a Verilog module to demonstrate the difference between full case and parallel case.

Verilog Module:

```
`timescale 1ns / 1ps
module CaseExample;
reg [1:0] ctrl;
initial begin
ctrl = 2'b01;


// Full Case Style
case (ctrl)
2'b00: $display("Full Case: Control is 00");
2'b01: $display("Full Case: Control is 01");
2'b10: $display("Full Case: Control is 10");
2'b11: $display("Full Case: Control is 11");
endcase
// Parallel Case Style
case (ctrl)
2'b00, 2'b01: $display("Parallel Case: Control is 00 or 01");
2'b10: $display("Parallel Case: Control is 10");
2'b11: $display("Parallel Case: Control is 11");
endcase
end
endmodule
```

Console Window:

```
Time resolution is 1 ps
Full Case: Control is 01
Parallel Case: Control is 00 or 01
```

In Verilog, "full case" and "parallel case" are terms used to describe different ways of writing case statements to describe the behaviour of a specific input condition. These terms pertain to the structure and style of the case statement.

Full Case:

A "full case" style of writing a case statement includes all possible input conditions explicitly. In other words, each possible value of the control expression is listed along with its corresponding set of actions. If any value is not explicitly listed, it is assumed to have no effect.

```
case (ctrl)
2'b00: action_00;
2'b01: action_01;
2'b10: action_10;
2'b11: action_11;
endcase
```

In this example, all four possible 2-bit values of ctrl (00, 01, 10, and 11) are listed along with their respective actions.

Parallel Case:

A "parallel case" style of writing a case statement groups together actions that share common behaviour for certain input conditions. In other words, multiple input values may result in the same set of actions, and those actions are grouped together.

Here's an example of a parallel case statement:

```
case (ctrl)
2'b00, 2'b01: action_0x; // Both 00 and 01 cases trigger action_0x
2'b10: action_10;
2'b11: action_11;
endcase
```

In this example, both 2'b00 and 2'b01 cases trigger the action_0x behaviour. In summary: "Full case" style includes all possible input conditions explicitly. "Parallel case" style groups together input conditions that have the same behaviour.

DAY-37

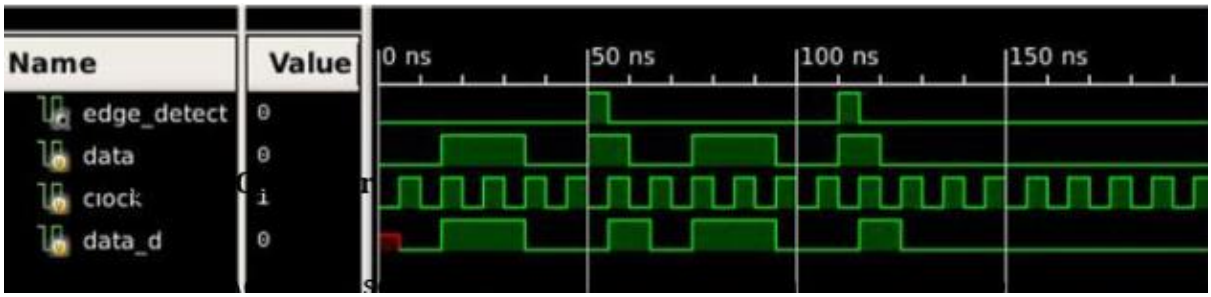
POS_EDGE DETECTOR

```
module posedge_detector(
    input  data,
     input clock,
    output edge_detect
    );
        reg data_d;

    always @ (posedge clock) begin
        data_d <= data;
    end
    assign edge_detect = data & ~data_d;
endmodule
```

Test Bench:

```
 `timescale 1ns / 1ps
module tb_edge_detector;
     reg data, clock;
     wire edge_detect;
     posedge_detector dut (
          .data(data),
          .clock(clock),
          .edge_detect(edge_detect));
initial begin
          data = 0;
          clock = 0;
          #15 data = 1;
          #20 data= 0;
          #15 data = 1;
          #10 data = 0;
          #15 data = 1;
          #20 data= 0;
          #15 data = 1;
          #10 data = 0;
     end
     always #5 clock=~clock;
     initial begin
     $monitor("Data =%b, Edge_detect=%b ", data,edge_detect);
     #200 $stop;
     end
endmodule
```

Output Simulation Wave form:



DAY-38

1010 SEQUENCE DETECTOR BY MOORE MACHINE

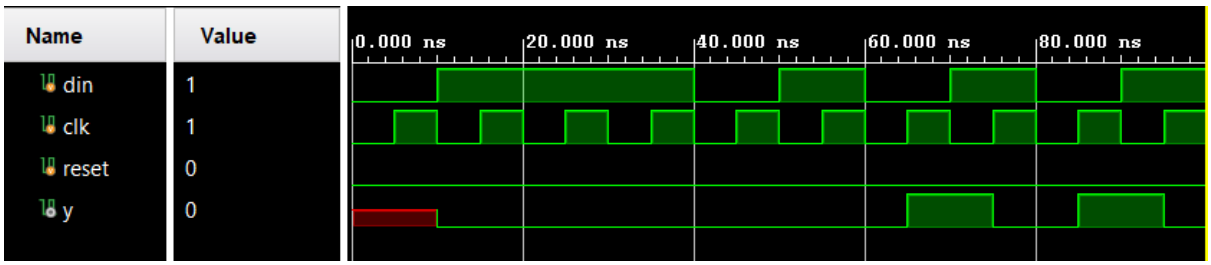
```
`timescale 1ns / 1ps
module moore_fsm(din, reset, clk, y);
input din;
input clk;
input reset;
output reg y;
reg [2:0] cst, nst;
localparam S0 = 3'b000,
           S1 = 3'b001,
           S2 = 3'b010,
           S3 = 3'b100,
             S4 = 3'b101;
always @(cst or din)
 begin
 case (cst)
   S0: if (din == 1'b1)
           begin
          nst = S1;
           y=1'b0;
           end
      else nst = cst;
   S1: if (din == 1'b0)
           begin
         nst = S2;
           y=1'b0;
           end
       else
           begin
           nst = cst;
           y=1'b0;
           end
```

```
   S2: if (din == 1'b1)
           begin
          nst = S3;
           y=1'b0;
           end
             else
           begin
           nst = S0;
           y=1'b0;
           end
   S3: if (din == 1'b0)
           begin
          nst = S4;
           y=1'b0;
           end
       else
           begin
           nst = S1;
           y=1'b0;
           end
S4: if (din == 1'b0)
           begin
          nst = S1;
           y=1'b1; end
           else
           begin
           nst = S3;
           y=1'b1; end
   default: nst = S0;
  endcase
end
always@(posedge clk)
begin
           if (reset)
           cst <= S0;
           else
             cst <= nst;
end
endmodule
```

TEST BENCH:

```
module tb_moore_fsm();
reg din,clk,reset;
wire y;
moore_fsm m1(din, reset, clk, y);
initial
begin
reset=0;clk=0;din=0;
$monitor($time, , ,"c=%b",clk,,"y=%b",y,,"r=%b",reset,,"d=%b",din);
#10 din=1;
#10 din=1;
#10 din=1;
#10 din=0;
#10 din=1;
#10 din=0;
#10 din=1;
#10 din=0;
#10 din=1;

end
always
#5 clk=~clk;
initial
#100 $finish ;
endmodule
```

OUTPUT WAVEFORM:



DAY-39

1001 SEQUENCE DETECTOR BY MEALY MACHINE

VERILOG CODE:

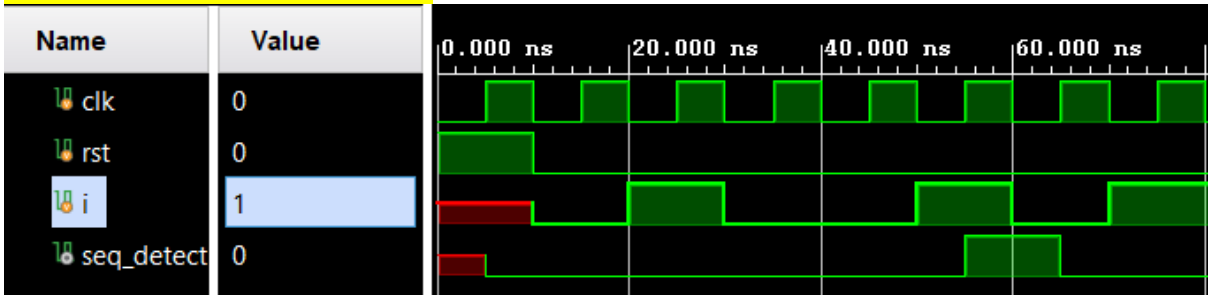
```
`timescale 1ns / 1ps
module melay_1001_overlap(
input clk,rst,i,output seq_detect);
    reg [2:0]ps,ns;
    parameter s0=3'b000;
    parameter s1=2'b001;
    parameter s2=2'b010;
    parameter s3=2'b011;
    parameter s4=3'b100;

    always @(posedge clk)
    begin
    if(rst)
    ps<=s0;
    else
    ps<=ns;
    end

    assign seq_detect=(ps==s4)?1'b1:1'b0;

    always @(ps,i) begin
    case(ps)
    s0: if(i==1)
    ns<=s1;
    else
    ns<=s0;

    s1: if(i==0)
    ns<=s2;
    else
    ns<=s1;

    s2: if(i==0)
    ns<=s3;
    else
    ns<=s1;

    s3: if(i==1)
    ns<=s4;
    else
    ns<=s0;
```

```
    s4: if(i==1)
    ns<=s1;
    else
    ns<=s0;

    default:ns<=s0;
    endcase
    end


endmodule
```

TEST BENCH:

```
`timescale 1ns / 1ps
module tb_seq_melay;
reg clk,rst,i;
wire seq_detect;
melay_1001_overlap dut ( clk,rst,i,seq_detect );
always #5 clk=~clk;
initial begin
clk=0;rst=1;
#10 rst=0;i=0;
#10 i=1;
#10 i=0;
#10 i=0;
#10 i=1;
#10 i=0;
#10 i=1;
#10 i=1;
#10 i=1;
#150 $stop();
end
endmodule
```

OUTPUT SIMULATION WAVEFORM:



DAY-40

1011_SEQUENCE DETECTOR MELAY MACHINE(OVERLAP):

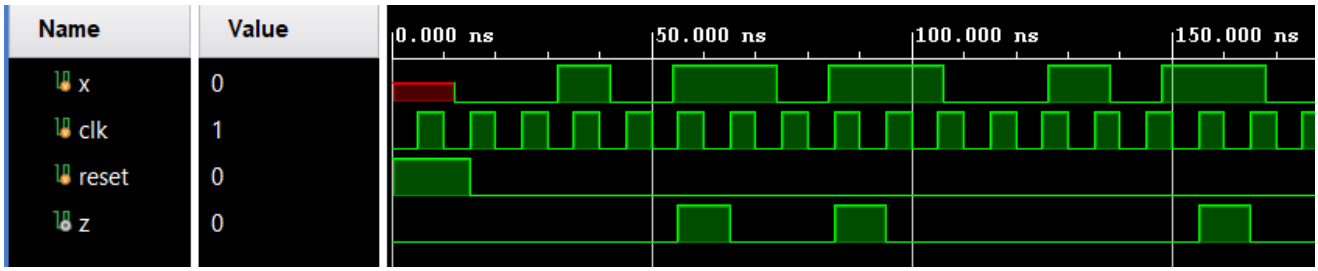
VERILOG CODE :

```
module melay_1011_overlap(
input x,clk,reset,
output reg z);
parameter S0 = 0 , S1 = 1 , S2 = 2 , S3 = 3 ;
reg [1:0] PS,NS ;

    always@(posedge clk or posedge reset)
        begin
            if(reset)
                PS <= S0;
            else
                PS <= NS ;
        end

    always@(PS or x)
        begin
            case(PS)
                S0 : begin
                            z = 0 ;
                            NS = x ? S1 : S0 ;
                        end
                S1 : begin
                            z = 0 ;
                            NS = x ? S1 : S2 ;
                        end
                S2 : begin
                            z = 0 ;
                            NS = x ? S3 : S0 ;
                        end
                S3 : begin
                            z = x ? 1 : 0 ;
                            NS = x ? S1 : S2 ;
                        end

            endcase
        end
endmodule
```

TESTBENCH:

```
module test_1011;
    reg x;
    reg clk;
    reg reset;
    wire z;
   melay_1011_overlap  dut (x,clk,reset,z);

initial
    begin
        clk = 1'b0;
        reset = 1'b1;
        #15 reset = 1'b0;
    end

always #5 clk = ~ clk;

initial begin
        #12 x = 0;#10 x = 0 ; #10 x = 1 ; #10 x = 0 ;
        #12 x = 1;#10 x = 1 ; #10 x = 0 ; #10 x = 1 ;
        #12 x = 1;#10 x = 0 ; #10 x = 0 ; #10 x = 1 ;
        #12 x = 0;#10 x = 1 ; #10 x = 1 ; #10 x = 0 ;
        #10 $stop;
    end
    endmodule
```

OUTPUT SIMULATION WAVEFORM:



DAY-41

2 X 2 VEDIC MULITPLIER:

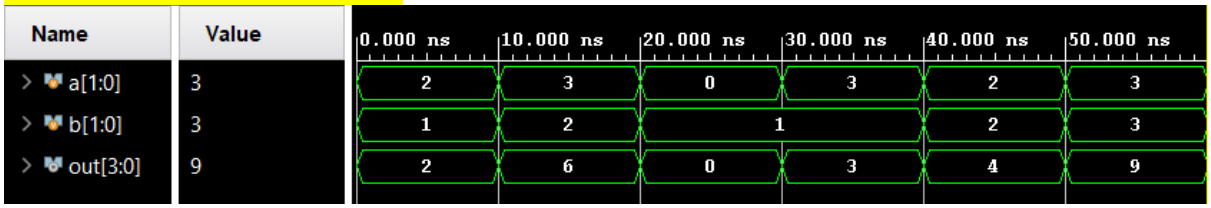
VERILOG CODE:

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

module vedic_mul_2_2(
 input [1:0] a,b,
    output [3:0] out
    );
    wire [3:0] w;

    and m1(out[0],a[0],b[0]);
    and m2(w[0],a[0],b[1]);
    and m3(w[1],a[1],b[0]);
    and m4(w[2],a[1],b[1]);

    half_adder ha1(w[0],w[1],out[1],w[3]);
    half_adder ha2(w[3],w[2],out[2],out[3]);

endmodule
```

TEST BENCH:

```
module test_bench_vedic();
reg [1:0]a,b;wire [3:0]out;
vedic_mul_2_2 dut(a,b,out);
always begin
    a=2'd2;
    b=2'd1;
    #10;
    a=2'd3;
    b=2'd2;
    #10;
    a=2'd0;
    b=2'd1;
    #10;
    a=2'd3;
    b=2'd1;
    #10;
    a=2'd2;
    b=2'd2;
    #10;
    a=2'd3;
    b=2'd3;
    #10;
    end
initial begin
$monitor("%d * %d = %d", a,b,out);
#60 $stop;
end
endmodule
```

OUTPUT SIMULATION WINDOW:



CONSOLE:

```
2 * 1 =  2
3 * 2 =  6
0 * 1 =  0
3 * 1 =  3
2 * 2 =  4
3 * 3 =  9
```

DAY 42

IMPLEMENTATION OF 15 X 15 ROM

A ROM, or Read-Only Memory, is a type of digital memory device that stores data permanently. Unlike RAM (Random Access Memory), ROM retains its contents even when the power is turned off, hence the term "read-only." This means that the data stored in ROM cannot be modified or overwritten after it has been programmed.

VERILOG CODE:

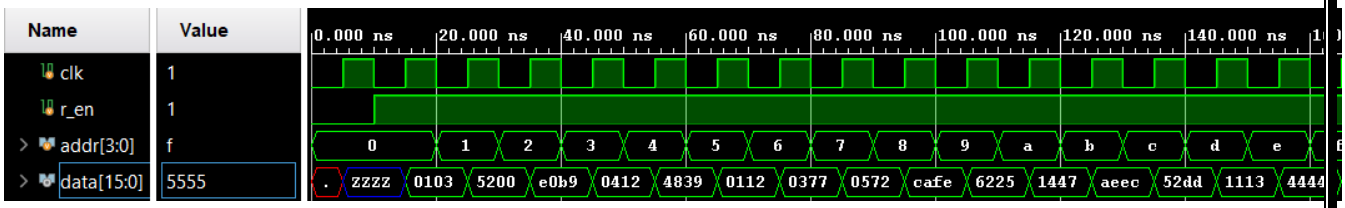
```
module rom(
    input clk, r_en,
    input [3:0] addr,
    output reg [15:0] data);

    reg [15:0] mem [15:0];

    always@(posedge clk) begin
        if(r_en)
            data <= mem[addr];
        else
            data <= 'bz;
    end
endmodule
```

TEST BENCH:

```
`timescale 1ns / 1ps

module test_bench_rom();
reg clk;
reg r_en;
reg [3:0] addr;
wire [15:0] data;

rom dut(clk, r_en, addr, data);

initial
    begin
      clk = 0;
      forever #5 clk <= ~clk;
      end
initial begin

// Initialize memory
      dut.mem[0] = 16'h0103;
      dut.mem[1] = 16'h5200;
      dut.mem[2] = 16'he0b9;
      dut.mem[3] = 16'h0412;
      dut.mem[4] = 16'h4839;
      dut.mem[5] = 16'h0112;
      dut.mem[6] = 16'h0377;
      dut.mem[7] = 16'h0572;
      dut.mem[8] = 16'hcafe;
      dut.mem[9] = 16'h6225;
      dut.mem[10] = 16'h1447;
      dut.mem[11] = 16'haeec;
      dut.mem[12] = 16'h52dd;
      dut.mem[13] = 16'h1113;
      dut.mem[14] = 16'h4444;
      dut.mem[15] = 16'h5555;
```

```
// Apply stimulus to address input
        addr = 0;
        r_en =0;
        #10 r_en = 1;
        #10 addr = 1;
        #10 addr = 2;
        #10 addr = 3;
        #10 addr = 4;
        #10 addr = 5;
        #10 addr = 6;
        #10 addr = 7;
        #10 addr = 8;
        #10 addr = 9;
        #10 addr = 10;
        #10 addr = 11;
        #10 addr = 12;
        #10 addr = 13;
        #10 addr = 14;
        #10 addr = 15;

        #10 $finish;
    end
endmodule
```

OUTPUT SIMULATION WAVFORM:



DAY-43

CREATION OF SIMPLE PULSE WIDTH MODULATION of different duty cycles.

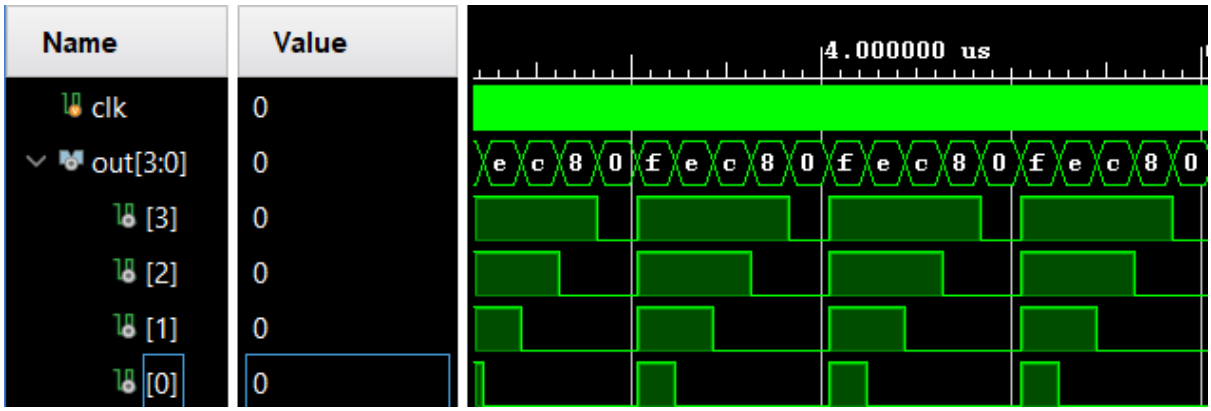
VERILOG CODE:

```
module PWM(input clk,output [3:0]out);

 reg [7:0] counter=0;
 always @(posedge clk) begin
 if(counter <100)
 begin
 counter<=counter+1;
 end
 else counter<=0;
 end

 assign out[0]=(counter<20)?1:0;  //duty cycle 20 %//
 assign out[1]=(counter<40)?1:0;  //duty cycle 40 %//
 assign out[2]=(counter<60)?1:0;  //duty cycle 60 %//
 assign out[3]=(counter<80)?1:0;  //duty cycle 80 %//

endmodule
```

TEST BENCH:

```
`timescale 1ns / 1ps
module tb_pwm;
reg clk;
wire [3:0]out;
PWM dut (clk,out);
initial clk=1'b0;
always #5 clk=~clk;
endmodule
```

OUTPUT SIMULATION WAVEFORM:



DAY-44

COMPUTATION OF SQAURE ROOT AND CUBE ROOT OF A NUMBER:

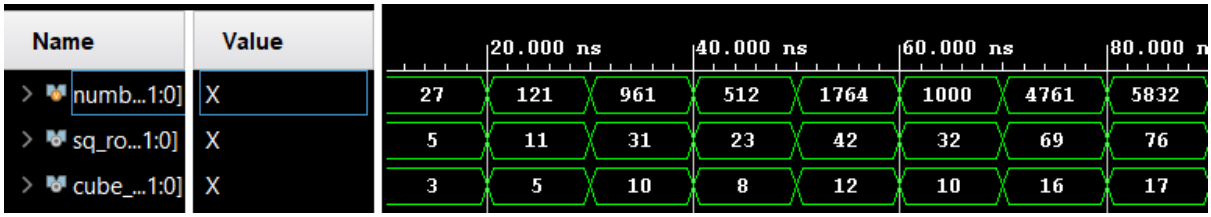
VERILOG CODE:

```
module root(
    input [31:0] number,
    output reg [31:0] sq_root, cube_root
    );
    real red;
    always@(number) begin
        find_sq_root(number, sq_root);
        find_cube_root(number, cube_root);
        end

    task find_sq_root;
        input [31:0] num;
        output [31:0] res;
        begin
            res = num**(0.5);
        end
    endtask

    task find_cube_root;
        input [31:0] num;
        output [31:0] res;
        begin
            res= num**(0.33);
        end
    endtask
endmodule
```

TEST BENCH:

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

reg [31:0] number;
wire [31:0] sq_root, cube_root;

root dut(number, sq_root, cube_root);

initial begin
    #10 number = 27;
    #10 number = 121;
    #10 number = 961;
    #10 number = 512;
    #10 number = 1764;
    #10 number = 1000;
    #10 number = 4761;
    #10 number = 5832;
    #10 $finish;
end
endmodule
```

OUTPUT SIMUATION WAVEFORM:



DAY-45

PARITY GENERATOR:

Assigns result 1 if number of 1's in input is odd.
Assigns result 0 if number of 1's in input is even.

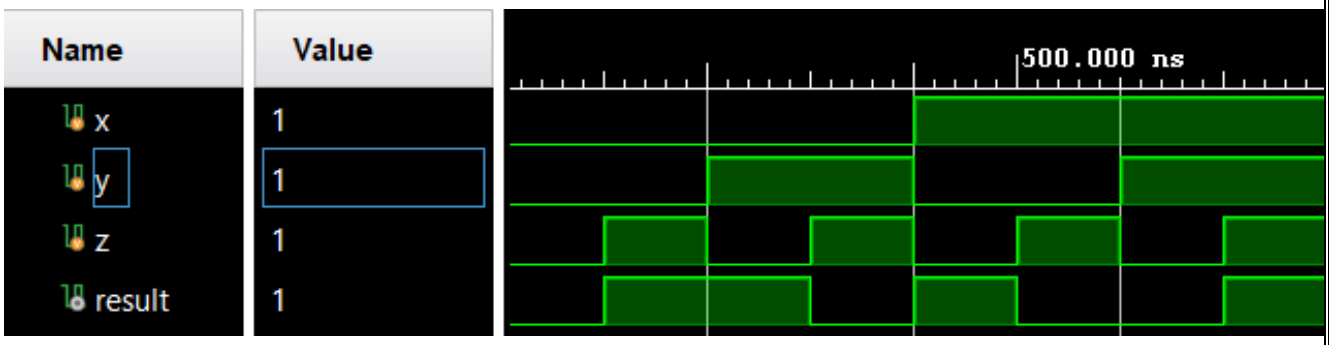
VERILOG CODE:

```
`timescale 1ns / 1ps
module parity_generator(
x,y,z,result);
input x,y,z;
output result;
xor (result,x,y,z);
endmodule
```

TEST_BENCH:

```
`timescale 1ns / 1ps
module tb_parity_generator;
reg x;reg y;reg z;
    wire result;
parity_generator uut (x,y,z,result);
    initial begin
x = 0;y = 0;z = 0;
#100;
x = 0;y = 0;z = 1;
#100;
x = 0;y = 1;z = 0;
#100;
x = 0;y = 1;z = 1;
#100;
x = 1;y = 0;z = 0;
#100;
x = 1;y = 0;z = 1;
#100;
x = 1;y = 1;z = 0;
#100;
x = 1;y = 1;z = 1;
#100;
end
endmodule
```

OUTPUT SIMULATION WAVEFORM:



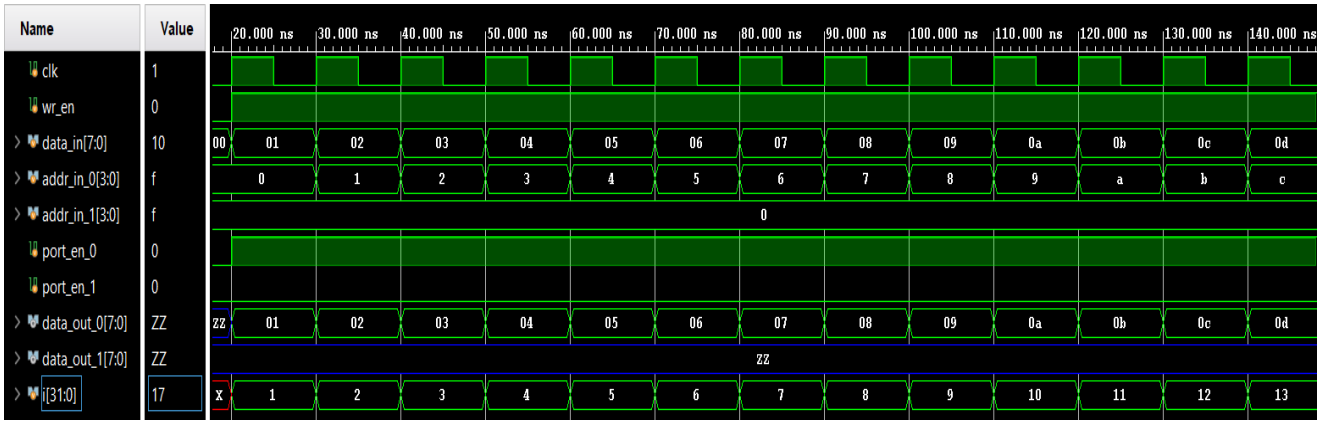
DAY-46

IMPLEMENTATION OF A SIMPLE DUAL PORT RAM

VERILOG CODE:

```
module Dual_port_ram
    (   input clk, //clock
        input wr_en,    //write enable for port 0
        input [7:0] data_in,    //Input data to port 0.
        input [3:0] addr_in_0,  //address for port 0
        input [3:0] addr_in_1,  //address for port 1
        input port_en_0,    //enable port 0.
        input port_en_1,    //enable port 1.
        output [7:0] data_out_0,    //output data from port 0.
        output [7:0] data_out_1 //output data from port 1.
    );

//memory declaration.
reg [7:0] ram[0:15];

//writing to the RAM
always@(posedge clk)
begin
    if(port_en_0 == 1 && wr_en == 1)    //check enable signal and if write enable is ON
        ram[addr_in_0] <= data_in;
end

//always reading from the ram, irrespective of clock.
assign data_out_0 = port_en_0 ? ram[addr_in_0] : 'dZ;
assign data_out_1 = port_en_1 ? ram[addr_in_1] : 'dZ;
endmodule
```

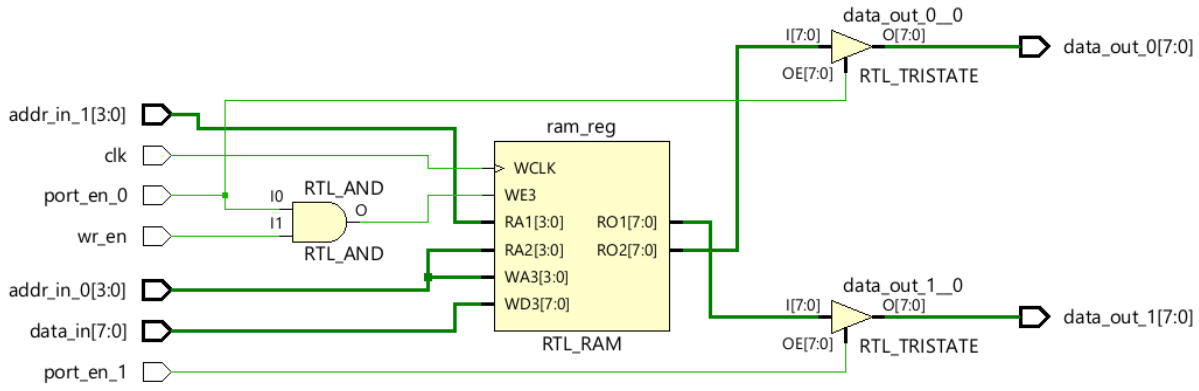
OUTPUT:



TEST BENCH:

```
`timescale 1ns / 1ps
module tb_dual_port_ram();
// Inputs
    reg clk;
    reg wr_en;
    reg [7:0] data_in;
    reg [3:0] addr_in_0;
    reg [3:0] addr_in_1;
    reg port_en_0;
    reg port_en_1;

    // Outputs
    wire [7:0] data_out_0;
    wire [7:0] data_out_1;
    integer i;

    // Instantiate the Unit Under Test (UUT)
    Dual_port_ram uut (
        .clk(clk),
        .wr_en(wr_en),
        .data_in(data_in),
        .addr_in_0(addr_in_0),
        .addr_in_1(addr_in_1),
        .port_en_0(port_en_0),
        .port_en_1(port_en_1),
        .data_out_0(data_out_0),
        .data_out_1(data_out_1));
```

```
    always
        #5 clk = ~clk;

    initial begin
        // Initialize Inputs
        clk = 1;
        addr_in_1 = 0;
        port_en_0 = 0;
        port_en_1 = 0;
        wr_en = 0;
        data_in = 0;
        addr_in_0 = 0;
        #20;
        //Write all the locations of RAM
        port_en_0 = 1;
        wr_en = 1;
      for(i=1; i <= 16; i = i + 1) begin
            data_in = i;
            addr_in_0 = i-1;
            #10;
        end
        wr_en = 0;
        port_en_0 = 0;
        //Read from port 1, all the locations of RAM.
        port_en_1 = 1;
        for(i=1; i <= 16; i = i + 1) begin
            addr_in_1 = i-1;
            #10;
        end
        port_en_1 = 0;
    end
endmodule
```



DAY – 47

CLOCK PHASE CHANGER

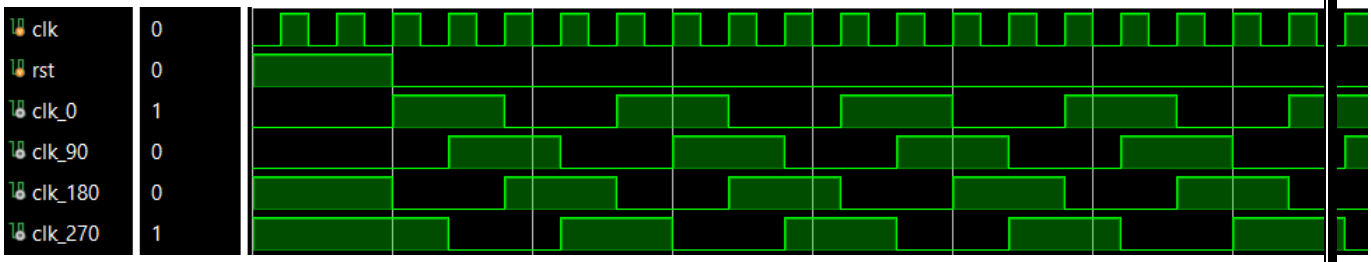
VERILOG CODE:

```
module clk_phase(
    input clk, rst,
    output clk_0, clk_90, clk_180, clk_270 );
    reg [1:0] count;
    reg div_2;
    always @(posedge clk or posedge rst)
    begin
        if(rst)
            count <= 0;
        else
            count <= {~count[0], count[1]};
    end

    always @(posedge clk or posedge rst)
    begin
        if (rst)
            div_2 <= 0;
        else
            div_2 = ~div_2;
    end

    assign clk_0 = count[1];
    assign clk_90 = count[1] ^ div_2;
    assign clk_180 = ~count[1];
    assign clk_270 = ~clk_90;

endmodule
```

TEST BENCH:

```
module test_bench();
    reg clk;
    reg rst;
    wire clk_0;
    wire clk_90;
    wire clk_180;
    wire clk_270;
clk_phase dut(clk, rst, clk_0, clk_90, clk_180, clk_270);
initial begin
clk = 0;
forever #1 clk = ~clk;
end
initial begin
rst = 1;
#5 rst = 0;
#50 $finish;
end
endmodule
```

OUTPUT SIMULATION WAVEFORM:



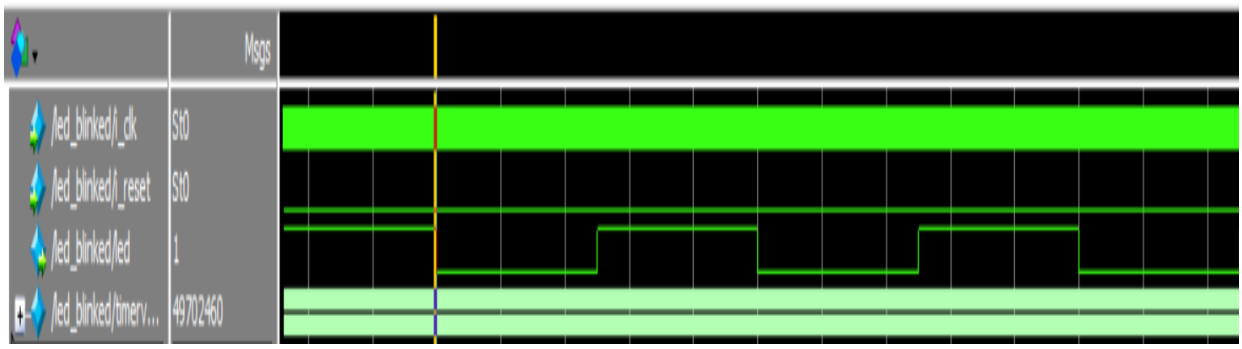
DAY -48

LED BLINKER

VERILOG CODE:

```
`timescale 1ns / 1ps
module led_blinked(
input i_clk, //input 100Mz clock
input i_reset,
output reg led
    );
 reg [25:0]timervalue;

 //TIMER
 always@(posedge i_clk or posedge i_reset)
 begin
 if (i_reset)
      timervalue <=0;
      else if(timervalue<26'd50000000)
      begin
      timervalue<=timervalue+1'b1;
      end
      else
      timervalue <=26'd0;
end

always @(posedge i_clk or posedge i_reset)
begin
if(i_reset)
led<=1'b0;
else
begin
   if(timervalue ==26'd50000000)
   led<=~led;

   end
end
endmodule
```

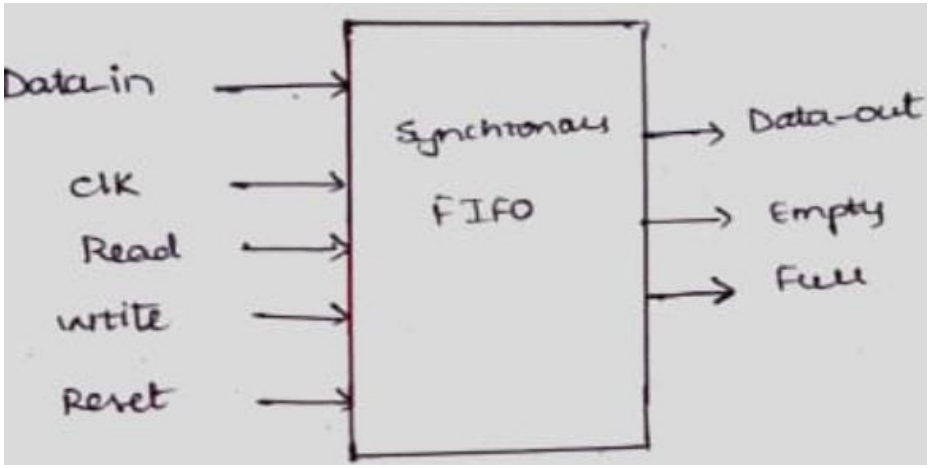
OUTPUT SIMULATION WAVEFORM:



LED turns on after 50000000 ps and turns off after 50000000 ps.

DAY-49

DESIGNING OF SYNCHRONOUS FIFO

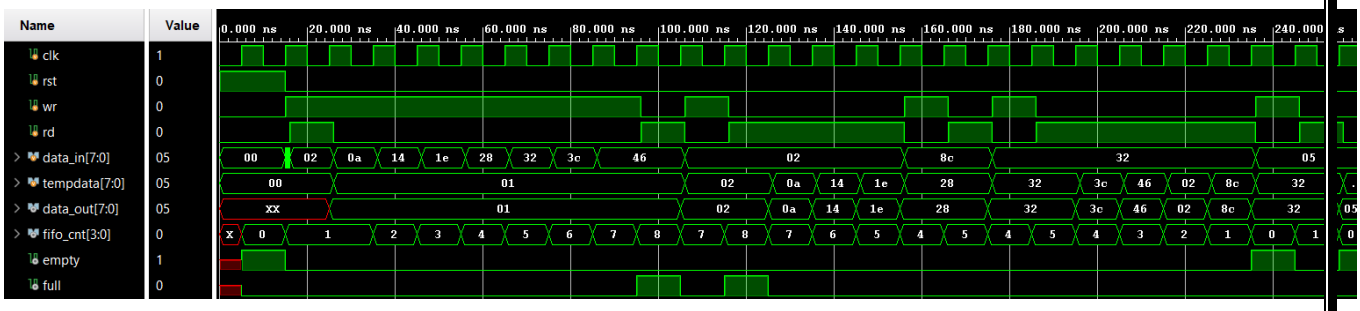


VERILOG CODE:

```
`timescale 1ns / 1ps
module FIFO (
input [7:0] data_in, input clk, rst, rd, wr,
output empty, full, output reg [3:0]fifo_cnt,
output reg [7:0] data_out);

reg [7:0] fifo_ram [0:7];
reg [2:0] rd_ptr, wr_ptr;
assign empty= (fifo_cnt==0);
assign full =(fifo_cnt==8);
always @(posedge clk) begin: write
if (wr && ! full)
fifo_ram [wr_ptr] <= data_in;
else if (wr && rd)
fifo_ram [wr_ptr] <= data_in;
end

//Read and Write Clock
always @ (posedge clk) begin: read
if (rd && !empty)
data_out <= fifo_ram [rd_ptr];
else if (rd && wr)
data_out <= fifo_ram [rd_ptr];
end
```

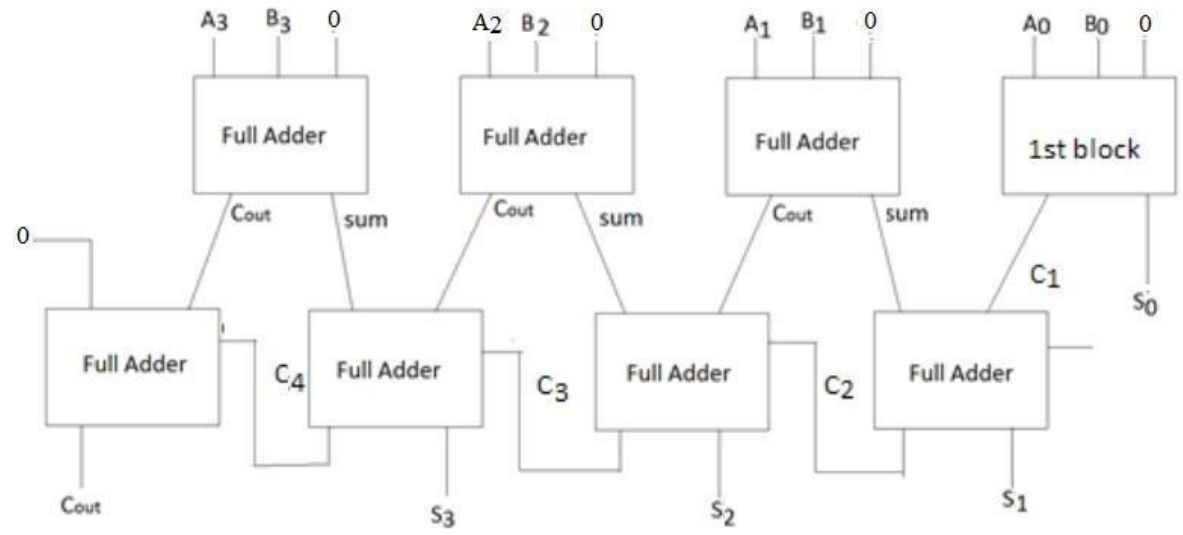
```
//pointer block
always @(posedge clk) begin: pointer
if (rst) begin
wr_ptr <= 0;
rd_ptr <= 0;
end
else begin
wr_ptr <= ((wr && ! full) || (wr && rd)) ? wr_ptr+1 :
wr_ptr;
rd_ptr <= ((rd && !empty) || (wr && rd)) ? rd_ptr+1:
rd_ptr;
end
end

//counter
always @(posedge clk) begin: count
if (rst) fifo_cnt <= 0;
else begin
case ({wr, rd})
2'b00: fifo_cnt <= fifo_cnt;
2'b01: fifo_cnt <= (fifo_cnt==0) ? 0: fifo_cnt-1;
2'b10: fifo_cnt <= (fifo_cnt==8) ? 8: fifo_cnt+1;
2'b11 : fifo_cnt <= fifo_cnt;
default: fifo_cnt <= fifo_cnt;
endcase
end
end
endmodule
```

OUTPUT WAVEFORM:



DAY-50

IMPLEMENTATION OF 4 BIT CARRY SAVE ADDER



VERILOG CODE FOR FULL ADDER:

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

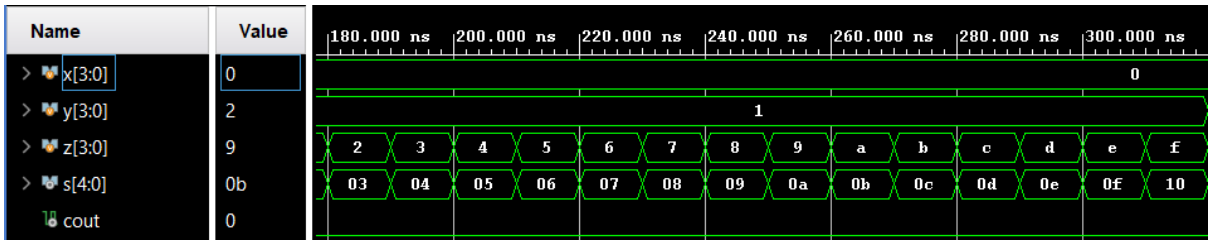
VERILOG CODE FOR CARRY SAVE ADDER:

```
module Carry_Save_Adder
(input [3:0] x,y,z,
output [4:0] s,output cout);

wire [3:0] c1,s1,c2;

full_adder fa_inst10(x[0],y[0],z[0],s1[0],c1[0]);
full_adder fa_inst11(x[1],y[1],z[1],s1[1],c1[1]);
full_adder fa_inst12(x[2],y[2],z[2],s1[2],c1[2]);
full_adder fa_inst13(x[3],y[3],z[3],s1[3],c1[3]);

full_adder fa_inst20(s1[1],c1[0],1'b0,s[1],c2[1]);
full_adder fa_inst21(s1[2],c1[1],c2[1],s[2],c2[2]);
full_adder fa_inst22(s1[3],c1[2],c2[2],s[3],c2[3]);
full_adder fa_inst23(1'b0,c1[3],c2[3],s[4],cout);

assign s[0] = s1[0];
endmodule
```

TEST BENCH:

```
module Tb_CSA();
reg [3:0] x,y,z;
    wire [4:0] s;
    wire cout;
    integer i,j,k,error;

    Carry_Save_Adder uut (x,y,z,s,cout);
    initial begin
        x = 0;y = 0;z = 0;
        error = 0;
      for(i=0;i<16;i=i+1) begin
            for(j=0;j<16;j=j+1) begin
                for(k=0;k<16;k=k+1) begin
                    x = i;
                    y = j;
                    z = k;
                    #10;
                    if({cout,s} != (i+j+k))
                        error <= error + 1;
                end
            end
      end
    end
endmodule
```

OUTPUT SIMULATION WAVEFORM:



THANK YOU

B.MITHUN CHAKRAVARTHI