13EC437 ECAD & VLSI Lab

(Lab Manual)
Verilog Programs
For IV Year I Sem ECE

Prepared by Dr. K. Swaminathan
(swami.nitt@gmail.com)

E-CAD AND VLSI LAB :

COURSE OBJECTIVES:

1. To learn the HDL programming language.
2. To learn the simulation of basic gates using the basic programming language.
3. To learn the simulation of combinational circuits using programming language.
4. To learn the simulation of sequential circuits using programming language.
5. To learn the synthesis and layouts of analog and digital CMOS circuits.
6. To develop an ability to simulate and synthesize various digital circuits.

COURSE OUTCOMES:

The students will be able to

1. Simulate various digital circuits.
2. Simulate and synthesize various CMOS circuits.
3. Understand the layout design rules for both static CMOS and dynamic clocked CMOS Circuits.
4. Develop an ability of designing of analog and digital CMOS circuits.

List of Experiments

List of Experiments

Design and implementation of the following CMOS digital/analog circuits using Cadence/ Mentor Graphics/ Synopsys/ GEDA/Equivalent CAD tools. The design shall include Gate-level design. Transistor –level design, Hierarchical design, Verilog HDL/VHDL design, Logic synthesis, Simulation and verification, scaling of CMOS inverter for different technologies,

study of secondary effects (temperature, power supply and process corners), circuit optimization with respect to area, performance and/or power, Layout Extraction of parasitics and back annotation, modifications in circuit parameters and layout consumption, DC/ transient analysis, Verification of layours (DRC, LVS)

E-CAD programs:

Programming can be done using any complier. Down load the programs on FPGA/CPLD boards and performance testing may be done using pattern generator (32 channels) and logic analyzer apart from verification by simulation with any of the front end tools.

1. **HDL code to realize all the logic gates**
2. **Design of 2-to-4 decoder**
3. **Design of 8-to-3 encoder (without and with parity)**
4. **Design of 8-to-1 Multiplexer/ Demultiplexer**
5. **Design of 4 bit binary to gray converter**
6. **Design of comparator**
7. **Design of Full adder using 3 modeling styles**
8. **Design of flip flops: SR, D, JK, T**
9. **Design of 4-bit binary, BCD counters (synchronous/ asynchronous reset) or any sequence counter**
10. **Finite State Machine Design**

VLSI programs:

1. **Introduction to layout design rules**
2. **Layout, physical verification, placement & route for complex design, static timing analysis, IR drop analysis and crosstalk analysis of the following:**

Basic logic gates

a. **CMOS inverter**
b. **CMOS NOR/ NAND gates**
c. **CMOS XOR and MUX gates**
d. **CMOS 1-bit full adder**
e. **Static / Dynamic logic circuit (register cell)**
f. **Latch**
g. **Pass transistor**
h. **Layout of any combinational circuit (complex CMOS logic gate)- Learning about data paths**
i. **Introduction to SPICE simulation and coding of NMOS/CMOS circuit**
j. **SPICE simulation of basic analog circuits: Inverter / Differential amplifier**
k. **Analog Circuit simulation (AC analysis) – CS & CD amplifier**
l. **System level design using PLL**

1. **Simulation using all the modeling styles and Synthesis of all the logic gates using Verilog HDL**

Aim: Implement and verify the functionality of AND gate using Xilinx ISE

Apparatus required: Electronics Design Automation Tools used

- Xilinx Spartan 3 FPGA
- Xilinx ISE Simulator tool
- Xilinx XST Synthesis tool
- Xilinx Project Navigator 8.1i
- JTAG cable
- Adaptor 5v/4A

Logic gates:

A logic gate is an idealized or physical device implementing a Boolean function; that is, it performs a logical operation on one or more binary inputs, and produces a single binary output

Boolean equations:

AND Gate:Y = (A.B)

OR Gate: Y = (A + B)

NAND Gate: Y = (A.B)'

NOR Gate: Y = (A+B)'

XOR Gate: Y = A.B' + A'.B

XNOR Gate: Y = A.B + A'.B'

NOT gate: Y=A'

AND Gate - Block diagram:

OR gate - Truth Table

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

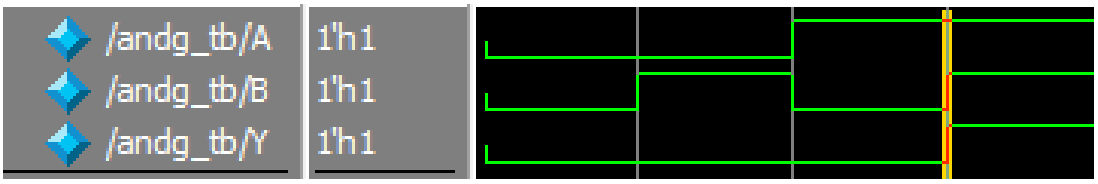
Verilog program for AND gate:

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

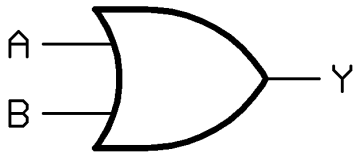
Verilog test bench program for AND gate:

```
module andg_tb;
reg A, B;
wire Y;
andg andgate(.A(A), .B(B),.Y(Y));
initial begin
       A =1'b0;B= 1'b0;
       #10 A =1'b0;B= 1'b1;
       #10 A =1'b1;B= 1'b0;
       #10 A =1'b1;B= 1'b1;
       #10
  $finish;
    end
always @(Y)
    $display( "time =%0t \tINPUT VALUES: \t A=%b B =%b \t output value Y  =%b",$time,A,B,Y);
endmodule
```

Wave Form:



OR Gate - Block diagram:



OR gate - Truth Table

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

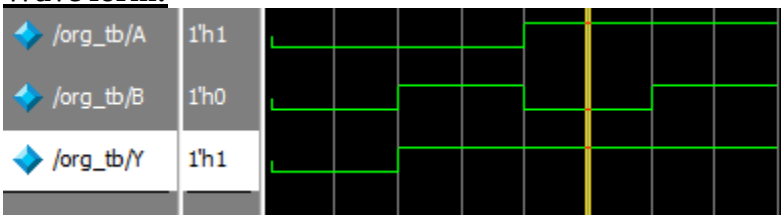
Verilog program for OR gate:

```
module org (A, B, Y);
        input A, B;
        output Y;
        assign Y = A | B;
endmodule
```

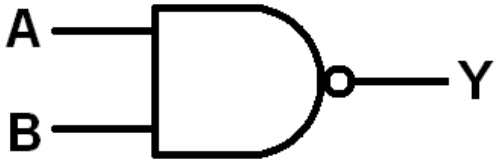
Verilog test bench program for AND gate:

```
module org_tb;
reg A, B;
wire Y;
andg andgate(.A(A), .B(B),.Y(Y));
initial begin
      A =1'b0;B= 1'b0;
      #10 A =1'b0;B= 1'b1;
      #10 A =1'b1;B= 1'b0;
      #10 A =1'b1;B= 1'b1;
      #10
         $finish;
   end
always @(Y)
   $display( "time =%0t \tINPUT VALUES: \t A=%b B =%b \t output value Y  =%b",$time,A,B,Y);
endmodule
```

Wave form:



NAND Gate - Block diagram:



OR gate - Truth Table

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

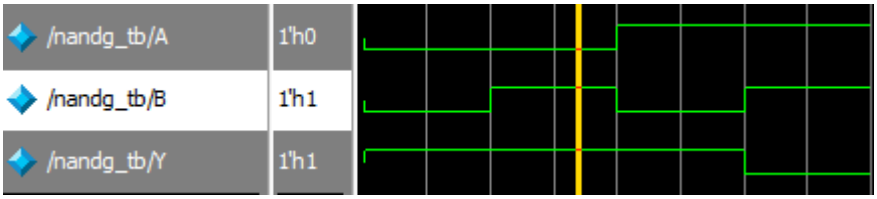
Verilog program for nand gate:

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

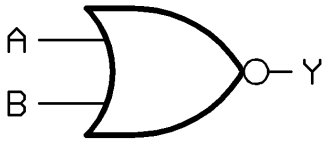
Verilog testbench program for nand gate:

```
module nandg_tb;
reg A, B;
wire Y;
nandg nandgate(.A(A), .B(B),.Y(Y));
initial begin
        A =1'b0;B= 1'b0;
        #10 A =1'b0;B= 1'b1;
        #10 A =1'b1;B= 1'b0;
        #10 A =1'b1;B= 1'b1;
        #10 $finish;
    end
always @(Y)
    $display( "time =%0t \tINPUT VALUES: \t A=%b B =%b \t output value Y  =%b", $time,A,B,Y);
endmodule
```

Wave Form:



NOR Gate - Block diagram:



NOR gate - Truth Table

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

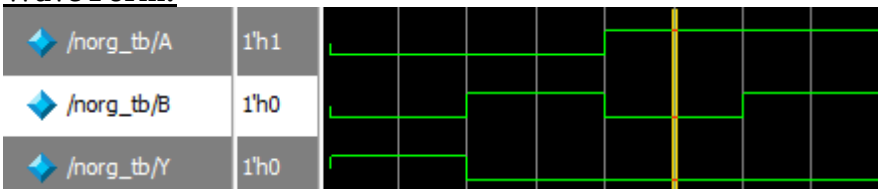
Verilog program for XOR gate:

```
module norg (A, B, Y);
        input A, B;
        output Y;
        assign Y = ~(A | B);
endmodule
```

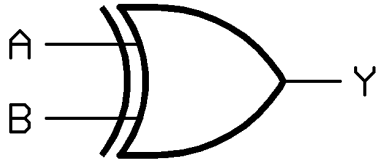
Verilog teshbench program for NOR gate:

```
module norg_tb;
reg A, B;
wire Y;
norg norgate(.A(A), .B(B),.Y(Y));
initial begin
      A =1'b0;B= 1'b0;
      #10 A =1'b0;B= 1'b1;
      #10 A =1'b1;B= 1'b0;
      #10 A =1'b1;B= 1'b1;
      #10
         $finish;
   end
always @(Y)
   $display( "time =%0t \tINPUT VALUES: \t A=%b B =%b \t output value Y  =%b",$time,A,B,Y);
endmodule
```

Wave Form:



XOR Gate - Block diagram:



NOR gate - Truth Table

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

Verilog program for XOR gate:

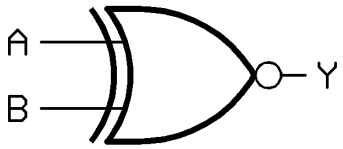
```
module xorg_dataflow (A, B, Y);
input A, B;
output Y;
assign Y =  A ^ B ;
endmodule
```

Verilog teshbench program for XOR gate:

```
module xorg_tb;
reg A, B;
wire Y;
xorg xorgate(.A(A), .B(B),.Y(Y));
initial begin
       A =1'b0;B= 1'b0;
       #10 A =1'b0;B= 1'b1;
       #10 A =1'b1;B= 1'b0;
       #10 A =1'b1;B= 1'b1;
       #10
          $finish;
   end
always @(Y)
   $display( "time =%0t \tINPUT VALUES: \t A=%b B =%b \t output value Y  =%b",$time,A,B,Y);
endmodule
```

Wave Form:

XNOR Gate - Block diagram:



NOR gate - Truth Table

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

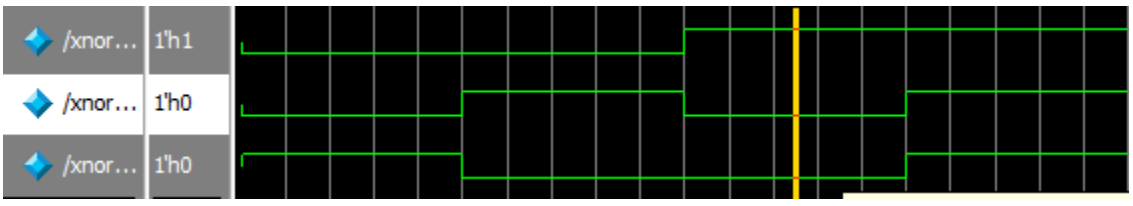
Verilog program for XNOR gate:

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

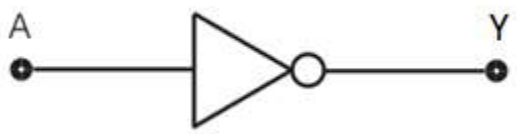
Verilog testbench program for XNOR gate:

```
module xnorg_tb;
reg A, B;
wire Y;
xnorg xnorgate(.A(A), .B(B),.Y(Y));
initial begin
        A =1'b0;B= 1'b0;
        #10 A =1'b0;B= 1'b1;
        #10 A =1'b1;B= 1'b0;
        #10 A =1'b1;B= 1'b1;
        #10
            $finish;
    end
always @(Y)
    $display( "time =%0t \tINPUT VALUES: \t A=%b B =%b \t output value Y  =%b",$time,A,B,Y);
endmodule
```

Wave Form:



NOT Gate - Block diagram:



NOT gate - Truth Table

A	Y
0	1
1	0

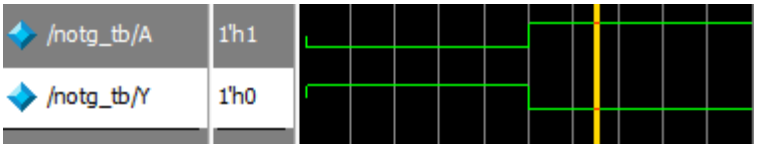
Verilog program for NOT gate:

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

Verilog testbench program for NOT gate:

```
module notg_tb;
reg A;
wire Y;
notg norgate(.A(A),.Y(Y));
initial begin
       A =1'b0;
       #10 A =1'b1;
       #10
          $finish;
   end
always @(Y)
   $display( "time =%0t \tINPUT VALUES: \t A=%b   output value Y  =%b",$time,A,Y);
endmodule
```

Wave Form:



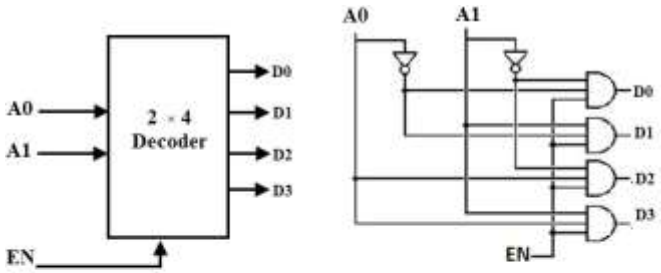
2. Design of 2-to-4 decoder usingVerilog HDL

Aim: To design the 2x4 decoder using Verilog and simulate the design

Apparatus required: Electronics Design Automation Tools used

- Xilinx Spartan 3 FPGA
- Xilinx ISE Simulator tool
- Xilinx XST Synthesis tool
- Xilinx Project Navigator 8.1i
- JTAG cable
- Adaptor 5v/4A

Block diagram:



Inputs			Outputs			
EN	A1	A0	D3	D2	D1	D0
0	×	×	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

THEORY:

As a decoder is a combinational circuit takes an n-bit binary number and produces an output on one of 2n output lines.

In a 2-to-4 binary decoder, two inputs are decoded into four outputs hence it consists of two input lines and 4 output lines. Only one output is active at any time while the other outputs are maintained at logic 0 and the output which is held active or high is determined the two binary inputs A1 and A0. The figure below shows the truth table for a 2-to-4 decoder. For a given input, the outputs D0 through D3 are active high if enable input EN is active high (EN = 1). When both inputs A1 and A1 are low (or A1= A0= 0), the output D0 will be active or High and all other outputs will be low.

When A1 = 0 and A0 = 1, the output D1 will be active and when A1 = 1 and A0 = 0, then the output D2 will be active. When both the inputs are high, then the output D3 will be high. If the enable bit is zero then all the outputs will be set to zero. This relationship between the inputs and outputs are illustrated in below truth table clearly.

a) Verilog coding for 2 to 4 decoder using data_flow model

```
module decoder_2_to_4_df(EN, A0, A1, D0, D1, D2, D3);
input EN, A0, A1;
output D0, D1, D2, D3;
        assign D0 =(EN & ~A1 & ~A0);
        assign D1 =(EN & ~A1 & A0);
        assign D2 =(EN & A1 & ~A0);
        assign D3 =(EN & A1 & A0);
endmodule
```

b) Verilog code 2 to 4 decoder using case statement (behavior level)

```
module decoder_2_to_4_beh(EN, A0, A1, D0, D1, D2, D3);
        input EN, A0, A1;
        output D0, D1, D2, D3;
        reg D3, D2, D1, D0;
    always @(A0 or A1 or EN) begin
        if (EN == 1'b1)
                case ( {A1, A0} )
                        2'b00: { D3, D2, D1, D0} = 4'b0001;
                        2'b01: { D3, D2, D1, D0} = 4'b0010;
                        2'b10: { D3, D2, D1, D0} = 4'b0100;
                        2'b11: { D3, D2, D1, D0} = 4'b1000;
                        default: {{ D3, D2, D1, D0} = 4'bxxxx;
                endcase
        if (EN == 0)
        { D3, D2, D1, D0} = 4'b0000;
    end
    endmodule
```

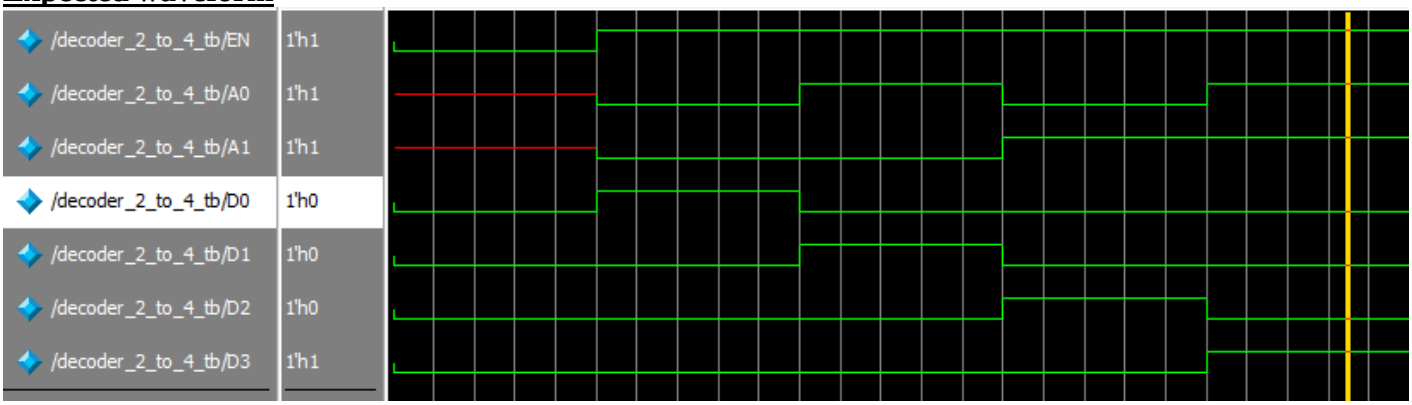
c) Verilog code for 2 to 4 decoder using structural level

```
module decoder_2_to_4(EN, A0, A1, D0, D1, D2, D3);
                input EN, A0, A1;
                output D0, D1, D2, D3;
                wire x,y;
                not u1(y,A1);
                not u2(x,A0);
                and u3(D0,EN,y,x);
                and u4(D1,EN,A0,y);
                and u5(D2,EN,x,A1);
                and u6(D3,EN,A0,A1);
  endmodule
```

Verilog testbench code for 2 to 4 decoder

```
module decoder_2_to_4_tb;
reg EN, A0, A1;
wire  D0, D1, D2, D3;
decoder_2_to_4 decoder(.EN(EN), .A0(A0), .A1(A1), .D0(D0), .D1(D1), .D2(D2), .D3(D3));
initial begin
EN=1'b0; A1 =1'bX ; A0 =1'bX;
#10 EN=1'b1; A1 =1'b0 ; A0 =1'b0;
#10 EN=1'b1; A1 =1'b0 ; A0 =1'b1;
#10 EN=1'b1; A1 =1'b1 ; A0 =1'b0;
#10 EN=1'b1; A1 =1'b1 ; A0 =1'b1;
#10$stop;
end
always @(D0, D1, D2, D3)
   $display( "time =%0t \tINPUT VALUES: \t EN=%b \tA1 =%b \t A0 =%b \t output value D3 D2 D1 D0  =
%b%b%b%b",$time,EN,A1, A0, D3,D2, D1, D0);
endmodule
```

Expected waveform



3. Design of 8-to-3 encoder (without and with parity) usingVerilog HDL

Aim To design the 8x3 encoder using Verilog and simulate the design

Apparatus required:- Electronics Design Automation Tools used:-

- Xilinx Spartan 3 FPGA
- Xilinx ISE Simulator tool
- Xilinx XST Synthesis tool
- Xilinx Project Navigator 8.1i
- JTAG cable
- Adaptor 5v/4A

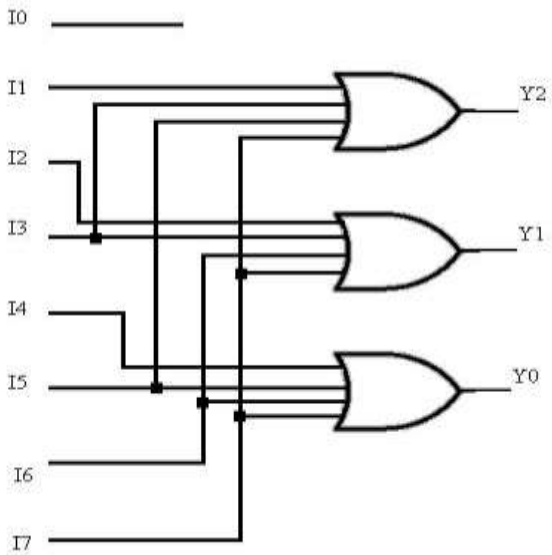
THEORY:

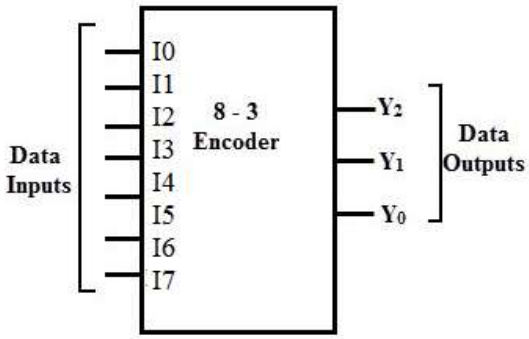
An encoder is a combinational logic circuit that essentially performs a "reverse" of decoder functions. An encoder has 2^N input lines and N output lines. In encoder the output lines generate the binary code corresponding to input value. An encoder accepts an active level on one of its inputs, representing digit, such as a decimal or octal digits, and converts it to a coded output such as BCD or binary. Encoders can also be devised to encode various symbols and alphabetic characters. The process of converting from familiar symbols or numbers to a coded format is called ***encoding***. An encoder has a number of input lines, only one of which input is activated at a given time and produces an N-bit output code, depending on which input is activated.

For an 8-to-3 binary encoder with inputs I0-I7 the logic expressions of the outputs Y0-Y2 are:

Y0 = I1 + I3 + I5 + I7
Y1= I2 + I3 + I6 + I7
Y2 = I4 + I5 + I6 +I7

Circuit Diagram, Block diagram and Truth Table:





I0	I1	I2	I3	I4	I5	I6	I7	Y2	Y1	Y0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Verilog program for 8x3 encoder structural:

```
module encoder_8_to_3(input [7:0] I,output reg [2:0] Y );
                or(Y[2],I[4],I[5],I[6],I[7]);
                or(Y[1],I[2],I[3],I[6],I[7]);
                or(Y[0],I[1],I[3],I[5],I[7]);
endmodule
```

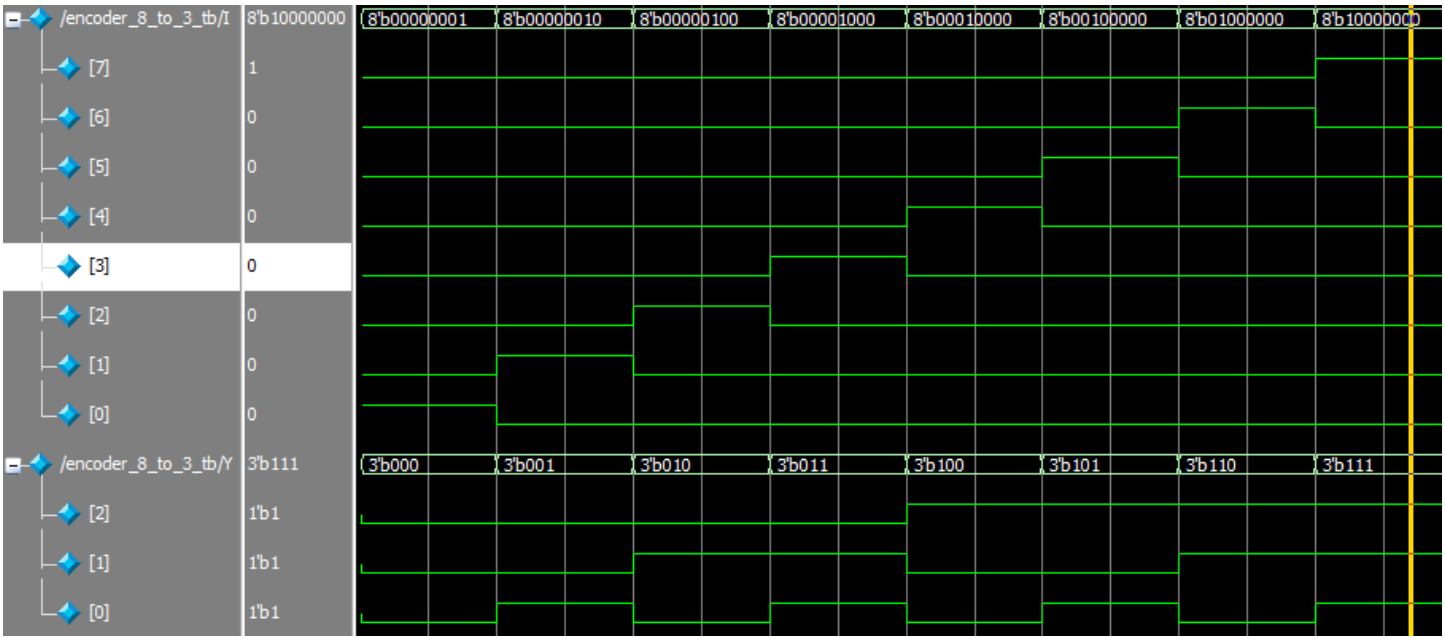
Verilog program for 8x3 encoder behavioral:

```
module encoder_8_to_3(input [7:0] I,output reg [2:0] Y );
always@(*)
        begin
        case(I)
                8'b00000001: Y<= 3'b000;
                8'b00000010: Y <= 3'b001;
                8'b00000100: Y <= 3'b010;
                8'b00001000: Y <= 3'b011;
                8'b00010000: Y <= 3'b100;
                8'b00100000: Y <= 3'b101;
                8'b01000000: Y <= 3'b110;
                8'b10000000: Y <= 3'b111;
                default: Y<= 3'bxxx;
endcase
end
endmodule
```

Verilog testbench program for 8x3 encoder behavioral:

```
module encoder_8_to_3_tb;
                reg [7:0] I;
                wire  [2:0] Y;
encoder_8_to_3 encoder(.I(I),.Y(Y));
initial begin
        I= 8'b00000001;
        #10 I=8'b00000010;
        #10 I=8'b00000100;
        #10 I=8'b00001000;
        #10 I=8'b00010000;
        #10 I=8'b00100000;
        #10 I=8'b01000000;
        #10 I=8'b10000000;
        #10$stop;
        end
always @(Y)
   $display("time =%0t \tINPUT VALUES: \t I=%b  \t output value Y  = %b ",$time,I, Y);
endmodule
```

Expected Wave form:



4. Design of 8-to-1 multiplexer/Demultiplexer

Aim: To design 8x1 multiplexer/ Demultiplexer using verilog and simulate the design

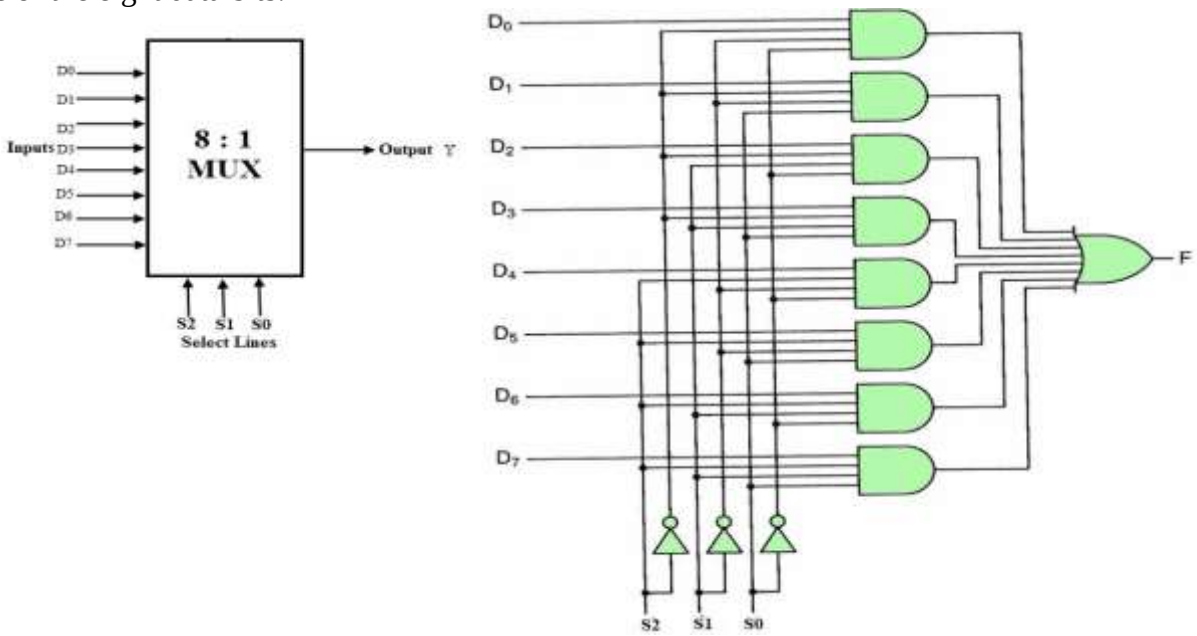
Apparatus required:- Electronics Design Automation Tools used:-

- Xilinx Spartan 3 FPGA
- Xilinx ISE Simulator tool
- Xilinx XST Synthesis tool
- Xilinx Project Navigator 8.1i
- JTAG cable
- Adaptor 5v/4A

Theory:

An 8-to-1 multiplexer consists of eight data inputs D0 through D7, three input select lines S2 through S0 and a single output line Y. Depending on the select lines combinations, multiplexer decodes the inputs.

The below figure shows the block diagram of an 8-to-1 multiplexer with enable input that enable or disable the multiplexer. Since the number data bits given to the MUX are eight then 3 bits (23=8) are needed to select one of the eight data bits.



The truth table for an 8-to1 multiplexer is given below with eight combinations of inputs so as to generate each output corresponds to input.

Select Data Inputs			Output
S_2	S_1	S_0	Y
0	0	0	D_0
0	0	1	D_1
0	1	0	D_2
0	1	1	D_3
1	0	0	D_4
1	0	1	D_5
1	1	0	D_6
1	1	1	D_7

For example, if S2= 0, S1=1 and S0=0 then the data output Y is equal to D2. Similarly the data outputs D0 to D7 will be selected through the combinations of S2, S1 and S0 as shown in below figure.

From the above truth table, the Boolean equation for the output is given as

$$Y = D0\,\overline{S2}\,\overline{S1}\,\overline{S0} + D1\,\overline{S2}\,\overline{S1}\,S0 + D2\,\overline{S2}\,S1\,\overline{S0} + D3\,\overline{S2}\,S1\,S0 + D4\,S2\,\overline{S1}\,\overline{S0} + D5\,S2\,\overline{S1}\,S0 + D6\,S2\,S1\,\overline{S0} + D7\,S2\,S1\,S0$$

From the above Boolean equation, the logic circuit diagram of an 8-to-1 multiplexer can be implemented by using 8 AND gates, 1 OR gate and 7 NOT gates as shown in below figure. In the circuit, when enable pin is set to one, the multiplexer will be disabled and if it is zero then select lines will select the corresponding data input to pass through the output.

Verilog code for 8 to 1 mux using if else statement:

```
module mux8_1(input D0,D1,D2,D3,D4,D5,D6,D7, input [2:0]S,output reg Y);
always @( * )
begin
  if (S==3'b000) Y=D0;
        else if (S==3'b001) Y=D1;
        else if (S==3'b010) Y=D2;
        else if (S==3'b011) Y=D3;
        else if (S==3'b100) Y=D4;
        else if (S==3'b101) Y=D5;
        else if (S==3'b110) Y=D6;
        else if    (S==3'b111) Y=D7;
        else Y=0;
end
endmodule
```

Verilog code for 8 to 1 mux using Case statement:

```
module mux8_1_case(input D0,D1,D2,D3,D4,D5,D6,D7, input [2:0]S,output reg Y);
  always @ (*)
   begin
    case (S)
      3'b000 : Y = D0;
         3'b001 : Y = D1;
         3'b010 : Y = D2;
         3'b011 : Y = D3;
         3'b100 : Y = D4;
         3'b101 : Y = D5;
         3'b110 : Y = D6;
         3'b111 : Y = D7;
         default : Y = 1'b0;
    endcase
   end
endmodule
```

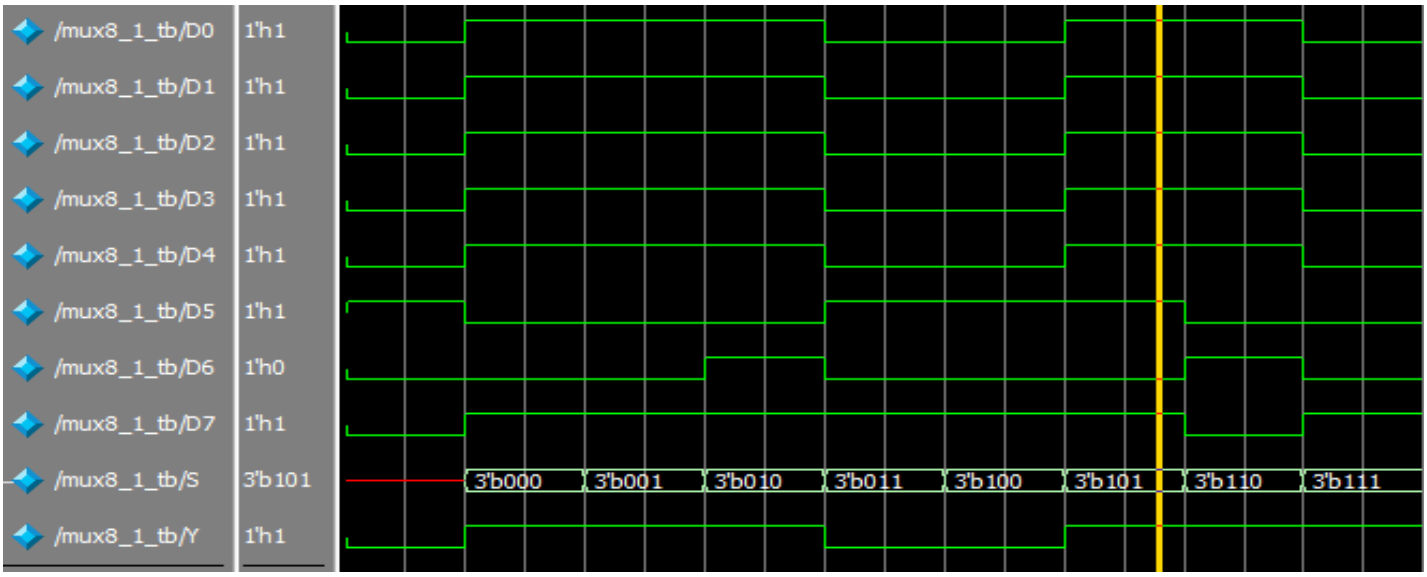
Verilog code for 8 to 1 mux – Gate level realization:

```
module mux8_1_gate(input D0,D1,D2,D3,D4,D5,D6,D7, input [2:0]S,output Y);
wire inv0, inv1, inv2; // Inverter outputs.
wire a0, a1, a2, a3, a4, a5, a6, a7 ; // AND gates outputs.
not not_0 (inv0, S[0]); //// Inverters.
not not_1 (inv1, S[1]);
not not_2 (inv2, S[2]);
and and_0 (a0, inv2, inv1, inv0,D0); //// 3-input AND gates.
and and_1 (a1, inv2, inv1, S[0],D1);
and and_2 (a2, inv2, S[1], inv0,D2);
and and_3 (a3, inv2, S[1], S[0],D3);
and and_4 (a4, S[2], inv1, inv0,D4);
and and_5 (a5, S[2], inv1, S[0],D5);
and and_6 (a6, S[2], S[1], inv0,D6);
and and_7 (a7, S[2], S[1], S[0],D7);
or or_0(Y, a0, a1, a2, a3, a4, a5, a6, a7); //// 8-input OR gate.
endmodule
```

Verilog testbench code for 8 to 1 mux:

```
module mux8_1_tb;
reg   D0,D1,D2,D3,D4,D5,D6,D7;
reg  [2:0] S;
wire  Y;
mux8_1 mux(.D0(D0),.D1(D1),.D2(D2),.D3(D3),.D4(D4),.D5(D5),.D6(D6),.D7(D7), .S(S),.Y(Y));
initial begin
        {D0,D1,D2,D3,D4,D5,D6,D7}=$random%8;
         #10  S= 000;
        {D0,D1,D2,D3,D4,D5,D6,D7}=$random%8;
        #10 S= 001;
        {D0,D1,D2,D3,D4,D5,D6,D7}=$random%8;
        #10 S= 010;
        {D0,D1,D2,D3,D4,D5,D6,D7}=$random%8;
        #10 S= 011;
        {D0,D1,D2,D3,D4,D5,D6,D7}=$random%8;
        #10 S= 100;
        {D0,D1,D2,D3,D4,D5,D6,D7}=$random%8;
        #10 S= 101;
        {D0,D1,D2,D3,D4,D5,D6,D7}=$random%8;
        #10 S= 110;
        {D0,D1,D2,D3,D4,D5,D6,D7}=$random%8;
        #10 S= 111;
        {D0,D1,D2,D3,D4,D5,D6,D7}=$random%8;  #10$stop;
end
always @(Y)
   $display("time =%0t \tINPUT VALUES: \t D0 D1 D2 D3 D4 D5 D6 D7=%b%b%b%b%b%b%b%b \t S =
%b  \t output value Y  = %b ",$time,D0,D1,D2,D3,D4,D5,D6,D7,S, Y);
endmodule
```

Expected Wave form:

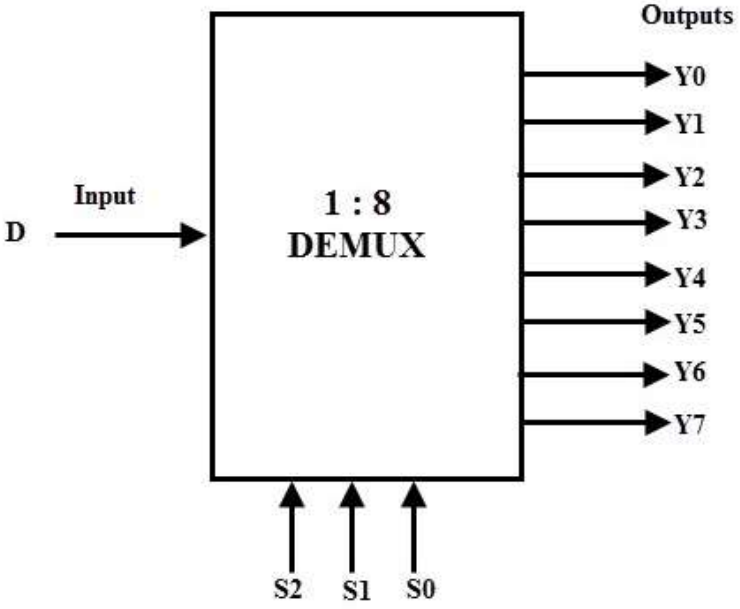


Demultiplexor

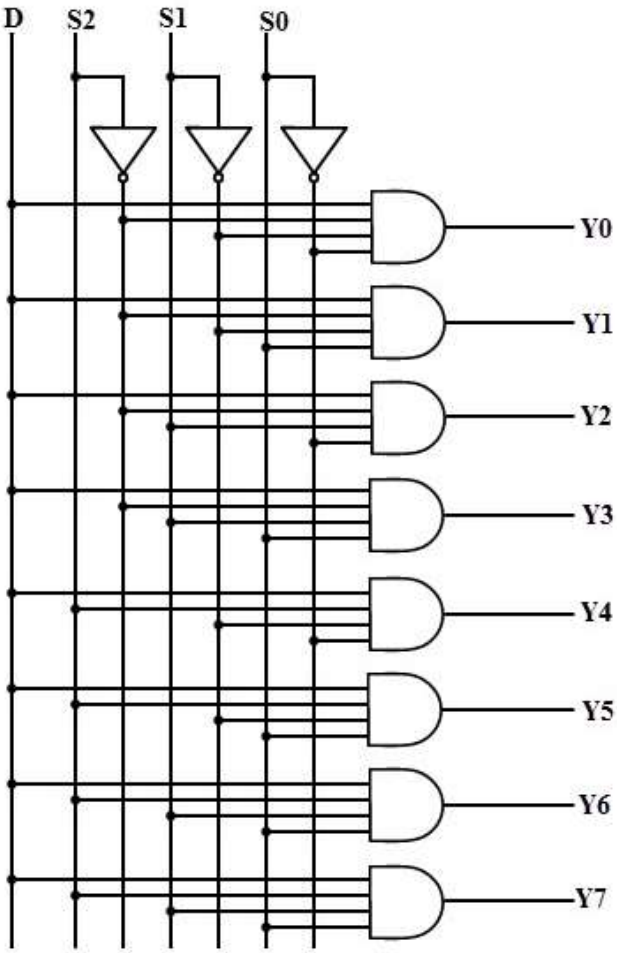
A demultiplexer is a combinational logic circuit that receives the information on a single input and transmits the same information over one of 2n possible output lines.. The action or operation of a demultiplexer is opposite to that of the multiplexer. As inverse to the MUX , demux is a one-to-many circuit. With the use of a demultiplexer , the binary data can be bypassed to one of its many output data lines. Demultiplexers are mainly used in Boolean function generators and decoder circuits.

The below figure shows the block diagram of a 1-to-8 demultiplexer that consists of single input D, three select inputs S2, S1 and S0 and eight outputs from Y0 to Y7.

It is also called as 3-to-8 demultiplexer due to three select input lines. It distributes one input line to one of 8 output lines depending on the combination of select inputs.



Data Input	Select Inputs			Outputs							
D	S_2	S_1	S_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
D	0	0	0	0	0	0	0	0	0	0	D
D	0	0	1	0	0	0	0	0	0	D	0
D	0	1	0	0	0	0	0	0	D	0	0
D	0	1	1	0	0	0	0	D	0	0	0
D	1	0	0	0	0	0	D	0	0	0	0
D	1	0	1	0	0	D	0	0	0	0	0
D	1	1	0	0	D	0	0	0	0	0	0
D	1	1	1	D	0	0	0	0	0	0	0



The truth table for this type of demultiplexer is shown below. The input D is connected with one of the eight outputs from Y0 to Y7 based on the select lines S2, S1 and S0.

For example, if S2S1S0=000, then the input D is connected to the output Y0 and so on.

From this truth table, the Boolean expressions for all the outputs can be written as follows.

$Y0 = D\,\overline{S2}\,\overline{S1}\,\overline{S0}$

$Y1 = D\,\overline{S2}\,\overline{S1}\,S0$

$Y2 = D\,\overline{S2}\,S1\,\overline{S0}$

$Y3 = D\,\overline{S2}\,S1\,S0$

$Y4 = D\,S2\,\overline{S1}\,\overline{S0}$

$Y5 = D\,S2\,\overline{S1}\,S0$

$Y6 = D\,S2\,S1\,\overline{S0}$

$Y7 = D\,S2\,S1\,S0$

From these obtained equations, the logic diagram of this demultiplexer can be implemented by using eight AND gates and three NOT gates as shown in below figure. The different combinations of the select lines , select one AND gate at given time , such that data input will appear at a particular output.

Applications of Demultiplexer

Since the demultiplexers are used to select or enable the one signal out of many, these are extensively used in microprocessor or computer control systems such as

- Selecting different IO devices for data transfer
- Choosing different banks of memory
- Depends on the address, enabling different rows of memory chips

Enabling different functional units.

Other than these, demultiplexers can be found in a wide variety of application such as

Synchronous data transmission systems

- Boolean function implementation (as we discussed full subtractor function above)
- Data acquisition systems
- Combinational circuit design
- Automatic test equipment systems
- Security monitoring systems (for selecting a particular surveillance camera at a time), etc.

Verilog code for 1 to 8 Demux – Gate level realization:

```
module demux_8_1_gate(input D, input [2:0]S,output Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7);
    wire inv0, inv1, inv2; // Inverter outputs.
    // Inverters.
    not not_0 (inv0, S[0]);
    not not_1 (inv1, S[1]);
    not not_2 (inv2, S[2]);
     // 3-input AND gates.
    and and_0 (Y0, inv2, inv1, inv0, D );
    and and_1 (Y1, inv2, inv1, S[0], D );
    and and_2 (Y2, inv2, S[1], inv0, D );
    and and_3 (Y3, inv2, S[1], S[0], D );
    and and_4 (Y4, S[2], inv1, inv0, D );
    and and_5 (Y5, S[2], inv1, S[0], D );
    and and_6 (Y6, S[2], S[1], inv0, D );
    and and_7 (Y7, S[2], S[1], S[0], D );
endmodule
```

Verilog code for 1 to 8 Demux – using assign statement:

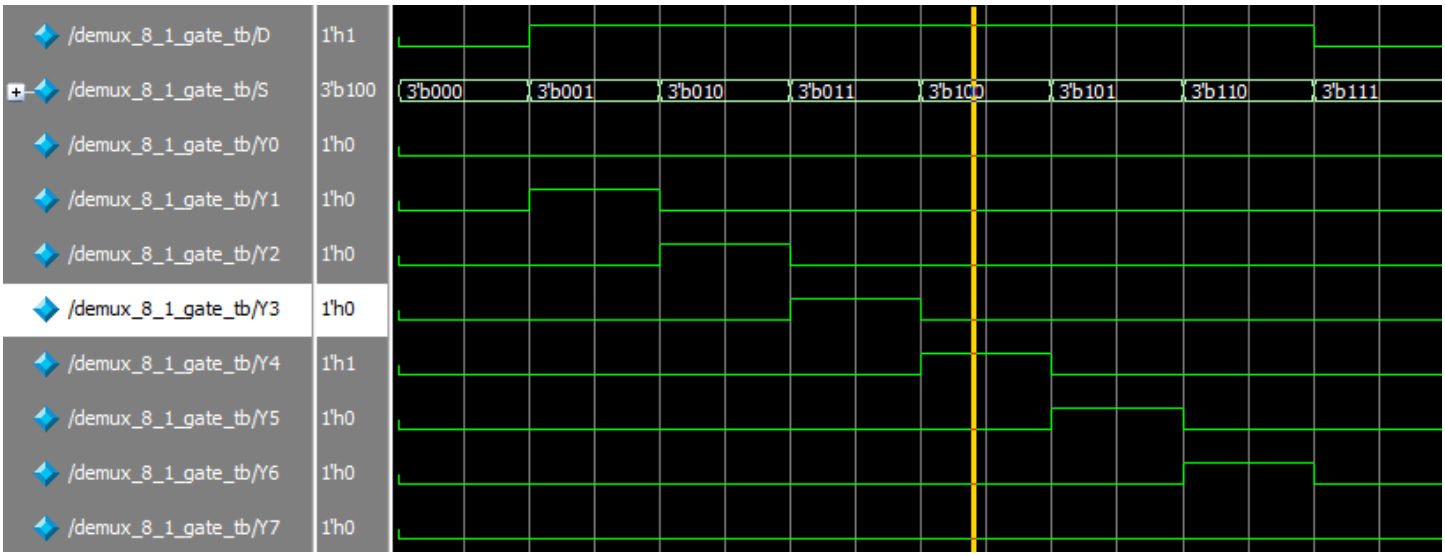
```
module demux_8_1(D, S, Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7);
  input       D;
  input   [2:0] S;
  output  Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7;
  assign {Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7} = D << S;
endmodule
```

Verilog testbench for 1 to 8 Demux – using assign statement:

```
module demux_8_1_gate_tb;
reg   D;
reg  [2:0] S;
wire  Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7;
demux_8_1_gate
demux(.D(D),.S(S),.Y0(Y0),.Y1(Y1),.Y2(Y2),.Y3(Y3),.Y4(Y4),.Y5(Y5),.Y6(Y6),.Y7(Y7));
initial begin
D=$random;
  S= 000;
#10 S= 001;
D=$random;
#10 S= 010;
D=$random;
#10 S= 011;
D=$random;
#10 S= 100;
D=$random;
#10 S= 101;
D=$random;
#10 S= 110;
D=$random;
#10 S= 111;
```

```
D=$random;
#10$stop;
end
always @(*)
   $display("time =%0t \tINPUT VALUES: \t D =%b \t S = %b  \t output value Y0 Y1 Y2 Y3 Y4 Y5 Y6
Y7  = %b%b%b%b%b%b%b%b ",$time,D,S,Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7);
endmodule
```

Expected Waveform



5. Design of 4bit binary to gray converter/ Gray to Binary Code Converter

Aim Design of 4 bit gray to binary converter using Verilog and simulate the design

Apparatus required:- Electronics Design Automation Tools used:-

- Xilinx Spartan 3 FPGA
- Xilinx ISE Simulator tool
- Xilinx XST Synthesis tool
- Xilinx Project Navigator 8.1i
- JTAG cable
- Adaptor 5v/4A

Theory: Binary to Gray converter

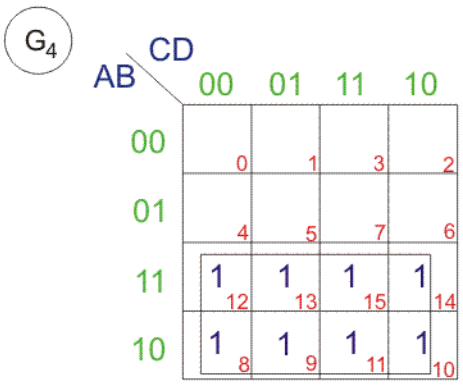
The logical circuit which converts binary code to equivalent gray code is known as **binary to gray code converter**. The gray code is a non weighted code. The successive gray code differs in one bit position only that means it is a unit distance code. It is also referred as cyclic code. It is not suitable for arithmetic operations. It is the most popular of the unit distance codes. It is also a reflective code. An n-bit Gray code can be obtained by reflecting an n-1 bit code about an axis after 2^{n-1} rows, and putting the MSB of 0 above the axis and the MSB of 1 below the axis. Reflection of Gray codes is shown below. The 4 bits binary to gray code conversion table is given below,

Decimal Number	4 bit Binary Number ABCD	4 bit Gray Code $G_1G_2G_3G_4$
0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 1
3	0 0 1 1	0 0 1 0
4	0 1 0 0	0 1 1 0
5	0 1 0 1	0 1 1 1
6	0 1 1 0	0 1 0 1
7	0 1 1 1	0 1 0 0
8	1 0 0 0	1 1 0 0
9	1 0 0 1	1 1 0 1
10	1 0 1 0	1 1 1 1
11	1 0 1 1	1 1 1 0
12	1 1 0 0	1 0 1 0
13	1 1 0 1	1 0 1 1
14	1 1 1 0	1 0 0 1
15	1 1 1 1	1 0 0 0

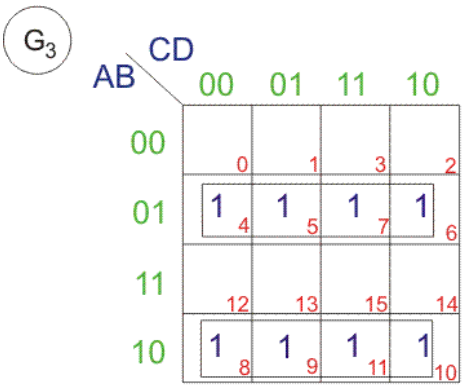
That means, in 4 bit gray code, (4-1) or 3 bit code is reflected against the axis drawn after $(2^{4-1})^{th}$ or 8^{th} row. The bits of 4 bit gray code are considered as $G_4G_3G_2G_1$. Now from conversion table,From above SOPs, let us draw K -maps for G_4, G_3, G_2 and G_1.

$$G_4 = \sum m(8,9,10,11,12,13,14,15),\ G_3 = \sum m(4,5,6,7,8,9,10,11)$$
$$G_2 = \sum m(2,3,4,5,10,11,12,13),\ G_1 = \sum m(1,2,5,6,9,10,13,14)$$

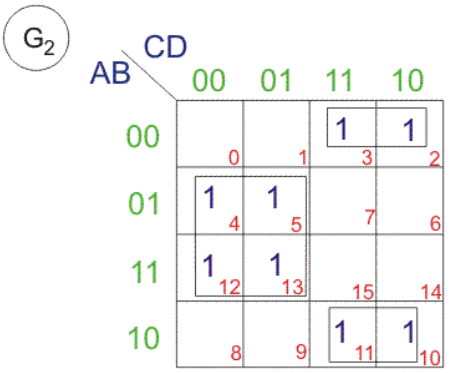
K-Map



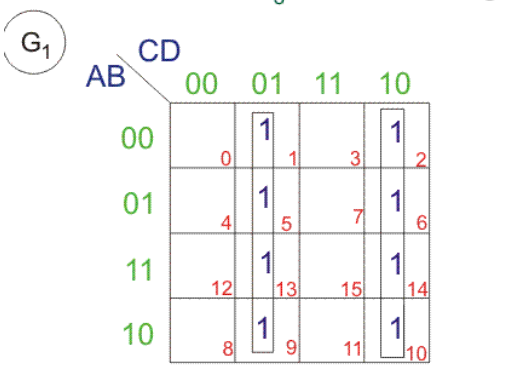
$G_4 = A$



$G_3 = \bar{A}B + A\bar{B} = A \oplus B$

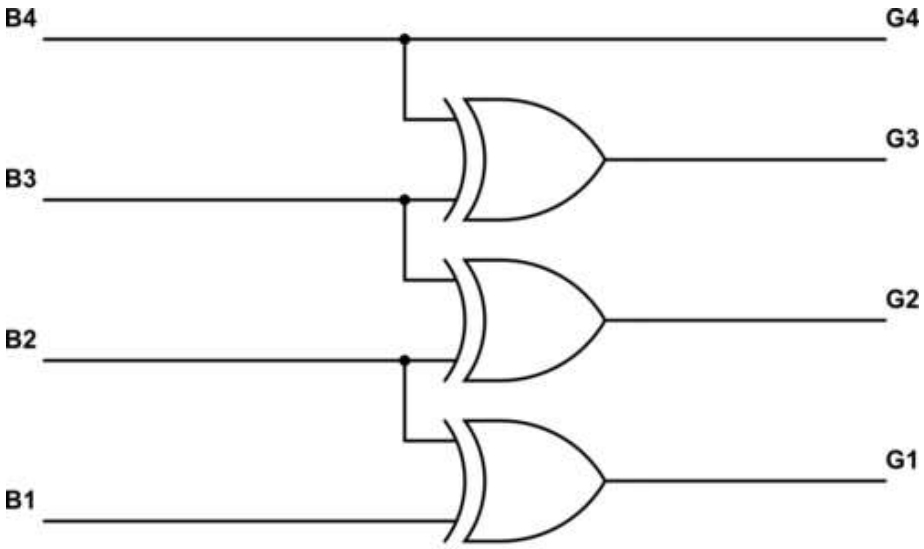


$G_2 = B\bar{C} + \bar{B}C = B \oplus C$



$G_1 = \bar{C}D + C\bar{D} = C \oplus D$

Circuit Diagram : Binary to Gray Convertor



Gray to Binary Code Converter

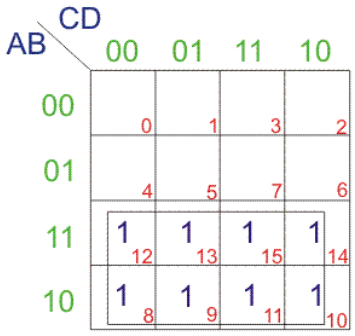
In **gray to binary code converter**, input is a multiplies gray code and output is its equivalent binary code.

Let us consider a 4 bit gray to binary code converter. To design a 4 bit gray to binary code converter, we first have to draw a conversion table. From above gray code we get,

4 bit Gray Code	4 bit Binary Code
A B C D	B_4 B_3 B_2 B_1
0 0 0 0	0 0 0 0
0 0 0 1	0 0 0 1
0 0 1 1	0 0 1 0
0 0 1 0	0 0 1 1
0 1 1 0	0 1 0 0
0 1 1 1	0 1 0 1
0 1 0 1	0 1 1 0
0 1 0 0	0 1 1 1
1 1 0 0	1 0 0 0
1 1 0 1	1 0 0 1
1 1 1 1	1 0 1 0
1 1 1 0	1 0 1 1
1 0 1 0	1 1 0 0
1 0 1 1	1 1 0 1
1 0 0 1	1 1 1 0
1 0 0 0	1 1 1 1

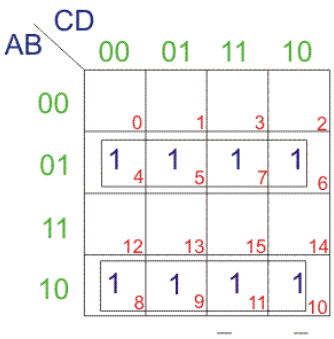
K-Map

B_4



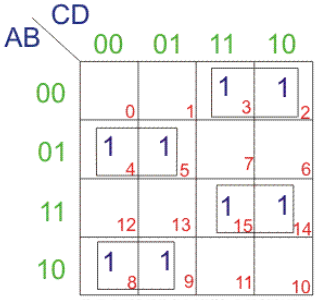
$B_4 = A$

B_3



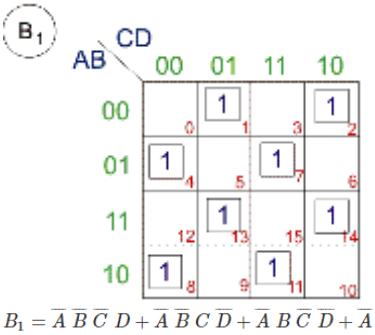
$B_3 = \overline{A}B + A\overline{B} = A \oplus B$

B_2



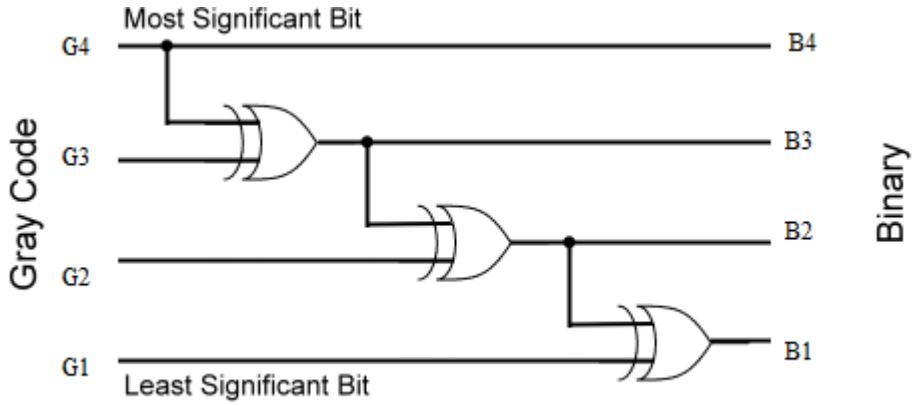
$$B_2 = \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C + ABC$$
$$= A(\overline{B}\,\overline{C} + BC) + \overline{A}(B\overline{C} + \overline{B}C)$$
$$= A(B \odot C) + \overline{A}(B \oplus C)$$
$$= A(\overline{B \oplus C}) + \overline{A}(B \oplus C) = A \oplus B \oplus C$$

B_1



$$B_1 = \overline{A}\,\overline{B}\,\overline{C}\,D + \overline{A}\,\overline{B}\,C\,\overline{D} + \overline{A}\,B\,\overline{C}\,\overline{D} + \overline{A}\,BCD + AB\overline{C}\,D + ABC\overline{D} + A\overline{B}\,\overline{C}\,\overline{D} + A\overline{B}CD = A \oplus B \oplus C \oplus D$$

Circuit diagram:

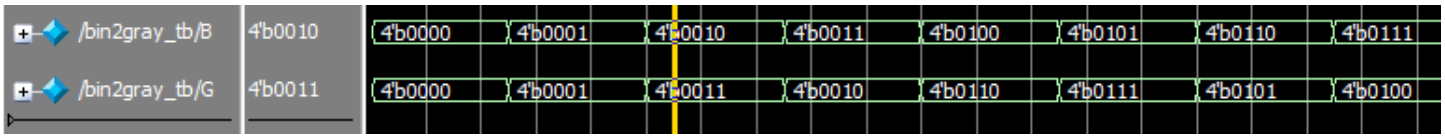


Verilog Code for Binary to Gray convertor:

```
module bin2gray ( B ,G );
        input [3:0] B ;
        wire [3:0] B ;
        output [3:0] G ;
        wire [3:0] G ;
                assign G[3] = B[3];
                assign G[2:0] = B[3:1] ^ B[2:0];
endmodule
```

Verilog Testbench for Binary to Gray convertor:

```
module bin2gray_tb;
        reg [3:0]B;
        wire [3:0]G;
        bin2gray B2G (.B(B) ,.G(G));
                always #5 B=B+1'b1;
        initial
                begin
                $monitor($time,  "\tB=%b\t ,  G=%b\t", B,G);
                B <= 4'b0000;
                #80 $finish;
                end
endmodule
```



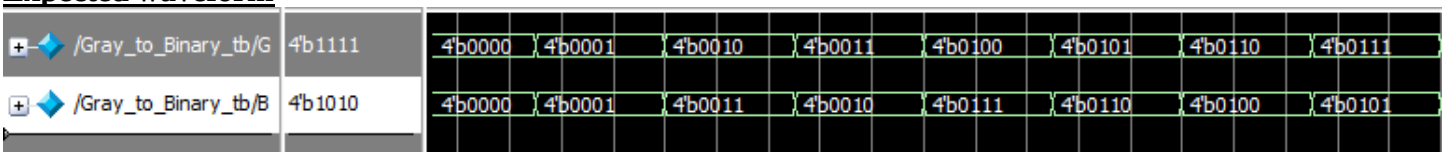
Verilog code for Gray to Binary convertor:

```
module Gray_to_Binary ( G ,B );
        output [3:0] B ;
        input [3:0] G ;
        assign B[3] = G[3];
        assign B[2] = G[3]^G[2];
        assign B[1] = G[3]^G[2]^G[1];
        assign B[0] = G[3]^G[2]^G[1]^G[0];
endmodule
```

Verilog Testbench for Gray to Binary convertor:

```
module Gray_to_Binary_tb;
        reg [3:0]G;
        wire [3:0]B;
        Gray_to_Binary G2B (.G(G) ,.B(B));
        always #5 G=G+1'b1;
        initial
        begin
        $monitor($time,  "\tG=%b\t, B=%b\t", G,B);
        G <= 4'b0000;
        #80 $finish;
        end
endmodule
```

Expected waveform



6. Design of Comparator

Aim: Design of Comparator using Verilog and simulate the design

Apparatus required:- Electronics Design Automation Tools used:-

- Xilinx Spartan 3 FPGA
- Xilinx ISE Simulator tool
- Xilinx XST Synthesis tool
- Xilinx Project Navigator 8.1i
- JTAG cable
- Adaptor 5v/4A

Theory

A magnitude comparator is a combinational circuit that compares two numbers A & B to determine whether:

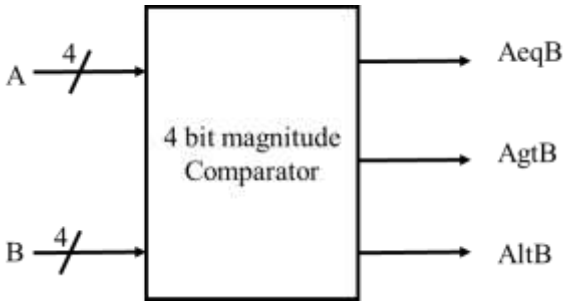
A > B, or A = B, or A < B

Inputs: First *4*-bit number A

Second *4*-bit number B

Outputs:3 output signals (GT, EQ, LT), where:

1. AgtB= 1 IFF A > B
2. AeqB = 1 IFF A = B
3. AltB = 1 IFF A < B



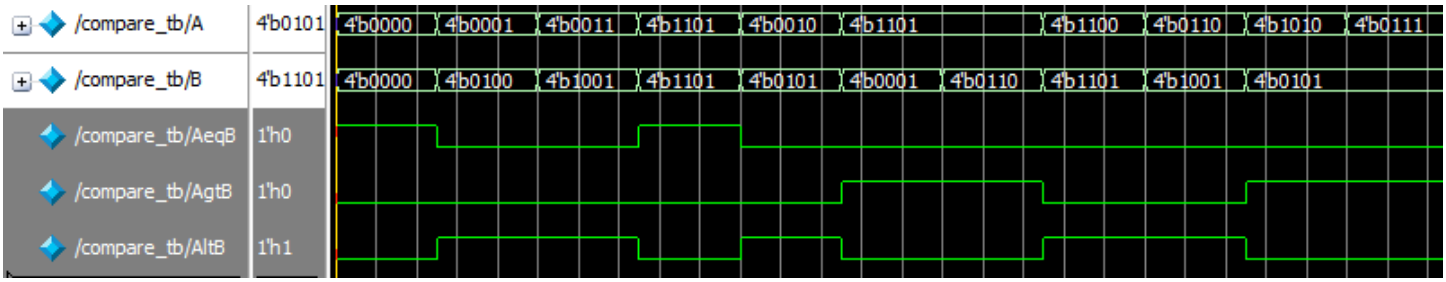
Verilog code for 4 bit magnitude comparator

```
module compare (A, B, AeqB, AgtB, AltB);
    input [3:0] A, B;
    output AeqB, AgtB, AltB;
    reg AeqB, AgtB, AltB;
        always @(A or B)
        begin
        AeqB = 0;
        AgtB = 0;
        AltB = 0;
        if(A == B)
                AeqB = 1;
        else if (A > B)
                AgtB = 1;
        else
                AltB = 1;
    end
endmodule
```

Verilog testbench for 4 bit magnitude comparator

```
module compare_tb;
        reg [3:0] A, B;
        wire AeqB, AgtB, AltB;
compare comp(A, B, AeqB, AgtB, AltB);
always #5 B=$random%16;
always #5 A=$random%16;
initial
begin
$monitor($time,  "\tA=%b\t ,  B=%b\t,  AeqB=%b\t,  AgtB=%b\t,  AltB=%b\t", A,B, AeqB, AgtB, AltB);
A=4'b0000;
B=4'b0000;
#80 $finish;
end
endmodule
```

Expected Waveform:



7. Design of Full adder using 3 modeling styles

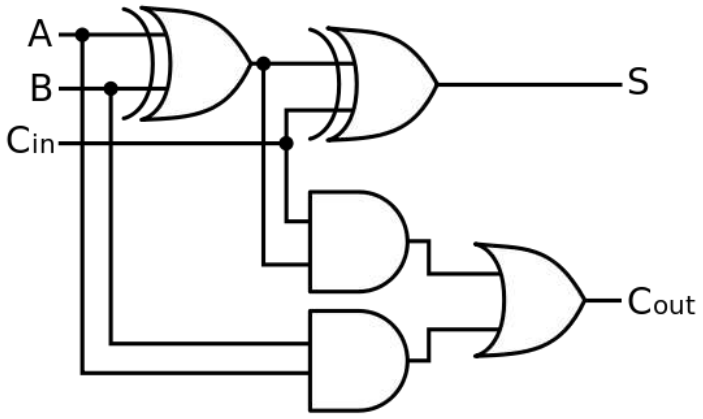
Aim: Design of Full adder using 3 modeling styles using Verilog and simulates the design

Apparatus required:- Electronics Design Automation Tools used:-

- Xilinx Spartan 3 FPGA
- Xilinx ISE Simulator tool
- Xilinx XST Synthesis tool
- Xilinx Project Navigator 8.1i
- JTAG cable
- Adaptor 5v/4A

Theory:

A **full adder** adds binary numbers and accounts for values carried in as well as out. A one-bit full adder adds three one-bit numbers, often written as *A*, *B*, and C_{in}; *A* and *B* are the operands, and C_{in} is a bit carried in from the previous less-significant stage. The full adder is usually a component in a cascade of adders, which add 8, 16, 32, etc. bit binary numbers. The circuit produces a two-bit output. Output carry and sum typically represented by the signals C_{out} and *S*



The truth table for the full adder

Inputs			Outputs	
A	***B***	C_{in}	C_{out}	***S***
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Gate level modeling

The module is implemented in terms of logic gates and interconnections between these gates. Designer should know the gate-level diagram of the design.

```
wire Z, Z1, OUT, OUT1, OUT2, IN1, IN2;

and a1(OUT1, IN1, IN2);
nand na1(OUT2, IN1, IN2);
xor x1(OUT, OUT1, OUT2);
not (Z, OUT);
buf final (Z1, Z);
```

• Essentially describes the topology of a circuit
• All instances are executed concurrently just as in hardware
• Instance name is not necessary
• The first terminal in the list of terminals is an output and the other terminals are inputs
• Not the most interesting modeling technique for our class

Data Flow Modeling

Module is designed by specifying the data flow, where the designer is aware of how data flows between hardware registers and how the data is processed in the design

• The continuous assignment is one of the main constructs used in dataflow modeling

- `assign out = i1 & i2;`
- `assign addr[15:0] = addr1[15:0] ^ addr2[15:0];`
- `assign {c_out, sum[3:0]}=a[3:0]+b[3:0]+c_in;`

• A continuous assignment is always active and the assignment expression is evaluated as soon as one of the right-hand-side variables change

• Assign statements describe hardware that operates concurrently − ordering does not matter

• Left-hand side must be a scalar or vector net. Right-hand side operands can be wires, (registers, integers, and real)

Behavioral level Modeling:

This is the highest level of abstraction. A module can be implemented in terms of the design algorithm. The designer no need to have any knowledge of hardware implementation.

Verilog program for full adder – Gate level modeling

```
module fulladder_gate_level
( input A,
 input B,
 input Cin,

 output S,
 output Cout );
        wire  p,r,s;
                xor (p,A,B);
                xor (S,p,Cin);

                and(r,p,Cin);
                and(s,A,B);
                or(Cout,r,s);
 endmodule
```

Verilog program for full adder – dataflow_modeling

```
module fulladder_data_flow
( input A,
 input B,
 input Cin,

 output   S,
 output   Cout);

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

Verilog program for full adder – Behavioral level modeling

```
module fulladder_behav
```

```
( input A,
 input B,
 input Cin,

 output S,
 output Cout );

assign {Cout,S} =  Cin + A + B;
endmodule
```

Verilog testbench program for full adder

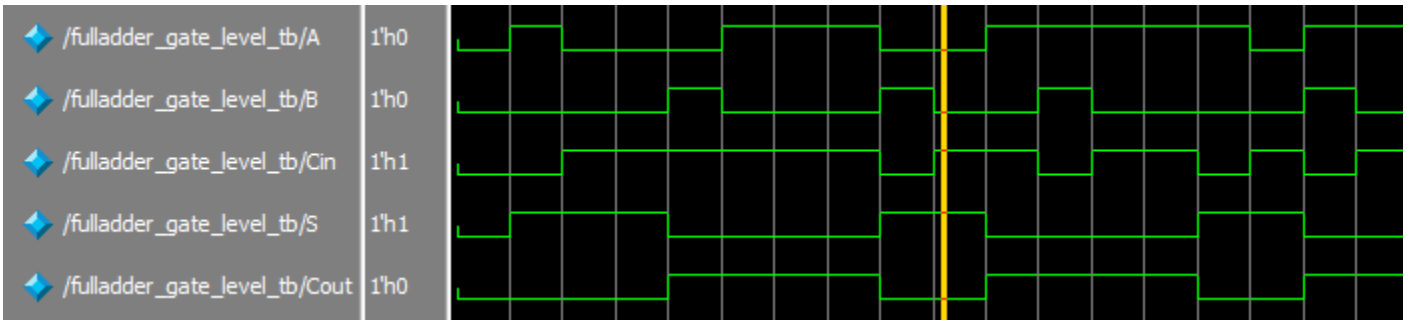
```
module fulladder_gate_level_tb;
reg A, B, Cin;
wire S, Cout;
 integer i;
fulladder_gate_level FA (.A(A),.B(B) ,.Cin(Cin), .S(S), .Cout(Cout));

initial
begin
{A,B,Cin} <= 3'b000;
for(i=1;i<20; i=i+1)
begin
$monitor($time, "\tA=%b\t   B=%b\t   Cin=%b\t   S=%b\t   Cout=%b", A, B,Cin, S,Cout);
#5{A,B,Cin} <= $random%8;
end
//always #5 {A,B,Cin}<=$random%8;

 #45 $finish;
end

endmodule
```

Expected Waveform:



8. Design of Latches and flip flops: D-latch SR, D,JK, T

Aim: Design of Latches and flip flops (D-latch SR, D,JK, T) using Verilog and simulates the design

Apparatus required: - Electronics Design Automation Tools used: -

- **Xilinx Spartan 3 FPGA**
- **Xilinx ISE Simulator tool**
- **Xilinx XST Synthesis tool**
- **Xilinx Project Navigator 8.1i**
- **JTAG cable**
- **Adaptor 5v/4A**

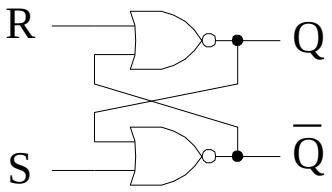
Theory:

LATCH AND FLIP-FLOP

Latch and *flip-flop* are memory devices and implemented using *bistable* circuit - its output will remain in either 0 or 1 state. The output state of a *latch* is controlled by its excitation input signals. A *flip-flop* (FF) is predominately controlled by a *clock* and its output state is determinted by its excitation input signals. *Note that if the clock is replaced by a gated control signal, the flip-flop becomes a gated latch.*

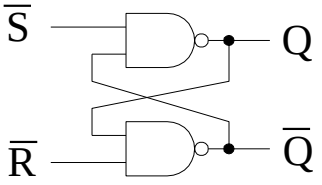
a. RS (reset-set) latch circuit

When S (set) is set to 1, the output Q will be set to 1. Likewise, when R (reset) is set to 1, the output Q will be set to 0. It is invalid to set both S and R to 1.



S	R	Q	$\overline{Q}$	
0	0	Unchanged		
0	1	0	1	
1	0	1	0	
1	1	0	0	Invalid input

NOR gate implementation



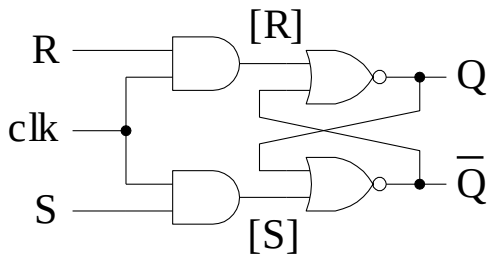
$\overline{S}$	$\overline{R}$	Q	$\overline{Q}$
0	0	Invalid input	
0	1	1	0
1	0	0	1
1	1	Unchanged	

NAND gate implementation

Note that the input is ***active high*** for NOR gate implementation, whereas the input is ***active low*** for NAND gate implementation.

b. Clocked RS FF

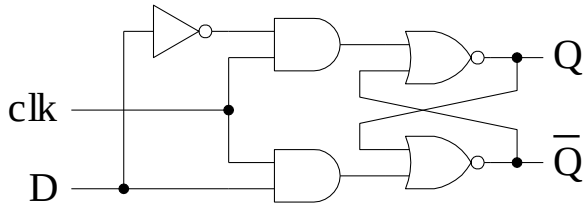
The major problem of RS latch is its susceptibility to voltage noise which could change the output states of the FF. With the clocked RS FF, the problem is remedied. With the clock held low, [S] & [R] held low, the output remains unchanged. With the clock held high, the output follows R & S. Thus the output will be latched its states when the clock goes low.



		clk= 1	clk = 0
S	R	Q_{n+1}	Q_{n+1}
0	0	Q_n	Q_n
0	1	0	Q_n
1	0	1	Q_n
1	1	Invalid i/p	Q_n

c. <u>D-type FF</u>

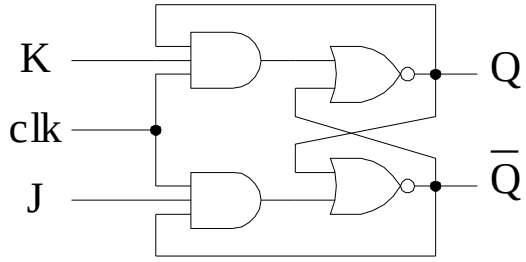
The D-type FF remedies the indeterminate state problem that exists when both inputs to a clocked RS FF are high. The schematic is identical to a RS FF except that an inverter is used to produce a pair of complementary input.



clk	D	Q_{n+1}
0	x	Q_n
1	0	0
1	1	1

d. <u>JK FF</u>

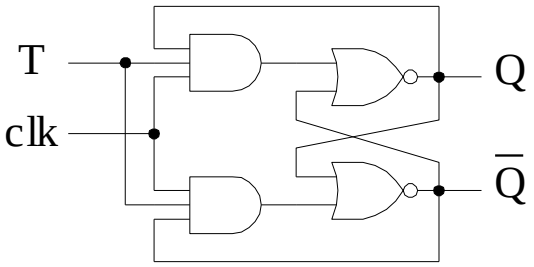
The JK FF is a refinement of the RS FF in that the undetermined state of the RS type is defined in the JK type. Inputs J and K behave like inputs S and R to set and reset (clear) the FF, respectively. The input marked J is for *set* and the input marked K is *reset*.



		clk=0	clk=1
J	K	Q_{n+1}	Q_{n+1}
0	0	Q_n	Q_n
0	1	Q_n	0
1	0	Q_n	1
1	1	Q_n	$\overline{Q}_n$

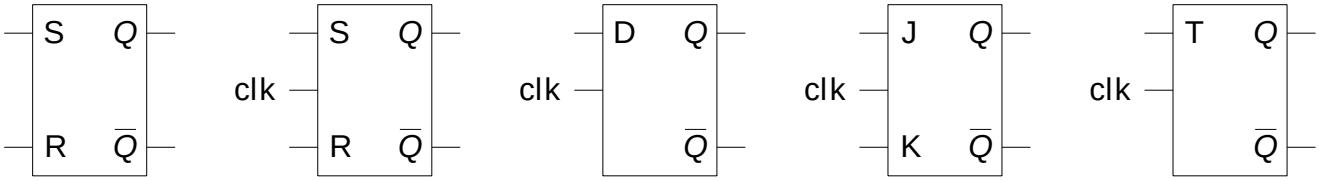
e. T-type FF

The toggle (T) FF has a clock input which causes the output state changed for each clock pulse if T is in its active state. It is useful in counter design.



clk	T	Q_{n+1}
0	X	Q_n
1	0	Q_n
1	1	$\overline{Q}_n$

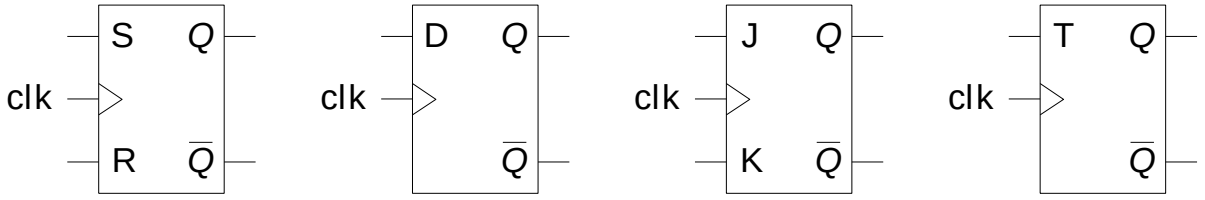
*Logic smybols of various latch and **level-triggered** flip-flops*

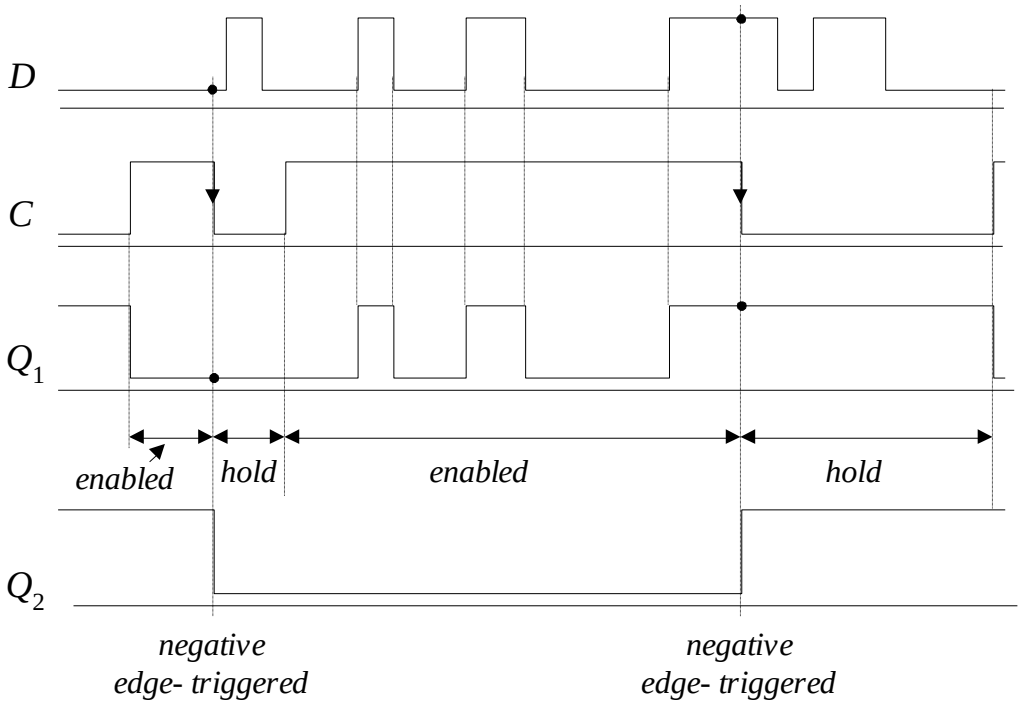


f. Edge-triggered FF

Clocked FF is a ***level-triggered*** device, its output responses to the input during the clock active period and this is referred to as the "0" and "1" catching problem. For sequential synchronous circuit, the data transfer is required to be synchronized with the clock signal. Additional circuit is included in the FF to ensure that it will only response to the input at the transition edge of the clock pulse. These type of devices are called ***edge-triggered*** FFs.

*Logic smybols of various **edge-triggered** flip-flops*

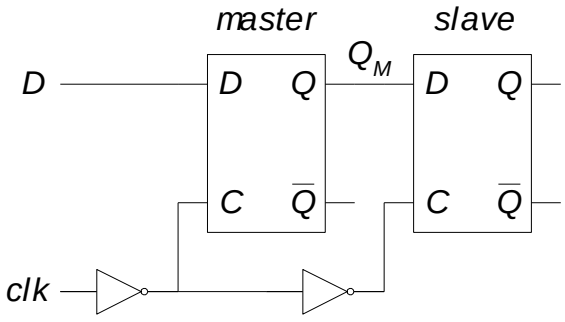




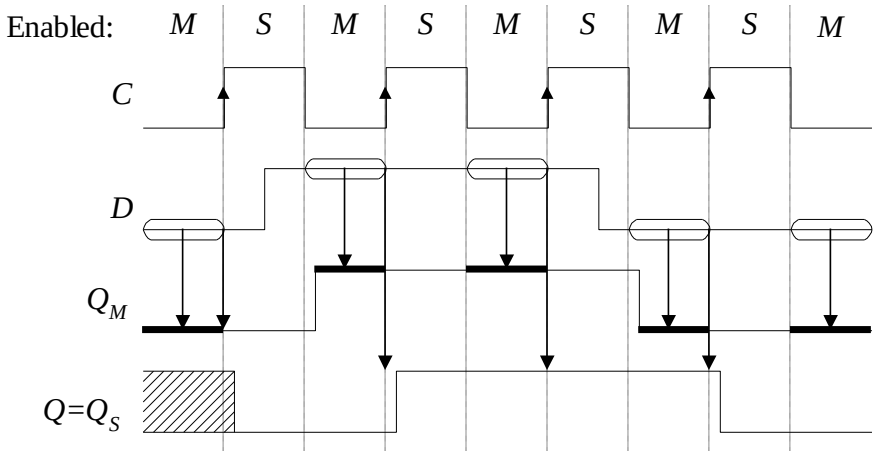
Timing diagram for a gated D latch and a negative edge-triggered D FF

g. Master-slave FF

A master-slave type FF consists of two FFs, a master stage and a slave stage. The output states responding to the inputs are transmitted to slave output through the master stage in different time slots of the clock pulse. Hence the output will not be affected by the undesirable changes at the input after the output of the master FF has been latched to the slave FF.



Master-slave D flip-flop



Timing diagram

h. FF timing parameters

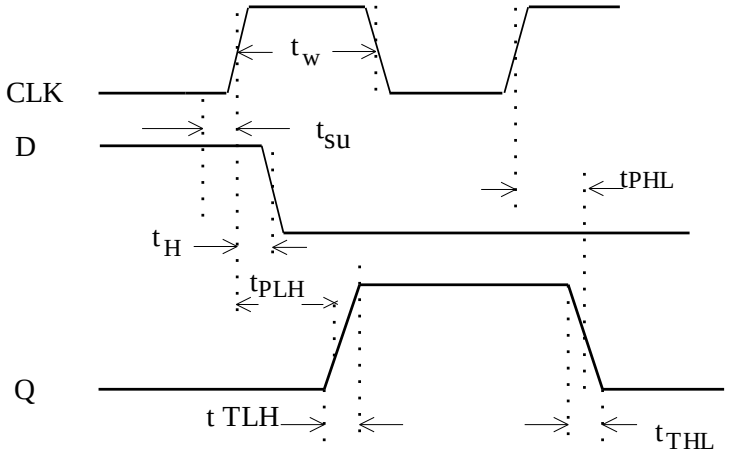
Propagation delay: propagation delay for a FF is the amount of time it takes for the output of the FF to change its state from a clock trigger or asynchronous set or rest. It is defined from the 50% point of the input pulse to the 50% point of the output pulse. Propagation delay is specified as t_{PHL} - the propagation time from a HIGH to a LOW, and as t_{PLH} - the propagation time from a LOW to a HIGH.

Output transition time: the output transition time is defined as the rise time or fall time of the output. The t_{TLH} is the 10% to 90% time, or LOW to HIGH transition time. The t_{THL} is the 90% to 10% time, or the HIGH to LOW transition time.

Setup time: the setup time is defined as the interval immediately preceding the active transition of the clock pulse during which the control or data inputs must be stable (at a valid logic level). Its parameter symbol is t_{su}.

Hold time: the hold time is the amount of time that the control or data inputs must be stable after the clock trigger occurs. The t_H is the parameter symbol for hold time.

Minimum pulse width: the minimum pulse width is required to guarantee a correct state change for the flip-flop. It is usually denoted by t_w.



Verilog program for D-latch

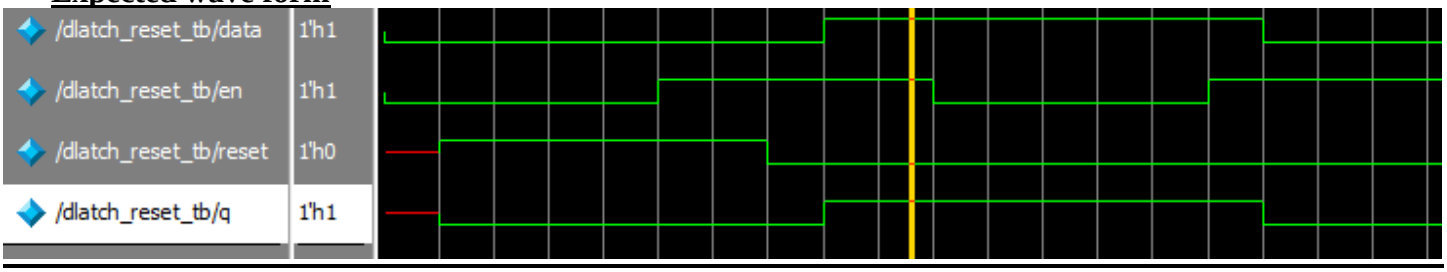
```
module dlatch_reset (data, en,  reset,  q);
    input data, en, reset ;
    output q;
    reg q;
always @ ( en or reset or data)
    if (reset) begin
    q <= 1'b0;
    end else if (en) begin
    q <= data;
    end
endmodule //End Of Module dlatch_reset
```

Verilog testbench program for D-latch

```
module  dlatch_reset_tb;
    reg data, en, reset ;
    wire q;
    dlatch_reset dlatch(data, en,  reset,  q);

    initial
    begin
    en=0;
    data = 0;
    #5 reset = 1;
    #30 reset = 0;
    $monitor($time, "\ten=%b\t ,reset=%b\t, data=%b\t, q=%b",en,reset,data,q);
    #160 $finish;
    end
    always #25  en = ~en;
    always #40 data = ~data;
 endmodule
```

Expected wave form



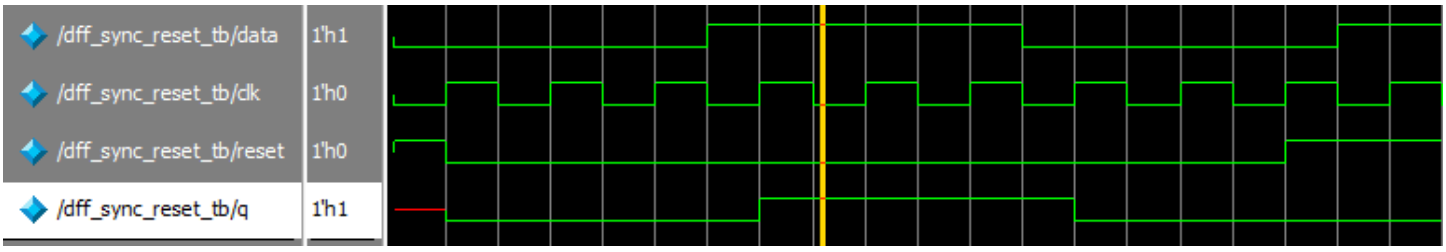
Verilog program for D-flip flop with sync reset

```
module dff_sync_reset (
                        data   , // Data Input
                        clk    , // Clock Input
                        reset  , // Reset input
                        q        // Q output
                        );
    input data, clk, reset ;
    output q;
    reg q;
always @ ( posedge clk)
    if (reset) begin
            q <= 1'b0;
    end  else begin
            q <= data;
end
endmodule
```

Verilog testbench program for D-flip flop with sync reset

```
module dff_sync_reset_tb;
    reg data, clk, reset ;
    wire q;
    dff_sync_reset dffr (.data(data), .clk(clk), .reset(reset)  ,.q(q));
    initial
    begin
    clk=0;
     data = 0;
    reset = 1;
            #5 reset = 0;
            #80 reset = 1;
            $monitor($time, "\tclk=%b\t ,reset=%b\t, data=%b\t,  q=%b",clk,reset,data,q);
            #100 $finish;
            end
            always #5  clk = ~clk;
            always #30 data = ~data;
 endmodule
```

Expected Waveform



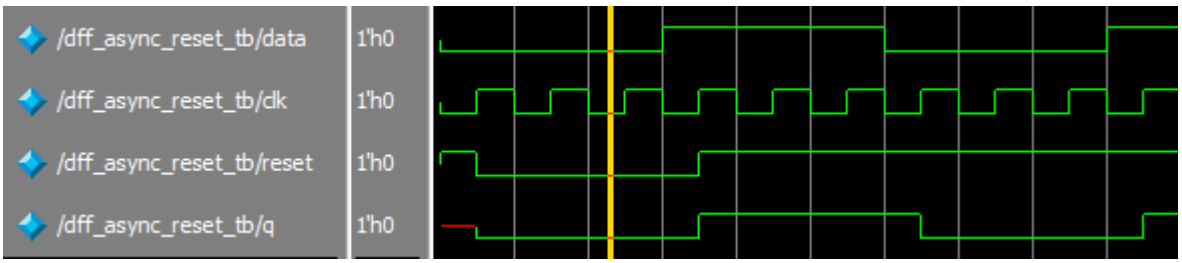
Verilog testbench program for D-flip flop with async reset

```
module dff_async_reset (
                        data  , // Data Input
                        clk    , // Clock Input
                        reset , // Reset input
                        q         // Q output
                        );
input data, clk, reset ;
     output q;
     reg q;
always @ ( posedge clk or negedge reset)
     if (~reset) begin
               q <= 1'b0;
               end  else begin
               q <= data;
               end
endmodule
```

Verilog testbench program for D-flip flop with async reset

```
module dff_async_reset_tb;
     reg data, clk, reset ;
     wire q;
dff_async_reset dffr (.data(data), .clk(clk), .reset(reset)  ,.q(q));
     initial
     begin
     clk=0;
     data = 0;
     reset = 1;
     #5 reset = 0;
     #30 reset = 1;
     $monitor($time, "\tclk=%b\t ,reset=%b\t, data=%b\t,  q=%b",clk,reset,data,q);
     #100 $finish;
     end
     always #5  clk = ~clk;
     always #30 data = ~data;
 endmodule
```

Expected Waveform



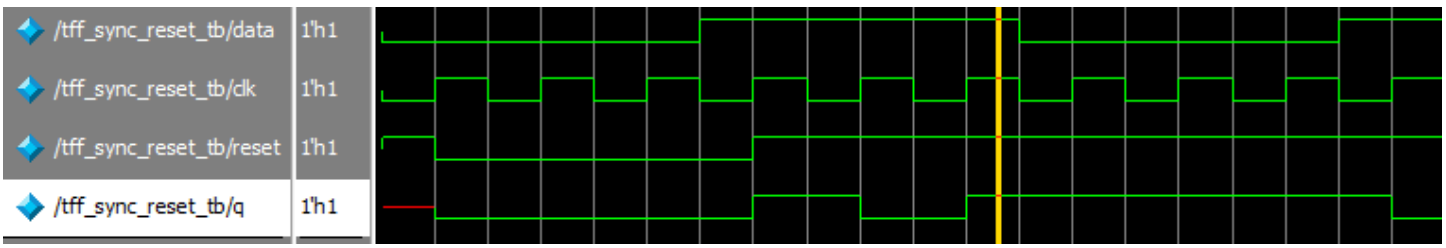
Verilog program for T-flip flop with sync reset

```
module tff_sync_reset (
                        data  , // Data Input
                        clk   , // Clock Input
                        reset , // Reset input
                        q       // Q output
                        );
    input data, clk, reset ;
    output q;
    reg q;
    always @ ( posedge clk)
            if (~reset) begin
                        q <= 1'b0;
                        end else if (data) begin
            q <= !q;
    end
endmodule
```

Verilog testbench program for T-flip flop with sync reset

```
module tff_sync_reset_tb;
    reg data, clk, reset ;
    wire q;
tff_sync_reset tffr (.data(data), .clk(clk), .reset(reset)  ,.q(q));
    initial
    begin
    clk=0;
    data = 0;
    reset = 1;
    #5 reset = 0;
    #30 reset = 1;
    $monitor($time, "\tclk=%b\t ,reset=%b\t, data=%b\t,  q=%b",clk,reset,data,q);
    #100 $finish;
    end
    always #5  clk = ~clk;
    always #30 data = ~data;
 endmodule
```

Expected Waveform



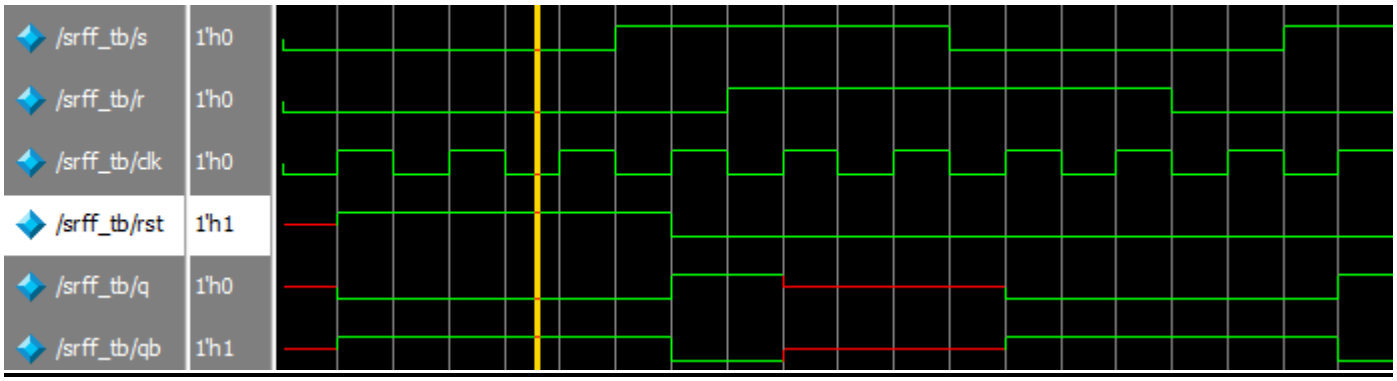
Verilog program for SR-flip flop

```
module srff(s,r,clk,rst,q,qb);
    input s,r,clk,rst;
    output q,qb;
    wire s,r,clk,rst,qb;
    reg q;
always @ (posedge clk)
    begin
    if(rst)
    q<=1'b0;
            else if (s==1'b0 && r==1'b0) q<=q;
            else if (s==1'b0&& r==1'b1)  q<=1'b0;
            else if (s==1'b1 && r==1'b0)  q<=1'b1;
            else if (s==1'b1 && r==1'b1)   q<=1'bx;
            end
            assign qb=~q;
endmodule
```

Verilog testbench program for SR-flip flop

```
module srff_tb;
    reg s,r,clk,rst;
    wire q,qb;
    srff srflipflop(.s(s),.r(r),.clk(clk),.rst(rst),.q(q),.qb(qb));
    initial
    begin
    clk=0;
    s = 0;   r = 0;
    #5 rst = 1;       #30 rst = 0;
    $monitor($time, "\tclk=%b\t ,rst=%b\t, s=%b\t,r=%b\t,  q=%b\t, qb=%b",clk,rst,s,r,q,qb);
    #100 $finish;
    end
    always #5  clk = ~clk;
    always #30 s = ~s;
    always #40 r = ~r;
 endmodule
```

Expected Waveform



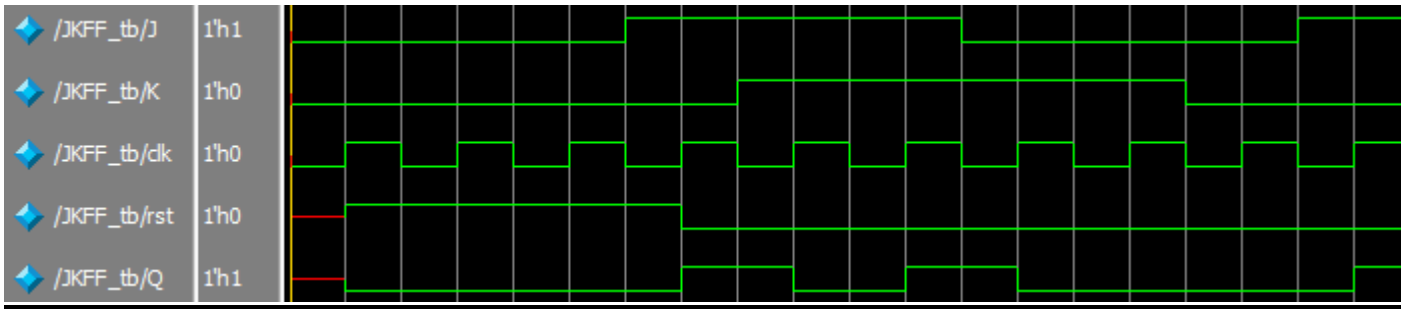
Verilog program for JK-flip flop

```
module JKFF ( input J,input K, input clk, input rst, output reg Q);
    always @(posedge clk or posedge rst) //asynch reset
    begin
                if(rst == 1)
                        begin
                        Q <= 0;
                        end
                else begin
                        case({J, K})
                        2'b00: Q <= Q; //no change
                        2'b01: Q <= 1'b0; //Clear
                        2'b10: Q <= 1'b1; //Set
                        2'b11: Q <= ~Q; //Complement
                        endcase
                end
    end
endmodule
```

Verilog testbench program for JK-flip flop with

```
module JKFF_tb;
    reg J,K,clk,rst;
    wire Q;
    JKFF JKflipflop(.J(J),.K(K),.clk(clk),.rst(rst),.Q(Q));
    initial
    begin
    clk=0;  J = 0;   K = 0;
    #5 rst = 1;
    #30 rst = 0;
    $monitor($time, "\tclk=%b\t ,rst=%b\t, J=%b\t,K=%b\t,   Q=%b",clk,rst,J,K,Q);
    #100 $finish;
    end
    always #5  clk = ~clk;
    always #30 J = ~J;
    always #40 K = ~K;
 endmodule
```

Expected Waveform



9. Design of 4-bit binary, BCD counters (Synchronous/Asynchronous reset) or any sequence Counter

Aim: To Design 4-bit binary, BCD counters using Verilog and Simulate the Design.

Apparatus required:- Electronics Design Automation Tools used:-

- **Xilinx Spartan 3 FPGA**
- **Xilinx ISE Simulator tool**
- **Xilinx XST Synthesis tool**
- **Xilinx Project Navigator 8.1i**
- **JTAG cable**
- **Adaptor 5v/4A**

Theory

The counters which use clock signal to change their transition are called "Synchronous counters". This means the synchronous counters depends on their clock input to change state values. In synchronous counters, all flip flops are connected to the same clock signal and all flip flops will trigger at the same time.

Synchronous counters are also known as ' Simultaneous counters '.There is no propagation delay and no ripple effect in synchronous counters.

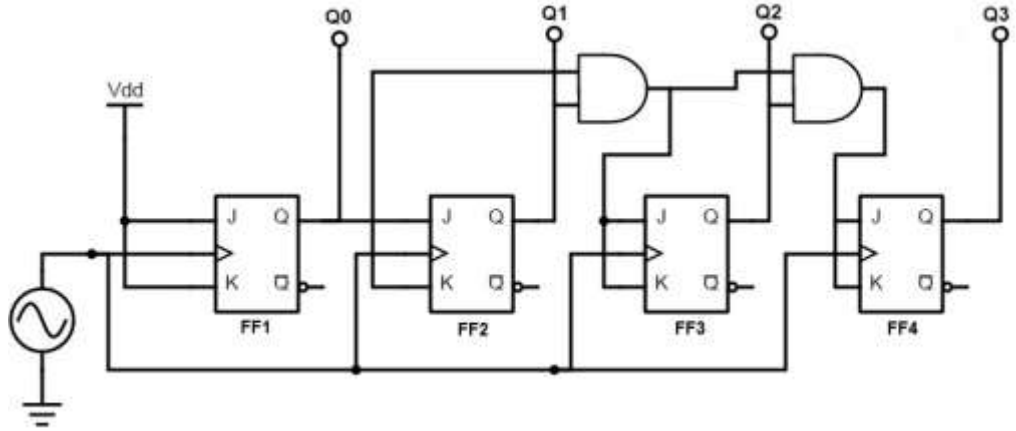
4 bit Synchronous UP Counter

The 4 bit up counter shown in below diagram is designed by using JK flip flop. External clock pulse is connected to all the flip flops in parallel.

For designing the counters JK flip flop is preferred .The significance of using JK flip flop is that it can toggle its state if both the inputs are high, depending on the clock pulse.

The inputs of first flip flop are connected to HIGH (logic 1), which makes the flip flop to toggle, for every clock pulse entered into it. So the synchronous counter will work with single clock signal and changes its state with each pulse.

The output of first JK flip flop (Q) is connected to the input of second flip flop. The AND gates (which are



connected externally) drives the inputs of other two flip flops . The inputs of these AND gates , are supplied from previous stage lip flop outputs.

If inputs of FF2 are connected directly to the Q1 output of FF1 , the counter would not function properly. This is because , the Q1 value is high at count of 210 , this means that the FF2 flip flop will toggle for the 3rd clock pulse. This results in wrong counting operation, gives the count as 710 instead of 410.

To prevent this problem AND gates are used at the input side of FF2 and FF3. The output of the AND gate will be high only when the Q0, Q1 outputs are high. So for the next clock pulse, the count will be 00012.

Similarly, the flip flop FF3 will toggle for the fourth clock pulse when Q0, Q1 and Q2 are high. The Q3 output will not toggle till the 8th clock pulse and will again remain high until 16th clock pulse. After the 16th clock pulse, the q outputs of all flip flops will return to 0.

Operation

In the up counter the 4 bit binary sequence starts from 0000 and increments up to 1111.Before understanding the working of the above up counter circuit know about JK Flip flop.

In the above circuit as the two inputs of the flip flop are wired together. So , there are only two possible conditions that can occur, that is, either the two inputs are high or low.

If the two inputs are high then JK flip-flop toggles and if both are low JK flip flop remembers i.e. it stays in the previous state.

Let us see the operation. Here clock pulse indicates edge triggered clock pulse .

1.) In the first clock pulse, the outputs of all the flip flops will be at 0000.

2.)In the second clock pulse, as inputs of J and k are connected to the logic high, output of JK flip flop(FF0) change its state .Thus the output of the first flip-flop(FF0) changes its state for every clock pulse .This can be observed in the above shown sequence .The LSB changes its state alternatively. Thus producing -0001

3.) In the third clock pulse next flip flop (FF1) will receive its J K inputs i.e (logic high) and it changes its state. At this state FF0 will change its state to 0. And thus input on the FF1 is 0.Hence output is -0010

4.) Similarly, in the fourth clock pulse FF1 will not change its state as its inputs are in low state, it remains in its previous state. Though it produces the output to FF2, it will not change its state due to the presence of AND gate. FF0 will again toggle its output to logic high state. Thus Output is 0011.

5.) In the fifth clock pulse, FF2 receives the inputs and changes its state. While, FF0 will have low logic on its output and FF1 will also be low state producing 0100.

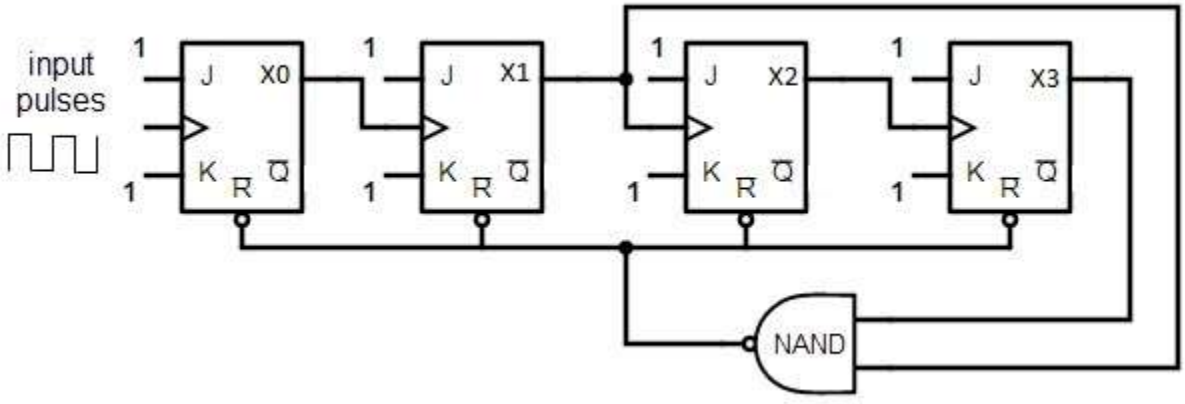
This process continuous up to 1111.

After reaching zero again the three flip flops toggles to logic low i.e 0000 and again count starts. Timing diagram for up counter is shown below.

BCD or Decade Counter Circuit

A binary coded decimal (BCD) is a serial digital counter that counts ten digits .And it resets for every new clock input. As it can go through 10 unique combinations of output, it is also called as "Decade counter". A BCD counter can count 0000, 0001, 0010, 1000, 1001, 1010, 1011, 1110, 1111, 0000, and 0001 and so on.

A 4 bit binary counter will act as decade counter by skipping any six outputs out of the 16 (24) outputs. There are some available ICs for decade counters which we can readily use in our circuit, like 74LS90. It is an asynchronous decade counter.



The above figure shows a decade counter constructed with JK flip flop. The J output and K outputs are connected to logic 1. The clock input of every flip flop is connected to the output of next flip flop, except the last one.

The output of the NAND gate is connected in parallel to the clear input 'CLR' to all the flip flops. This ripple counter can count up to 16 i.e. 24.

Decade Counter Operation

When the Decade counter is at REST, the count is equal to 0000. This is first stage of the counter cycle. When we connect a clock signal input to the counter circuit, then the circuit will count the binary sequence. The first clock pulse can make the circuit to count up to 9 (1001). The next clock pulse advances to count 10 (1010).

Then the ports X1 and X3 will be high. As we know that for high inputs, the NAND gate output will be low. The NAND gate output is connected to clear input, so it resets all the flip flop stages in decade counter. This means the pulse after count 9 will again start the count from count 0.

Truth Table of Decade Counter

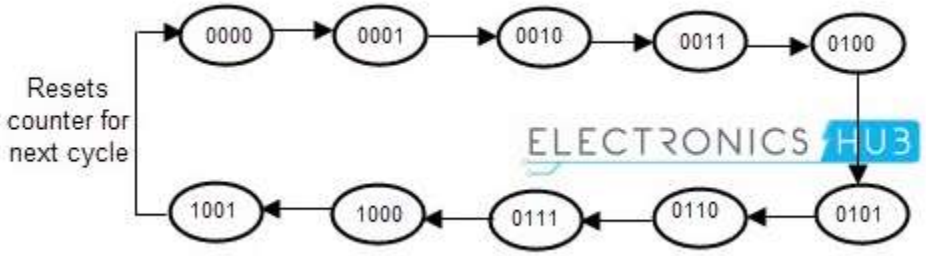
Input Pulses	D	C	B	A
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
0	0	0	0	0 (resets)

The above table describes the counting operation of Decade counter. It represents the count of circuit for decimal count of input pulses. The NAND gate output is zero when the count reaches 10 (1010).

The count is decoded by the inputs of NAND gate X1 and X3. After count 10, the logic gate NAND will trigger its output from 1 to 0, and it resets all flip flops.

State Diagram of Decade Counter

The state diagram of Decade counter is given below.If we observe the decade counter circuit diagram, there are



four stages in it, in which each stage has single flip flop in it. So it is capable of counting 16 bits or 16 potential states, in which only 10 are used. The count starts from 0000 (zero) to 1001 (9) and then the NAND gate will reset the circuit.

Multiple counters are connected in series, to count up to any desired number. The number that a counter circuit can count is called “Mod” or “Modulus”. If a counter resets itself after counting n bits is called “Mod- n counter” “Modulo- n counter”, where n is an integer.

The Mod n counter can calculate from 0 to 2n-1. There are several types of counters available, like Mod 4 counter, Mod 8 counter, Mod 16 counter and Mod 5 counters etc.

Verilog program for 4 bit binary counter

```
module counter    (out, enable, clk,reset);
output [3:0] out;//----------Output Ports------
input enable, clk, reset; ////--- Input Ports----
reg [3:0] out;//---Internal Variables--------

always @(posedge clk)
     if (reset) begin
     out <= 4'b0 ;
     end else if (enable) begin
     out <= out + 1'b1;
     end
     endmodule
```

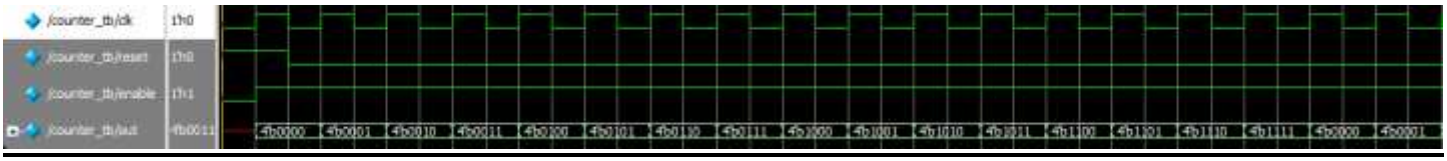
Verilog testbench program for 4 bit binary counter

```
module counter_tb;
  reg clk, reset, enable;
  wire [3:0] out;
    counter U0 ( .clk    (clk), .reset  (reset), .enable (enable),.out  (out));
  initial begin
    clk = 0;  enable = 0;
reset = 1;
#5 enable = 1;
#5   reset = 0;
#100 $finish;
end

  always #5 clk = !clk;

initial  begin
    $display("\t\ttime,\tclk,\treset,\tenable,\tout");
    $monitor("%d,\t%b,\t%b,\t%b,\t%d",$time, clk,reset,enable,out);
  end
endmodule
```

Expected waveform



Verilog program BCD counter

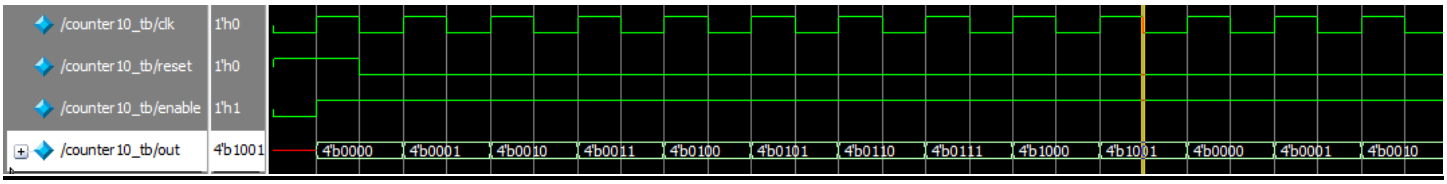
```
module counter10    (out     , enable  ,   clk     ,  reset);
   output [3:0] out;
   input enable, clk, reset;
   reg [3:0] out;

always @(posedge clk)
     if (reset) begin
              out <= 4'b0 ;
     end else if (enable) begin
              out <= (out + 1)%10;
     end
endmodule
```

Verilog testbench program BCD counter

```
module counter10_tb;
  reg clk, reset, enable;
  wire [3:0] out;
  counter10 U0 (.clk(clk), .reset  (reset), .enable (enable),  .out  (out) );
 initial begin
    clk = 0;
    reset = 1;
    enable = 0;
    #5 enable = 1;
    #5 reset = 0;
#1000 $finish;
  end
     always     #5 clk = !clk;
  initial  begin
    $display("\t\ttime,\tclk,\treset,\tenable,\tout");
    $monitor("%d,\t%b,\t%b,\t%b,\t%d",$time, clk,reset,enable,out);
  end
endmodule
```

Expected waveform



10. Finite State Machine Design

Aim: To Design the finite state machine using verilog and Simulate the Design.
Apparatus required :- Electronics Design Automation Tools used:-

- Xilinx Spartan 3 FPGA
- Xilinx ISE Simulator tool
- Xilinx XST Synthesis tool
- Xilinx Project Navigator 8.1i
- JTAG cable
- Adaptor 5v/4A

Theory:

Basically a FSM consists of combinational, sequential and output logic. Combinational logic is used to decide the next state of the FSM, sequential logic is used to store the current state of the FSM. The output logic is a mixture of both combo and seq logic as shown in the figure below.

Types of State Machines

There are many ways to code these state machines, but before we get into the coding styles, let's first understand the basics a bit. There are two types of state machines:

- Mealy State Machine : Its output depends on current state and current inputs. In the above picture, the blue dotted line makes the circuit a mealy state machine.
- Moore State Machine : Its output depends on current state only. In the above picture, when blue dotted line is removed the circuit becomes a Moore state machine.

Combinational always blocks are always blocks that are used to code combinational logic functionality and are strictly coded using blocking assignments. A combinational always block has a combinational sensitivity list, a sensitivity list without "posedge" or "negedge" Verilog keywords.
Sequential always blocks are always blocks that are used to code clocked or sequential logic and are always coded using nonblocking assignments. A sequential always block has an edge-based sensitivy list.

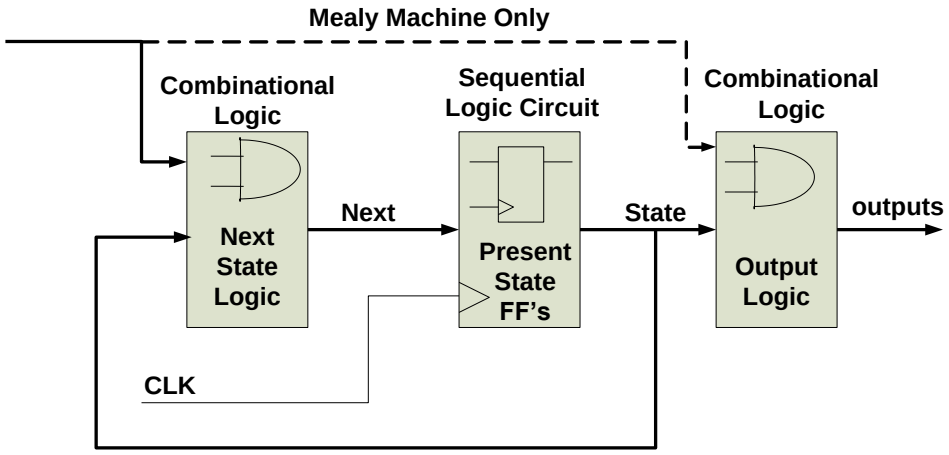
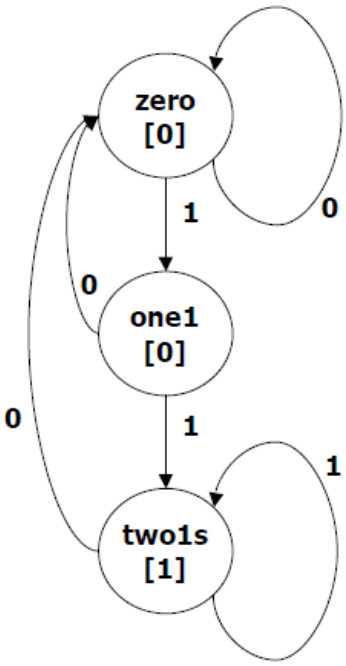


Fig: Mealy & Moore FSMs

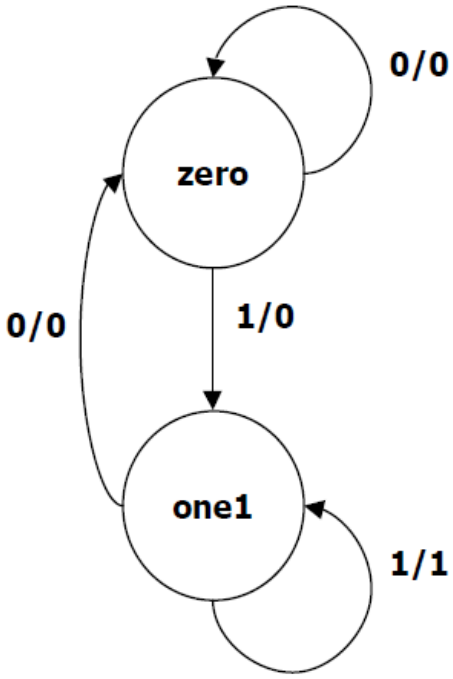
Encoding Style

Since we need to represent the state machine in a digital circuit, we need to represent each state in one of the following ways:
Binary encoding : each state is represented in binary code (i.e. 000, 001, 010....)
Gray encoding : each state is represented in gray code (i.e. 000, 001, 011,...)
One Hot : only one bit is high and the rest are low (i.e. 0001, 0010, 0100, 1000)
One Cold : only one bit is low, the rest are high (i.e. 1110,1101,1011,0111)

State diagram for Moore and Mealy machine



Detect 2 Consecutive 1 inputs (Moore)



Detect 2 Consecutive 1 inputs (Mealy)

General View of State Machine:

```
module FSM (CLK, in, out);
input CLK;
input in;
output out;
reg out;
// state variable
reg [1:0] state;
// local variable
reg [1:0] next_state;
always @(posedge CLK) // registers
state = next_state;
always @(state or in)
// Compute next-state and output logic whenever state or inputs change.
// (i.e. put equations here for next_state[1:0])
// Make sure every local variable has an assignment in this block!
endmodule
```

Verilog coding for moore FSM

```
`define zero 2'b00
`define one1 2'b01
`define two1s 2'b10
module moore_fsm (CLK, reset, in, out);
input CLK, reset, in;
output out;
reg out;
reg [1:0] state; // state variables
reg [1:0] next_state;
        always @(posedge CLK)
        if (reset) state = `zero;
        else state = next_state;

always @(in or state)
        case (state)
        `zero: // last input was a zero
                begin
                if (in) next_state = `one1;
                else next_state = `zero;
                end
        `one1: // we've seen one 1
                begin
                if (in) next_state = `two1s;
                else next_state = `zero;
                end
        `two1s: // we've seen at least 2 ones
                begin
                if (in) next_state = `two1s;
                else next_state = `zero;
                end
                endcase

always @(state)
                case (state)
                `zero: out = 0;
                `one1: out = 0;
                `two1s: out = 1;
                endcase
endmodule
```

Verilog coding for Mealy FSM

```
`define zero 2'b00
`define one1 2'b01
`define two1s 2'b10
module mealy_fsm (clk, reset, in, out);
        input clk, reset, in;
        output out;
        reg out;
        reg state; // state variables
always @(posedge clk)
        if (reset) state = `zero;
        else
        case (state)
                `zero: // last input was a zero
                begin
                out = 0;
                if (in) state = `one;
                else state = `zero;
                end
                `one: // we've seen one 1
                if (in) begin
                state = `one; out = 1;
                end else begin
                state = `zero; out = 0;
                end
        endcase
endmodule
```

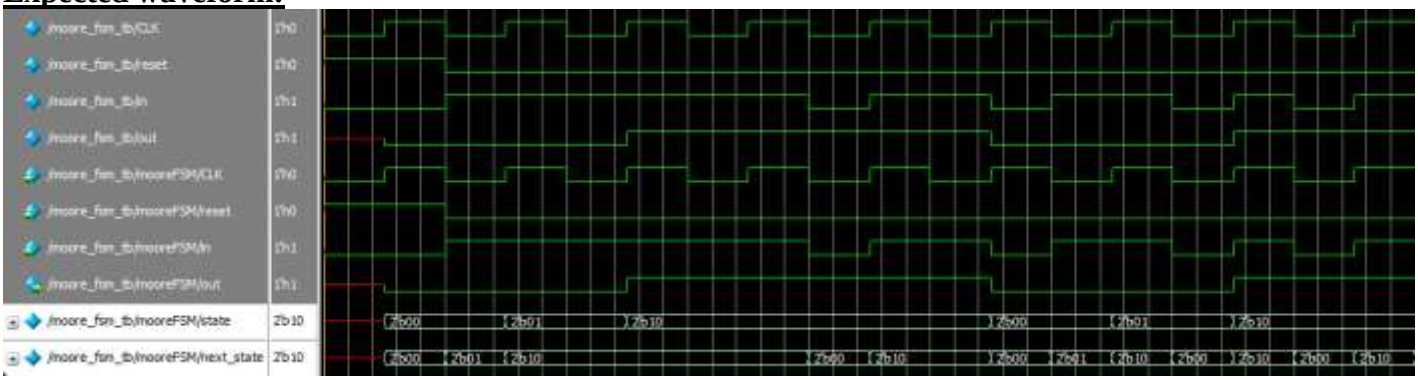
Verilog testbench program for Moore FSM

```
module moore_fsm_tb;
        reg CLK, reset, in;
        wire out;
moore_fsm mooreFSM(.CLK(CLK), .reset(reset), .in(in), .out(out));
initial
        begin
        CLK = 0;
         reset = 1;
         in = 0;
        #10 reset = 0;
        #100   $finish;
 end

 always
        #5  CLK =  ! CLK;
always
        #5 in = $random;

initial  begin
  $display("\t\ttime,\tCLK,\treset,\tin,\tout");
  $monitor("%d,\t%b,\t%b,\t%b,\t%b",$time, CLK,reset,in,out);
  end
  endmodule
```

Expected waveform:



Verilog Testbench for Mealy Machine:

```
module mealy_fsm_tb;
reg clk, reset, in;
wire out;
mealy_fsm mooreFSM(.clk(clk), .reset(reset), .in(in), .out(out));
initial
 begin
 clk = 0;
 reset = 1;
 in = 0;
#10 reset = 0;
#100   $finish;
 end

 always
  #5  clk =  ! clk;
always
#5 in = $random;

initial  begin
  $display("\t\ttime,\tclk,\treset,\tin,\tout");
$monitor("%d,\t%b,\t%b,\t%b,\t%b",$time, clk,reset,in,out);
  end
endmodule
```

Expected waveform: