NITTE MEENAKSHI
INSTITUTE OF TECHNOLOGY

DIGITAL DESIGN USING VERILOG

LAB MANUAL

18ECL47

SEMESTER: IV

DSD USING VERILOG LAB

Course Code	**18ECL47**	**Credits**	**01**
Hours/Week(L-T-P)	**0-0-2**	**CIE Marks**	**50**
Total Hours	**26(P)**	**SEE Marks**	**50**
Exam Hours	**03**	**Course Type**	**Core Lab**

PRE-REQUISITES

- Digital Electronics

COURSE OUTCOMES

1. Familiarize with EDA tools for simulation, verification and synthesis of digital design.
2. Simulate Combinational circuits in Dataflow, Behavioral and Gate level Abstraction using Verilog HDL code
3. Program sequential circuits like flip flops and counters in Behavioral description and obtain simulation waveforms.
4. Implement Combinational and Sequential circuits on FPGA and test the hardware.
5. Interface/Emulate the hardware to the FPGA and obtain the required output

LIST OF EXPERIMENTS

1. Realization of logic gates using Verilog HDL code and three/ four variable expression
2. Realization of combinational designs using Verilog HDL code
   a. 2 to 4 decoders
   b. 8 to 3 (encoder without priority & with priority)
   c. 8 to 1 multiplexer
   d. 1 to 8 De Multiplexer
   e. 4 bit binary to gray converter and vice versa
   f. Comparator.
3. Realize and verify the function of a Full Adder in three modelling styles using HDL code .
4. Realization of N- bit ALU for the give function
5. Develop the HDL Code for the following flip-flops: SR, D, JK, and T
6. Design BCD counters (Synchronous reset and asynchronous reset) and "any sequence" counters.

Note: Test bench program should be written for all the above experiments

CO-PO-PSO MAPPING

CO/ PO	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3	BT
CO1	3		1		2				2	2		3	3			2
CO2	3	2	2		2				2	2		3	3		1	3
CO3	3	2	2		2				2	2		3	3		1	3
CO4	3	3	2	2	2				2	2		3	3	1	1	3
CO5	3	3	2	2	2				2	2		3	3	1	1	3

Experiment 1

Aim : Realization of logic gates using Verilog HDL code in all three modelling style .

Gate Level Modelling

```
module LogicGates(a,b,y);
input a,b;
output [0:6]y;
not(y[0],a);
and(y[1], a,b);
or(y[2],a,b);
nand(y[3],a,b);
nor(y[4],a,b);
xor(y[5],a,b);
xnor(y[6],a,b);
endmodule
```

Behavioral Modelling

```
module gates (a, b, y);
input a, b;
output reg [0:6]y;

always @ (a or b)
begin
y[0]=~a;
y[1]=a&b;
y[2]=a|b;
y[3]=~(a &b);
y[4]=~(a|b);
y[5]=a^b;
y[6]=~(a^b);
end
endmodule
```

Data flow modelling

```
module gates (a, b, y);
input a, b;
output [0:6]y;

assign y[0]=~a;
```

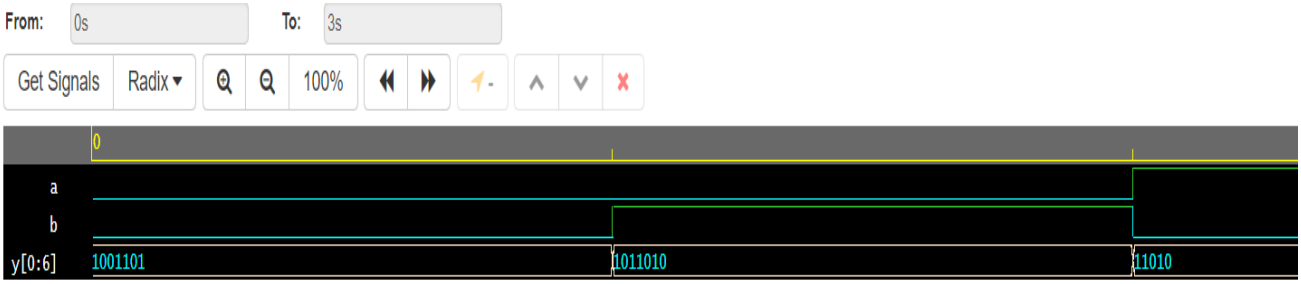
```
assign y[1]=a&b;
assign y[2]=a|b;
assign y[3]=~(a &b);
assign y[4]=~(a|b);
assign y[5]=a^b;
assign y[6]=~(a^b);
end
endmodule
```

Test bench

```
module TestModule;

// Inputs

reg a;

reg b;

// Outputs

  wire [0:6]y;

// Instantiate the Unit Under Test (UUT)

gates uut (a,b,y);

initial begin

// Initialize Inputs

a = 0; b = 0;

// Wait 100 ns for global reset to finish

#1

a = 0;b = 1;

#1

a = 1;

b = 0;

#1

a = 1;
```

```
b = 1;

end

endmodule
```

Waveform



1 b

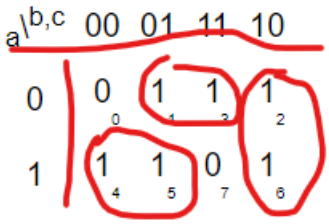
To design and develop Verilog HDL code to realize the given three variables and four variable functions.

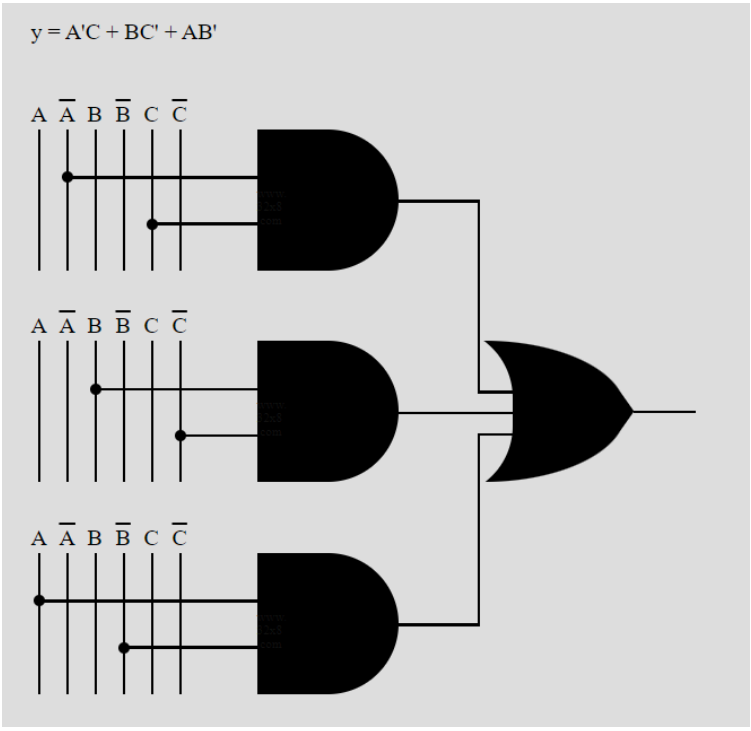
(a,b,c) = ∑m(1,2,3,4,5,6)

Solution:

Minterm = $\sum m(1, 2, 3, 4, 5, 6)$

Variable = 3
As Max of Minterm is 6, So we have taken N = 3
and Variable = a,b,c





Code

```
Dataflow Modelling
module exp(a,b,c,y);
input a,b,c;
output y;
assign y = ( b & ~c) | (~a & c)| (a & ~b);
endmodule


Behavioural Modelling
module exp(a,b,c,y);
input a,b,c;
output reg  y;
always @(*)

 begin

y = ( b & ~c) | (~a & c)| (a & ~b);
end
endmodule


Structural Modelling
module exp(a,b,c,y);
input a,b,c;
output y;
wire t1,t2,t3,u1,u2,u3;

not n1(t1,a);
not n2(t2,b);
not n3(t3,c);

and a1(u1,b,t3);
and a2(u2,t1,c);
and a3(u3,a,t2);

or o1(y,u1,u2,u3);
endmodule
```

Test bench

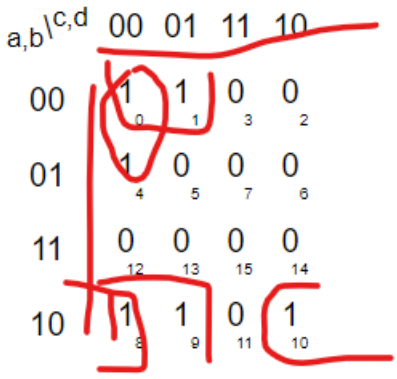
```
module ex_tb;
wire y;
reg a,b,c;

exp a1(a,b,c,y);

initial begin
a=0; b=0; c=0;
a=0; b=0; c=1;
a=0; b=1; c=0;
a=0; b=1; c=1;
a=1; b=0; c=0;
a=1; b=0; c=1;
a=1; b=1; c=0;
a=1; b=1; c=1;
end

endmodule
```

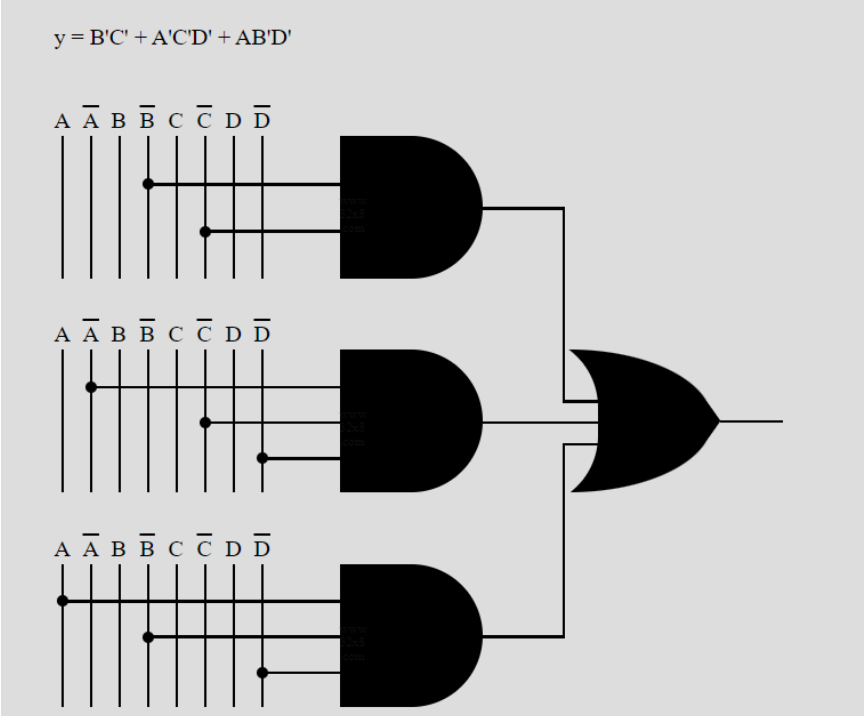
2. f(a,b,c,d) = ∑m(0,1,4,8,9,10)

Solution:

Minterm = $\sum m(0, 1, 4, 8, 9, 10)$

Variable = 4
As Max of Minterm is 10, So we have taken N = 4
and Variable = a,b,c,d





Code

```
Dataflow Modelling
module exp(a,b,c,d,y);
input a,b,c,d;
output y;
assign y = ( ~b & ~c) | (~a & ~c& ~d)| (a & ~b &~d);
endmodule


Behavioural Modelling
module exp(a,b,c,d,y);
input a,b,c,d;
output reg  y;
always @(*)

 begin

 y = ( ~b & ~c) | (~a & ~c& ~d)| (a & ~b &~d);
end
endmodule


Structural Modelling
module exp(a,b,c,d,y);
input a,b,c,d;
output y;
wire t1,t2,t3,t4,u1,u2,u3;

not n1(t1,a);
not n2(t2,b);
not n3(t3,c);
not n4(t4,d);

and a1(u1,t2,t3);
and a2(u2,t1,t3,t4);
and a3(u3,a,t2,t4);

or o1(y,u1,u2,u3);
endmodule
```

Test Bench

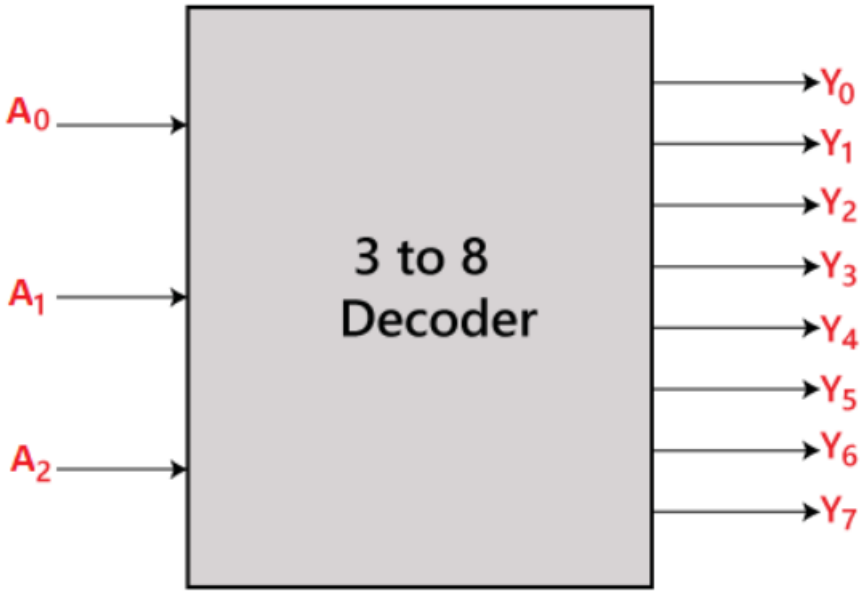
```
module ex_tb;
wire y;
reg a,b,c,d;

exp a1(a,b,c,d,y);

initial begin
a=0; b=0; c=0;d=0;
a=0; b=0; c=0;d=1;
a=0; b=0; c=1;d=0;
a=0; b=0; c=1;d=1;
a=0; b=1; c=0;d=0;
a=0; b=1; c=0;d=1;
a=0; b=1; c=1;d=0;
a=0; b=1; c=1;d=1;
a=1; b=0; c=0;d=0;
a=1; b=0; c=0;d=1;
a=1; b=0; c=1;d=0;
a=1; b=0; c=1;d=1;
a=1; b=1; c=0;d=0;
a=1; b=1; c=0;d=1;
a=1; b=1; c=1;d=0;
a=1; b=1; c=1;d=1;

end

endmodule
```

Experiment -2

Aim : To design and develop Verilog HDL code and implement for the following combinational circuits .

Decoder

Block Diagram:



Truth Table:

Enable	INPUTS			Outputs							
E	A_2	A_1	A_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	x	x	x	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

The logical expression of the term Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, and Y_7 is as follows:

$Y_0=A_0'.A_1'.A_2'$

$Y_1=A_0.A_1'.A_2'$

$Y_2=A_0'.A_1.A_2'$

$Y_3=A_0.A_1.A_2'$

$Y_4=A_0'.A_1'.A_2$

$Y_5=A_0.A_1'.A_2$

$Y_6=A_0'.A_1.A_2$

$Y_7=A_0.A_1.A_2$

```
//declare the Verilog module - The inputs and output port names.
module decoder3to8(Data_in,en,Data_out);
    //what are the input ports and their sizes.
    input [2:0] Data_in;
    input en;

    //what are the output ports and their sizes.
```

```
    output [7:0] Data_out;
    //Internal variables
    reg [7:0] Data_out;

    //Whenever there is a change in the Data_in, execute the always block.
    always @(Data_in or en)
    begin
    if(en==0)
    begin
    Data_out=8'b0;
    end
    else
    begin
    case (Data_in)   //case statement. Check all the 8 combinations.
        3'b000 : Data_out = 8'b00000001;
        3'b001 : Data_out = 8'b00000010;
        3'b010 : Data_out = 8'b00000100;
        3'b011 : Data_out = 8'b00001000;
        3'b100 : Data_out = 8'b00010000;
        3'b101 : Data_out = 8'b00100000;
        3'b110 : Data_out = 8'b01000000;
        3'b111 : Data_out = 8'b10000000;
//To make sure that latches are not created create a default value for output.
        default : Data_out = 8'b00000000;
    endcase
    end
endmodule
```

Testbench

```
module tb_decoder;

    // Declaring Inputs
    reg [2:0] Data_in;
    reg en;

    // Declaring Outputs
    wire [7:0] Data_out;

    // Instantiate the Unit Under Test (UUT)
    decoder3to8 uut (.Data_in(Data_in), .en(en),.Data_out(Data_out));

    initial begin
     #100 en=1'b0;
     #100 en=1'b1;
        Data_in = 3'b000;     #100;
        Data_in = 3'b001;     #100;
        Data_in = 3'b010;     #100;
        Data_in = 3'b011;     #100;
        Data_in = 3'b100;     #100;
        Data_in = 3'b101;     #100;
        Data_in = 3'b110;     #100;
        Data_in = 3'b111;     #100;
    end
```

endmodule

8 : 3 Encoder (Octal to Binary) –

The 8 to 3 Encoder or octal to Binary encoder consists of **8 inputs** : Y7 to Y0 and **3 outputs** : A2, A1 & A0. Each input line corresponds to each octal digit and three outputs generate corresponding binary code.
The figure below shows the logic symbol of octal to binary encoder:

Inputs								Outputs			
A7	A6	A5	A4	A3	A2	A1	A0	Y2	Y1	Y0	Valid
0	0	0	0	0	0	0	0	X	X	X	0
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	1	X	0	0	1	1
0	0	0	0	0	1	X	X	0	1	0	1
0	0	0	0	1	X	X	X	0	1	1	1
0	0	0	1	X	X	X	X	1	0	0	1
0	0	1	X	X	X	X	X	1	0	1	1
0	1	X	X	X	X	X	X	1	1	0	1
1	X	X	X	X	X	X	X	1	1	1	1

Verilog code

```
// Code your design here

module encoder8_3(en, a_in, y_op,v);

input en;

input [7:0] a_in;

output reg v;

output [2:0] y_op;

reg [2:0] y_op;

always @ (a_in,en)

begin
```

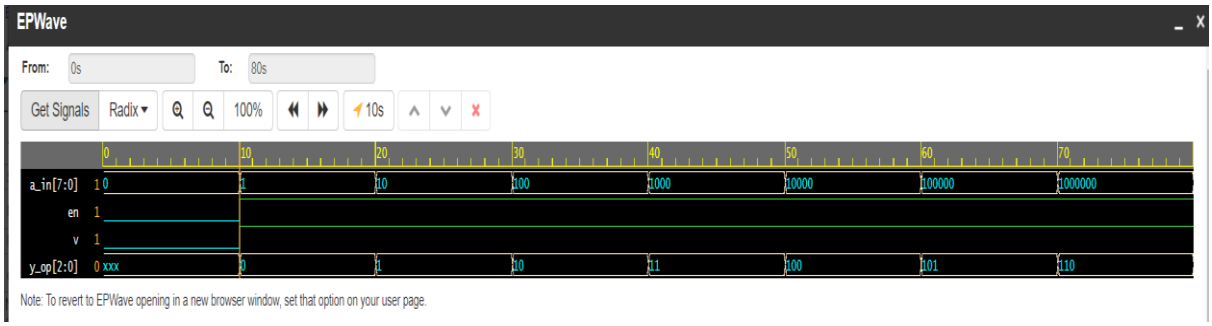
```
  if(en==0 )
    begin
      y_op =3'bxxx;
      v=1'b0;
    end
else
  begin
    v=1'b1;
case (a_in)
8'b00000001: y_op = 3'b000;
8'b00000010: y_op = 3'b001;
8'b00000100: y_op = 3'b010;
8'b00001000: y_op = 3'b011;
8'b00010000: y_op = 3'b100;
8'b00100000: y_op = 3'b101;
8'b01000000: y_op = 3'b110;
8'b10000000: y_op = 3'b111;
default: y_op =3'bzzz;
endcase
end
end
endmodule
```

Test bench

```
// Code your testbench here
// or browse Examples
```

```
module encodertest_tb;

// Inputs
reg en;
reg [7:0] a_in;


// Outputs
wire v;
wire [2:0] y_op;


// Instantiate the Unit Under Test (UUT)
encoder8_3 uut(en,a_in,y_op,v);

    initial begin
      $dumpfile("dump.vcd");
      $dumpvars(1);
        a_in = 8'b00000000;
        en = 1'b0;

        #10 en =1'b1; a_in = 8'b00000001;
        #10 a_in = 8'b00000010;
        #10 a_in = 8'b00000100;
        #10 a_in = 8'b00001000;
        #10 a_in = 8'b00010000;
        #10 a_in = 8'b00100000;
```

```
    #10 a_in = 8'b01000000;
    #10 a_in = 8'b10000000;
  end

endmodule
```



8 to 3 Priority Encoder

This kind of encoder is also named an 8-bit or Octal to Binary priority encoder. This type of encoder consists of 8 inputs and 3 outputs. When multiple inputs are active high at the same time, the input with the highest priority is considered to represent the output.

Inputs								Outputs			
A7	A6	A5	A4	A3	A2	A1	A0	Y2	Y1	Y0	Valid
0	0	0	0	0	0	0	0	X	X	X	0
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	1	X	0	0	1	1
0	0	0	0	0	1	X	X	0	1	0	1
0	0	0	0	1	X	X	X	0	1	1	1
0	0	0	1	X	X	X	X	1	0	0	1
0	0	1	X	X	X	X	X	1	0	1	1
0	1	X	X	X	X	X	X	1	1	0	1
1	X	X	X	X	X	X	X	1	1	1	1

```
module prio_enco(en, a_in, y_op,v);

input en;
```

```
input [7:0] a_in;
output reg v;
output [2:0] y_op;
reg [2:0] y_op;
always @ (a_in,en)
begin
  if(en==0 )
    begin
      y_op =3'bxxx;
      v=1'b0;
    end
else
  begin
    v=1'b1;
case (a_in)
8'b00000001: y_op =3'b000;
8'b0000001x: y_op= 3'b001;
8'b000001xx: y_op= 3'b010;
8'b00001xxx: y_op= 3'b011;
8'b0001xxxx: y_op= 3'b100;
8'b001xxxxx: y_op= 3'b101;
8'b01xxxxxx: y_op= 3'b110;
8'b1xxxxxxx: y_op= 3'b111;
default: y_op=3'bzzz;
endcase
```

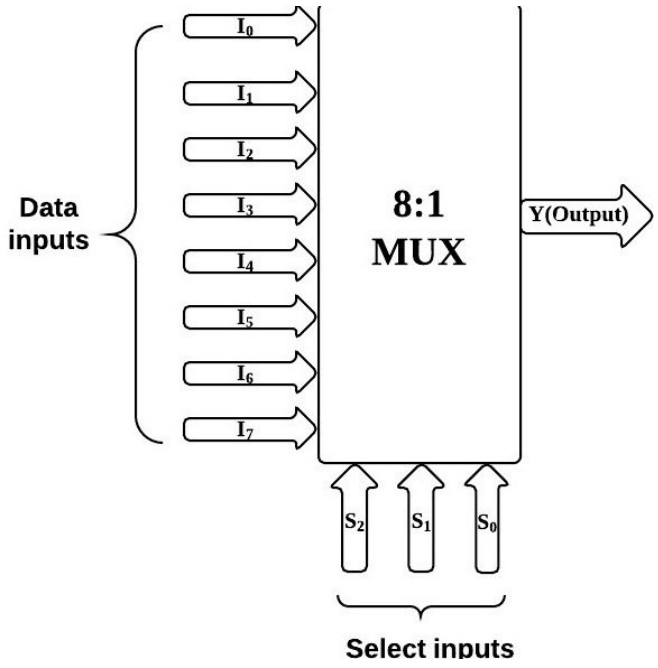
```
end

end

endmodule
```

Test bench

```
// Code your testbench here

// or browse Examples

module encoder_prio_test_tb;


// Inputs

reg en;

reg [7:0] a_in;



// Outputs

wire v;

wire [2:0] y_op;



// Instantiate the Unit Under Test (UUT)

prio_enco uut(en,a_in,y_op,v);


    initial begin

      $dumpfile("dump.vcd");

      $dumpvars(1);
```

```
        a_in = 8'b00000000;
        en = 1'b0;

        #10 en =1'b1; a_in = 8'b00000001;
        #10 a_in = 8'b00000010;
        #10 a_in = 8'b00000100;
        #10 a_in = 8'b00001000;
        #10 a_in = 8'b00010000;
        #10 a_in = 8'b00100000;
        #10 a_in = 8'b01000000;
        #10 a_in = 8'b10000000;
    end

endmodule
```

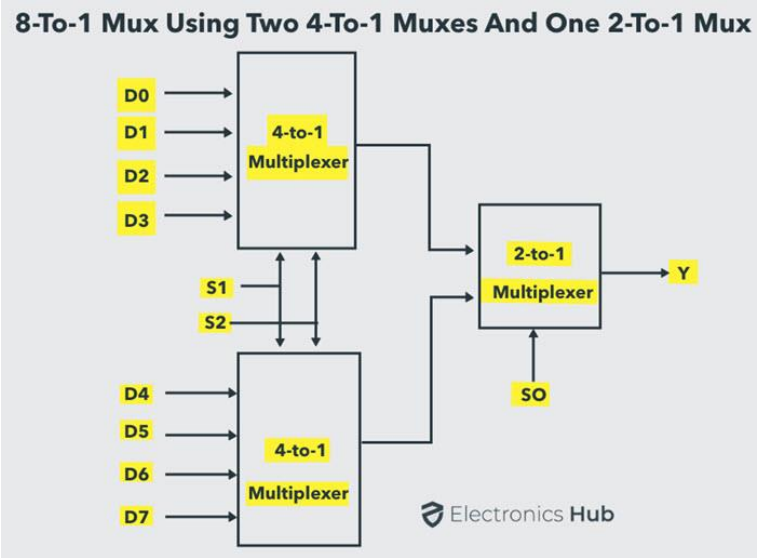
3. 8 : 1 mux using 4:1 mux and 2:1 mux

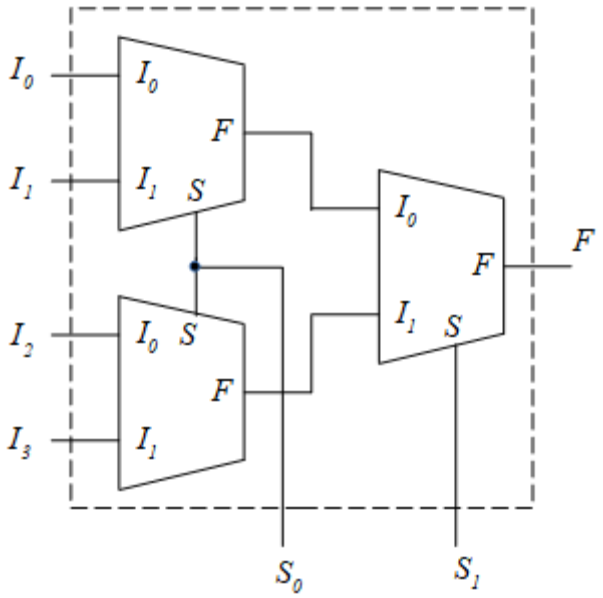


8 :1 Mux

Inputs			Output
S_2	S_1	S_0	O
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

8:1 mux using 4:1 and 2:1






```
// Code your design here
module mux2to1(a,b,sel,out);
      input a,b,sel;
      output out;
      bufif1 (out,a,sel);
      bufif0 (out,b,sel);
endmodule

module mux4to1(a,sel,out);
input [3:0] a;
input [1:0] sel;
output out;
wire mux_1,mux_2;
  mux2to1 m1 (a[3],a[2],sel[0],mux_1);
  mux2to1   m2 (a[1],a[0],sel[0],mux_2);
  mux2to1    m3 (mux_1,mux_2,sel[1],out);
endmodule

module mux8to1(a,sel,out);
      input [7:0] a;
      input [2:0] sel;
      output out;

      wire mux_1,mux_2;

      mux4to1 m1 (a[7:4],sel[1:0],mux_1),
              m2 (a[3:0],sel[1:0],mux_2);
      mux2to1 m3 (mux_1,mux_2,sel[2],out);
endmodule
```

```
// Code your testbench here
// or browse Examples
// Code your testbench here
// or browse Examples
module TB_MUX_8;
reg [7:0] d;
reg [2:0] sel;
wire out;

// Basic Gates is what we are going to test.
mux8to1 U1(d,sel,out);

initial
begin
$dumpvars(1);
  d[0] = 1;
d[1] = 1;
d[2] = 1;
d[3] = 1;
d[4] = 1;
d[5] = 0;
d[6] = 0;
d[7] = 0;
  sel[2] = 1;
  sel[1] = 0;
  sel[0] = 0;

#1
  d[0] = 0;
d[1] = 0;
  d[2] = 0;
  d[3] = 0;
d[4] = 0;
d[5] = 1;
d[6] = 1;
d[7] = 1;
  sel[2] = 1;
sel[1] = 1;
  sel[0] = 0;
```

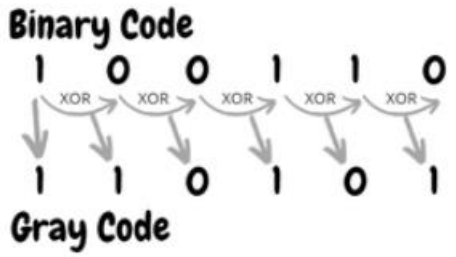
```
  #1
d[0] = 0;
d[1] = 0;
d[2] = 0;
d[3] = 0;
d[4] = 1;
d[5] = 0;
d[6] = 1;
  d[7] = 0;
sel[2] = 1;
sel[1] = 1;
sel[0] = 1;

end
endmodule
```

4) 4- bit Binary to Gray Code Converter

Using Exclusive-Or (⊕) operation −

This is very simple method to get Gray code from Binary number. These are following steps for n-bit binary numbers −

- The most significant bit (MSB) of the Gray code is always equal to the MSB of the given Binary code.
- Other bits of the output Gray code can be obtained by XORing binary code bit at the index and previous index.

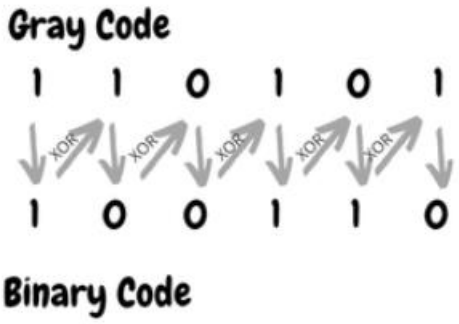


Gray to binary

Using Exclusive-Or (⊕) operation −

This is very simple method to get Binary number from Gray code. These are following steps for *n*-bit binary numbers −

- The Most Significant Bit (MSB) of the binary code is always equal to the MSB of the given binary number.
- Other bits of the output binary code can be obtained by checking gray code bit at that index. If current gray code bit is 0, then copy previous binary code bit, else copy invert of previous binary code bit.

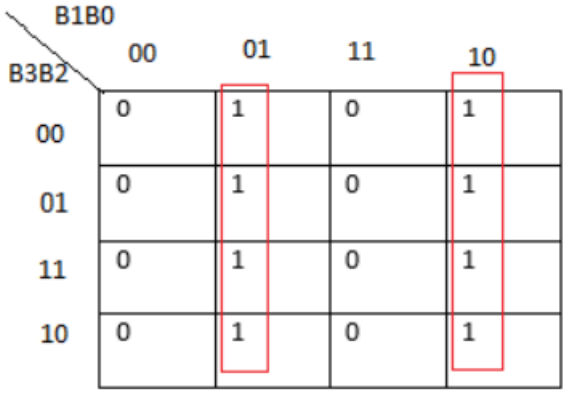


The truth table for the conversion is-

BINARY INPUT				GRAY CODE OUTPUT			
B3	B2	B1	B0	G3	G2	G1	G0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

K-Map Simplification

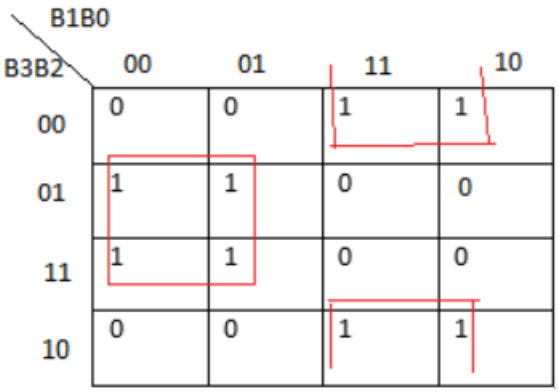
K – map for Go:



$G0 = B1\overline{B0} + \overline{B1}B0$

$= B1 \oplus B0$

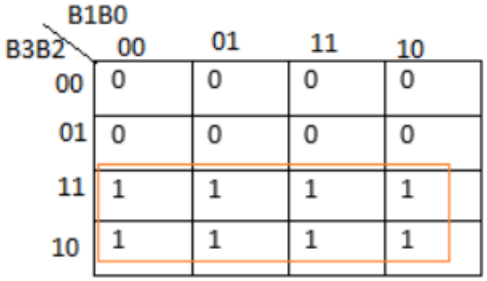
K- map for G1:



$G1 = \overline{B1}B2 + B1\overline{B2}$

$= B1 \oplus B2$

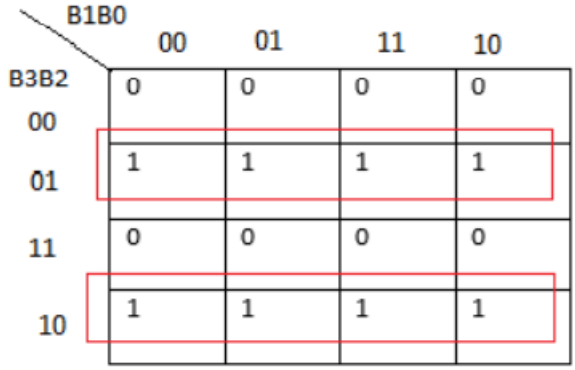
K- map for G2:



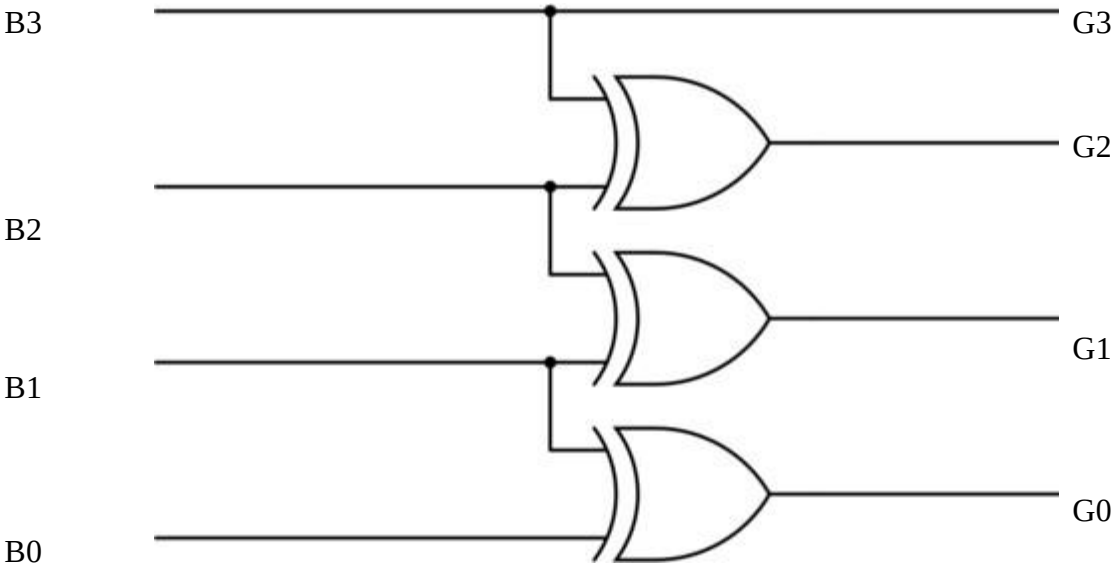
$G2 = \overline{B3}B2 + B3\overline{B2}$

$= B3 \oplus B2$

K- map for G3:



$G3 = B3$



Data Flow

```
`timescale 1 ns/1 ns
module binary_gray (b,g);
input [3:0]b;
output [3:0]g;
assign g[0] =b[1] ^ b[0];
assign g[1] =b[2] ^ b[1];
assign g[2]=b[3] ^ b[2];
assign g[3] = b[3];
endmodule
```

Structural Style

```
`timescale 1ns/1ns
module binary_gray (b,g);
input [3:0]b;
output [3:0]g;
xor A1( g[0],b[1], b[0]);
xor A2( g[1],b[2], b[1]);
xor A3( g[2],b[3], b[2]);
buf A4( g[3] , b[3]);
endmodule
```

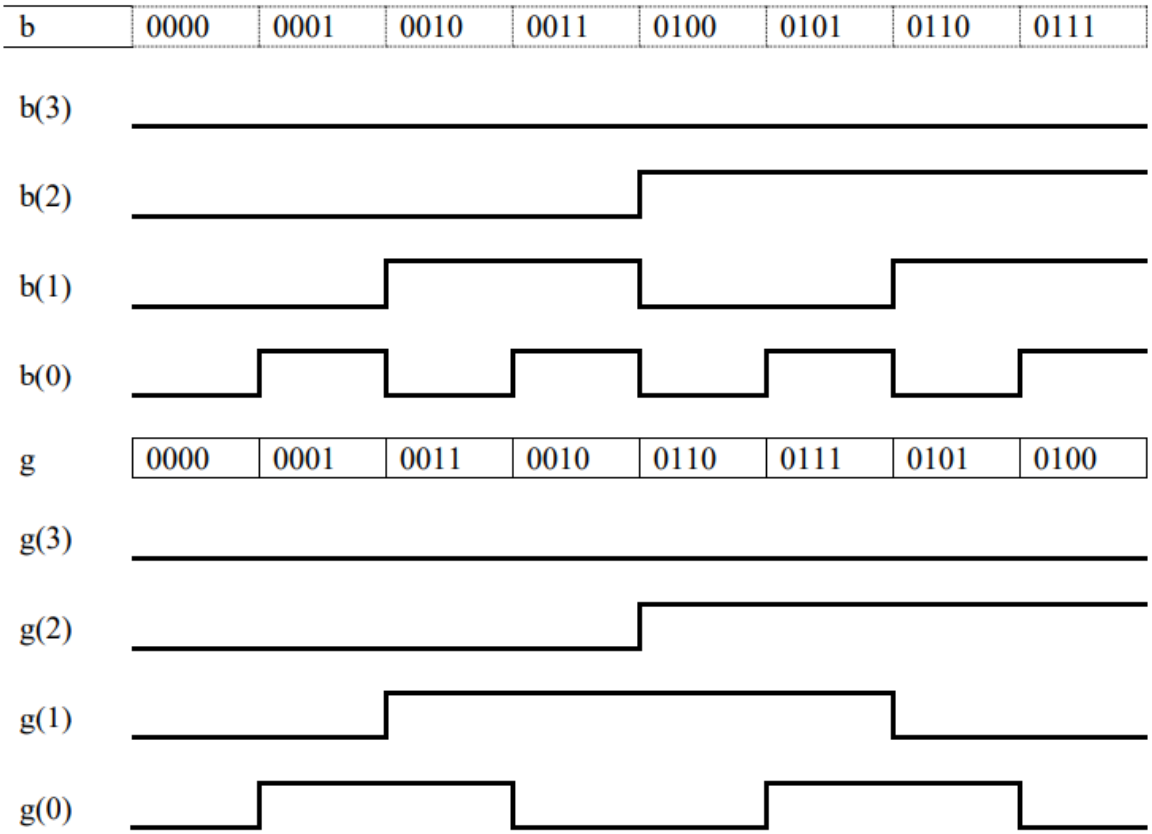
Behavioural Flow

```
`timescale 1ns/1ns
module binary_gray (b,g);
input [3:0]b;
output [3:0]g;
reg[3:0]g;
always@(b)
begin
g[0] =b[1] ^ b[0];
g[1] =b[2] ^ b[1];
g[2]=b[3] ^ b[2];
g[3] = b[3];
end
endmodule
```

Test Bench

```
`timescale 1ns/1ns
module binary_gray_tb;
reg [3:0]b;
wire [3:0]g;
binary_gray A1(.b(b) , .g(g));
initial
begin
b=4'b0000;
#100 b=4'b0001;
#100 b=4'b0010;
#100 b=4'b0011;
end
endmodule
```

Expected Waveform



Binary to Gray

Gray Code Input				Binary Code Output			
G3	G2	G1	G0	B3	B2	B1	B0
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

Table1

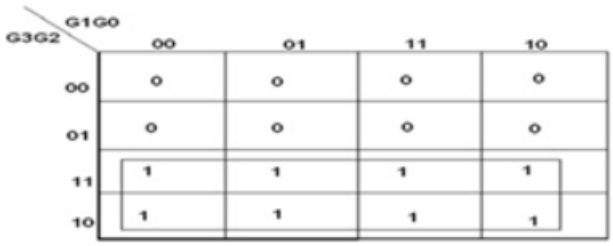


Fig6 (a): K – map for B_3

Simplified expression $B_3 = G_3$

For output B_2

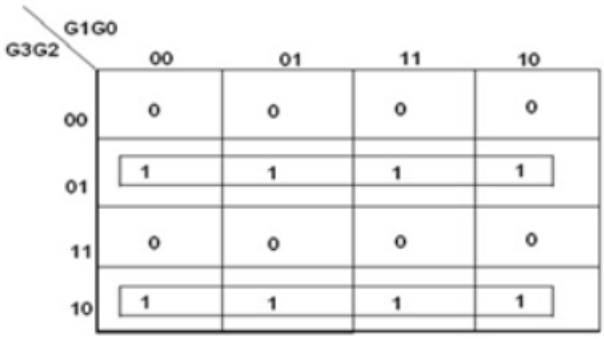


Fig6 (b): K – map for B_2

$B_(2) = G_(3)(G_2)\overline{}+ G_(2)(G_3)\overline{} = G_(3) \bigoplus G_(2)$

For output B_1

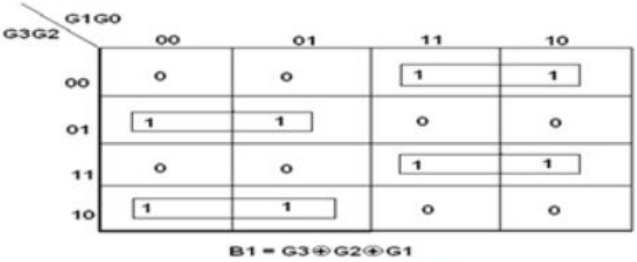


Fig6 (c): K – map for B_1

$B_1 = G_1\overline{G_2G_3} + G_2\overline{G_3G_1} + G_3G_2G_1 + G_3\overline{G_1G_2} = \overline{G_1}(G_3 \oplus G_2) + G_1(\overline{G_3 \oplus G_2})$

$= \overline{G_1}X + G_1\overline{X}$ where $X = G_3 \oplus G_2$

$= G_1 \oplus X = G_1 \oplus G_2 \oplus G_3$

For output B_0

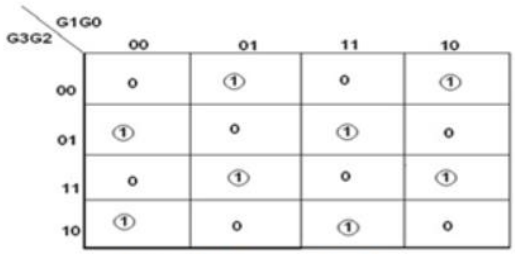


Fig6 (d): K – map for B_0

$$B_0 = G_0\overline{G_1G_2G_3} + G_2\overline{G_3G_1G_0} + G_1\overline{G_3G_2G_0} + G_1\overline{G_3G_2G_0} + G_2\overline{G_3}G_1G_0 + G_2\overline{G_1}G_3G_0$$
$$+ G_2\overline{G_0}G_3G_1 + G_3\overline{G_1G_2G_0} + G_0\overline{G_2}G_3G_1$$
$$= \overline{G_1G_0}(G_3 \oplus G_2) + G_0\overline{G_1}(\overline{G_3 \oplus G_2}) + G_0\overline{G_1}(\overline{G_3 \oplus G_2}) + G_1\overline{G_0}(\overline{G_3 \oplus G_2})$$
$$= (G_3 \oplus G_2) + (\overline{G_1 \oplus G_0}) + (G_1 \oplus G_0) + (\overline{G_3 \oplus G_2})$$
$$= \overline{Y}\,X + Y\overline{X} \text{ where } X = G_3 \oplus G_2 \text{ and } Y = G_1 \oplus G_0$$
$$= (G_2 \oplus G_3) \oplus (G_1 \oplus G_0)$$
$$= G_2 \oplus G_3 \oplus G_1 \oplus G_0$$

Step3: Realization

The Gray to Binary code converter is as shown in figure 7.

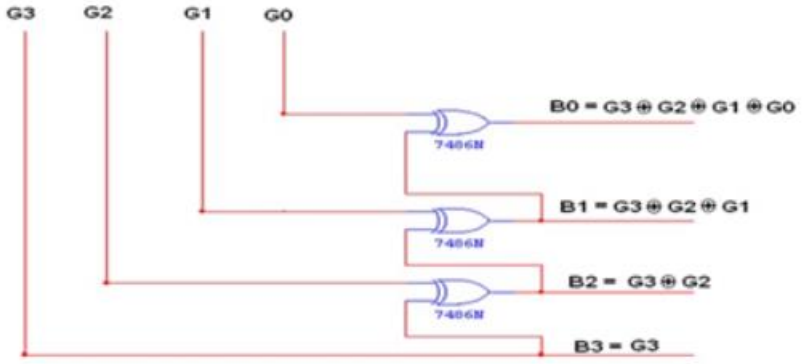


Fig7: Gray to binary code converter

Gate level Modelling

```
module gray_to_binary(g,b);
input [3:0]g;
output [3:0]b;
buf(b[3],g[3]);
xor x1 (b[2],g[2],g[3]);
xor x2(b[1],g[1],g[2],g[3]);
xor x3 (b3,g[0],g[1],g[2],g[3]);
endmodule
```

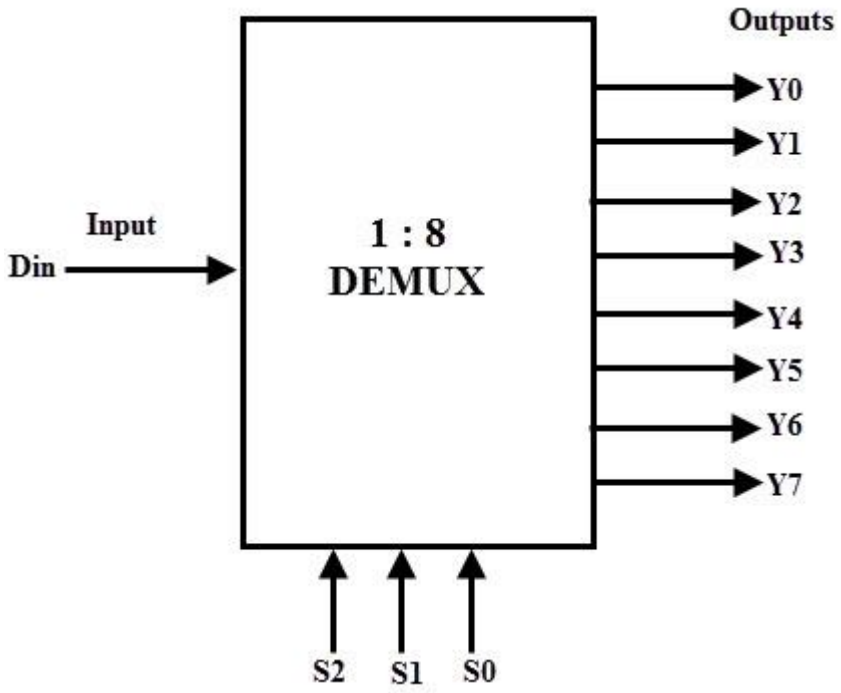
```
Dataflow
module gray_to_binary(g,b);
input [3:0]g;
output [3:0]b;

assign b[3]=g[3];
assign b[2]=g[2]^g[3];
assign b[1]= g[1]^g[2]^g[3];
assign b[0]= g[0]^g[1]^g[2]^g[3];
endmodule
```

```
Behavioral Modelling


module gray_to_binary(g,b);
input [3:0]g;
output reg [3:0]b;

always @(g)
begin
 b[3]=g[3];
b[2]=g[2]^g[3];
b[1]= g[1]^g[2]^g[3];
b[0]= g[0]^g[1]^g[2]^g[3];
end
endmodule
```

Testbench

```
module g_to_b_tb;
reg [3:0]g;
wire [3:0]b;

gray_to_binary uut(.g(g),.b(b));

initial begin

g= 4'b0000;
#10 g=4'b0001;
#10 g=4'b0011;
#10 g=4'b1111;
end
endmodule
```

5) 1: 8 Multiplexer



S2	S1	S0	O0	O1	O2	O3	O4	O5	O6	O7
0	0	0	I	0	0	0	0	0	0	0
0	0	1	0	I	0	0	0	0	0	0
0	1	0	0	0	I	0	0	0	0	0
0	1	1	0	0	0	I	0	0	0	0
1	0	0	0	0	0	0	I	0	0	0
1	0	1	0	0	0	0	0	I	0	0
1	1	0	0	0	0	0	0	0	I	0
1	1	1	0	0	0	0	0	0	0	I

Verilog Code :

```
module Demultiplexer(in,s0,s1,s2,d0,d1,d2,d3,d4,d5,d6,d7);
input in,s0,s1,s2;
output d0,d1,d2,d3,d4,d5,d6,d7;
assign d0=(in & ~s2 & ~s1 &~s0),
d1=(in & ~s2 & ~s1 &s0),
d2=(in & ~s2 & s1 &~s0),
d3=(in & ~s2 & s1 &s0),
d4=(in & s2 & ~s1 &~s0),
d5=(in & s2 & ~s1 &s0),
d6=(in & s2 & s1 &~s0),
d7=(in & s2 & s1 &s0);
endmodule
```

Test Bench

Test Bench

`timescale 1ns / 1ps

// Inputs

reg in;

reg s0;

reg s1;

reg s2;

// Outputs

wire d0;

wire d1;

wire d2;

wire d3;

wire d4;

wire d5;

wire d6;

wire d7;

```
// Instantiate the Unit Under Test (UUT)
Demultiplexer uut (
.in(in),
.s0(s0),
.s1(s1),
.s2(s2),
.d0(d0),
.d1(d1),
.d2(d2),
.d3(d3),
.d4(d4),
.d5(d5),
.d6(d6),
.d7(d7)
);
initial begin
// Initialize Inputs
in = 0;
s0 = 0;
s1 = 0;
s2 = 0;
// Wait 100 ns for global reset to finish
#100;
in = 1;
s0 = 0;
s1 = 1;
s2 = 0;
// Wait 100 ns for global reset to finish
```

#100;

// Add stimulus here

end

endmodule

6) 2 BIT COMPARATOR

Inputs				Outputs		
A_1	A_0	B_1	B_0	A>B	A=B	A<B
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

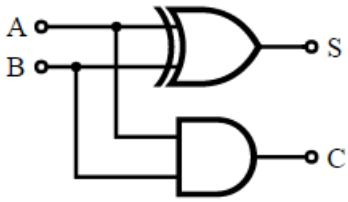
```
module comparator(input [1:0] A,B, output A_less_B,
A_equal_B, A_greater_B);
 wire tmp1,tmp2,tmp3,tmp4,tmp5, tmp6, tmp7, tmp8;
 // A = B output
 xnor u1(tmp1,A[1],B[1]);
 xnor u2(tmp2,A[0],B[0]);
 and u3(A_equal_B,tmp1,tmp2);
 // A less than B output
 assign tmp3 = (~A[0])& (~A[1])& B[0];
 assign tmp4 = (~A[1])& B[1];
 assign tmp5 = (~A[0])& B[1]& B[0];
 assign A_less_B = tmp3 | tmp4 | tmp5;
 // A greater than B output
```

```
assign tmp6 = (~B[0])& (~B[1])& A[0];
assign tmp7 = (~B[1])& A[1];
assign tmp8 = (~B[0])& A[1]& A[0];
assign A_greater_B = tmp6 | tmp7 | tmp8;
endmodule
`timescale     10 ps/ 10 ps
// FPGA projects using Verilog/ VHDL
// fpga4student.com
// Verilog testbench code for 2-bit comparator
module tb_comparator;
reg [1:0] A, B;
wire A_less_B, A_equal_B, A_greater_B;
integer i;
// device under test
comparator dut(A,B,A_less_B, A_equal_B, A_greater_B);
initial begin
     for (i=0;i<4;i=i+1)
     begin
          A = i;
          B = i + 1;
          #20;
     end
     for (i=0;i<4;i=i+1)
     begin
          A = i;
          B = i;
          #20;
     end
     for (i=0;i<4;i=i+1)
     begin
          A = i+1;
          B = i;
          #20;
     end
end
endmodule
```

Experiment 3

Write an HDL code to describe the functions of a Half Adder and full adder

Half Adder Truth Table			
A	**B**	**Carry**	**Sum**
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0



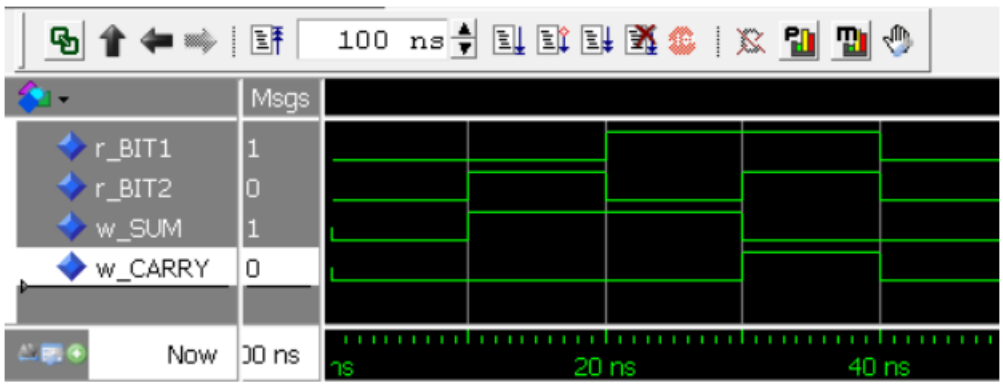
Half-Adder Schematic - From Wikipedia

```
//////////////////////////////////////////////////////////////////////////////////
// File Downloaded from http://www.nandland.com
//////////////////////////////////////////////////////////////////////////////////
module half_adder
  (
   i_bit1,
   i_bit2,
   o_sum,
   o_carry
   );

  input  i_bit1;
  input  i_bit2;
  output o_sum;
  output o_carry;

  assign o_sum   = i_bit1 ^ i_bit2;  // bitwise xor
  assign o_carry = i_bit1 & i_bit2;  // bitwise and

endmodule // half_adder
```

Test bench

```
/ File Downloaded from http://www.nandland.com
//////////////////////////////////////////////////////////////////////////////////
`include "half_adder.v"

module half_adder_tb;

  reg r_BIT1 = 0;
  reg r_BIT2 = 0;
  wire w_SUM;
```

```
  wire w_CARRY;

  half_adder half_adder_inst
    (
     .i_bit1(r_BIT1),
     .i_bit2(r_BIT2),
     .o_sum(w_SUM),
     .o_carry(w_CARRY)
     );

  initial
    begin
      r_BIT1 = 1'b0;
      r_BIT2 = 1'b0;
      #10;
      r_BIT1 = 1'b0;
      r_BIT2 = 1'b1;
      #10;
      r_BIT1 = 1'b1;
      r_BIT2 = 1'b0;
      #10;
      r_BIT1 = 1'b1;
      r_BIT2 = 1'b1;
      #10;
    end

endmodule // half_adder_tb
```



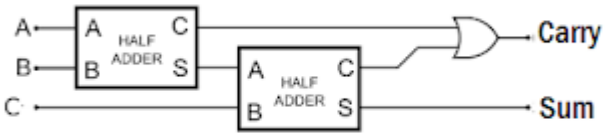
Modelsim Simulation of Half Adder

Full adder using half adder

Full adder Truth table

A	B	C	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

A Full adder can be implemented using half adders as shown below:



The Verilog code can be written in structural modelling for the above circuit.

```
//declare the Full adder verilog module.
module full_adder(
    Data_in_A,  //input A
    Data_in_B,  //input B
    Data_in_C,  //input C
     Data_out_Sum,
     Data_out_Carry
    );

    //what are the input ports.
    input Data_in_A;
    input Data_in_B;
     input Data_in_C;
    //What are the output ports.
    output Data_out_Sum;
     output Data_out_Carry;
     //Internal variables
     wire ha1_sum;
     wire ha2_sum;
     wire ha1_carry;
     wire ha2_carry;
     wire Data_out_Sum;
     wire Data_out_Carry;

     //Instantiate the half adder 1
    half_adder  ha1(
        .Data_in_A(Data_in_A),
        .Data_in_B(Data_in_B),
        .Data_out_Sum(ha1_sum),
        .Data_out_Carry(ha1_carry)
```

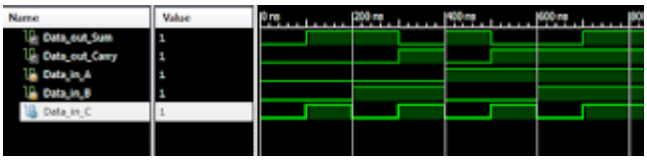
```
    );

    //Instantiate the half adder 2
    half_adder  ha2(
        .Data_in_A(Data_in_C),
        .Data_in_B(ha1_sum),
        .Data_out_Sum(ha2_sum),
        .Data_out_Carry(ha2_carry)
    );

    //sum output from 2nd half adder is connected to full adder output
    assign Data_out_Sum = ha2_sum;
    //The carry's from both the half adders are OR'ed to get the final
carry./
    assign Data_out_Carry = ha1_carry | ha2_carry;

endmodule
```

Testbench Code for Full Adder:

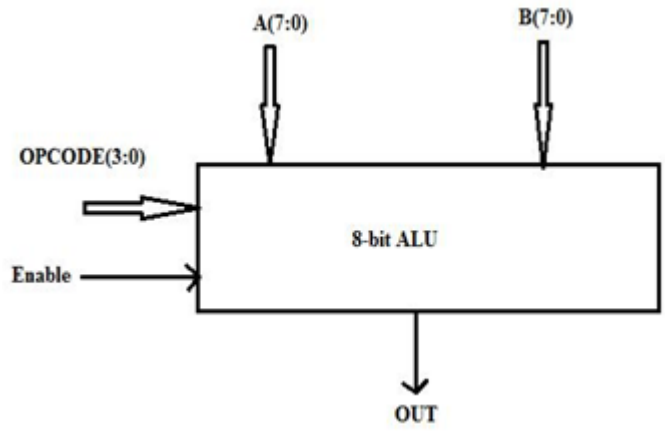
```
module tb_fullAdd;

    // Inputs
    reg Data_in_A;
    reg Data_in_B;
    reg Data_in_C;

    // Outputs
    wire Data_out_Sum;
    wire Data_out_Carry;

    // Instantiate the Unit Under Test (UUT)
    full_adder uut (
        .Data_in_A(Data_in_A),
        .Data_in_B(Data_in_B),
        .Data_in_C(Data_in_C),
        .Data_out_Sum(Data_out_Sum),
        .Data_out_Carry(Data_out_Carry)
    );

    initial begin
        //Apply inputs. 8 combinations of inputs are possible.
        //They are given below.
        Data_in_A = 0;  Data_in_B = 0;  Data_in_C = 0;  #100;
        Data_in_A = 0;  Data_in_B = 0;  Data_in_C = 1;  #100;
        Data_in_A = 0;  Data_in_B = 1;  Data_in_C = 0;  #100;
        Data_in_A = 0;  Data_in_B = 1;  Data_in_C = 1;  #100;
        Data_in_A = 1;  Data_in_B = 0;  Data_in_C = 0;  #100;
        Data_in_A = 1;  Data_in_B = 0;  Data_in_C = 1;  #100;
        Data_in_A = 1;  Data_in_B = 1;  Data_in_C = 0;  #100;
        Data_in_A = 1;  Data_in_B = 1;  Data_in_C = 1;  #100;
    end

endmodule
```

Simulated Waveform:

The code was synthesised and simulated in Xilinx ISE 13.1. The waveform is shown below:



Experiment 4
Write a model for n bit ALU using the schematic diagram shown below



AIM:To design a verilog HDL for designing ALU using behaviral model.TOP MODULE

```
`timescale 1ns/1ns
module alu_32(en,opcode,a,b,y);

input en;
input [31:0]a,b; input
[2:0]opcode; output reg
[32:0]y;always@(*)
begin if(en==0)
y=33'bz; else
begin
case       (opcode)
3'b000:y=a+b;
3'b001:y=a/b;

3'b010:y=a&b;
3'b011:y=a+b;

3'b100:y=a-b;

3'b101:y=a*b;
3'b110:y=a^b;
3'b111:y=`a;

endcase


endmodule
```

Experiment 5

Develop the HDL Code for the following flip-flops: SR, D, JK, and T

AIM:To design a verilog HDL for designing j-k flip flop.

TOP MODULE

```
`timescale 1ns/1ns
module jkff(j,k,clk,rst,q,nq);
input j,k,clk,rst;
output reg q,nq;
always @(posedge clk,rst) begin
if(reset)
q=1'b0;
else
begin
case({j,k})
2'b00:begin q=q;
2'b01:begin q=0;
2'b10:begin q=1;
2'b11:begin q=~q;
end
endcase
nq=~q;
end
endmodule
```

TEST BENCH

```
`timescale 1ns/1ns
module jkff_tb;
reg j,k,clk,rst;
wire q,nq;
jkff uut(.j(j),.k(k),.clk(clk),.rst(rst),.q(q),.nq(nq));
initial begin
rst=1;
j=0;k=0;
#100 rst=0;
#100 k=1;
#100 j=1;k=0;
#100 k=1;
end
endmodule
```

for SR FF 2'b11;q=1'bz; rest all is same. change the input and output variables name as SR. rest all same

AIM:To design a verilog HDL for designing t-flip flop

TOP MODULE

```
`timescale 1ns/1ns
module tff(t,clk,rst,q,nq);input
t,clk,rst,;
output reg q,nq; always
@(posedge clk)begin
if(rst)
q=1'b0;
else begin
case(t)
1'b0:q=0;
1'b1:q=~q;
endcase end
endmodule
```

TEST BENCH

```
`timescale 1ns/1nsmodule
tff_tb;
reg t,clk,rst;wire
q,nq;
tff uut(.t(t), .clk(clk) .rst(rst),.q(q),.nq(nq));initial begin
rst=1;
#100   rst=0:t=0;
#100 rst=0;t=1;
end endmodule
```

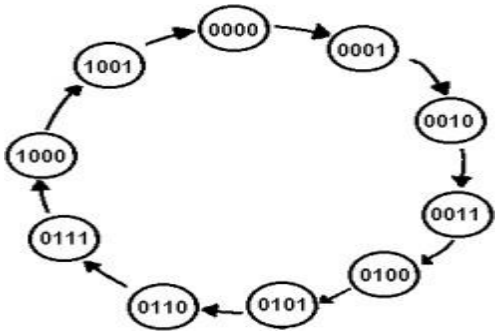
For D FF 1'b1:q=1; rest all same

Experiment 6

Synchronous and Asynchronous counterSynchronous BCD

(Decade)Counter

A synchronous decade counter will count from zero to nine and repeat the sequence. The state diagramof this counter is shown in Fig

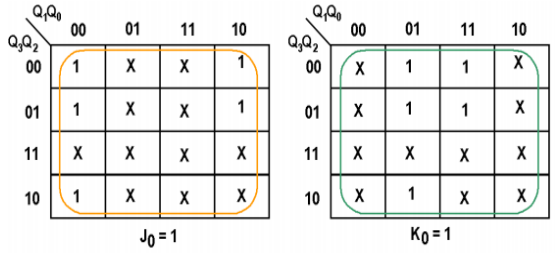


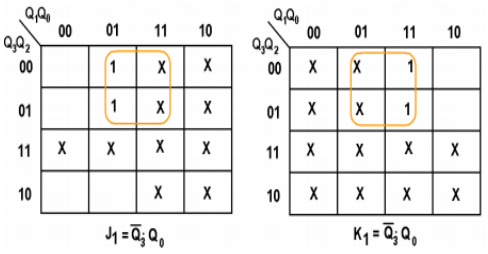
J	K	Q	Q+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

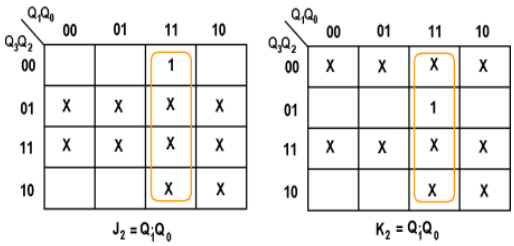
Characteristic Table

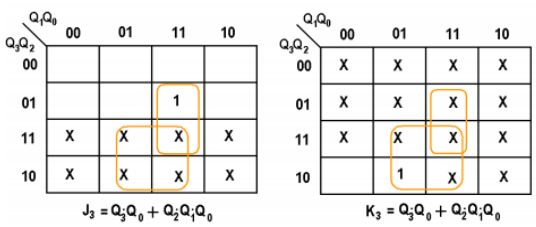
Q	Q+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

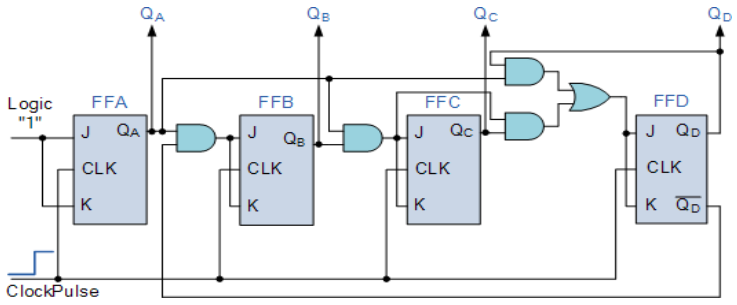
Present State				Next State				Output							
Q_3	Q_2	Q_1	Q_0	Q_3	Q_2	Q_1	Q_0	J_3	K_3	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	0	0	0	1	0	X	0	X	0	X	1	X
0	0	0	1	0	0	1	0	0	X	0	X	1	X	X	1
0	0	1	0	0	0	1	1	0	X	0	X	X	0	1	X
0	0	1	1	0	1	0	0	0	X	1	X	X	1	X	1
0	1	0	0	0	1	0	1	0	X	X	0	0	X	1	X
0	1	0	1	0	1	1	0	0	X	X	0	1	X	X	1
0	1	1	0	0	1	1	1	0	X	X	0	X	0	1	X
0	1	1	1	1	0	0	0	1	X	X	1	X	1	X	1
1	0	0	0	1	0	0	1	X	0	0	X	0	X	1	X
1	0	0	1	0	0	0	0	X	1	0	X	0	X	X	1


$J_0 = 1$
$K_0 = 1$


$J_1 = \overline{Q}_3 Q_0$
$K_1 = \overline{Q}_3 Q_0$


$J_2 = Q_1 Q_0$
$K_2 = Q_1 Q_0$

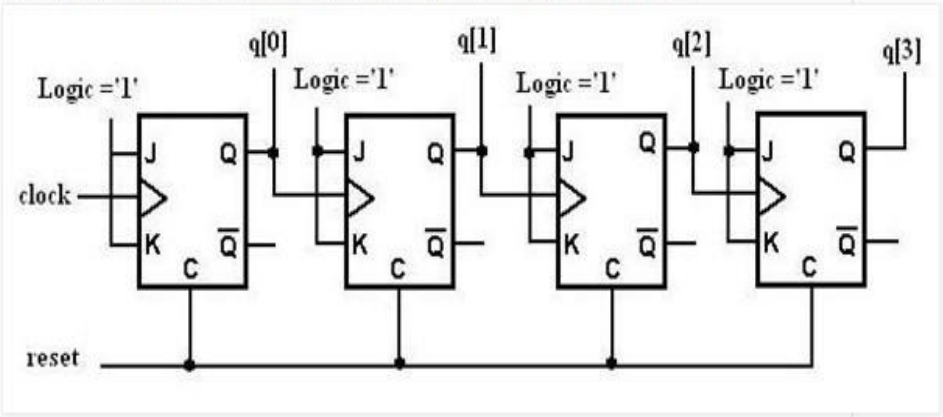

$J_3 = Q_3 Q_0 + Q_2 Q_1 Q_0$
$K_3 = Q_3 Q_0 + Q_2 Q_1 Q_0$


Q_A
Q_B
Q_C
Q_D
Logic "1"
FFA
FFB
FFC
FFD
J
CLK
K
ClockPulse

```
module jkfflop(input  J, K , clk ,rst, output reg Q);
  always @(negedge clk) begin
    if(rst)
      Q <= 1'b0;
    else begin
    case({J,K})
      2'b00 : Q <= Q   ;
      2'b01 : Q <= 1'b0;
      2'b10 : Q <= 1'b1;
      2'b11 : Q <= ~Q  ;
    endcase
  end
  end
endmodule


module BCD_Counter(input clk, rst, output [3:0] Q);

  jkfflop first(1'b1 , 1'b1, clk ,rst, Q[0]);

  jkfflop second(~Q[3],~Q[3],Q[0],rst, Q[1]);

  jkfflop third(1'b1 , 1'b1 , Q[1] ,rst,Q[2]);

  jkfflop fourth(Q[1]&Q[2] , Q[3] , Q[0] ,rst, Q[3]);

endmodule


module TB;
  reg CLK = 0, rst = 1;
  wire[3:0] Q;
  BCD_Counter UUT(CLK ,rst, Q);

  initial repeat(40) #50 CLK=~CLK;
  initial #70 rst = 0;

endmodule
```

4-bit Asynchronous up counter using JK-FF (Structural model)

Circuit Diagram for 4-bit Asynchronous up counter using JK-FF:



4-bit Asynchronous up counter using JK-FF (Structural model):

```
module async_count(j,k,clock,reset,q,qb);
input j,k,clock,reset;
output wire [3:0]q,qb;

jkff JK1(j,k,clock,reset,q[0],qb[0]);
jkff JK2(j,k,q[0],reset,q[1],qb[1]);
jkff JK3(j,k,q[1],reset,q[2],qb[2]);
jkff JK4(j,k,q[2],reset,q[3],qb[3]);

endmodule
```

Test bench

```
module async_tb;

reg  j,k,clock,reset;

wire [3:0]q,qb;

async_count    uut (j,k,clock,reset,q,qb);

initial begin

reset=1;

#10 reset=0;
```

```
j=4'b1111;

k=4'b1111;

end

endmodule
```