DIGITAL SYSTEM DESIGN (USING VERILOG) E-Journal

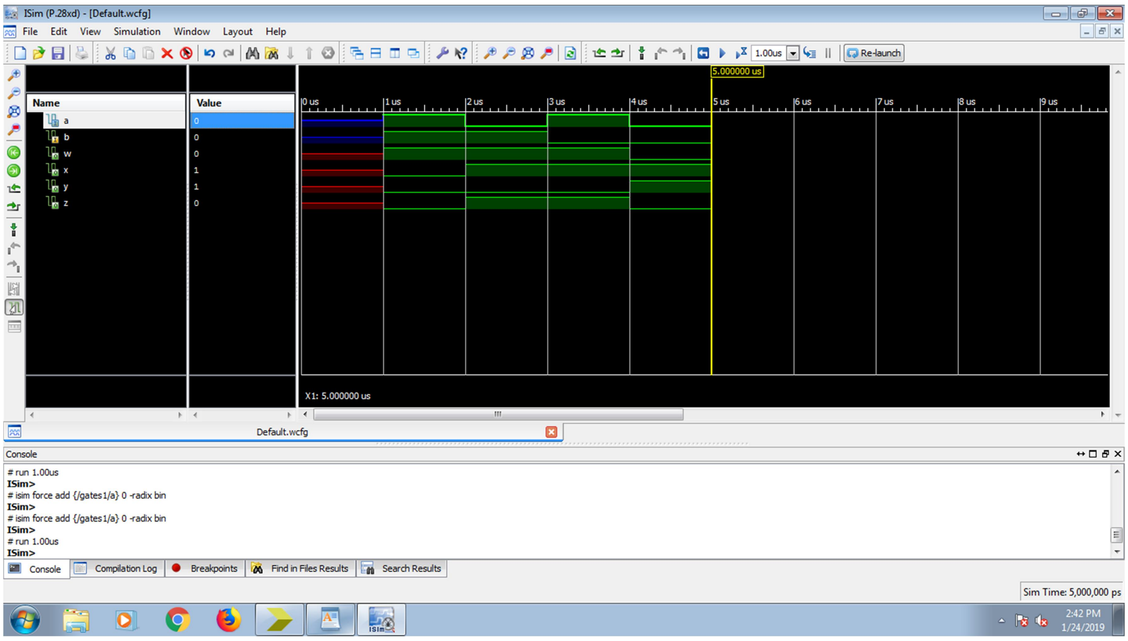
Submitted by:

Keerthi G

103 B (B1 batch)

Implementation of OR gate, NAND gate, NOR gate, XOR gate

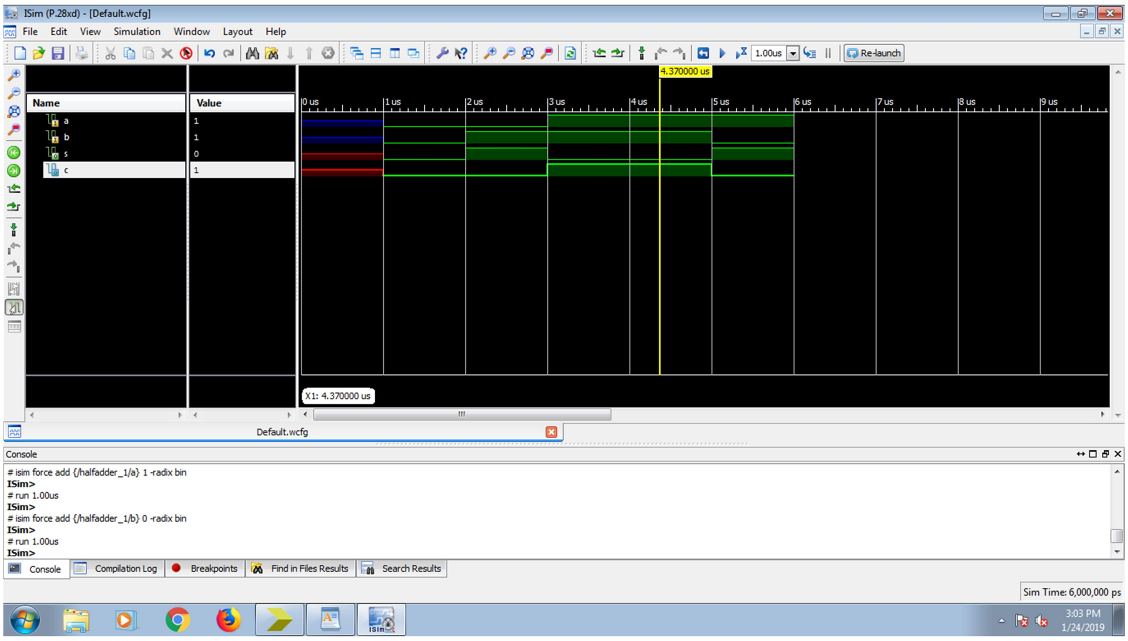
```
module gates(

input a,

input b,

output y,

output x,

output z,

output w);

assign y=a|b;

assign x=~(a&b);

assign w=~( a|b);

assign z=a^b;

endmodule
```



Half adder

```
module halfadder(

input a,
```

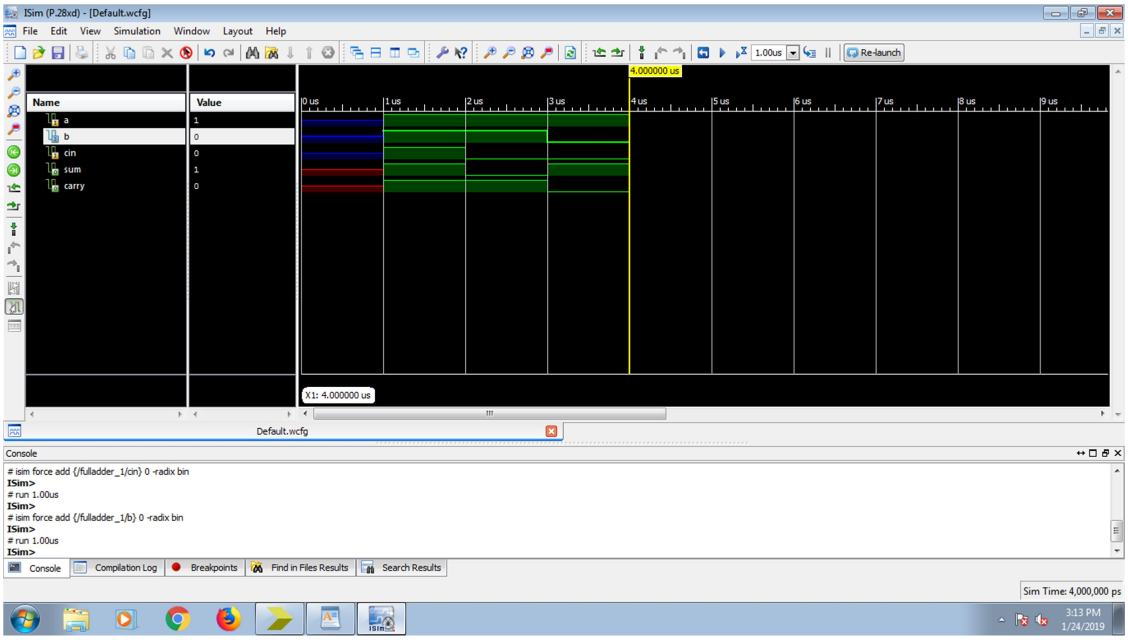
```
input b,

output s,

output c,

);

assign s=a^b;

assign c=a&b;

endmodule
```



Full Adder

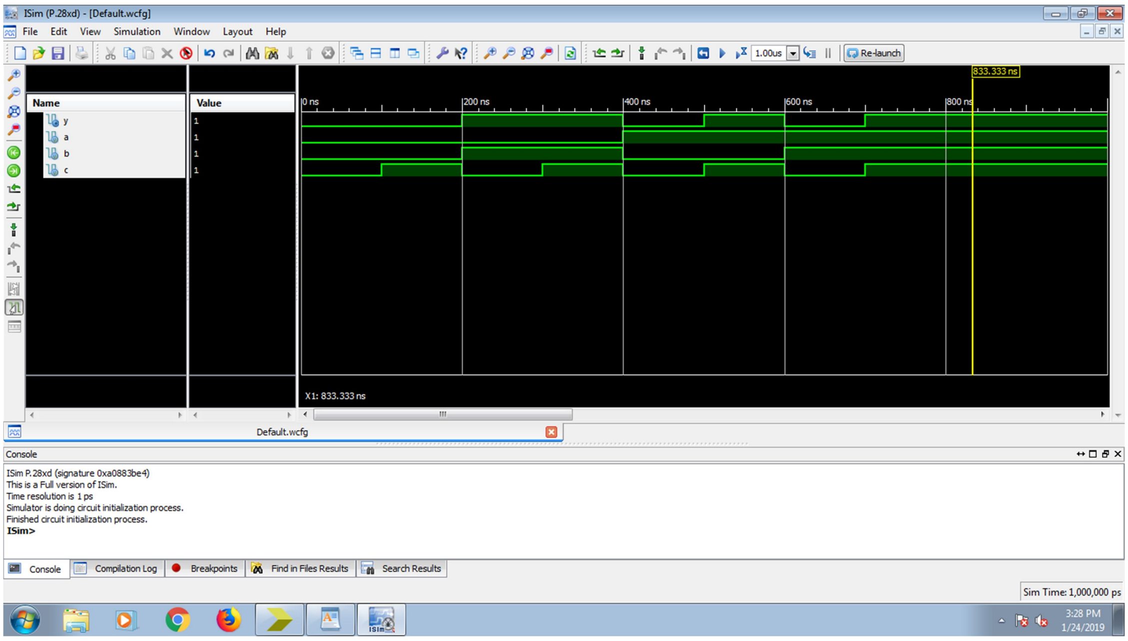
```
module fulladder(

input a,

input b,

input ci,

output s,

output co);

assign s= a^b^ci;

assign co=(a&b)|(b&ci) |(ci&a);
```

endmodule



2:1 mux

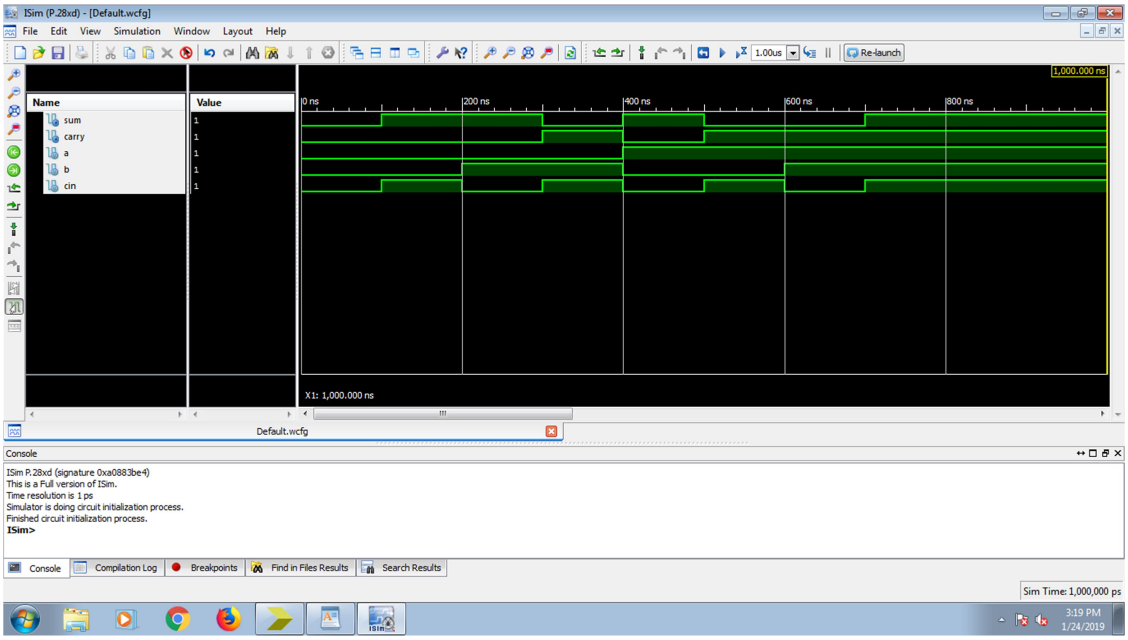
```
module mux(

input s;

input a;

input b;

output z);

assign z=((~s)&a) |(s&b) ;

endmodule
```



4 bit Carry look ahead adder

```
module 4bitcla(

output [3:0]s,

output cout,pg,gg,

input [3:0]a,b,

input cin

);

wire[3:0]g,p,c;

assign g=a&b;

assign p=a^b;

assign c[0]=cin;

assign c[1]=g[0] |(p[0]&c[0]);

assign c[2]=g[1] |(p[1]&g[0])|(p[1]&p[0]&c[0]);

assign c[3]=g[2] |(p[2]&g[1])|(p[2]&p[1]&g[0]) | (p[2]&p[1]&p[0]&c[0]);

assign cout=g[3] |(p[3]&g[2])|(p[3]&p[2]&g[1]) | (p[3]&p[2]&p[1]&g[0]) | (p[3]&p[2]&p[1]&p[0]&c[0]);

assign s=p^c;
```

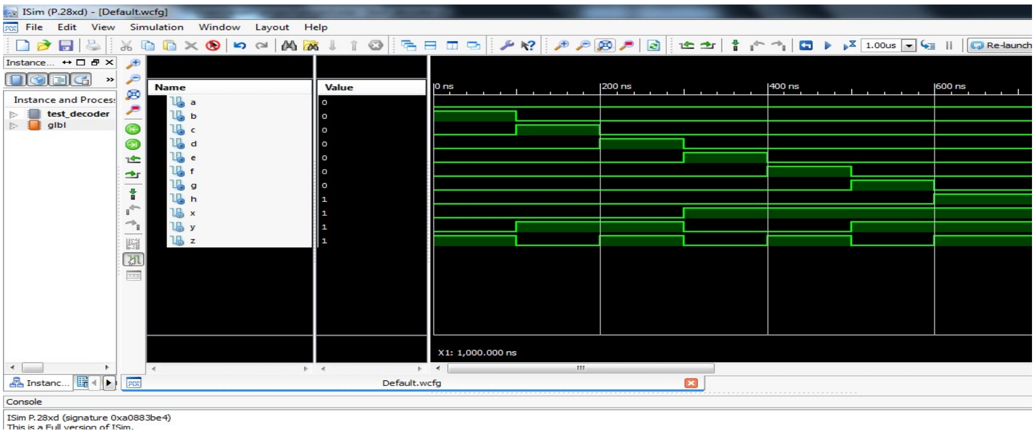
```
assign pg= p[3]&p[2]&p[1]&p[0];

assign gg= g[3] |(p[3]&g[2])|(p[3]&p[2]&g[1]) | (p[3]&p[2]&p[1]&g[0]);

endmodule
```



Decoder 3:8:

```
module decoder3_8(

   input x,

   input y,

   input z,

   output a,

   output b,

   output c,

   output d,

   output e,

   output f,

   output g,

   output h );

         assign a=(~x)&(~y)&(~z);
```

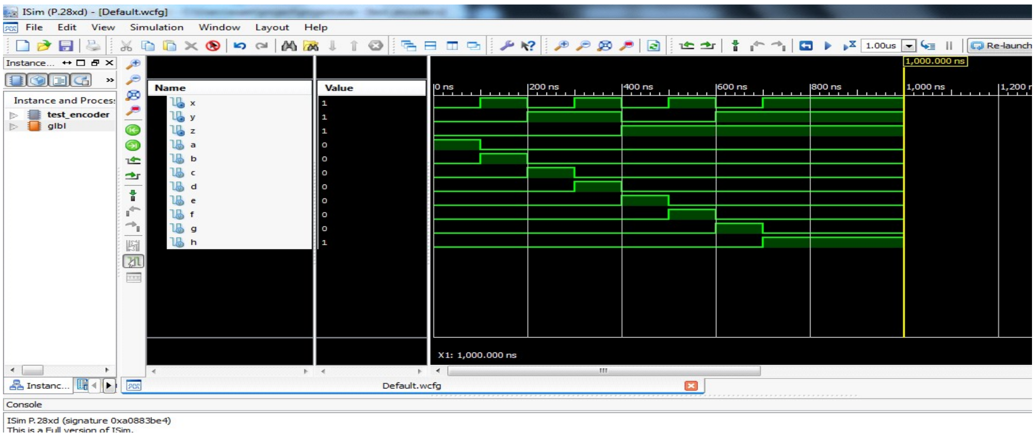
```
    assign b=(~x)&(~y)&(z);
    assign c=(~x)&(y)&(~z);
    assign d=(~x)&(y)&(z);
    assign e=(x)&(~y)&(~z);
    assign f=(x)&(~y)&(z);
    assign g=(x)&(y)&(~z);
    assign h=(x)&(y)&(z);
endmodule
```



Encoder 3:8:

```
module encoder3_8(
  output x,
  output y,
  output z,
  input a,
  input b,
  input c,
  input d,
  input e,
  input f,
  input g,
  input h);
```

```
assign x = b|d|f|h;

assign y = h|g|d|c;

assign z = h|g|e|f;

endmodule
```



Gray to binary Conversion (4 bit):

```
module gray_binary(
   input g0,
   input g1,
   input g2,
   input g3,
   output b0,
   output b1,
   output b2,
   output b3);
assign b0 = g0^g1^g2^g3;
assign b1 = g1^g2^g3;
assign b2 = g2^g3;
assign b3 = g3;
endmodule
```

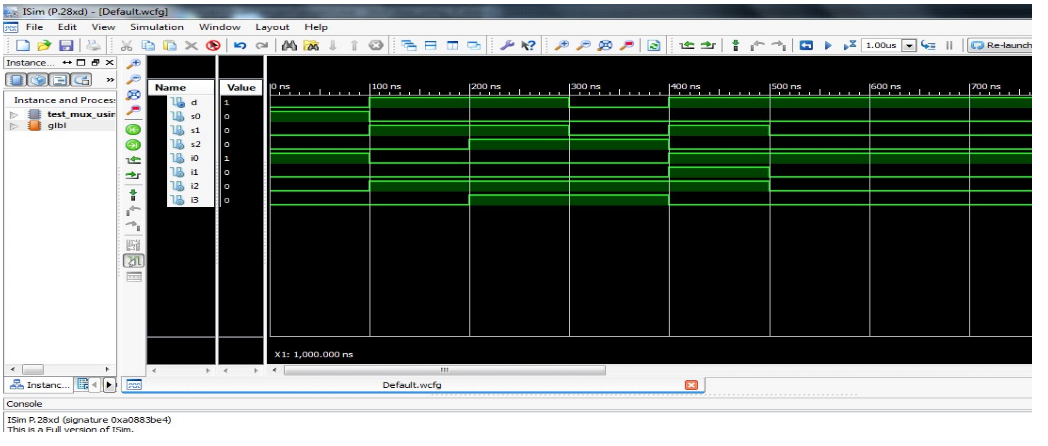
Binary to gray (4 bit):

```
module binary_gray(
   input b1,
   input b2,
   input b3,
   input b0,
   output g0,
   output g1,
   output g2,
   output g3 );
assign g0 = b1^b0;
assign g1 = b2^b1;
assign g2 = b3^b2;
assign g3 = b3;
endmodule
```

Mux using if else…

```
module mux_using_if_else(s0,s1,s2,i0,i1,i2,i3,d );
input s0,s1,s2,i1,i2,i3,i0;
output reg d;
always@(s0,s1,s2)
begin
if(s1)
        begin
                if(s0)
                        d=i3;
                else
                        d=i2;
        end
```

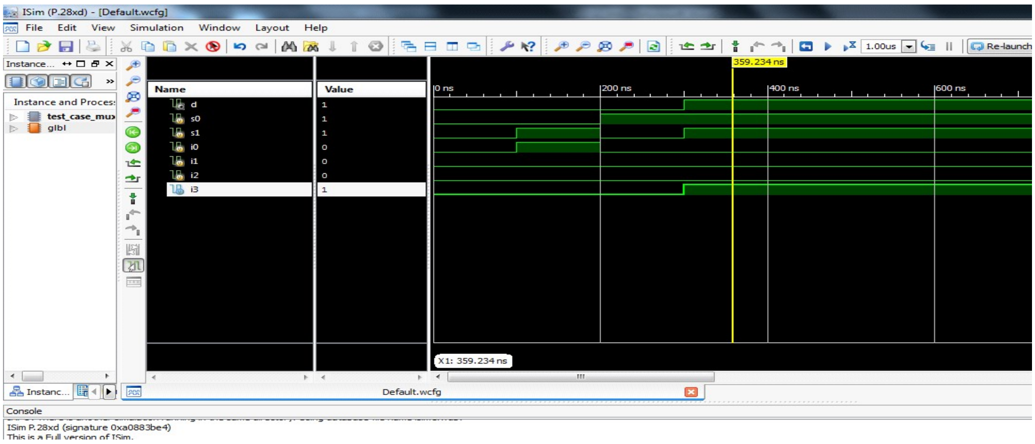
```
else
        begin
                if(s0)
                        d=i1;
                else
                        d=i0;
        end
end
```



Mux using case:

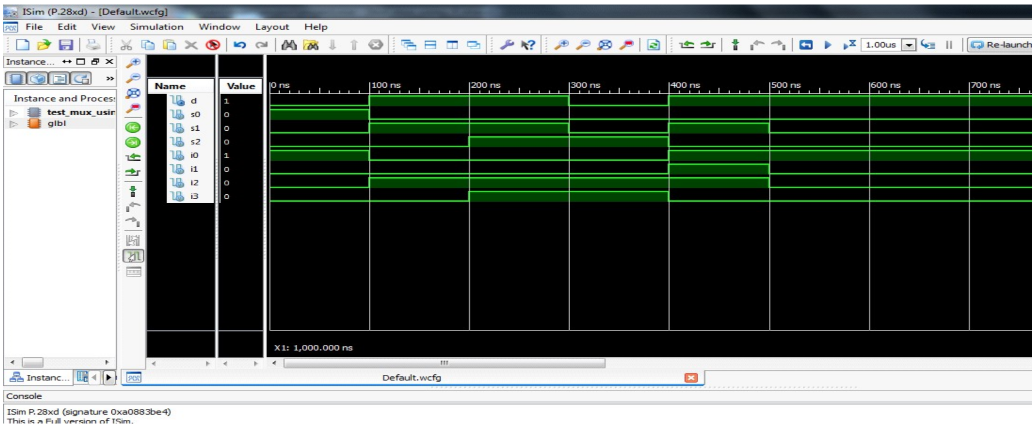
```
module mux_case(s0,s1,i0,i1,i2,i3,d );
input s0,s1,i1,i2,i3,i0;
output reg d;
always@(s0,s1)
begin
case({s0,s1})
        2'b0_0:d=i0;
        2'b0_1:d=i1;
        2'b1_0:d=i2;
        2'b1_1:d=i3;
```

```
endcase

end

endmodule
```



JK flip flop using case:

```
module jk_using_case(j,k,Q,clk);

input j,k,clk;

output reg Q;

always@(posedge clk)

begin

case({j,k})

2'b0_0:Q<=Q;

2'b0_1:Q<=1'b0;

2'b1_0:Q<=1'b1;

2'b1_1:Q<=~Q;

endcase

end

endmodule
```



8 bit booth multiplier

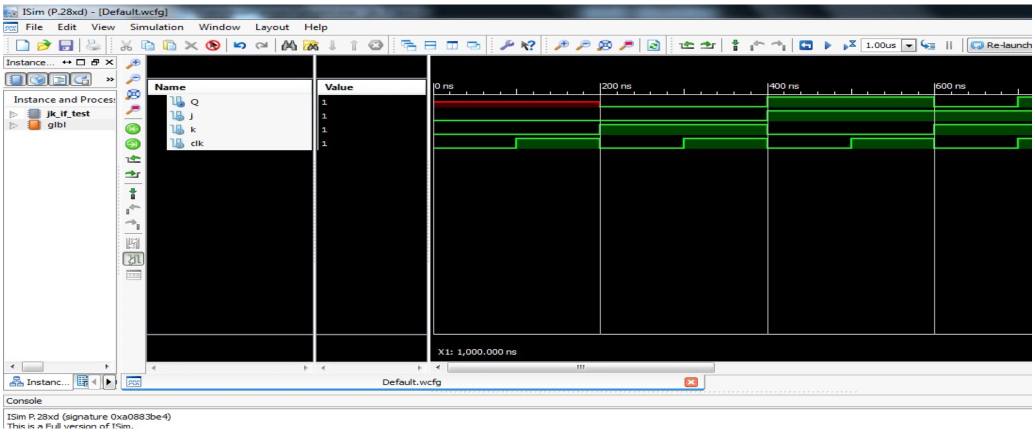
```
module booth_multiplier(x, y, z);
input signed [7:0] x, y;
output signed [31:0] z;
      reg signed [31:0] z;
      reg [1:0] temp;
      integer i;
      reg e1;
      reg [7:0] y1;
      always @ (x, y)
      begin
      z = 31'd0;
      e1 = 1'd0;
      for (i = 0; i < 4; i = i + 1)
      begin
      temp = {x[i], e1};
      y1 = - y;
  case (temp)
```

```
      2'd2 : z [31 : 15] = z [31 : 15] + y1;
      2'd1 : z [31: 15] = z [31 : 15] + y;
      default : begin end
      endcase
      z = z >> 1;
      z[31] = z[30];
      e1 = x[i];
         end
      if (y == 16'd32)
  begin
            z = - z;
         end
      end
endmodule
```

JK using if else.

```
module jk_using_ifelse(j,k,clk,Q);
input j,k,clk;
output reg Q;
always@(j,k,clk)
begin
if(j)
begin
if(k)
Q=~Q;
else
Q=1;
```

```
end
else
begin
if(k)
Q=0;
else
Q=Q;
end
end
endmodule
```



D, S-R,T flip-flop

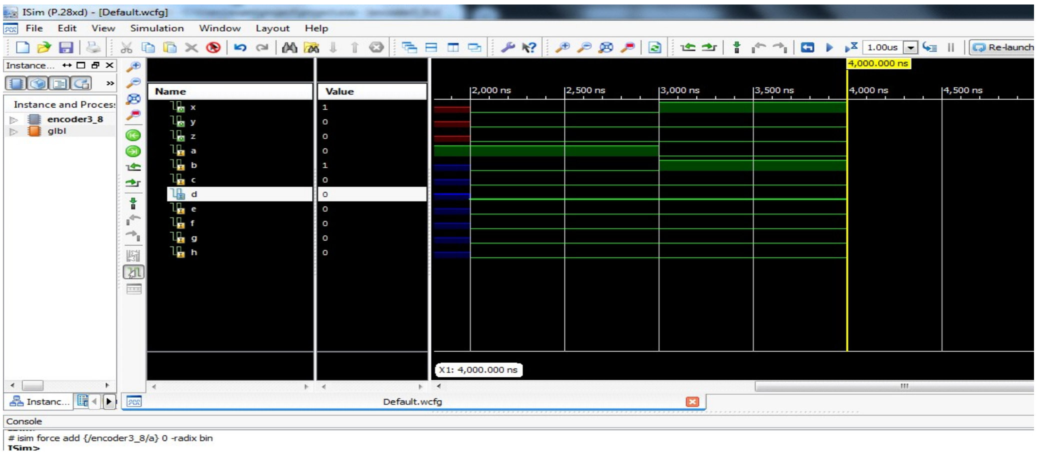
```
module sscdt(
        input D,
        input S,
        input R,
        input T,
        output reg Q1,
        output reg Q2,
        output reg Q3,
);
```

```
always@( D, S ,R ,T)
begin
case({D})
	1'b0 : Q1<=1'b0;
	1'b1 : Q1<=1'b1;
endcase
case({S,R})
2'b00 : Q2<=Q2;
2'b00 : Q2<=1'b0;
2,b10 : Q2<=1'b1;
2'b11 : Q2<=1'b2;
endcase
case ({T}
	1'b0 : Q3<=Q3;
	1'b1 : Q3<=~Q3;
endcase
end
endmodule
```

Trailing zeroes:

```
module trailing_zeroes_using_while(output reg [3:0]out,input  [7:0]data);
integer i;
always@(data)
begin
	out=0;
	i=0;
	while(data[i]==0&&i<=7)
		begin
			out=out+1;
```

```
                    i=i+1;
          end
end
endmodule
```



Upcounter using case:

```
module upcounter_case(input clk,clr,output reg [2:0]data);
initial
data = 3'b0_0_0;
always@(posedge clk)
begin
if(clr == 0)
     begin
          case(data)
                    3'd0:data=3'd1;
                    3'd1:data=3'd2;
                    3'd2:data=3'd3;
                    3'd3:data=3'd4;
                    3'd4:data=3'd5;
                    3'd5:data=3'd6;
```

```
                        3'd6:data=3'd7;
                        3'd7:data=3'd0;
                endcase
        end
else
         data=3'b000;
end
```

BCD UP/DOWN counter U/D:

```
module updowncounter(input ud,
        input qa,
        input qb,
        output reg j1,
        output reg j2,
        output reg k1,
        output reg k2,
);
always@(ud, qa, qb)
begin
        j1=ud^qb;
        j2=1’b1;
        k1=ud^qb;
        k2=1’b1;
end
endmodule
```

4 bit asynchronous counter

```
module acounter(clk,count);
input clk;
```

```
output[3:0] count;
reg[3:0]count;
wire clk;
initial
        coutt=4'b0;
always@(negedge clk)
        count[0]<= ~count[0];
always@(negedge count[0])
        coumt[1]<= ~count[1]
always@(negedge count[1])
        count[2]<= ~count[2];
always@(negedge count[2])
        count[3]<= ~count[3];
endmodule
```

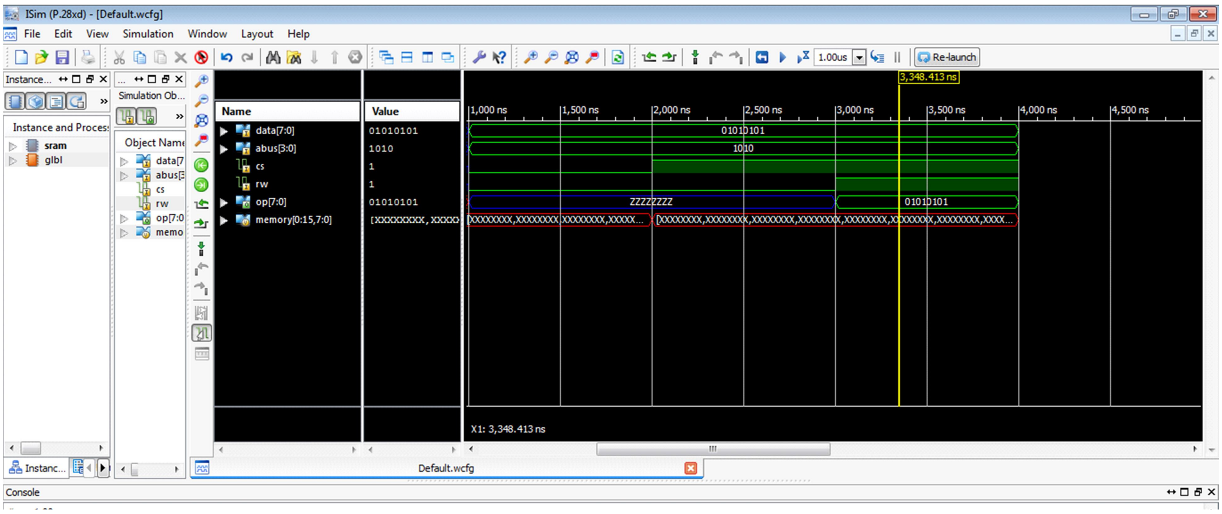
Factorial using for loop:

```
module fact(
        Input n,
        Output reg fact);
always@(n)
begin
integer i;
fact = 1;
if(n)
        begin
                fact=1;
        end
else
        begin
                for(i=0;i<=n;i=i+1)
```

```
                begin
                        fact=fact*i;
                end
        end
end
endmodule
```

SRAM:

```
module sram(input [7:0] data, input [3:0] abus, input cs, input rw, output reg [7:0] op);
reg [7:0] memory[0:15];
begin
always@(data,cs,rw,abus)
begin
if(cs == 1'b1)
        begin
                if(rw == 1'b0)
                        memory[abus] = data;
                else
                        op = memory[abus];
        end
else
        op = 8'bzzzzzzzz;
end


end
endmodule
```


ISim (P.28xd) - [Default.wcfg]
File Edit View Simulation Window Layout Help
Re-launch
1.00us
Instance and Process
sram
glbl
Simulation Ob...
Object Name
data[7:0]
abus[3:0]
cs
rw
op[7:0]
memory[0:15,7:0]
Name
Value
01010101
1010
1
1
01010101
3,348.413 ns
1,000 ns
1,500 ns
2,000 ns
2,500 ns
3,000 ns
3,500 ns
4,000 ns
4,500 ns
ZZZZZZZZ
X1: 3,348.413 ns
Default.wcfg
Console

Ripple carry adder: (Using 2 half adder and 4 full adder made of 2 half adder)

Half adder:

```
module half_adder(s,c,x,y);
input x,y;
output s,c;
xor (s,x,y);
and (c,x,y);


endmodule
```

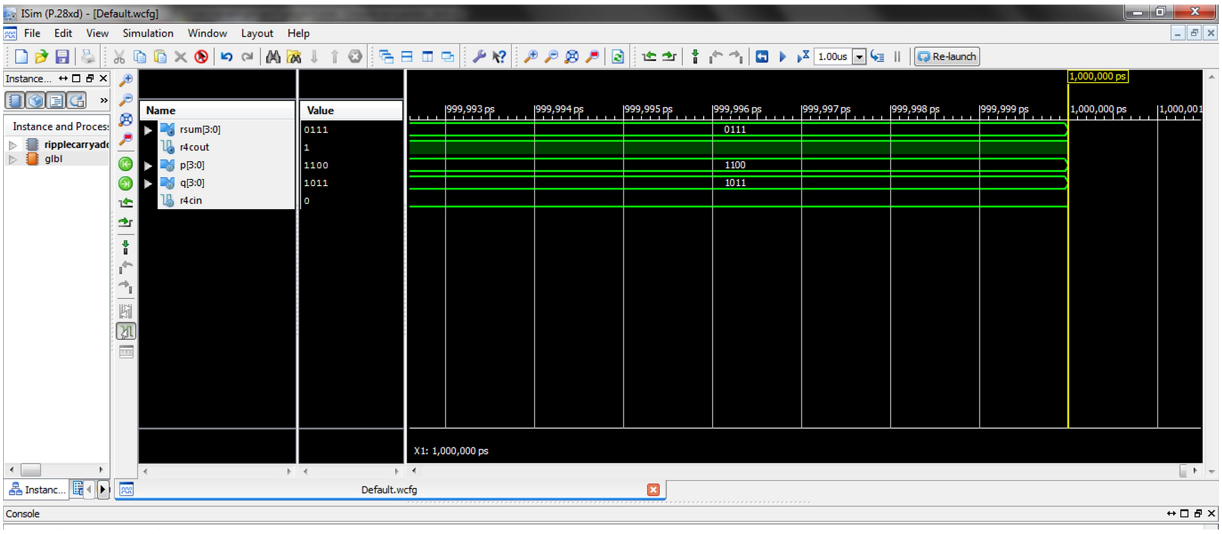
Full adder:

```
module full_adder(fsum,carry,a,b,cin);
input a,b,cin;
output fsum,carry;
wire w1,w2,w3;


half_adder ha1(w1,w2,a,b);
half_adder ha2(fsum,w3,w1,cin);
or(carry,w3,w2);
endmodule
```

Ripple carry adder:

```
module ripplecarry_adder(rsum,r4cout,p,q,r4cin);
input [3:0]p;
input [3:0]q;
output r4cout;
input r4cin;
output [3:0]rsum;
wire rco,rc1,rc2;
full_adder fa1(rsum[0],rco,p[0],q[0],r4cin);
full_adder fa2(rsum[1],rc1,p[1],q[1],rco);
```

```
full_adder fa3(rsum[2],rc2,p[2],q[2],rc1);

full_adder fa4(rsum[3],r4cout,p[3],q[3],rc2);

endmodule
```



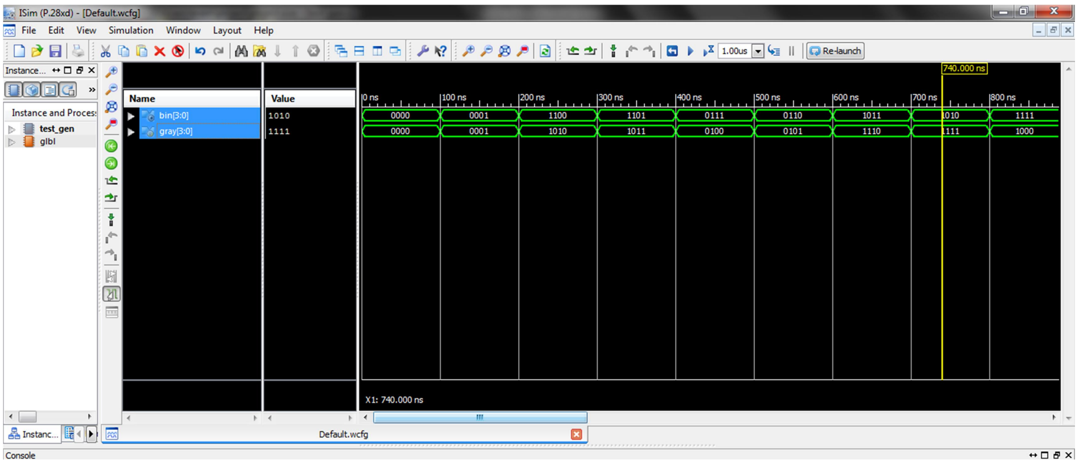
4:1 mux using structural modelling:

```
module mux2_1_struct(y,s,i0,i1);

input i0,i1,s;

output y;

wire sn,y1,y2;

not(sn,s);

and(y1,i0,sn);

and(y2,i1,s);

or(y,y1,y2);

endmodule
```

Convert gray code to binary code using generate statements:

```
module generate_bit_to_gray(gray ,bin );

parameter SIZE=4;

input [SIZE - 1:0]gray;

output [SIZE-1:0]bin;
```

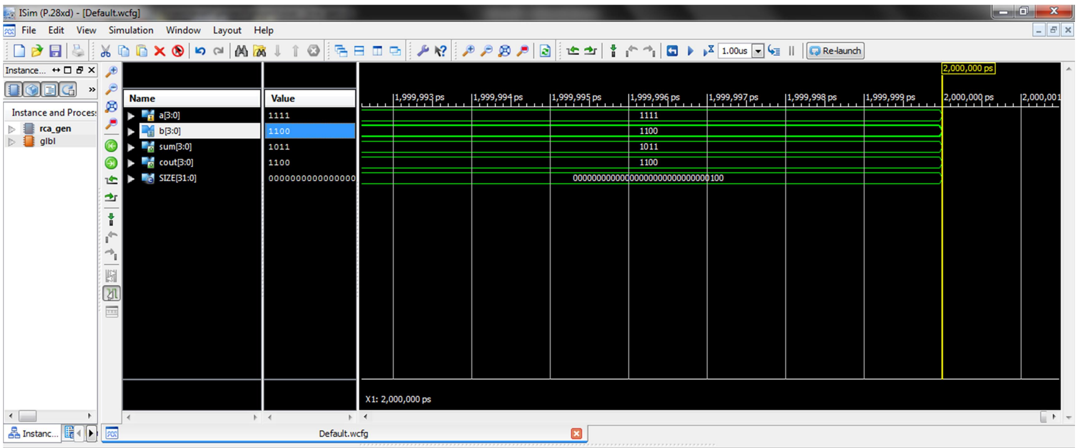
```
genvar i;

generate for(i=0;i<SIZE;i=i+1)

begin:bit

assign bin[i] = ^gray[SIZE-1:i];

end

endgenerate


endmodule
```



Ripple carry adder using generate statement:

```
module rca_gen(a,b,sum,cout);

parameter SIZE = 4;

input [SIZE-1:0]a;

input [SIZE-1:0]b;

output [SIZE-1:0]sum;

output [SIZE-1:0]cout;

genvar i;

assign sum[0]=(a[SIZE-1:0])^(b[SIZE-1:0]^0);

assign cout[0] = (a[SIZE-1:0] & b[SIZE-1:0]);

generate for(i=1;i<SIZE;i=i+1)

begin:rca
```

```
assign sum[i]=(a[SIZE-1:i])^(b[SIZE-1:i])^(cout[SIZE-1:(i-1)]);

assign cout[i] = (a[SIZE-1:i] & b[SIZE-1:i]) | (b[SIZE-1:i] & cout[SIZE-1:(i-1)]) | (cout[SIZE-1:(i-1)] & a[SIZE-1:i]);

end

endgenerate

endmodule
```



Half adder using Task:

```
module adddertask(a,b,ci,y,co);

input a,b,ci;

output reg y,co;

reg c1,c2,y1;

always@(a or b or ci)

        begin

                half(a,b,y1,c1);

                half(y1,ci,y,c2);

                co=c1|c2;

        end

task half;

        input a,b;

        output y,c;

                begin
```

```
                    y=a^b;
                    c=a&b;
            end
endtask
endmodule
```

Full adder using Task:

```
module fulladder_task(a,b,cin,sum,carry);
input a,b,cin;
output sum,carry;
reg sum,carry;
reg w1,w2,w3;
always @(a or b or cin)
begin
        half(a,b,w1,w2);
        half(w1,cin,sum,w3);
        or_gate(w2,w3,carry);
end
task half;
        input a,b;
        output sum,carry;
        begin
                sum=a^b;
                carry=a&b;
        end
endtask
task or_gate;
input c,d;
```

```
output carry;
begin
carry=c|d;
end
endtask
endmodule
```

XOR using Automatic task

```
module automatic_xor(clk1,clk2,c,d,e,f,cd_xor,ef_xor);
input [15:0]c,d,e,f;
output reg [15:0]cd_xor,ef_xor;
input clk1,clk2;

task automatic bitwise_xor;
	output [15:0]ab_xor;
	input [15:0]a,b;
	#10 ab_xor = a^b;
endtask

always@(posedge clk1)
bitwise_xor(ef_xor,e,f);
always@(posedge clk2)
bitwise_xor(cd_xor,c,d);
endmodule
```

Clock division for counter

```
module clkdiv(clk,clr,cout);
```

```
input clk,clr;
output reg [3:0]cout;
reg [24:0]clk_div=0;
initial cout=0;
always@(posedge clk)
	begin
		clk_div=clk_div+1;
	end
	always@(posedge clk_div[24])
	begin
		if(clr==0)
			cout=cout+1;
		else
			cout=4'b0000;
	end
```

File operation to write in file:

```
module file(a,b
   );
input [1:0]a;
output [2:0]b;
reg [2:0]b;
integer ch1;
always@(a)
begin
	b = 2*a;
end
initial
```

```
begin
	ch1 = $fopen("sample.txt");
	$fdisplay(ch1,"\n\t\t\tThis file is for testing\n");
	$fdisplay(ch1,"Enter 'a' in decimal\t\tOutput 'b' in decimal\t\tOutput 'b' in binary\n");
	$fmonitor(ch1,"\t%d\t\t\t\t%d\t\t\t\t%b\n",a,b,b);
	$fclose("sample.txt");
end
endmodule
```

Half adder using Function:

```
module ha_function(a,b,ci,sum,co);
input a,b,ci;
output reg sum,co;
reg c1,c2,c3,y1;
always@(a or b or ci)
	begin
		y1 = halfsum(a,b);
		c1 = carry(a,b);
		sum = halfsum(y1,ci);
		c2 = carry(ci,b);
		c3 = carry(ci,a);
		co = c1|c2|c3;
	end
function halfsum;
	input a,b;
		begin
			halfsum = a^b;
		end
```

```
endfunction


function carry;
	input a,b;
		begin
			carry = a&b;
		end
endfunction
endmodule
```

Full adder using task

```
	module fulladder_task(a,b,cin,y,co);
	input a,b,cin;
	output s,cout;
	reg w1,w2,w3;
	always@(a,b,cin)
	begin
		halfadder(a,b,w1,w2);
		halfadder(w1,cin,y,w3);
		assign cout=w3|w2
	end
	task halfadder;
	input a,b;
	output y,cout;
	begin
		y=a^b;
		cout=a&b;
	end
	endtask
	endmodule
```

Factorial using function

```
module recursive_function(a);

output reg [31:0]a;

initial a = calfactorial(4);


function automatic [31:0]calfactorial;

input [7:0]number;

begin

	if(number == 1 || number == 0)

		calfactorial = 1;

	else

		calfactorial = number*(calfactorial(number - 1));

end

endfunction

endmodule
```

Interfacing Programs

Up-Down Counter on LED

```
module updown_counter(select,q);

input select;

output reg[3:0]q;

integer i;

always @(select)

begin

	i=0;

	q=0;

	if(select==1)

		begin

			for(i=0;i<=8;i=i+1)

				#50 q=q+1;

		end

	else

		begin

			q=9;

			for(i=9;i>0;i=i-1)

				#50 q=q-1;

		end

end


endmodule
```

Counter on seven segment

```
module seven_segment(relay,clk,a,b,c,d);

input clk;
```

```
output relay;

output reg [7:0]a,b,c,d;

reg [22:0]cnt;

reg [15:0]val;


always@(posedge clk)

        cnt = cnt + 1;

        always@(posedge cnt[22])

                begin

                        val = val + 1;

                        case (val[3:0])

                                0:a = 8'b00111111;

                                1:a = 8'b00000110;

                                2:a = 8'b11011010;

                                3:a = 8'b11110010;

                                4:a = 8'b01100110;

                                5:a = 8'b10110110;

                                6:a = 8'b10111110;

                                7:a = 8'b11100000;

                                8:a = 8'b11111110;

                                9:a = 8'b11110110;

                                10:a = 8'b11101101;

                                11:a = 8'b00111110;

                                12:a = 8'b10011100;

                                13:a = 8'b01111010;

                                14:a = 8'b10011110;

                                15:a = 8'b10001110;
```

```
                    default: a = 8'b00000000;
                endcase
            end
assign relay = cnt[22];
endmodule
```

Stepper motor code:

```
module stepper(clk,stepper);
input clk;
output reg [3:0]stepper;
reg [23:0] cnt = 0;
reg [1:0] select = 0;

always@(posedge clk)
begin
        cnt = cnt + 1;
end

always@(posedge cnt[22])
begin
        select <= select + 2'b01;
        case(select)
                2'b00 : stepper = 4'b0110;
                2'b01 : stepper = 4'b0101;
                2'b10 : stepper = 4'b1001;
                2'b11 : stepper = 4'b1010;
                default : stepper = 4'b0000;
        endcase
```

```
end
endmodule
```

Keypad code:

```
module keypad(
   input [3:0] row,
   output reg [3:0]column,
   output reg [3:0] value,
   input clk
   );
reg [1:0]state;
reg [17:0]count;
always@(posedge clk)
count=count+1;

always@(posedge count[17])
begin
state=state+1;
case(state)
0:begin
  column=4'b1000;
  case(row)
  4'h8: value=4'b1101;//d
  4'h4: value=4'b1111;//0
  4'h2: value=4'b0000;//0
  4'h1: value=4'b1110;//0
  default value=value;
  endcase
  end
1:begin
        column=4'b0100; //p23
```

```
        case(row)
        4'h8: value=4'b1010;//A
        4'h4: value=4'b0011;//3
        4'h2: value=4'b0010;//2
        4'h1: value=4'b0001;//1
        default value=value;
        endcase
        end
2:begin
        column=4'b0010;
        case(row)
        4'h8: value=4'b1011;//b
        4'h4: value=4'b0110;//6
        4'h2: value=4'b0101;//5
        4'h1: value=4'b0100;//4
        default value=value;
        endcase
        end

3: begin
        column=4'b0001;
        case(row)
        4'h8: value=4'b1100;//c
        4'h4: value=4'b1001;//9
        4'h2: value=4'b1000;//8
        4'h1: value=4'b0111;//7
        default value=value;
        endcase
        end
endcase
```

```
end
endmodule
```

LCD code to display “HELLO”!

```
module lcd(clk,data,rst,rs,en,rw
    );
input clk;
output rst;
output reg [7:0]data;

reg pre_clk,rs,en;
reg [5:0]step;
reg [32:0]clk_div;

parameter H = 8'b01001000,
                              E = 8'b01000101,
                                        L = 8'b01001100,
                                        O = 8'b01001111;
                                        assign rw = 1'b0;
                                        always@(posedge clk)
                                        begin
                                                    clk_div = clk_div + 1;
                                                    pre_clk = clk_div[14];
                                       end

                                       always@(posedge pre_clk)
                                       begin
```

```
		if(rst == 1'b1)
			step = 6'b0;
		else
			step = step + 1;
	end
	always@(step)
	begin
		case(step)
			1: begin
				en = 0;
				end
			2: begin
				data= 8'h38;
				rs = 0;
				en = 1;
				end
			3: begin
				en = 0;
				end
			4: begin
				data = 8'h38;
				rs = 0;
				en = 1;
				end
			5: begin
				en = 0;
				end
```

```
6: begin
		data = 8'h0c;
		rs = 0;
		en = 1;
		end
7: begin
		en = 0;
		end
8: begin
		data= 8'h06;
		rs = 0;
		en = 1;
		end
9: begin
		en = 0;
		end
10: begin
		data = 8'h01;
		rs = 0;
		en = 1;
		end
11: begin
		en = 0;
		end
12: begin
		data = 8'h80;
		rs = 0;
```

```
        en = 1;
        end
13: begin
        en = 0;
        end

14: begin
        data = H;
        rs = 1;
        en = 1;
        end
15: begin
        en = 0;
        end
16: begin
        data = E;
        rs = 1;
        en = 1;
        end
17: begin
        en = 0;
        end
18: begin
        data = L;
        rs = 1;
        en = 1;
        end
```

```
                19: begin
                    en = 0;
                    end
                20: begin
                    data = L;
                    rs = 1;
                    en = 1;
                    end
                21: begin
                    en = 0;
                    end
                22: begin
                    data = 0;
                    rs = 1;
                    en = 1;
                    end
                23: begin
                    en = 0;
                    end
            endcase
        end
endmodule
```

Melay Model (0101 detector)

```
module mealy_fsm(din,reset,clk,y);
input din;
input clk;
input reset;
```

```
output reg y;
reg[1:0]cst,nst;
parameter s0=2'b00, s1=2'b01, s2=2'b10, s3=2'b11;
always@(cst,din)
begin
case(cst)
s0:if(din==1'b1)
        begin nst=s0;y=1'b0;end
  else
        begin nst=s1;y=1'b0;end
s1:if(din==1'b0)
        begin nst=s1;y=1'b0;end
  else
        begin nst=s2;y=1'b0;end
s2:if(din==1'b1)
        begin nst=s0;y=1'b0;end
  else
        begin nst=s3;y=1'b0;end
s3:if(din==1'b0)
        begin nst=s1;y=1'b0;end
  else
        begin nst=s2;y=1'b1;end
default : nst=s0;
endcase
end
always@(posedge clk)
begin
        if(reset)
                cst<=s0;
        else
```

```
                    cst<=nst;
end
endmodule
```

Moore model(0101 detector)

```
module moore(din,reset,clk,y);
input din;
input clk;
input reset;
output reg y;
reg[1:0]cst,nst;
parameter s0=3'b000, s1=3'b001, s2=3'b010, s3=3'b011, s4=3'b100;
always@(cst or din)
begin
case(cst)
s0:begin y=1'b0;
          if(din==1'b1)nst=s0;
          else nst=s1;end
s1:begin y=1'b0;
          if(din==1'b0)nst=s1;
          else nst=s2;end

s2:begin y=1'b0;
          if(din==1'b1)nst=s0;
          else nst=s3;end
s3:begin y=1'b0;
          if(din==1'b0)nst=s1;
          else nst=s4;end
s4:begin y=1'b1;
          if(din==1'b0)nst=s1;
```

```
        else nst=s0;end
default:nst=s0;
endcase
end
always@(posedge clk)
begin
        if(reset)
                cst<=s0;
        else
                cst<=nst;
end
endmodule
```