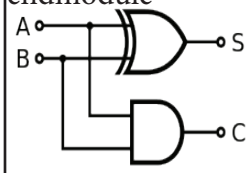
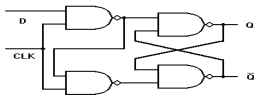
```
//Half adder
Module ha(a,b,c,s);
Input a,b;
Output c,s;
Assign c=a&b;
Assign c=a&b;
Assign s=a^b;
endmodule
```



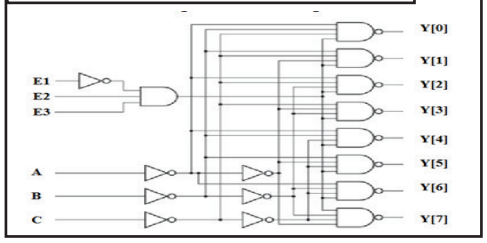

```
`timescale 1ns / 1ps
module ha_tb
   reg a,b;
   wire c,s;
   ha uut(a,b,c,s);
   initial
   begin
   a=0; b=0;
   #20 a=0; b=1;
   #20 $stop;
   end
endmodule
```

```
//D-Flip Flop, posedge trig, async rst
module dff(d,q,clk,rst);
input d, clk, rst;
ouput q;
reg q;
always @(posedge clk or posedge rst)
 if (rst)
   q<=1'b0;
 else
   q<=d;
endmodule
```

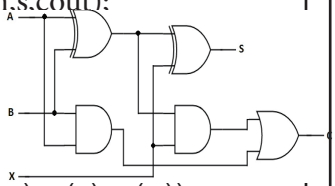



```
`timescale 1ns/1ps
module dff_tb;
reg d, clk, rst;
wire q;
dff uut(d,q,clk,rst);
initial
clk=1'b0;
always
  #5 clk=~clk;
initial
begin
d=0; rst=0;
#20 d=1; rst=1;
#20 $stop;
end
endmodule
```

```
//3-8 decoder w/ enable (nand)
module dcdr38(en,a,b,c,y);
input en, a, b, c;
output [7:0] y;
assingn y[0]=~((~c&~b&~a)&en;
assingn y[1]=~((~c&~b&a)&en;
assingn y[2]=~((~c&b&~a)&en;
assingn y[3]=~((~c&b&a)&en;
assingn y[4]=~((c&~b&~a)&en;
assingn y[5]=~((c&~b&a)&en;
assingn y[6]=~((c&b&~a)&en;
assingn y[7]=~((c&b&a)&en;
endmodule
```



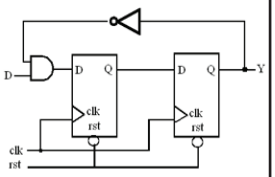

```
//Full adder
 module fa(a,b,cin,s,cout);
input a,b,cin;
output cout,s;
wire m,n,p;
ha g1(a,b,n,m);
ha g2(.a(cin), .b(m), .s(s), .c(p));
assign cout=p|n;
endmodule
```



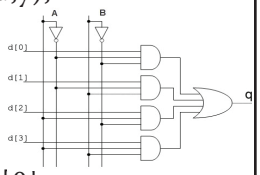

```
//Shift register (4 bit)
module dffb(clk,clr,load,sft,db,qb);
input clk, clr, load, sft;
input [3:0] db;
output [3:0] qb;
reg [3:0] qb;
always @(posedge clr or posedge clk)
begin
 if (clr) qb<=0;
 else if (load)
    qb<=db;
 else if (sft)
    qb<={1'b0; qb[3:1]};
    //qb[3]<=1'b0;
    //qb[2]<=qb[3];
    //qb[1]<=qb[2];
    //qb[0]<=qb[1];
end
endmodule
```

```
`timescale 1ns/1ps
module dffa_tb;
reg      clk, clr, load;
reg [3:0] da;
wire [3:0] qa;
dffa uut ( clk, clr, load, da, qa );
initial  clk = 0;
always   #10 clk = ~ clk;
initial
begin
   clr = 1;    da  = 4'b1011;
   #24  clr = 0;  load = 1;
   #24  load = 0;
   #60 $stop;
end
endmodule
```

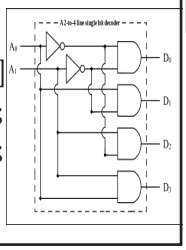
```
//pmidterm dff with not and
module prob2(clk,d,rst,y);
input clk,d,rst;
output y;
reg y,n;
wire m;
assign m=d&~y;
always @(posedge clk or negedge rst)
beigin
 if (~rst)
 begin
   n<=0; y<=0;
 end
 else
 begin
   n<=m; y<=n;
 end
end
endmodule
```




```
//Multiplexor 2-1
module mux(s,d,y);
input s;
input [1:0] d;
ouput y;
wire m, n;
assign m=~s&d[0];
assign n=s&d[1];
assign y=m|n;
endmodule
```




```
//decoder 2-4
module decoder(d,y);
input [1:0] d;
output [3:0] y;
assign y[0]=~d[0]&~d[1];
assign  y[1]=~d[0]&d[1];
assign  y[2]=d[0]&~d[1];
assign  y[3]=d[0]&d[1];
endmodule
```




```
//3-bit ripple carry adder hierachical design
module fa(a,b,c,cout,s);
input a,b,c;
output cout,s;
assign s=a^b^c;
assign cout=((a^b)&cin)|(a&b);
endmoudle
module rca3(a,b,c,cout,s);
input [2:0] a,b;
input c;
output [2:0] s
output cout;
wire [1:0] m;
fa g1(.a(a[0]), .b(b[0]), .c(c), .cout(m[0]), .s(s[0]));
fa g2(a(a[1]), .b(b[1]), .c(m[0]), .cout(m[1]), .s(s[1]));
fa g3(a[2], b[2], m[1], cout, s[2]);
endmodule
```

```
module rca3_tb;
reg [2:0] a,b;
reg c;
wire [2:0] s;
wire cout;
rca3 uut(a,b,c,cout,s);
initial
begin
  a=0; b=0; c=0;
  #10 a=4; b=5;
  #20 a=6; b=2; c=1;
  # 20 $stop;
end
endmodule
```

```
module mux(s,d1,d0,y);
input [3:0] d1, d0;
input s;
output [3:0] y;
reg [3:0] y;
always @(s or d1 or d0)
begin
  if (s) y=d1;
  else  y=d0;
end
endmodule
```

```
//4 bit shift right serial in/out
module shift_r4(clk,din,dout);
input clk, din;
output dout;
reg [3:0] q;
assign dout=q[0];
always@(poedge clk)
begin
  q<={din, q[3:1]};
end
endmodule
```

```
module mytt(A, B, Y);
input [5:0] A, B;
output[11:0] Y;
reg [11:0] Y;
parameter C=3'b100;
always@(A or B)
begin
  Y[2:0]=A[2:0];
  Y[3] = A[3] & B[3];
  Y[5:4]= {A[5]&B[5], A[4]|B[4]};
  Y[8:6]= B[2:0];
  Y[11:9]=C;
end
```

```
//FF w/ Pos edge Clk, Synchro Set
module flop (C, D, S, Q);
input C, D, S;
output Q;
reg Q;
always @(posedge C)
begin
  if (S)
    Q <= 1'b1;
  else
    Q <= D;
end
endmodule
```

```
//4-bit Reg w/ Pos-Edge Clk,
//Asynchronous Set, Clk Enable
module flop (C, D, CE, PRE, Q);
input C, CE, PRE;
input [3:0] D;
output [3:0] Q;
reg [3:0] Q;
always @(posedge C or posedge PRE)
begin
  if (PRE)
    Q <= 4'b1111;
  else if (CE)
    Q <= D;
end
endmodule
```

```
//4-bit up/down counter w/ asynch clr
module counter (C, CLR, UP_DOWN, Q);
input C, CLR, UP_DOWN;
output [3:0] Q;
reg [3:0] tmp;
always @(posedge C or posedge CLR)
begin
  if (CLR)
    tmp <= 4'b0000;
  else
    if (UP_DOWN)
    tmp <= tmp + 1'b1;
  else
    tmp <= tmp - 1'b1;
end
assign Q = tmp;
endmodule
```

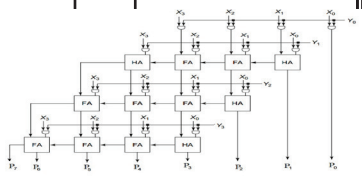
```
//Concatenation example:
module mytt2(A, B, Y);
input [2:0] A, B;
output[14:0] Y;
parameter C=3'b100;
assign Y = { A, B, {2{C}}, 3'b110};
endmodule
```

```
//Flip-flop with Pos-Edge Clock
module flop (CLK, D, Q);
input CLK, D;
output Q;
reg Q;
always @(posedge CLK)
begin
  Q <= D;
end
endmodule
```

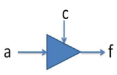
```
//FF w/Neg Edge Clk and Async Clr
module flop (C, D, CLR, Q);
input C, D, CLR;
output Q;
reg Q;
always@(negedge C or posedge CLR)
begin
if (CLR)
  Q <= 1'b0;
else
  Q <= D;
end
endmodule
```

```
//Tristate
module three_st (T, I, O);
input T, I;
output O;
reg O;
always @(T or I)
begin
  if (~T)
    O = I;
  else
    O = 1'bZ;
end
endmodule
```

```
//tristate
module three_st (T, I, O);
input T, I;
output O;
assign O = (~T) ? I: 1'bZ;
endmodule
```



Tri-State Buffer

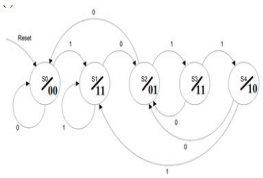


c	a	f
0	0	Z
0	1	Z
1	0	0
1	1	1

```
module multiply(x,y,p);
input [3:0] x, y;
output [7:0] p;
wire [14:0] ad;
wire [7:0] fs, fc;
wire [7:0] hs, hc;
assign p[0]  = x[0] & y[0];
assign ad[0] = x[1] & y[0];
assign ad[1] = x[2] & y[0];
assign ad[2] = x[3] & y[0];
assign ad[3] = x[0] & y[1];
assign ad[4] = x[1] & y[1];
assign ad[5] = x[2] & y[1];
assign ad[6] = x[3] & y[1];
assign ad[7]  = x[0] & y[2];
assign ad[8]  = x[1] & y[2];
assign ad[9]  = x[2] & y[2];
assign ad[10] = x[3] & y[2];
assign ad[11] = x[0] & y[3];
assign ad[12] = x[1] & y[3];
assign ad[13] = x[2] & y[3];
assign ad[14] = x[3] & y[3];

ha g1(ad[3], ad[0], hc[0], hs[0]);
assign p[1] = hs[0];
fa g2(ad[4], ad[1], hc[0], fs[0], fc[0]);
fa g3(ad[5], ad[2], fc[0], fs[1], fc[1]);
ha g4(ad[6], fc[1], hc[1], hs[1]);
ha g5(ad[7], fs[0], hc[2], hs[2]);
assign p[2] = hs[2];
fa g6(ad[8], fs[1], hc[2], fs[2], fc[2]);
fa g7(ad[9], hs[1], fc[2], fs[3], fc[3]);
fa g8(ad[10], hc[1], fc[3], fs[4], fc[4]);
ha g9(ad[11], fs[2], hc[3], hs[3]);
assign p[3] = hs[3];
fa g10(ad[12], fs[3], hc[3], fs[5], fc[5]);
assign p[4] = fs[5];
fa g11(ad[13], fs[4], fc[5], fs[6], fc[6]);
assign p[5] = fs[6];
fa g12(ad[14], fc[4], fc[6], fs[7], fc[7]);
assign p[6] = fs[7];
assign p[7] = fc[7];
endmodule
```

```
module moorefsm(clk,rst,a,y); //moore fsm
input clk, rst,a;
output [1:0] y;
reg [1:0] y;
parameter s0=3'b000, s1=3'b0001,
s2=3'b010, s3=3'b011, s4=3'b100;
reg [2:0] cs, ns;
always @(posedge clk or posedge rst)
begin
  if (rst) cs<=s0;
  else cs<=ns;
end
always @(cs or a)
begin
  case (cs)
     s0: if(a)   ns = s1;
         else    ns = s0;
     s1: if(~a) ns = s2;
         else    ns = s0;
     s2: if(a)   ns = s3;
         else    ns = s0;
     s3: if(a)   ns = s4;
         else    ns = s2;
     s4: if(a)   ns = s1;
         else    ns = s2;
     default: ns = s0;
  endcase
end
always@(cs )
begin
  case(cs)
    s0: y = 2'b00;
    s1: y = 2'b11;
    s2: y = 2'b01;
    s3: y = 2'b11;
    s4: y = 2'b10;
    default: y = 2'b00;
  endcase
end
endmodule
```



```
//8-bit Shift-Left Reg w/ PosEdge
//Clk, Serial In, and Serial Out
module shift (C, SI, SO);
input C,SI;
output SO;
reg [7:0] tmp;
always @(posedge C)
begin
  tmp <= tmp << 1;
  tmp[0] <= SI;
end
assign SO = tmp[7];
endmodule
```

```
//8-bit Shift-Left Reg w/ PosEdge
//Clk, Async Clr, Serial In, Serial Out
module shift (C, CLR, SI, SO);
input C, SI, CLR;
output SO;
reg [7:0] tmp;
always @(posedge C or posedge CLR)
begin
  if (CLR)
    tmp <= 8'b00000000;
  else
    tmp <= {tmp[6:0], SI};
end
assign SO = tmp[7];
endmodule
```

```
//8-bit Shift-Left Reg w/ PosEdge Clk,
//Async paralel Load, Serial In, Serial Out
module shift (C, ALOAD, SI, D, SO);
input C, SI, ALOAD;
input [7:0] D;
output SO;
reg [7:0] tmp;
always @(posedge C or posedge ALOAD)
begin
  if (ALOAD)
    tmp <= D;
  else
    tmp <= {tmp[6:0], SI};
end
assign SO = tmp[7];
endmodule
```

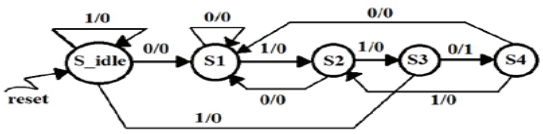
```
//8-bit Shift-Left Reg w/ NegEdge
//Clk, Clk Enabl, Serial In/Serial Out
module shift (C, CE, SI, SO);
input C, SI, CE;
output SO;
reg [7:0] tmp;
always @(negedge C)
begin
  if (CE)
  begin
    tmp <= tmp << 1;
    tmp[0] <= SI;
  end
end
assign SO = tmp[7];
endmodule
```

```
4-to-1 MUX using IF Statement
module mux (a, b, c, d, s, o);
input a,b,c,d;
input [1:0] s;
output o;
reg o;
always @(a or b or c or d or s)
begin
  if (s == 2'b00)
    o = a;
  else if (s == 2'b01)
    o = b;
  else if (s == 2'b10)
    o = c;
  else
    o = d;
end
endmodule
```

```
//4-to-1 MUX Using CASE
module mux (a, b, c, d, s, o);
input a, b, c, d;
input [1:0] s;
output o;
reg o;
always @(a or b or c or d or s)
begin
  case (s)
    2'b00 : o = a;
    2'b01 : o = b;
    2'b10 : o = c;
    default : o = d;
  endcase
end
endmodule
```

```
//Decoders
module dec (sel, res);
input [2:0] sel;
output [7:0] res;
reg [7:0] res;
always @(sel or res)
begin
  case (sel)
    3'b000 : res = 8'b00000001;
    3'b001 : res = 8'b00000010;
    3'b010 : res = 8'b00000100;
    3'b011 : res = 8'b00001000;
    3'b100 : res = 8'b00010000;
    3'b101 : res = 8'b00100000;
    3'b110 : res = 8'b01000000;
    default : res = 8'b10000000;
  endcase
end
endmodule
```

```
//state machine to find 0110
module fsm_detector(reset, clk, a, y);
input reset, a, clk;
output y;
reg y;
parameter s_idle = 3'b000, s1=3'b001,
s2=3'b010, s3=3'b011, s4=3'b100;
reg [2:0] cs, ns;
always@(posedge clk or posedge reset)
begin
  if(reset) cs <= s_idle;
  else cs <= ns;
end
always@(cs or a)
begin
  case(cs)
    s_idle: if(a) ns = s_idle;
            else ns = s1;
    s1: if(a) ns = s2;
        else ns = s1;
    s2: if(a) ns = s3;
        else ns = s1;
    s3: if(~a) ns = s4;
        else ns = s_idle;
    s4: if(a) ns = s2;
        else ns = s1;
    default: ns = s_idle;
  endcase
end
always@(cs or a)
begin
  case(cs)
    s_idle: y = 0;
    s1: y = 0;
    s2: y = 0;
    s3: if(~a) y = 1;
        else y = 0;
    s4: y = 0;
    default: y = 0;
  endcase
end
endmodule
```




```
//Priority Encoder
module priority (sel, code);
input [7:0] sel;
output [2:0] code;
reg [2:0] code;
always @(sel)
begin
  if (sel[0]) code = 3'b000;
  else if (sel[1]) code = 3'b001;
  else if (sel[2]) code = 3'b010;
  else if (sel[3]) code = 3'b011;
  else if (sel[4]) code = 3'b100;
  else if (sel[5]) code = 3'b101;
  else if (sel[6]) code = 3'b110;
  else if (sel[7]) code = 3'b111;
  else code = 3'bxxx;
end
endmodule
```

```
//Unsigned 8-bit Adder w/ CO
module adder(A, B, SUM, CO);
input [7:0] A;
input [7:0] B;
output [7:0] SUM;
output CO;
wire [8:0] tmp;
assign tmp = A + B;
assign SUM = tmp [7:0];
assign CO = tmp [8];
endmodule
```

```
//Unsigned 8-bit Adder/Subtractor
module addsub(A, B, OPER, RES);
input OPER;
input [7:0] A;
input [7:0] B;
output [7:0] RES;
reg [7:0] RES;
always @(A or B or OPER)
begin
  if (OPER==1'b0)
    RES = A + B;
  else
    RES = A - B;
end
```

```
//Comparators
//Verilog: (==, !=,<, <=, >, >=)
module compar(A, B, CMP);
input [7:0] A;
input [7:0] B;
output CMP;
assign CMP = (A >= B) ? 1'b1 : 1'b0;
endmodule
```

```
//Unsigned 8x4-bit Multiplier
module compar(A, B, RES);
input [7:0] A;
input [3:0] B;
output [11:0] RES;
assign RES = A * B;
```

<u>Dataflow</u>

```
assign y=a&b;
```

<u>behaviorial design</u>

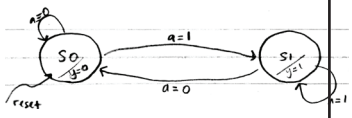
```
always @(posedge clk)
begin
```

```
//Finite State Machine
module fsm (clk, reset, x1, outp);
input clk, reset, x1;
output outp;
reg outp;
reg [1:0] state;
reg [1:0] next_state;
parameter s1 = 2'b00, s2 = 2'b01,
                s3 = 2'b10, s4 = 2'b11;
always @(posedge clk or posedge reset)
begin
  if (reset)
    state <= s1;
  else
     state <= next_state;
end
always @(state or x1)
begin
  case (state)
    s1: if (x1 == 1'b1)
          next_state = s2;
        else
          next_state = s3;
    s2: next_state = s4;
    s3: next_state = s4;
    s4: next_state = s1;
  endcase
end
always @(state)
begin
  case (state)
    s1: outp = 1'b1;
    s2: outp = 1'b1;
    s3: outp = 1'b0;
    s4: outp = 1'b0;
  endcase
end
endmodule
```

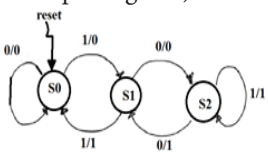
```
//Hierarchical Design
module my_block (in1, in2, dout);
  input in1, in2;
  output dout;
endmodule

module top (DI_1, DI_2, DI_3,DI_4, DOUT1, DOUT2);
  input DI_1, DI_2, DI_3, DI_4;
  output DOUT1, DOUT2;
  my_block inst1 (.in1(DI_1), .in2(DI_2), .dout(DOUT));
  my_block inst2(DI_3,DI_4, DOUT);
endmodule
```

```
//Moore State Machine
module moore_fsm(rst, clk, a y);
input rst, clk, a;
output y;
reg y;
parameter s0=1'b0, s1=1'b1;
reg cs, ns;
 //state register block
always @(posedge rst or posedge clk)
begin
  fif (rst) cs<=s0;
  else cs<=ns;
end
//combinational next state block
always @(cs or a)
begin
  case (cs)
    s0: if (a) ns=s1;
          else ns=s0;
    s1: if (a) ns=s1;
          else ns=s0;
    default: ns=s0;
  endcase
end
//output combinational block
always @(cs)
begin
  case(cs)
    s0: y=0;
    s1: y=1;
    default y=0;
  endcase
end
endmodule
```




```
//finite state machine, 3 states
module fsm(clk,rst,a,y);
input clk,rst,a;
output y;
reg y;
parameter s0=2'b00, s1=2'b01,
s2=2'b10;
reg [1:0] cs,ns;
always @(posedge clk or posedge rst)
begin
  if (rst) cs<=s0;
  else cs<=ns;
end
always @(cs or a)
begin
  case (cs)
    s0: if (a) ns=s1;
          else ns=s0;
    s1: if (~a) ns=s2;
          else ns=s0;
    s2: if (a) ns=s2;
          else ns=s1;
    default: ns=s0;
  endcase
end
```




```
always @(cs or a)
begin
  case (cs)
    s0: y=0;
    s1: if (a) y=1;
          else y=0;
    s2: y=1;
    default: y=0;
  endcase
end
endmodule
```