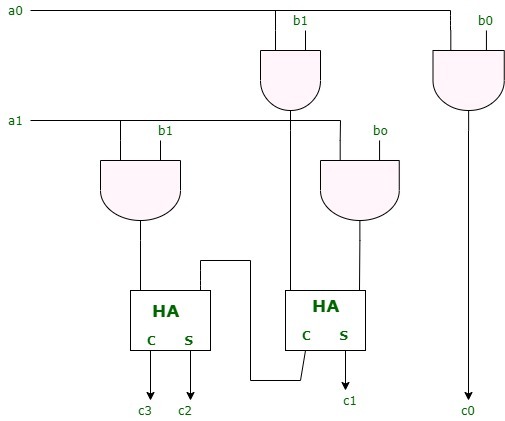
**FPGA TOPICS  
2-Bit Multiplier:**

  
module twobitmultiplier(input [1:0]a,b, output [3:0]c);

assign c[0] = (a[0]&b[0]);

assign c[1] = (a[0]&b[1]) ^ (a[1]&b[0]);

assign c[2] = (a[1]&b[1]) ^ ((a[0]&b[1]) & (a[1]&b[0]));

assign c[3] = (a[1]&b[1]) & ((a[0]&b[1]) & (a[1]&b[0]));

endmodule

**Carry Lookahead Adder (All Outputs Produced):**

module lab2(input [3:0]a,b, output reg [3:0]c,s);

reg [3:0]p,g;

integer i;

reg cin=1'b0;

always@(\*)

begin

s=4'b0000;

p[0] = a[0]^b[0];

g[0] = a[0]&b[0];

s[0] = p[0]^cin;

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

for(i=1; i<4; i=i+1)

begin

p[i] = a[i]^b[i];

g[i] = a[i]&b[i];

s[i] = p[i]^c[i-1];

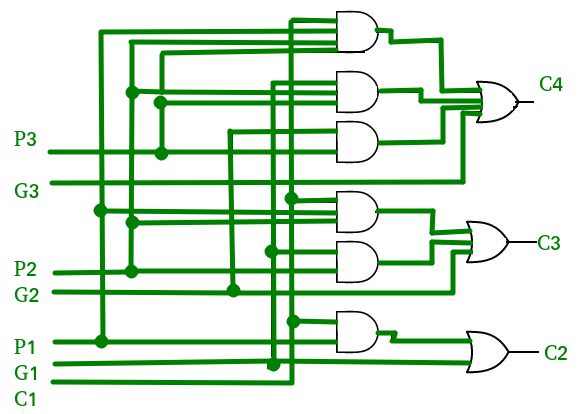
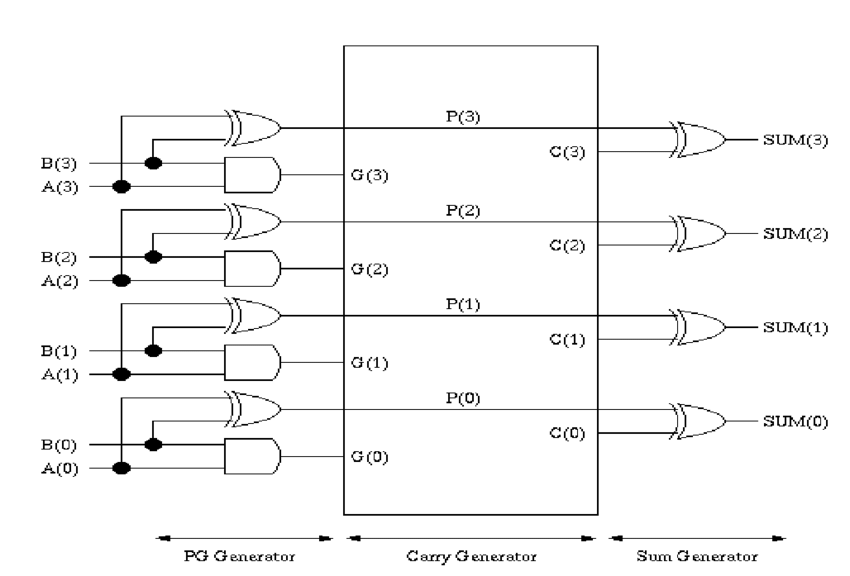
c[i] = g[i]|(p[i]&c[i-1]);

end

end

endmodule

**Carry Lookahead Adder (Required Outputs Produced for FPGA Kit):**



module lab2(input [3:0]a,b, output reg [4:0]o);

reg [3:0]p,g,c,s;

integer i;

reg cin=1'b0;

always@(\*)

begin

s=4'b0000;

p[0] = a[0]^b[0];

g[0] = a[0]&b[0];

s[0] = p[0]^cin;

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

for(i=1; i<4; i=i+1)

begin

p[i] = a[i]^b[i];

g[i] = a[i]&b[i];

s[i] = p[i]^c[i-1];

c[i] = g[i]|(p[i]&c[i-1]);

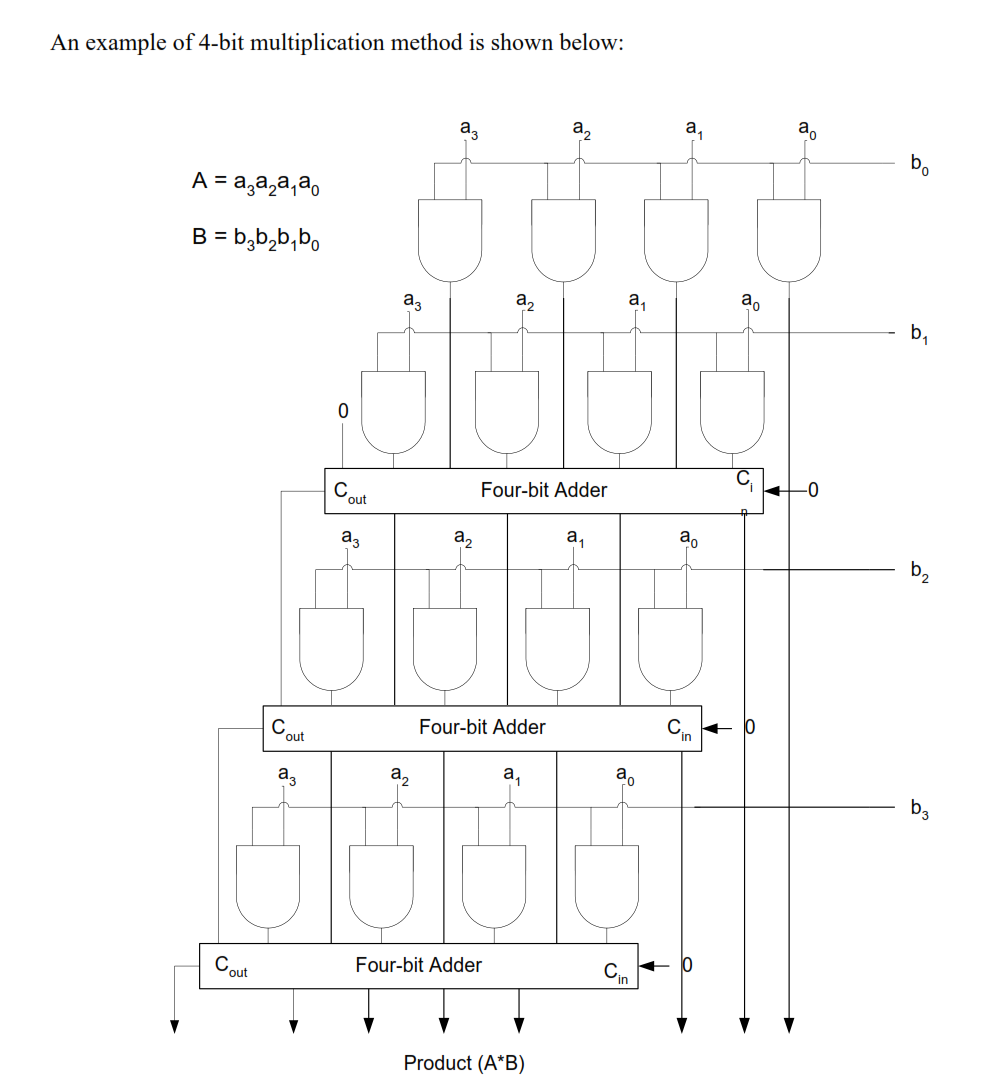
end

o[4:0]={c[3],s[3:0]};

end

endmodule

**Parallel Multiplier(Learnt in DSD):**

  
module parallelmultiplier(input [3:0]a,b, output reg [7:0]o);

reg [3:0]ands[3:0];

reg [3:0]sums[2:0];

reg [4:0]cars[2:0];

integer i,j;

always @(\*) begin

for(i=0; i<4; i=i+1) begin

for(j=0; j<4; j=j+1) begin

ands[i][j]=a[i]&b[j];

end

end

for(i=0; i<3; i=i+1) begin

cars[i][0]=0;

end

for(i=0; i<3; i=i+1)begin

sums[0][i]=(ands[0][i+1])^(ands[1][i])^(cars[0][i]);

cars[0][i+1]=(ands[0][i+1]&ands[1][i])|(ands[1][i]&cars[0][i])|(cars[0][i]&ands[0][i+1]);

end

sums[0][3]=(ands[1][3])^(cars[0][3]^1'b0);

cars[0][4]=(ands[1][3]&cars[0][3])|(cars[0][3]&1'b0)|(1'b0&ands[1][3]);

for(j=1; j<3; j=j+1)begin

for(i=0; i<3; i=i+1)begin

sums[j][i]=(sums[j-1][i+1])^(ands[j+1][i])^(cars[j][i]);

cars[j][i+1]=(sums[j-1][i+1]&ands[j+1][i])|(ands[j+1][i]&cars[j][i])|(cars[j][i]&sums[j-1][i+1]);

end

sums[j][3]=(ands[j+1][3])^(cars[j][3])^(cars[j-1][4]);

cars[j][4]=(ands[j+1][3]&cars[j][3])|(cars[j][3]&cars[j-1][4])|(cars[j-1][4]&ands[j+1][3]);

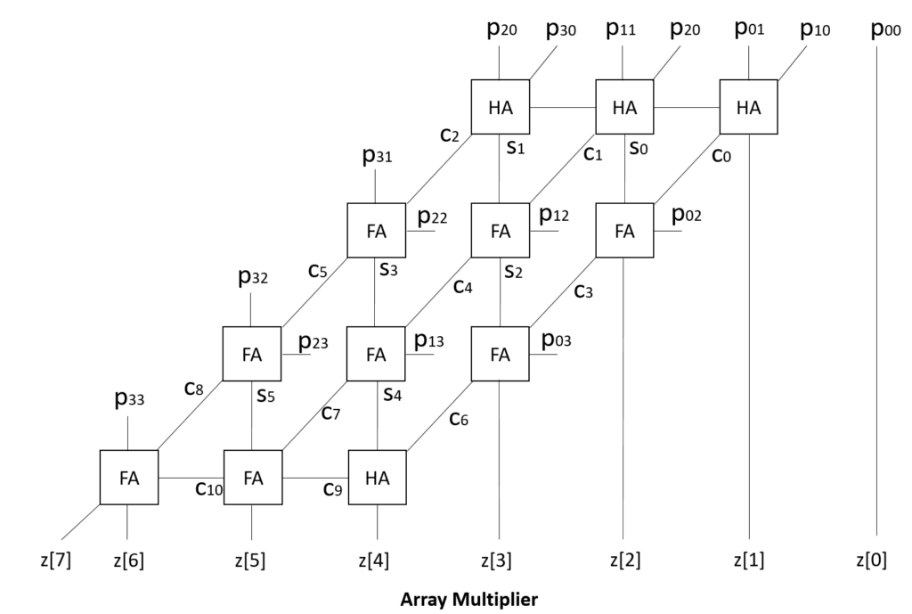
end

o[7:0]={cars[2][4],sums[2][3:0],sums[1][0],sums[0][0],ands[0][0]};

end

endmodule

**Array Multiplier(no one in the world uses):**

  
module arrarmultiplier(input [3:0]a,b, output reg [7:0]o);

reg [3:0]ands[3:0];

reg [2:0]sums[3:0];

reg [2:0]cars[3:0];

integer i,j;

always @(\*)

begin

for(i=0; i<4; i=i+1)begin

for(j=0; j<4; j=j+1)begin

ands[i][j]=a[i]&b[j];

end

end

for(i=0; i<3; i=i+1)begin //initial stage

sums[0][i]=ands[0][i+1]^ands[1][i];

cars[0][i]=ands[0][i+1]&ands[1][i];

end

for(i=1; i<3; i=i+1)begin //intermediate stages

for(j=0; j<2; j=j+1)begin

sums[i][j]=ands[i+1][j]^cars[i-1][j]^sums[i-1][j+1];

cars[i][j]=(ands[i+1][j]&cars[i-1][j])|(cars[i-1][j]&sums[i-1][j+1])|(sums[i-1][j+1]&ands[i+1][j]);

end

sums[i][2]=ands[i+1][2]^cars[i-1][2]^ands[i][3];

cars[i][2]=(ands[i+1][2]&cars[i-1][2])|(cars[i-1][2]&ands[i][3])|(ands[i][3]&ands[i+1][2]);

end

//final stage

sums[3][0]=sums[2][1]^cars[2][0];

cars[3][0]=sums[2][1]&cars[2][0];

sums[3][1]=cars[3][0]^cars[2][1]^sums[2][2];

cars[3][1]=(cars[3][0]&cars[2][1])|(cars[2][1]&sums[2][2])|(sums[2][2]&cars[3][0]);

sums[3][2]=cars[3][1]^cars[2][2]^ands[3][3];

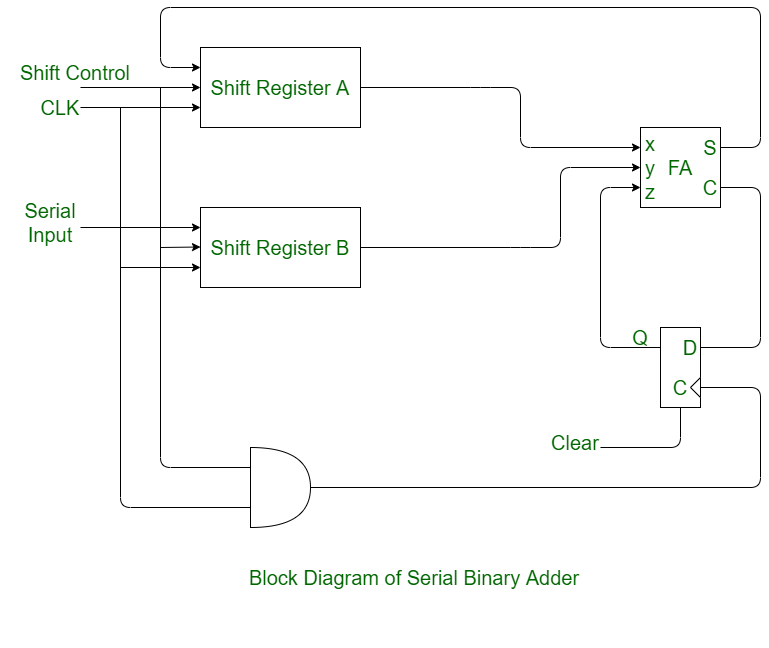
cars[3][2]=(cars[3][1]&cars[2][2])|(cars[2][2]&ands[3][3])|(ands[3][3]&cars[3][1]);

o[7:0]={cars[3][2],sums[3][2:0],sums[2][0],sums[1][0],sums[0][0],ands[0][0]};

end

endmodule

**Serial Adder Sujal Code:**



SSerialSSsmodule full\_adder(a, b, cin, sum, cout);

input a, b, cin;

output sum, cout;

assign {cout, sum} = a + b + cin;

endmodule

module d\_flipflop(d, clk, enable, reset, out);

input d, clk, enable, reset;

output out;

reg out;

always @ (posedge clk or posedge reset) begin

if (reset)

out = 0;

else

if (enable)

out = d;

end

endmodule

module piso(clk, enable, rst, data, out);

input enable, clk, rst;

input [3:0] data;

output out;

reg out;

reg [3:0] memory;

always @ (posedge clk, posedge rst) begin

if (rst == 1'b1) begin

out <= 1'b0;

memory <= data;

end

else begin

if (enable) begin

out = memory[0];

memory = memory >> 1'b1;

end

end

end

endmodule

module seriadd(data\_a, data\_b, clk, reset, out, cout);

input [3:0] data\_a, data\_b;

input clk, reset;

output cout;

output [3:0] out;

reg [3:0] out;

reg [2:0] count;

reg enable, cout;

wire wire\_a, wire\_b, cout\_temp, cin, sum;

piso piso\_a(clk, enable, reset, data\_a, wire\_a);

piso piso\_b(clk, enable, reset, data\_b, wire\_b);

full\_adder adder(wire\_a, wire\_b, cin, sum, cout\_temp);

d\_flipflop dff(cout\_temp, clk, enable, reset, cin);

always @ (posedge clk or posedge reset) begin

if (reset) begin

enable = 1; out = 4'b0000;

end

else begin

cout = cout\_temp;

out = out >> 1;

out[3] = sum;

end

end

endmodule

**Serial Adder Simple Code(Found on some website, no guarantee of functioning):**

**Verilog CODE:**  
  
//serial adder for N bits. Note that we dont have to mention N here.   
**module** serial\_adder   
    (   **input** clk,reset,  //clock and reset  
        **input** a,b,cin,  //note that cin is used for only first iteration.  
        **output** **reg** s,cout  //note that s comes out at every clock cycle and cout is valid only for last clock cycle.  
        );  
  
**reg** c,flag;  
  
**always**@(**posedge** clk **or** **posedge** reset)  
**begin**  
    **if**(reset == 1) **begin** //active high reset  
        s = 0;  
        cout = c;  
        flag = 0;  
    **end** **else** **begin**  
        **if**(flag == 0) **begin**  
            c = cin;  //on first iteration after reset, assign cin to c.  
            flag = 1;  //then make flag 1, so that this if statement isnt executed any more.  
        **end**   
        cout = 0;  
        s = a ^ b ^ c;  //SUM  
        c = (a & b) | (c & b) | (a & c);  //CARRY  
    **end**   
**end**  
  
**endmodule**   
  
**TESTBENCH CODE:**  
 **module** tb;  
  
    // Inputs  
    **reg** clk;  
    **reg** reset;  
    **reg** a;  
    **reg** b;  
    **reg** cin;  
  
    // Outputs  
    **wire** s;  
    **wire** cout;  
  
    // Instantiate the Unit Under Test (UUT)  
    serial\_adder uut (  
        .clk(clk),   
        .reset(reset),   
        .a(a),   
        .b(b),   
        .cin(cin),   
        .s(s),   
        .cout(cout)  
    );  
  
//generate clock with 10 ns clock period.  
    **always**  
        #5 clk = ~clk;  
          
    **initial** **begin**  
        // Initialize Inputs  
        clk = 1;  
        reset = 0;  
        a = 0;  
        b = 0;  
        cin = 0;  
        reset = 1;    
        #20;  
        reset = 0;  
        //add two 4 bit numbers, 1111 + 1101 = 11101  
        a = 1; b = 1; cin = 1;    #10;  
        a = 1; b = 0; cin = 0;  #10;  
        a = 1; b = 1; cin = 0;  #10;  
        a = 1; b = 1; cin = 0;  #10;  
        reset = 1;  
        #10;  
        reset = 0;  
        //add two 5 bit numbers, 11011 + 10001 = 101101  
        a = 1; b = 1; cin = 1;    #10;  
        a = 1; b = 0; cin = 0;  #10;  
        a = 0; b = 0; cin = 0;  #10;  
        a = 1; b = 0; cin = 0;  #10;  
        a = 1; b = 1; cin = 0;  #10;  
        reset = 1;  
        #10;  
  
    **end**  
        
**endmodule**

**Seven Segment Display**

module sevseg(

input wire clk,

output reg [6:0] HEX0,

output reg [6:0] HEX1,

output reg [6:0] HEX2,

output reg [6:0] HEX3,

output reg [6:0] HEX4,

output reg [6:0] HEX5

);

integer i=0,j=0;

// 7-segment encoding (Common Anode: Active LOW)

function [6:0] seg\_decode;

input [3:0] digit;

case (digit)

4'h0: seg\_decode = 7'b1000000;

4'h1: seg\_decode = 7'b1111001;

4'h2: seg\_decode = 7'b0100100;

4'h3: seg\_decode = 7'b0110000;

4'h4: seg\_decode = 7'b0011001;

4'h5: seg\_decode = 7'b0010010;

4'h6: seg\_decode = 7'b0000010;

4'h7: seg\_decode = 7'b1111000;

4'h8: seg\_decode = 7'b0000000;

4'h9: seg\_decode = 7'b0010000;

4'hE: seg\_decode = 7'b0000110;

4'hC: seg\_decode = 7'b1000110;

default: seg\_decode = 7'b1111111;

endcase

endfunction

always@(posedge clk) begin

if(i==0)begin

HEX0 = seg\_decode(4'h7);

HEX1 = seg\_decode(4'h8);

HEX2 = seg\_decode(4'h0);

HEX3 = seg\_decode(4'h1);

HEX4 = seg\_decode(4'hC);

HEX5 = seg\_decode(4'hE);

end

if(i==1)begin

HEX0 = seg\_decode(4'h5);

HEX1 = seg\_decode(4'h0);

HEX2 = seg\_decode(4'h2);

HEX3 = seg\_decode(4'h1);

HEX4 = seg\_decode(4'hC);

HEX5 = seg\_decode(4'hE);

end

j=j+1;

i=j%2;

end

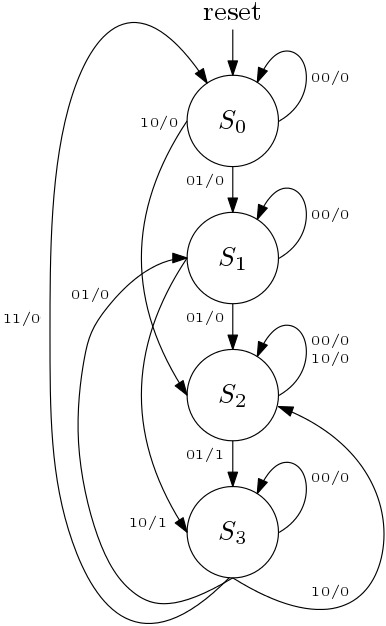
endmodule

**Mealy Sequence Detector:**  
module mealyofive(input clk, rst, ip, output reg op);  
reg [2:0]y,Y,A,B,C,D,E,F,G,H;  
  
initial begin  
A=4'b000; B=4'b001; C=4'b010; D=4'b011;  
E=4'b100; F=4'b101; G=4'b110; H=4'b111;  
y=A; Y=3'bxxx; op=1'b0;  
//y=currentstate Y=nextstate op=currentoutput  HEX1=op HEX0=st  
end  
  
always@(y,ip)begin //COMBINATIONAL LOGIC  
case(y)  
A: if(ip==0) begin Y=B; op=1'b0; end  
else    begin Y=A; op=1'b0; end  
B: if(ip==0) begin Y=C; op=1'b0; end  
else    begin Y=A; op=1'b0; end  
C: if(ip==0) begin Y=D; op=1'b0; end  
else    begin Y=A; op=1'b0; end  
D: if(ip==0) begin Y=E; op=1'b0; end  
else    begin Y=A; op=1'b0; end  
E: if(ip==0) begin Y=F; op=1'b0; end  
else    begin Y=A; op=1'b0; end  
F: if(ip==1) begin Y=G; op=1'b0; end  
else    begin Y=F; op=1'b0; end  
G: if(ip==0) begin Y=H; op=1'b0; end  
else    begin Y=A; op=1'b0; end  
H: if(ip==1) begin Y=G; op=1'b1; end  
else    begin Y=A; op=1'b0; end  
default:    begin Y=3'bxxx;  op=1'bx; end  
endcase  
end  
  
always@(posedge clk)begin //SEQUENTIAL PART  
if(rst==1) begin y<=A;          end  
else     begin y<=Y;          end  
  
end  
endmodule

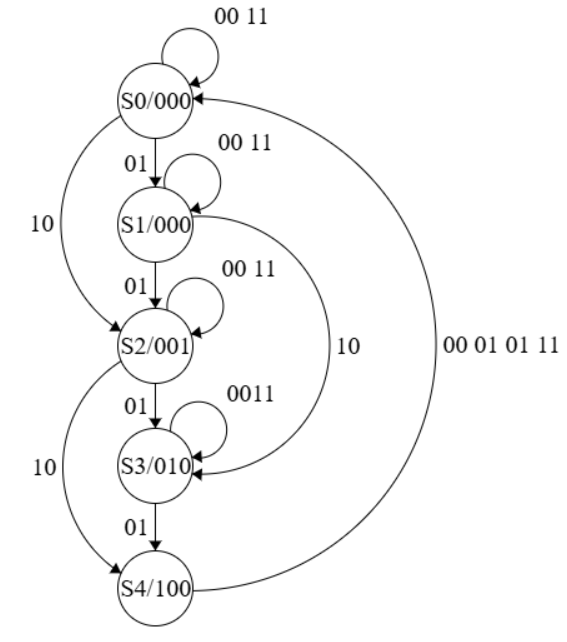
**Moore Sequence Detector:**

module mooreofive(input clk, rst, ip, output reg [3:0]y, output reg op);  
reg [3:0]Y,A,B,C,D,E,F,G,H,I;  
reg ph;  
  
initial begin  
A=4'b0000; B=4'b0001; C=4'b0010; D=4'b0011;  
E=4'b0100; F=4'b0101; G=4'b0110; H=4'b0111;  
I=4'b1000;  
y=A; Y=4'bxxxx; op=1'bx;  
//y=currentstate Y=nextstate op=currentoutput ph=placeholder HEX1=op HEX0=st  
end  
  
always@(y,ip)begin //COMBINATIONAL LOGIC  
case(y)  
A: if(ip==0) begin Y=B; ph=1'b0; end  
else    begin Y=A; ph=1'b0; end  
B: if(ip==0) begin Y=C; ph=1'b0; end  
else    begin Y=A; ph=1'b0; end  
C: if(ip==0) begin Y=D; ph=1'b0; end  
else    begin Y=A; ph=1'b0; end  
D: if(ip==0) begin Y=E; ph=1'b0; end  
else    begin Y=A; ph=1'b0; end  
E: if(ip==0) begin Y=F; ph=1'b0; end  
else    begin Y=A; ph=1'b0; end  
F: if(ip==1) begin Y=G; ph=1'b0; end  
else    begin Y=F; ph=1'b0; end  
G: if(ip==0) begin Y=H; ph=1'b0; end  
else    begin Y=A; ph=1'b0; end  
H: if(ip==1) begin Y=I; ph=1'b1; end  
else    begin Y=A; ph=1'b0; end  
I: if(ip==0) begin Y=H; ph=1'b0; end  
else    begin Y=A; ph=1'b0; end  
default:    begin Y=4'bxxxx;  ph=1'bx; end  
endcase  
end  
  
always@(posedge clk)begin //SEQUENTIAL PART  
if(rst==1) begin y<=A; op=1'b0; end  
else     begin y<=Y; op<=ph;  end  
  
end  
  
endmodule

**FSM for 1st Task:**

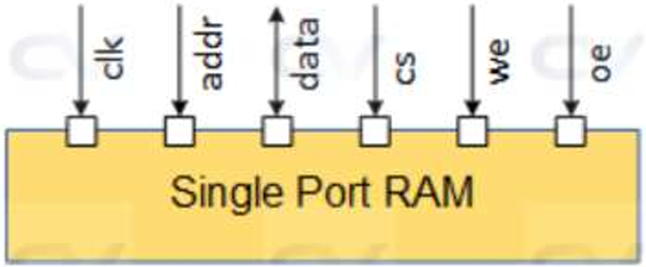
  
module vendmach(input clk, rst, input [1:0]ip, output reg op);  
reg [1:0]y,Y,A,B,C,D;  
  
initial begin  
A=2'b00; B=2'b01; C=2'b10; D=2'b11;  
y=A; Y=2'bxx; op=1'b0;  
//y=currentstate Y=nextstate op=currentoutput  HEX1=op HEX0=st  
end  
  
always@(y,ip)begin //COMBINATIONAL LOGIC  
case(y)  
A: if(ip==2'b01)         begin Y=B; op=1'b0; end  
   else if (ip==2'b10)   begin Y=C; op=1'b0; end  
   else if (ip==2'b00)   begin Y=A; op=1'b0; end  
B: if(ip==2'b01)         begin Y=C; op=1'b0; end  
   else if (ip==2'b10)   begin Y=D; op=1'b1; end  
   else if (ip==2'b00)   begin Y=B; op=1'b0; end  
C: if(ip==2'b01)         begin Y=D; op=1'b1; end  
   else if (ip==2'b10)   begin Y=C; op=1'b0; end  
   else if (ip==2'b00)   begin Y=C; op=1'b0; end  
D: if(ip==2'b01)         begin Y=B; op=1'b0; end  
   else if (ip==2'b10)   begin Y=C; op=1'b0; end  
   else if (ip==2'b00)   begin Y=D; op=1'b0; end  
   else if (ip==2'b11)   begin Y=A; op=1'b0; end //jump to start after transaction  
default:     begin Y=2'bxx;  op=1'bx; end  
endcase  
end  
  
always@(posedge clk)begin //SEQUENTIAL PART  
if(rst==1) begin y<=A;          end  
else       begin y<=Y;          end  
end  
endmodule

**FSM for 2nd Task:**

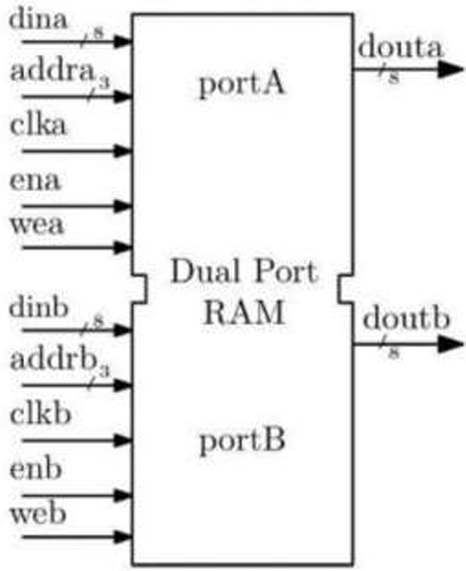


module venmachm(  
    input clk,  
    input rst,  
    input [1:0] coin,  
    output reg chocolate,  
output reg chips,  
output reg biscuit  
);  
  
parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010, S3 = 3'b011 ,S4= 3'b100;  
reg[2:0]  state,next\_state;  
  
    always @(posedge clk or posedge rst) begin  
        if (rst)  
            state <= S0;  
        else  
            state <= next\_state;  
    end  
  
    always @(\*) begin  
        next\_state = state;  
        chocolate = 0;  
 chips=0;  
 biscuit=0;  
        case (state)  
            S0: begin  
                if (coin == 2'b01)  
                    next\_state = S1;  
                else if (coin == 2'b10)  
                    next\_state = S2;  
            end  
            S1: begin  
                if (coin == 2'b01)  
                    next\_state = S2;  
                else if (coin == 2'b10)  
                    next\_state = S3;  
            end  
            S2: begin  
biscuit =1;  
                if (coin == 2'b01)  
                    next\_state = S3;  
 else if(coin == 2'b10)  
 next\_state = S4;  
            end  
            S3: begin  
chocolate =1;  
                if(coin==2'b01)  
next\_state=S4;  
  
            end  
S4: begin  
chips =1;  
next\_state=S0;  
  
            end  
        endcase  
    end  
  
endmodule  
module vending\_machine\_tb;  
    reg clk;  
    reg rst;  
    reg [1:0] coin;  
    wire chocolate,chips,biscuit;

**Single Port RAM:**

  
module singpram(  
input [2:0] data,  
input [5:0] addr,  
input we, clk,  
output [2:0] q  
);  
  
reg [2:0] ram[63:0];  
reg [5:0] addr\_reg;  
  
always @ (posedge clk)begin  
// Write  
if (we)  
ram[addr] <= data;  
addr\_reg <= addr;  
end  
  
assign q = ram[addr\_reg];  
  
endmodule

**Dual Port RAM (No guarantee of working):**

  
module Dual\_Port\_RAM (   
input clk,   
input we\_a,   
input we\_b,   
input [3:0] addr\_a,   
input [3:0] addr\_b,   
input [7:0] din\_a,   
input [7:0] din\_b,   
output reg [7:0] dout\_a,   
output reg [7:0] dout\_b   
);

reg [7:0] ram [15:0];

always @(posedge clk) begin   
 // Port A operations   
if (we\_a)   
ram[addr\_a] <= din\_a;   
dout\_a <= ram[addr\_a]; // Ensure read after write is handled correctly

// Port B operations   
if (we\_b)   
ram[addr\_b] <= din\_b;   
dout\_b <= ram[addr\_b]; // Ensure read after write is handled correctly

end

endmodule

**Single Port ROM:**module singpram(  
input [5:0] addr,  
input clk,  
output [3:0] q  
);  
reg [3:0] rom[63:0];  
reg [5:0] addr\_reg;  
  
initial begin  
rom[0]=4'b0000;  
rom[1]=4'b0001;  
rom[2]=4'b0010;  
rom[3]=4'b0011;  
rom[4]=4'b0100;  
rom[5]=4'b0101;  
rom[6]=4'b0110;  
rom[7]=4'b0111;  
rom[8]=4'b1000;  
rom[9]=4'b1001;  
rom[10]=4'b1010;  
rom[11]=4'b1011;  
rom[12]=4'b1100;  
rom[13]=4'b1101;  
rom[14]=4'b1110;  
rom[15]=4'b1111;  
end  
  
always @ (posedge clk)begin  
addr\_reg <= addr;  
end  
  
assign q = rom[addr\_reg];  
endmodule

**Dual Port ROM (No guarantee of working):**

module Dual\_Port\_ROM (   
 input [3:0] addr\_a, addr\_b, // Two 4-bit address inputs   
 output reg [7:0] data\_a, data\_b // Two 8-bit data outputs   
);

// ROM Memory Initialization (Preloaded values)   
 reg [7:0] rom [15:0];   
 initial begin   
 rom[0] = 8'h12; rom[1] = 8'h34; rom[2] = 8'h56; rom[3] = 8'h78;   
 rom[4] = 8'h9A; rom[5] = 8'hBC; rom[6] = 8'hDE; rom[7] = 8'hF0;   
 rom[8] = 8'h11; rom[9] = 8'h22; rom[10] = 8'h33; rom[11] = 8'h44;   
 rom[12] = 8'h55; rom[13] = 8'h66; rom[14] = 8'h77; rom[15] = 8'h88;   
 end

always @(\*) begin   
 data\_a = rom[addr\_a]; // Read data from address A   
 data\_b = rom[addr\_b]; // Read data from address B   
 end

endmodule

**DSD Topics:**

**//Gate level modelling**

module test(a,b,i,j,k,l,m,n,o);

input a,b;

output i,j,k,l,m,n,o;

and(i,a,b);

or(j,a,b);

not(k,a);

nand(l,a,b);

nor(m,a,b);

xor(n,a,b);

xnor(o,a,b);

endmodule

**//Dataflow modelling**

module test1(a,b,i,j,k,l,m,n,o);

input a,b;

output i,j,k,l,m,n,o;

assign i = a&b;

assign j = a|b;

assign k = ~a;

assign l = ~(a&b);

assign m = ~(a|b);

assign n = (a^b);

assign o = ~(a^b);

endmodule

**Exp1 Behavioral**

//AND Gate

module and\_gate\_b(Y, A, B);

input A,B;

output reg Y;

always @ (A or B) begin

if (A == 1'b1 & B == 1'b1) begin Y = 1'b1; end

else begin Y = 1'b0; end

end

endmodule

//OR Gate

module or\_gate\_b(y,a,b);

input a,b;

output reg y;

always @(a or b)begin

if(a==1'b0 & b==1'b0) begin y = 1'b0; end

else y = 1'b1;

end

endmodule

//Not Gate

module not\_gate\_b(Y,A);

input A;

output reg Y;

always@(A) begin

if (A == 1'b0) begin Y = 1'b1; end

else if(A==1'b1) begin Y = 1'b0; end

end

endmodule

//NAND Gate

module nand\_gate\_b(y,a,b);

input a,b;

output reg y;

always@(a or b) begin

if(a==1'b1 & b==1'b1) begin y=1'b0; end

else y=1'b1; end

endmodule

//NOR gate

module nor\_gate\_b(y,a,b);

input a,b;

output reg y;

always @(a or b)begin

if(a==1'b0 & b==1'b0) begin y = 1'b1; end

else y = 1'b0; end

endmodule

//EXOR Gate:

module exor\_gate\_b(y,a,b);

input a,b;

output reg y;

always @(a or b)begin

if(a==1'b0 & b==1'b0) begin y = 1'b0; end

if(a==1'b1 & b==1'b1) begin y = 1'b0; end

if(a==1'b0 & b==1'b1) begin y = 1'b1; end

if(a==1'b1 & b==1'b0) begin y = 1'b1; end

end

endmodule

//EXNOR gate:

module enxor\_gate\_b(y,a,b);

input a,b;

output reg y;

always @(a or b)begin

if(a==1'b0 & b==1'b0) begin y = 1'b1; end

if(a==1'b1 & b==1'b1) begin y = 1'b1; end

if(a==1'b0 & b==1'b1) begin y = 1'b0; end

if(a==1'b1 & b==1'b0) begin y = 1'b0; end

end

endmodule

**HA HS FA FS**

**Half adder:**

//gate level

module half\_add(a,b,c,s);

input a,b;

output c,s;

xor (s,a,b);

and (c,a,b);

endmodule

//data flow

module half\_add(a,b,c,s);

input a,b;

output c,s;

assign s = a^b;

assign c = a&b;

endmodule

//common test bench

module half\_add\_tb;

reg a,b;

wire c,s;

half\_add dut(.a(a), .b(b), .c(c), .s(s));

initial

begin

a=0; b=0; #100;

a=0; b=1; #100;

a=1; b=0; #100;

a=1; b=1; #100;

end

endmodule

**Full adder:**

//gatelevel

module full\_add(a,b,c,s,co);

input a,b,c;

output s,co;

wire w1,w2,w3,w4;

xor(w1,a,b);

xor(s,w1,c);

and(w3,a,b);

and(w4,c,w1);

or(co,w3,w4);

endmodule

//dataflow:

module full\_add(a,b,c,s,co);

input a,b,c;

output s,co;

assign s = a^b^c;

assign co = (a&b)|c&(a^b);

endmodule

//

module full\_add\_tb;

reg a,b,c;

wire s,co;

full\_add uut(.a(a), .b(b), .c(c), .s(s), .co(co));

initial

begin

a=0; b=0; c=0; #100;

a=0; b=0; c=1; #100;

a=0; b=1; c=0; #100;

a=0; b=1; c=1; #100;

a=1; b=0; c=0; #100;

a=1; b=0; c=1; #100;

a=1; b=1; c=0; #100;

a=1; b=1; c=1; #100;

end

endmodule

**Half subtractor:**

//gatelevel

module half\_sub(

input a, b,

output d, bo);

wire x;

xor (d,a,b);

not (x,a);

and (bo,x,b);

endmodule

//dataflow

module half\_sub(

input a, b,

output d, bo);

assign d = a^b;

assign bo = ((~a)&b);

endmodule

//common testbench

module half\_sub\_tb;

reg a,b;

wire d,bo;

half\_sub dut(.a(a), .b(b), .d(d), .bo(bo));

initial

begin

a=0; b=0; #100;

a=0; b=1; #100;

a=1; b=0; #100;

a=1; b=1; #100;

end

endmodule

**Full Subtractor:**

//gatelevel

module full\_sub(

input a,b,bi,

output d,bo);

wire x1,x2,x3,x4;

xor (x1,a,b);

not (x2,x1);

not (x3,a);

and (x4,x3,b);

xor (d,x1,bi);

and (x5,x2,bi);

or (bo,x5,x4);

endmodule

//dataflow

module full\_sub(

input a,b,bi,

output d,bo);

assign d = ((a^b)^bi);

assign bo = (((~a)&b) + (bi&(~(a^b))));

endmodule

//common testbench

module full\_sub\_tb;

reg a,b,bi;

wire d,bo;

full\_sub dut(.a(a), .b(b), .bi(bi), .d(d), .bo(bo));

initial

begin

a=0; b=0; bi=0; #100;

a=0; b=0; bi=1; #100;

a=0; b=1; bi=0; #100;

a=0; b=1; bi=1; #100;

a=1; b=0; bi=0; #100;

a=1; b=0; bi=1; #100;

a=1; b=1; bi=0; #100;

a=1; b=1; bi=1; #100;

end

endmodule

**4 bit Parallel Adder**

//Full Adder module

module FAforPA(a,b,cin,sum,cout);

input a,b,cin;

output wire sum,cout;

wire s1,c1,c2,c3;

xor(s1,a,b);

xor(sum,s1,cin);

and(c1,a,b);

and(c2,b,cin);

and(c3,a,cin);

or(cout,c1,c2,c3);

endmodule

//4 bit parallel adder using instantiation of Full Adder

module PA(a,b,cin,sum,cout);

input [3:0]a,b;

input cin;

output wire [3:0]sum;

output cout;

FAforPA FA1(a[0],b[0],cin,sum[0],cout1);

FAforPA FA2(a[1],b[1],cout1,sum[1],cout2);

FAforPA FA3(a[2],b[2],cout2,sum[2],cout3);

FAforPA FA4(a[3],b[3],cout3,sum[3],cout);

endmodule

//Testbench for parallel adder

module PA\_tb;

reg [3:0]a;

reg [3:0]b;

reg cin; //inputs

wire [3:0]sum;

wire cout; // Outputs

// Instantiate the Unit Under Test (UUT)

PA dut(.a(a), .b(b), .cin(cin), .sum(sum), .cout(cout));

initial

begin

// Initialize Inputs

assign cin = 1'b0;

a = 5; b = 5; #100;

a = 15; b = 12; #100;

end

endmodule

**Flip-flop codes:**

//SR Flip Flop Verilog code

module SR\_FF(Q,QB,S,R,CLK);

input S,R,CLK;

output Q,QB;

reg Q,QB;

initial begin

Q=1'b0;

QB=1'b1;

end

always @(posedge CLK)

begin

case({S,R})

2'b00:Q=Q;

2'b01:Q=0;

2'b10:Q=1;

2'b11:Q=1'bx;

endcase

QB=~Q;

end

endmodule

//SR Flip Flop Testbench code

module SR\_FF\_TB;

reg S; reg R;

reg CLK;

wire Q; wire QB;

SR\_FF uut ( .Q(Q), .QB(QB), .S(S), .R(R), .CLK(CLK));

always #100 CLK=~CLK;

initial begin

CLK=1;

#200 S=1; R=0;

#200 S=0; R=0;

#200 S=0; R=1;

#200 S=1; R=1;

end

endmodule

//JK Flip Flop Verilog code

module JK\_FF(Q,QB,J,K,CLK);

input J,K,CLK;

output Q,QB;

reg Q,QB;

initial begin

Q=1'b0;

QB=1'b1;

end

always @(posedge CLK)

begin

case({J,K})

2'b00:Q=Q;

2'b01:Q=0;

2'b10:Q=1;

2'b11:Q=~Q;

endcase

QB=~Q;

end

endmodule

//JK Flip Flop Testbench code

module JK\_FF\_TB;

reg J;

reg K;

reg CLK;

wire Q;

wire QB;

JK\_FF uut ( .Q(Q), .QB(QB), .J(J), .K(K), .CLK(CLK));

always #100 CLK=~CLK;

initial begin

CLK=1;

#200 J=1;K=0;

#200 J=0; K=0;

#200 J=0; K=1;

#200 J=1; K=1;

end

endmodule

//T Flip Flop Verilog code

module T\_FF(Q,QB,T,CLK);

input T,CLK;

output Q,QB;

reg Q=0,QB;

initial begin

Q=1'b0;

QB=1'b1;

end

always @(posedge CLK)

begin

case(T)

1'b0:Q=Q;

1'b1:Q=~Q;

endcase

QB=~Q;

end

endmodule

//T Flip Flop Testbench code

module T\_FF\_TB;

reg T;

reg CLK;

wire Q;

wire QB;

T\_FF uut ( .Q(Q), .QB(QB), .T(T), .CLK(CLK));

always #100 CLK=~CLK;

initial begin

CLK=1;

#200 T=0;

#200 T=1;

#200 T=0;

#200 T=1;

end

endmodule

//D Flip Flop Verilog code

module D\_FF(Q,QB,D,CLK);

input D,CLK;

output Q,QB;

reg Q,QB;

initial begin

Q=1'b0;

QB=1'b1;

end

always @(posedge CLK)

begin

Q=D;

QB=~Q;

end

endmodule

//D Flip Flop Testbench code

module D\_FF\_TB;

reg D;

reg CLK;

wire Q;

wire QB;

D\_FF uut ( .Q(Q), .QB(QB), .D(D), .CLK(CLK));

always #100 CLK=~CLK;

initial begin

CLK=1;

#200 D=1;

#200 D=0;

end

endmodule

**Ripple Counter**

//D flip flop for Ripple Counter

module dff (input d,

input clk,

input rstn,

output reg q,

output qn);

always @ (posedge clk or negedge rstn)

if (!rstn)

q <= 0;

else

q <= d;

assign qn = ~q;

endmodule

//Ripple Counter using Dff instantiation

module ripple ( input clk,

input rstn,

output [3:0] out);

wire q0;

wire qn0;

wire q1;

wire qn1;

wire q2;

wire qn2;

wire q3;

wire qn3;

dff dff0 ( .d (qn0),

.clk (clk),

.rstn (rstn),

.q (q0),

.qn (qn0));

dff dff1 ( .d (qn1),

.clk (q0),

.rstn (rstn),

.q (q1),

.qn (qn1));

dff dff2 ( .d (qn2),

.clk (q1),

.rstn (rstn),

.q (q2),

.qn (qn2));

dff dff3 ( .d (qn3),

.clk (q2),

.rstn (rstn),

.q (q3),

.qn (qn3));

assign out = {qn3, qn2, qn1, qn0};

endmodule

//Testbench for ripple counter

module tb\_ripple;

reg clk;

reg rstn;

wire [3:0] out;

ripple r0 ( .clk (clk),

.rstn (rstn),

.out (out));

always #5 clk = ~clk;

initial begin

rstn <= 0;

clk <= 0;

repeat (4) @ (posedge clk);

rstn <= 1;

repeat (25) @ (posedge clk);

$finish;

end

endmodule

**ALU**

//ALU verilog code

module alu(

input [7:0] A,B, // ALU 8-bit Inputs

input [3:0] ALU\_Sel,// ALU Selection

output [7:0] ALU\_Out, // ALU 8-bit Output

output CarryOut // Carry Out Flag

);

reg [7:0] ALU\_Result;

wire [8:0] tmp;

assign ALU\_Out = ALU\_Result; // ALU out

assign tmp = {1'b0,A} + {1'b0,B};

assign CarryOut = tmp[8]; // Carryout flag

always @(\*)

begin

case(ALU\_Sel)

4'b0000: // Addition

ALU\_Result = A + B ;

4'b0001: // Subtraction

ALU\_Result = A - B ;

4'b0010: // Multiplication

ALU\_Result = A \* B;

4'b0011: // Division

ALU\_Result = A/B;

4'b0100: // Logical shift left

ALU\_Result = A<<1;

4'b0101: // Logical shift right

ALU\_Result = A>>1;

4'b0110: // Rotate left

ALU\_Result = {A[6:0],A[7]};

4'b0111: // Rotate right

ALU\_Result = {A[0],A[7:1]};

4'b1000: // Logical and

ALU\_Result = A & B;

4'b1001: // Logical or

ALU\_Result = A | B;

4'b1010: // Logical xor

ALU\_Result = A ^ B;

4'b1011: // Logical nor

ALU\_Result = ~(A | B);

4'b1100: // Logical nand

ALU\_Result = ~(A & B);

4'b1101: // Logical xnor

ALU\_Result = ~(A ^ B);

4'b1110: // Greater comparison

ALU\_Result = (A>B)?8'd1:8'd0 ;

4'b1111: // Equal comparison

ALU\_Result = (A==B)?8'd1:8'd0 ;

default: ALU\_Result = A + B ;

endcase

end

endmodule

//ALU Testbench

module tb\_alu;

//Inputs

reg[7:0] A,B;

reg[3:0] ALU\_Sel;

//Outputs

wire[7:0] ALU\_Out;

wire CarryOut;

// Verilog code for ALU

integer i;

alu test\_unit(

A,B, // ALU 8-bit Inputs

ALU\_Sel,// ALU Selection

ALU\_Out, // ALU 8-bit Output

CarryOut // Carry Out Flag

);

initial begin

// hold reset state for 100 ns.

A = 8'h0A;

B = 4'h02;

ALU\_Sel = 4'h0;

for (i=0;i<=15;i=i+1)

begin

ALU\_Sel = ALU\_Sel + 8'h01;

#10;

end

A = 8'hF6;

B = 8'h0A;

end

endmodule  
  
  
  
FLIPFLOPS:

Tflipflop:

module tff(t, clk, rst, q, qb);

input t, clk, rst;

output q, qb;

reg q, qb;

reg temp = 0;

always @(posedge clk, posedge rst) begin

if (rst == 0) begin

if (t == 1) begin

temp = ~temp;

end else temp = temp;

end

q = temp;

qb = ~temp;

end

endmodule

DFLIPFLOP:

module dff(d, clk, rst, q, qb);

input d, clk, rst;

output q, qb;

reg q, qb;

reg temp = 0;

always @(posedge clk, posedge rst) begin

if (rst == 0)

temp = d;

else

temp = temp;

q = temp;

qb = ~temp ;

end

endmodule

SR FLIPFLOP:

module srff(s, r, clk, rst, q, qb);

input s, r, clk, rst;

output q, qb;

reg q, qb;

reg [1:0] sr;

always @(posedge clk, posedge rst) begin

sr = {s, r};

if (rst == 0) begin

case (sr)

2'd1: q = 1'b0;

2'd2: q = 1'b1;

2'd3: q = 1'b1;

default: begin end

endcase

end else begin

q = 1'b0;

end

qb = ~q;

end

endmodule

JK Flip Flop:

module jkff(j, k, clk, rst, q, qb);

input j, k, clk, rst;

output q, qb;

reg q, qb;

reg [1:0] jk;

always @(posedge clk, posedge rst) begin

jk = {j, k};

if (rst == 0) begin

case (jk)

2'd1: q = 1'b0;

2'd2: q = 1'b1;

2'd3: q = ~q;

default: begin end

endcase

end else

q = 1'b0;

qb = ~q;

end

endmodule

4:1 Multiplexer Code:

module mux4\_1(I0,I1,I2,I3,s2,s1,y,en);

input I0,I1,I2,I3,s2,s1,en;

output y;

assign y <= ((~s2)&(~s1)&en&I0)| ((~s2)&(s1)&en&I1)|(s2&(~s1)&en&I2)|(s2&s1&en&I3);

endmodule

1:4 DEMUX code:

module demux (s2,s1,I,en,y0,y1,y2,y3);

input s2,s1,I,en;

output y0,y1,y2,y3;

assign y0 = (~s2)&(~s1)& I& en;

assign y1 = (~s2)& s1& I& en;

assign y2 = s2&(~s1)& I & en;

assign y3 = s2& s1 & I & en;

endmodule