NOTE: code for or gate, and gate, xor gate and not gate has to be used along with main programs.

Verilog Code for Implementation of OR Gate---

```
//OR GATE implementation
module or_gate(X,Y,Z);
input X,Y;
output reg Z;
always @ (X or Y) begin
   if (X==1'b0 && Y==1'b0)
      begin
      Z=1'b0;
      end
      else
      begin
      Z=1'b1;
      end
end
endmodule
```

Verilog Code for Implementation of AND Gate---

```
//AND GATE implementation
module and_gate(X,Y,Z);
input X,Y;
output reg Z;
always @ (X or Y) begin
   if (X==1'b1 && Y==1'b1)
      begin
      Z=1'b1;
      end
      else
      begin
      Z=1'b0;
      end
```

```
end
endmodule
```

Verilog Code for Implementation of NOT Gate---

```
//NOT GATE implementation
module not_gate(X,Z);
input X;
output reg Z;
always @ (X ) begin
   if (X == 1'b1)
      begin
      Z=1'b0;
      end
   else
      begin
      Z=1'b1;
      end
end
endmodule
```

Verilog Code for Implementation of XOR Gate---

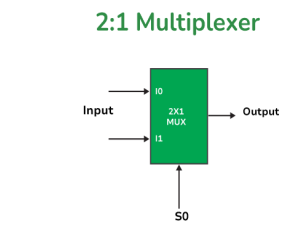
```
//XOR GATE implementation
module xor_gate(X,Y,Z);
input X,Y;
output reg Z;
always @ (X or Y) begin
   if (X == Y)
      begin
      Z=1'b0;
      end
      else
      begin
      Z=1'b1;
```

```
    end
end
endmodule
```

Verilog Code for Implementation of Half Subtractor---

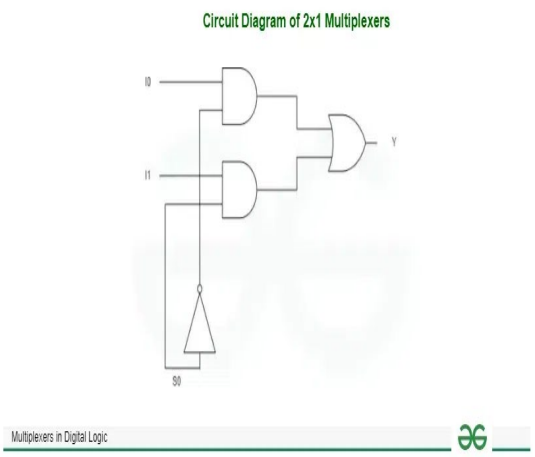
```
//Half Subtractor
module half_sub (A,B,SUB,BOUT);
input A,B;
output SUB,BOUT;
wire a;
xor_gate(.X(A),.Y(B),.Z(SUB));
not_gate(.X(A),.Z(a));
and_gate(.X(a),.Y(B),.Z(BOUT));
endmodule
```

Task 1: - 2:1 mux Implementation



Truth Table

S_0	I_0	I_1	Y
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1



Verilog Code for Implementation of MUX 2x1 ---

This code utilizes not gate, and gate and or gate mentioned above to implement 2:1 mux

```
//MUX 2:1 implementation
module mux_2x1(X,Y,SEL,OUT);
input X,Y,SEL;
output OUT;
wire a,b,c,d;
not_gate uut1 (.X(SEL),.Z(a));
and_gate uut2 (.X(X),.Y(a),.Z(b));
and_gate uut3 (.X(Y),.Y(SEL),.Z(c));
or_gate uut4 (.X(b),.Y(c),.Z(d));
assign OUT = d;
endmodule
```

Task 2: - And Gate using 2:1 mux

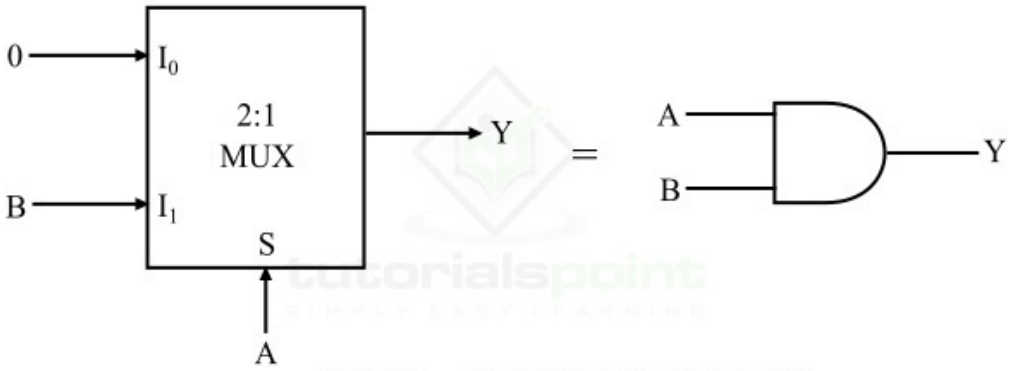


Figure 3 - AND Gate using 2:1 MUX

Verilog Code for Implementation of AND gate using 2:1 mux ---

This code utilizes not gate, and gate, or gate and mux_2x1 mentioned above to implement AND gate

```
//And gate using mux_2x1
module mux_and_gate (A,B,Z);
input A,B;
output Z;
mux_2x1 uut1 (.X(1'b0),.Y(B),.SEL(A),.OUT(Z));
endmodule
```

Task 3: - Or Gate using 2:1 mux

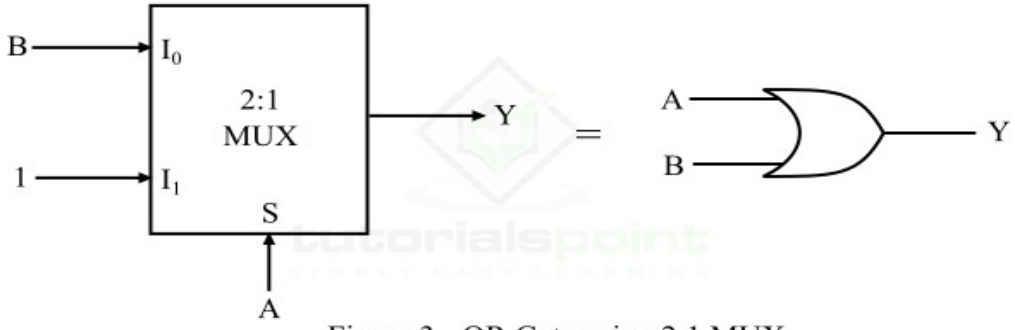


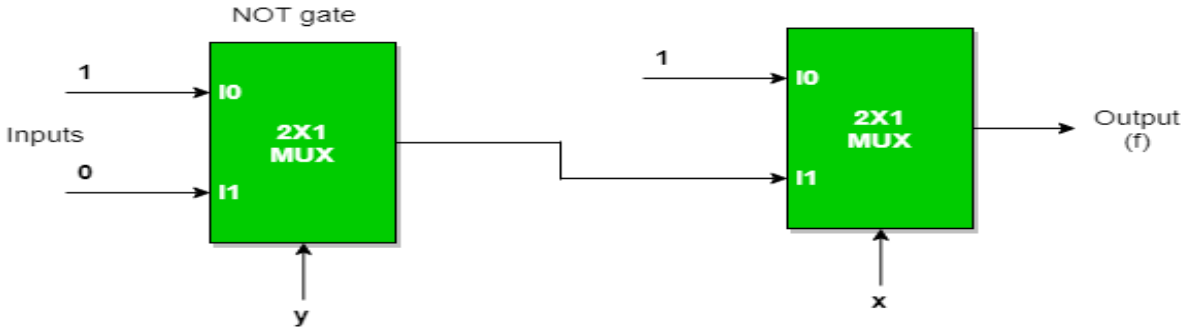
Figure 3 - OR Gate using 2:1 MUX

Verilog Code for Implementation of OR gate using 2:1 mux ---

This code utilizes not gate, and gate, or gate and mux_2x1 mentioned above to implement OR gate

```
//OR gate using mux_2x1
module mux_or_gate (A,B,Z);
input A,B;
output Z;
mux_2x1 uut1 (.X(B),.Y(1'b1),.SEL(A),.OUT(Z));
endmodule
```

Task 4: - NAND Gate using 2:1 mux

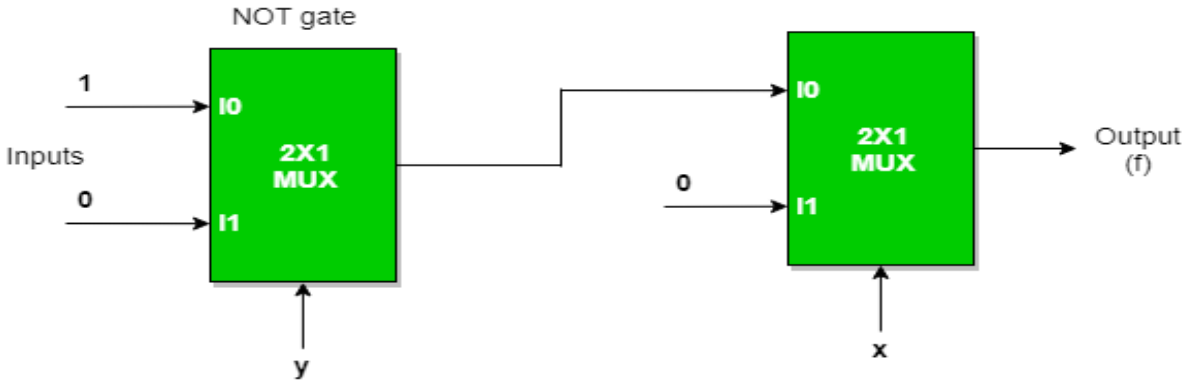


Verilog Code for Implementation of NAND gate using 2:1 mux ---

This code utilizes not gate, and gate, or gate and mux_2x1 mentioned above to implement NAND gate

```
/NAND gate using mux_2x1
module mux_nand_gate (A,B,Z);
input A,B;
output Z;
wire a;
mux_2x1 uut1 (.X(1'b1),.Y(1'b0),.SEL(B),.OUT(a));
mux_2x1 uut2 (.X(1'b1),.Y(a),.SEL(A),.OUT(Z));
endmodule
```

Task 5: - NOR Gate using 2:1 mux

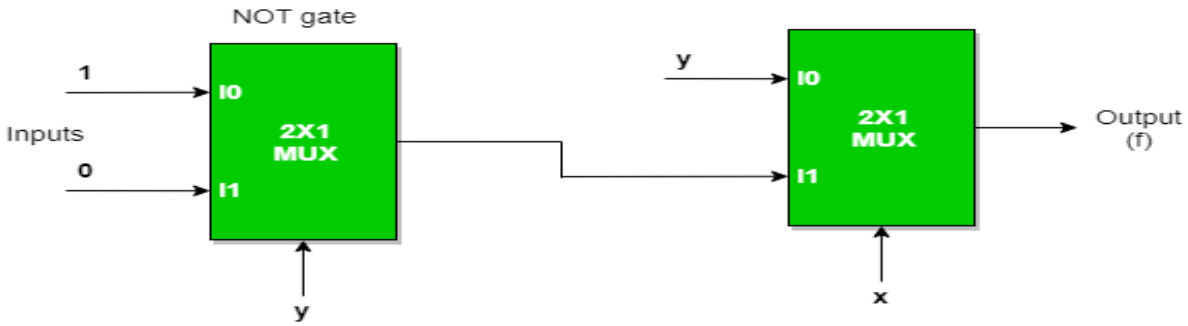


Verilog Code for Implementation of NOR gate using 2:1 mux ---

This code utilizes not gate, and gate, or gate and mux_2x1 mentioned above to implement NOR gate

```
/NOR gate using mux_2x1
module mux_nor_gate (A,B,Z);
input A,B;
output Z;
wire a;
mux_2x1 uut1 (.X(1'b1),.Y(1'b0),.SEL(B),.OUT(a));
mux_2x1 uut2 (.X(a),.Y(1'b0),.SEL(A),.OUT(Z));
endmodule
```

Task 6: - XOR Gate using 2:1 mux

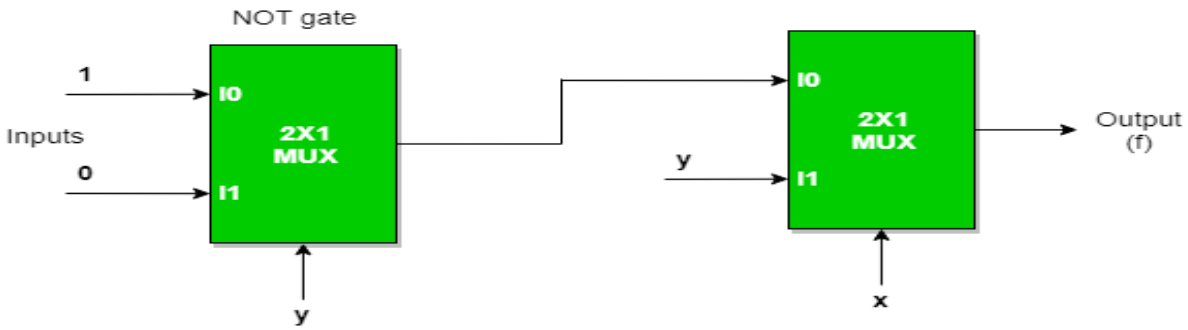


Verilog Code for Implementation of XOR gate using 2:1 mux ---

This code utilizes not gate, and gate, or gate and mux_2x1 mentioned above to implement XOR gate

```
XOR gate using mux_2x1
module mux_xor_gate (A,B,Z);
input A,B;
output Z;
wire a;
mux_2x1 uut1 (.X(1'b1),.Y(1'b0),.SEL(B),.OUT(a));
mux_2x1 uut2 (.X(B),.Y(a),.SEL(A),.OUT(Z));
endmodule
```

Task 7: - XNOR Gate using 2:1 mux



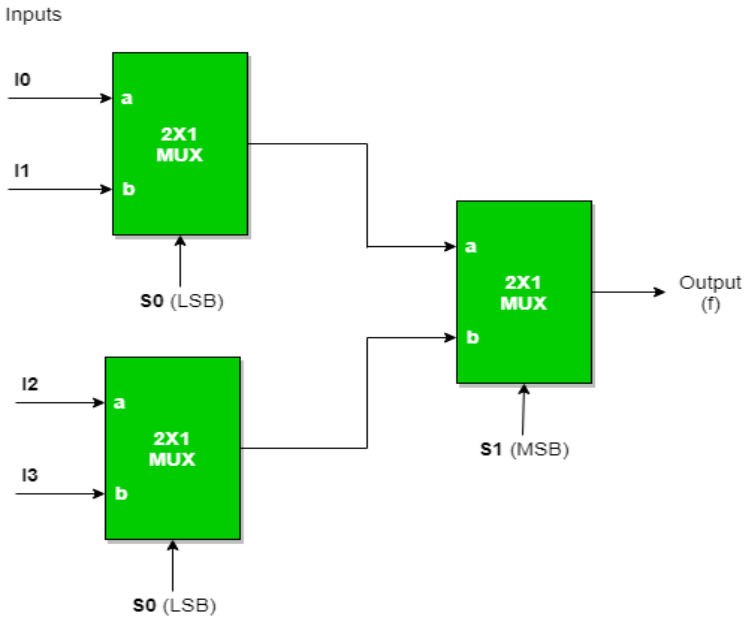
Verilog Code for Implementation of XNOR gate using 2:1 mux ---

This code utilizes not gate, and gate, or gate and mux_2x1 mentioned above to implement XNOR gate

```
/NAND gate using mux_2x1
module mux_xnor_gate (A,B,Z);
input A,B;
```

```
output Z;
wire a;
mux_2x1 uut1 (.X(1'b1),.Y(1'b0),.SEL(B),.OUT(a));
mux_2x1 uut2 (.X(a),.Y(B),.SEL(A),.OUT(Z));
endmodule
```

Task 8: - 4:1 MUX using 2:1 MUX



Truth Table

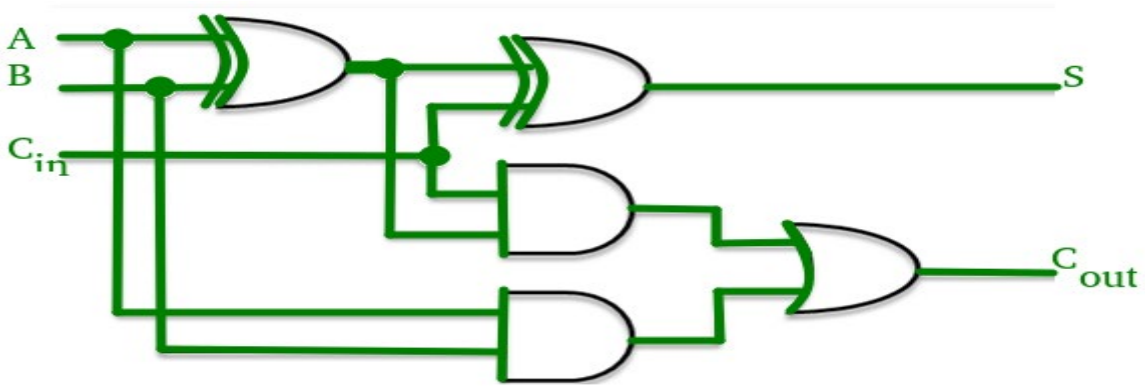
S0	S1	f
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Verilog Code for Implementation of 4:1mux using three 2:1 mux ---

This code utilizes not gate, and gate, or gate and mux_2x1 mentioned above to implement 4:1 mux.

```
module mux_4x1 (input [3:0]IN,input [1:0]SEL,output Z);
wire a,b;
mux_2x1 uut1 (.X(IN[0]),.Y(IN[1]),.SEL(SEL[0]),.OUT(a));
mux_2x1 uut2 (.X(IN[2]),.Y(IN[3]),.SEL(SEL[0]),.OUT(b));
mux_2x1 uut3 (.X(a),.Y(b),.SEL(SEL[1]),.OUT(Z));
endmodule
```

Task 9: - Carry Look-Ahead Adder

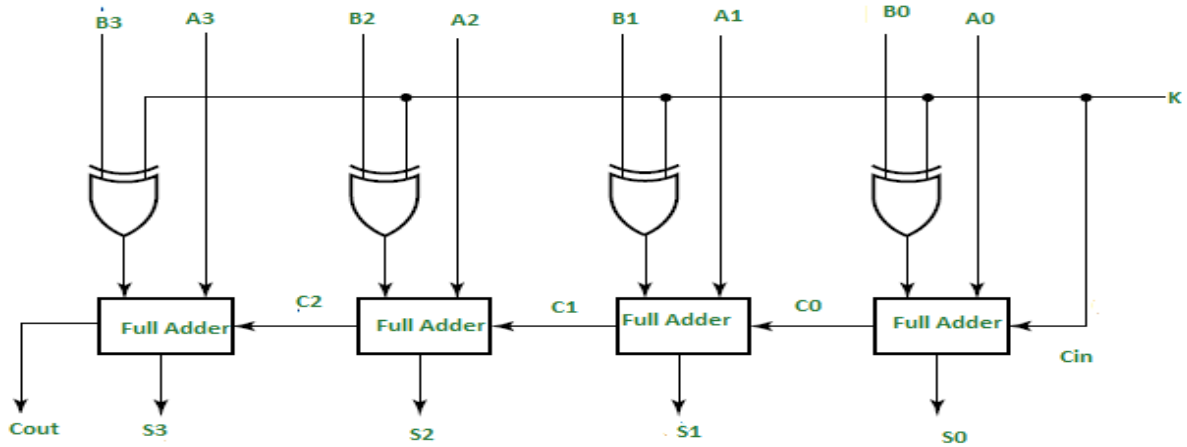


Verilog Code for Implementation of Carry Look-Ahead Adder ---

This code utilizes and gate, or gate and xor gate mentioned above to implement 4:1 mux.

```
/ Carry Look-Ahead Adder gate using mux_2x1
module CLA_FA (A,B,CIN,SUM,COUT);
input A,B,CIN;
output SUM,COUT;
wire a,b,c;
xor_gate uut1 (.X(A),.Y(B),.Z(a));
xor_gate uut2 (.X(a),.Y(CIN),.Z(SUM));
and_gate uut3 (.X(a),.Y(CIN),.Z(b));
and_gate uut4 (.X(A),.Y(B),.Z(c));
or_gate uut5 (.X(b),.Y(c),.Z(COUT));
endmodule
```

Task 10: - 4-bit binary Adder-Subtractor

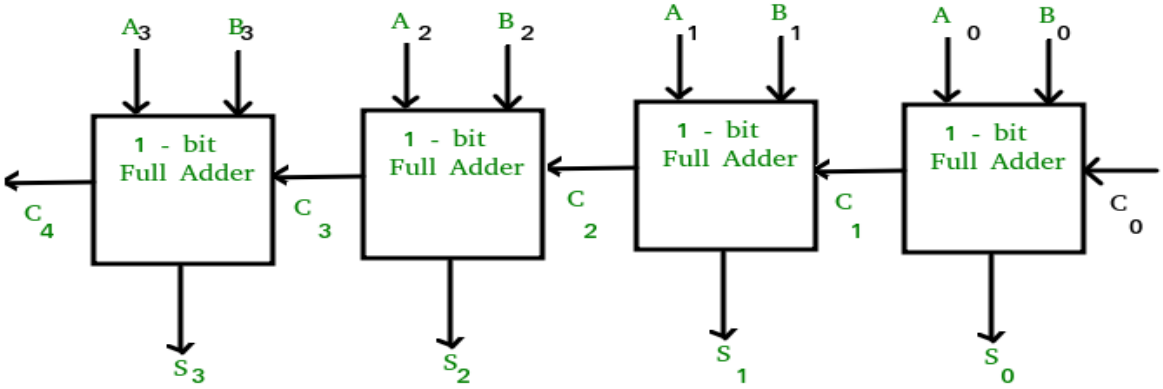


Verilog Code for Implementation of 4-bit binary Adder -Subtractor---

This code utilizes and gate, or gate, xor gate and CLA_FA mentioned above to implement 4-bit binary Adder-Subtractor.

```
module CLA_FA (A,B,CIN,SUM,COUT);
input A,B,CIN;
output SUM,COUT;
wire a,b,c;
xor_gate uut1 (.X(A),.Y(B),.Z(a));
xor_gate uut2 (.X(a),.Y(CIN),.Z(SUM));
and_gate uut3 (.X(a),.Y(CIN),.Z(b));
and_gate uut4 (.X(A),.Y(B),.Z(c));
or_gate uut5 (.X(b),.Y(c),.Z(COUT));
endmodule
```

Task 11: - Ripple Carry Adder



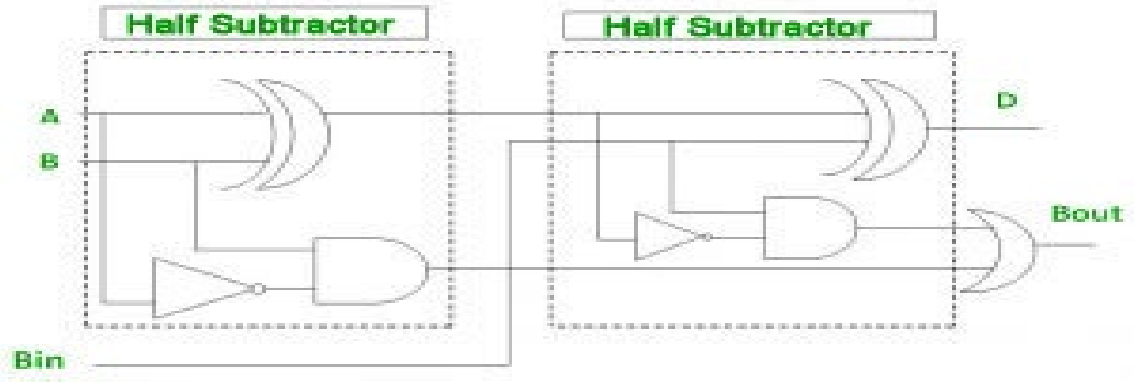
Verilog Code for Implementation of Ripple Carry Adder ---

This code utilizes and gate, or gate, xor gate and CLA_FA mentioned above to implement ripple carry Adder.

```
//Four-bit Ripple Carry full adder
module RC_FA (input [3:0]A, input [3:0]B,input CIN,output [3:0]SUM,output
[3:0]COUT);
//---stage1-----
CLA_FA uut1 (.A(A[0]),.B(B[0]),.CIN(CIN),.SUM(SUM[0]),.COUT(COUT[0]));
//---stage2-----
```

```
CLA_FA uut2 (.A(A[1]),.B(B[1]),.CIN(COUT[0]),.SUM(SUM[1]),.COUT(COUT[1]));
//---stage3-----
CLA_FA uut3 (.A(A[2]),.B(B[2]),.CIN(COUT[1]),.SUM(SUM[2]),.COUT(COUT[2]));//---
stage4-----
CLA_FA uut4 (.A(A[3]),.B(B[3]),.CIN(COUT[2]),.SUM(SUM[3]),.COUT(COUT[3]));
endmodule
```

Task 12: - Full Subtractor



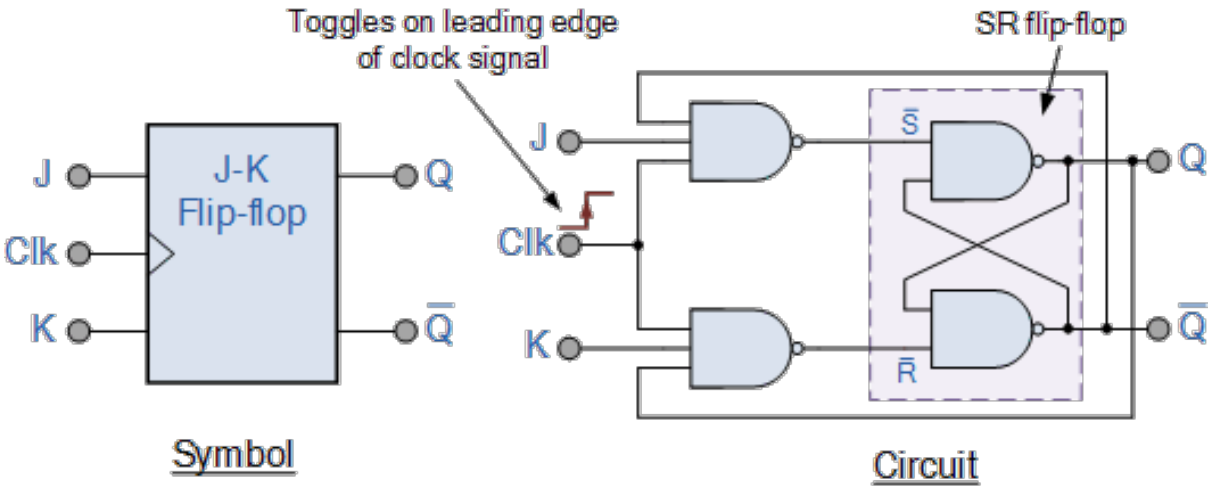
INPUT			OUTPUT	
A	B	Bin	D	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Verilog Code for Implementation of Full Subtractor---

This code utilizes and gate, not gate, xor gate and half sub mentioned above to implement Full Subtractor

```
module full_substractor (A,B,BIN,SUB,BOUT);
input A,B,BIN;
output SUB,BOUT;
wire a,b,c;
half_sub uut1 (.A(A),.B(B),.SUB(a),.BOUT(b));
half_sub uut2 (.A(a),.B(BIN),.SUB(SUB),.BOUT(c));
or_gate uut3 (.X(c),.Y(b),.Z(BOUT));
endmodule
```

Task 13: - JK-Flip Flop



same as for the SR Latch	Clock	Input		Output		Description
	Clk	J	K	Q	Q	
	X	0	0	1	0	Memory no change
	X	0	0	0	1	
	⁻↓_	0	1	1	0	Reset Q » 0
	X	0	1	0	1	
	⁻↓_	1	0	0	1	Set Q » 1
	X	1	0	1	0	
toggle action	⁻↓_	1	1	0	1	Toggle
	⁻↓_	1	1	1	0	

Verilog Code for Implementation of JK Flipflop---

This code utilizes clockdivider module mentioned to implement JK Flipflop. The clock divider module converts 50MHz clock to 1hz for led to respond to toggling action. (Here no gates are involved in design but pure logic/behaviour is considered in code implementation)

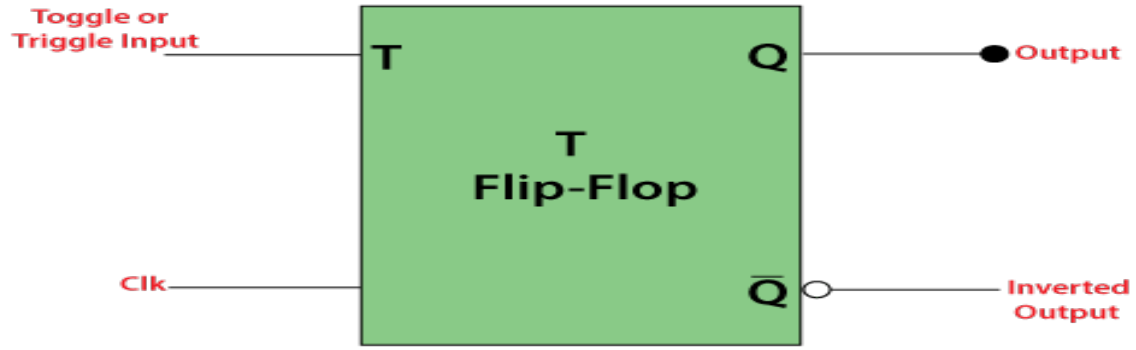
```
module clockdivider(clk_in, clk_out);
input clk_in;
output clk_out;
reg [31:0] counter;
reg out;
initial begin
counter <= 32'b0;
out <= 1'b0;
```

```
end
always @ (posedge clk_in)
begin
counter <= counter + 1'b1;
if (counter > 50000000)
begin
out <= !out;
counter <= 32'b0;
end
end
assign clk_out = out;
endmodule
```

```
/JK Flipflop using Behavioural modelling
module JKFlipflop(j,k,clk,q,qbar);
  input j,k,clk;
  output q,qbar;
  reg q,qbar;
clockdivider u1(clk, clk_out);
always@(posedge clk_out)begin
     if(j == 0 && k == 0)begin
          q <= q;
          qbar<=qbar;
          end
     else if(j == 0 && k == 1)begin
          q <= 0;
          qbar<=1;
          end
     else if(j == 1 && k == 0)begin
          q <= 1;
          qbar<=0;
          end
```

```
    else if(j == 1 && k == 1)begin
        q <= !q;
        qbar<=!qbar;
        end
end
endmodule
```

Task 14: - T-Flipflop



	Previous		Next	
T	Q	Q'	Q	Q'
0	0	1	0	1
0	1	0	1	0
1	0	1	1	0
1	1	0	0	1

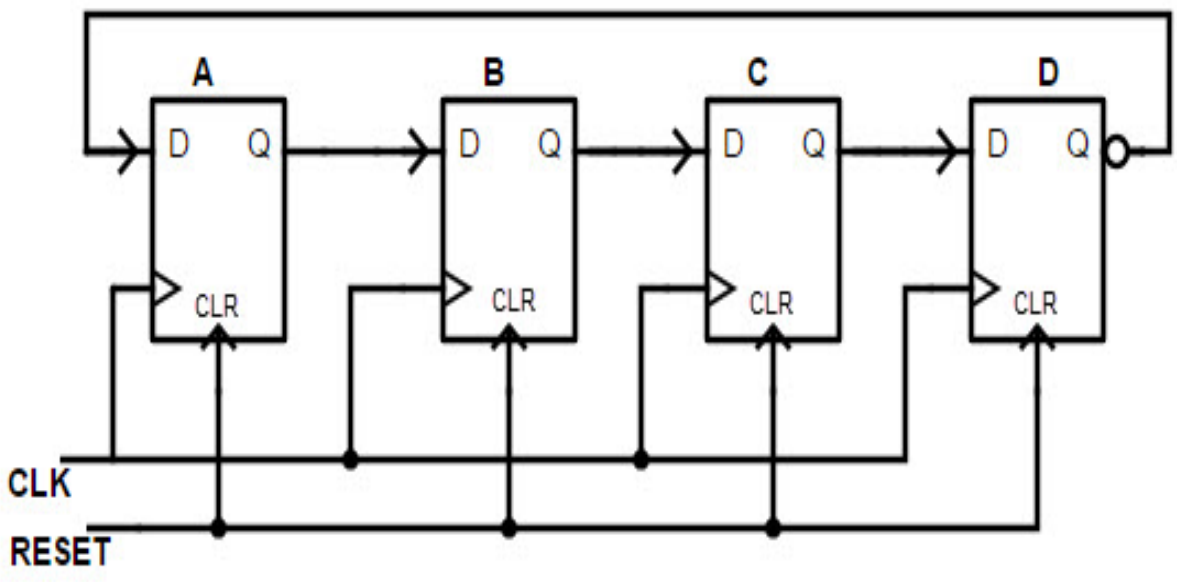
Verilog Code for Implementation of T-Flipflop ---

This code utilizes clockdivider module mentioned above to implement T-Flipflop (Here no gates are involved in design but pure logic/behaviour is considered in code implementation)

```
//T flipflop implementation
module t_flipflop(T,clk,q,qbar);
input T,clk;
output reg q,qbar;

clockdivider uut1(clk, clk_out);
```

```
initial begin
q = 1'b0;
qbar = 1'b1;
end
 always@(negedge clk_out)
   begin
    if (T == 1'b1)
        begin
         q = !q;
        end
    else
        begin
           q = q;
        end
   qbar = !q;
end
endmodule
```

Task 15: - Ring Counter

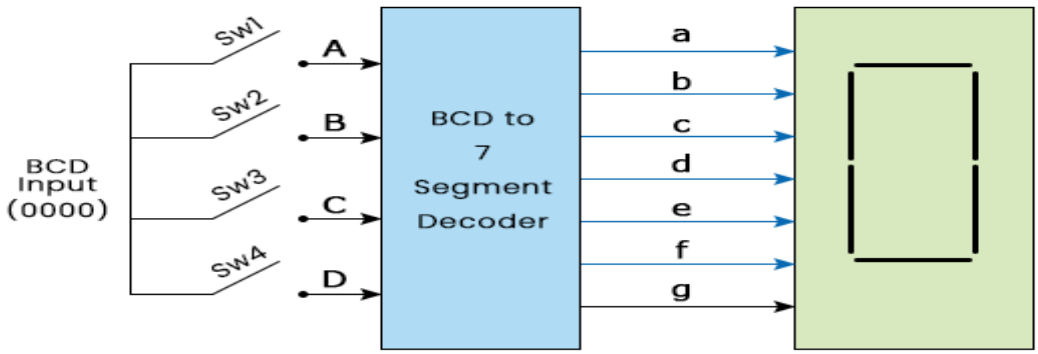


Verilog Code for Implementation of ring counter---

This code utilizes clockdivider module mentioned above to implement ring counter. (Here no gates are involved in design but pure logic/behaviour is considered in code implementation).

```
module ringcounter(q,clk,clr);
 input clk,clr;
 output  reg [3:0]q;
 clockdivider u1(clk, clk_out);
 always @(posedge clk_out)
 begin
    if(clr==1)
        q<=4'b1000;
    else
    begin
        q[3]<=q[0];
        q[2]<=q[3];
        q[1]<=q[2];
        q[0]<=q[1];
        end
end
endmodule
```

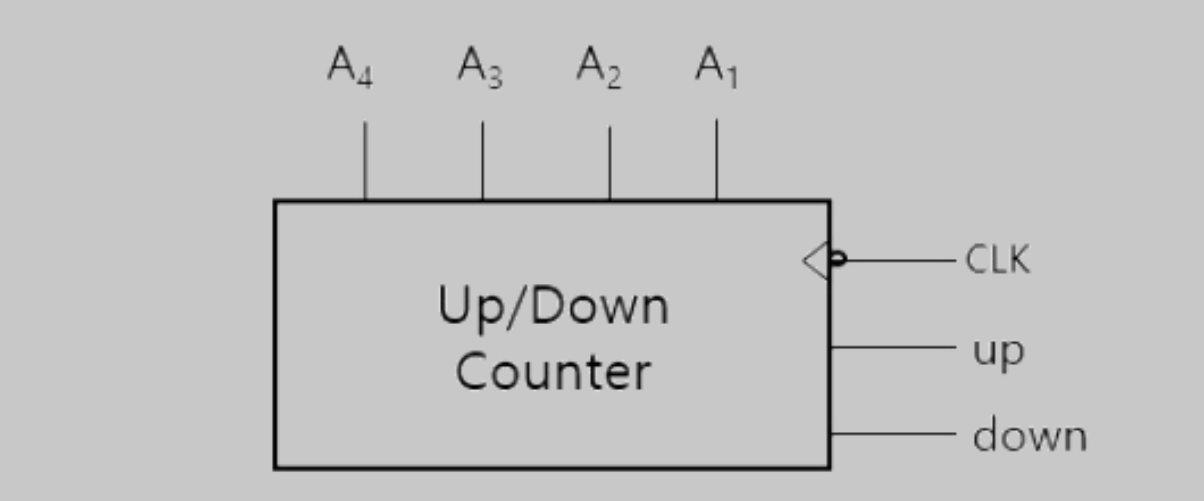
Task16: - BCD to Seven Segment Display Decoder



Verilog Code for Implementation of BCD to Seven Segment Display Decode---

```
// BCD to Seven Segment Display Decoder
module bcd_to_7seg(seg,bcd);
    input [3:0] bcd;
    output [6:0] seg;
    reg [6:0] seg;
always @(bcd)
 begin
case (bcd) //case statement
   0 : seg = 7'b0000001;
   1 : seg = 7'b1001111;
   2 : seg = 7'b0010010;
   3 : seg = 7'b0000110;
   4 : seg = 7'b1001100;
   5 : seg = 7'b0100100;
   6 : seg = 7'b0100000;
   7 : seg = 7'b0001111;
   8 : seg = 7'b0000000;
   9 : seg = 7'b0000100;
   10: seg = 7'b0001000;//A
   11: seg = 7'b1100000;//b
   12: seg = 7'b0110001;//C
   13: seg = 7'b1000010;//d
   14: seg = 7'b0110000;//E
   15: seg = 7'b0111000;//F
endcase
end
endmodule
```

Task17: - Up-Down Counter



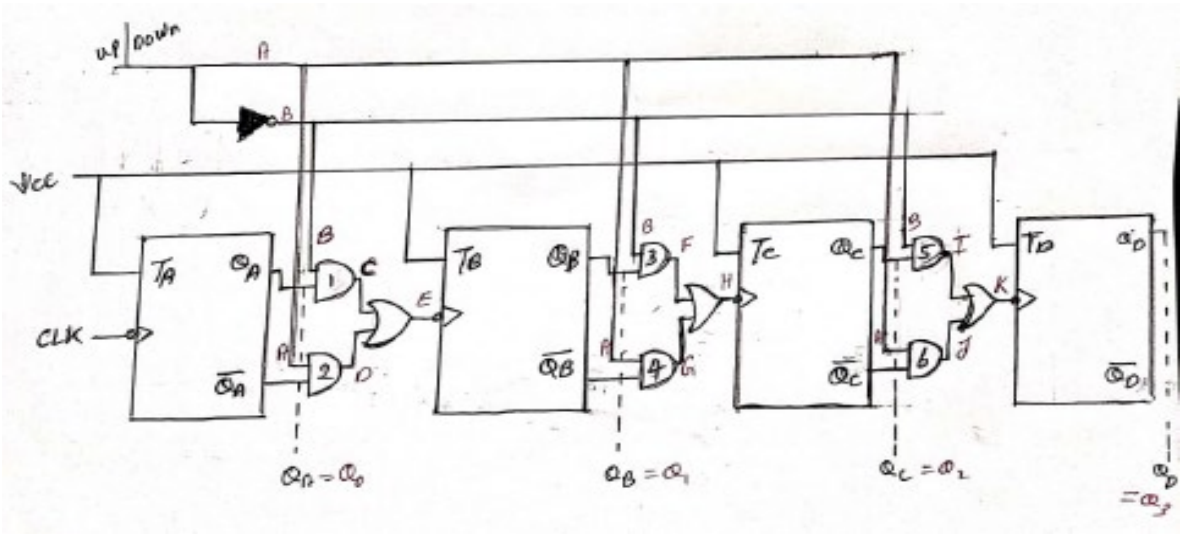
Verilog Code for Implementation of Up-Down Counter ---

This code utilizes clockdivider module mentioned above to implement Up-Down Counter (Here no gates are involved in design but pure logic/behaviour is considered in code implementation)

```
//Behavioural modelling of updown counter
module  updowncounter (clk,updown,out,seg);
  input clk,updown;
  output reg [3:0] out;
  output [6:0] seg;

clockdivider u1(clk, clk_out);

always @(posedge clk_out)
    begin
        if(updown)
         if(out == 4'b1111)
               out <= 4'b0000;
            else
               out <= out + 4'b1;
         else
          if(out == 4'b0000)
               out <= 4'b1111;
```

```
            else
                out <= out - 4'b1;
    end
 bcd_to_7seg uut1(seg,out);
endmodule
```

Task18: - Ripple updown counter using T-Flipflop (Asynchronous)



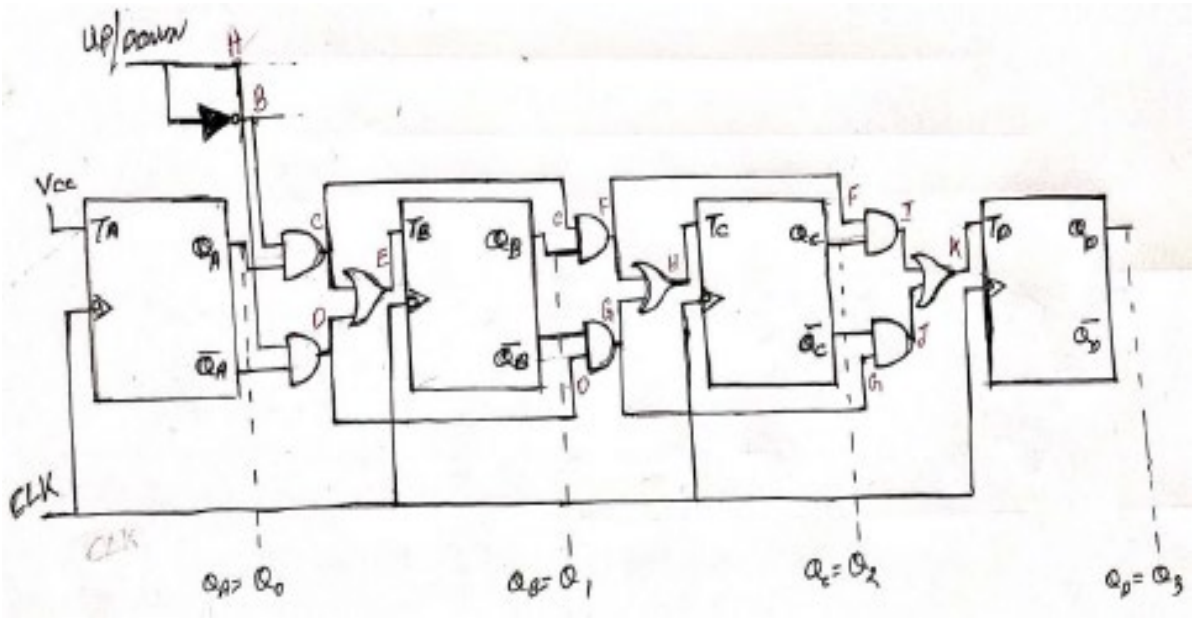
Verilog Code for Implementation of Ripple updown counter using T-Flipflop ---

This code utilizes clockdivider module, bcd_to_7seg module and T-Flipflop module mentioned above to implement (Here a 4-bit up down ripple counter using T-flip-lops is considered in code implementation)

```
//Ripple updown counter using T-Flipflop (Asynchronous)
module ripple_updown_counter (input clk,input updown,output [3:0] out,output [6:0]
seg);
wire q0,qn0,q1,qn1,q2,qn2,q3,qn3;
wire A,B,C,D,E,F,G,H,I,J,K;

assign A = updown;
assign B = ~updown;

clockdivider u1 (clk, clk_out);
```

```
 t_flipflop uut1 (.T(1'b1),.clk(clk_out),.q(q0),.qbar(qn0));
      and(C,B,q0);
      and(D,A,qn0);
      or(E,C,D);
 t_flipflop uut2(.T(1'b1),.clk(E),.q(q1),.qbar(qn1));
      and(F,B,q1);
      and(G,A,qn1);
      or(H,F,G);
  t_flipflop uut3 (.T(1'b1),.clk(H),.q(q2),.qbar(qn2));
      and(I,B,q2);
      and(J,A,qn2);
      or(K,I,J);
  t_flipflop uut4 (.T(1'b1),.clk(K),.q(q3),.qbar(qn3));
assign out = {q3,q2,q1,q0};

bcd_to_7seg uut5(seg,out);
endmodule
```

Task19: - Synchronous updown counter using T-Flipflop

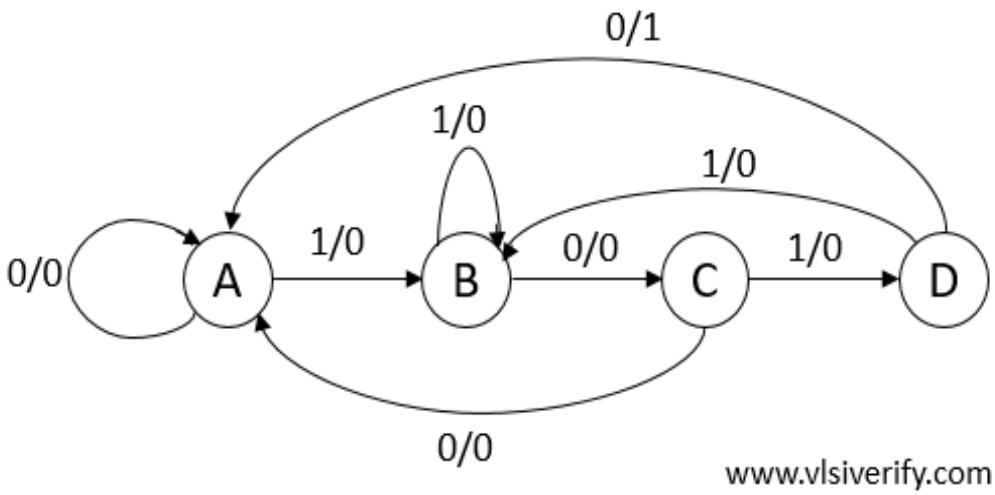


Verilog Code for Implementation of Synchronous updown counter using T-Flipflop ---

This code utilizes clockdivider module and T-Flipflop module mentioned above to implement (Here a 4-bit up down Synchronous counter using T-flip-lops is considered in code implementation)

```
//Synchronous updown counter using T-Flipflop
module sync_updown_counter (input clk, input updown, output [3:0] out,output [6:0]
seg);
wire q0,qn0,q1,qn1,q2,qn2,q3,qn3;
wire A,B,C,D,E,F,G,H,I,J,K;

assign A = updown;
assign B = ~updown;

clockdivider u1 (clk, clk_out);

t_flipflop  u2 (.T(1'b1),.clk(clk_out),.q(q0),.qbar(qn0));
and(C,B,q0);
and(D,A,qn0);
or(E,C,D);
t_flipflop  u3 (.T(E),.clk(clk_out),.q(q1),.qbar(qn1));
and(F,C,q1);
and(G,D,qn1);
or(H,F,G);
t_flipflop u4 (.T(H),.clk(clk_out),.q(q2),.qbar(qn2));
and(I,F,q2);
and(J,G,qn2);
or(K,I,J);
t_flipflop  u5 (.T(K),.clk(clk_out),.q(q3),.qbar(qn3));
 assign out = {q3,q2,q1,q0};

bcd_to_7seg u6 (seg,out);
endmodule
```

Task20: - 1010 non-Overlapping Mealy Sequence Detector



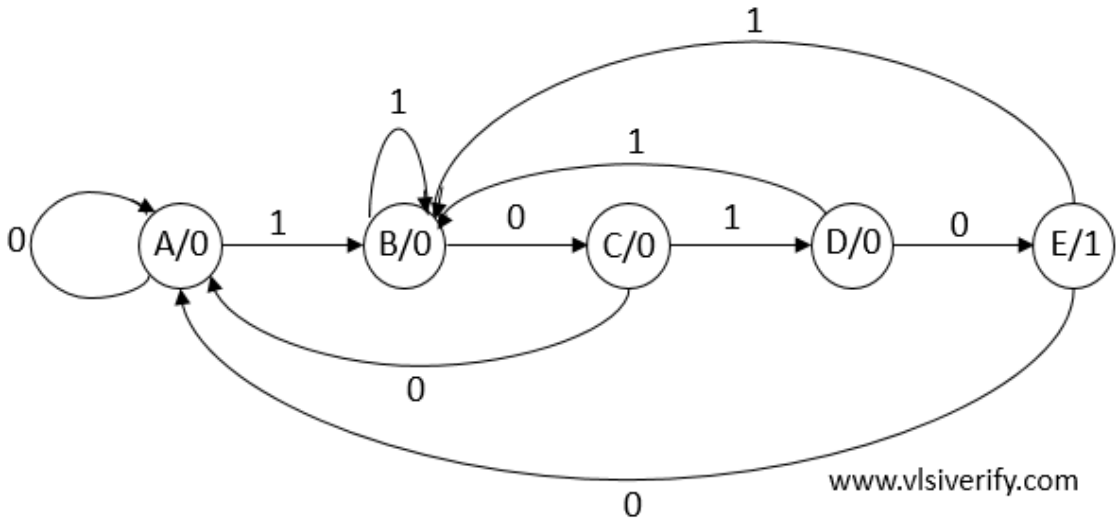
1010 Non-Overlapping Mealy Sequence Detector

Verilog Code for Implementation of 1010 non-Overlapping Mealy Sequence Detector

```
//1010 non-Overlapping Mealy Sequence Detector
module fsm_mealy(clk, reset,x,y,z);
input clk, reset, x;
output reg[1:0] y;
output reg z;
reg [1:0] present_state,next_state;
parameter s0 = 2'b00;
parameter s1 = 2'b01;
parameter s2 = 2'b10;
parameter s3 = 2'b11;

clockdivider uut1 (clk, clk_out);

always @(posedge clk_out or posedge reset)
begin
   if (reset) present_state = s0;
   else  present_state = next_state;
end
```

```
always @( present_state or x)
 begin
      case (present_state)
        s0: begin
           if (x==1'b0) next_state = s0;
           else         next_state = s1;
           end
        s1: begin
           if (x==1'b0) next_state = s2;
           else         next_state = s1;
           end
        s2: begin
           if (x==1'b0) next_state = s0;
           else         next_state = s3;
           end
        s3: begin
           if (x==1'b0) next_state = s0;
           else         next_state = s1;
           end
        default: next_state = s0;
      endcase
end
always @(present_state)
begin
    y = next_state;
    z = (present_state == s3) && (x == 0)? 1:0;
end
endmodule
```

Task21: - 1010 non-Overlapping Moore Sequence Detector



1010 Non-Overlapping Moore Sequence Detector

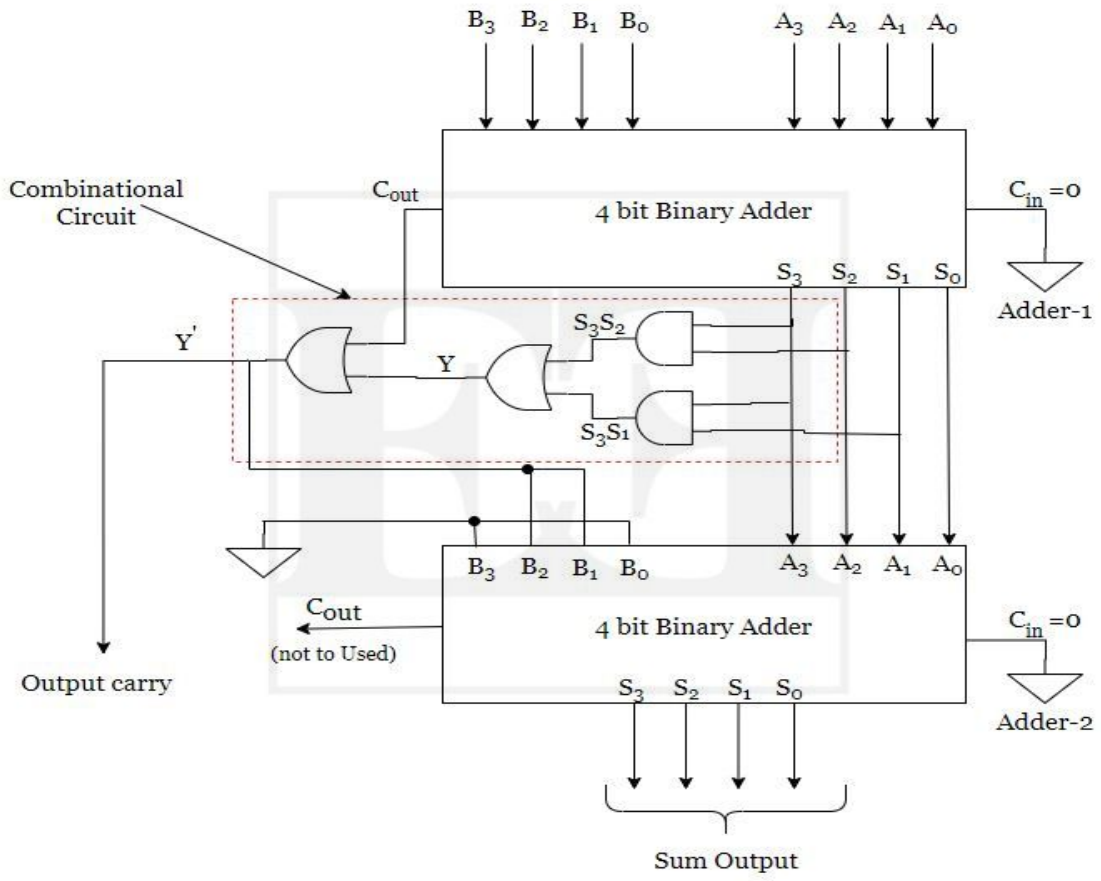
Verilog Code for Implementation of 1010 non-Overlapping Moore Sequence Detector

```
//1010 non-Overlapping Mealy Sequence Detector

module fsm_moore(clk, reset,x,y,z);
input clk, reset, x;
output reg[2:0] y;
output reg z;
reg [2:0] present_state,next_state;

parameter s0 = 3'b001;
parameter s1 = 3'b010;
parameter s2 = 3'b011;
parameter s3 = 3'b110;
parameter s4 = 3'b111;

clockdivider uut1 (clk, clk_out);

always @(posedge clk_out or posedge reset)
begin
```

```
   if (reset) present_state = s0;
   else  present_state = next_state;
end
always @( present_state or x)
 begin
      case (present_state)
        s0: begin
              if (x==1'b0)
              next_state = s0;
           else
              next_state = s1;
              end
        s1: begin
              if (x==1'b0)
              next_state = s2;
           else
              next_state = s1;
              end
        s2: begin
              if (x==1'b0)
              next_state = s0;
           else
              next_state = s3;
              end
        s3: begin
              if (x==1'b0)
              next_state = s4;
           else
              next_state = s1;
              end
          s4: begin
              if (x==1'b0)
```

```
                next_state = s0;
            else
                next_state = s1;
                end
           default: next_state = s0;
       endcase
end
always @(present_state)
begin
    y = next_state;
    z = (present_state == s4)? 1:0;
end
endmodule
```

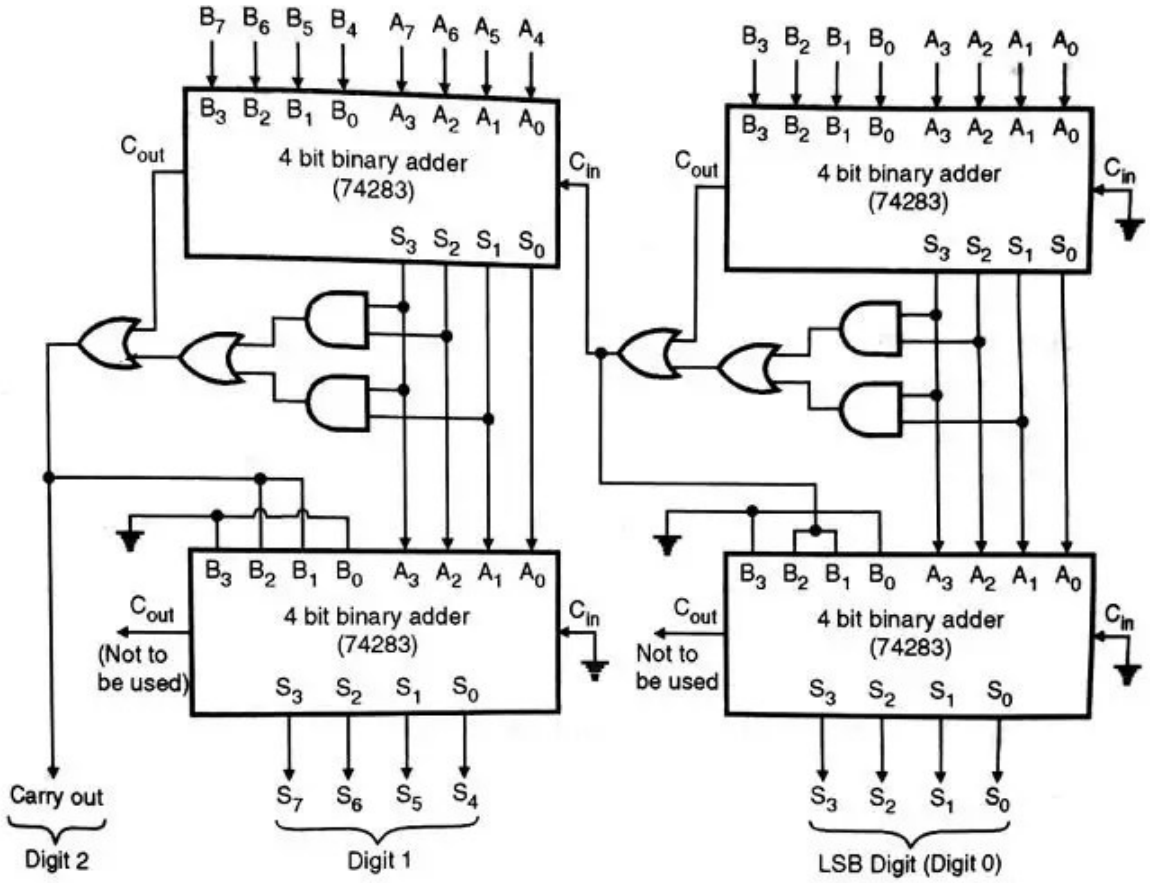
Task 22: - 4-bit BCD adder using 4-bit binary adder



Verilog Code for Implementation of 4-bit BCD adder using 4-bit binary adder

```
//4-bit BCD adder using 4-bit binary adder
module FB_BCD_U_FBFA (B,A,CIN,SUM,COUT_Main,SEG0,SEG1);
input [3:0]B;
input [3:0]A;
input CIN;
output [3:0]SUM;
output COUT_Main;
output [6:0]SEG0;
output [6:0]SEG1;
wire [3:0]COUT;
wire [3:0]temp1;
wire [3:0]temp2;
wire [3:0]temp3;
wire a,b,c,d;
//First adder --- Four bit Ripple Carry full adder
RC_FA uut1(.B(B),.A(A),.CIN(CIN),.SUM(temp1),.COUT(temp2));
//Combinational Logic
and_gate uut2 (.X(temp1[3]),.Y(temp1[2]),.Z(a));
and_gate uut3 (.X(temp1[3]),.Y(temp1[1]),.Z(b));
or_gate uut4 (.X(a),.Y(b),.Z(c));
or_gate uut5 (.X(temp2[3]),.Y(c),.Z(d));
assign temp3 ={1'b0,d,d,1'b0};
//Second adder --- Four bit Ripple Carry full adder
RC_FA uut6(.B(temp3),.A(temp1),.CIN(CIN),.SUM(SUM),.COUT(COUT));
assign COUT_Main = d;
//7-Segment display output
bcd_to_7seg uut7 (SEG0,SUM);
bcd_to_7seg uut8 (SEG1,COUT_Main);
endmodule
```

Task 23: - 8-bit BCD adder using two 4-bit BCD adder



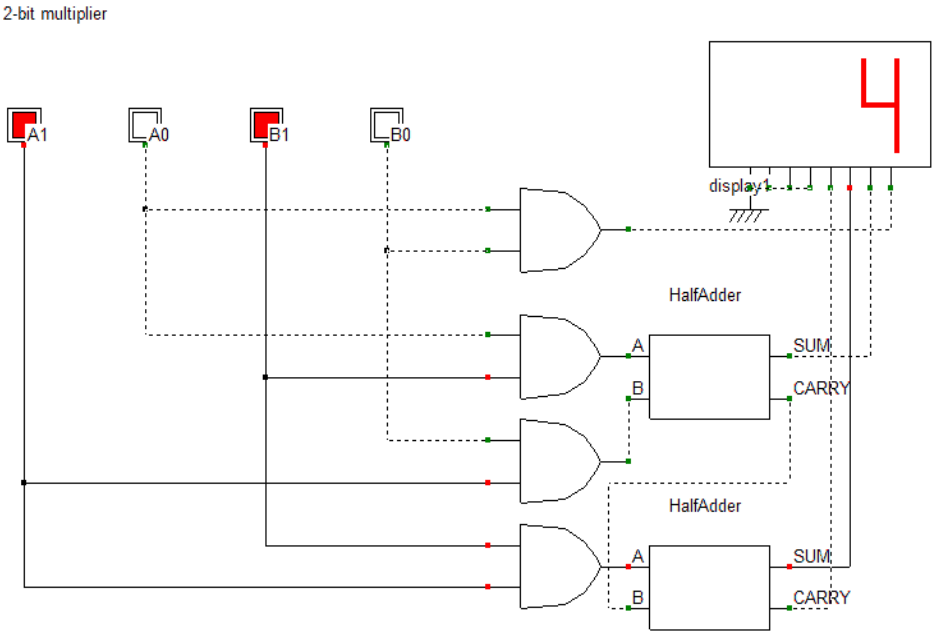
8-bit BCD adder using 74283

Verilog Code for Implementation of 8-bit BCD adder using two 4-bit BCD adder

```
//Eight bit BCD Adder using Two Four bit BCD Adder
module EB_BCD_U_FB_BCD (B,A,CIN,SUM,COUT_Main,SEG0,SEG1);
input [7:0]B;
input [7:0]A;
input CIN;
output [7:0]SUM;
output COUT_Main;
output [13:0]SEG0;
output [6:0]SEG1;
wire temp1;
wire temp2;
```

```
//Four bit BCD using Four bit Full Adder 1
FB_BCD_U_FBFA uut1
(.B(B[3:0]),.A(A[3:0]),.CIN(CIN),.SUM(SUM[3:0]),.COUT_Main(temp1),.SEG0(SEG0[6:0])
);
//Four bit BCD using Four bit Full Adder 2
FB_BCD_U_FBFA uut2
(.B(B[7:4]),.A(A[7:4]),.CIN(temp1),.SUM(SUM[7:4]),.COUT_Main(temp2),.SEG0(SEG0[1
3:7]));
assign COUT_Main = temp2;
//7-Segment display output
bcd_to_7seg uut3 (SEG1,COUT_Main);
endmodule
```

Task 24: - 2-bit Multiplier

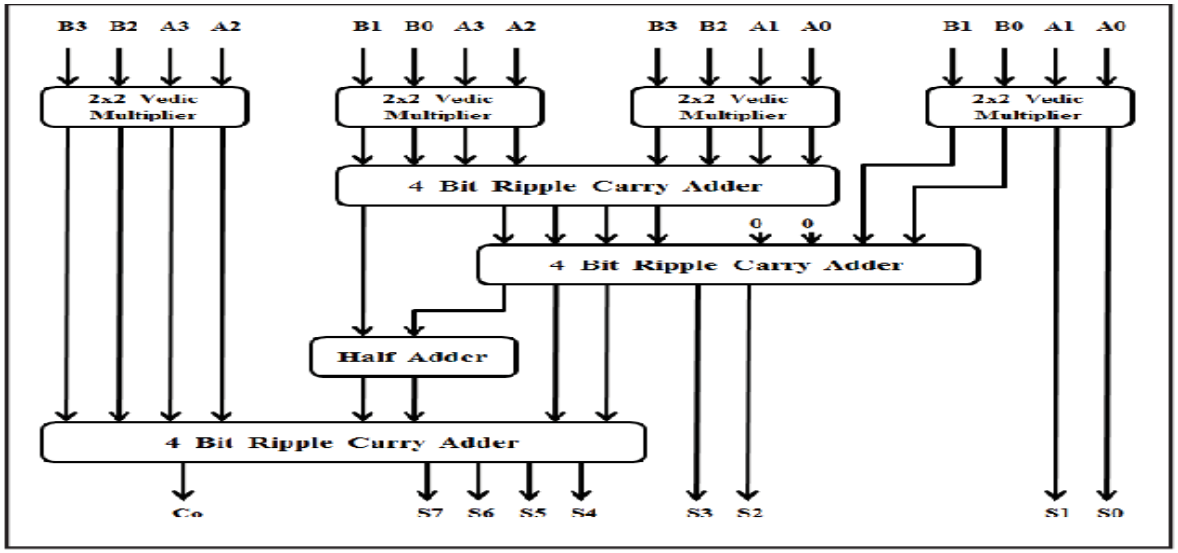


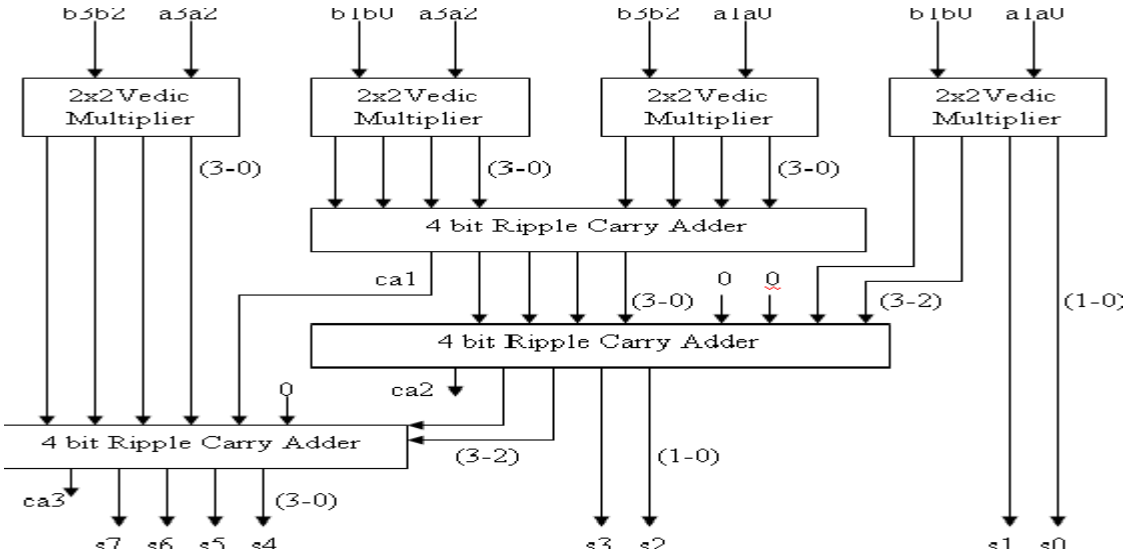
Verilog Code for Implementation of 2-bit Multiplier

```
//2-bit multiplier
module Two_bit_Multiplier (A,B,C,SEG);
input [1:0]A;
```

```
input [1:0]B;
output [3:0]C;
output [6:0]SEG;
wire a,b,c,d,e;

and_gate uut1 (.X(A[0]),.Y(B[0]),.Z(C[0]));
and_gate uut2 (.X(A[0]),.Y(B[1]),.Z(a));
and_gate uut3 (.X(A[1]),.Y(B[0]),.Z(b));
and_gate uut4 (.X(A[1]),.Y(B[1]),.Z(c));

half_adder uut5 (.X(a), .Y(b), .SUM(C[1]), .COUT(d));
half_adder uut6 (.X(c), .Y(d), .SUM(C[2]), .COUT(C[3]));

bcd_to_7seg(SEG,C);
endmodule
```

Task 25: - 4-bit Multiplier using 2-bit Multipliers and 4-bit Ripple Carry Adder.





Verilog Code for Implementation of 4-bit Multiplier using 2-bit Multipliers and 4-bit Ripple Carry Adder.

```
module FB_Multiplier (A,B,SUM,Cout_Main,SEG0,SEG1);

input [3:0]A;
input [3:0]B;
output [7:0]SUM;
output Cout_Main;
output [13:0]SEG0;
output [6:0]SEG1;

wire [3:0]temp1;
wire [3:0]temp2;
wire [3:0]temp3;
wire [3:0]temp4;
wire [3:0]temp5;
wire [3:0]temp6;
wire [3:0]temp7;
wire [3:0]temp8;
wire [3:0]temp9;
wire [3:0]temp10;
wire [3:0]temp11;
wire [3:0]temp12;
wire [11:0]temp13;
wire a,b;

//The 4 2-bit vedic multiplier stage
Two_bit_Multiplier uut1 (.A(A[1:0]),.B(B[1:0]),.C(temp1[3:0]));
assign SUM[0] = temp1[0];
assign SUM[1] = temp1[1];

Two_bit_Multiplier uut2 (.A(A[1:0]),.B(B[3:2]),.C(temp2[3:0]));
```

```
Two_bit_Multiplier uut3 (.A(A[3:2]),.B(B[1:0]),.C(temp3[3:0]));
Two_bit_Multiplier uut4 (.A(A[3:2]),.B(B[3:2]),.C(temp4[3:0]));

//The 3 4-bit ripple carry full adder stage with one halfadder stage
RC_FA uut5
(.B(temp2[3:0]),.A(temp3[3:0]),.CIN(1'b0),.SUM(temp5[3:0]),.COUT(temp6[3:0]));

assign temp7 ={1'b0,1'b0,temp1[3],temp1[2]};
RC_FA uut6
(.B(temp5[3:0]),.A(temp7[3:0]),.CIN(1'b0),.SUM(temp8[3:0]),.COUT(temp9[3:0]));
assign SUM[2] = temp8[0];
assign SUM[3] = temp8[1];

half_adder uut7(.X(temp6[3]),.Y(temp9[3]),.SUM(a),.COUT(b));

assign temp10 ={b,a,temp8[3],temp8[2]};
RC_FA uut8
(.B(temp4[3:0]),.A(temp10[3:0]),.CIN(1'b0),.SUM(temp11[3:0]),.COUT(temp12[3:0]));
assign SUM[4] = temp11[0];
assign SUM[5] = temp11[1];
assign SUM[6] = temp11[2];
assign SUM[7] = temp11[3];
assign Cout_Main = temp12[3];

//The binary to bcd converter
binary_to_bcd uut9(.bin(SUM),.bcd(temp13));

//bcd seven segment display
bcd_to_7seg uut10(.seg(SEG0[6:0]),.bcd(temp13[3:0]));
bcd_to_7seg uut11(.seg(SEG0[13:7]),.bcd(temp13[7:4]));
bcd_to_7seg uut12(.seg(SEG1[6:0]),.bcd(temp13[11:8]));

endmodule
```