4 - bit adders

4 Bit Ripple Carry Adder in Verilog

Structural Model : Half Adder

```
module half_adder(
    output S,C,
  input A,B
    );
 xor(S,A,B);
 and(C,A,B);
endmodule
```

Structural Model : Full Adder

```
module full_adder(
    output S,Cout,
    input A,B,Cin
    );
 wire s1,c1,c2;
 half_adder HA1(s1,c1,A,B);
 half_adder HA2(S,c2,s1,Cin);
 or OG1(Cout,c1,c2);
endmodule
```

Structural Model : 4 Bit Ripple Carry Adder

```
module ripple_adder_4bit(
    output [3:0] Sum,
    output Cout,
    input [3:0] A,B,
    input Cin
    );
 wire c1,c2,c3;
 full_adder FA1(Sum[0],c1,A[0],B[0],Cin),
    FA2(Sum[1],c2,A[1],B[1],c1),
    FA3(Sum[2],c3,A[2],B[2],c2),
    FA4(Sum[3],Cout,A[3],B[3],c3);

endmodule
```

Test Bench : 4 Bit Ripple Carry Adder

```
module test_ripple_adder_4bit;
 // Inputs
 reg [3:0] A;
 reg [3:0] B;
 reg Cin;
 // Outputs
 wire [3:0] Sum;
 wire Cout;
 // Instantiate the Unit Under Test (UUT)
 ripple_adder_4bit uut (
  .Sum(Sum),
  .Cout(Cout),
  .A(A),
  .B(B),
  .Cin(Cin)
 );
 initial begin
  // Initialize Inputs
  A = 0;
  B = 0;
  Cin = 0;
  // Wait 100 ns for global reset to finish
  #100;
  // Add stimulus here
  A=4'b0001;B=4'b0000;Cin=1'b0;
  #10 A=4'b1010;B=4'b0011;Cin=1'b0;
  #10 A=4'b1101;B=4'b1010;Cin=1'b1;
 end
 initial begin
  $monitor("time=",$time,, "A=%b B=%b Cin=%b : Sum=%b Cout=
%b",A,B,Cin,Sum,Cout);
 end
```

```
Endmodule
```

---

4 Bit Carry Look Ahead Adder in Verilog

DataFlow Model : 4 Bit CLA

```
module CLA_4bit(
    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; //Generate
    assign P = A ^ B; //Propagate
    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
```

Test Bench : 4 Bit CLA

```
module Test_CLA_4bit;
    // Inputs
    reg [3:0] A;
    reg [3:0] B;
    reg Cin;

    // Outputs
    wire [3:0] S;
    wire Cout;
    wire PG;
    wire GG;

    // Instantiate the Unit Under Test (UUT)
    CLA_4bit uut (
    .S(S),
    .Cout(Cout),
    .PG(PG),
    .GG(GG),
    .A(A),

    .B(B),
    .Cin(Cin)
    );

    initial begin
    // Initialize Inputs
    A = 0;  B = 0;  Cin = 0;
    // Wait 100 ns for global reset to finish
    #100;

    // Add stimulus here
    A=4'b0001;B=4'b0000;Cin=1'b0;
    #10 A=4'b100;B=4'b0011;Cin=1'b0;
    #10 A=4'b1101;B=4'b1010;Cin=1'b1;
    #10 A=4'b1110;B=4'b1001;Cin=1'b0;
    #10 A=4'b1111;B=4'b1010;Cin=1'b0;
    end

    initial begin
 $monitor("time=",$time,, "A=%b B=%b Cin=%b : Sum=%b Cout=%b PG=%b
GG=%b",A,B,Cin,S,Cout,PG,GG);
    end
endmodule
```

Simulation Results

```
time = 0 A=0000 B=0000 Cin=0 : Sum=0000 Cout=0 PG=0 GG=0
time = 100 A=0001 B=0000 Cin=0 : Sum=0001 Cout=0 PG=0 GG=0
time = 110 A=0100 B=0011 Cin=0 : Sum=0111 Cout=0 PG=0 GG=0
time = 120 A=1101 B=1010 Cin=1 : Sum=1000 Cout=1 PG=0 GG=1
time = 130 A=1110 B=1001 Cin=0 : Sum=0111 Cout=1 PG=0 GG=1
time = 140 A=1111 B=1010 Cin=0 : Sum=1001 Cout=1 PG=0 GG=1
```

Verilog code for Carry Save Adder:

```
module carrysave(p0,p1,p2,p3,p4,p5,s,c,a,b);
output [5:0]p0,p1,p2,p3,p4,p5;
output [10:0]s;
output [7:0]c;
input  [5:0]a,b;
wire
d,d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12,d13,d14,d15,d16,d17,e1,e2,e3,e4,e5,e6,e7,e8,e
9,e10,e11,e13,e14,e15,e16,e17;

assign p0=b[0]?a:0;
assign p1=b[1]?a:0;
assign p2=b[2]?a:0;
assign p3=b[3]?a:0;
assign p4=b[4]?a:0;
assign p5=b[5]?a:0;
assign s[0]=p0[0];

HA h1(s[1],d,p0[1],p1[0]);
HA h2(e5,d5,p1[5],p2[4]);
FA m1(e1,d1,p0[2],p1[1],p2[0]);
FA m2(e2,d2,p0[3],p1[2],p2[1]);
FA m3(e3,d3,p0[4],p1[3],p2[2]);
FA m4(e4,d4,p0[5],p1[4],p2[3]);

HA h3(e6,d6,p3[1],p4[0]);
HA h4(e11,d11,p4[5],p5[4]);
```

```
FA m5(e7,d7,p3[2],p4[1],p5[0]);
FA m6(e8,d8,p3[3],p4[2],p5[1]);
FA m7(e9,d9,p3[4],p4[3],p5[2]);
FA m8(e10,d10,p3[5],p4[4],p5[3]);

HA h5(s[2],d12,d,e1);
FA m9(e13,d13,d1,e2,p3[0]);
FA m10(e14,d14,d2,e3,e6);
FA m11(e15,d15,d3,e4,e7);
FA m12(e16,d16,d4,e5,e8);
FA m13(e17,d17,d5,e6,p2[5]);


HA h6(s[3],c[0],d12,e13);
HA h7(s[4],c[1],d13,e14);
HA h8(s[9],c[6],d10,e11);
HA h9(s[10],c[7],d11,p5[5]);
FA m14(s[5],c[2],d6,d14,e15);
FA m15(s[6],c[3],d7,d15,e16);
FA m16(s[7],c[4],d8,d16,e17);
FA m17(s[8],c[5],d9,d17,e10);
endmodule
```

Address Decoders

a) 2 to 4 decoder
b) 3 to 8 decoder
c) priority encoder

---

2 : 4 DECODER

VERILOG CODE:

```
module Decoder(A, B, D);

   input A, B;
   output [3:0] D;
   reg [3:0] D;

   always @ (A or B)
   begin
```

```
      if( A == 0 && B == 0 )
         D <= 4'b0001;

      else if ( A == 0 && B == 1 )
         D <= 4'b0010;

      else if ( A == 1 && B == 0 )
         D <= 4'b0100;

      else
         D <= 4'b1000;
   end

endmodule
```

<u>TEST BENCH:</u>

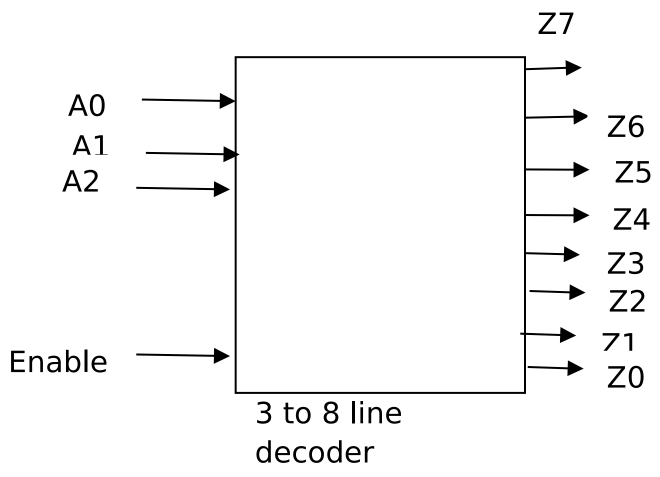
```
module Testbench;

   reg A_t, B_t;
   wire [3:0] D_t;

   Decoder Decoder_1(A_t, B_t, D_t);

   initial
   begin

      //case 0
      A_t <= 0; B_t <= 0;
      #1 $display("D_t = %b", D_t);

      //case 1
      A_t <= 0; B_t <= 1;
      #1 $display("D_t = %b", D_t);

      //case 2
      A_t <= 1; B_t <= 0;
      #1 $display("D_t = %b", D_t);

      //case 3
      A_t <= 1; B_t <= 1;
      #1 $display("D_t = %b", D_t);

   end
endmodule
```

3 : 8 DECODER

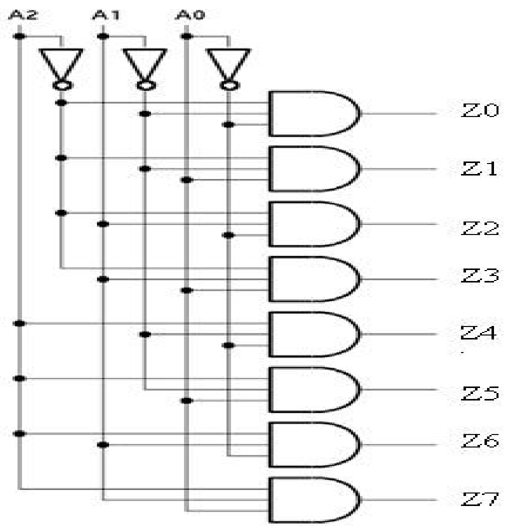
Function Table

	Input			**Output**							
enable	**A1**	**A1**	**A0**	**Z7**	**Z6**	**Z5**	**Z4**	**Z3**	**Z2**	**Z1**	**Z0**
0	**x**	**X**	**x**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**
1	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**
1	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**
1	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**
1	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**
1	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**
1	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**
1	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Block Diagram 3-8 line Decoder



Circuit Diagram 3-8 line Decoder



Verilog Code for 3 to 8 line decoder

```
Module dec(bin,decout,en);
  input [0:2] bin;
  input en;
  output [7:0] decout;
  reg decout;
  always @(en or bin)
  begin
  decout=0;
  if(en)
```

```
 begin
 case(bin)
 3’b000:decout=8’o001;
 3’b001:decout=8’o002;
 3’b010:decout=8’o004;
 3’b011:decout=8’o010;
 3’b100:decout=8’o020;
 3’b101:decout=8’o040;
 3’b110:decout=8’o100;
 3’b111:decout=8’o200;
 endcase
 end
 end
endmodule
```

(OR)

VERILOG CODE

```
module decodermod(e, a, b, d);

input e;

input a;

input b;

output [7:0] d;
```

```
assign d[0]=(~e)&(~a)&(~b);

assign d[1]=(~e)&(~a)&(b);

assign d[2]=(~e)&(a)&(~b);

assign d[3]=(~e)&(a)&(b);

assign d[4]=(e)&(~a)&(~b);

assign d[5]=(e)&(~a)&(b);

assign d[6]=(e)&(a)&(~b);

assign d[7]=(e)&(a)&(b);

endmodule
```

<u>TEST BENCH</u>

```
module decodert_b;

reg e;

reg a;

reg b;

wire [7:0] d;

decodermod uut ( .e(e),.a(a),.b(b),.d(d) );

initial begin

#10 e=1’b0;a=1’b0;b=1’b0;
```

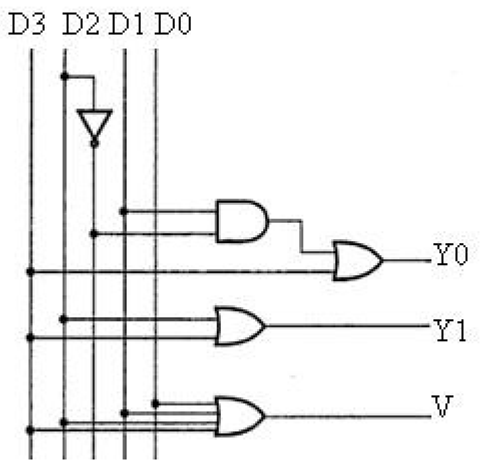
```
#10 e=1′b0;a=1′b0;b=1′b1;

#10 e=1′b0;a=1′b1;b=1′b0;

#10 e=1′b0;a=1′b1;b=1′b1;

#10 e=1′b1;a=1′b0;b=1′b0;

#10 e=1′b1;a=1′b0;b=1′b1;

#10 e=1′b1;a=1′b1;b=1′b0;

#10 e=1′b1;a=1′b1;b=1′b1;

#10$stop;

end

endmodule
```

---

4 Bit Priority Encoder in Verilog

Priority Encoder is an encoder circuit that includes a priority function. The operation is such that if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence.
Here, the priority decreases from right to left in the input. D[3] has the highest priority. V indicate the validity of the input (atleast one input should be 1) and the Y gives the output.(01 means 1, 10 means 2 like that...)

Inputs				Outputs		
D_0	D_1	D_2	D_3	Y_1	Y_0	V
0	0	0	0	×	×	0
1	0	0	0	0	0	1
×	1	0	0	0	1	1
×	×	1	0	1	0	1
×	×	×	1	1	1	1



Behavioural Model : 4 Bit Priority Encoder

```
module PriorityEncoder_4Bit(
    input [0:3] D,
    output [1:0] Y,
    output V
    );
  reg [1:0] Y;
  reg V;
  always @(D)
  begin
  Y[1] <= D[2] | D[3];
  Y[0] <= D[3] | D[1] & ~D[2];
  V = D[0] | D[1] | D[2] | D[3];
 end
endmodule
```

Test Bench : 4 Bit Priority Encoder

```
module PriorityEncoder_4Bit_Test;
 // Inputs
 reg [3:0] D;
 // Outputs
 wire [1:0] Y;
 wire V;
 // Instantiate the Unit Under Test (UUT)
 PriorityEncoder_4Bit uut (
   .D(D),
   .Y(Y),
   .V(V)
 );
 initial begin
   // Initialize Inputs
   D = 0;
   // Wait 100 ns for global reset to finish
   #100;
   // Add stimulus here
   #10 D = 4'b0000;
   #10 D = 4'b1000;
   #10 D = 4'b0100;
   #10 D = 4'b0010;
   #10 D = 4'b0001;
   #10 D = 4'b1010;
   #10 D = 4'b1111;
 end
 initial begin
   $monitor("time=",$time,, "D=%b : Y=%b V=%b",D,Y,V);
 end
endmodule
```

Simulation Results

```
time= 000, D=0000 : Y=00 V=0
time= 120, D=1000 : Y=00 V=1
time= 130, D=0100 : Y=01 V=1
time= 140, D=0010 : Y=10 V=1
time= 150, D=0001 : Y=11 V=1
time= 160, D=1010 : Y=10 V=1
time= 170, D=1111 : Y=11 V=1
```

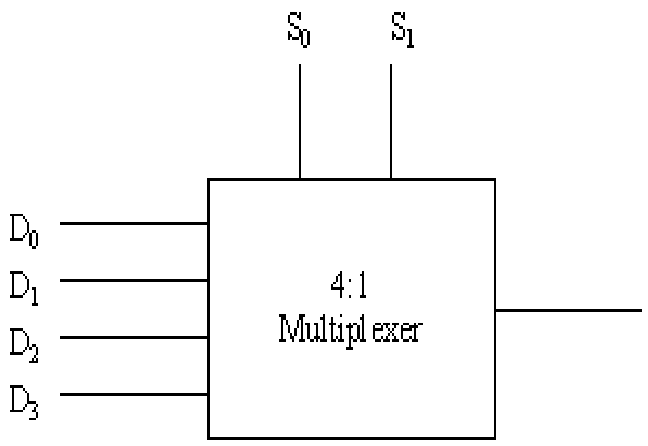
MULTIPLEXERS

4 : 1 MULTIPLEXER

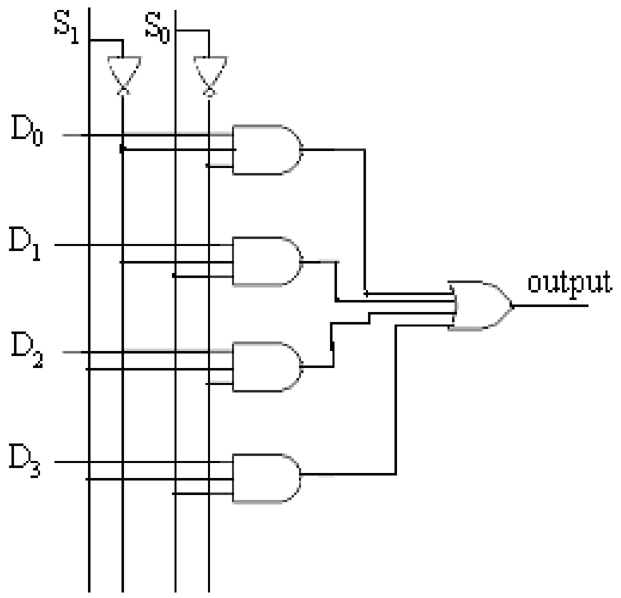
Function Table

Selection Inputs		Output
S1	S0	
0	0	D0
0	1	D1
1	0	D2
1	1	D3

Block Diagram 4:1 Multiplexer



Circuit Diagram 4:1 Multiplexer



Verilog code 4 to 1 for multiplexer

```
module mux4to1(Y, I0,I1,I2,I3, sel);
   output Y;
   input I0,I1,I2,I3;
   input [1:0] sel;
   reg Y;
always @ (sel or I0 or I1 or I2 or I3)
case (sel)
             2'b00:Y=I0;
             2'b01:Y=I1;
             2'b10: Y=I2;
             2'b11: Y=I3;
```

```
        default: Y=2b’00;
endcase
endmodule
```

(OR)

VERILOG CODE:

```
module mux4bit(a, s, o);

input [3:0] a;

input [1:0] s;

output o;

reg o;

always @(a or s)

begin

case (s)

2’b00:o=a[0];

2’b01:o=a[1];

2’b10:o=a[2];

2’b11:o=a[3];

default:o=0;

endcase
```

```
end

endmodule
```

TEST BENCH:

```
module muxt_b;

reg [3:0] a;

reg [1:0] s;

wire o;

mux4bit uut (.a(a),   .s(s),.o(o));

initial begin

#10 a=4’b1010;

#10 s=2’b00;

#10 s=2’b01;

#10 s=2’b10;

#10 s=2’b11;

#10 $stop;

end

endmodule
```

1:4 Demultiplexer

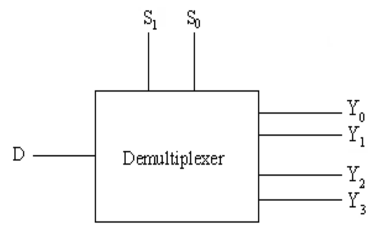
Function table

Data Input	Selection Input		Output			
D	S1	S0	Y3	Y2	Y1	Y0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

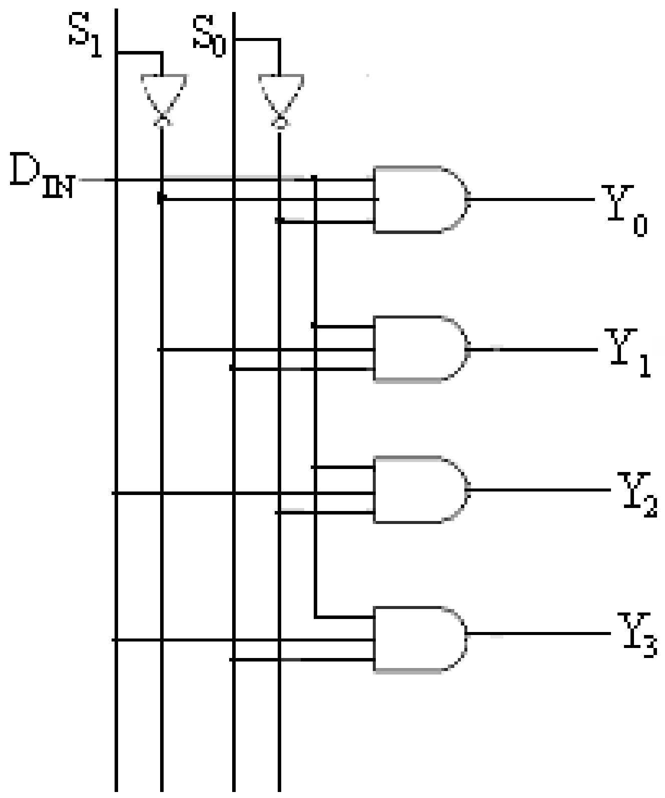
Function Table

Inputs		Output
S_1	S_0	
0	0	Y_0=D
0	1	Y_1=D
1	0	Y_2=D
1	1	Y_3=D

Block Diagram 1:4 Demultiplexer



Circuit Diagram 1:4 Demultiplexer



<u>Verilog code for 1 to 4 demultiplexer</u>

```
module demux(S,D,Y);
         Input [1:0] S;
         Input D;
         Output [3:0] Y;
         reg Y;
        always @(S OR D)
        case({D,S})
       3'b100:Y=4'b0001;
       3'b101:Y=4'b0010;
```

```
        3’b110:Y=4’b0100;

        3’b111:Y=4’b1000;

        default:Y=4’b0000;

        endcase

endmodule
```

TEST BENCH:

```
#10 s=2’b00;

#10 s=2’b01;

#10 s=2’b10;

#10 s=2’b11;

#10 $stop;

end

endmodule
```

COUNTERS

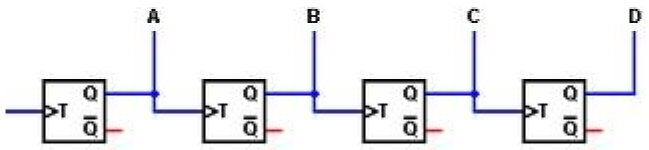
4-Bit Binary Ripple Counter

Function Table

Output(count 0-15)

A	B	C	D
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Circuit Diagram



<u>Verilog Code for Ripple Counter</u>

```
module ripple(clkr,st,,t,A,B,C,D);

input clk,rst,t;

output A,B,C,D;
```

```
Tff T0(D,clk,rst,t);

Tff T1(C,clk,rst,t);

Tff T2(B,clk,rst,t);

Tff T3(A,clk,rst,t);

endmodule

module Tff(q,clk,rst,t);

input clk,rst,t;

output q;

reg q;

always @(posedge clk)

begin

if(rst)

q<=1’b0;

else

if(t)

q<=~q;

end

endmodule
```

4-bit Up-Down Counter

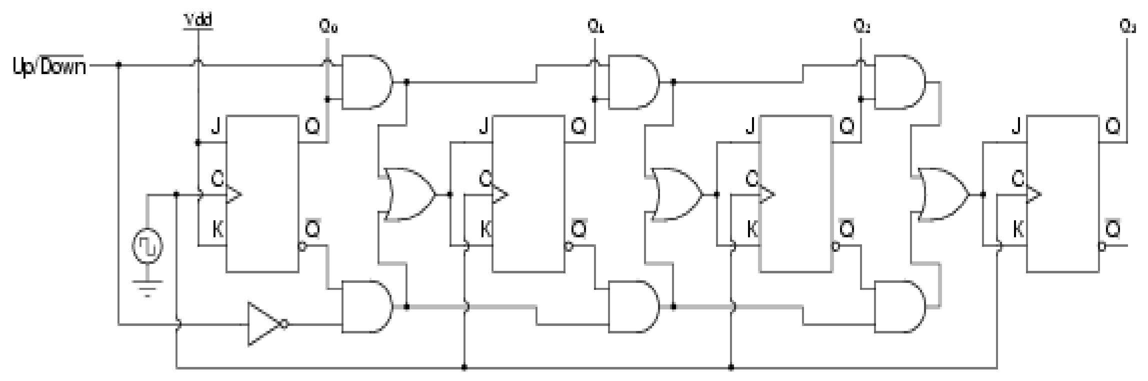
Count Table

Output(count	**Output (count up)**

down)							
Q0	Q1	Q2	Q3	Q0	Q1	Q2	Q3
1	1	1	1	0	0	0	0
1	1	1	0	0	0	0	1
1	1	0	1	0	0	1	0
1	1	0	0	0	0	1	1
1	0	1	1	0	1	0	0
1	0	1	0	0	1	0	1
1	0	0	1	0	1	1	0
1	0	0	0	0	1	1	1
0	1	1	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	0	1	1	0	1	0
0	1	0	0	1	0	1	1
0	0	1	1	1	1	0	0
0	0	1	0	1	1	0	1
0	0	0	1	1	1	1	0
0	0	0	0	1	1	1	1

Circuit diagram

A four-bit synchronous "up/down" counter



Verilog code for up-down counter

```
module updowncount (R, Clock, clr, E, up_down, Q);

parameter n = 4;

input [n-1:0] R;

input Clock, clr, E, up_down;

output [n-1:0] Q;

reg [n-1:0] Q;

integer direction;

always @(posedge Clock)

begin

if (up_down)  direction = 1;

else direction = -1;

if (clr)  Q <= R;

else if (E) Q <= Q + direction;
```

```
end

endmodule
```

(OR)

UP-DOWNCOUNTER VERIOLG CODE

```
module updowncountermod(clk, clear, updown, q);

input clk;

input clear;

input updown;

output [3:0] q;

reg [3:0] q;

always@(posedge clear or posedge clk)

begin

if(clear)

q <=4′b0000;

else if(updown)

q <= q+1′b1;

else

q <= q-1′b1;
```

```
end

endmodule
```

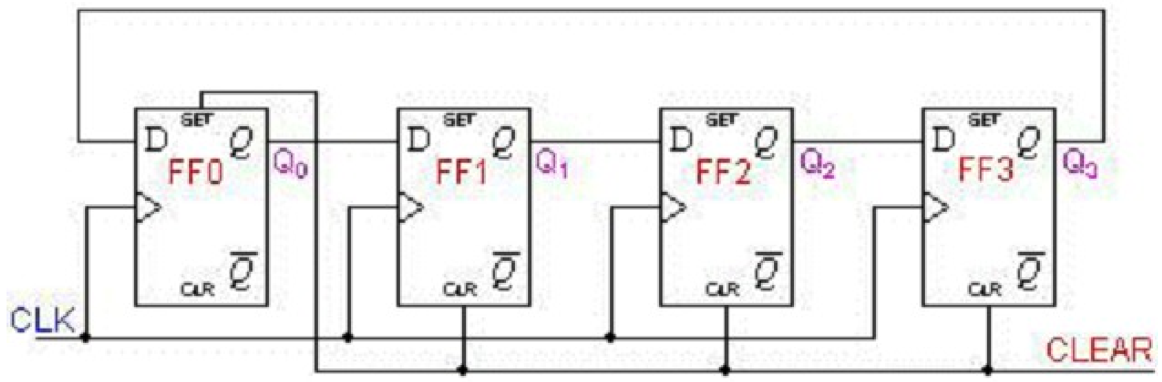
TEST BENCH

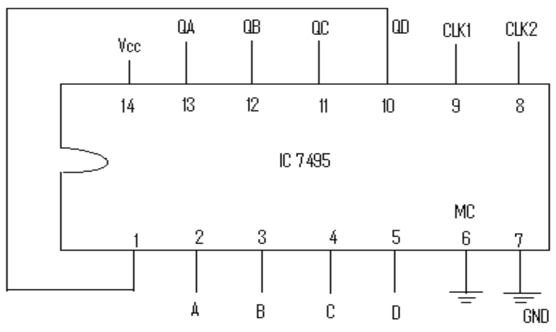
```
module updowncountert_b;

reg clk;

reg clear;

reg updown;

wire [3:0] q;

updowncountermod uut (.clk(clk),.clear(clear), .updown(updown), .q(q)   );

initial begin

clk = 0;

clear = 0;

updown = 0;


#5 clear=1’b1;

#5 clear=1’b0;

#100 updown=1’b1;
```

```
end

always #5 clk=~clk;

initial #150 $stop;

endmodule
```

Ring Counter

Circuit Diagram for ring counter





Count Table

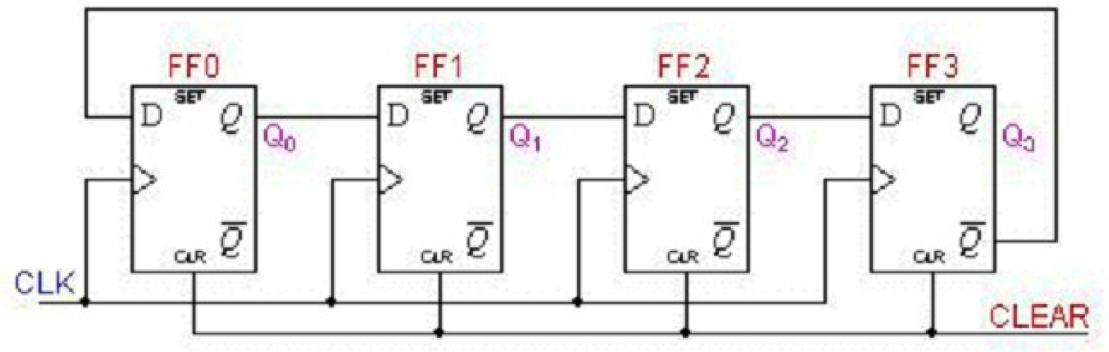
Verilog code for ring counter

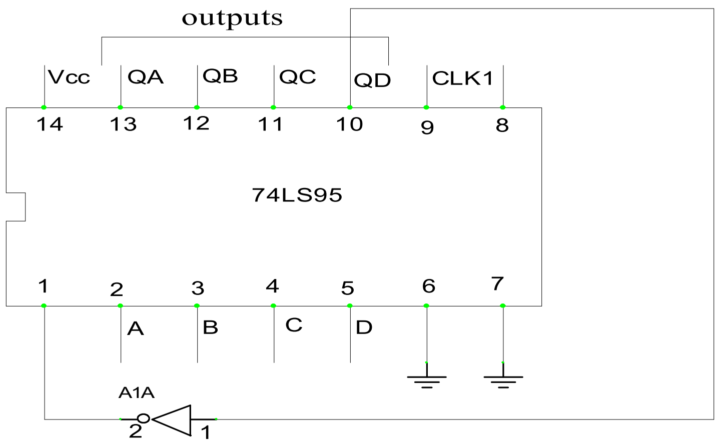
clk	Q_a	Q_b	Q_c	Q_d
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

```
module ring_count (Resetn, Clock, Q);
parameter n = 5;
        input Resetn, Clock;
        output [n-1:0] Q;
        reg [n-1:0] Q;
        always @(posedge Clock)
        if (!Resetn)
        begin
        Q[4:1] <= 0;
        Q[0] <= 1;
        end
        else
        Q <= {{Q[3:0]}, {Q[4]}};
endmodule
```

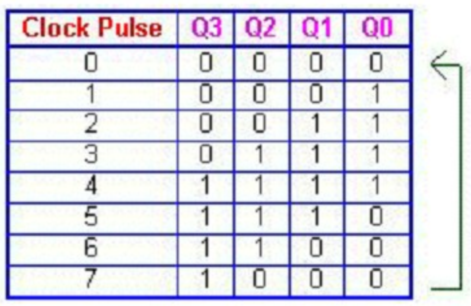
<u>Johnson Counter</u>

Circuit Diagram for Johnson counter





Count Table



Clock Pulse	Q3	Q2	Q1	Q0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	1
3	0	1	1	1
4	1	1	1	1
5	1	1	1	0
6	1	1	0	0
7	1	0	0	0

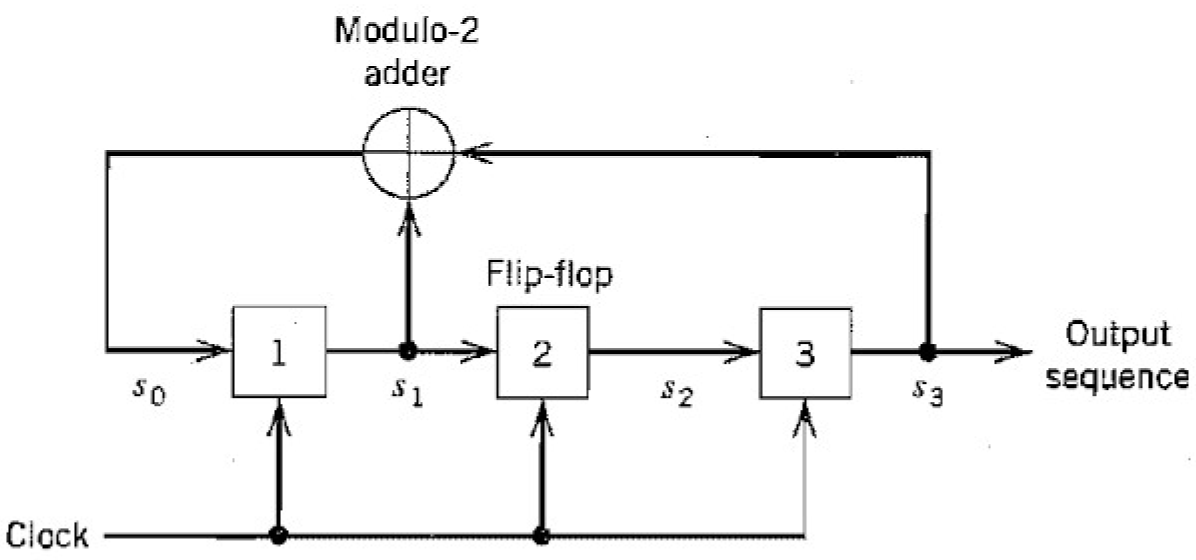
<u>Verilog code</u>

```
module stc(clk,clr,q,r,s,t);

input clk,clr;
output q,r,s,t;

reg q,r,s,t;

always@(negedge clk)

begin

if(~clr)

begin

q=1'b0;

end

else

begin

q<=~t;

r<=q;

s<=r;

t<=s;

end

end
```

endmodule

Pseudo Random Sequence Generator in Verilog

Pseudo Random Sequence is widely used in spread spectrum communication, to spread and de-spread the information sequence. The following diagram shows a PN sequence generator which has 3 memory elements, leads to processing gain 7 (2^3 - 1). According to the diagram, if the initial state is all zero, then the sequence will also be all zeros which is not usable.



Behavioural Model

```
module PNSeqGen(
    input clk, reset,
    output s3
    );
 reg s1, s2, s3;
 wire s0;
 // MODULO 2 ADDITION
 assign s0 = s1 ^ s3;
 // STATE MACHINE
 always @ (posedge clk or reset) begin
  // INITIAL STATE SHOULDN'T BE 000 => 100
  if(reset) begin
   s1 <= 1;
   s2 <= 0;
   s3 <= 0;
  end else begin
   s1 <= s0;
   s2 <= s1;
   s3 <= s2;
  end
 end
endmodule
```

Test Bench

```
module Test_PNSeqGen;
 // Inputs
 reg clk;
 reg reset;
 // Outputs
 wire s3;
 // Instantiate the Unit Under Test (UUT)
 PNSeqGen uut (
  .clk(clk),
  .reset(reset),
  .s3(s3)
 );
 initial begin
  // Initialize Inputs
  clk = 0;
  reset = 0;
  // Wait 100 ns for global reset to finish
  #100;
  // Add stimulus here
  #10 reset = 1;
  #10 reset = 0;
  #200 $finish;
 end
 always begin
  #5 clk = ~clk;
 end
 // PRINT SEQUENCE
 always @ (posedge clk) $write("%b",s3);
endmodule
```

Output

```
001110100111010011101
```

ACCUMULATOR VERILOG CODE

module accumod (in, acc, clk, reset);

input [7:0] in;

input clk, reset;

output [7:0] acc;

reg [7:0] acc;

always@(clk) begin

if(reset)

acc <= 8′b00000000;

else

acc <= acc + in;

end

endmodule

TEST BENCH

```
module accumt_b;

reg [7:0] in;

reg clk;

reg reset;

wire [7:0] acc;

accumod uut ( .in(in), .acc(acc),.clk(clk),.reset(reset) );

initial begin

#5 reset<=1′b1;

#5 reset<=1′b0;

clk =1′b0;

in = 8′b00000001;

#50 in = 8′b00000010;

#50 in = 8′b00000011;

end

always #10 clk = ~clk;

initial#180 $stop;
```

```
endmodule
```

Implementation of 4-bit Magnitude Comparator

Consider two 4-bit binary numbers A and B such that

$A = A_3A_2A_1A_0$

$B = B_3B_2B_1B_0$

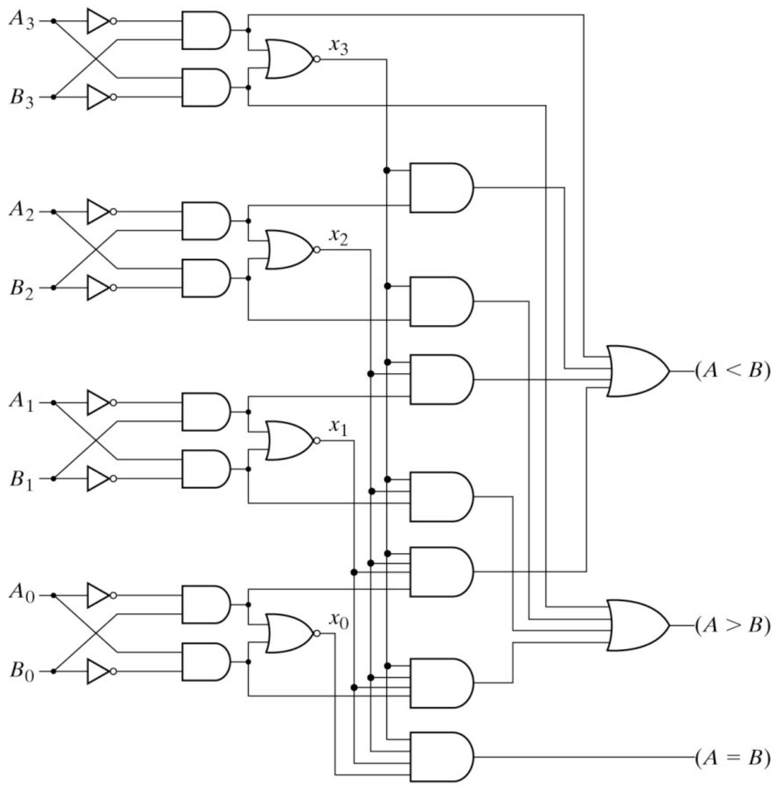
$$x_i = A_i \cdot B_i + \overline{A}_i \cdot \overline{B}_i$$

$$(A = B) = x_3x_2x_1x_0$$

$$(A > B) = A_3 \cdot \overline{B}_3 + x_3A_2\overline{B}_2 + x_3x_2A_1\overline{B}_1 + x_3x_2x_1A_0\overline{B}_0$$

$$(A < B) = \overline{A}_3 \cdot B_3 + x_3\overline{A}_2B_2 + x_3x_2\overline{A}_1B_1 + x_3x_2x_1\overline{A}_0B_0$$

Circuit Diagram



4-Bit Magnitude Comparator

<u>Verilog code(Abstract level)</u>

```
module compare(A,B,y);

input [3:0] A,B;

output [2:0] y;

reg y;

always @(A or B)

if(A==B)

y=3'b001;

else if(A<B)

y=3'b010;
```

```
else

y=3'b100;

endmodule
```

Verilog code for 4-bit magnitude comparator

```
module compare(A,B,x,y,z);

input [3:0] A,B;

output x, y,z;

wire x0,x1,x2,x3;

assign x0=((~A[0]&B[0])| (A[0]&~B[0]));

assign x1=((~A[1]&B[1])| (A[1]&~B[1]));

assign x2=((~A[2]&B[2])| (A[2]&~B[2]));

assign x3=((~A[3]&B[3])| (A[3]&~B[3]));

assign x=x0&x1&x2&x3;

assign y=((A[3]&~B[3]|(x3&A[2]&~B[2])|(x3&x2&A[1]&~B[1])|(x3&x2&x1&A[0]&~B[0]));

assign z=((~A[3]&B[3]|(x3&~A[2]&B[2])|(x3&x2&~A[1]&B[1])|(x3&x2&x1&~A[0]&B[0]));

endmodule
```

DESIGN OF MODIFIED BOOTH MULTIPLIER

AIM:
To design a modified booth multiplier in Verilog and to simulate & synthesis the same using XILINX ISE Tool.

THEORY:
Bit Pair Encoding (BPE) is a modified booth multiplication technique where the number of additions (no of partial products so generated) is reduced from n to n/2 for an 'n' bit multiplier.

Bit pair encoding of a multiplier examines 3 bits at a time and creates a 2-bit code whose values determines whether to

Add the multiplicand
Shift the multiplicand by 1 bit and then add
Subtract the multiplicand (adding 2s complement of the multiplicand to the product)
Shift the 2s complement of the multiplicand to the left by 1 bit and then add
To only shift the multiplicand to the location corresponding to the next bit-pair.

The first step of BPE algorithm is seeded with a value of 0 in the register cell to the right of the LSB of the multiplier word. Subsequent actions depend on the value of the recoded bit-pair. The index 'i' increments by two until the word is exhausted. If the word contains an odd number of bits, its sign bit must be extended by one bit to accommodate the recoding scheme. Recoding divides the multiplier word by 2, so the number of possible additions is reduced by a factor of 2. The rules for bit-pair encoding are summarized in the table provided in the left.

SOURCE CODE:

```
module booth #(parameter WIDTH=4)
 (  input     clk,
    input     enable,
    input    [WIDTH-1:0] multiplier,
    input    [WIDTH-1:0] multiplicand,
   output reg [2*WIDTH-1:0] product);

parameter   IDLE  = 2'b00,                // state encodings
            ADD   = 2'b01,
                 SHIFT = 2'b10,
                 OUTPUT = 2'b11;

reg  [1:0]    current_state, next_state;  // state registers.
reg  [2*WIDTH+1:0] a_reg,s_reg,p_reg,sum_reg;  // computational values.
reg  [WIDTH-1:0]      iter_cnt;     // iteration count for determining when done.
wire [WIDTH:0]       multiplier_neg;  // negative value of multiplier

always @(posedge clk)
  if (!enable) current_state = IDLE;
  else         current_state = next_state;

always @* begin
next_state = 2'bx;
case (current_state)
   IDLE: if (enable)  next_state = ADD;
          else  next_state = IDLE;
   ADD:  next_state = SHIFT;
   SHIFT: if (iter_cnt==WIDTH) next_state = OUTPUT;
            else   next_state = ADD;
   OUTPUT:  next_state = IDLE;
```

```
endcase
end

// negative value of multiplier.
assign multiplier_neg = -{multiplier[WIDTH-1],multiplier};

// algorithm implemenation details.
always @(posedge clk) begin
  case (current_state)
    IDLE :  begin
              a_reg    <= {multiplier[WIDTH-1],multiplier,{(WIDTH+1){1'b0}}};
              s_reg    <= {multiplier_neg,{(WIDTH+1){1'b0}}};
              p_reg    <= {{(WIDTH+1){1'b0}},multiplicand,1'b0};
              iter_cnt <= 0;
            end
    ADD  :  begin
               case (p_reg[1:0])
                 2'b01       : sum_reg <= p_reg+a_reg;
                 2'b10       : sum_reg <= p_reg+s_reg;
                 2'b00,2'b11 : sum_reg <= p_reg;
               endcase
               iter_cnt <= iter_cnt + 1;
            end
    SHIFT :  begin
               p_reg <= {sum_reg[2*WIDTH+1],sum_reg[2*WIDTH+1:1]};
             end
    OUTPUT: product =  p_reg>>1;
endcase
end//always ends

endmodule  //end of source code
```

TEST BENCH:

```
module testbench;
        // Inputs
        reg clk;
        reg enable;
        reg [3:0] multiplier;
        reg [3:0] multiplicand;
        // Outputs
        wire done;
        wire [7:0] product;
        // Instantiate the Unit Under Test (UUT)
        booth uut (
                .clk(clk),
                .enable(enable),
                .multiplier(multiplier),
                .multiplicand(multiplicand),
                .product(product)
        );

        initial
                clk=1'b0;
        always
                #5     clk=~clk;

        initial begin
                enable = 0;
                multiplier = 0;
                multiplicand = 0;

#250 enable = 1;
                multiplier = 4'b0011;
                multiplicand = 4'b0011;

          #250 enable = 1;
                multiplier = 4'b1010;
                multiplicand = 4'b0100;

          #250 enable = 1;
                multiplier = 4'b0010;
                multiplicand = 4'b1001;

          #250 enable = 1;
                multiplier = 4'b1010;
                multiplicand = 4'b1100;
          #250;
        end
endmodule //end of testbench
```

[redacted]

(OR)

BOOTHS MULTIPLIER

VERILOG CODE:

```
module multiplier(prod, busy, mc, mp, clk, start);

output [15:0] prod;

output busy;

input [7:0] mc, mp;

input clk, start;

reg [7:0] A, Q, M;

reg Q_1;

reg [3:0] count;

wire [7:0] sum, difference;

always @(posedge clk)

begin

   if (start) begin

      A <= 8'b0;

      M <= mc;

      Q <= mp;

      Q_1 <= 1'b0;

      count <= 4'b0;

   end

   else begin

      case ({Q[0], Q_1})

         2'b0_1 : {A, Q, Q_1} <= {sum[7], sum, Q};

         2'b1_0 : {A, Q, Q_1} <= {difference[7], difference, Q};
         default: {A, Q, Q_1} <= {A[7], A, Q};
        endcase
        count <= count + 1'b1;
    end
end
alu adder (sum, A, M, 1'b0); );//MODULE INSTANTIATION
//alu subtracter (difference, A, ~M, 1'b1); );//MODULE INSTANTIATION
assign prod = {A, Q};
assign busy = (count < 8);
endmodule

//The following is an alu.
//It is an adder, but capable of subtraction:
//Recall that subtraction means adding the two's complement--
//a - b = a + (-b) = a + (inverted b + 1)
//The 1 will be coming in as cin (carry-in)



module alu(out, a, b, cin);
output [7:0] out;
input [7:0] a;
input [7:0] b;
input cin;
assign out = a + b + cin;
endmodule
```

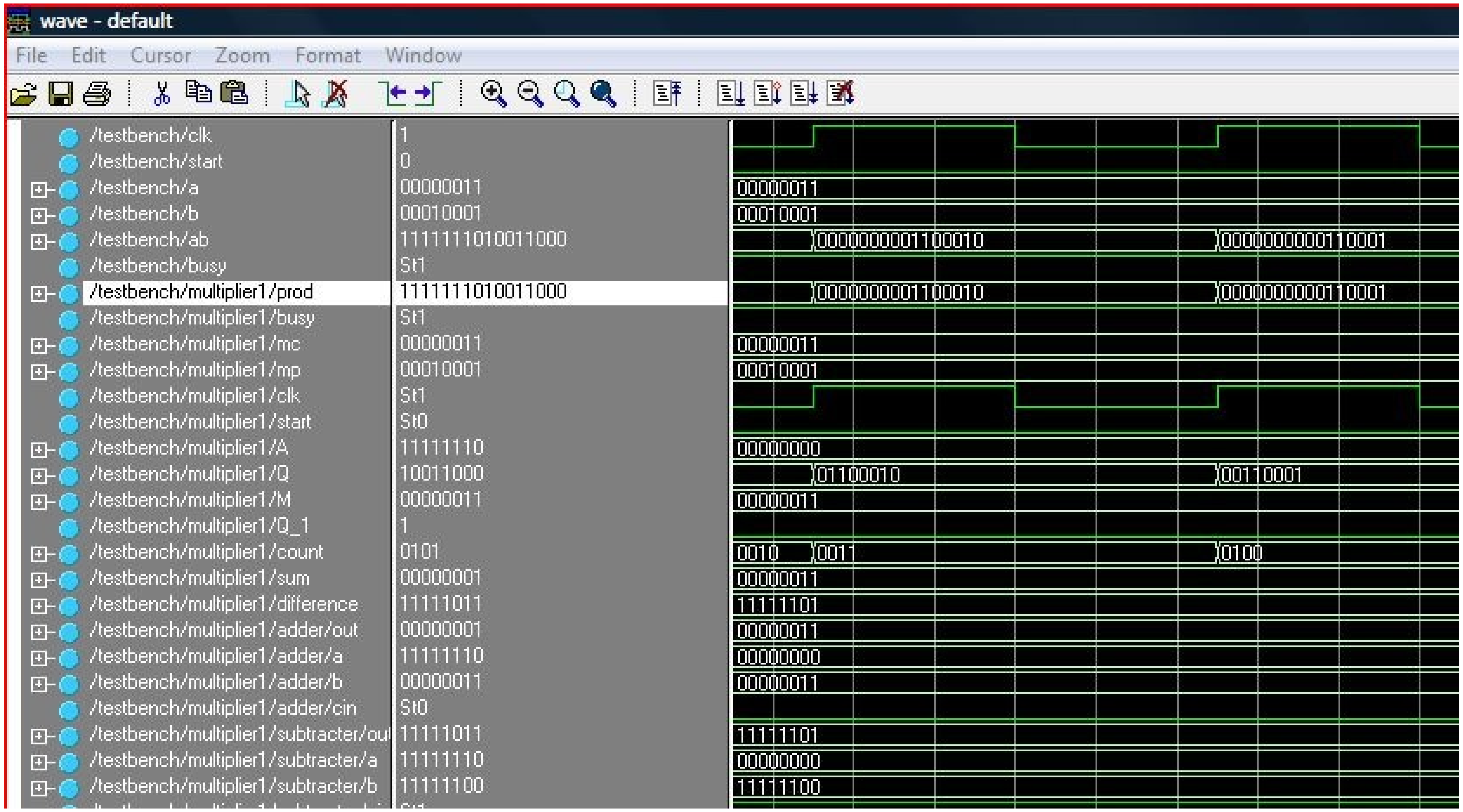
<u>TEST BENCH:</u>

```
module testbench;

reg clk, start;

reg [7:0] a, b;

wire [15:0] ab;

wire busy;

multiplier multiplier1(ab, busy, a, b, clk, start);//MODULE INSTANTIATION

initial begin

clk = 0;

$display("first example: a = 3 b = 17");

a = 3; b = 17; start = 1; #50 start = 0;

#80 $display("first example done");

$display("second example: a = 7 b = 7");

a = 7; b = 7; start = 1; #50 start = 0;

#80 $display("second example done");

$finish;

end

always #5 clk = !clk;

always @(posedge clk) $strobe("ab: %d busy: %d at time=%t", ab, busy, $stime);

endmodule
```

<u>OUTPUT:</u>

```
first example: a = 3 b = 17
ab:    17 busy: 1 at time=                    5
```

```
ab:    17 busy: 1 at time=                  15
ab:    17 busy: 1 at time=                  25
ab:    17 busy: 1 at time=                  35
ab:    17 busy: 1 at time=                  45
ab: 65160 busy: 1 at time=                  55
ab:   196 busy: 1 at time=                  65
ab:    98 busy: 1 at time=                  75
ab:    49 busy: 1 at time=                  85
ab: 65176 busy: 1 at time=                  95
ab:   204 busy: 1 at time=                 105
ab:   102 busy: 1 at time=                 115
ab:    51 busy: 0 at time=                 125
first example done
second example: a = 7 b = 7
ab:     7 busy: 1 at time=                 135
ab:     7 busy: 1 at time=                 145
ab:     7 busy: 1 at time=                 155
ab:     7 busy: 1 at time=                 165
ab:     7 busy: 1 at time=                 175
ab: 64643 busy: 1 at time=                 185
ab: 65089 busy: 1 at time=                 195
ab: 65312 busy: 1 at time=                 205
ab:   784 busy: 1 at time=                 215
ab:   392 busy: 1 at time=                 225
ab:   196 busy: 1 at time=                 235
ab:    98 busy: 1 at time=                 245
ab:    49 busy: 0 at time=                 255
second example done
```



```
/testbench/clk                          1
/testbench/start                        0
/testbench/a                            00000011
/testbench/b                            00010001
/testbench/ab                           1111111010011000
/testbench/busy                         St1
/testbench/multiplier1/prod             1111111010011000
/testbench/multiplier1/busy             St1
/testbench/multiplier1/mc               00000011
/testbench/multiplier1/mp               00010001
/testbench/multiplier1/clk              St1
/testbench/multiplier1/start            St0
/testbench/multiplier1/A                11111110
/testbench/multiplier1/Q                10011000
/testbench/multiplier1/M                00000011
/testbench/multiplier1/Q_1              1
/testbench/multiplier1/count            0101
/testbench/multiplier1/sum              00000001
/testbench/multiplier1/difference       11111011
/testbench/multiplier1/adder/out        00000001
/testbench/multiplier1/adder/a          11111110
/testbench/multiplier1/adder/b          00000011
/testbench/multiplier1/adder/cin        St0
/testbench/multiplier1/subtracter/out   11111011
/testbench/multiplier1/subtracter/a     11111110
/testbench/multiplier1/subtracter/b     11111100
```