Basics in Verilog Design

1.Logic Gates using Gate Level model

2.Logic Gates using Data model

3.Half Adder

4.Full Adder

5.Full Adder using Half adders

6.Half Subtractor

7.Full Subtractor

8.Full Subtractor using Half Subtractors

9.Ripple Carry Adder(4-bit)

*10.Multiplexer(2*1)*

*11.Multiplexer(8*1)*

*12.Multiplexer(8*1) using Multiplexer(2*1)*

*13.DeMultiplexer(1*8)*

*14.Encoder(8*3)*

15.Priority Encoder

16. 4-bit Comparator

17.Decimal-BCD Encoder

18.Octal-Binary Encoder

19.Hexadecimal-Binary Encoder

20.Binary-Gray Converter

21.Gray-Binary Converter

22.Even Parity Generator

23.Odd Parity Generator

24.Even Parity Checker

25.Odd Parity Checker

26.Binary-Seven Segment Display Converter

27.Carry Look Ahead Adder

28.BCD Adder

29.BCD-Excess_3 Converter

30.Carry Save Adder

31.Squares of 3bit numbers

32.Tristate Buffer

33.RS Latch using NOR gates

34.RS Latch using NAND gates

35.SR Flipflop

36.JK Flipflop

37.D Flipflop

38.T Flipflop

39.D Latch

40.Asynchronous counter using T Flipflops

41.Synchronous Up and Down Counter

42.Johnson Counter

43.Ring Counter

44.Serial In Serial Out Shift Register(SISO)

45.Serial In Parllel Out Shift Register(SIPO)

46.Parllel In Serial Out Shift Register(PISO)

47.Parllel In Parllel Out Shift Register(PIPO)

48.Master-Slave D Flipflop

49.Sequence Detector using Mealy FSM

50.Sequence Detector using Moore FSM

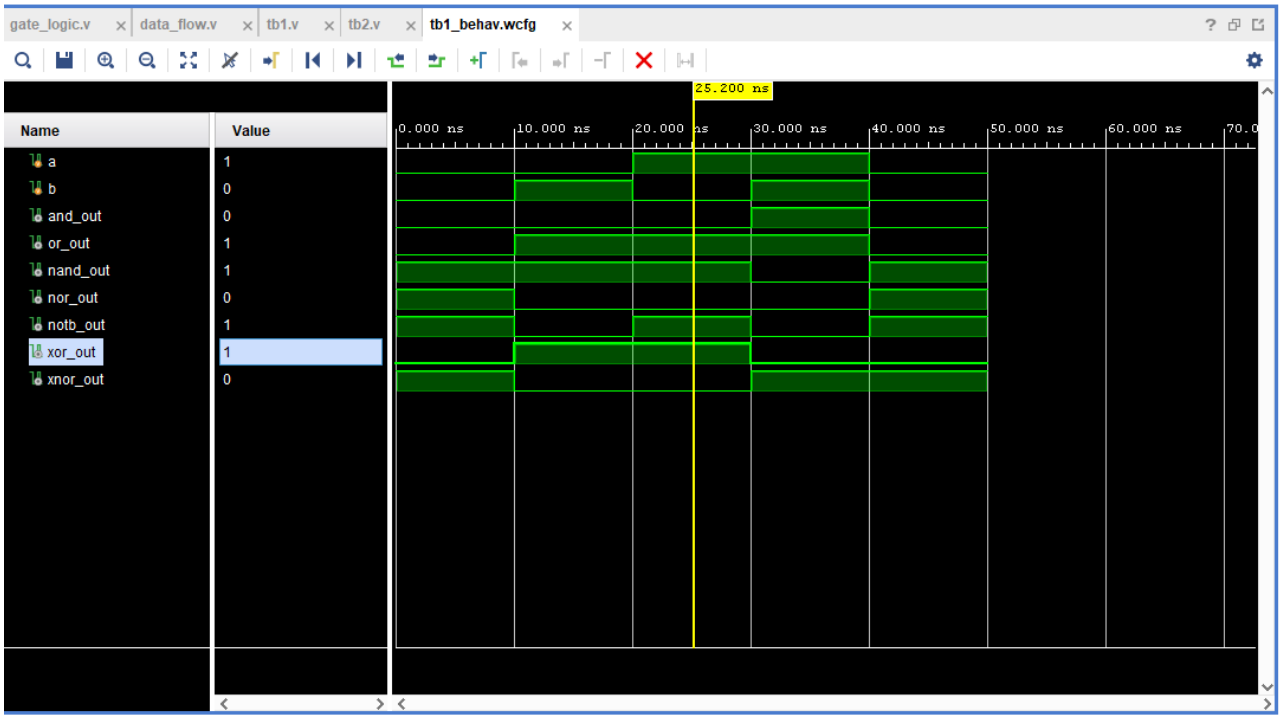
1.Logic Gates using Gate Level model

Verilog Code:

```
module gate_logic(a,b,and_out,or_out,nand_out,nor_out,notb_out,xor_out,xnor_out);
input a,b;
output and_out,or_out,nand_out,nor_out,notb_out,xor_out,xnor_out;
and a1 (and_out,a,b);
or o1(or_out,a,b);
nand n1(nand_out,a,b);
nor n2(nor_out,a,b);
not n3(notb_out,b);
xor x1(xor_out,a,b);
xnor x2 (xnor_out,a,b);
endmodule
```

Test Bench:

```
module tb1( );
reg a,b;
wire and_out,or_out,nand_out,nor_out,notb_out,xor_out,xnor_out;
gate_logic uut(
  .a(a),
  .b(b),
  .and_out(and_out),
  .or_out(or_out),
  .notb_out(notb_out),
  .nand_out(nand_out),
  .nor_out(nor_out),
  .xor_out(xor_out),
  .xnor_out(xnor_out)
);
initial
begin
  a=0; b=0; #10
  a=0; b=1; #10
  a=1; b=0; #10
  a=1; b=1; #10
  a=0; b=0; #10
  $finish;
end
endmodule
```



2.Logic Gates using Data model

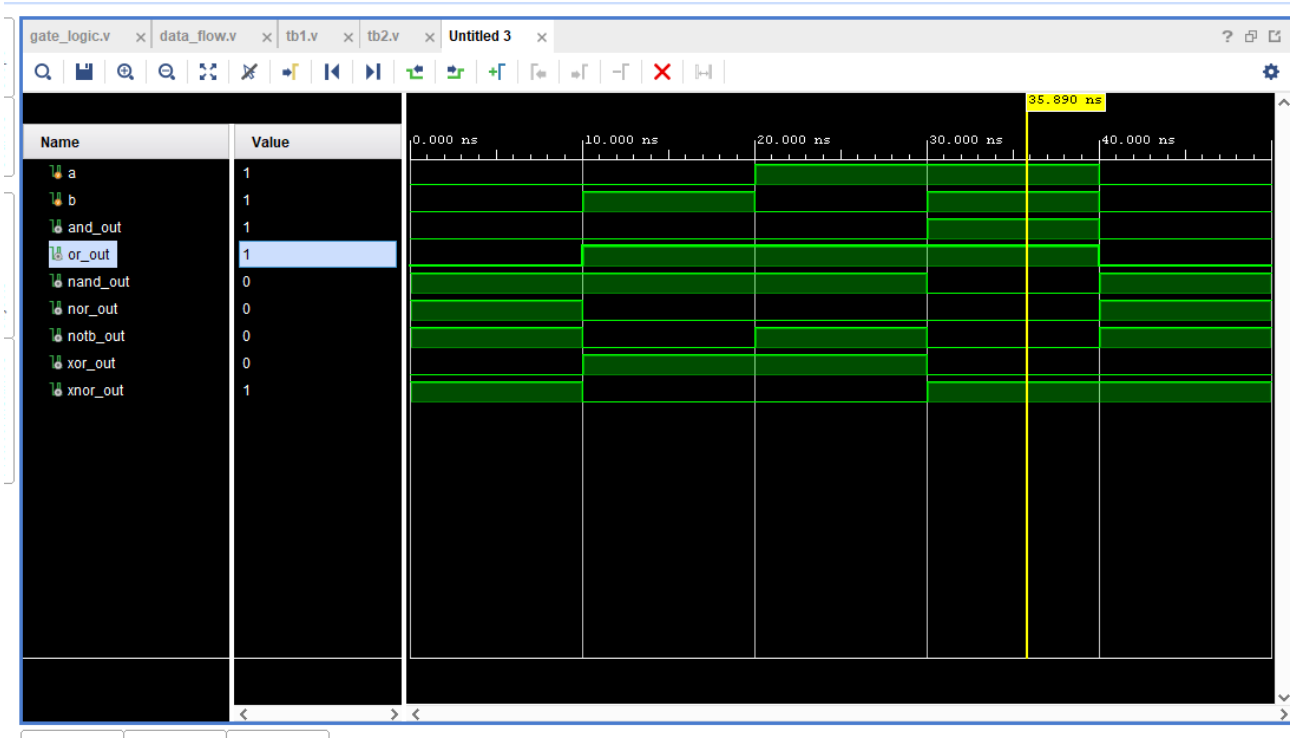
Verilog Code:

```
module data_flow(a,b,and_out,or_out,nand_out,nor_out,notb_out,xor_out,xnor_out);
input a,b;
output and_out,or_out,nand_out,nor_out,notb_out,xor_out,xnor_out;
assign and_out=a&b;
assign or_out=a|b;
assign nand_out=~(a&b);
assign nor_out=~(a|b);
assign notb_out=~b;
assign xor_out=a^b;
assign xnor_out=~(a^b);
endmodule
```

Test Bench:

```
module tb2( );
reg a,b;
wire and_out,or_out,nand_out,nor_out,notb_out,xor_out,xnor_out;

data_flow uut(
  .a(a),
  .b(b),
  .and_out(and_out),
  .or_out(or_out),
  .notb_out(notb_out),
  .nand_out(nand_out),
  .nor_out(nor_out),
```

```
  .xor_out(xor_out),
  .xnor_out(xnor_out)
);

initial
begin
  a=0; b=0; #10
  a=0; b=1; #10
  a=1; b=0; #10
  a=1; b=1; #10
  a=0; b=0; #10
  $finish;
end
endmodule
```



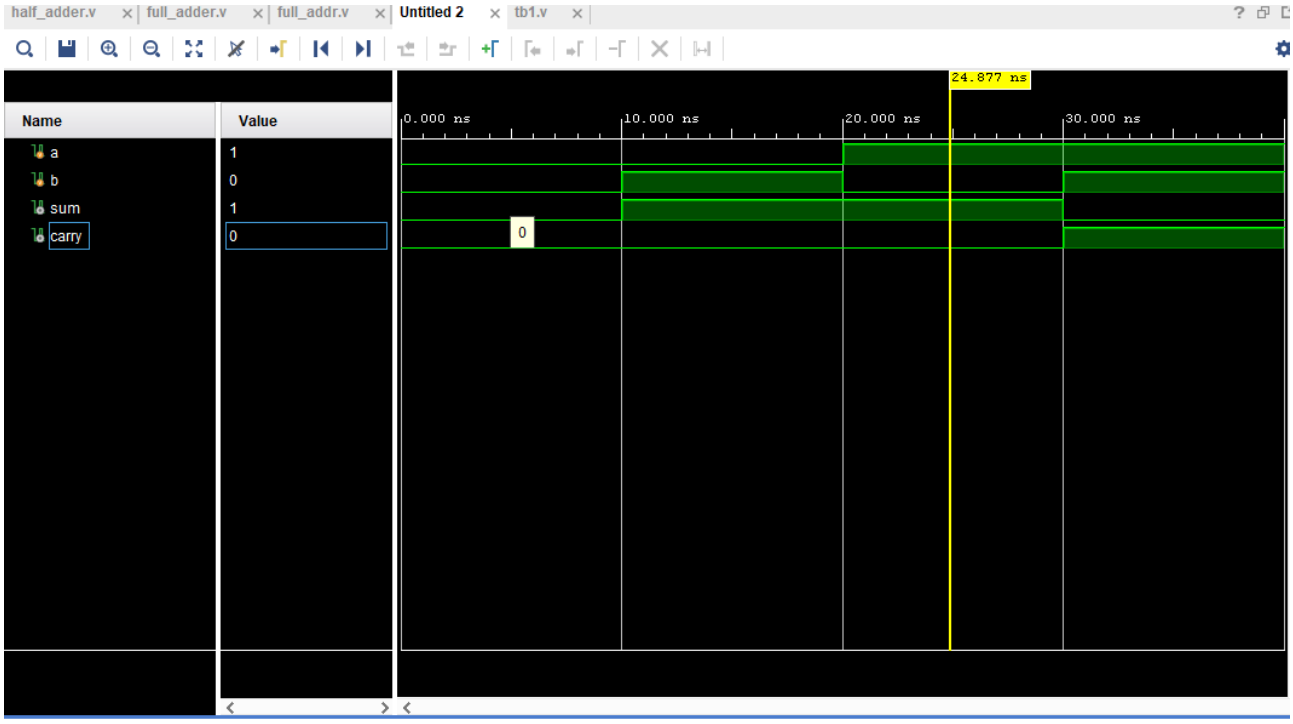
3.Half Adder

Verilog Code:

```
module half_adder(input a,b ,
output sum, carry );
assign sum=a^b;
assign carry= a&b;
endmodule
```

Test Bench:

```
module tb1();
reg a,b; wire sum,carry;
half_adder uut(a,b,sum,carry);
initial
begin
a=0;b=0; #10
a=0;b=1; #10
a=1;b=0; #10
a=1;b=1;#10
$finish;
end
endmodule
```



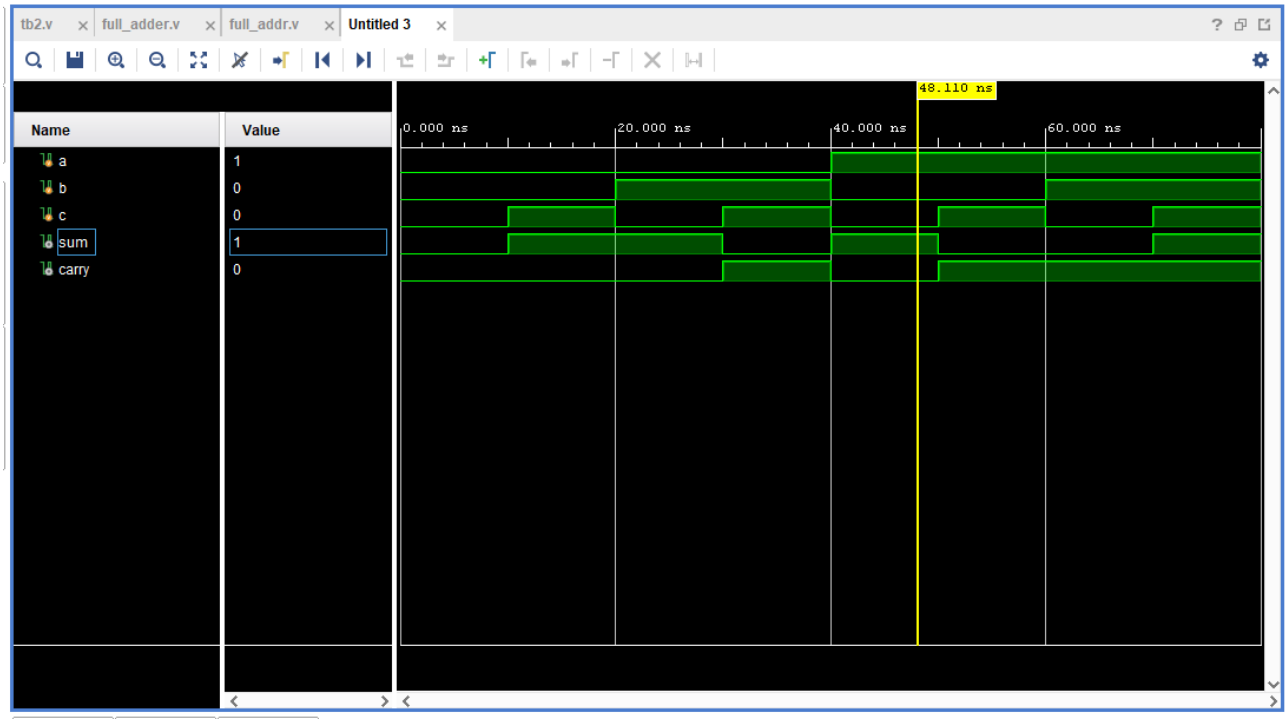
4.Full Adder

Verilog Code:

```
module full_adder(input a,b,c,output sum, carry);
assign sum= a^b^c;
assign w1=a&b;
assign w2=b&c;
assign w3=a&c;
assign carry=w1|w2|w3;
endmodule
```

TestBench:

```
module tb2();
reg a,b,c; wire sum,carry;
full_adder uut(a,b,c ,sum,carry);
initial
begin
a=0;b=0;c=0; #10
a=0;b=0;c=1; #10
a=0;b=1;c=0; #10
a=0;b=1;c=1; #10
a=1;b=0;c=0; #10
a=1;b=0;c=1; #10
a=1;b=1;c=0; #10
a=1;b=1;c=1; #10
$finish;
end
endmodule
```



5.Full Adder using Half adders

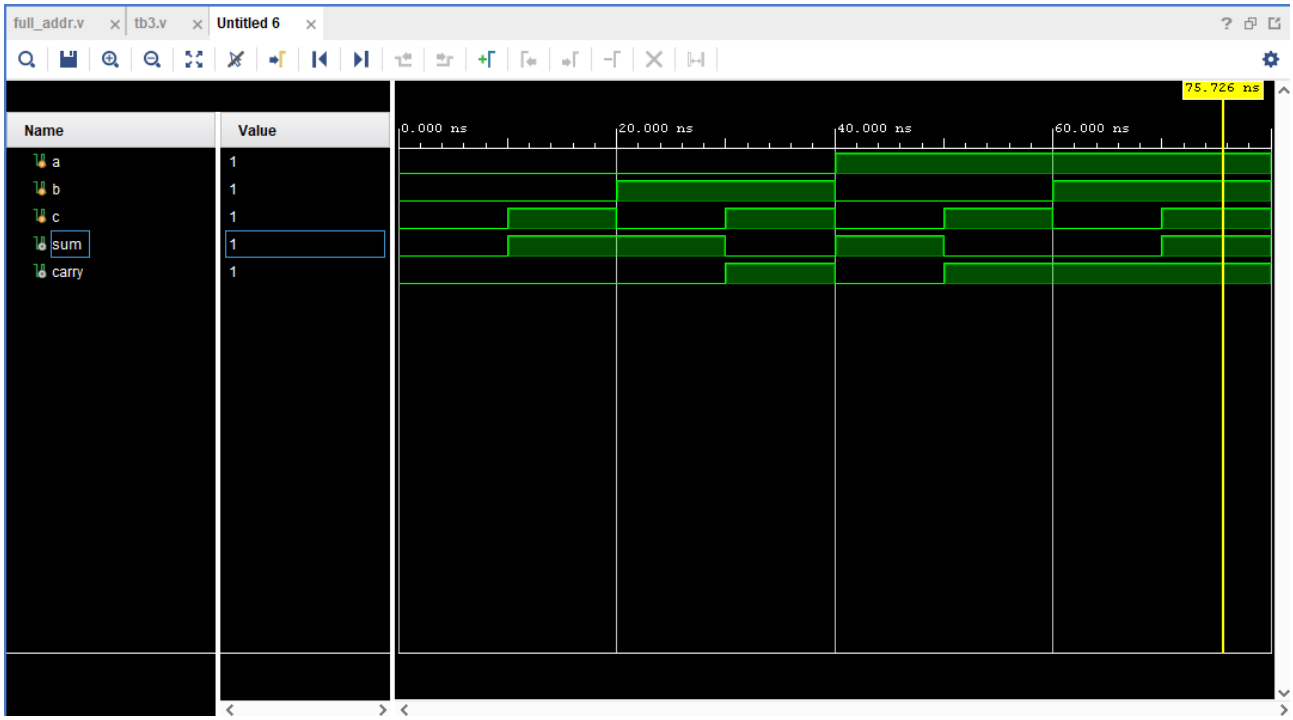
Verilog Code:

```
module full_addr(input a,b,c, output sum, carry);
half_adder h1(a,b,w1,w2);
half_adder h2(w1,c,sum,w3);
```

```
assign carry=w2|w3;
endmodule
```

TestBench:

```
module tb3( );
reg a,b,c; wire sum,carry;
full_addr uut(a,b,c,sum,carry);
initial
begin
 for ( integer count = 0; count < 8; count = count + 1)
        begin
        {a, b, c}= count[2:0];
        #10;
         end
 $finish;
 end
endmodule
```



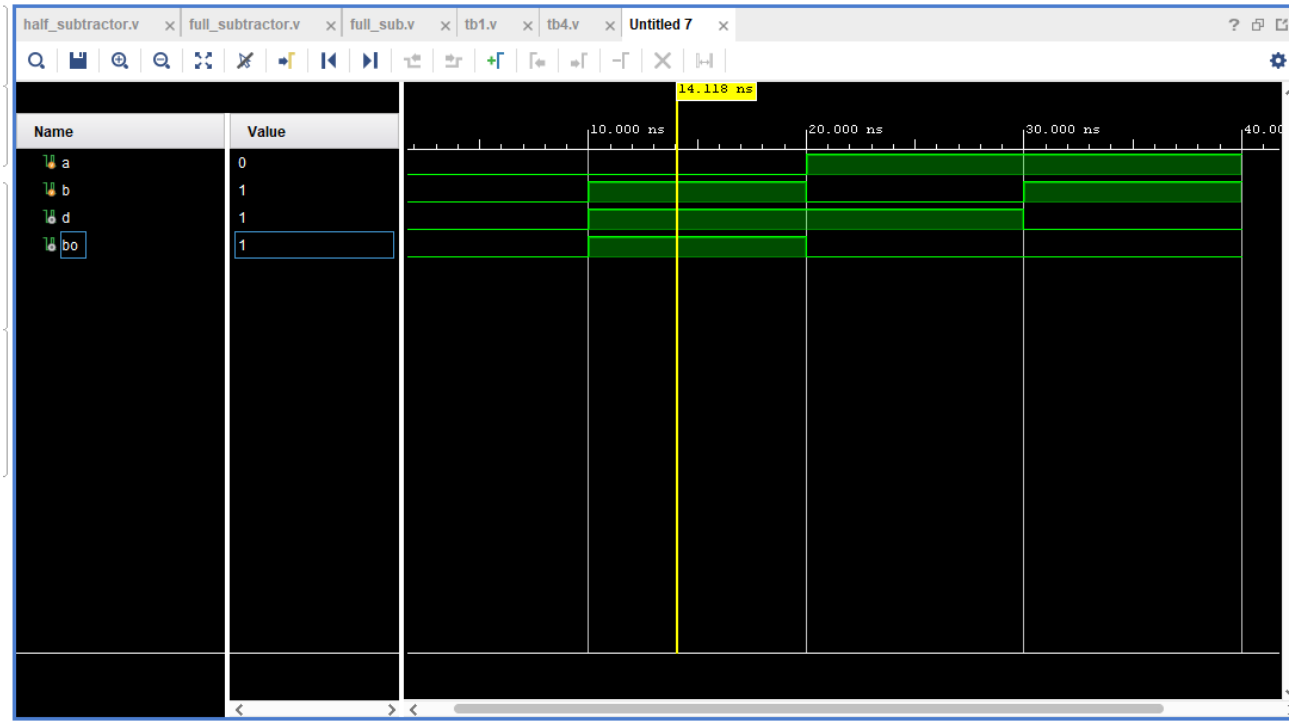
6.Half Subtractor

Verilog Code:

```
module half_subtractor( input a,b,output d,bo);
assign d=a^b;
assign bo=(~a)&b;
endmodule
```

TestBench:

```
module tb4();
reg a,b; wire d,bo;
half_subtractor uut(a,b,d,bo);
initial
begin
a=0;b=0; #10
a=0;b=1; #10
a=1;b=0; #10
a=1;b=1;#10
$finish;
end
endmodule
```



7.Full Subtractor

Verilog Code:

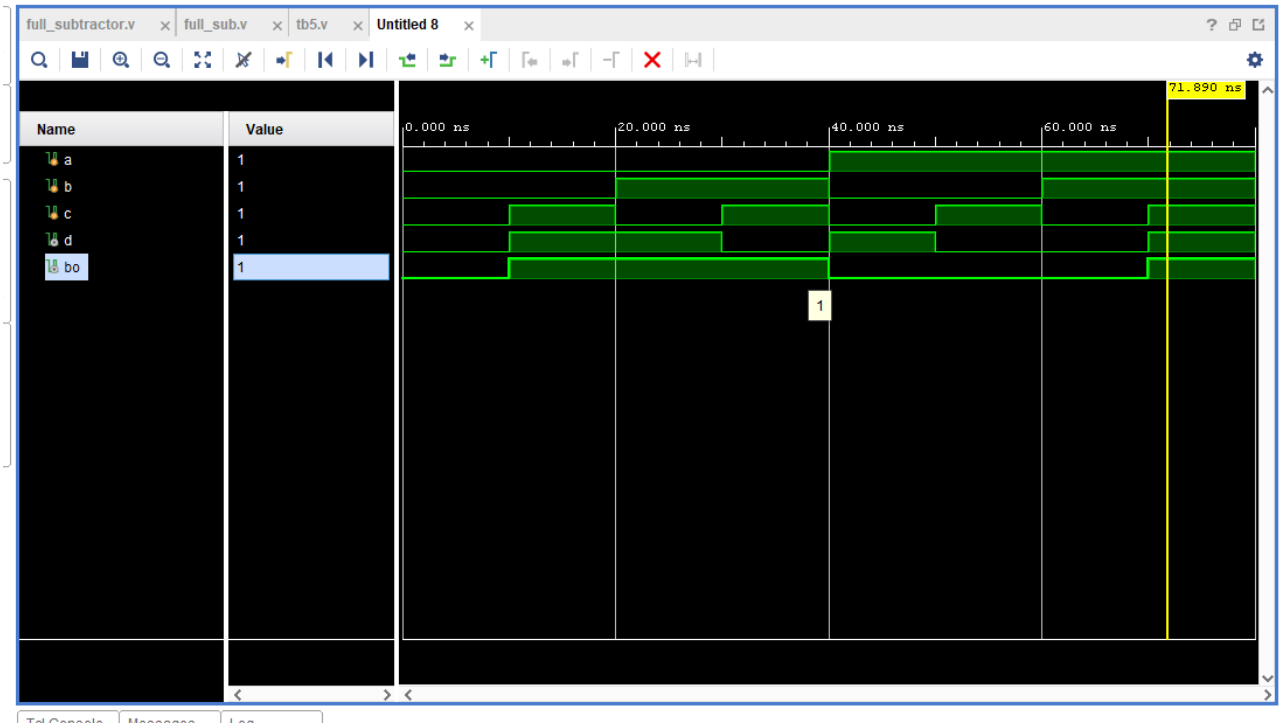
```
module full_subtractor(input a,b,c ,output d, bo );
assign d=a^b^c;
assign w1=(~a)&b;
assign w2=(~a)&c;
assign w3=b&c;
assign bo= w1|w2|w3;
endmodule
```

TestBench:

```
module tb5( );
reg a,b,c ; wire d, bo;
```

full_subtractor uut(a,b,c ,d,bo);
initial
begin
a=0;b=0;c=0; #10
a=0;b=0;c=1; #10
a=0;b=1;c=0; #10
a=0;b=1;c=1; #10
a=1;b=0;c=0; #10
a=1;b=0;c=1; #10
a=1;b=1;c=0; #10
a=1;b=1;c=1; #10
$finish;
end
endmodule

8.Full Subtractor using Half Subtractors



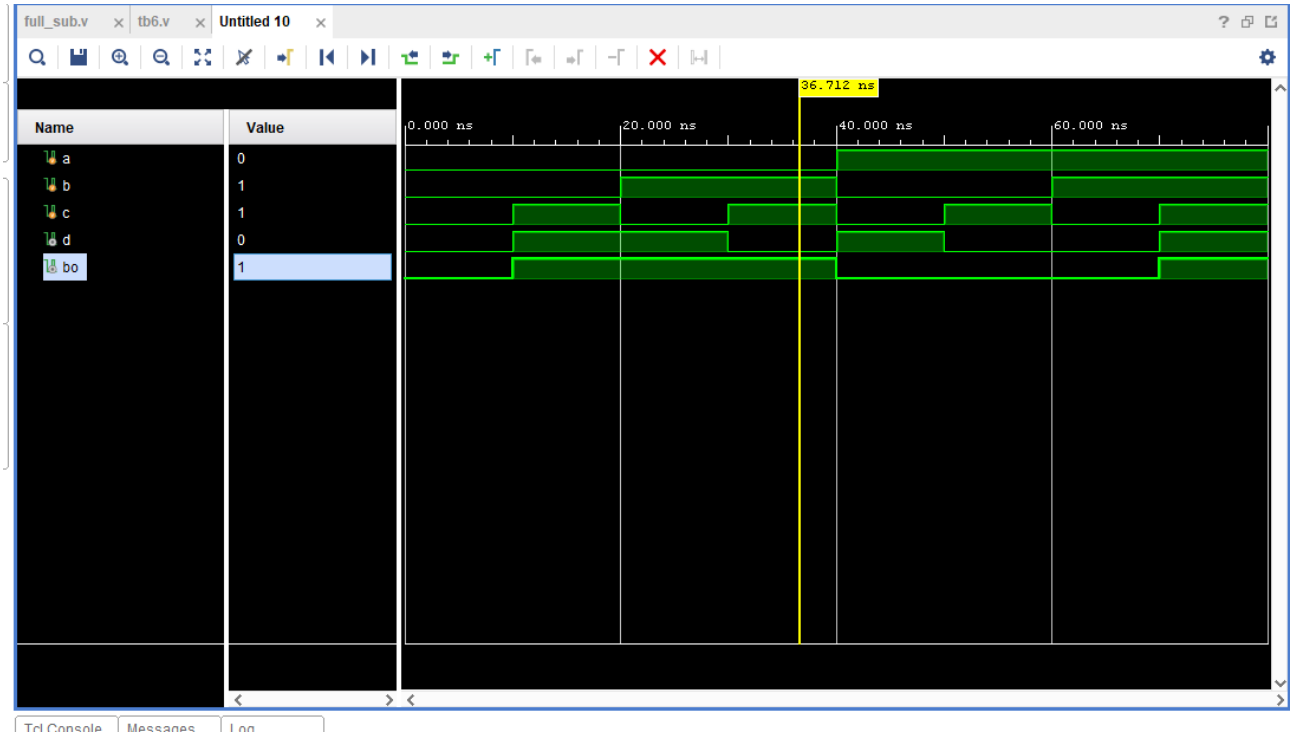
Verilog Code:

module full_sub(input a,b,c,output d,bo);
half_subtractor h1(a,b,w1,w2);
half_subtractor h2(w1,c,d,w3);
assign bo= w2|w3;
endmodule

TestBench:

module tb6( );
reg a,b,c; wire d,bo;
full_sub uut(a,b,c ,d,bo);

initial
begin
a=0;b=0;c=0; #10
a=0;b=0;c=1; #10
a=0;b=1;c=0; #10
a=0;b=1;c=1; #10
a=1;b=0;c=0; #10
a=1;b=0;c=1; #10
a=1;b=1;c=0; #10
a=1;b=1;c=1; #10
$finish;
end
endmodule



9.Ripple Carry Adder(4-bit)

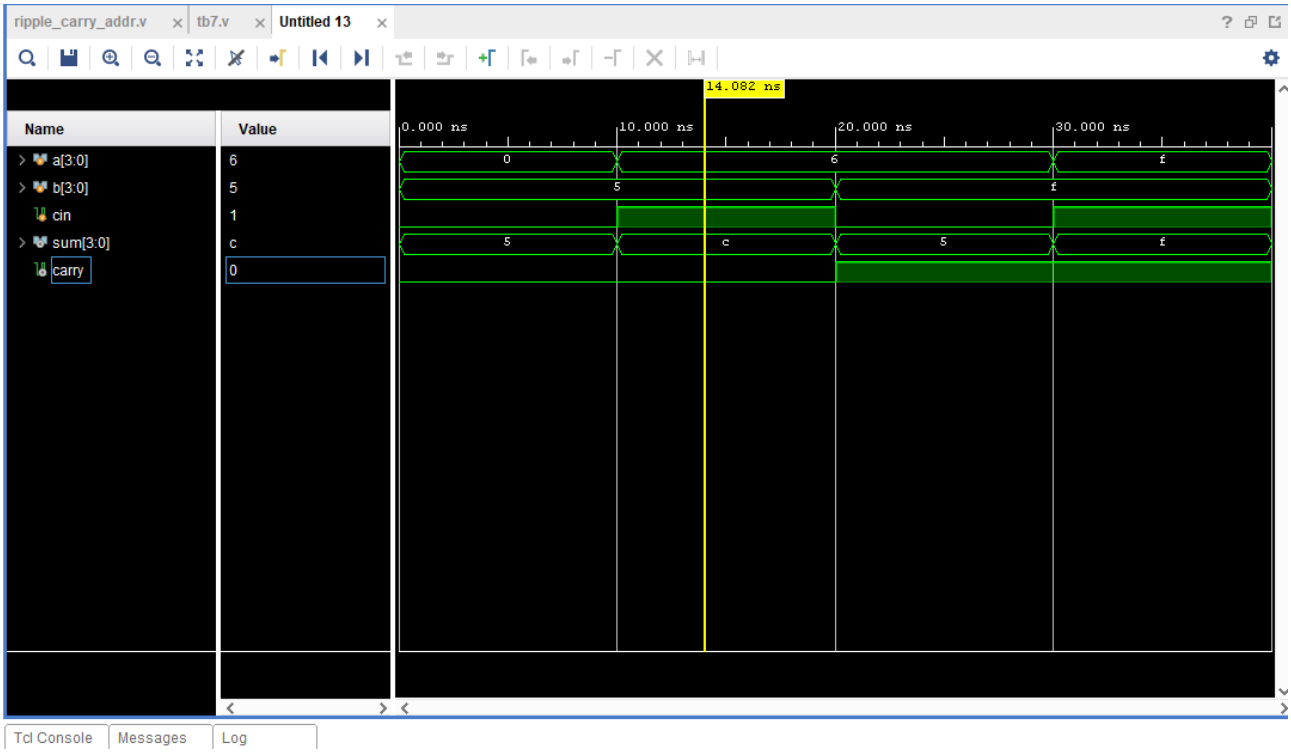
Verilog Code:

module ripple_carry_addr(a,b,cin, sum,carry);
input [3:0] a,b;
input cin;
output [3:0] sum;
output carry;
wire[2:0] c;
full_adder f1(a[0],b[0],cin,sum[0],c[0]);
full_adder f2(a[1],b[1],c[0],sum[1],c[1]);
full_adder f3(a[2],b[2],c[1],sum[2],c[2]);

full_adder f4(a[3],b[3],c[2],sum[3],carry);
endmodule

TestBench:

module tb7( );
reg [3:0] a,b; reg cin;wire [3:0] sum;wire carry;
ripple_carry_addr uut (a,b, cin,sum,carry);
initial begin
a=4'b0000; b=4'b0101; cin=0; #10
a=4'b0110; b=4'b0101; cin=1; #10
a=4'b0110; b=4'b1111; cin=0; #10
a=4'b1111; b=4'b1111; cin=1; #10
$finish;
end
endmodule



*10.Multiplexer(2*1)*

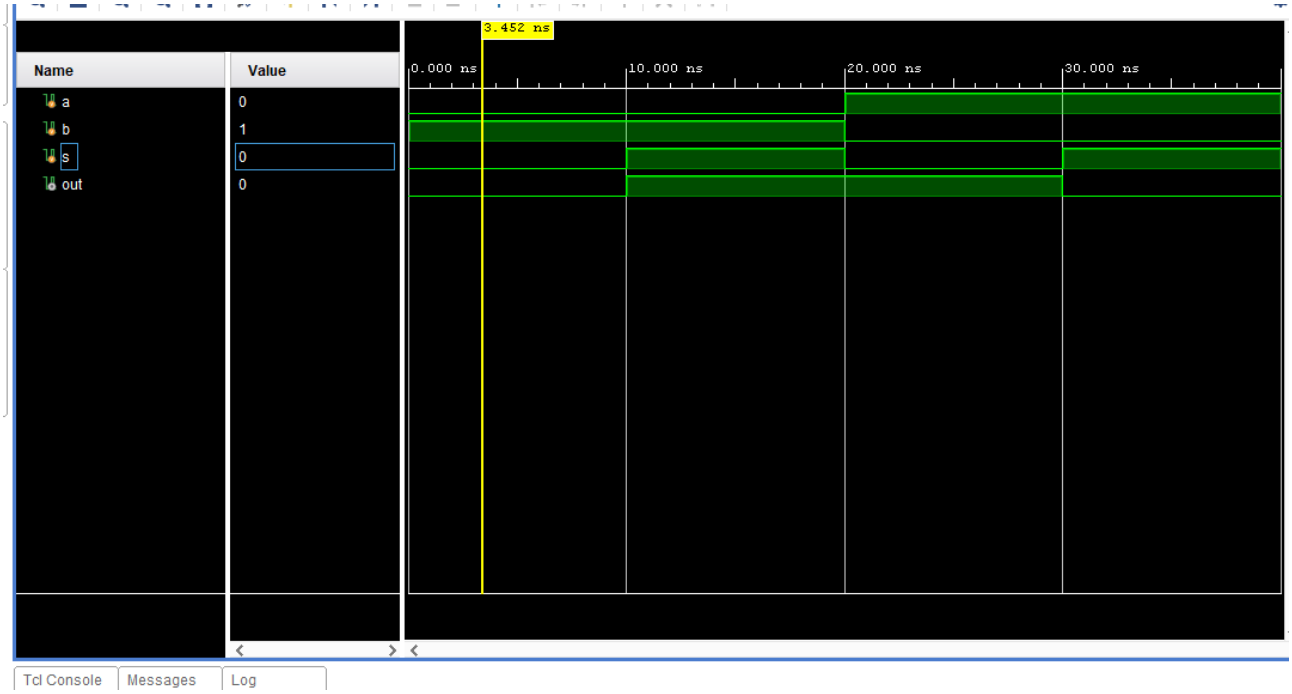
Verilog Code:

module x1_mux(a,b,s,out);
input a,b,s;
output out;
assign out=((~s)&a)|(s&b);
endmodule

TestBench:

module tb1( );
reg a,b,s; wire out;

x1_mux uut(a,b,s,out);
initial
begin
a=0;b=1; s=0; #10
a=0;b=1; s=1; #10
a=1;b=0; s=0; #10
a=1;b=0; s=1; #10
$finish;
end
endmodule



*11.Multiplexer(8*1)*

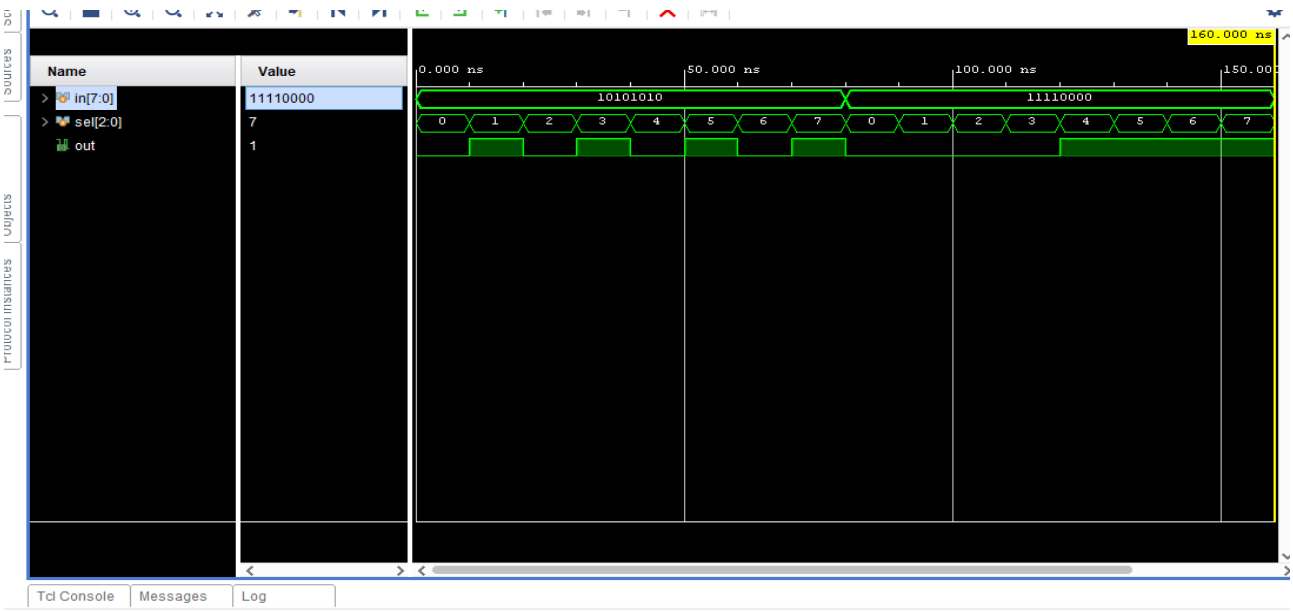
Verilog Code:

module eight_x_one_mux(in,sel,out);
input [7:0] in;
input [2:0] sel;
output reg out;
always @ (*)
begin
case(sel)
3'b000: out=in[0];
3'b001: out=in[1];
3'b010: out=in[2];
3'b011: out=in[3];
3'b100: out=in[4];
3'b101: out=in[5];
3'b110: out=in[6];

```
3'b111: out=in[7];
default: out=0;
endcase
end
endmodule
```

TestBench:

```
module tb3(out);
reg [7:0] in;
reg [2:0] sel;
output out;
eight_x_one_mux dut (in,sel,out);
initial
begin
in=8'b10101010;
 sel=3'b000; #10
sel=3'b001; #10
sel=3'b010; #10
sel=3'b011; #10
sel=3'b100; #10
sel=3'b101; #10
sel=3'b110; #10
sel=3'b111; #10
in=8'b11110000;
sel=3'b000; #10
sel=3'b001; #10
sel=3'b010; #10
sel=3'b011; #10
sel=3'b100; #10
sel=3'b101; #10
sel=3'b110; #10
sel=3'b111; #10
$finish;

end

endmodule
```



*12.Multiplexer(8*1) using Multiplexer(2*1)*

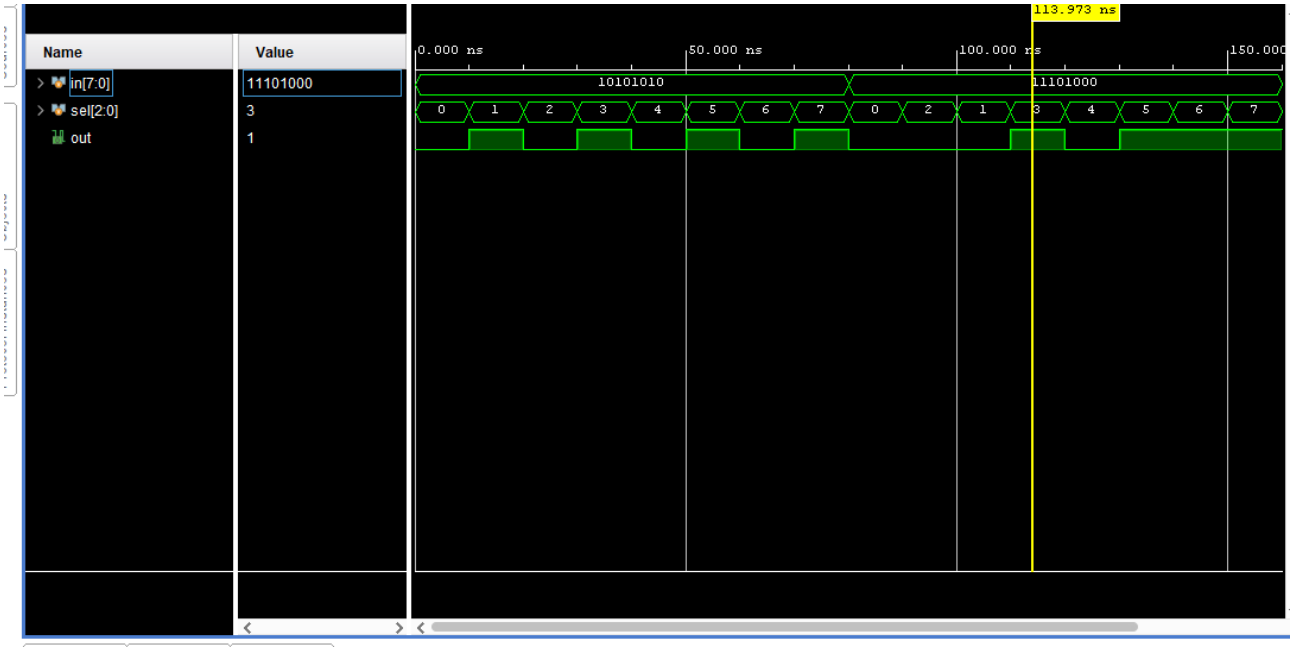
Verilog Code:

```
module x8(in,sel,out);
input [7:0]in;
input [2:0] sel;
output out;
x1_mux m1(in[0],in[1],sel[0],w1);
x1_mux m2(in[2],in[3],sel[0],w2);
x1_mux m3(in[4],in[5],sel[0],w3);
x1_mux m4(in[6],in[7],sel[0],w4);
x1_mux m5(w1,w2,sel[1],w5);
x1_mux m6(w3,w4,sel[1],w6);
x1_mux m7(w5,w6,sel[2],out);
endmodule
```

TestBench:

```
module tb2(out);
reg [7:0] in;
reg [2:0] sel;
output out;
x8 dut (in,sel,out);
initial
begin
in=8'b10101010;
 sel=3'b000; #10
sel=3'b001; #10
sel=3'b010; #10
sel=3'b011; #10
```

sel=3'b100; #10
sel=3'b101; #10
sel=3'b110; #10
sel=3'b111; #10
in=8'b11101000;
sel=3'b000; #10
sel=3'b010; #10
sel=3'b001; #10
sel=3'b011; #10
sel=3'b100; #10
sel=3'b101; #10
sel=3'b110; #10
sel=3'b111; #10
$finish;
end
endmodule



*13.DeMultiplexer(1*8)*

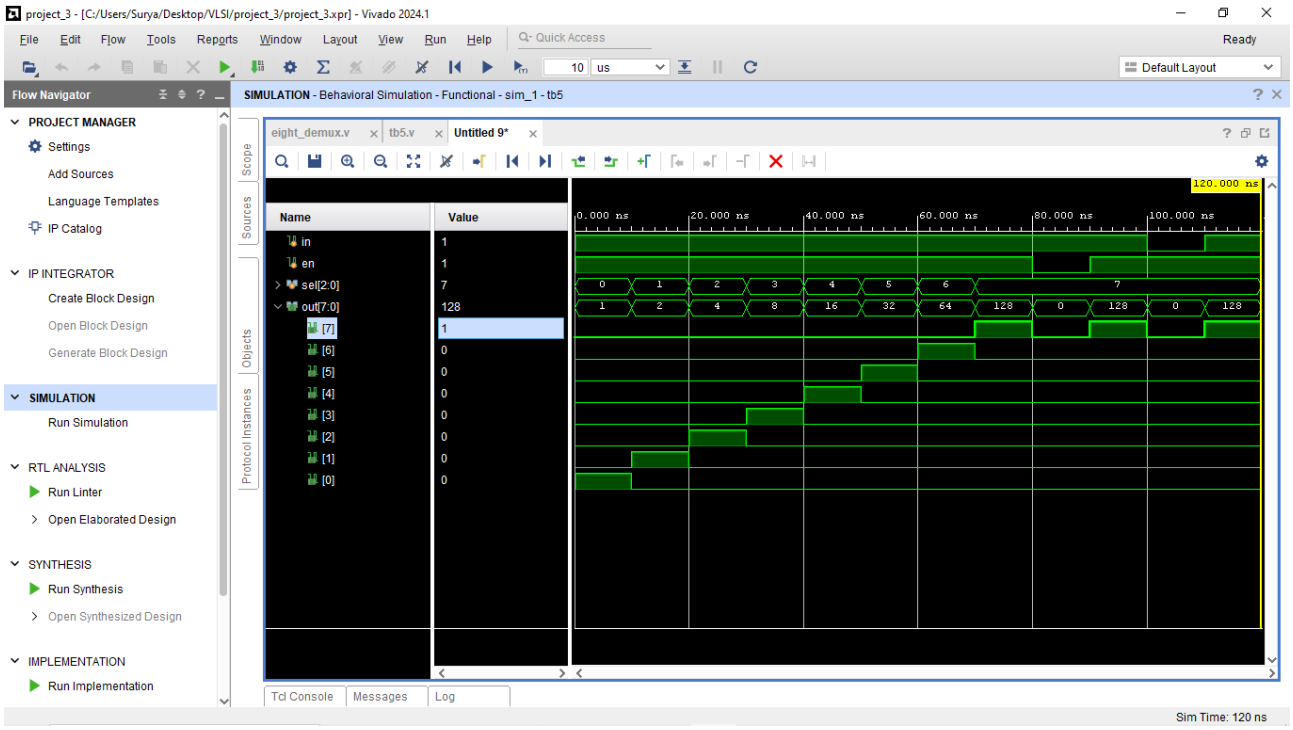
Verilog Code:

module eight_demux(in,en,sel,out);
input in,en;
input[2:0]sel;
output reg [7:0]out;
always @(*)
begin
out=8'b00000000;
if(en)

```
begin
case(sel)
3'b000: out[0]=in;
3'b001: out[1]=in;
3'b010: out[2]=in;
3'b011: out[3]=in;
3'b100: out[4]=in;
3'b101: out[5]=in;
3'b110: out[6]=in;
3'b111: out[7]=in;
default:out=8'b00000000;
endcase
end
else
out=8'b00000000;
end
endmodule
```

TestBench:

```
module tb5(out );
reg in,en;
reg[2:0]sel;
output [7:0] out;
eight_demux dut(in,en,sel,out);
initial
begin
in=1;en=1;
sel=3'b000; #10
sel=3'b001; #10
sel=3'b010; #10
sel=3'b011; #10
sel=3'b100; #10
sel=3'b101; #10
sel=3'b110; #10
sel=3'b111; #10
en=0; #10
en=1; #10
in=0; #10
in=1; #10
$finish;
end
endmodule
```



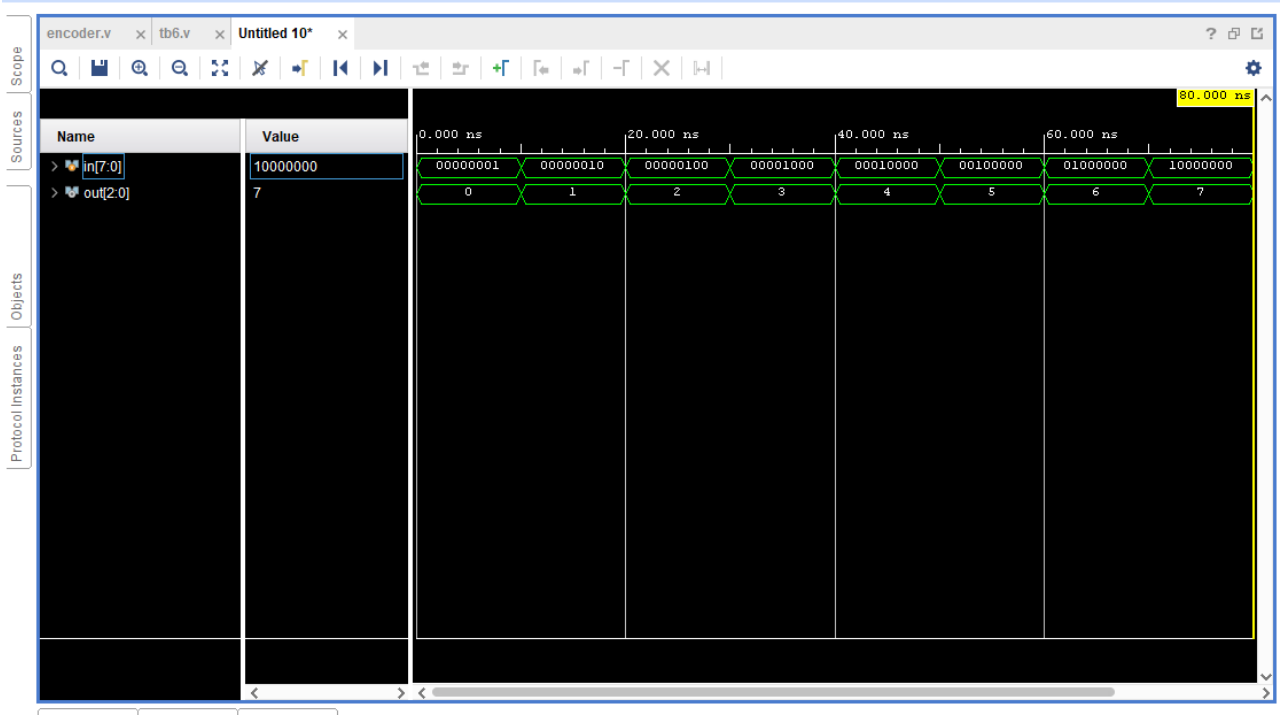
14.Encoder

Verilog Code:

```
module encoder(in,out);
input[7:0] in;
output reg[2:0] out;
always @(*)
begin
out=3'b000;
case(in)
8'b00000000: out=3'b000;
8'b00000001: out=3'b000;
8'b00000010: out=3'b001;
8'b00000100: out=3'b010;
8'b00001000: out=3'b011;
8'b00010000: out=3'b100;
8'b00100000: out=3'b101;
8'b01000000: out=3'b110;
8'b10000000: out=3'b111;
default: out=3'b000;
endcase
end
endmodule
```

TestBench:

```
module tb6(   );
reg [7:0]in;
wire [2:0] out;
encoder dut(in,out);
initial
begin
in=8'b00000001; #10
in=8'b00000010; #10
in=8'b00000100; #10
in=8'b00001000; #10
in=8'b00010000; #10
in=8'b00100000; #10
in=8'b01000000; #10
in=8'b10000000; #10
$finish;
end
endmodule
```



15.Priority Encoder

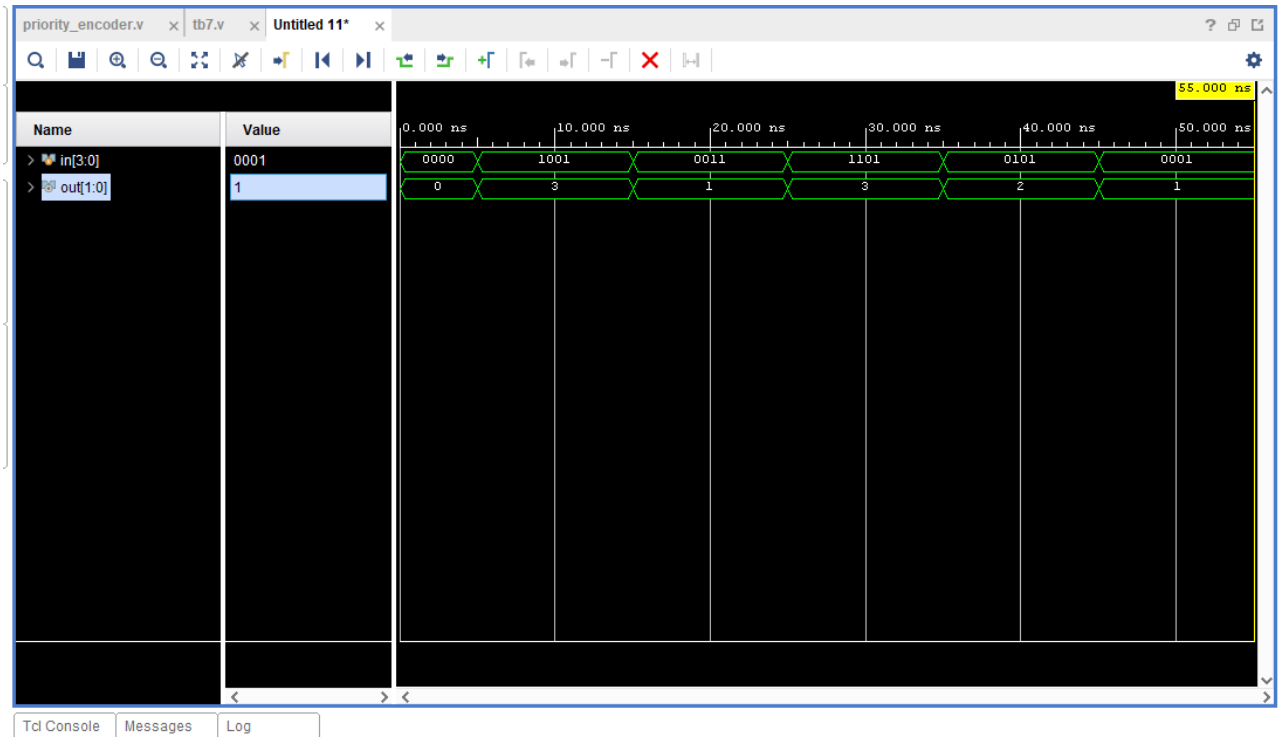
Verilog Code:

```
module priority_encoder(in,out );
input [3:0] in;
output [1:0] out;
assign out[1]= in[2]|in[3];
```

```
assign out[0]=in[3]|((~in[2])&in[0]);
endmodule
```

TestBench:

```
module tb7( );
reg [3:0] in;
wire [1:0] out;
priority_encoder uut(in,out);
initial
begin
in=4'b0000; #5
in=4'b1001; #10
in=4'b0011; #10
in=4'b1101; #10
in=4'b0101; #10
in=4'b0001; #10
$finish;
end
endmodule
```



16. 4-bit Comparator

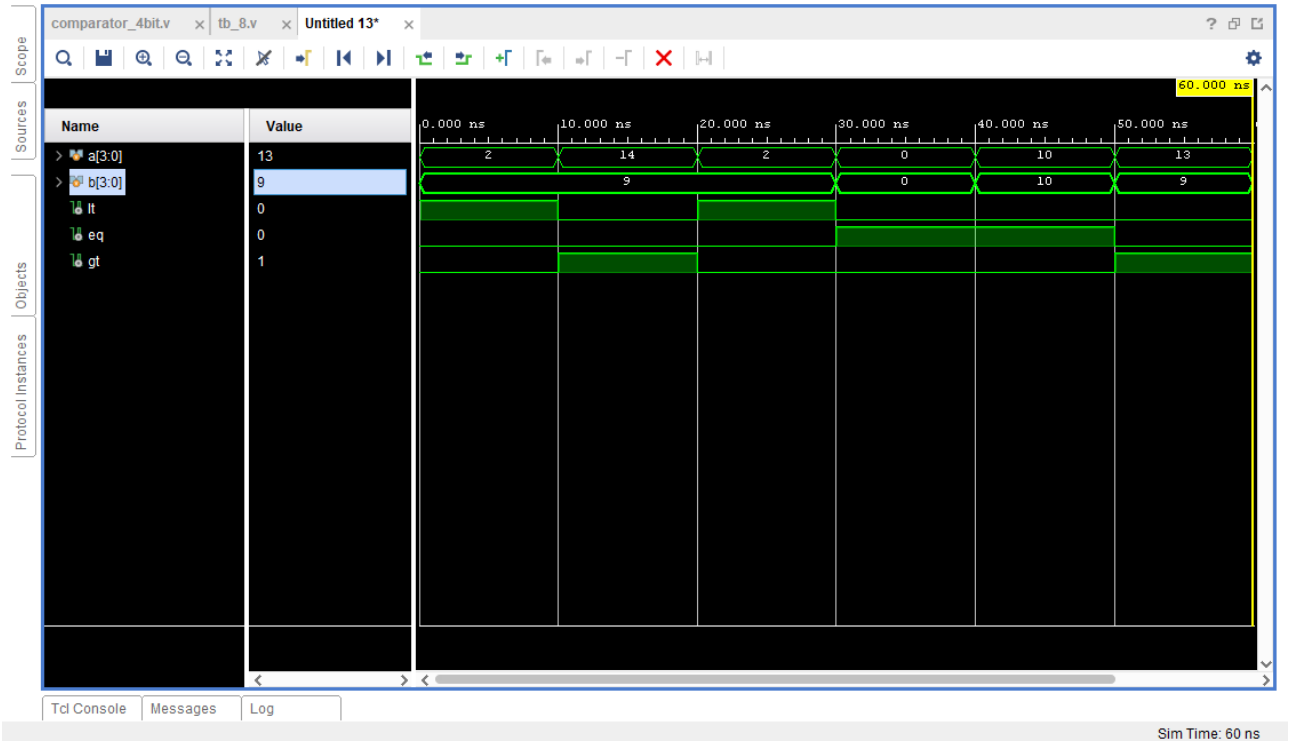
Verilog Code:

```
module comparator_4bit(a,b,lt,eq,gt );
input [3:0] a,b;
```

```
output lt,eq,gt;
assign lt=(a<b);
assign eq=(a==b);
assign gt=(a>b);
endmodule
```

TestBench:

```
module tb_8();
reg [3:0] a,b;
wire lt,eq,gt;
comparator_4bit uut(a,b,lt,eq,gt);
initial
begin
a=4'b0010; b=4'b1001; #10
a=4'b1110; b=4'b1001; #10
a=4'b0010; b=4'b1001; #10
a=4'b0000; b=4'b0000; #10
a=4'b1010; b=4'b1010; #10
a=4'b1101; b=4'b1001; #10
$finish;
end
endmodule
```



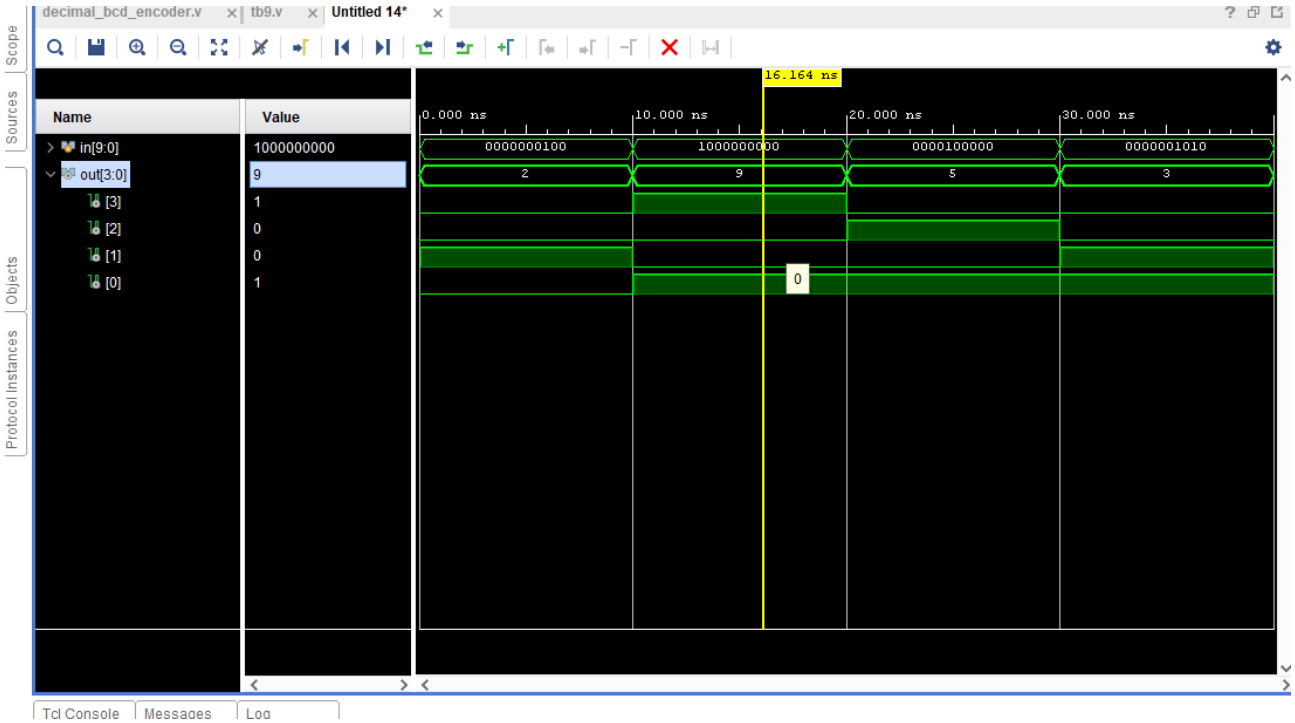
17.Decimal-BCD Encoder

VerilogCode:

```
module decimal_bcd_encoder(in,out );
input[9:0]in;
output [3:0] out;
assign out[3]=in[8]|in[9];
assign out[2]=in[4]|in[5]|in[6]|in[7];
assign out[1]=in[2]|in[3]|in[6]|in[7];
assign out[0]=in[1]|in[3]|in[5]|in[7]|in[9];
endmodule
```

TestBench:

```
module tb9( );
reg [9:0] in;
wire [3:0] out;
decimal_bcd_encoder uut (in,out);
initial
begin
in=10'b0000000100; #10
in=10'b1000000000; #10
in=10'b0000100000; #10
in=10'b0000001010; #10
$finish;
end
endmodule
```



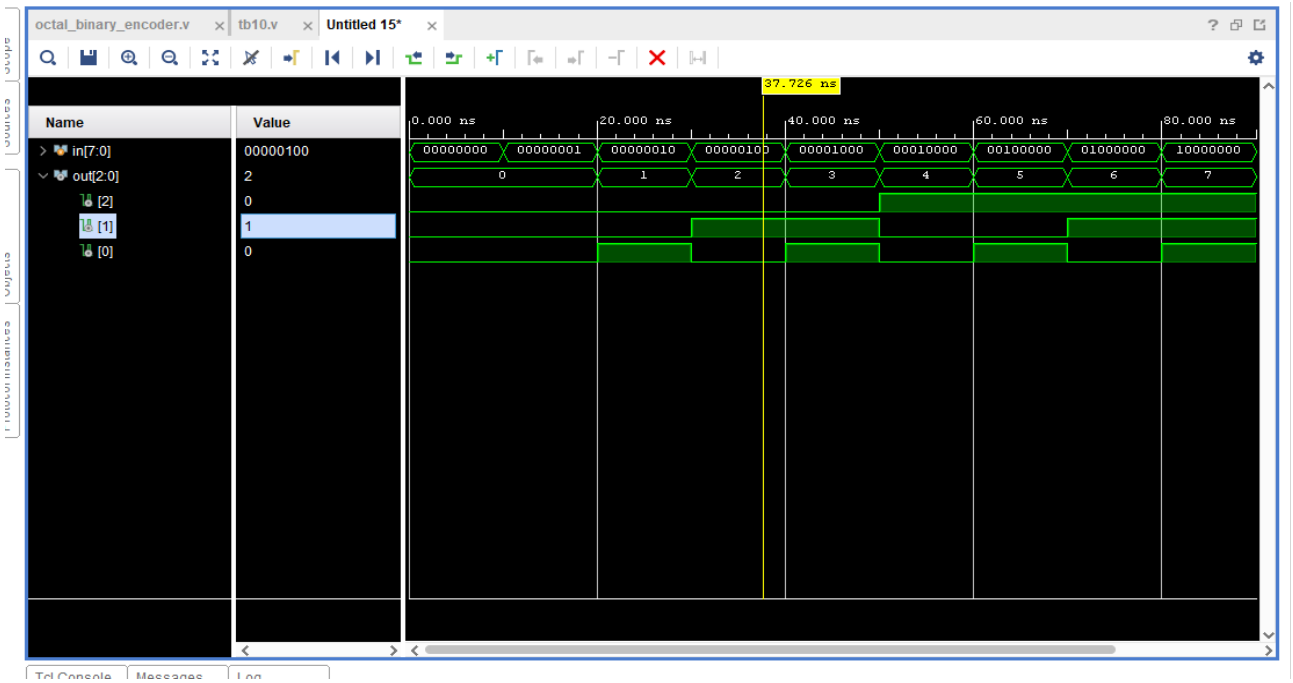
18.Octal-Binary Encoder

Verilog Code:

```
module octal_binary_encoder(in,out);
input[7:0]in;
output[2:0] out;
assign out[2]=in[4]|in[5]|in[6]|in[7];
assign out[1]=in[2]|in[3]|in[6]|in[7];
assign out[0]=in[1]|in[3]|in[5]|in[7];
endmodule
```

TestBench:

```
module tb10();
reg [7:0] in;
wire [2:0] out;
octal_binary_encoder uut(in,out);
initial
begin
in=8'b00000000; #10
in=8'b00000001; #10
in=8'b00000010; #10
in=8'b00000100; #10
in=8'b00001000; #10
in=8'b00010000; #10
in=8'b00100000; #10
in=8'b01000000; #10
in=8'b10000000; #10
$finish;
end
endmodule
```



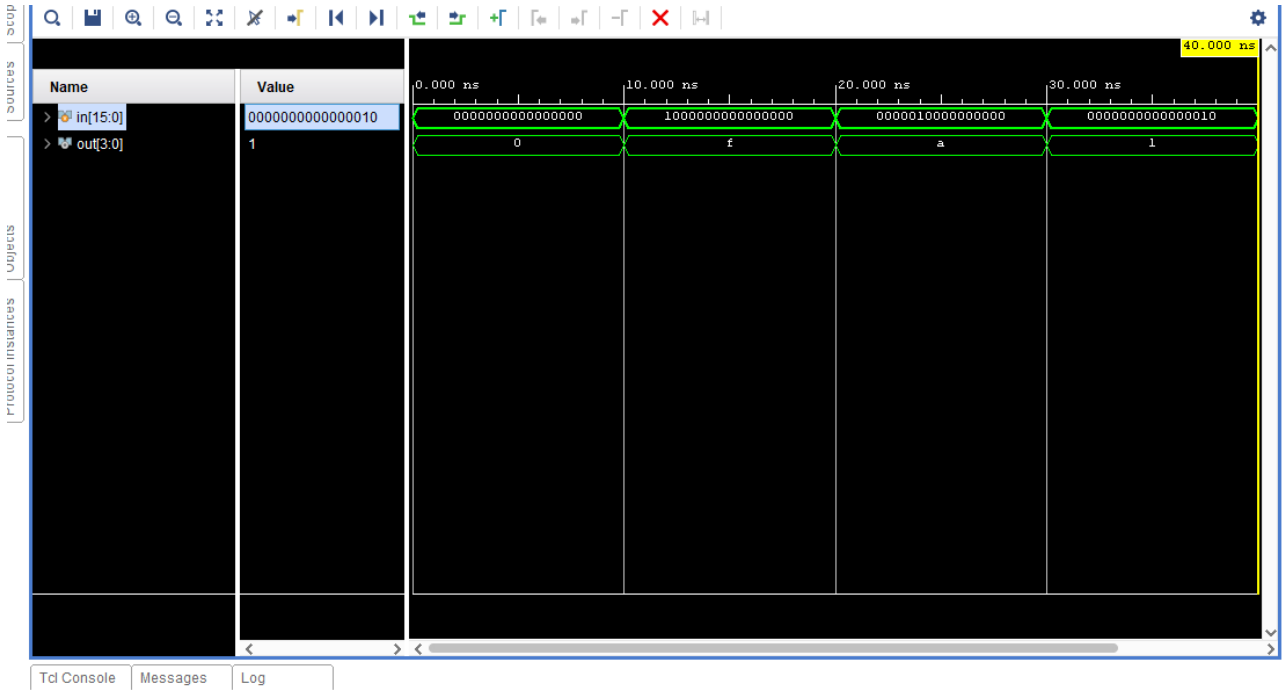
19.Hexadecimal-Binary Encoder

Verilog Code:

```
module hexadecimal_binary(in,out );
input [15:0] in;
output[3:0] out;
assign out[3]=in[8]|in[9]|in[10]|in[11]|in[12]|in[13]|in[14]|in[15];
assign out[2]=in[4]|in[5]|in[6]|in[7]|in[12]|in[13]|in[14]|in[15];
assign out[1]=in[2]|in[3]|in[6]|in[7]|in[10]|in[11]|in[14]|in[15];
assign out[0]=in[1]|in[3]|in[5]|in[7]|in[9]|in[11]|in[13]|in[15];
endmodule
```

TestBench:

```
module tb11();
reg [15:0] in;
wire [3:0]out;
hexadecimal_binary uut(in,out);
initial
begin
in=16'h0000; #10
in=16'h8000; #10
in=16'h0400; #10
in=16'h0002; #10
$finish;
end
endmodule
```



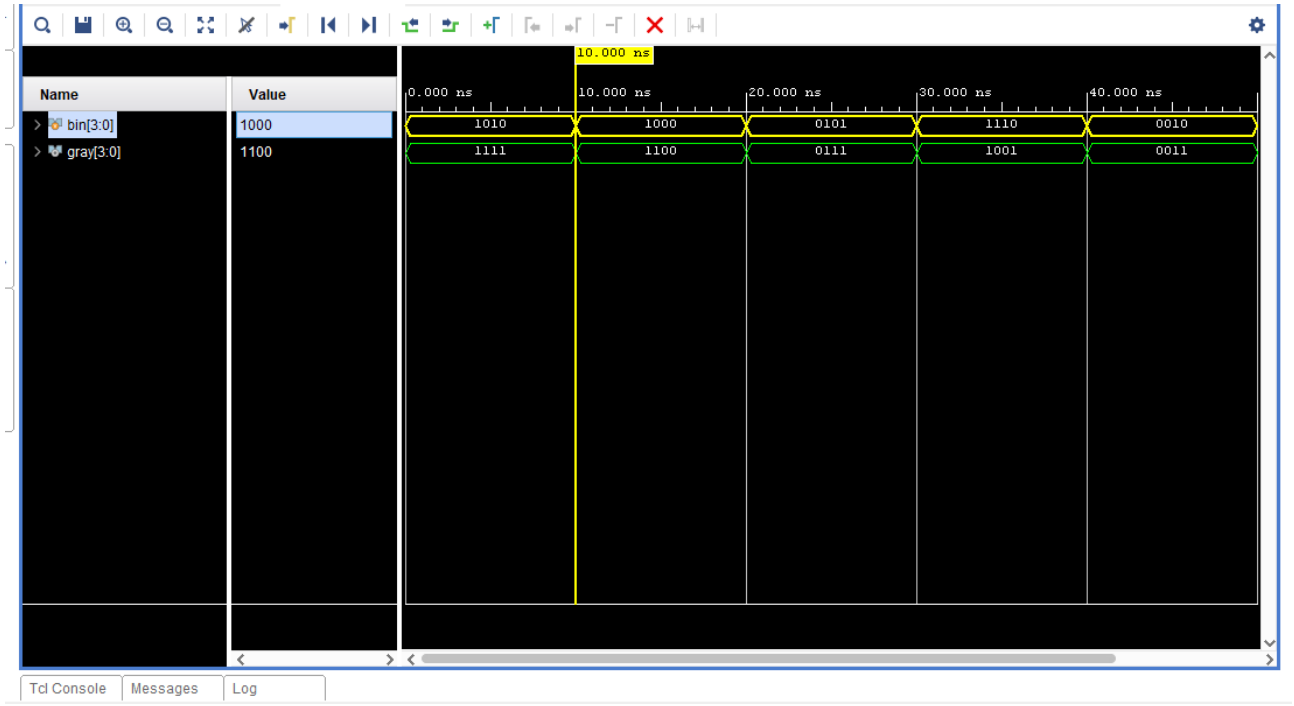
20.Binary-Gray Converter

Verilog Code:

```
module binary_gray_conv(bin,gray);
input[3:0] bin;
output[3:0]gray;
assign gray[3]=bin[3];
assign gray[2]=bin[3]^bin[2];
assign gray[1]=bin[2]^bin[1];
assign gray[0]=bin[1]^bin[0];
endmodule
```

TestBench:

```
module tb1();
reg[3:0]bin;
wire[3:0]gray;
binary_gray_conv uut(bin,gray);
initial
begin
bin=4'b1010; #10
bin=4'b1000; #10
bin=4'b0101; #10
bin=4'b1110; #10
bin=4'b0010; #10
$finish;
end
endmodule
```



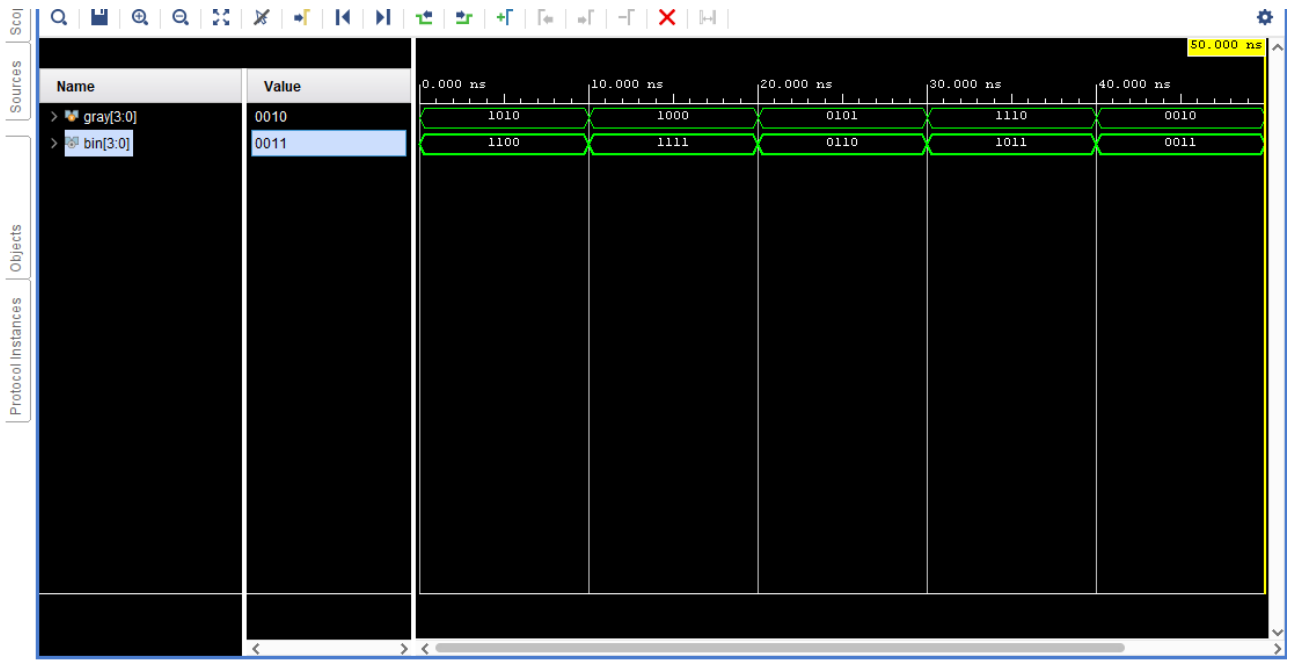
21.Gray-Binary Converter

Verilog Code:

```
module gray_binary_conv(gray,bin);
input[3:0] gray;
output[3:0] bin;
assign bin[3]= gray[3];
assign bin[2]=gray[2]^bin[3];
assign bin[1]=gray[1]^bin[2];
assign bin[0]=gray[0]^bin[1];
endmodule
```

TestBench:

```
module tb2( );
reg[3:0]gray;
wire[3:0]bin;
gray_binary_conv uut(gray,bin);
initial
begin
gray=4'b1010; #10
gray=4'b1000; #10
gray=4'b0101; #10
gray=4'b1110; #10
gray=4'b0010; #10
$finish;
end
endmodule
```



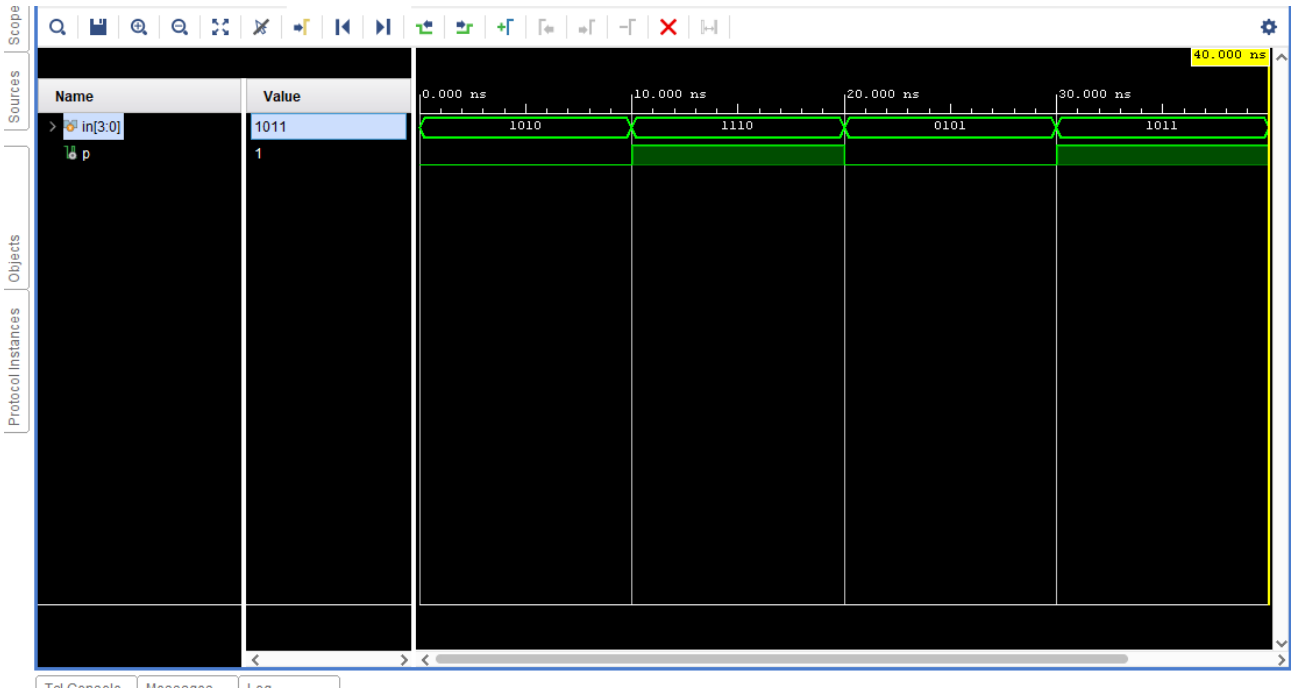
22.Even Parity Generator

Verilog Code:

```
module even_parity_gen(in,p  );
input [3:0]in;
output p;
assign p=in[3]^in[2]^in[1]^in[0];
endmodule
```

TestBench:

```
module tb3();
reg[3:0]in;
wire p;
even_parity_gen uut(.p(p),.in(in));
initial
begin
in=4'b1010; #10
in=4'b1110; #10
in=4'b0101; #10
in=4'b1011; #10
$finish;
end
endmodule
```



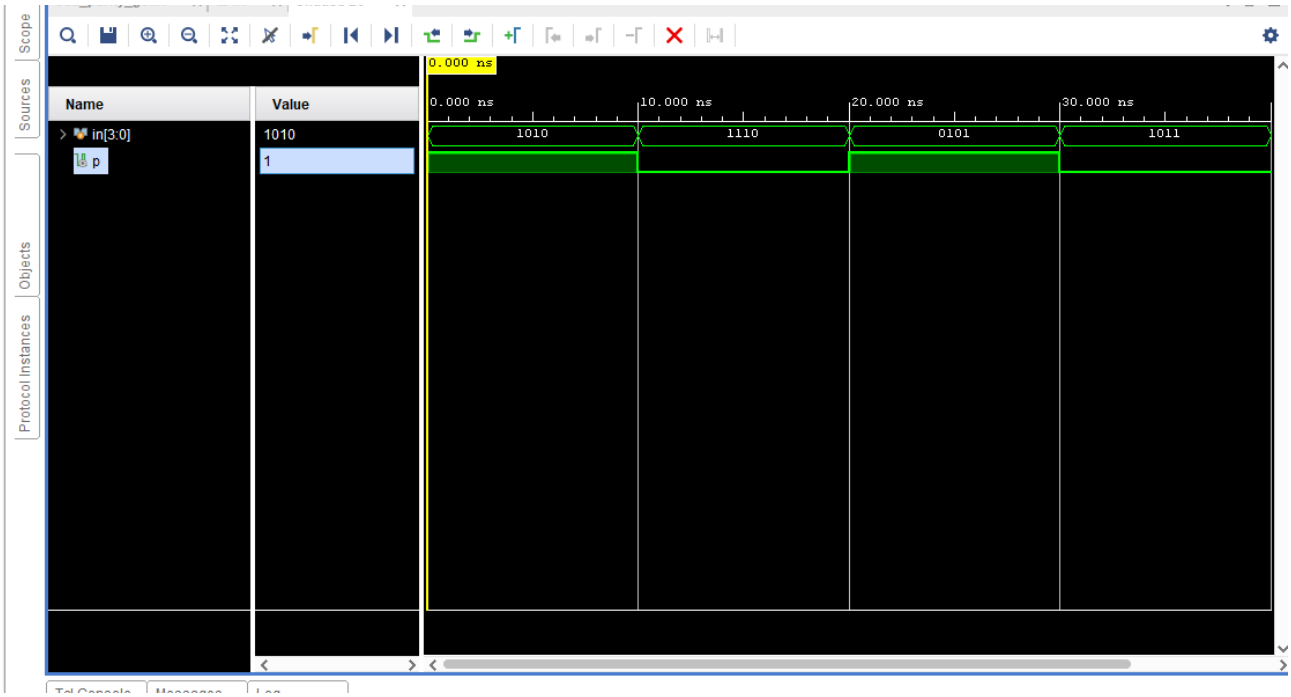
23.Odd Parity Generator

Verilog Code:

```
module odd_parity_gen(in,p);
input[3:0]in;
output p;
assign p=~(in[3]^in[2]^in[1])^in[0];
endmodule
```

TestBench:

```
module tb4();
reg[3:0]in;
wire p;
odd_parity_gen uut(.p(p),.in(in));
initial
begin
in=4'b1010; #10
in=4'b1110; #10
in=4'b0101; #10
in=4'b1011; #10
$finish;
end
endmodule
```



24.Even Parity Checker

Verilog Code:

```
module even_parity_check(in,p,error );
input[3:0]in;
input p;
output error;
assign error=in[3]^in[2]^in[1]^in[0]^p;
endmodule
```

TestBench:

```
module tb5();

reg[3:0]in;

reg p;

wire error;

even_parity_check uut(in,p,error);

initial

begin

in=4'b0101; p=1'b0; #10

in=4'b1101; p=1'b1; #10

in=4'b0111; p=1'b0; #10

in=4'b1001; p=1'b1; #10

$finish;

end

endmodule
```

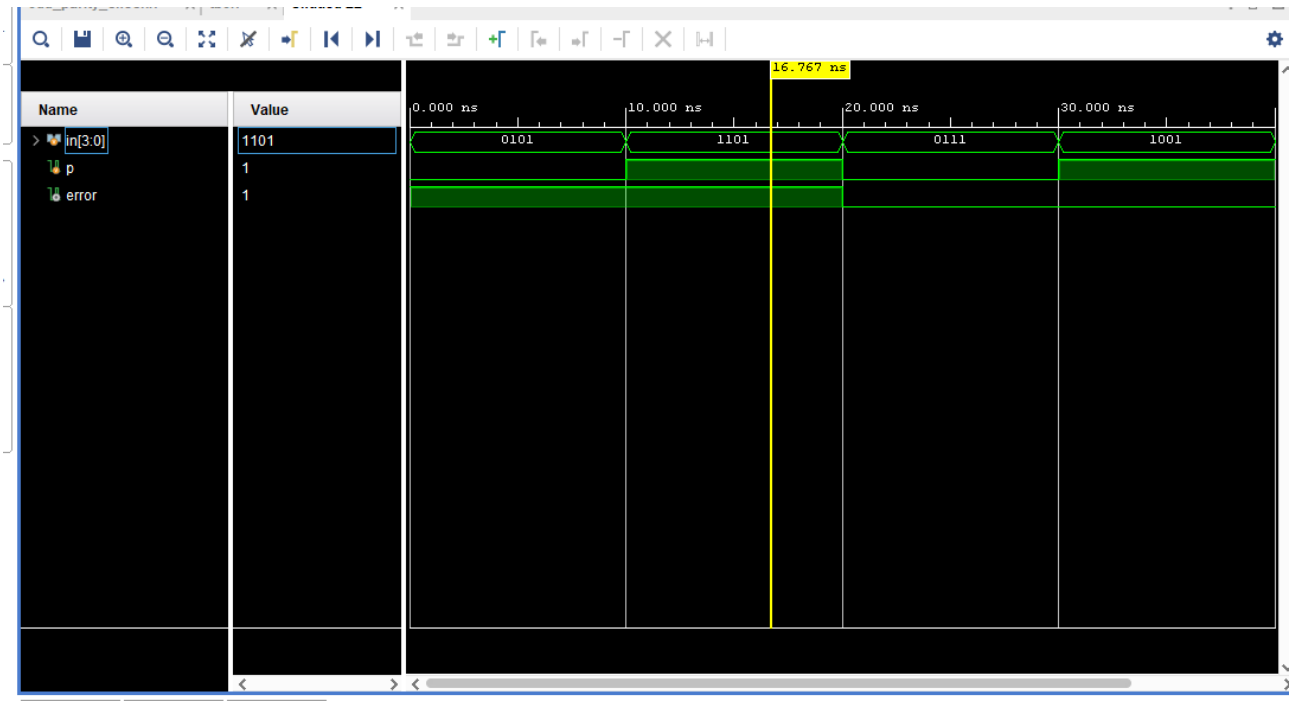
25.Odd Parity Checker

Verilog Code:

```
module odd_parity_check(in,p,error );
input[3:0]in;
input p;
output error;
assign error=~(in[3]^in[2]^in[1]^in[0])^p;
endmodule
```

TestBench:

```
module tb6();
reg[3:0]in;
reg p;
wire error;
odd_parity_check uut(in,p,error);
initial
begin
in=4'b0101; p=1'b0; #10
in=4'b1101; p=1'b1; #10
in=4'b0111; p=1'b0; #10
in=4'b1001; p=1'b1; #10
$finish;
end
endmodule
```



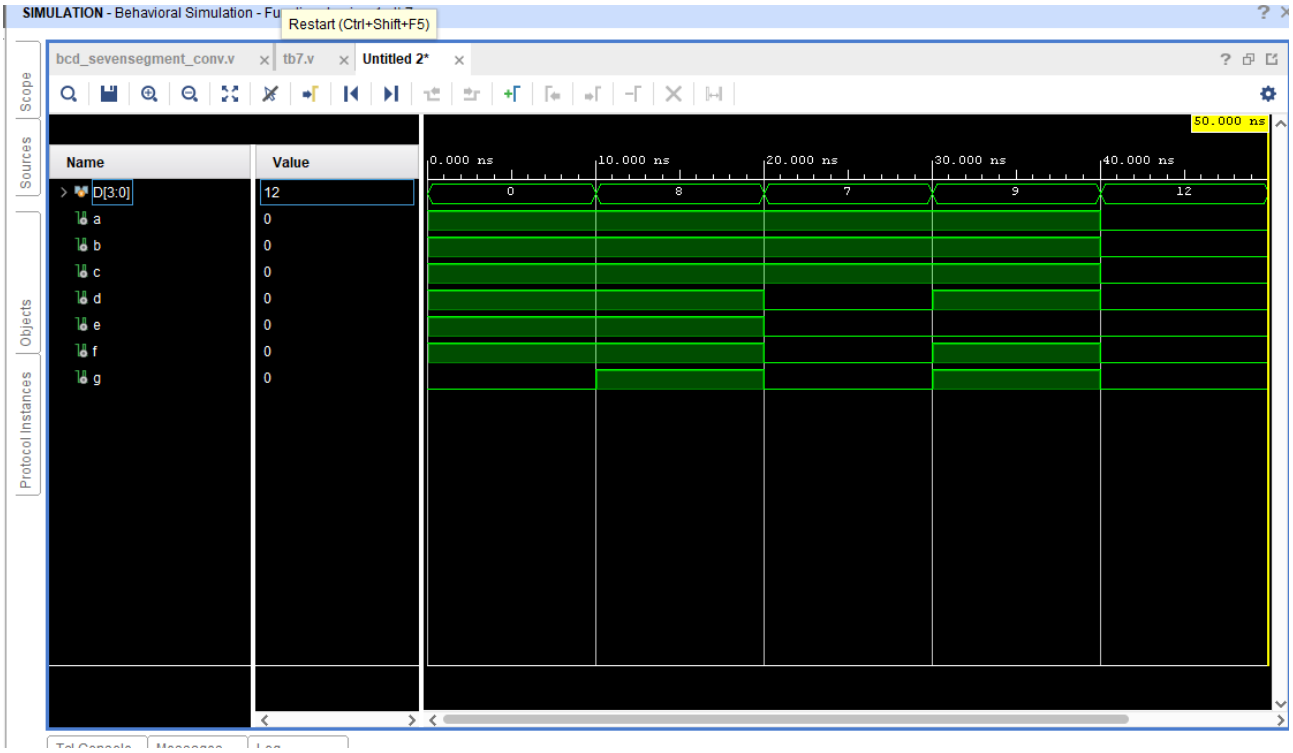
26.Binary-Seven Segment Display Converter

Verilog Code:

```
module bcd_sevensegment_conv(D,a,b,c,d,e,f,g );
input[3:0]D;
output reg a,b,c,d,e,f,g;
  always @(*)
  begin
  case(D)
          4'b0000: {a,b,c,d,e,f,g} = 7'b1111110;
          4'b0001: {a,b,c,d,e,f,g} = 7'b0110000;
          4'b0010: {a,b,c,d,e,f,g} = 7'b1101101;
          4'b0011: {a,b,c,d,e,f,g} = 7'b1111001;
          4'b0100: {a,b,c,d,e,f,g} = 7'b0110011;
          4'b0101: {a,b,c,d,e,f,g} = 7'b1011011;
          4'b0110: {a,b,c,d,e,f,g} = 7'b1011111;
          4'b0111: {a,b,c,d,e,f,g} = 7'b1110000;
          4'b1000: {a,b,c,d,e,f,g} = 7'b1111111;
          4'b1001: {a,b,c,d,e,f,g} = 7'b1111011;
          default: {a,b,c,d,e,f,g} = 7'b0000000;
       endcase
    end
endmodule
```

TestBench:

```
module tb7();
reg[3:0]D;
wire a,b,c,d,e,f,g;
bcd_sevensegment_conv uut(D,a,b,c,d,e,f,g);
initial
begin
D=4'b0000; #10
D=4'b1000; #10
D=4'b0111; #10
D=4'b1001; #10
D=4'b1100; #10
$finish;
end
endmodule
```



27.Carry Look Ahead Adder

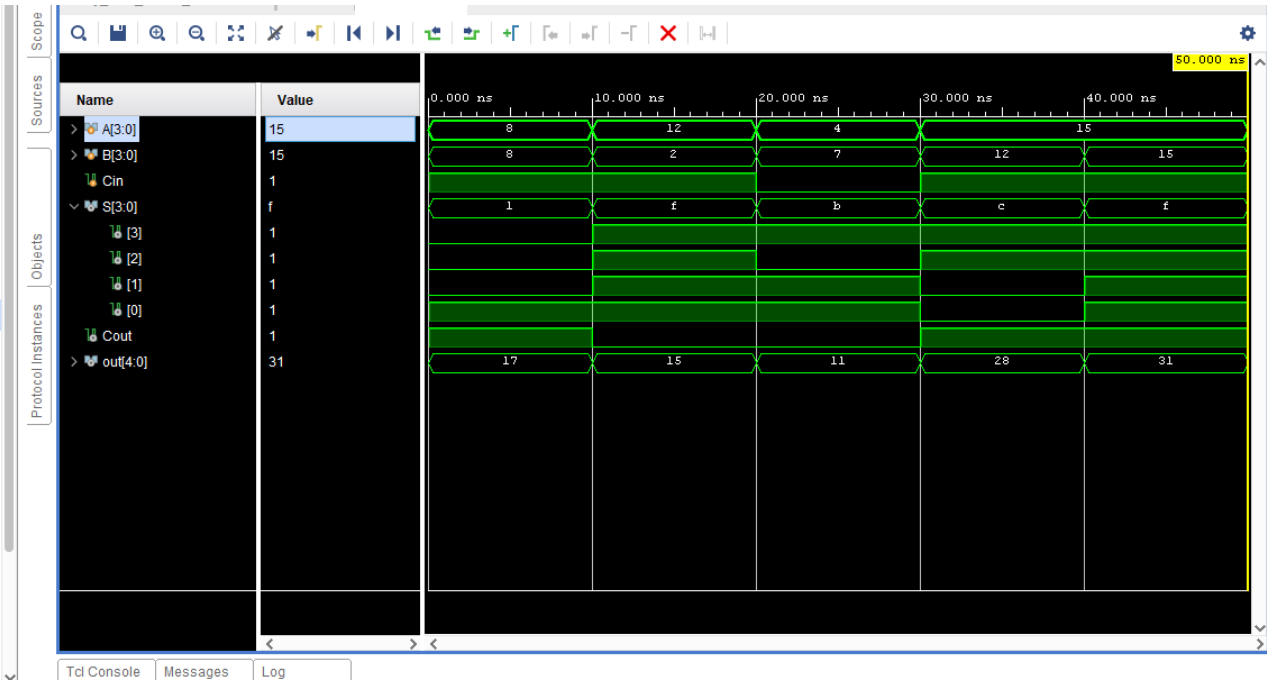
Verilog Code:

```
module carry_look_ahead_adder(
    input [3:0] A, B,
    input Cin,
    output [3:0] S,
    output Cout,
    output [4:0]out);
    wire G[3:0], P[3:0];
    wire C[4:0];

    assign G[0] = A[0] & B[0];
    assign G[1] = A[1] & B[1];
    assign G[2] = A[2] & B[2];
    assign G[3] = A[3] & B[3];

    assign P[0] = A[0] | B[0];
    assign P[1] = A[1] | B[1];
    assign P[2] = A[2] | B[2];
    assign P[3] = A[3] | B[3];

    assign C[0] = Cin;
    assign C[1] = G[0] | (P[0] & Cin);
    assign C[2] = G[1] | (P[1] & (G[0] | (P[0] & Cin)));
    assign C[3] = G[2] | (P[2] & (G[1] | (P[1] & (G[0] | (P[0] & Cin)))));
```

```
    assign Cout = G[3] | (P[3] & (G[2] | (P[2] & (G[1] | (P[1] & (G[0] | (P[0] & Cin)))))));

    assign S[0] = A[0] ^ B[0] ^ C[0];
    assign S[1] = A[1] ^ B[1] ^ C[1];
    assign S[2] = A[2] ^ B[2] ^ C[2];
    assign S[3] = A[3] ^ B[3] ^ C[3];

    assign out={Cout,S};

endmodule
```

TestBench:

```
module tb8( );
reg[3:0] A, B;
reg Cin;
wire[3:0] S;
wire Cout;
wire [4:0]out;
carry_look_ahead_adder uut(A,B,Cin,S,Cout,out);
initial
begin
A=4'b1000; B=4'b1000; Cin=1'b1; #10
A=4'b1100; B=4'b0010; Cin=1'b1; #10
A=4'b0100; B=4'b0111; Cin=1'b0; #10
A=4'b1111; B=4'b1100; Cin=1'b1; #10
A=4'b1111; B=4'b1111; Cin=1'b1; #10
$finish;
end
endmodule
```



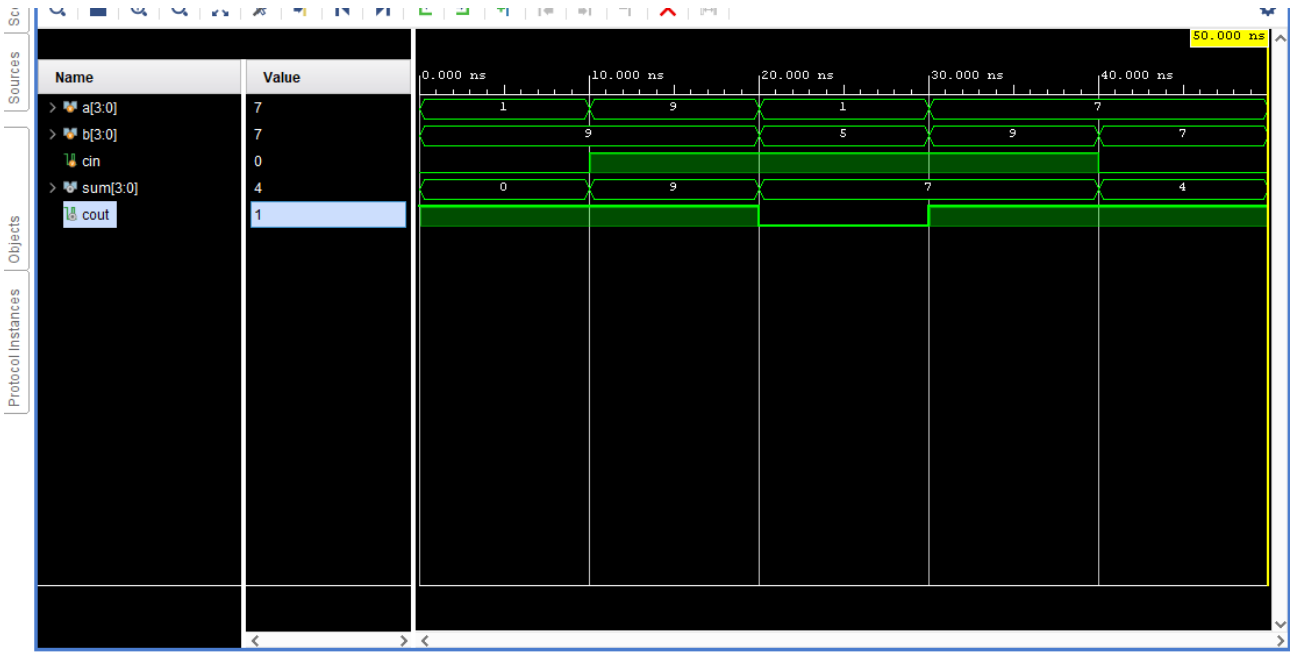
28.BCD Adder

Verilog Code:

```
module bcd_addition(a,b,cin,sum,cout  );
input[3:0]a,b;
input cin;
output[3:0]sum;
output cout;
reg cout_temp;reg[4:0]sum_temp;
always @(*)
begin
sum_temp=a+b+cin;
if(sum_temp>9)
begin
sum_temp=sum_temp+6;
cout_temp=1;
end
else
sum_temp=sum_temp[3:0];
cout_temp=sum_temp[4];
end
assign sum=sum_temp;
assign cout=cout_temp;
endmodule
```

TestBench:

```
module tb9( );
reg [3:0]a,b;
reg cin;wire[3:0]sum;wire cout;
bcd_addition uut(a,b,cin,sum,cout);
initial
begin
a=4'b0001; b=4'b1001; cin=0; #10;
a=4'b1001; b=4'b1001; cin=1; #10;
a=4'b0001; b=4'b0101; cin=1; #10;
a=4'b0111; b=4'b1001; cin=1; #10;
a=4'b0111; b=4'b0111; cin=0; #10;
$finish;
end
endmodule
```



29.BCD-Excess_3 Converter

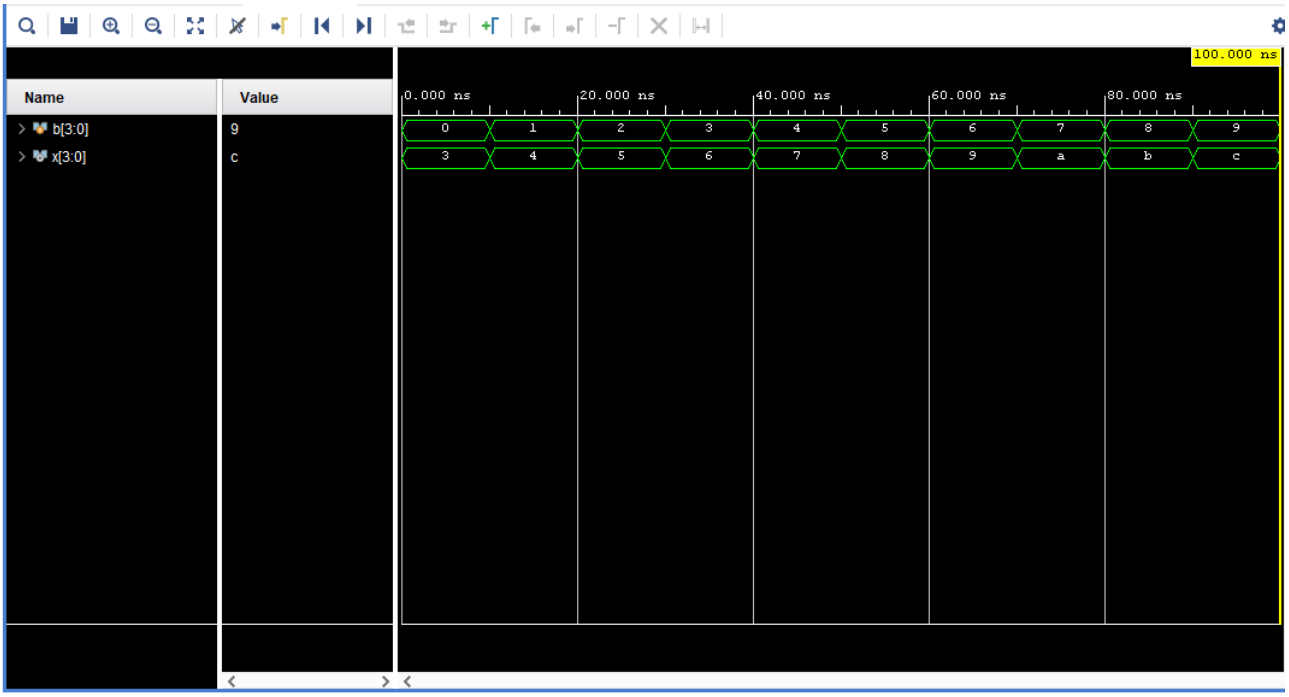
Verilog Code:

```
module bcd_x3_conv(b,x);
input[3:0]b;
output[3:0]x;
assign x[3]=b[3]|(b[2]&b[1])|(b[2]&b[0]);
assign x[2]=(~b[2]&b[0])|(~b[2]&b[1])|(b[2]&~b[1]&~b[0]);
assign x[1]=(b[1]&b[0])|(~b[1]&~b[0]);
assign x[0]=~b[0];
endmodule
```

TestBench:

```
module tb1(  );
reg[3:0]b;
wire[3:0]x;
bcd_x3_conv uut(b,x);
initial
begin
b=4'b0000; #10
b=4'b0001; #10
b=4'b0010; #10
b=4'b0011; #10
b=4'b0100; #10
b=4'b0101; #10
b=4'b0110; #10
b=4'b0111; #10
b=4'b1000; #10
```

b=4'b1001; #10
$finish;
end
endmodule



30.Carry Save Adder

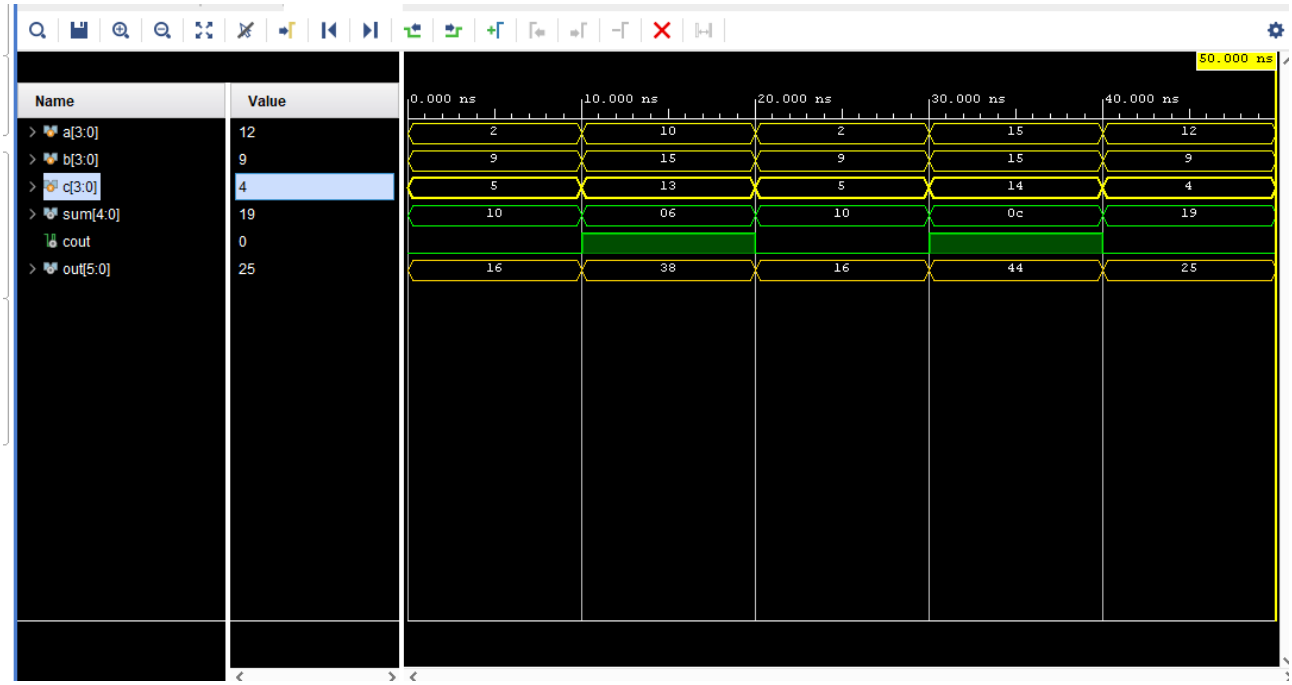
Verilog Code:

module full_addr(a,b,c,sum,carry);
input a,b,c; output sum,carry;
assign sum=a^b^c;
assign carry=(a&b)|(b&c)|(a&c);
endmodule
module carry_save_adder(a,b,c,sum,cout,out);
input[3:0]a,b,c;
output [4:0] sum;output cout;
output[5:0]out;
wire [3:0]sum_temp,cout_temp,co;
full_addr fa1(a[0],b[0],c[0],sum_temp[0],cout_temp[0]);
full_addr fa2(a[1],b[1],c[1],sum_temp[1],cout_temp[1]);
full_addr fa3(a[2],b[2],c[2],sum_temp[2],cout_temp[2]);
full_addr fa4(a[3],b[3],c[3],sum_temp[3],cout_temp[3]);
full_addr fa5(sum_temp[1],cout_temp[0],1'b0,sum[1],co[0]);
full_addr fa6(sum_temp[2],cout_temp[1],co[0],sum[2],co[1]);
full_addr fa7(sum_temp[3],cout_temp[2],co[1],sum[3],co[2]);
full_addr fa8(1'b0,cout_temp[3],co[2],sum[4],cout);
assign sum[0]=sum_temp[0];

assign out={cout,sum};
endmodule

TestBench:

module tb2();
reg[3:0]a,b,c;
wire[4:0]sum; wire cout; wire[5:0]out;
carry_save_adder uut(a,b,c,sum,cout,out);
initial
begin
a=4'b0010; b=4'b1001; c=4'b0101; #10
a=4'b1010; b=4'b1111; c=4'b1101; #10
a=4'b0010; b=4'b1001; c=4'b0101; #10
a=4'b1111; b=4'b1111; c=4'b1110; #10
a=4'b1100; b=4'b1001; c=4'b0100; #10
$finish;
end
endmodule



31.Squares of 3bit numbers

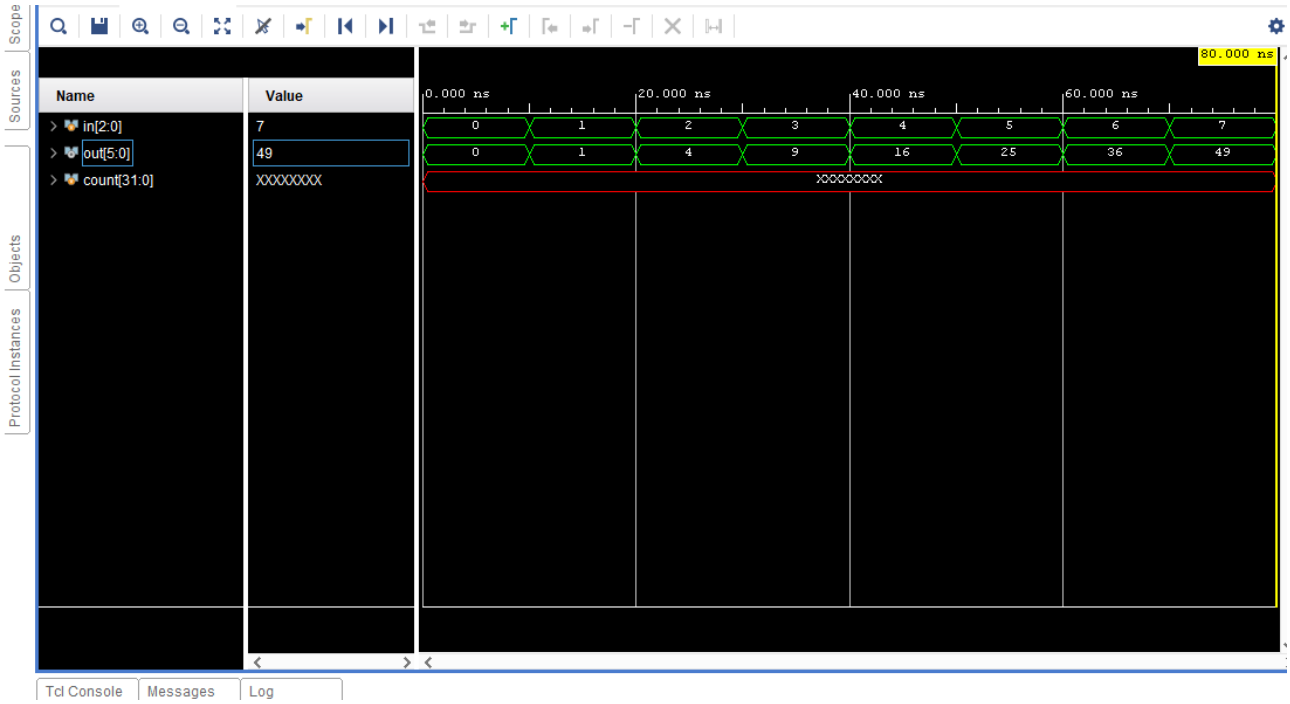
Verilog Code:

module squares_3bit(in,out);
input[2:0]in;output[5:0]out;
assign out[5]=in[2]&in[1];
assign out[4]=(in[2]&~in[1])|(in[0]&in[2]);
assign out[3]=in[0]&(in[1]^in[2]);
assign out[2]=in[1]&~in[0];

assign out[1]=1'b0;
assign out[0]=in[0];
endmodule

TestBench:

module tb3( );
reg[2:0]in;wire[5:0]out;
squares_3bit dut(in,out);
integer count;
initial
begin
in=3'b000;#10
in=3'b001;#10
in=3'b010;#10
in=3'b011;#10
in=3'b100;#10
in=3'b101;#10
in=3'b110;#10
in=3'b111;#10
$finish;
end
endmodule



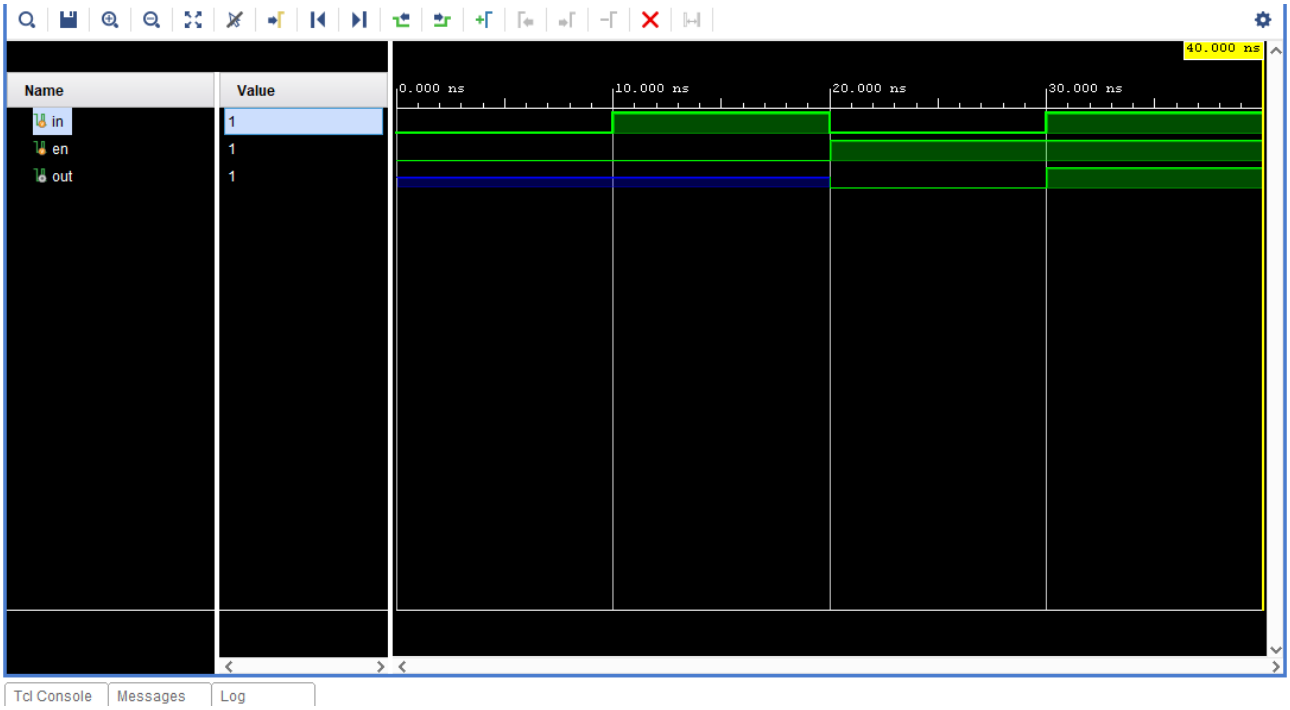
32.Tristate Buffer

Verilog Code:

```
module tristate_buffer(in,en,out);
input in,en;output out;
assign out=en?in:1'bz;
endmodule
```

TestBench:

```
module tb4();
reg in,en; wire out;
tristate_buffer uut(in,en,out);
initial
begin
in=0; en=0; #10
in=1; en=0; #10
in=0; en=1; #10
in=1; en=1; #10
$finish;
end
endmodule
```



33.RS Latch using NOR gates

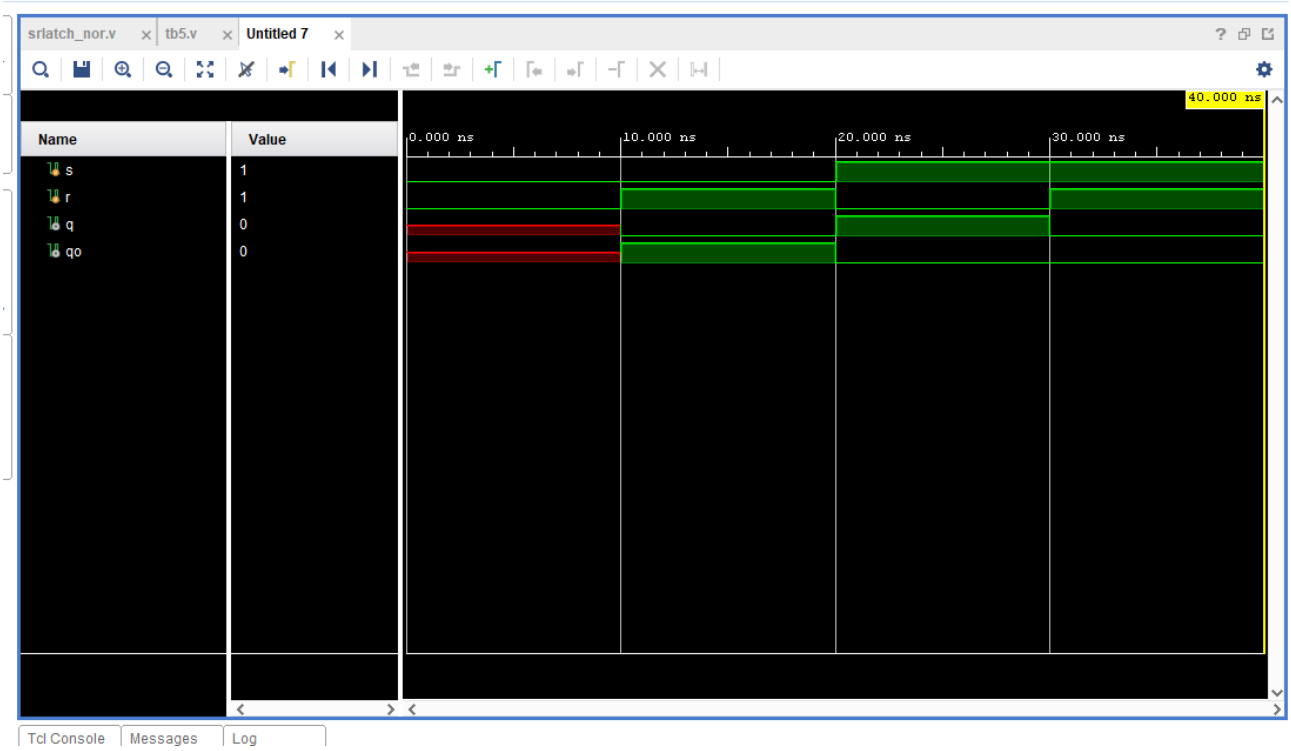
Verilog Code:

```
module srlatch_nor(s,r,q,qo);
input s,r;
```

output q,qo;
assign q=~(r|qo);
assign qo=~(s|q);
endmodule

TestBench:

module tb5();
reg s,r;
wire q, qo;
srlatch_nor uut(s,r,q,qo);
initial
begin
s=0; r=0; #10
s=0; r=1; #10
s=1; r=0; #10
s=1; r=1; #10
$finish;
end
endmodule



34.RS Latch using NAND gates

Verilog Code:

module srlatch_nand(s,r,q,qo);
input s,r;

inout q,qo;
assign q=~(r&qo);
assign qo=~(s&q);
endmodule

TestBench:

module tb6( );
reg s,r;
wire q,qo;
srlatch_nand uut(s,r,q,qo);
initial
begin
s=0; r=0; #10
s=0; r=1; #10
s=1; r=0; #10
s=1; r=1; #10
s=0; r=0; #10
s=1; r=0; #10
s=0; r=1; #10
s=1; r=1; #10
$finish;
end
endmodule

35.SR Flipflop

Verilog Code:

module sr_flipflop(clk,s,r,q,qo);
input clk,s,r;
output q,qo;

```
assign q=~(~(s&clk)&qo);
assign qo=~(~(r&clk)&q);
endmodule
```

TestBench:

```
module tb7( );
reg clk,s,r;wire q,qo;
sr_flipflop uut(clk,s,r,q,qo);
initial
begin
clk=0;
forever #5 clk=~clk;
end
initial
begin
 s=0; r=0; #10
 s=0; r=1; #10
 s=1; r=0; #10
 s=1; r=1; #10
 s=0; r=0; #10
 s=0; r=1; #10
 s=1; r=0; #10
$finish;
end
endmodule
```

36.JK Flipflop

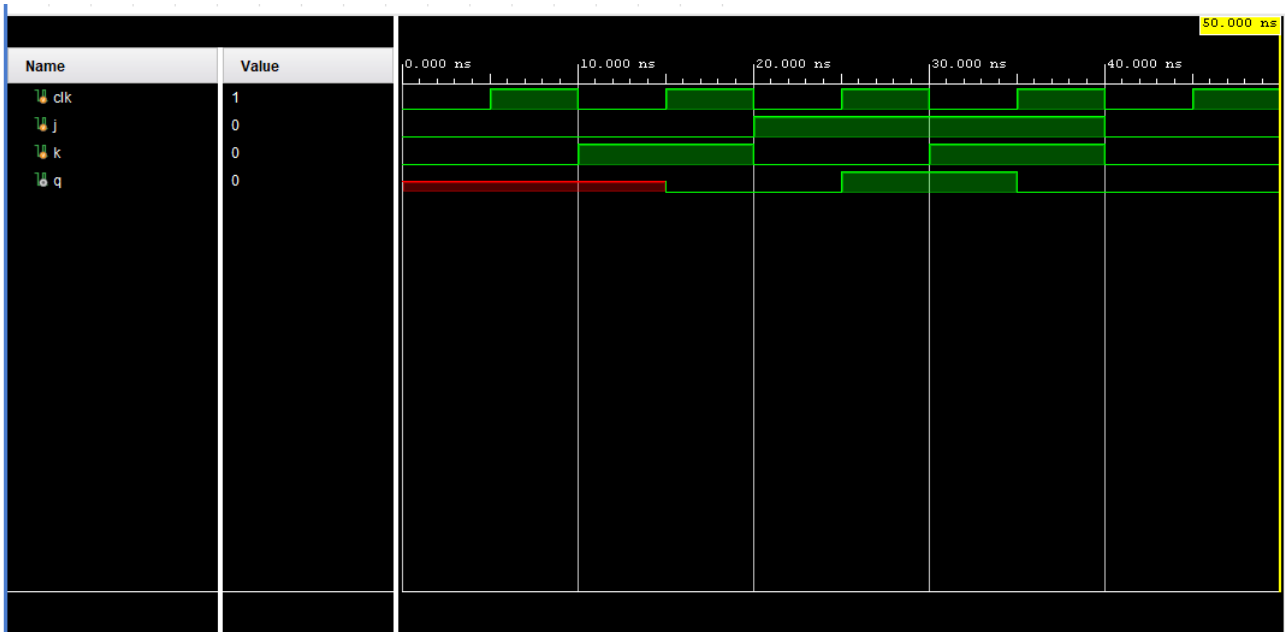
Verilog Code:

```
module jk_flipflop(j,k,clk,q );
input j,k,clk;
```

```
output reg q;
always @( posedge clk)
begin
case({j,k})
2'b00:q=q;
2'b01:q=1'b0;
2'b10:q=1'b1;
2'b11:q=~q;
endcase
end
endmodule
```

TestBench:

```
module tb8();
reg clk,j,k;
wire q;
jk_flipflop uut(j,k,clk,q);
initial
begin
clk=0;
forever #5 clk=~clk;
end
initial
begin
j=0; k=0; #10
j=0; k=1; #10
j=1; k=0; #10
j=1; k=1; #10
j=0; k=0; #10
$finish;
end
endmodule
```



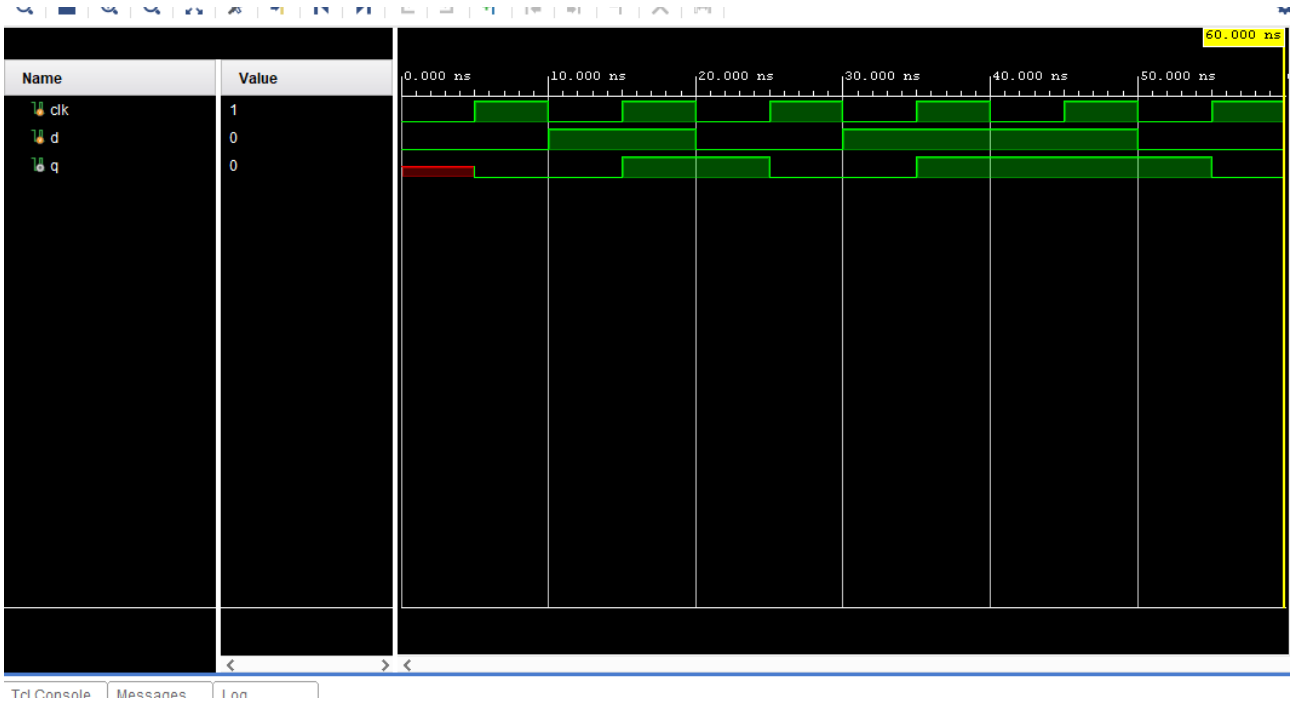
37.D Flipflop

Verilog Code:

```
module d_flipflop(d,clk,q);
input d,clk;
output reg q;
always @(posedge clk)
begin
case(d)
1'b0: q=0;
1'b1: q=1;
endcase
end
endmodule
```

TestBench:

```
module tb9();
reg clk,d;
wire q;
d_flipflop uut(d,clk,q);
initial
begin
clk=0;
forever #5 clk=~clk;
end
initial
begin
d=0;#10
d=1;#10
d=0;#10
d=1;#10
d=1;#10
d=0;#10
$finish;
end
endmodule
```



38.T Flipflop

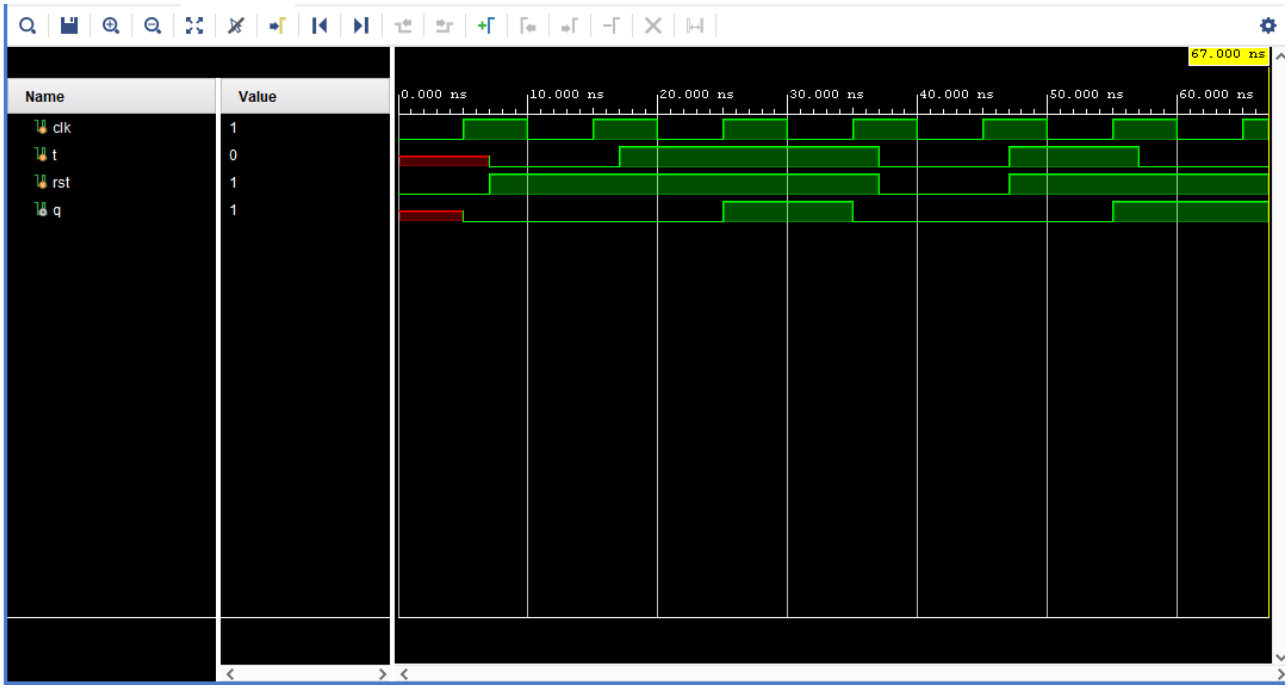
Verilog Code:

```
module t_ff(t,rst,clk,q  );
input clk,t,rst;
output reg q;
always@(posedge clk)
begin
if(!rst)
q=1'b0;
else if(t)
q=~q;
end
endmodule
```

TestBench:

```
module tb10();
reg clk,t,rst;
wire q;
t_ff uut(t,rst,clk,q);
initial
begin
clk=0;
forever #5 clk=~clk;
end
initial
begin
```

rst =0;#7
rst=1;t=0; #10
t=1; #10
t=1; #10
rst=0;t=0; #10
rst=1;t=1; #10
t=0; #10
$finish;
end
endmodule



39.D Latch

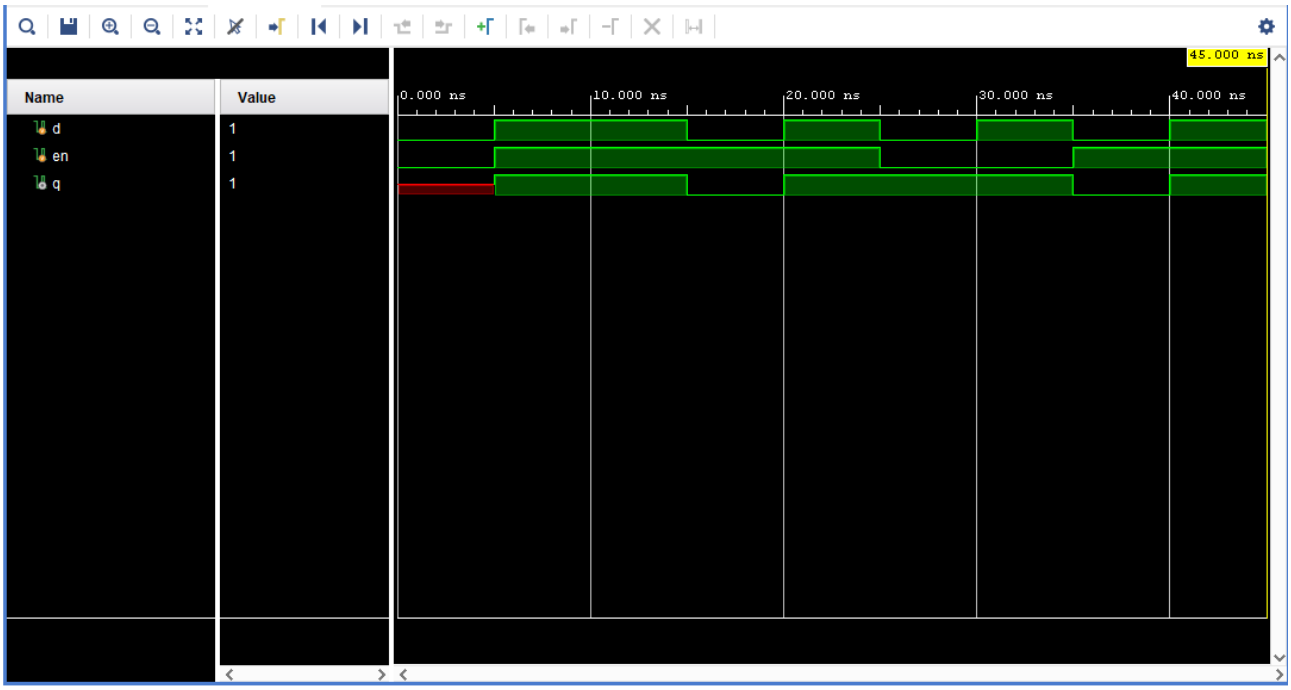
Verilog Code:

module d_latch(d,en,q);
input d,en;
output reg q;
always @(d,en)
begin
if(en)
q=d;
end
endmodule

TestBench:

module tb11( );
reg d,en;

wire q;
d_latch uut(d,en,q);
initial
begin
d=0; en=0; #5
d=1; en=1; #10
d=0; #5
d=1;#5
en=0; d=0;#5
d=1;#5
en=1; d=0; #5
d=1;#5
$finish;
end
endmodule



40.Asynchronous counter using T Flipflops

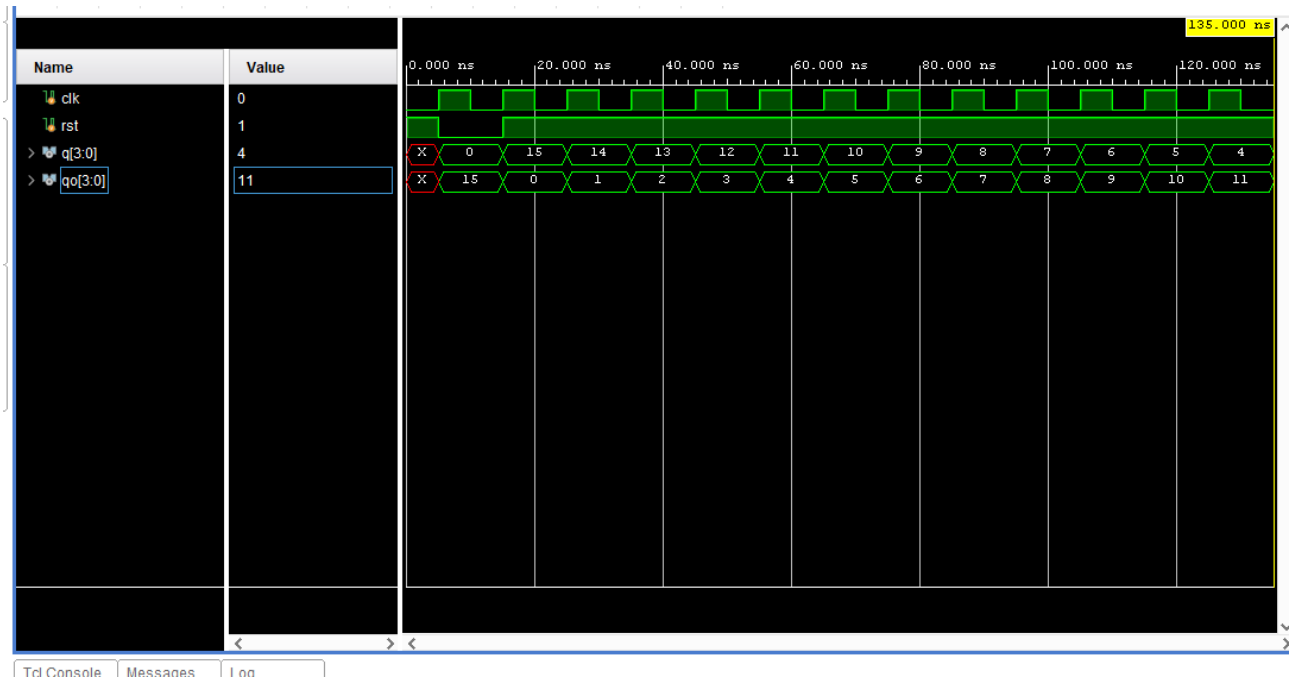
Verilog Code:

module t_ff(t,clk,rst,q);
input t,clk,rst;
output reg q;
always @(posedge clk,negedge rst)
begin
if(!rst)
q<=0;
else if(t)
q<=~q;

```
else
q<=q;
end
endmodule

module asyn_coun_t_ff(clk,rst,q,qo);
output [3:0]q;
output [3:0]qo;
input clk,rst;
assign t=1;
t_ff t1(t,clk,rst,q0);
t_ff t2(t,q0,rst,q1);
t_ff t3(t,q1,rst,q2);
t_ff t4(t,q2,rst,q3);
assign q={q3,q2,q1,q0};
assign qo=~q;
endmodule
```

TestBench:

```
module tb2( );
reg clk,rst;
wire[3:0]q,qo;
asyn_coun_t_ff uut(clk,rst,q,qo);
initial
begin
clk=0;
forever #5 clk=~clk;
end
initial
begin
rst=1; #5
rst=0;#10
rst=1;#120
$finish;
end
endmodule
```



41.Synchronous Up and Down Counter

Verilog Code:

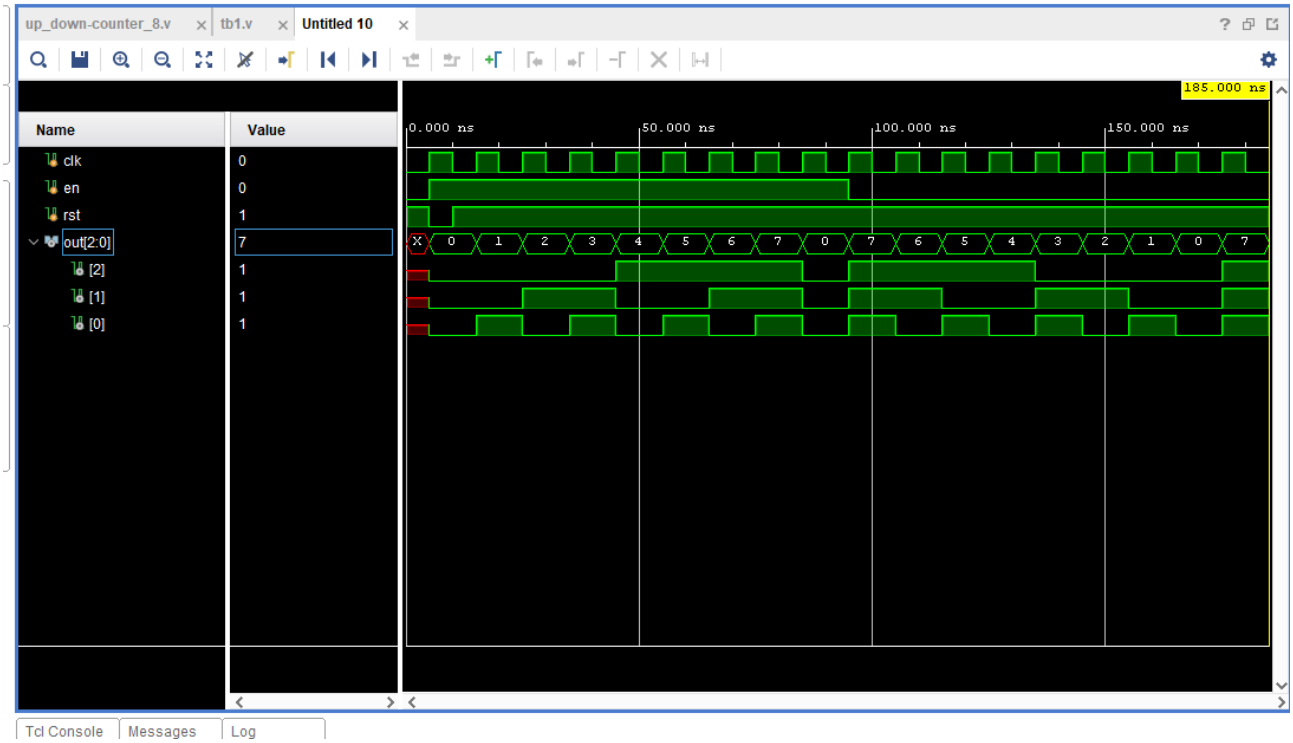
```
module up_down_counter_8(clk,rst,en,out);
input clk,rst,en;
output reg[2:0] out;
always @(posedge clk)
begin
if(!rst)
out=3'b000;
else if (en)
out=out+1;
else
out=out-1;
end
endmodule
```

TestBench:

```
module tb1();
reg clk,en,rst;
wire[2:0]out;
up_down_counter_8 uut(clk,rst,en,out);
initial
begin
clk=0;
forever #5 clk=~clk;
```

end
initial
begin
en=0;rst=1; #5
en=1;rst=0; #5

en=1;rst=1;#85
en=0;rst=1; #90
$finish;
end
endmodule



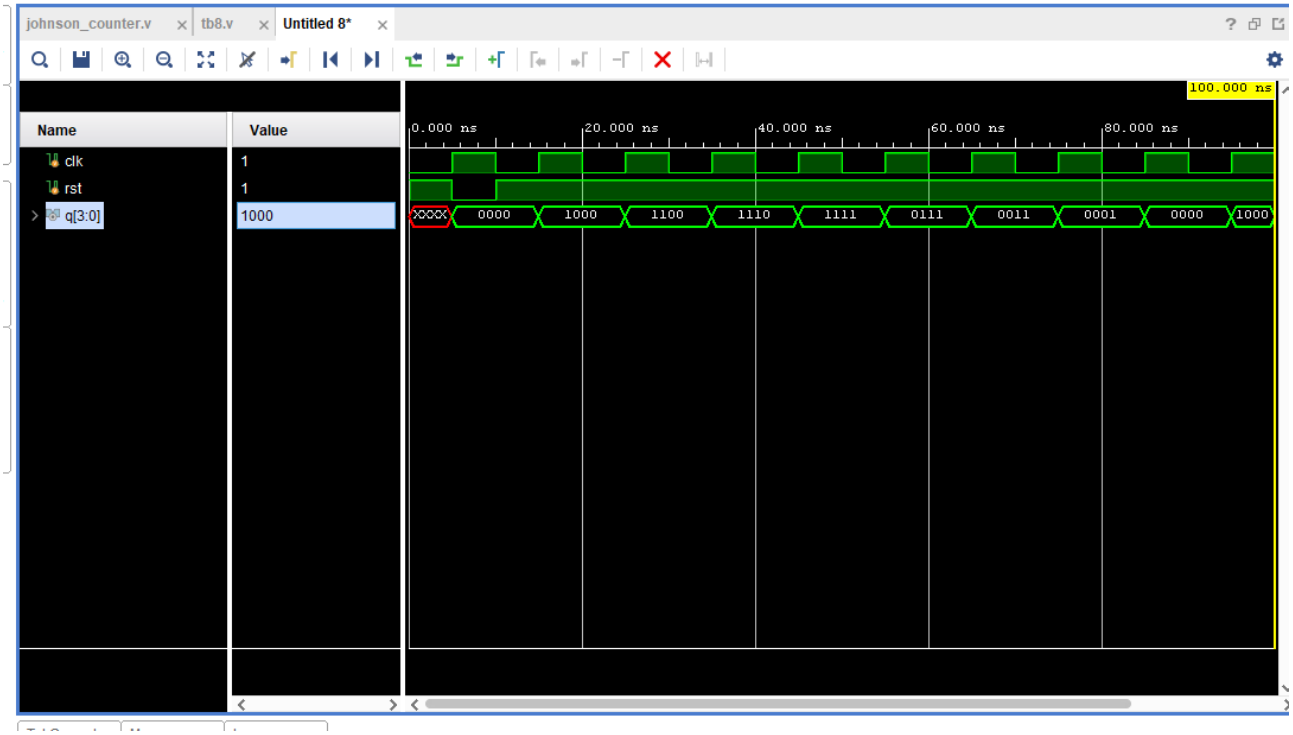
42.Johnson Counter

Verilog Code:

module johnson_counter(clk,rst,q);
input clk,rst;
output reg[3:0]q;
always@(posedge clk)
begin
if(!rst)
q<=4'b0000;
else
q<={~q[0],q[3:1]};
end
endmodule

TestBench:

```
module tb8();
reg clk,rst;
wire[3:0]q;
johnson_counter uut(clk,rst,q);
initial
begin
clk=0;
forever #5 clk=~clk;
end
initial
begin
rst=1; #5
rst=0;#5
rst=1; #90
$finish;
end
endmodule
```



43.Ring Counter

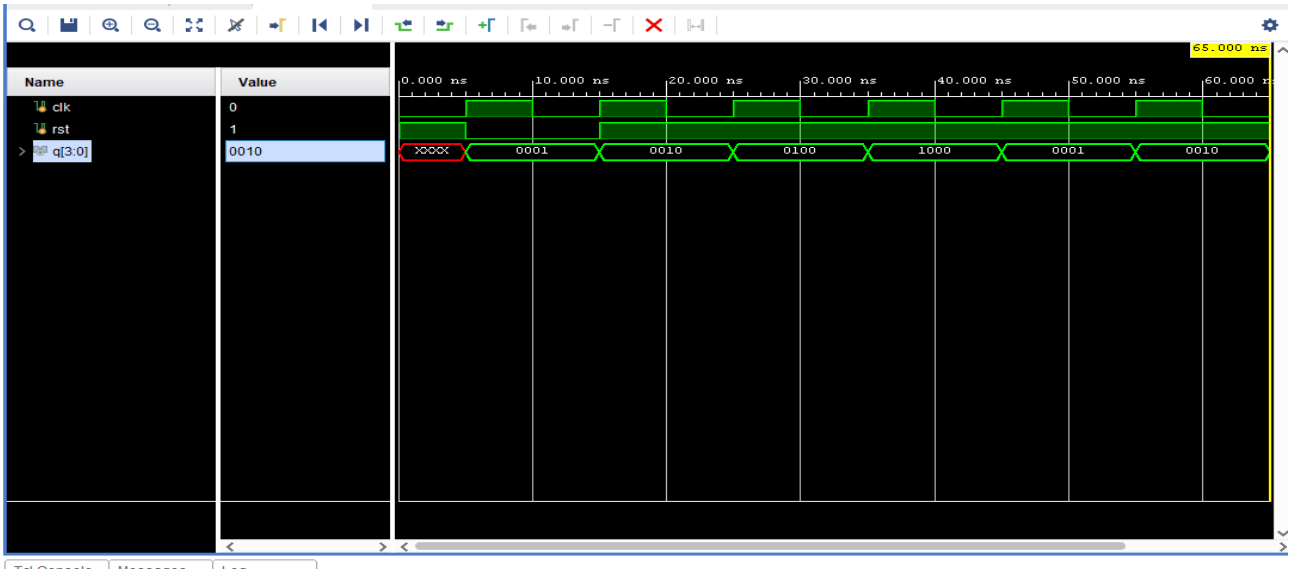
Verilog Code:

```
module ring_coun_d_ff(clk,rst,q);
input clk,rst;
output reg[3:0]q;
always@(posedge clk)
```

```
begin
if(!rst)
q=4'b0001;
else
q<={q[2:0], q[3]};
end
endmodule
```

TestBench:

```
module tb3( );
reg clk,rst;
wire [3:0]q;
ring_coun_d_ff uut(clk,rst,q);
initial
begin
clk=0;
forever #5 clk=~clk;
end
initial
begin
rst=1;#5
rst=0;#10
rst=1; #50
$finish;
end
endmodule
```



44.Serial In Serial Out Shift Register(SISO)

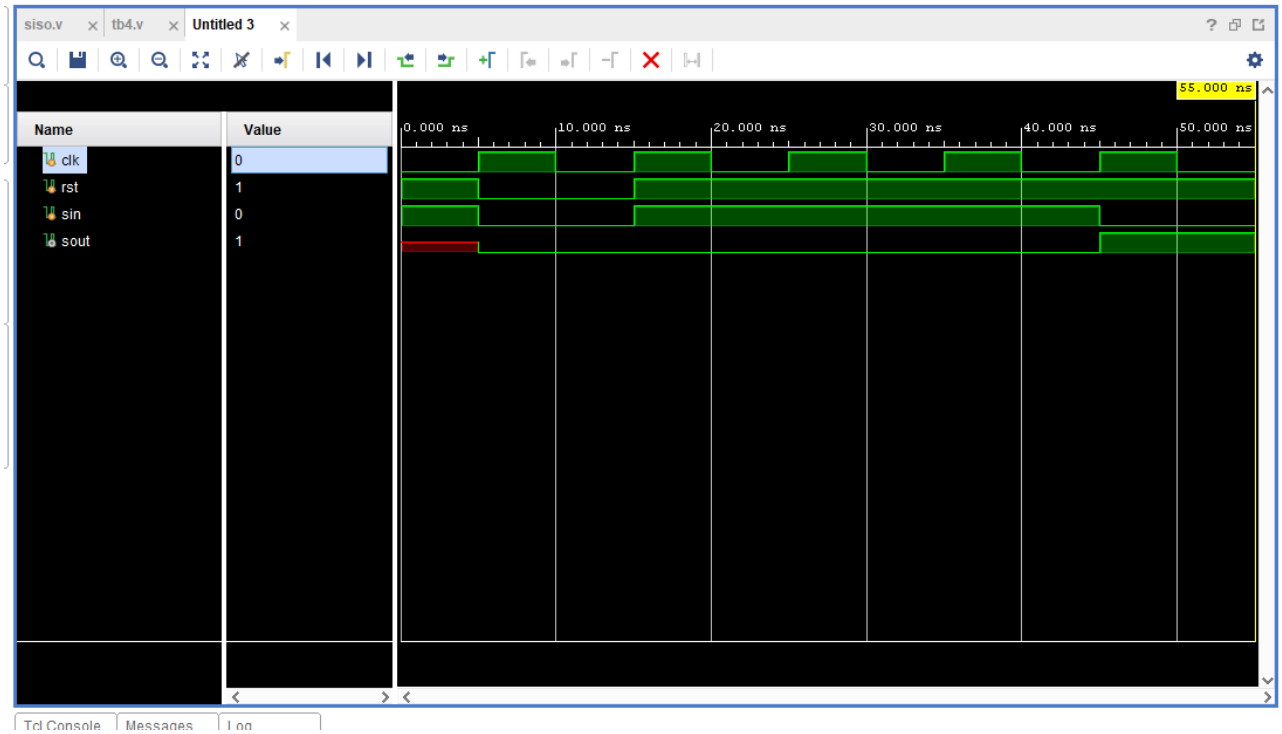
Verilog Code:

```
module siso(clk,rst,sin,sout );
input clk,sin,rst;
```

```
output sout;
reg[3:0]q;
always @(posedge clk)
begin
if(!rst)
q<=4'b0000;
else
q<={q[2:0],sin};
end
assign sout=q[3];
endmodule
```

TestBench:

```
module tb4();
reg clk,rst,sin;
wire sout;
siso uut(clk,rst,sin,sout);
initial
begin
clk=0;
forever #5 clk=~clk;
end
initial
begin
rst=1; sin=1; #5
rst=0; sin=0; #10
rst=1; sin=1; #10
sin=1; #10
sin=1; #10
sin=0; #10
$finish;
end
endmodule
```



45.Serial In Parllel Out Shift Register(SIPO)

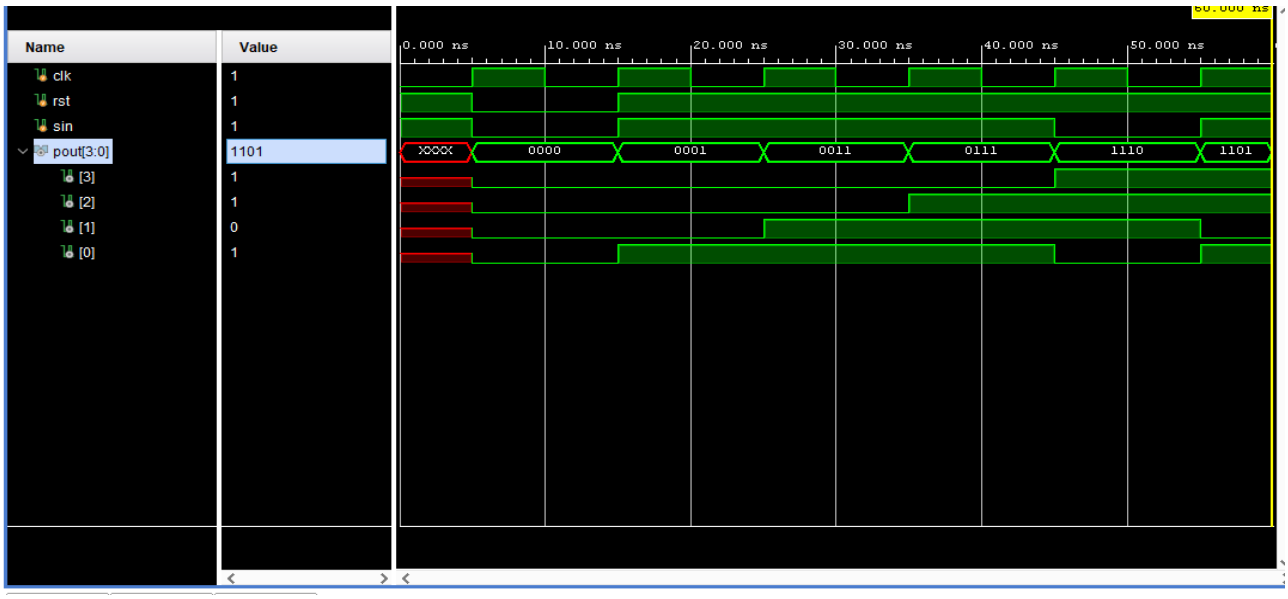
Verilog Code:

```
module sipo(clk,rst,sin,pout );
input clk,sin,rst;
output reg[3:0] pout;
always @(posedge clk)
begin
if(!rst)
pout<=4'b0000;
else
pout<={pout[2:0],sin};
end
endmodule
```

TestBench:

```
module tb5();
reg clk,rst,sin;
wire[3:0] pout;
sipo uut(clk,rst,sin,pout);
initial
begin
clk=0;
forever #5 clk=~clk;
```

end
initial
begin
rst=1; sin=1; #5
rst=0; sin=0; #10
rst=1; sin=1; #10
sin=1; #10
sin=1; #10
sin=0; #10
sin=1; #5
$finish;
end
endmodule



46.Parllel In Serial Out Shift Register(PISO)

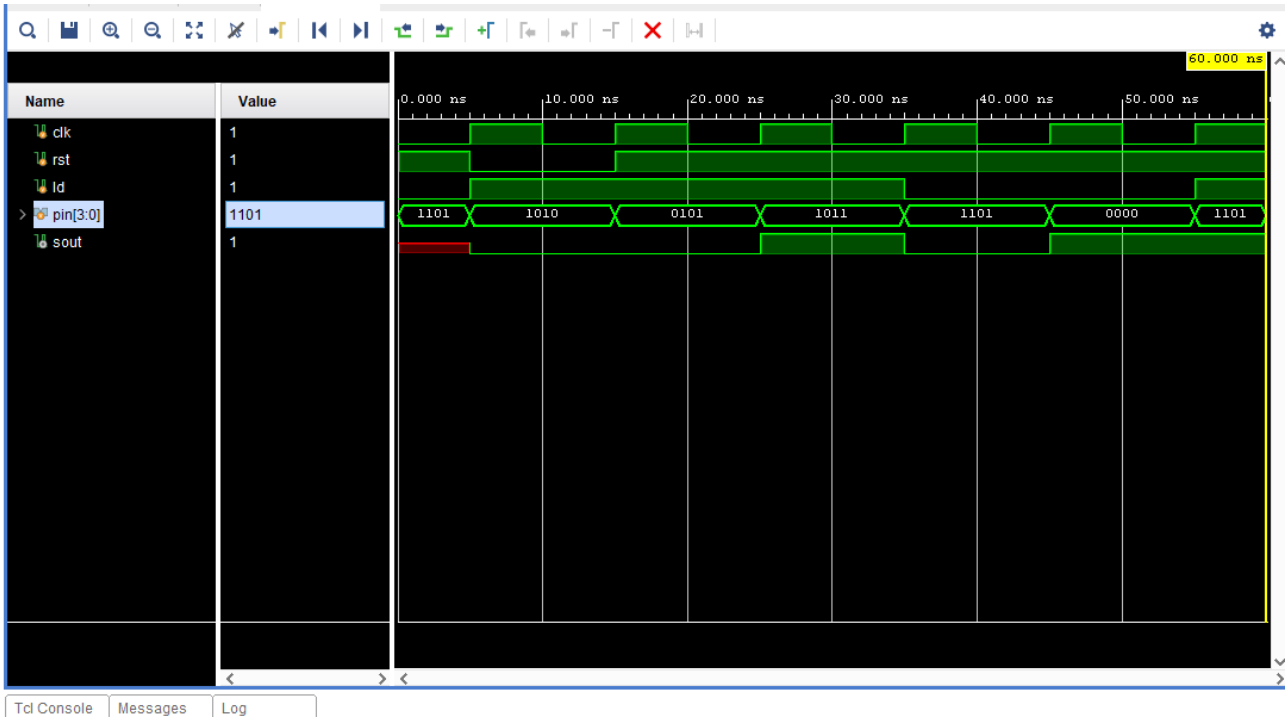
Verilog Code:

module piso(clk,rst,ld,pin,sout);
input clk,rst,ld;
input[3:0] pin;
output sout;
reg[3:0]q;
always @(posedge clk)
begin
if(!rst)
q<=4'b0000;
else if(ld)
q<=pin;
else
q<=(q<<1);

```
end
assign sout=q[3];
endmodule
```

TestBench:

```
module tb6();
reg clk,rst,ld;
reg [3:0]pin;
wire sout;
piso uut(clk,rst,ld,pin,sout);
initial
begin
clk=0;
forever #5 clk=~clk;
end
initial
begin
rst=1;ld=0; pin=4'b1101; #5
rst=0;ld=1; pin=4'b1010; #10
rst=1; pin=4'b0101; #10
pin=4'b1011; #10
pin=4'b1101;ld=0; #10
pin=4'b0000; #10
pin=4'b1101;ld=1; #5
$finish;
end
endmodule
```



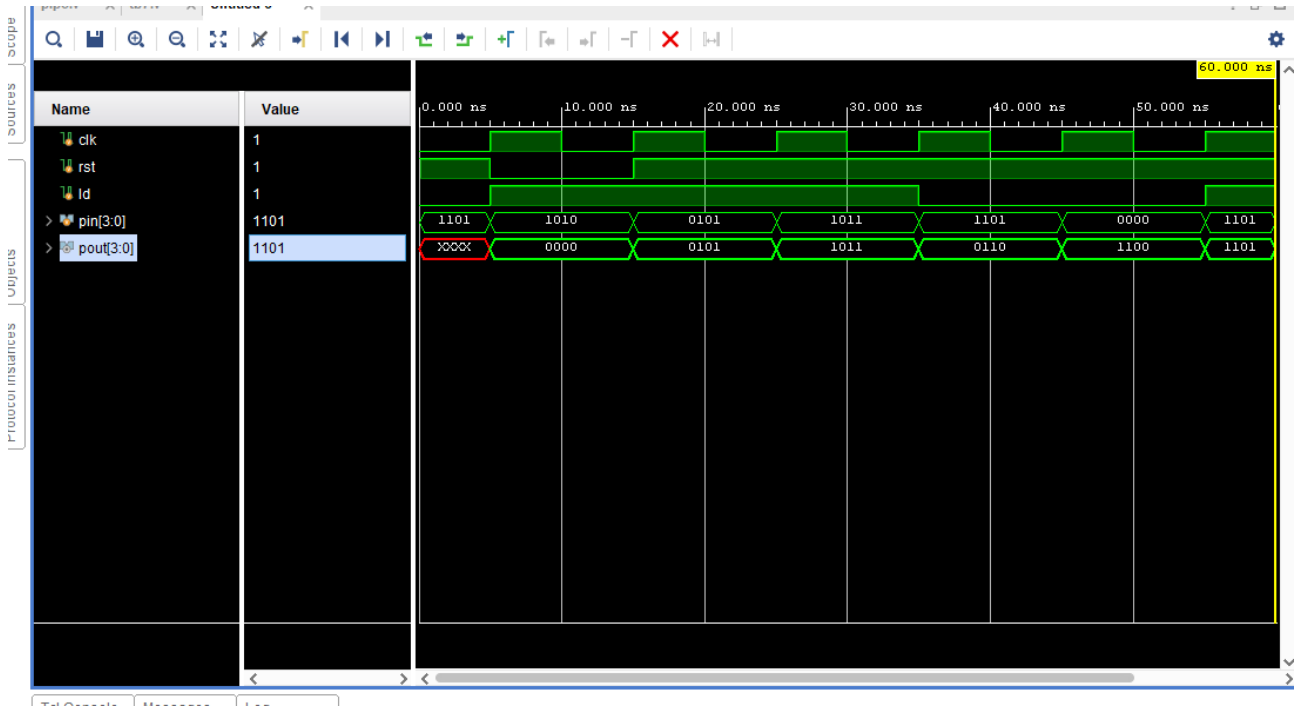
47.Parllel In Parllel Out Shift Register(PIPO)

Verilog Code:

```
module pipo(clk,rst,ld,pin,pout );
input clk,rst,ld;
input[3:0] pin;
output reg[3:0] pout;
always @(posedge clk)
begin
if(!rst)
pout<=4'b0000;
else if(ld)
pout<=pin;
else
pout<=(pout<<1);
end
endmodule
```

TestBench:

```
module tb7();
reg clk,rst,ld;
reg [3:0]pin;
wire [3:0]pout;
pipo uut(clk,rst,ld,pin,pout);
initial
begin
clk=0;
forever #5 clk=~clk;
end
initial
begin
rst=1;ld=0; pin=4'b1101; #5
rst=0;ld=1; pin=4'b1010; #10
rst=1; pin=4'b0101; #10
pin=4'b1011; #10
pin=4'b1101;ld=0; #10
pin=4'b0000; #10
pin=4'b1101;ld=1; #5
$finish;
end
endmodule
```



48.Master-Slave D Flipflop

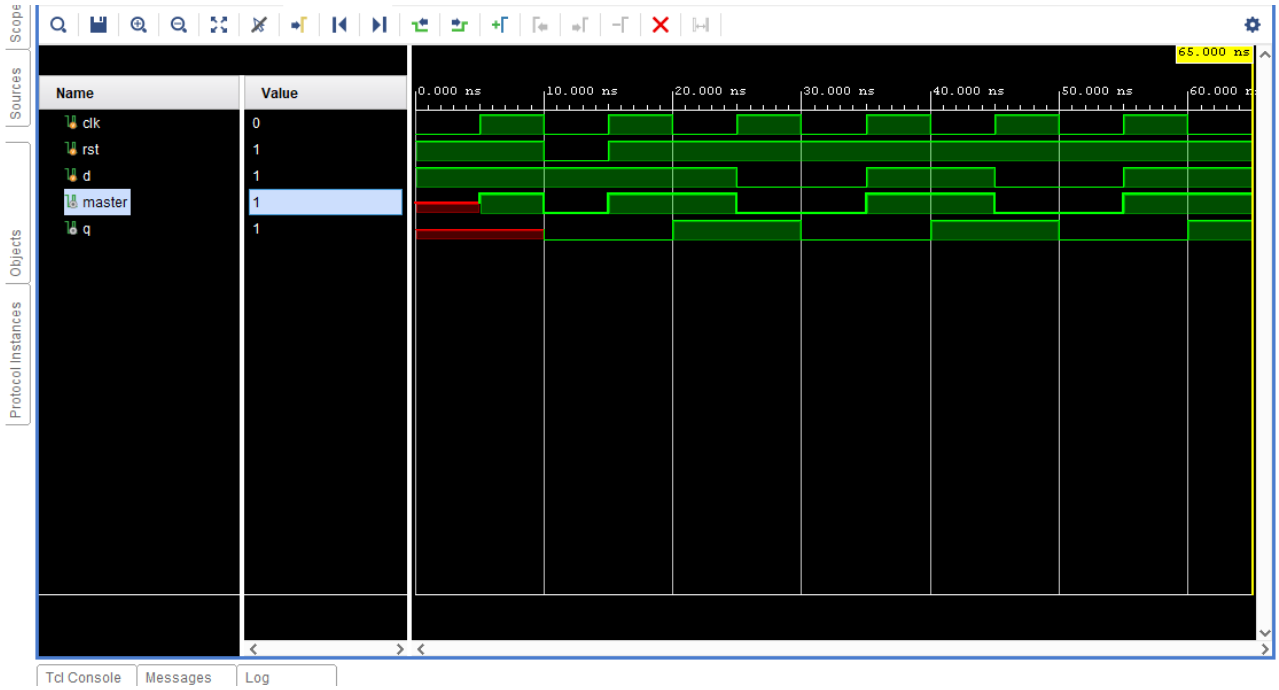
Verilog Code:

```
module master_slave_ff(clk,rst,d,master,q);
input clk,d,rst;
output reg q;
output reg master;
always@(posedge clk,negedge rst)
begin
if(!rst)
master<=0;
else
master<=d;
end
always@(negedge clk,negedge rst)
begin
if(!rst)
q<=0;
else
q<=master;
end
endmodule
```

TestBench:

```
module tb9();
reg clk,rst,d;
wire master,q;
```

master_slave_ff uut(clk,rst,d,master,q);
initial
begin
clk=0;
forever#5 clk=~clk;
end
initial
begin
d=1;rst=1; #10
d=1;rst=0; #5
d=1;rst=1; #10
d=0;rst=1; #10
d=1;rst=1; #10
d=0;rst=1; #10
d=1;rst=1; #10
$finish;
end
endmodule



49.Sequence Detector using Mealy FSM

Verilog Code:

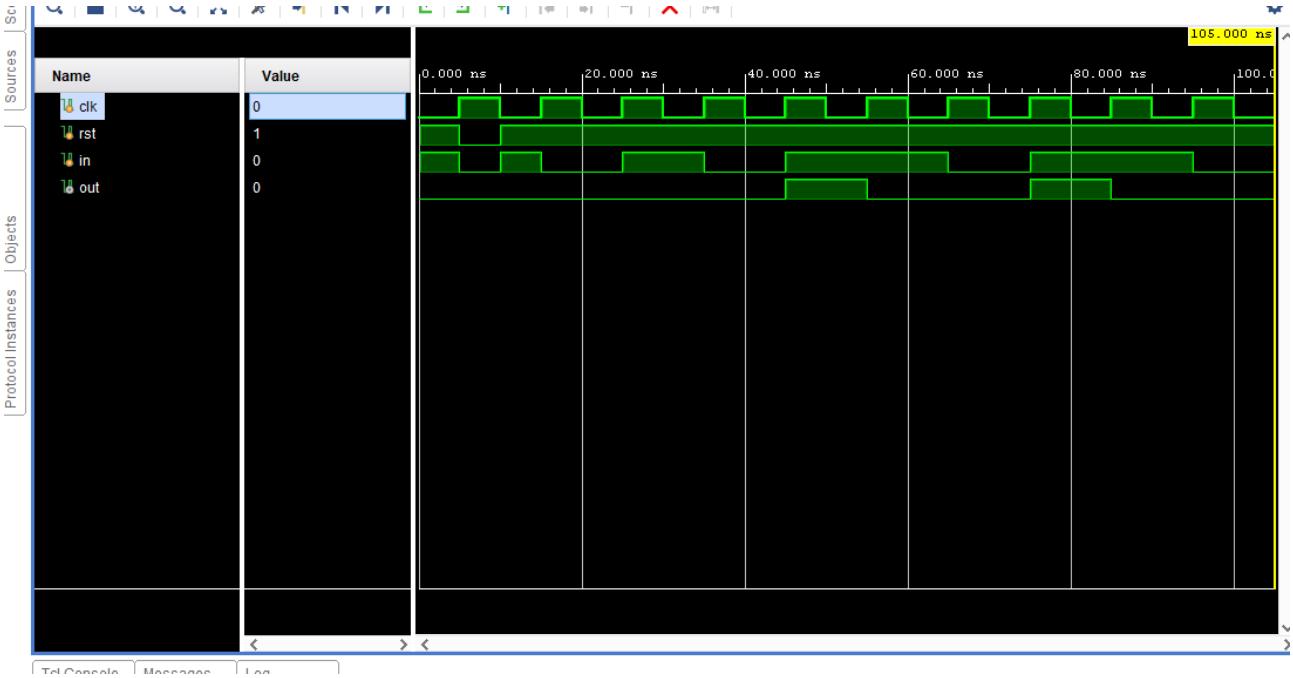
module seq_det_mealy (clk,rst, in,out);
input clk,rst, in;
output reg out;
reg [1:0] state, next_st;
parameter s0 = 2'b00, s1 = 2'b01,s2 = 2'b10,s3 = 2'b11;
always @(posedge clk or negedge rst)

```
begin
if (!rst)
state <= s0;
else
state <= next_st;
end
always @(*) begin
case (state)
s0: begin
next_st = in ? s1 : s0;
out = 0;
end
s1: begin
next_st = in ? s1 : s2;
out = 0;
end
s2: begin
next_st = in ? s3 : s0;
out = 0;
end
s3: begin
next_st = in ? s1 : s2;
out = in ? 1 : 0;
end
default: begin
next_st = s0;
out = 0;
end
endcase
end
endmodule
```

TestBench:

```
module tb10();
reg clk,rst, in;
wire out;
seq_det_mealy uut(clk,rst,in,out);
initial
begin
clk=0;
forever#5 clk=~clk;
end
initial
begin
in=1; rst=1; #5
```

in=0;rst=0;#5
in=1; rst=1; #5
in=0; #10
in=1; #10
in=0; #10
in=1; #10
in=1; #10
in=0; #10
in=1; #10
in=1; #10
in=0; #10
$finish;
end
endmodule



50.Sequence Detector using Moore FSM

Verilog Code:

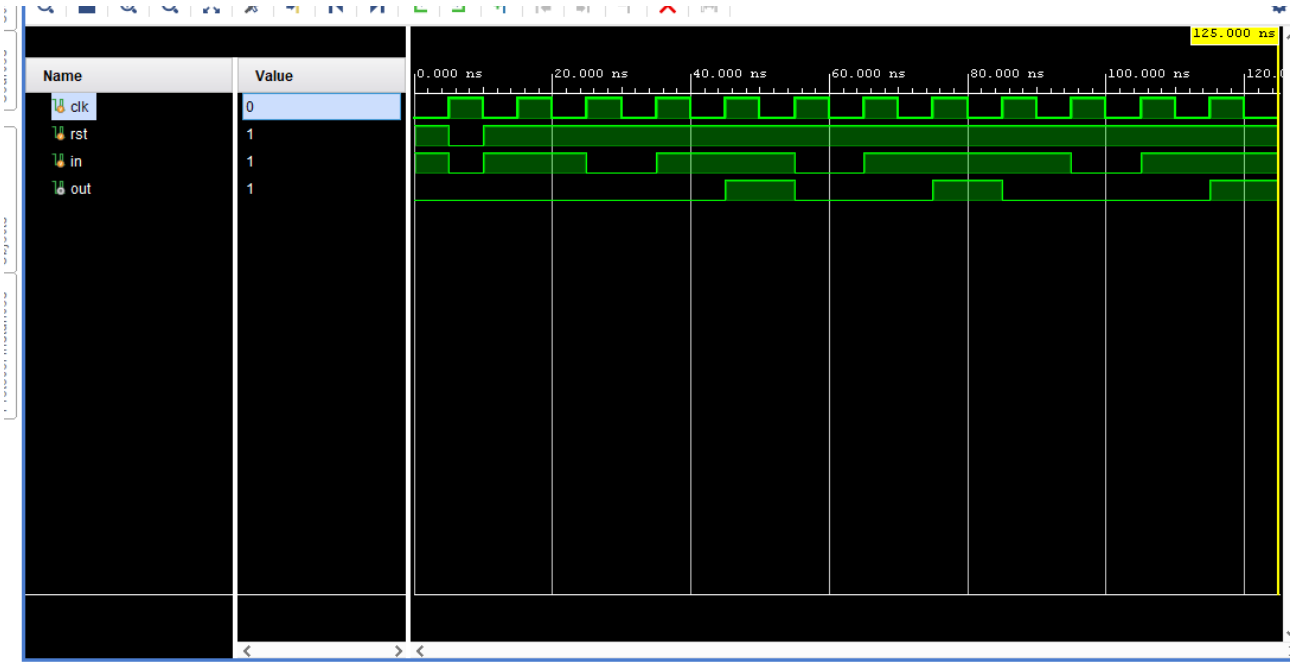
module seq_det_moore(clk,rst,in,out);
input clk,rst,in;
output reg out;
reg [2:0] state, next_st;
parameter s0 = 3'b000, s1 = 3'b001,s2 = 3'b010,s3 = 3'b011,s4=3'b100;
always @(posedge clk or negedge rst)
begin
if (!rst)
state <= s0;
else

```
state <= next_st;
end
always @(*) begin
case (state)
s0: begin
next_st = in ? s1 : s0;
out = 0;
end
s1: begin
next_st = in ? s1 : s2;
out = 0;
end
s2: begin
next_st = in ? s3 : s0;
out = 0;
end
s3: begin
next_st = in ? s4 : s2;
out = 0;
end
s4: begin
next_st = in ? s1 : s2;
out = 1;
end
default: begin
next_st = s0;
out = 0;
end
endcase
end
endmodule
```

TestBench:

```
module tb11();
reg clk,rst, in;
wire out;
seq_det_moore uut(clk,rst,in,out);
initial
begin
clk=0;
forever#5 clk=~clk;
end
initial
begin
in=1; rst=1; #5
```

in=0;rst=0;#5
in=1; rst=1; #15
in=0; #10
in=1; #10
in=1; #10
in=0; #10
in=1; #10
in=1; #20
in=0; #10
in=1; #10
in=1; #10
$finish;
end
endmodule



Thankyou

_SuryaPrakashTamma

n200122@rguktn.ac.in