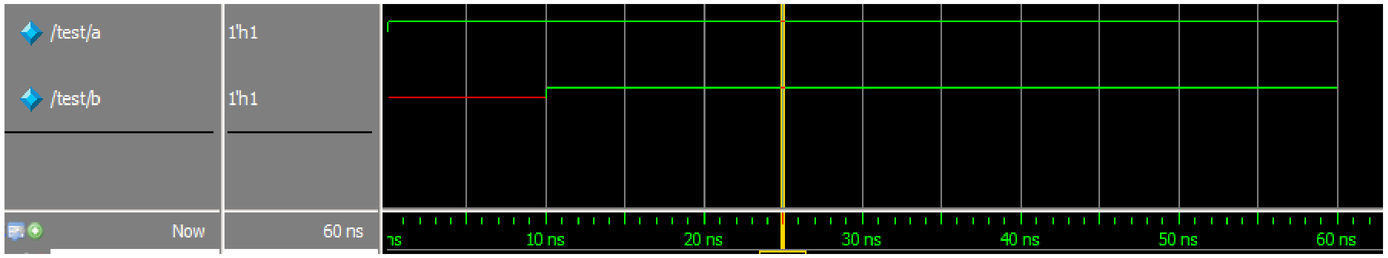
VERILOG CODE EXAMPLES

Examples Simulated using Model Sim Simulator

Verilog code to simulate non-blocking behavior

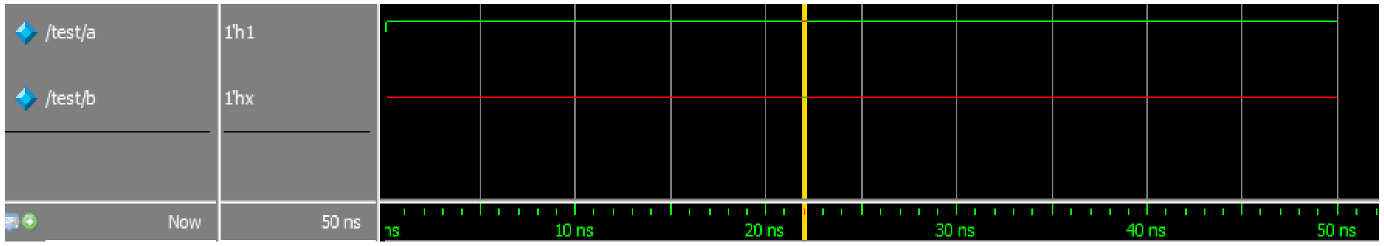
```
//Non-blocking example1
module test;
reg a, b;
initial
      begin
                  a <= 1'b0;
                  a <= 1'b1;
         #10     b <= a;
         #50     $stop;
      end
endmodule
```

Example1 Simulation output



```
//Non-blocking example2
module test;
reg a, b;
initial
      begin
                  a <= 1'b0;
                  a <= 1'b1;
                  b <= a;
         #50   $stop;
      end
endmodule
```

Example2 Simulation output

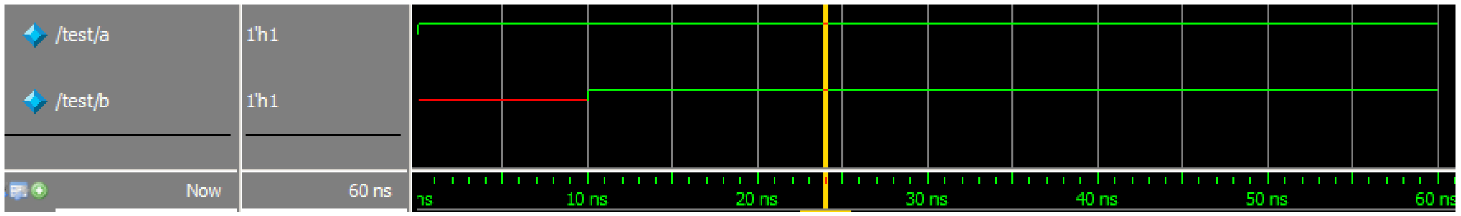


Change Non-Blocking (<=)to Blocking statements (=)in Example1 & Example2 and Predict Simulation output!!!

Verilog code to simulate blocking behavior

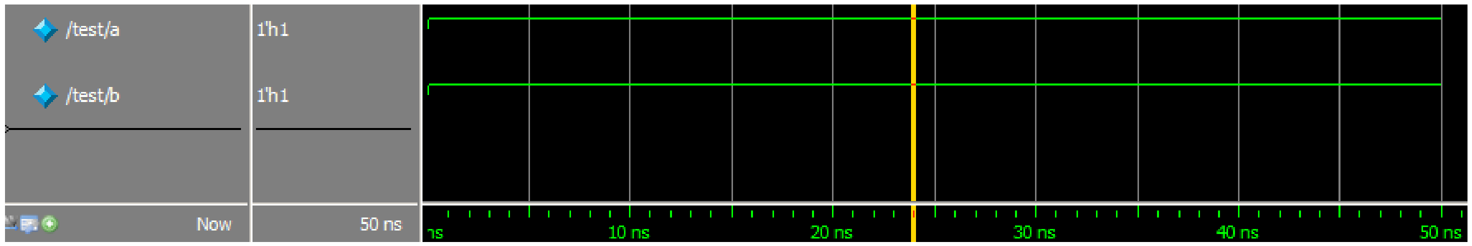
```
//blocking example1
module test;
reg a, b;
initial
    begin
            a = 1'b0;
            a = 1'b1;
        #10 b = a;
        #50 $stop;
    end
endmodule
```

Example1 Simulation output



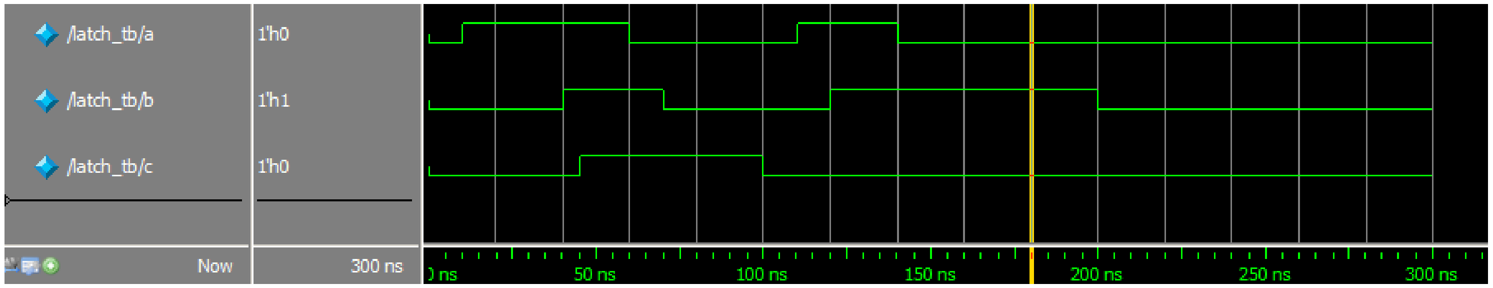
```
//blocking example2
module test;
reg a, b;
initial
    begin
            a = 1'b0;
            a = 1'b1;
            b = a;
        #50 $stop;
    end
endmodule
```

Example2 Simulation output



Nested sequential(being end) and parallel blocks(fork join)

```
`timescale 1ns/1ps //timescale directive
module latch_tb;
reg a,b,c;
initial
            begin
             a = 0; b =0; c =0;   #10 a = 1;   #30  b = 1;
                        fork
                                    #5 c = 1;     #20 a = 0;   #30 b = 0;   #60 c = 0;   #70 a=1; #80 b=1;
                                    #100 a =0;  #160 b =0;
                        join
                        #100 $finish;
            end
endmodule
```

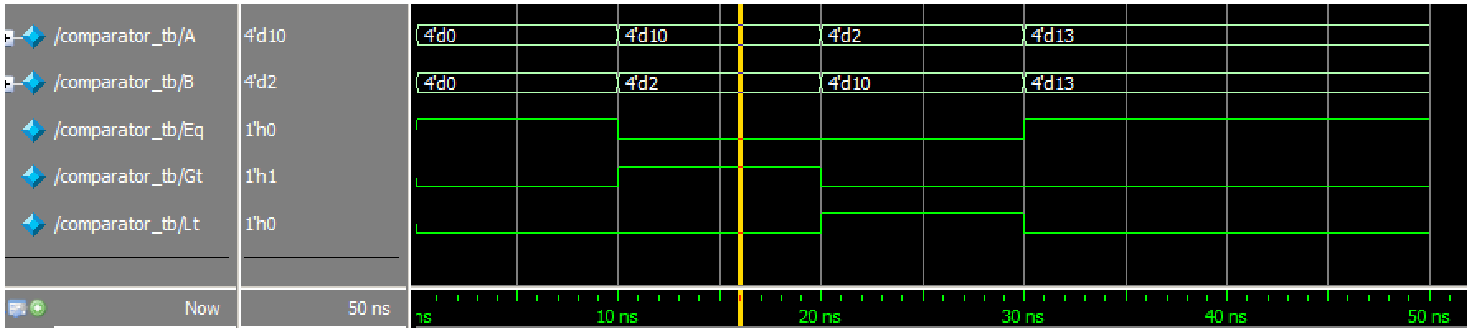


Combinational logic Verilog coding: 4-bit comparator using Data flow modeling

```
/*4-bit comparator code using Data Flow
modeling */
module comparator_4bit(Eq, Gt, Lt, A, B);
 input [3:0] A, B;
 output Eq, Gt, Lt;

 assign Eq = (A==B);
 assign Lt  = (A<B);
 assign Gt = (A>B);
endmodule
```

```
//4-bit comparator test bench
module comparator_tb;
reg [3:0]     A, B;  //inputs defined as reg data type
wire Eq, Gt, Lt; //outputs defined as wire
comparator_4bit c1(Eq, Gt, Lt, A, B); //comparator module instantiation
initial
    begin      A = 0; B = 0; #10 A = 10; B = 2; #10 A = 2; B = 10;
               #10 A = 13; B = 13; #60 $stop;  end
initial $monitor($time," %d %s %d",A, (Eq)?"==":(Lt)?"<":">", B);
endmodule
```

Simulation output



Displayed output:

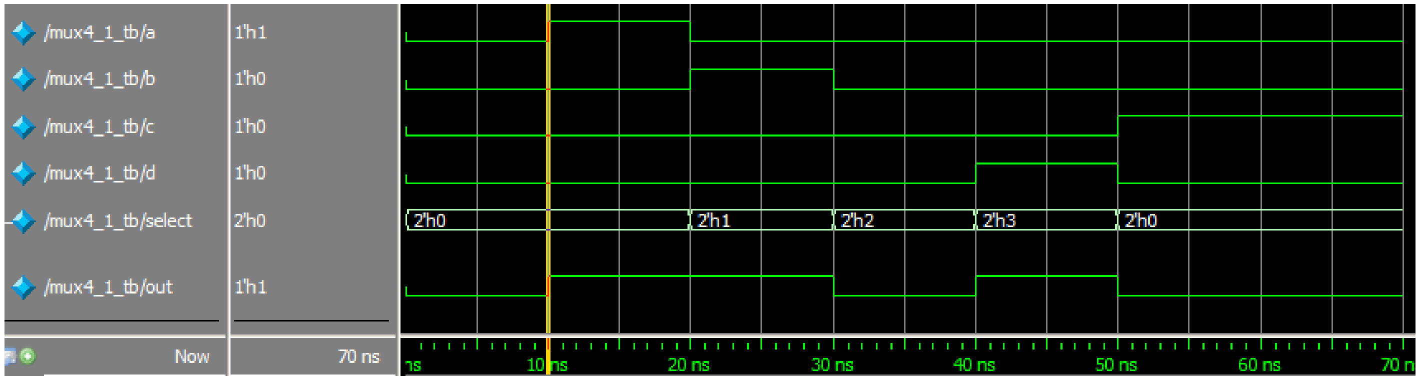
```
#   0     0 ==  0
#   10    10  >  2
#   20    2  < 10
#   30    13 == 13
```

Multiplexer implementation using function

```
module mux4_1(a, b, c, d, select, out);
   input      a, b, c, d;     input [1:0] select;  output      out;
assign out = mux (a, b, c, d, select); //function called using  assignment statement
   function    mux;
      input      a, b, c, d;
      input [1:0] select;
      case (select)
         2'b00:   mux = a;
         2'b01:   mux = b;
         2'b10:   mux = c;
          2'b11:   mux = d;
          default: mux = 'bx;
      endcase
   endfunction
endmodule
```

```
module mux4_1_tb;
reg a,b,c,d; reg[1:0] select; wire     out;
mux4_1 mux1(a, b, c, d, select, out);
initial
$monitor("At t=%t, a=%b, b=%b, c=%b, d=%b, select = %b, out
= %b", $time, a, b, c, d, select, out);
initial
begin
a = 1'b0; b =1'b0; c=1'b0; d=1'b0; select = 2'b00;
#10 a = 1'b1; b =1'b0; c=1'b0; d=1'b0; select = 2'b00;
#10 a = 1'b0; b =1'b1; c=1'b0; d=1'b0; select = 2'b01;
#10 a = 1'b0; b =1'b0; c=1'b0; d=1'b0; select = 2'b10;
#10 a = 1'b0; b =1'b0; c=1'b0; d=1'b1; select = 2'b11;
#10 a = 1'b0; b =1'b0; c=1'b1; d=1'b0; select = 2'b00;#20 $stop;
end
endmodule
```

Simulation output

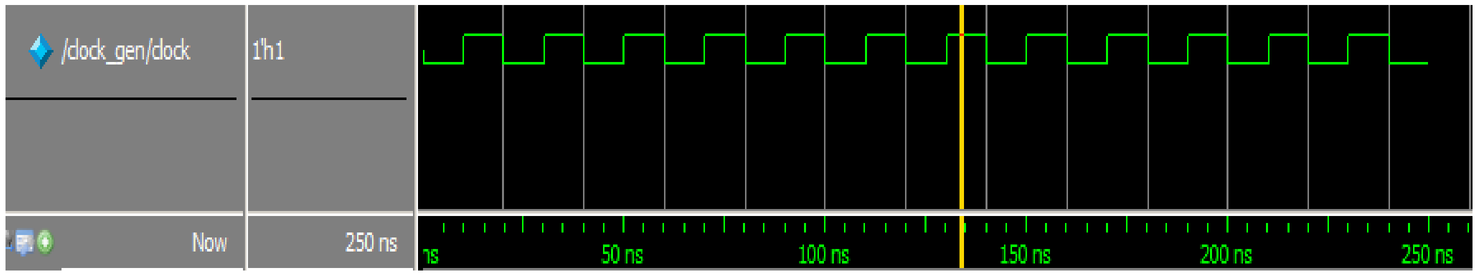


$monitor Output

```
# At t=      0, a=0, b=0, c=0, d=0, select = 00, out = 0
# At t=     10, a=1, b=0, c=0, d=0, select = 00, out = 1
# At t=     20, a=0, b=1, c=0, d=0, select = 01, out = 1
# At t=     30, a=0, b=0, c=0, d=0, select = 10, out = 0
# At t=     40, a=0, b=0, c=0, d=1, select = 11, out = 1
# At t=     50, a=0, b=0, c=1, d=0, select = 00, out = 0
```

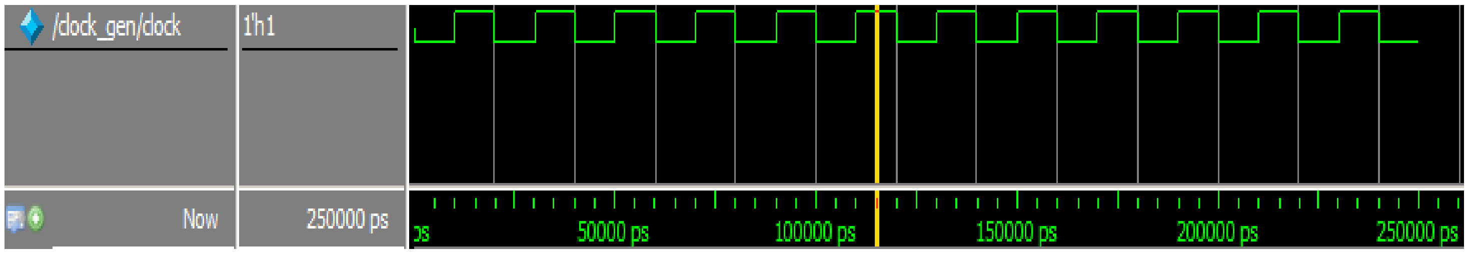
Clock Generator using always block

```
`timescale 1ns/1ns //time scale with time unit of 1ns and precision of 1ns
module clock_gen; //clock generator module
reg clock;
initial
        clock = 1'b0; //Initialize clock at time 0

always #10 clock = ~clock; //toggle clock every half cycle(clock period = 20)

initial #500 $finish;
endmodule
```



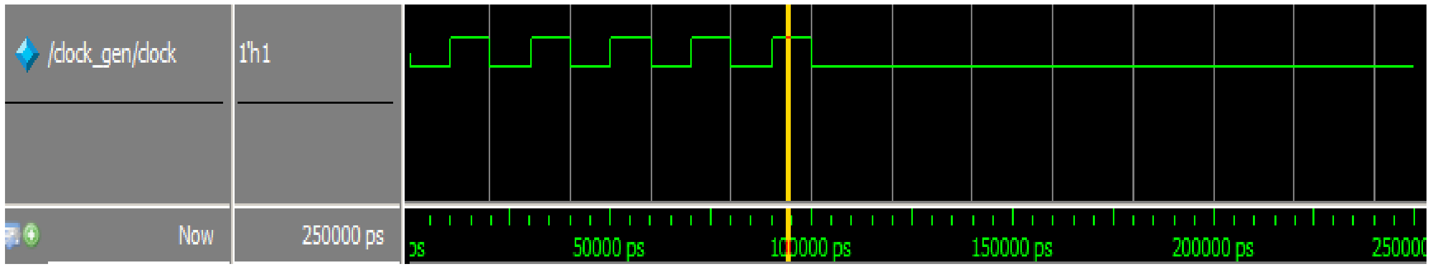
Clock Generator using forever statement

```
`timescale 1ns/100ps //time scale with time unit of 1ns and precision of 100ps
module clock_gen; //clock generator module
reg clock;
initial
    begin
            clock = 1'b0; //Initialize clock at time 0
            forever #10 clock = ~clock; //toggle clock every half cycle(clock period = 20)
    end
initial #250 $finish;
endmodule
```



Clock Generator using repeat statement

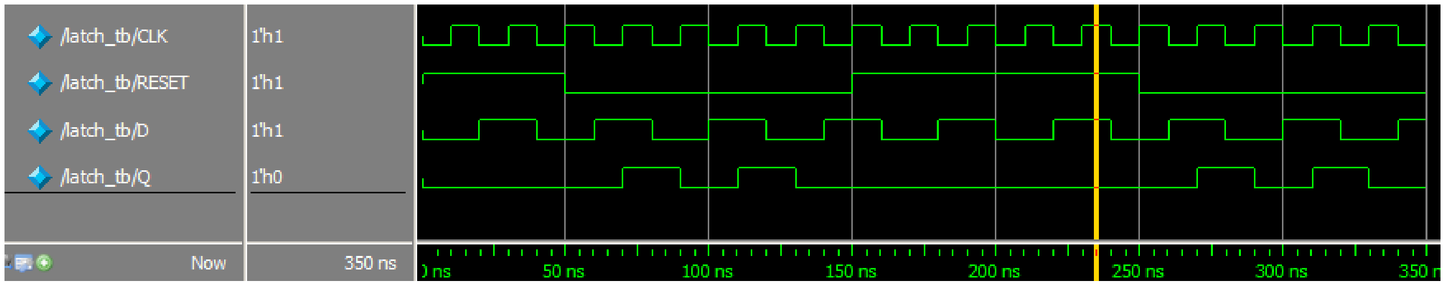
```
`timescale 1ns/100ps //time scale with time unit of 1ns and precision of 100ps
module clock_gen; //clock generator module
reg clock;
initial
    begin
            clock = 1'b0; //Initialize clock at time 0
            repeat(10)
              #10 clock = ~clock; //toggle clock every half cycle(clock period = 20)
    end
initial #250 $finish;
endmodule
```



Latch Verilog implementation

```
`timescale 1ns/1ps //timescale directive
//level sensitive latch with asynchronous reset
module latch(q, d, clock, reset);
output q;
input d, clock, reset;
reg q;
//Wait for reset or clock or d to change
always @( reset or clock or d) begin
            if (reset) // if reset signal is high, set q to 0.
               q = 1'b0;
            else if(clock) //if clock is high, latch input
               q = d;
  end
endmodule
```

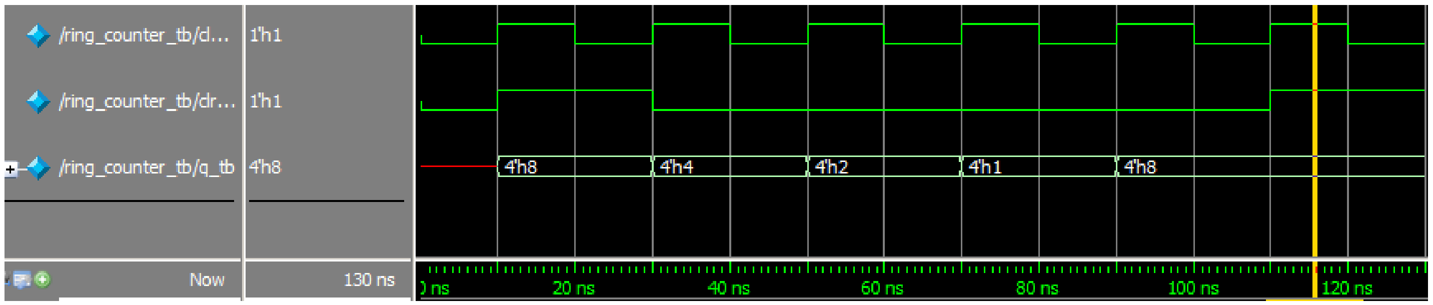
```
`timescale 1ns/1ps //timescale directive
module latch_tb;
wire Q; reg D, CLK, RESET;
initial
  begin
    CLK = 0;RESET = 1; #50 RESET = 0; #100 RESET = 1;
    #100 RESET = 0; #100 $finish;
  end
initial
  begin  D = 0; forever #20 D = ~D; end
always #10 CLK = ~CLK;
latch   lat1(.q(Q),   .d(D),   .clock(CLK),   .reset(RESET));
//instantiate latch
endmodule
```



Ring counter Verilog Code

```verilog
module ring_count(q,clk,clr);
  input clk,clr;
  output [3:0]q;
  reg [3:0]q;
  always @(posedge clk)
      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
endmodule
```

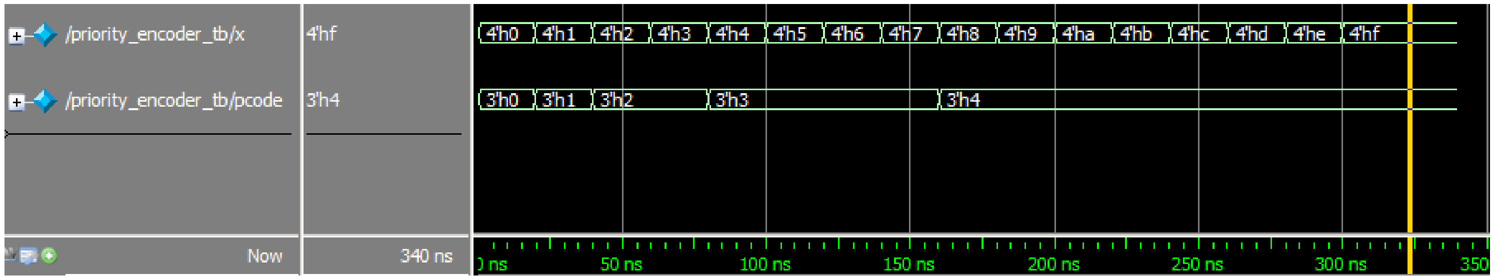
```verilog
`timescale 1ns/1ps
module ring_counter_tb();
reg clk_tb,clr_tb;
wire [3:0]q_tb;
ring_count dut1(q_tb,clk_tb,clr_tb);
initial
  begin
    clk_tb =0;
   $display("At t=%t / clk_tb =%b clr_tb =%b q_tb=%h",$time, clk_tb, clr_tb, q_tb);
   $monitor($time," clk_tb =%b clr_tb =%b q_tb=%h \n",clk_tb,clr_tb,q_tb);
   clr_tb=1'b0; #10 clr_tb=1'b1; #20 clr_tb=1'b0; #80 clr_tb = 1;
   #20 $finish;
 end
   always #10 clk_tb = ~clk_tb;
endmodule
```



Priority encoder Verilog Code

```
module  priority_encoder
( input wire  [4:1] x,
output reg  [2:0]  pcode );

always  @(x[4], x[3],x[2], x[1])
if (x[4] == 1'b1)
            pcode = 3'b100;
else if (x[3] == 1'b1)
            pcode = 3'b011;
else if (x[2] == 1'b1)
            pcode = 3'b010;
else if (x[1] == 1'b1)
            pcode = 3'b001;
else pcode = 3'b000;
endmodule
```

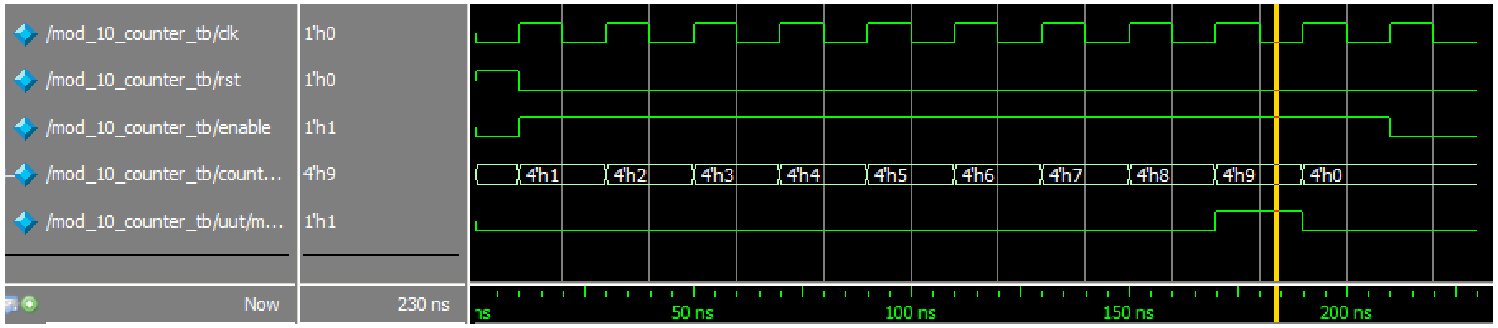
```
`timescale 1ns / 1ps
module priority_encoder_tb;
            reg [4:1] x;
            wire [2:0] pcode;
priority_encoder uut ( .x(x), .pcode(pcode) ); // Instantiate the Unit Under Test (UUT)
initial begin // Initialize Inputs
x = 4'b0000;
 #20 x = 4'b0001; #20 x = 4'b0010; #20 x = 4'b0011; #20 x = 4'b0100; #20 x = 4'b0101;
 #20 x = 4'b0110; #20 x = 4'b0111; #20 x = 4'b1000; #20 x = 4'b1001; #20 x = 4'b1010;
 #20 x = 4'b1011; #20 x = 4'b1100; #20 x = 4'b1101; #20 x = 4'b1110; #20 x = 4'b1111;
 #40 $finish;
 end
 initial begin
 $monitor("t=%3d x=%4b,pcode=%3b",$time,x,pcode );end
endmodule
```


/priority_encoder_tb/x
4'hf
4'h0 4'h1 4'h2 4'h3 4'h4 4'h5 4'h6 4'h7 4'h8 4'h9 4'ha 4'hb 4'hc 4'hd 4'he 4'hf
/priority_encoder_tb/pcode
3'h4
3'h0 3'h1 3'h2 3'h3 3'h4
Now
340 ns
0 ns
50 ns
100 ns
150 ns
200 ns
250 ns
300 ns
350

Mod-10 counter Verilog code

```
module counter( input clk,rst,enable,
output reg [3:0]counter_output );
assign mod_10_enable = (counter_output == 4'b1001);
always@ (posedge clk or posedge rst)
  begin
   if( rst)
      counter_output <= 4'b0000;
   else
     begin
        if(enable && ~mod_10_enable)
                    counter_output <= counter_output + 1;
        else
                    counter_output <= 0;
     end
end
endmodule
```

```
module mod_10_counter_tb;
reg clk; reg rst; reg enable;
wire [3:0] counter_output;
// Instantiate the Unit Under Test (UUT)
counter uut (.clk(clk), .rst(rst),
.enable(enable),.counter_output(counter_output));
always #50 clk= ~ clk;
initial
    begin
                clk=0; rst = 1; enable = 0;
                #10 rst=0; enable=1;
                #200 enable = 0;
                #20 $finish;
  end
endmodule
```



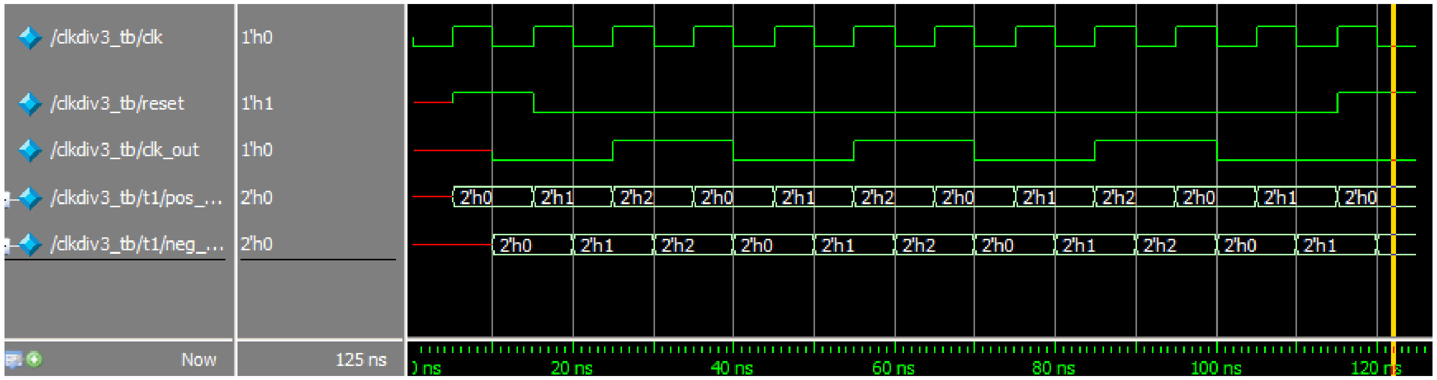
/mod_10_counter_tb/clk 1'h0
/mod_10_counter_tb/rst 1'h0
/mod_10_counter_tb/enable 1'h1
/mod_10_counter_tb/count... 4'h9
/mod_10_counter_tb/uut/m... 1'h1
Now 230 ns
4'h1 4'h2 4'h3 4'h4 4'h5 4'h6 4'h7 4'h8 4'h9 4'h0
ns 50 ns 100 ns 150 ns 200 ns

Clock Divide by 3 Verilog code

```
module clk_div3(clk,reset, clk_out);
input clk;
input reset;
output clk_out;
reg [1:0] pos_count, neg_count;
//if the count number for either of the two counters is 2.
assign clk_out = ((pos_count == 2) | (neg_count == 2));

always @(posedge clk)
if (reset)
            pos_count <=0;
else if (pos_count ==2)
            pos_count <= 0;
else
            pos_count<= pos_count +1;

always @(negedge clk)
if (reset)
            neg_count <=0;
else  if (neg_count ==2)
            neg_count <= 0;
else
            neg_count<= neg_count +1;

endmodule
```

```
`timescale 1ns/100ps

module clkdiv3_tb;
 reg clk,reset;
 wire clk_out;

   clk_div3 t1(clk,reset,clk_out);
     initial
       clk= 1'b0;
   always
     #5  clk=~clk;
     initial
        begin
          #5 reset=1'b1;
          #10 reset=1'b0;
          #100 reset = 1'b1;
          #10 $finish;
        end

     initial
          $monitor("clk=%b,reset=%b,clk_out=%b",clk,reset,clk_out);

  endmodule
```

Clock Divide by 3 Simulation output



/clkdiv3_tb/clk 1'h0
/clkdiv3_tb/reset 1'h1
/clkdiv3_tb/clk_out 1'h0
/clkdiv3_tb/t1/pos_... 2'h0
/clkdiv3_tb/t1/neg_... 2'h0
Now 125 ns
2'h0 2'h1 2'h2 2'h0 2'h1 2'h2 2'h0 2'h1 2'h2 2'h0 2'h1 2'h0
2'h0 2'h1 2'h2 2'h0 2'h1 2'h2 2'h0 2'h1 2'h2 2'h0 2'h1
0 ns 20 ns 40 ns 60 ns 80 ns 100 ns 120 ns

```
# clk=0,reset=x,clk_out=x
# clk=1,reset=1,clk_out=x
# clk=0,reset=1,clk_out=0
# clk=1,reset=0,clk_out=0
# clk=0,reset=0,clk_out=0
# clk=1,reset=0,clk_out=1
# clk=0,reset=0,clk_out=1
# clk=1,reset=0,clk_out=1
# clk=0,reset=0,clk_out=0
# clk=1,reset=0,clk_out=0
# clk=0,reset=0,clk_out=0
# clk=1,reset=0,clk_out=1
# clk=0,reset=0,clk_out=1
```

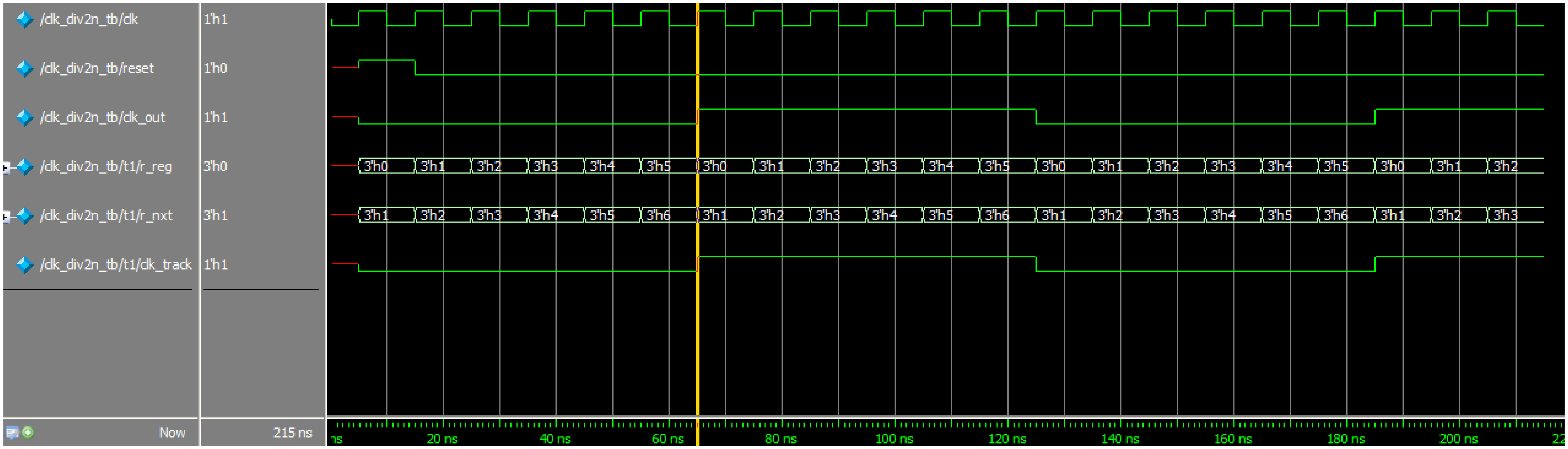
```
# clk=1,reset=0,clk_out=1
# clk=0,reset=0,clk_out=0
# clk=1,reset=0,clk_out=0
# clk=0,reset=0,clk_out=0
# clk=1,reset=0,clk_out=1
# clk=0,reset=0,clk_out=1
# clk=1,reset=0,clk_out=1
# clk=0,reset=0,clk_out=0
# clk=1,reset=0,clk_out=0
# clk=0,reset=0,clk_out=0
# clk=1,reset=1,clk_out=0
# clk=0,reset=1,clk_out=0
```

Clock Divide by even number 2N verilog

```
module clk_div2n
#(
parameter WIDTH = 3, // Width of the register required
parameter N = 6// We will divide by 12 for example in this case
) (clk,reset, clk_out);
input clk; input reset; output clk_out;
reg [WIDTH-1:0] r_reg;
wire [WIDTH-1:0] r_nxt;
reg clk_track;
/*We have triggered the always block at positive edge of clock
and count up to N and then invert the output clock.*/
always @(posedge clk or posedge reset)
begin
  if (reset)      begin
        r_reg <= 0; clk_track <= 1'b0; end
  else if (r_nxt == N)
                    begin     r_reg <= 0;   clk_track <= ~clk_track;  end
  else
      r_reg <= r_nxt;
end
 assign r_nxt = r_reg+1;
 assign clk_out = clk_track;
endmodule
```

```
module clkdiv2n_tb;
  reg clk,reset;
  wire clk_out;

     clk_div2n t1(clk,reset,clk_out);
        initial
          clk= 1'b0;
     always
        #5  clk=~clk;
        initial
            begin
              #5 reset=1'b1;
              #10 reset=1'b0;
              #200 $finish;
            end
        initial
$monitor("clk=%b,reset=%b,clk_out=%b",clk,reset,clk_out);
endmodule
```

Clock Divide by even number 2N simulation output

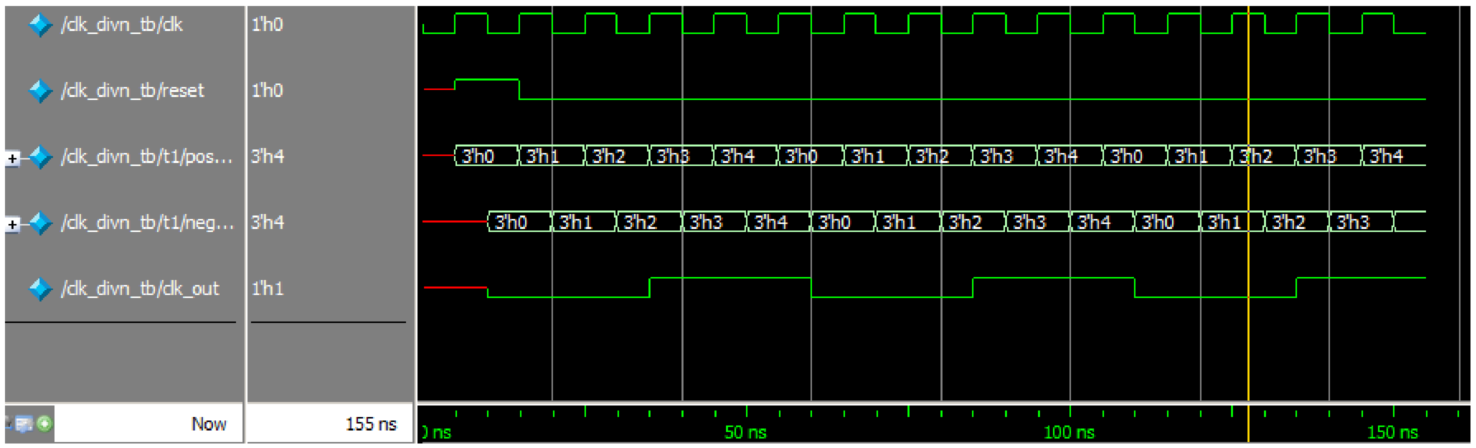

/clk_div2n_tb/clk
/clk_div2n_tb/reset
/clk_div2n_tb/clk_out
/clk_div2n_tb/t1/r_reg
/clk_div2n_tb/t1/r_nxt
/clk_div2n_tb/t1/clk_track
1'h1
1'h0
1'h1
3'h0
3'h1
1'h1
Now
215 ns
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns
160 ns
180 ns
200 ns
3'h0 3'h1 3'h2 3'h3 3'h4 3'h5 3'h0 3'h1 3'h2 3'h3 3'h4 3'h5 3'h0 3'h1 3'h2 3'h3 3'h4 3'h5 3'h0 3'h1 3'h2
3'h1 3'h2 3'h3 3'h4 3'h5 3'h6 3'h1 3'h2 3'h3 3'h4 3'h5 3'h6 3'h1 3'h2 3'h3 3'h4 3'h5 3'h6 3'h1 3'h2 3'h3

Clock Divide by N odd Verilog code

```
module clk_divn #(
parameter WIDTH = 3,
parameter N = 5) (clk,reset, clk_out);
input clk; input reset;
output clk_out;
reg [WIDTH-1:0] pos_count, neg_count;
assign clk_out = ((pos_count > (N>>1)) | (neg_count > (N>>1)));
 //We count the number of the positive edges
always @(posedge clk)
 if (reset)
                    pos_count <=0;
 else if (pos_count ==N-1)
                    pos_count <= 0;
 else
                    pos_count<= pos_count +1;
 //we count the negative edges
 always @(negedge clk)
 if (reset)
                    neg_count <=0;
 else  if (neg_count ==N-1)
                    neg_count <= 0;
 else
                    neg_count<= neg_count +1;
endmodule
```

```
`timescale 1ns/100ps
module clk_divn_tb;
  reg clk,reset;
  wire clk_out;

     clk_divn t1(clk,reset,clk_out);
        initial
          clk= 1'b0;
     always
        #5  clk=~clk;
        initial
            begin
               #5 reset=1'b1;
               #10 reset=1'b0;
               #140 $finish;
            end

        initial

$monitor("clk=%b,reset=%b,clk_out=%b",clk,reset,clk_out);

    endmodule
```

Clock Divide by n odd Simulation output



1. Notice the math involved. assign clk_out = ((pos_count > (N>>1)) | (neg_count > (N>>1)));
N>>1 needs some explanation. If N were an even number N>>1 is simply a division by 2.
But here N is an odd number. Consider for example N is 11 or 5'b01011. We know that when we do right shift, the right most bit is lost. - which is equivalent to N getting converted to N-1. And then the right shift divides the number by 2. So N>>1, when, N is off mean N becomes (N-1)/2. So, when N =11, N >> 1 make it 5. If = 5 or 3'b101, then the N>>1 makes it 3'b010 (The right most digits is lost) or 2.

2. To understand how the math works out consider the two counters in the figure below that keeps count of the number of positive and negative pulses. for N=7. The trick is to look at the waveforms for the counters for an example (set N to any fixed number) and find the right math to fit it. Once the math works, generalize it for N.