Chapter 5 Behavioral Modeling

Structured Procedures

All procedures in Verilog are specified within one of the following four statements:

1. **initial** statement

   The statement executes only once and its activity dies when the statement has finished.

2. **always** statement

   The statement executes repeatedly. Its activity dies only when the simulation is terminated.

3. task (Chap 9)
4. function (Chap 9)

With the increasing complexity of digital design, it has become vitally important to make wise design decisions early in a project. Designers need to be able to evaluate the trade-off of various architectures and algorithms before they decide on the optimum architecture and algorithm to implement in hardware.

5.1 Structures procedures

There are two structure procedure statements in Verilog: **always** and **initial**. These statements are the two most basic statements in behavioral modeling. All other behavioral statements can appear only inside these structured procedure statements.

5.1.1 **initial** Statement

All statements inside an initial statement constitute an initial block. **An initial block** starts at time 0, and executes exactly once during a simulation, and then does not execute again. If there are multiple initial blocks, each block starts to execute concurrently at time 0.

Ex1. **initial** statement program (set up & test signal generation)

Verilog Code (set up)

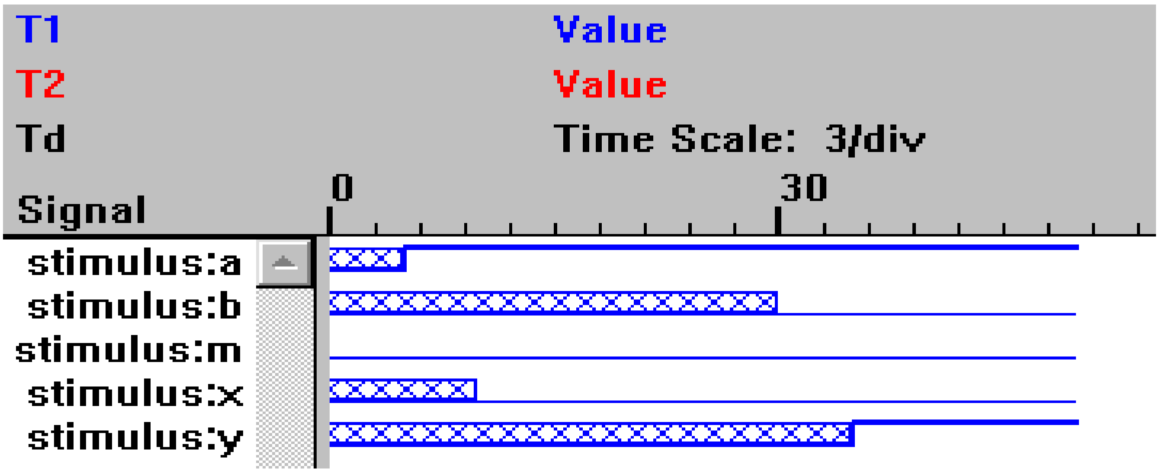
single statement;
does not need to be
grouped

```
module stimulus;
reg x,y, a,b, m;
initial
      m = 1'b0;
```

```
initial
    begin
       #5 a = 1'b1; end
initial
  begin
       #10 x = 1'b0;
       #20 b = 1'b0;
       #5    y = 1'b1;
  end
initial
       #50 $finish;
endmodule
```

multiple statements;
need to be grouped

Simulation Waveform



Simulation Result

Time	statement executes
0	m=1’b0;
5	a=1’b1;
10	x=1’b0;
30	b=1’b0;
35	y=1’b1;
50	$finish;

Example 8-34 : Use of initial statement (Clear the registers)

```
initial
     begin
           areg = 0;                                  //initialize a register
           for (index = 0; index < size; index = index +1)
                memory[index] = 0;                    //initialize a memory word
     end
```

Example 8-35 : Another use for initial statement

```
initial
     begin
           inputs = ‘b000000;
     //initialize at time zero
              #10 inputs = ‘b011001;       //fist test pattern
              #10 inputs = ‘b011011;       //second test pattern
              #10 inputs = ‘b011000;       //third test pattern
              #10 inputs = ‘b001000;       //last test pattern
     end
```

5.1.2 **always** Statement

All behavioral statements inside an always statement constitute an always block. **The always statement** starts at time 0 and executes the statement in the always block continuously in a looping fashion.

Ex2. always statement program

In the example, the always statement starts at time 0 and executes the statement clock = ~ clock every 10 time units.

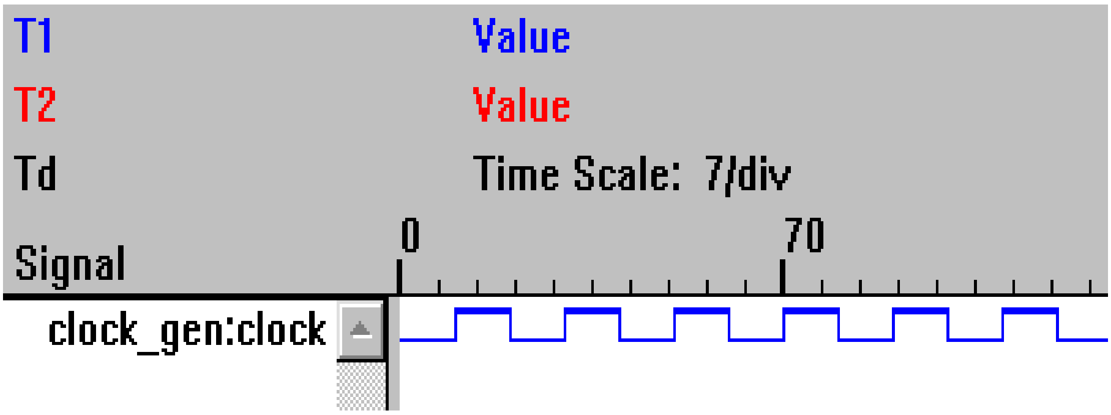
Verilog Code

Initialize clock at time zero

Toggle clock every half cycle (time period = 20)

```
module clock_gen;
reg clock;
initial
              clock = 1'b0;
always
              #10 clock = ~clock;
initial    forever #10 clock = ~clock;
initial     #1000 $finish;
endmodule
```

Simulation Waveform



5.2 Procedural Assignments (Hardware description)

Procedural assignments update values of **reg , integer , real ,** or **time** variables. The value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value. The syntax for the simplest form of procedural assignment is shown below.

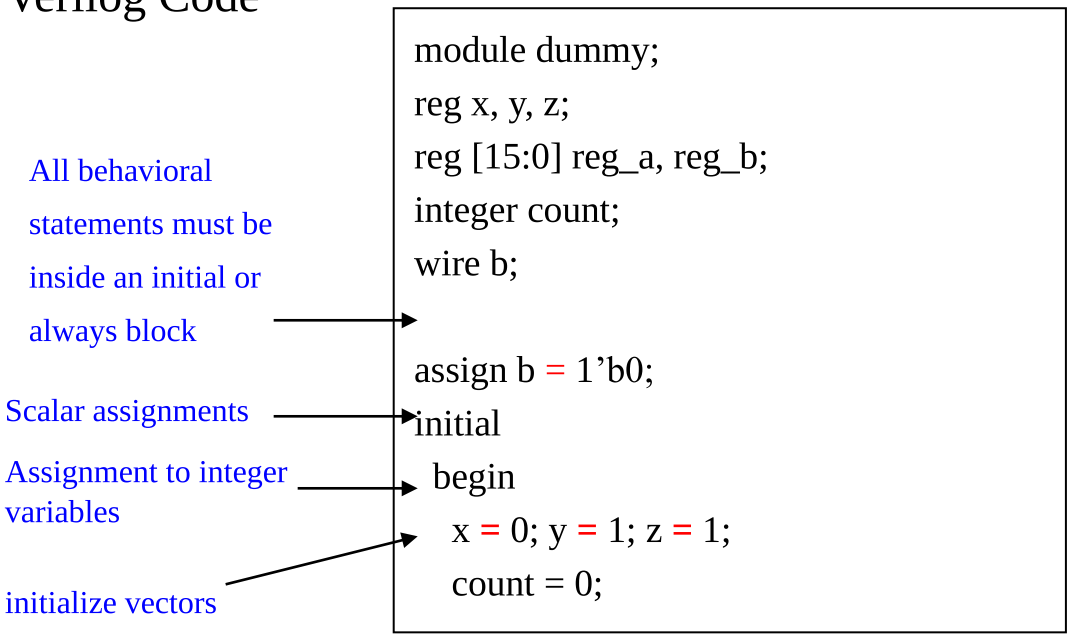
```
<assignment>
          : : = < value > = < expression >
```

5.2.1 Blocking assignments

Blocking assignment statements are executed in the order and they are specified in a sequential block. A blocking assignment will not block execution of statements that follow in a parallel block.

Ex3. Blocking Statements Program

Verilog Code



```
module dummy;
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
wire b;

assign b = 1’b0;
initial
  begin
    x = 0; y = 1; z = 1;
    count = 0;
```

All behavioral statements must be inside an initial or always block
Scalar assignments
Assignment to integer variables
initialize vectors

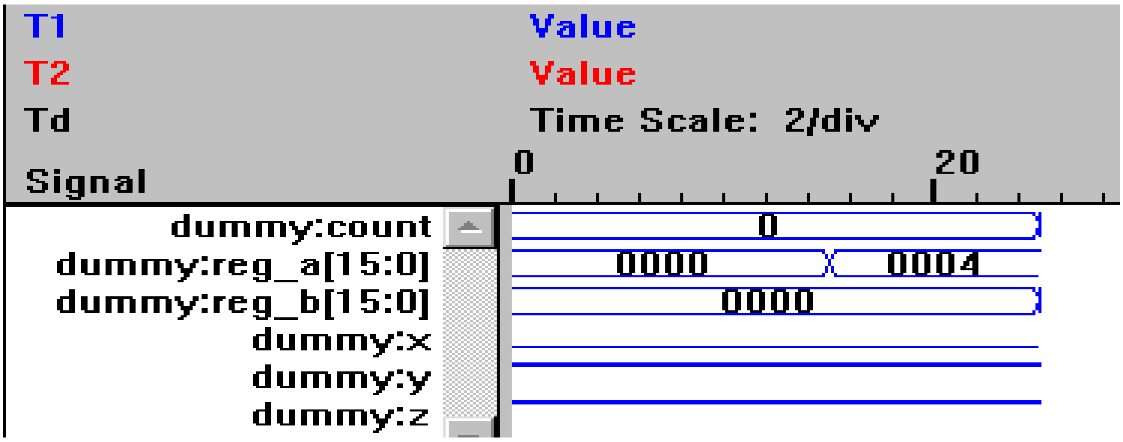
Bit select assignment with delay

Assign result of concatenation to part select of a vector

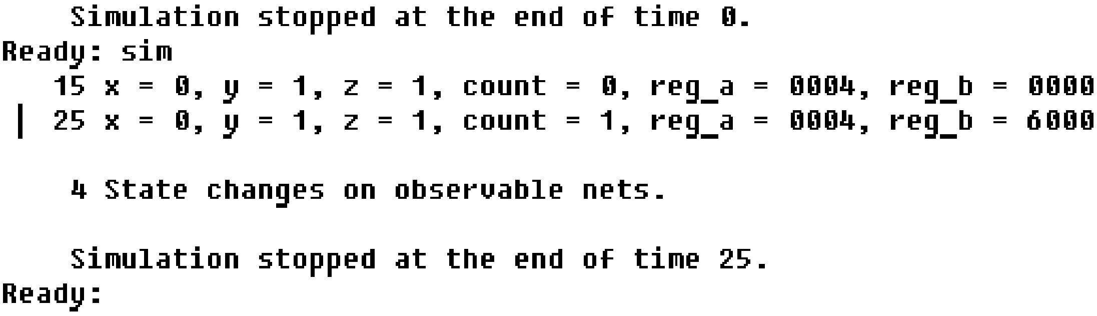
Show the result

```
   reg_a[2] = #15 1;
   reg_b[15:13] = #10 {x, y, z};
   count = count + 1;
  end

initial
$monitor($time, " x = %b, y = %b, z =    %b,
count = %0d, reg_a = %x, reg_b = %x",x, y,
z, count, reg_a, reg_b);

endmodule
```

Simulation Waveform



Simulation Result



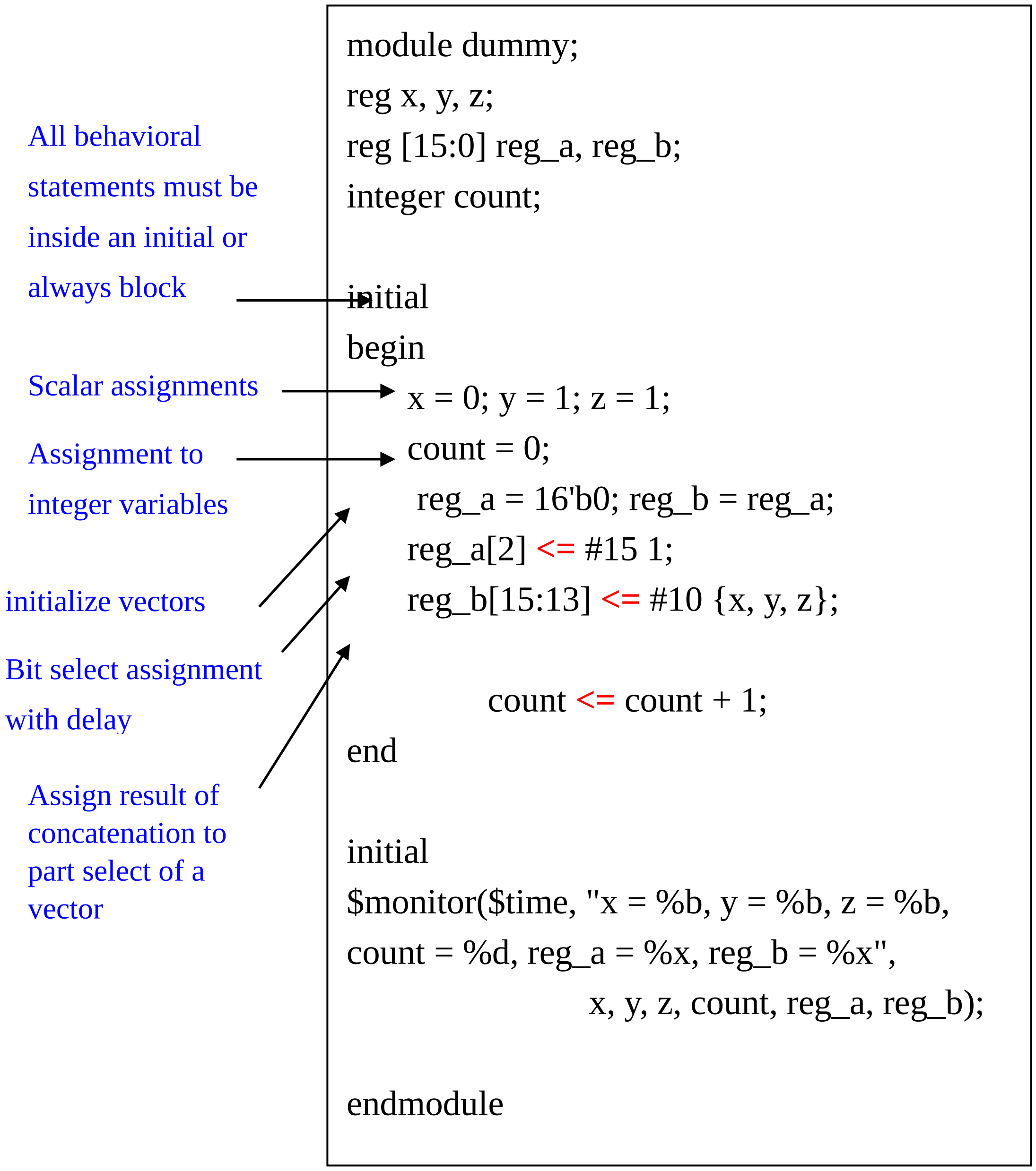
```
    Simulation stopped at the end of time 0.
Ready: sim
   15 x = 0, y = 1, z = 1, count = 0, reg_a = 0004, reg_b = 0000
|  25 x = 0, y = 1, z = 1, count = 1, reg_a = 0004, reg_b = 6000

    4 State changes on observable nets.

    Simulation stopped at the end of time 25.
Ready:
```

5.2.2 Nonblocking Assignments

Nonblocking assignment allows scheduling of assignments without blocking execution of the statements that follow in a sequential block. A <= operator is used to specify nonblocking assignments.

Ex4. Nonblocking Statements Program

Verilog Code



```
module dummy;
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;

initial
begin
    x = 0; y = 1; z = 1;
    count = 0;
     reg_a = 16'b0; reg_b = reg_a;
    reg_a[2] <= #15 1;
    reg_b[15:13] <= #10 {x, y, z};

         count <= count + 1;
end

initial
$monitor($time, "x = %b, y = %b, z = %b,
count = %d, reg_a = %x, reg_b = %x",
                x, y, z, count, reg_a, reg_b);

endmodule
```

All behavioral statements must be inside an initial or always block

Scalar assignments

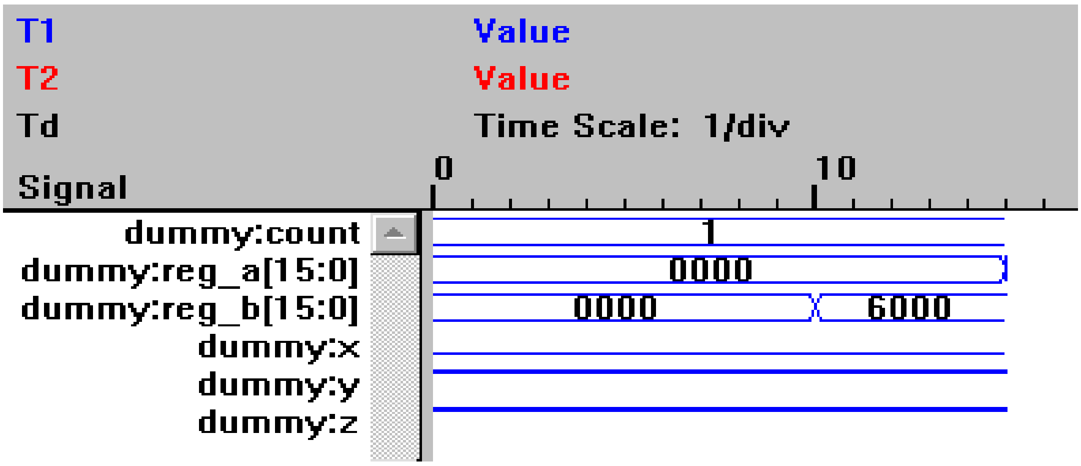
Assignment to integer variables

initialize vectors

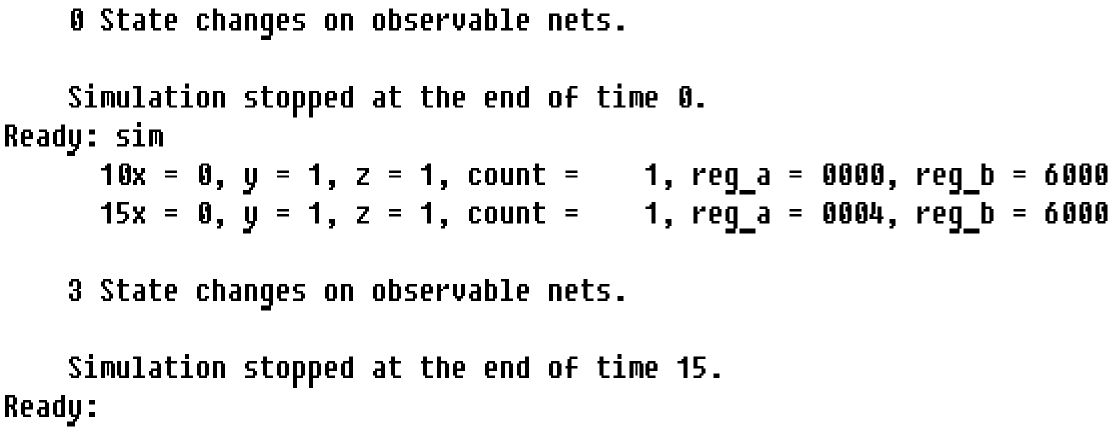
Bit select assignment with delay

Assign result of concatenation to part select of a vector

Simulation Waveform



Simulation Result



```
    0 State changes on observable nets.

    Simulation stopped at the end of time 0.
Ready: sim
      10x = 0, y = 1, z = 1, count =    1, reg_a = 0000, reg_b = 6000
      15x = 0, y = 1, z = 1, count =    1, reg_a = 0004, reg_b = 6000

    3 State changes on observable nets.

    Simulation stopped at the end of time 15.
Ready:
```

- Procedural assignments are for updating reg, integer, time, and memory variables (left -hand side).
- The left-hand side of a procedural assignment can take one of the following forms:

  1. **register, integer, real** or **time variable**;
  2. **bit-select** of a register, integer, real, or time variable
  3. **part-select** of a register, integer, real of time variable
  4. **memory element**
  5. **concatenation** of any of the above

- Assignment to a register does not sign-extend.
- The Verilog HDL contains two type of procedural assignment statements:

  1. **blocking** procedural assignment statements

  2. **non-blocking** procedural assignment statements

5.3 Timeing Controls

Various behavioral timing control constructs are available in Verilog. In Verilog, if there is no timing control statements, the simulation time does not advance. Timing controls provide a way to specify the simulation time at witch procedural statements will execute.

5.3.1 Delay-Based Timing Control

Delay-based timing control in an expression specifies the time duration between when the statement is encountered and when it is executed. We used delay-based timing control statements when writing few modules in the preceding chapters but did not explain them in detail.

```
< delay >
: : = # < NUMBER >
| | = # < identifier >
| | = # (< mintypmax_expression > < , <mintypmax_expression >> * )
```

Regular delay control

Regular delay control is used when a non-zero delay is specified to the left of a procedural assignment.

Ex5. Regular Delay Control Program

Verilog Code

define parameters

define register variables

no delay control

```
module regular_delay;

parameter latency = 20;
parameter delta = 2;
reg x, y, z, p, q;

initial
begin
        x = 0;
    #10 y = 1;   //Delay execution of y=1 by #10
```

Delay control with identifer. Delay of 20 units

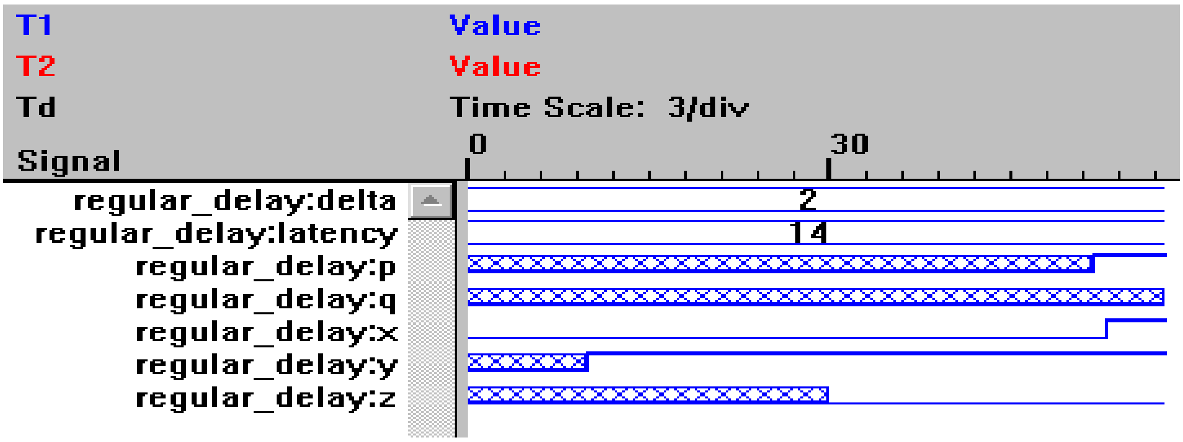
Delay control with expression

Delay control with identifier. Take value of y.

Minimum, typical and maximum delay values.

```
    #latency z = 0;

    #(latency + delta) p = 1;

     #y x = x + 1;

     #(4:5:6) q = 0;
     end

endmodule
```

Simulation Waveform



Intra-assignment delay control

Instead of specifying delay control to the left of the assignment, it is possible to assign a delay to the right of the assignment operator. Such delay specification alters the flow of activity in a different manner.

Example Intra-assignment Delays

```
//define register variables
reg   x,y,z ;
```

```
//intra assignment delays
initial
begin
      x = 0 ; z = 0 ;    #5 y= x+z;
      y = #5 x + z ; //Take value of x and z at the time = 0, evaluate x + z and then
//wait 5 time units to assign value to y .
end
```

Zero delay control

Procedural statements in different always-initial block may be evaluated at the same simulation time. Zero delay control is a method to ensure that a statement is executed last, after all other statements in that simulation time are executed.

Example Zero Delays

```
initial
begin
      x = 0 ;
      y = 0 ;
end

initial
begin
      #0    x = 1 ; //zero delay control
      #0    y = 1 ;
end
```

5.3.2 **Event-Based** Timing Control

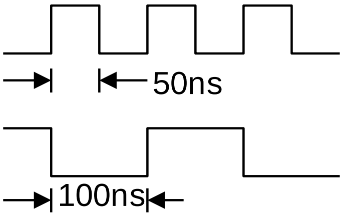
An event is the change in the value on a register or a net. Events can be utilized to trigger execution of a statement or a block of statements.

Regular event control

The **@** symbol is used to specify an event control. Statements can be executed on changes in signal value or at a positive or negative transition of the signal value.

Example 8-1 : Simple example of behavioral modeling

```
module behave ;
      reg [1 : 0 ] a , b ;
      initial
             begin
                  a = ‘b1 ;
                  b = ‘b0 ;
             end
      always
             begin
                    #50    b = ~b ;
                    #100    a = ~a ;
             end
endmodule
```



Named event control

Verilog provides the capability to declare an event and then trigger and recognize the occurrence of that event.

Ex6. Event OR control Program

Verilog Code

A level sensitive latch with asynchronous reset

Wait for reset or clock or d to change

if reset signal is high, set q to 0.

if clock is high, latch input

```
module latch(Q, d, clock, reset);

  output Q;
  input d, clock, reset;
  reg q;

  assign Q = q;

  always @( reset or clock or d)
begin
     if (reset)
                    q = 1'b0;
     else           if (clock)    q = d;

end

endmodule
```

Test Stimulus Code

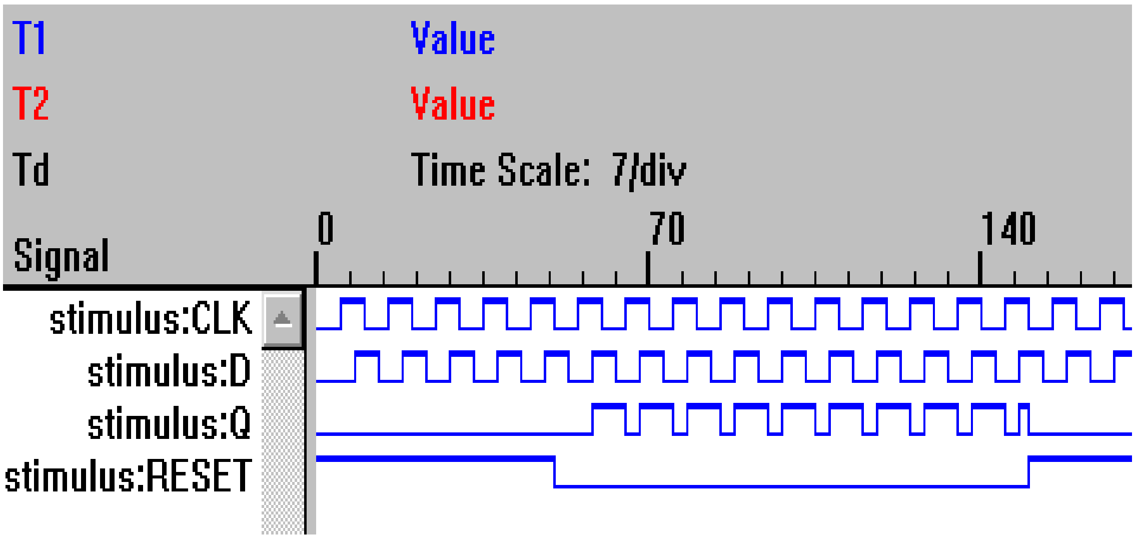
Set test value

```
module stimulus;
wire Q;
reg D, CLK, RESET;
initial
  begin
     CLK = 0;
     RESET = 1;
     #50 RESET = 0;
     #100 RESET = 1;
     #100 RESET = 0;
     #1000 $finish;
  end
```

Call latch model

```
initial
   begin
     D = 0;
     #3 forever #5 D = ~D;
   end
always #5 CLK = ~CLK;
latch l1(Q, D, CLK, RESET);

endmodule
```

Simulation Waveform



5.3.3 **Level-Sensitive** Timing Control

The symbol @ provided edge-sensitive control. Verilog also allows level-sensitive timing control, that is, the ability to wait for a certain condition to be true before a statement or a block of statements is executed.

```
always
      wait       (count_enable ) #20    count = count + 1 ;
```

5.4 Conditional Statements

Conditional statements are used for making decisions based upon certain conditions. These conditions are used to decide whether or not a statement should be executes. Keywords **if** and **else** are used for conditional statements.

```
// type 1 conditional statement . No else statement .
if ( < expression > )       true_statement ;

//type 2 conditional statement . One else statement
if ( < expression > )       true_statement ; else false_statement ;

//type3 conditional statement . Nested if-else-if
if      ( < expression 1> )     true_statement1 ;
else if ( < expression 2 > )    true_statement2 ;
else if ( < expression 3 > )    true_statement3 ;
else    defoult_statement ;
```

Example Conditional Statement

```
//Type 1 statements
if (!lock) buffer = data ;
if (enable)      out = in ;



//Type 2 statements
if (number_queued < MAX_Q_DEPTH)
   begin
      data_queued = data ;
      number_queued = number_queued + 1 ;
   end
else
      $display(“Queue Full . Try again “);
```

```
//Type 3 statements
if (alu_control == 0 )
        y = x + z ;
else if (alu_control == 1 )
        y = x – z ;
else if (alu_control == 2 )
        y = x * z ;
else
        $display(“Invalid ALU control signal “);
```

Example 8-9 : Association of else in nested if

```
if (index > 0)
          if (rega > regb)
                    result = rega ;
          else                                //else applies to preceding if
                  result = regb ;
```

Example 8-10 : Forcing correct association of else with if

```
if (index > 0)
        begin
           if (rega > regb )
             result = rega ;
        end
else
        result = regb ;
```

Example 8-11 : **Erroneous** association of else with if

```
if (index > 0)
   for (scani = 0; scani <index ; scani = scani +1)
      if (memory[scani] > 0)
         begin
            $display(“ ·  ·  · ”) ;
            memory [scani] = 0
         end
      else /* WRONG */
          $display (“error - index is zero”) ;
```

Example 8-12 : Use of semicolon in if statement

```
if (rega > regb )
    result = rega ;
else
    result = regb ;
```

- ## Example 8-13 : Use of if-else-if construct

```
//Declare registers and parameters
reg [31 : 0] instruction, segment_area [255 :0 ];
reg [7 : 0 ] index ;
reg [5 : 0 ]modify_seg1,
    modify_seg2,
    modify_seg3;
parameter
    segment1 = 0,      inc_seg1=1,
    segment2 = 20,     inc_seg2=2,
    segment3 = 64,     inc_seg3=4,
    data = 128 ;
//Test the index variable
if (index < segment2 )
```

```
      begin
              instruction = segment_area [index + modify_seg1] ;
              index = index +inc_seg1 ;
      end
else if (index <segment3 )
      begin
              instruction = segment_area [index +modify_seg2] ;
              index = index +inc_seg2 ;
      end
else if (index < data )
      begin
            instruction = segment_area [index +modify_seg3 ] ;
            index = index + inc_seg3 ;
      end
else
      instruction = segment_area [index ] ;
```

5.5 Multiway Branching

In nested if-else-if Conditional statements, there were many alternatives, from which one was chosen. Like if-else-if can become unwieldy if there are too many alternatives. A shortcut to achieve the same result is to use the **case** statement.

```
case      ( expression )
      alternative1 : statement1 ;
      alternative2 : statement2 ;
      alternative3 : statement3 ;
            ….
            ….
      default : default_statement ;
endcase
```

Ex7. 4_to_1 Multiplexer

Verilog Code

the I/O diagram.

Port declarations from the I/O

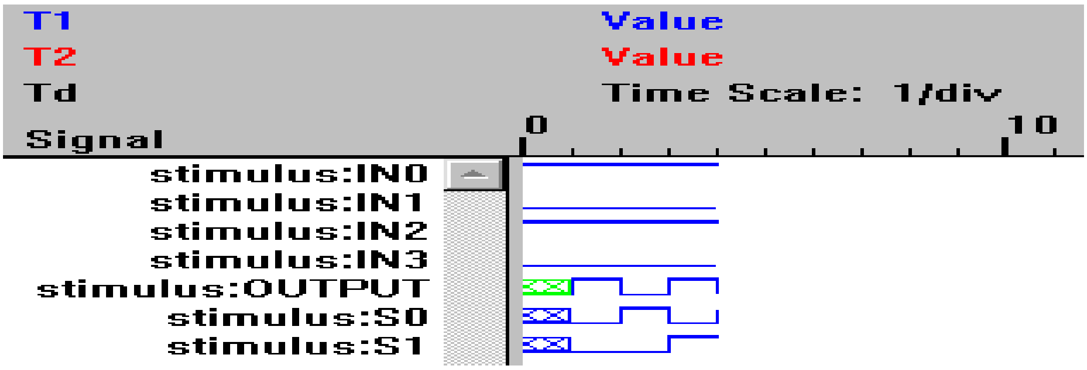
```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

output out;
```

output declared
as register

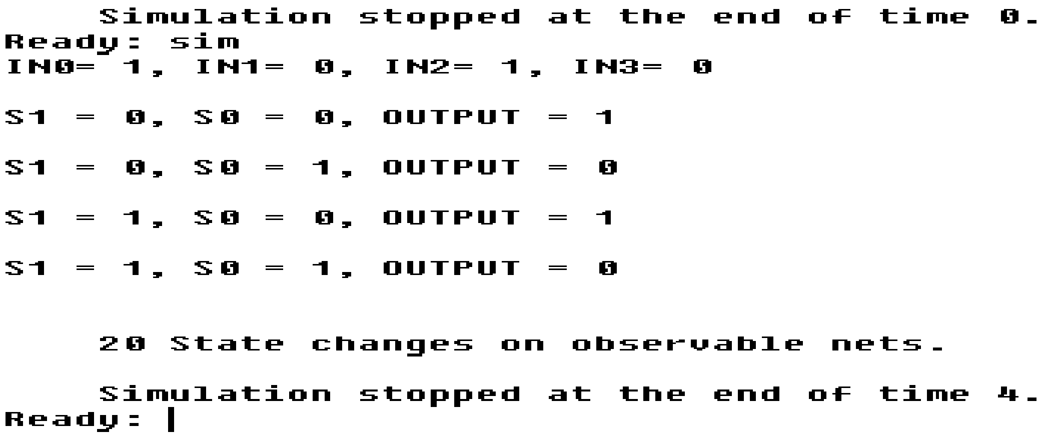
If input has any
change then run
case statement

```
input i0, i1, i2, i3;
input s1, s0;
reg out;

always @(s1 or s0 or i0 or i1 or i2 or i3)
begin
  case ({s1, s0})
        2'b00: out = i0;
        2'b01: out = i1;
        2'b10: out = i2;
        2'b11: out = i3;
        default: out = 1'bx;
  endcase
end

endmodule
```

Simulation Waveform



Simulation Result



```
        Simulation stopped at the end of time 0.
Ready: sim
IN0= 1, IN1= 0, IN2= 1, IN3= 0

S1  = 0, S0  = 0, OUTPUT  = 1

S1  = 0, S0  = 1, OUTPUT  = 0

S1  = 1, S0  = 0, OUTPUT  = 1

S1  = 1, S0  = 1, OUTPUT  = 0


        20 State changes on observable nets.

        Simulation stopped at the end of time 4.
Ready: |
```

Example 8-14 : Use of the case statement

```
reg [15 : 0] rega ;

reg [ 9 : 0 ] result ;
            .
            .
case (rega)
        16’d0 : result = 10’b0111111111;
        16’d1 : result = 10’b1011111111;
        16’d2 : result = 10’b1101111111;
        16’d3 : result = 10’b1110111111;
        16’d4 : result = 10’b1111011111;
        16’d5 : result = 10’b1111101111;
        16’d6 : result = 10’b1111110111;
        16’d7 : result = 10’b1111111011;
        16’d8 : result = 10’b1111111101;
        16’d9 : result = 10’b1111111110;
        default result = ‘bx ;
endcase
```

- Example 8-15 : Detecting x and z values with the case statement

```
case (select [ 1:2 ] )
      2’b00 : result = 0 ;
      2’b01 : result = flaga ;
      2’b0x,
      2’b0z : result = flaga ? ‘bx : 0 ;
      2’b10 : result = flagb ;
      2’bx0,
      2’bz0 : result = flagb ? ‘bx : 0;
      default result = ‘bx;
endcase
```

- Example 8-16 : Another example of detecting x and z with case

```
case (sig)
      1'bz :
                  $display ("signal is floating") ;
      1'bx :
                  $display ("signal is unknown") ;
      default :
                  $display ("signal is %b", sig);
endcase
```

8.4.1 Case statement with Don't-Cares

Example 8-17 : Using the casez statement

```
reg [7 : 0 ] ir ;
              •
              •
              •
casez    (ir)
      8'b1??????? : instruction1(ir);
      8'b01?????? : instruction2(ir);
      8'b00010??? : instruction3(ir);
      8'b000001?? : instruction4(ir);
endcase
```

Example 8-18 : Using the casex statement

```
reg [7 : 0 ]    r, mask;
             •
             •
             •
mask = 8'bx0x0x0x0 ;
```

```
casex    (r ^ mask)
     8'b001100xx : stat1 ;
     8'b1100xx00 : stat2 ;
     8'b00xx0011 : stat3 ;
     8'bxx001100 : stat4 ;

endcase
```

casez: allows for z values to be treated as don't-care values.

Casex: aloows for both z and x to be treated as don't-care.

Example:

```
module decode;
     reg [7:0]      r;
     always begin        //other statements
     r=8'bx1x0x1x0;
     casex (r)
          8'b001100xx   statement1;
          8'b1100xx00   statement2;
          8'b00xx0011   statement3;
          8'bxx001100   statement4;
     endcase;
endmodule
```

5.6 Loops

There are four types of looping statements in Verilog : **while**, **for**, **repeat**, and **forever**. All looping statements can appear only inside an initial or always block. Loops may contain delay expression.

5.6.1 While Loop

The while loop executes until the while-expression becomes false. If the loop is entered when the while-expression is false, the loop is not executed at all.

Ex8. Counter (while loop)

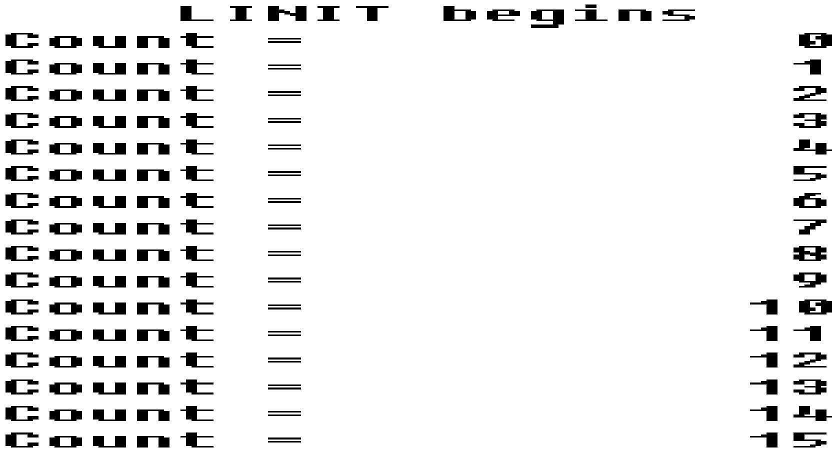
Verilog Code

Exit at count 16.
Display the count
variable.

Execute loop till
count is 15.

```
module count_mod;

integer count;

initial
begin
            count = 0;
            while (count < 16)

            begin
              $display(" Count = %d", count);
                  count = count + 1;
            end
end

endmodule
```

Simulation Result



```
      LINIT begins
Count =              0
Count =              1
Count =              2
Count =              3
Count =              4
Count =              5
Count =              6
Count =              7
Count =              8
Count =              9
Count =             10
Count =             11
Count =             12
Count =             13
Count =             14
Count =             15
```

5.6.2 For Loop

The keyword ***for*** is used to specify this loop. The ***for*** loop contains three parts.

- An initial condition
- A check to see if the terminating condition is true
- A procedural assignment to change value of the control variable

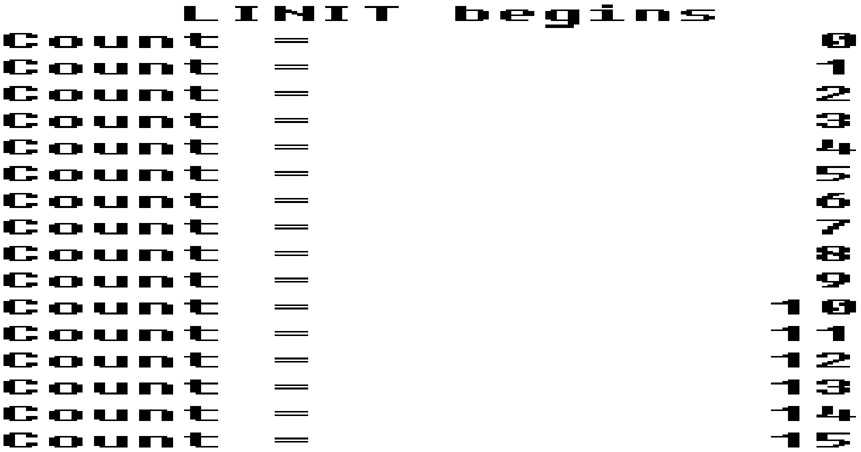
Ex9. Counter (for loop)

Verilog Code

For loop statement →

```
module counter;
integer count;
initial
      for ( count=0; count < 16; count = count + 1)
                  $display("Count = %d", count);
endmodule
```

Simulation Result



```
      LINIT begins
Count =           0
Count =           1
Count =           2
Count =           3
Count =           4
Count =           5
Count =           6
Count =           7
Count =           8
Count =           9
Count =          10
Count =          11
Count =          12
Count =          13
Count =          14
Count =          15
```

5.6.3 Repeat Loop

The keyword repeat is used for this loop . The repeat construct executes the loop a fixed number of times . A repeat construct cannot be used to loop on a general logical expression .

Ex10. Counter (repeat loop)

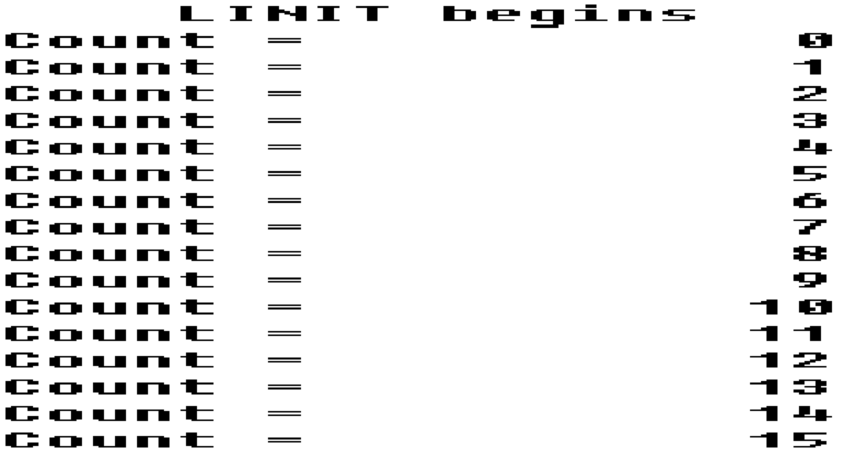
Verilog Code

increment and display count from 0 to 16

Repeat statement let count from 0 to 16

```
module counter;
integer count;

initial
begin
     count = 0;
     repeat (16)
     begin
             $display("Count = %d", count);
             count = count + 1;
     end
end


endmodule
```

Simulation Result



```
      LINIT begins
Count =                 0
Count =                 1
Count =                 2
Count =                 3
Count =                 4
Count =                 5
Count =                 6
Count =                 7
Count =                 8
Count =                 9
Count =                10
Count =                11
Count =                12
Count =                13
Count =                14
Count =                15
```

5.6.4 Forever Loop

The keyword forever is used to express this loop. The loop does not contain any expression and executes forever until the $finish task is encountered. The loop is equivalent to a while loop with an expression that always evaluates to true.

Verilog Code

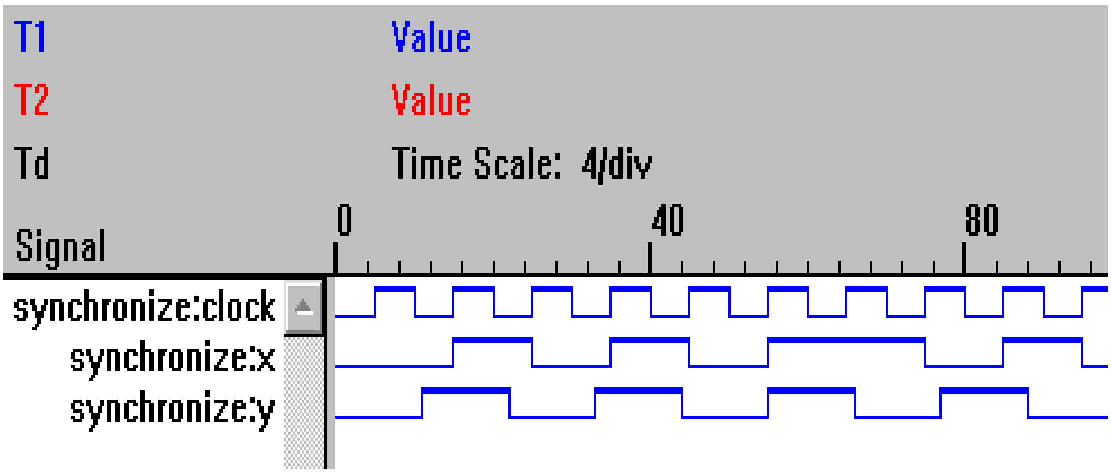
Synchronize two register values at every positive edge of clock

```
module synchronize;

reg clock;
reg x, y;

initial
begin
          clock = 1'b0;
          x = 1'b0;
          y = 1'b0;
          #100000 $finish;
end
```

Clock generator

```
always #5 clock = ~clock;
always #11 y = ~y;
initial
            forever @(posedge clock) x = y;

endmodule
```

Simulation Waveform



5.7 Sequential and Parallel Blocks

The statements in a **sequential** block are processed in the order they are specified. A statement is executed only after the preceding statement completes execution. If delay or event control is specified, it is relative to the simulation time when the previous statement in the block completes execution.

Ex12. Sequential block

Verilog Code

```
module sequential;

reg x, y;
reg [1:0] z, w;
```

```
initial
        $monitor($time, " x = %b, y = %b, z =
%b, w = %b\n", x, y, z, w);
initial
      begin
            x = 1'b0;
            #5 y = 1'b1;

            #10 z = {x, y};

            #20 w = {y, x};
      end
endmodule
```

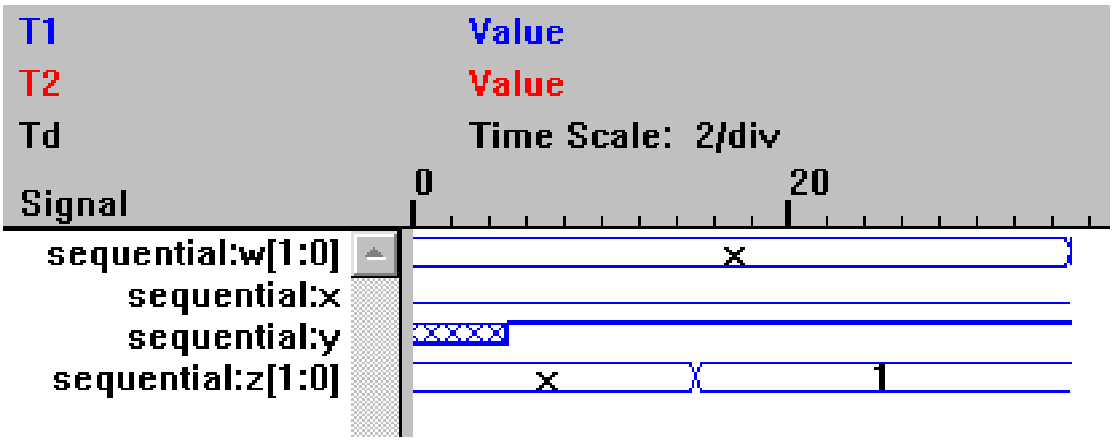
completes at simulation time 0

completes at simulation time 5

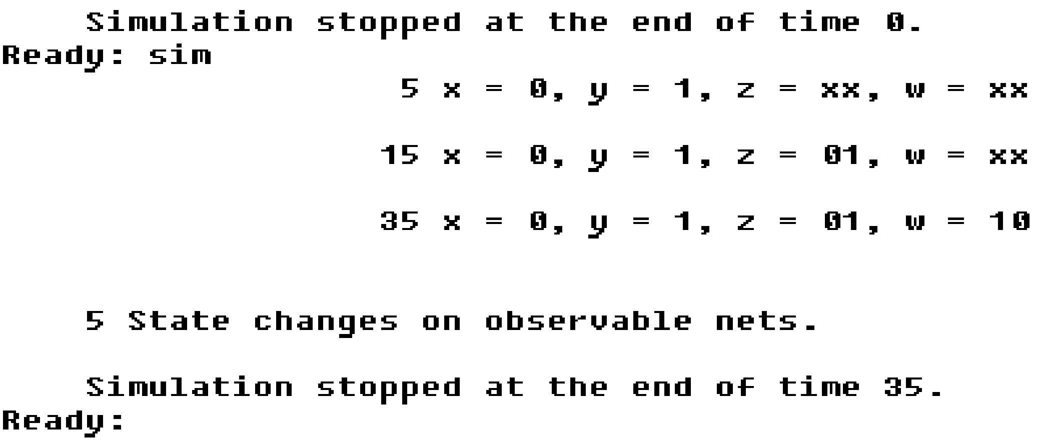
completes at simulation time 15

completes at simulation time 35

Simulation Waveform



Simulation Result



```
    Simulation stopped at the end of time 0.
Ready: sim
                   5 x = 0, y = 1, z = xx, w = xx

                  15 x = 0, y = 1, z = 01, w = xx

                  35 x = 0, y = 1, z = 01, w = 10


    5 State changes on observable nets.

    Simulation stopped at the end of time 35.
Ready:
```

The statements in a **parallel** block are executed concurrently. Ordering of statements is controlled by the delay or event control assigned to each statement. If delay or event control is specified, it is relative to the time the block was entered.

Ex13. Parallel block

Verilog Code

```
module parallel;

reg x, y;
reg [1:0] z, w;

initial
        $monitor($time, " x = %b, y = %b,
z = %b, w = %b\n", x, y, z, w);

initial
fork
        x = 1'b0;
       #5 y = 1'b1;

       #10 z = {x, y};
       #10 z = {x, y};
       #20 w = {y, x};
join
endmodule
```

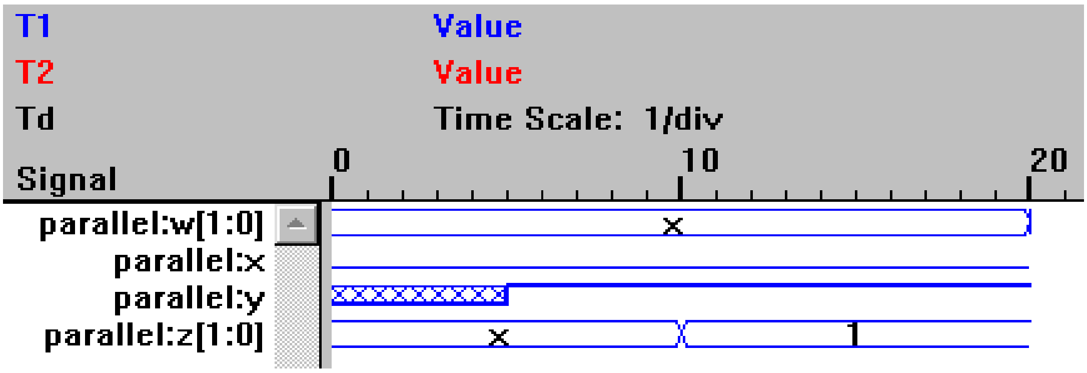
completes at
simulation time 0

completes at
simulation time 5

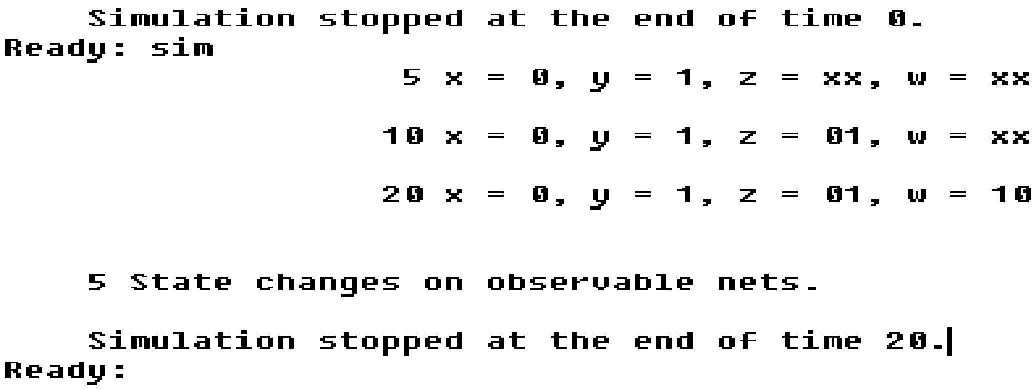
completes at
simulation time 15

completes at
simulation time 35

Simulation Waveform



Simulation Result



```
    Simulation stopped at the end of time 0.
Ready: sim
                         5 x = 0, y = 1, z = xx, w = xx

                        10 x = 0, y = 1, z = 01, w = xx

                        20 x = 0, y = 1, z = 01, w = 10


    5 State changes on observable nets.

    Simulation stopped at the end of time 20.
Ready:
```

5.8 Examples

- **Behavioral 4-to-1 Multiplexer**

Verilog Code

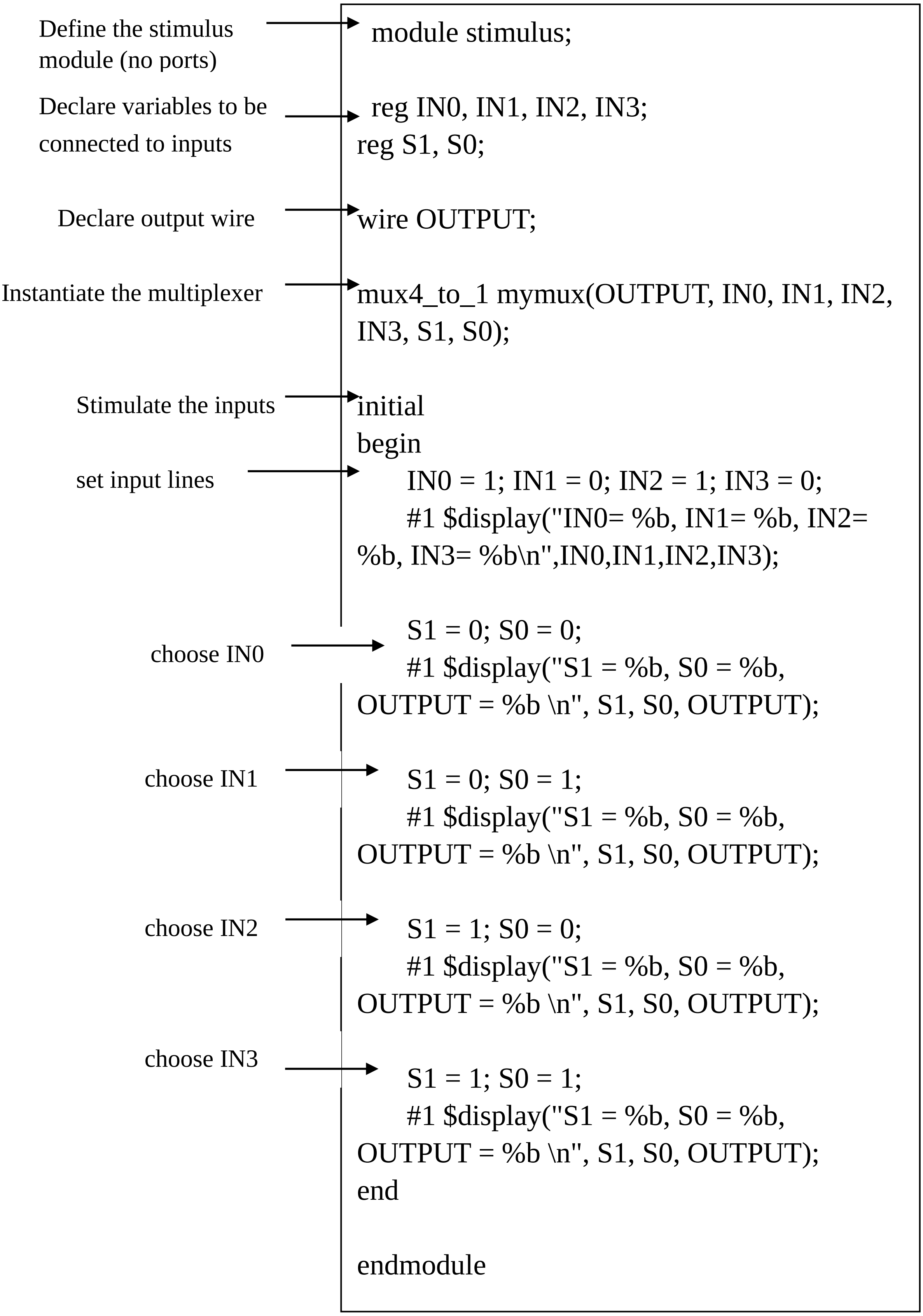
the I/O diagram.

Port declarations from the I/O diagram

output declared as register

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

output out;
input i0, i1, i2, i3;
input s1, s0;

reg out;

always @(s1 or s0 or i0 or i1 or i2 or i3)
begin
            case ({s1, s0})
            2'b00: out = i0;
            2'b01: out = i1;
            2'b10: out = i2;
            2'b11: out = i3;
            default: out = 1'bx;
            endcase
end
endmodule
```

Test Stimulus Code



Define the stimulus module (no ports)

Declare variables to be connected to inputs

Declare output wire

Instantiate the multiplexer

Stimulate the inputs

set input lines

choose IN0

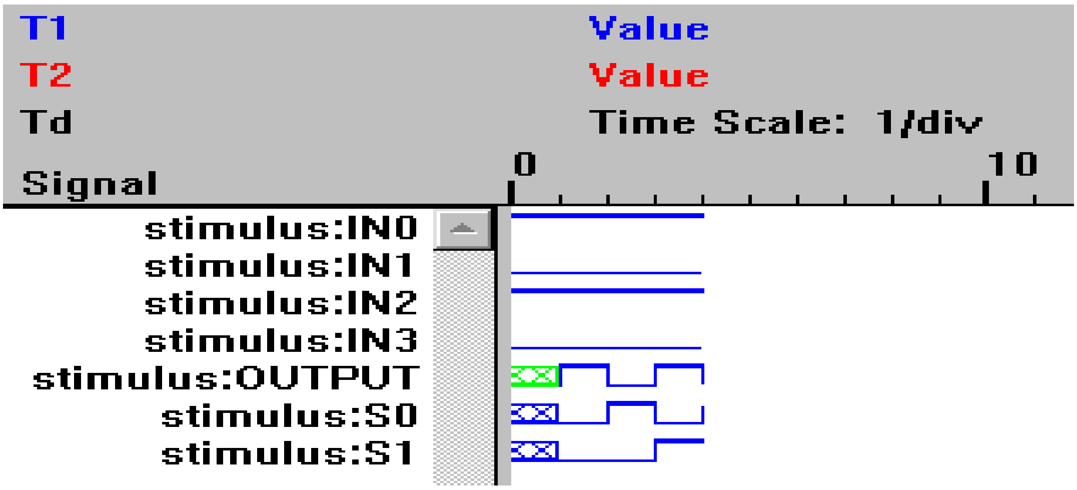
choose IN1

choose IN2

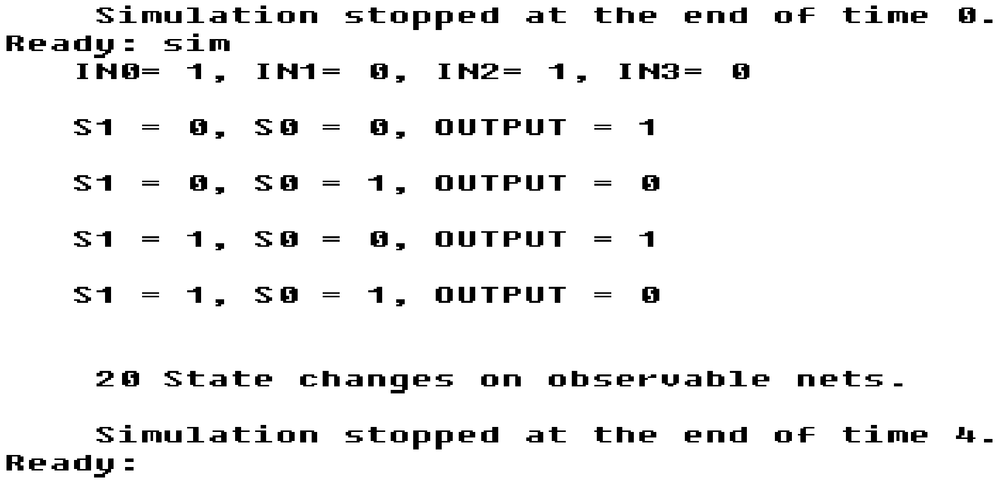
choose IN3

```
 module stimulus;

 reg IN0, IN1, IN2, IN3;
reg S1, S0;

wire OUTPUT;

mux4_to_1 mymux(OUTPUT, IN0, IN1, IN2,
IN3, S1, S0);

initial
begin
    IN0 = 1; IN1 = 0; IN2 = 1; IN3 = 0;
    #1 $display("IN0= %b, IN1= %b, IN2=
%b, IN3= %b\n",IN0,IN1,IN2,IN3);

    S1 = 0; S0 = 0;
    #1 $display("S1 = %b, S0 = %b,
OUTPUT = %b \n", S1, S0, OUTPUT);

    S1 = 0; S0 = 1;
    #1 $display("S1 = %b, S0 = %b,
OUTPUT = %b \n", S1, S0, OUTPUT);

    S1 = 1; S0 = 0;
    #1 $display("S1 = %b, S0 = %b,
OUTPUT = %b \n", S1, S0, OUTPUT);

    S1 = 1; S0 = 1;
    #1 $display("S1 = %b, S0 = %b,
OUTPUT = %b \n", S1, S0, OUTPUT);
end

endmodule
```

Simulation Waveform



Simulation Result



```
    Simulation stopped at the end of time 0.
Ready: sim
   IN0= 1, IN1= 0, IN2= 1, IN3= 0

   S1 = 0, S0 = 0, OUTPUT = 1

   S1 = 0, S0 = 1, OUTPUT = 0

   S1 = 1, S0 = 0, OUTPUT = 1

   S1 = 1, S0 = 1, OUTPUT = 0


    20 State changes on observable nets.

    Simulation stopped at the end of time 4.
Ready:
```

- **Behavioral 1-to-4 Demultiplexer**

Verilog Code

Port declarations from the I/O →

```
module demultiplexer1_to_4 (out0, out1, out2,
out3, in, s1, s0);

output out0, out1, out2, out3;
reg   out0,   out1,   out2,   out3;
input in;
```

Switch based on control signals

Account for unknown signals on select. If any select signal is x then outputs are x. If any select signal is z, outputs are z. If one is x and the other is z, x gets higher priority.

```
input s1, s0;
always @(s1 or s0 or in)

case ({s1, s0})
      2'b00 :   begin   out0 = in;   out1 = 1'bz;   out2
      = 1'bz;   out3 = 1'bz; end

       2'b01 :   begin   out0 = 1'bz;   out1 = in;   out2
      = 1'bz;   out3 = 1'bz; end

      2'bx0, 2'bx1, 2'bxz, 2'bxx, 2'b0x, 2'b1x, 2'bzx :
                begin
                         out0 = 1'bx;   out1 = 1'bx;   out2 =
                        1'bx;   out3 = 1'bx;
                end

       2'bz0, 2'bz1, 2'bzz, 2'b0z, 2'b1z :
              begin
                          out0 = 1'bz;   out1 = 1'bz;   out2
                        = 1'bz;   out3 = 1'bz;
                end

     default: $display("Unspecified control signals");

endcase

endmodule
```

Test Stimulus

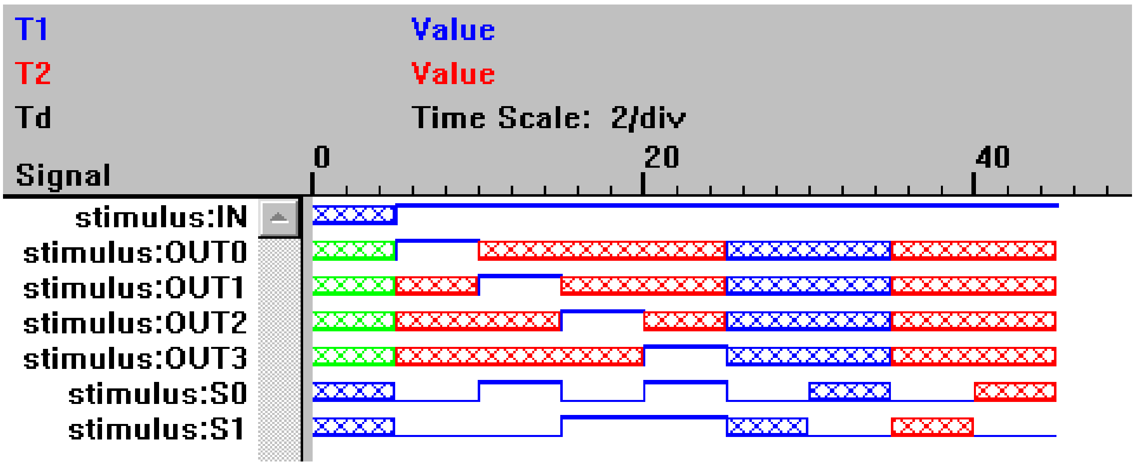
instantiate the decoder

```
module stimulus;
wire OUT0, OUT1, OUT2, OUT3;
reg IN, S1, S0;

demultiplexer1_to_4 dm0( OUT0, OUT1, OUT2,
OUT3,   IN, S1, S0);
initial
```

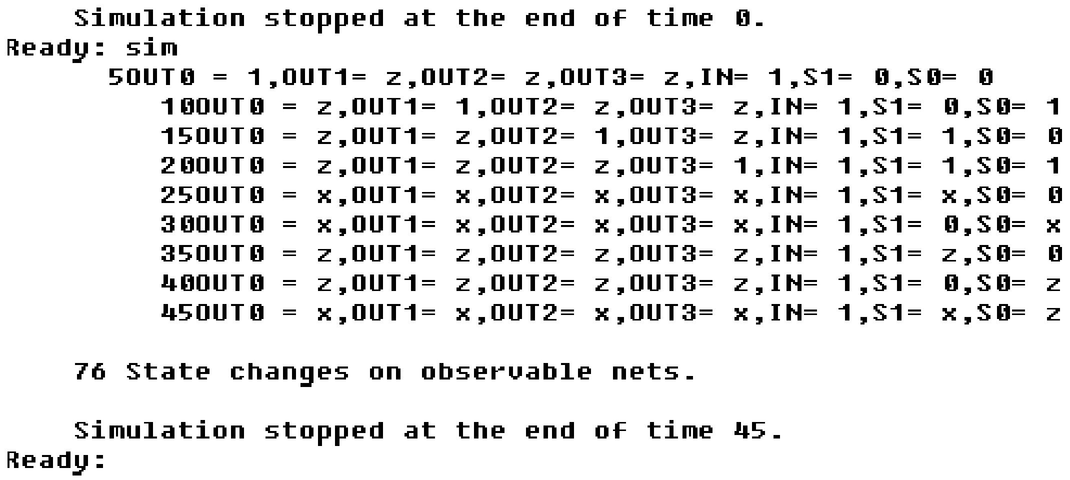
Set test value

```
$monitor($time,"OUT0 = %b,OUT1=
%b,OUT2= %b,OUT3= %b,IN= %b,S1=
%b,S0= %b ",OUT0, OUT1, OUT2, OUT3,
IN, S1, S0);

initial
begin
     #5  IN = 1;  S1 = 0; S0 = 0;
     #5  IN = 1;  S1 = 0; S0 = 1;
     #5  IN = 1;  S1 = 1; S0 = 0;
     #5  IN = 1;  S1 = 0; S0 = 1'bx;
     #5  IN = 1;  S1 = 1'bz; S0 = 0;
     #5  IN = 1;  S1 = 0; S0 = 1'bz;
     #5  IN = 1;  S1 = 1'bx; S0 = 1'bz;
end

endmodule
```

Simulation Waveform



Simulation Result

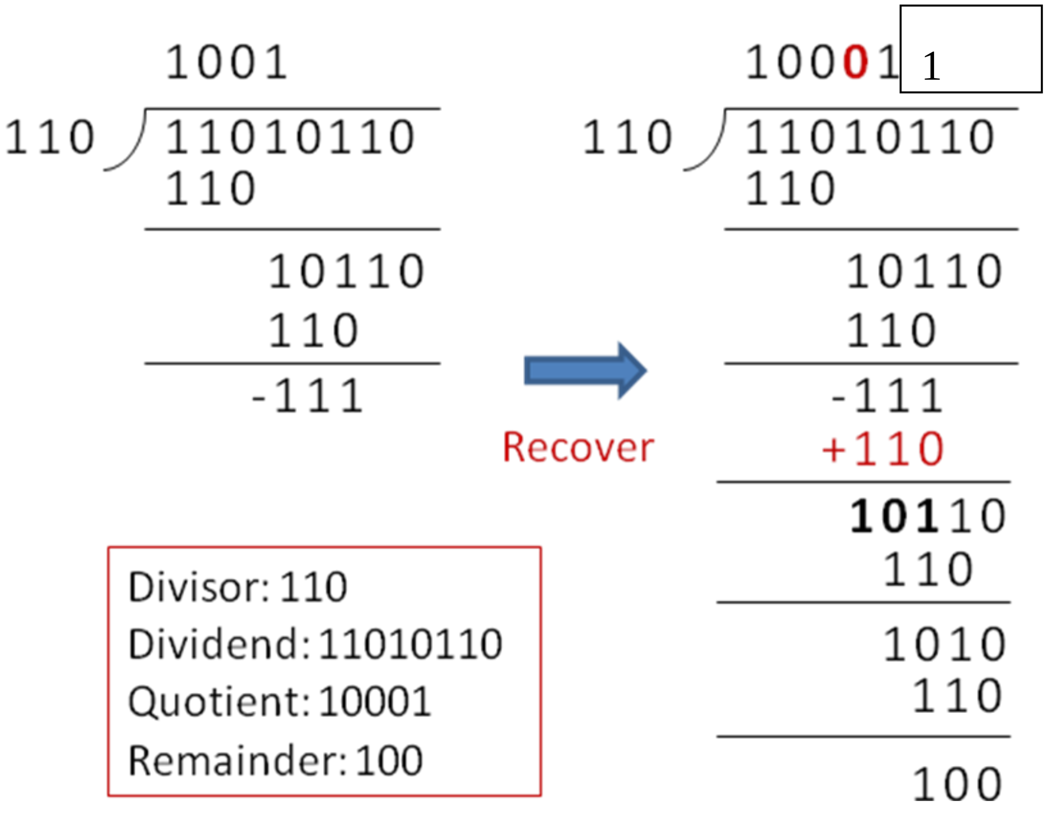


```
    Simulation stopped at the end of time 0.
Ready: sim
      5OUT0 = 1,OUT1= z,OUT2= z,OUT3= z,IN= 1,S1= 0,S0= 0
         10OUT0 = z,OUT1= 1,OUT2= z,OUT3= z,IN= 1,S1= 0,S0= 1
         15OUT0 = z,OUT1= z,OUT2= 1,OUT3= z,IN= 1,S1= 1,S0= 0
         20OUT0 = z,OUT1= z,OUT2= z,OUT3= 1,IN= 1,S1= 1,S0= 1
         25OUT0 = x,OUT1= x,OUT2= x,OUT3= x,IN= 1,S1= x,S0= 0
         30OUT0 = x,OUT1= x,OUT2= x,OUT3= x,IN= 1,S1= 0,S0= x
         35OUT0 = z,OUT1= z,OUT2= z,OUT3= z,IN= 1,S1= z,S0= 0
         40OUT0 = z,OUT1= z,OUT2= z,OUT3= z,IN= 1,S1= 0,S0= z
         45OUT0 = x,OUT1= x,OUT2= x,OUT3= x,IN= 1,S1= x,S0= z

    76 State changes on observable nets.

    Simulation stopped at the end of time 45.
Ready:
```

Divider Behavior

Dividend: 11010110 = 214, Divisor: 110 = 6, 214/6 = 35 ……4

Dividend: 11010110 = -86, Divisor: 0110 = 6, -86/6 = -14 …. -2



Divisor: 110
Dividend: 11010110
Quotient: 10001
Remainder: 100

```
`define DvLen 15
`define DdLen 31
`define QLen 15
`define HiDdMin 16
module divide (ddInput, dvInput, quotient, go, done);
    input [`DdLen:0] ddInput;     // [31:0]
    input [`DvLen:0] dvInput;     // [15:0]
    output [`QLen:0] quotient;    // [15:0]
    input go;                     // Start division
    output done;                  // Finish division
    reg [DdLen:0] dividend;
    reg [QLen:0] quotient;
    reg [DvLen:0] divisor;
    reg done, negDivisor, negDividend;
    always begin
        done = 0;
        wait (go);
        divisor = dvInput;
        dividend = ddInput;
        quotient = 0;
        if (divisor !== 0) begin
            negDivisor = divisor[DvLen];        // divisor[15]
            if (negDivisor) divisor = - divisor;
            divisor = divisor << 1;
            negDividend = dividend[DdLen]; // dividend[31]
            if (negDividend) dividend = - dividend;
            repeat (`DvLen + 1) begin
                quotient = quotient << 1;
                dividend = dividend << 1;
                dividend[`DdLen:`HiDdMin] =
                dividend[`DdLen:`HiDdMin] - divisor;

                if (! dividend [ `DdLen]) quotient = quotient + 1;
                        else
                        dividend[`DdLen:`HiDdMin] =
                                dividend[`DdLen:`HiDdMin] + divisor;
            end
            if (negDivisor != negDividend) quotient = - quotient;
```

```
          end
      done = 1;
      wait (~go);
      end
endmodule
```

- Example 8-36 : Behavioral model of traffic light sequencer

```
module traffic_lights;
          reg    clock, red, amber, green;
          parameter
                    on = 1,    off = 0,
                    red_tics = 350,    amber_tics = 30,    green_tics = 300;
          //the sequence to control the lights
          always
                begin
                    red = on;     amber = off;          green = off;
                    repeat (red_tics) @(posedge clock);
                    red = off;          green = on;
                    repeat (green_tics) @(posedge clock);
                    green = off;      amber = on;
                    repeat (amber_tics) @(posedge clock);
                 end
          //waveform for the clock
          always
                begin
                    #100 clock = 0 ;
                    #100 clock = 1 ;
                end
          //simulate for 10 changes on the red light
          initial
              begin
                    repeat (10) @red;
                    $finish
              end
```

```
    //display the time and changes made to the lights

        always @(red or amber or green)
                $display("%d red = %b amber=%b green=%b",
                         $time, red, amber, green);
    endmodule
```

Example 8-37 : Behavioral model with variable delays

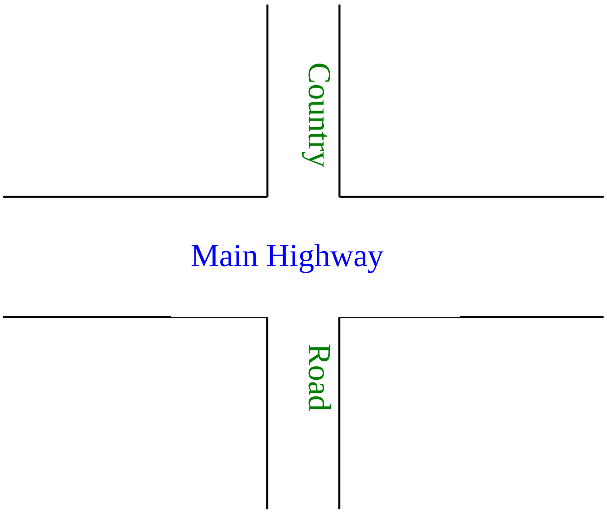
```
module synch_clocks;
    reg    clock, phase1, phase2;
    time   clock_time;

    initial clock_time = 0;
    always @(posedge clock)      // $time = #100 u.t.
          begin :phase_gen
                 time d;    // a local declaration is possible
                          // because the block is named
                 d = ($time - clock_time) / 8;      // d = 12.5 u.t.
                 clock_time = $time;
                 phase1 = 0 ;              // $time = #100 u.t.
                 #d phase2 = 1;            // $time = #112.5 u.t.
                 #d phase1 = 1;            // $time = #125 u.t.
                 #d phase2 = 0;
                 #d phase1 = 0;
                 #d phase2 = 1;
                  #d phase1 = 1;
                 #d phase2 = 0;            // $time = #187.5 u.t.
             end
    //set up a clock waveform, finish time,
    // and display
    always
       begin
            #100 clock = 0 ;
            #100 clock = 1 ;
       end
    initial #1000 $finish; //end simulation at time 1000
    always
```

```
        @(phase1 or phase2)
                $display($time,,
                        "clock=%b phase1=%b phase2=%b,
                        clock, phase1, phase2);
endmodule
```

Ex. 7.9.3 Traffic Signal Controller

Specification:

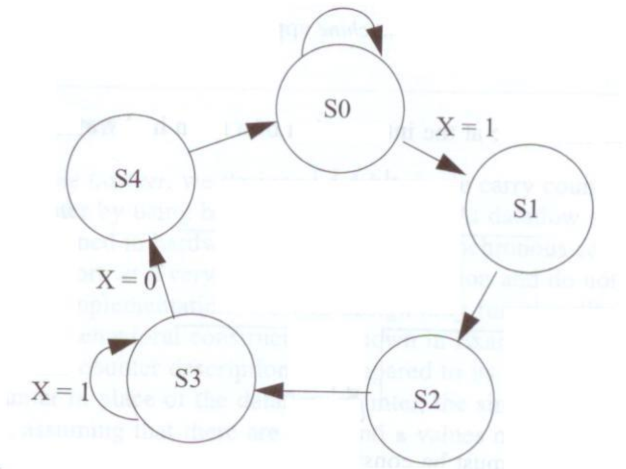
Consider a controller for traffic at the intersection of a main highway and a country road.



- The traffic signal for the main highway gets the highest priority. So, the main highway signal remains green by default.
- Occasionally, cars from the country road arrive at the traffic signal. The traffic signal for the country road must turn green only long enough to let the cars on the country road go.
- As soon as there are no cars on the country road, the country road traffic

signal turns yellow and then red.

- There is a sensor to detect cars waiting on the country road. The sensor sends a signal X as input to the controller. X=1 if there are cars on the country road; otherwise, X=0.
- There are delays on transitions from S1 to S2, from S2 to S3, and from S4 to S0. The delays must be controllable.



State	Signals
S0	*Hwy = G Cntry = R*
S1	*Hwy = Y Cntry = R*
S2	*Hwy = R Cntry = R*
S3	*Hwy = R Cntry = G*
S4	*Hwy = R Cntry = Y*

Figure 7-1 *FSM for Traffic Signal Controller*

```
`define TRUE     1'b1
`define FALSE   1'b0
//Delays
`define Y2RDELAY  3     //Yellow to red delay
`define R2GDELAY  2     //Red to Green Delay

module sig_control (hwy, cntry, X, clock, clear);
//I/O ports
output [1:0] hwy, cntry;          //2 bit output for 3 states of signal
                                  //GREEN, YELLOW, RED;
reg [1:0] hwy, cntry;             //declared output signals are registers
input X;                          //if TRUE, indicates that there is car on
                                  //the country road, otherwise FALSE
input clock, clear;
parameter     RED= 2'd0,    YELLOW = 2'd1, GREEN    = 2'd2;

//State definition               HWY                  CNTRY
parameter     S0 = 3'd0,    //GREEN                RED
              S1 = 3'd1,    //YELLOW               RED
              S2 = 3'd2,    //RED                  RED
              S3 = 3'd3,    //RED                  GREEN
              S4 = 3'd4;    //RED                  YELLOW

//Internal state variables
reg [2:0] state;              reg [2:0] next_state;

//state changes only at positive edge of clock
always @(posedge clock)
    if(clear)   state <= S0;       //Controller starts in S0 state
    else        state <= next_state;        //State change

//Compute values of main signal and country signal
always @(state)
begin
      hwy = GREEN; //Default Light Assignment for Highway light
      cntry = RED; //Default Light Assignment for Country light

     case(state)
```

```
        S0: ;                            //No change,use default
        S1:     hwy = YELLOW;
        S2:     hwy = RED;
        S3:     begin
                 hwy = RED;          cntry =   GREEN;
            end
        S4: begin
                 hwy =   RED;        cntry =   `YELLOW;
            end
        default: begin    hwy = GREEN; cntry = RED; end
    endcase
end

//State machine using case statements
always @(state or   X)
begin
        case (state)
             S0: if(X)   next_state =   S1;
                 else    next_state =   S0;

             S1: begin //delay some positive edges of clock
                         repeat(`Y2RDELAY) @(posedge clock) ;
                         next_state =   S2;
                end
             S2: begin //delay some positive edges of clock
                         repeat(`R2GDELAY) @(posedge clock)
                         next_state =   S3;
                end
             S3: if( X)  next_state =   S3;
                 else    next_state =   S4;

             S4: begin //delay some positive edges of clock
                         repeat(`Y2RDELAY) @(posedge clock) ;
                         next_state =   S0;
                             end
            default:   next_state =   S0;
        endcase
end
```

```
endmodule
```

修改

```
always @(state)
begin
       //Default Light Assignment for Highway light
       //Default Light Assignment for Country light

      case(state)
            S0:  begin    hwy = GREEN;         cntry = RED     end;
            S1: begin     hwy = YELLOW;    cntry = RED     end;
            S2: begin      hwy = RED;             cntry = RED     end;
            S3: begin      hwy = RED;             cntry = GREEN;     end
            S4:  begin      hwy =     RED;          cntry = YELLOW; end
            Default:   begin    hwy = GREEN;         cntry = RED     end;
      endcase
end
```

Basic Verilog Examples

https://web.csulb.edu/~rallison/Verilog_Examples_Table.htm

Useful verilog examples

http://jupiter.math.nctu.edu.tw/~weng/courses/IC_2007/PROJECT_MATH_CLASS1/3_DESIGN_COMPLEXITY/Verilog%20examples%20useful%20for%20FPGA%20&%20ASIC.htm

Design company

https://www.doulos.com/knowhow/verilog_designers_guide/

Design Examples

https://veriloggguide.readthedocs.io/en/latest/verilog/designs.html

UART, Serial Port, RS-232 Interface

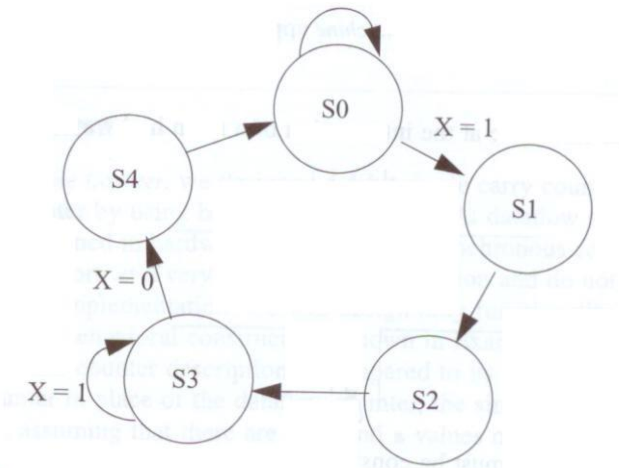
https://www.nandland.com/vhdl/modules/module-uart-serial-port-rs232.html

Homework 3: 7-segment Display with BCD converter and ALU

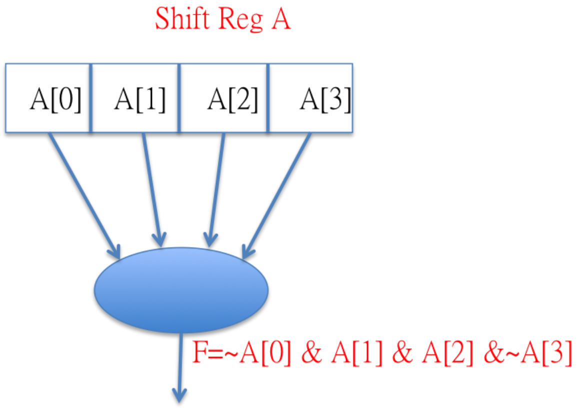
Homework 4:

Write a Verilog behavioral program for a state machine that samples a continuous stream of synchronized data on an input line X. The state machine is to an output Z any time the sequence ….0110… occurs. Consider that the sequence may be overlapping. For example,

Input: X = 0011001101100110….
Output: Z = 0000100010010001….



Special design:



Textbook Exercise p.167

2, 3, 4, 7, 8, 10, 11, 12, 14, 15, 16

7.8 Generate Blocks

(Generate loop, Generate conditional, Generate case)

a. Generate loop

```
module bitwise_xor   (out, io, i1)
    parameter   N = 32;           //32-bit bus by default
    output   [N-1:0]   out;
    input   [N-1: 0]   i0, i1;
//Declare a temporary loop variable. This variable is used only
//in the evaluation of generate blocks. This variable does not
//exist during the simulation of a Verilog design
    genavr j;
    generate for (j=0; j<N; j=j+1)   begin:   xor_loop
            xor g1 (out[j], i0[j], i1[j]);
        end   //end of the for loop inside the generate block
    endgenerate   //end of the generate block
   //As an alternate style,
   // reg [N-1:0] out;
   // generate for (j=0; j<N; j=j+1) begin: bit
   //     always @(i0[j] or i1[j]) out[j] = i0[j] ^ i1[j];
   // end
  //endgenerate
  Endmodule
```

** The relative hierarchical names of the xor gates will be

xor_loop[0].g1, …, xor_loop[31].g1.

Ex. 7-32 Generated Ripple Adder

```
module ripple_adder (co, sum, a0, a1, ci);
        // parameter declaration.
    parameter    n = 4;          // 4-bit bus by default
        //port declarations
    output [n-1:0] sum;       output co;
    input [n-1:0] a0, a1;     input ci;
        //local wire declaration
    wire [n-1:0] carry;
    assign carry[0] =ci;
    genvar i;
    generate for (i=0; i<n; i=i+1) begin: r_loop
            wire t1, t2, t3;
            xor g1 (t1, a0[i], a1[i]);
            xor g2 (sum[i], t1, carry[i]);
            and g3 (t2, a0[i], a1[i]);
            and g4 (t3, t1, carry[i]);
            or g5 (carry[i+1], t2, t3);
        end
    endgenerate
    assign co = carry[n];
endmodule
```

*** For the above generate loop, the following relative hierarchical instance names are generated**

```
xor : r_loop[0].g1, r_loop[1].g1, r_loop[2].g1, r_loop[3].g1
      r_loop[0].g2, r_loop[1].g2, r_loop[2].g2, r_loop[3].g2
and : r_loop[0].g3, r_loop[1].g3, r_loop[2].g3, r_loop[3].g3
      r_loop[0].g4, r_loop[1].g4, r_loop[2].g4, r_loop[3].g4
or :  r_loop[0].g5, r_loop[1].g5, r_loop[2].g5, r_loop[3].g5
```

* Generate instances are connected with the following generated nets

```
Nets : r_loop[0].t1, r_loop[0].t2, r_loop[0].t3
       r_loop[1].t1, r_loop[1].t2, r_loop[1].t3
       r_loop[2].t1, r_loop[2].t2, r_loop[2].t3
       r_loop[3].t1, r_loop[3].t2, r_loop[3].t3
```

7.8.2 Generate Conditional

Ex. 7-33

```
module multiplier   (product, a0, a1);
//parameter declaration, this can be redefined
parameter a0_width = 8;     // 8-bit bus by default
parameter a1_width = 8;     // 8-bit bus by default
//Local parameter declaration. This parameter cannot be modified
// with defparam or with mudule instance # statement.
localparam product_widt = a0_width + a1_width;
output [product_width – 1:0] product;
input [a0_width-1:0] a0;
input [a1_width-1:0] a1;
generate
if (a0_width < 8) || (a1_width < 8)
    cla_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
else
    tree_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
endgenerate     //end of the generate block
endmodule
```

5.9 Function and Task

Task: is similar to a software procedure. Parameters may be passed to it and results returned.

```
module mark1Task;
reg [15:0] m [0:8191]; // 8192 x 16 bit memory
reg [12:0] pc; // 13 bit program counter
reg [12:0] acc; // 13 bit accumulator
reg ck; // a clock signal
always begin: executeInstructions
    reg [15:0] ir; // 16 bit instruction register
    @(posedge ck) ir = m [pc];
    @(posedge ck)
        case (ir [15:13])
            // other case expressions as before
            3'b111 : multiply (acc, m [ir [12:0]]);
        endcase
    pc = pc + 1;
end

task multiply;
inout [12:0] a;
input [15:0] b;
    begin: serialMult
        reg [5:0] mcnd, mpy;//multiplicand and multiplier
        reg [12:0] prod;//product
        mpy = b[5:0];
        mcnd = a[5:0];
        prod = 0;
        repeat (6)
            begin
                if (mpy[0])
```

```
                    prod = prod + {mcnd, 6'b000000};
                prod = prod >> 1;
                mpy = mpy >> 1;
            end
        a = prod;
    end
endtask
endmodule
```

Function: It is called from within an expression and the value it return will be used in the expression. Unlike a task, a function may not include delay(#) or event control (@, wait) statements.

```
module mark1Fun;
    reg [15:0] m [0:8191]; // 8192 x 16 bit memory
    reg [12:0] pc; // 13 bit program counter
    reg [12:0] acc; // 13 bit accumulator
    reg ck; // a clock signal
    always
        begin: executeInstructions
            reg [15:0] ir; // 16 bit instruction register
            @(posedge ck)    ir = m [pc];
            @(posedge ck)
                case (ir [15:13])
                    //case expressions, as before
                    3'b111: acc = multiply(acc, m [ir [12:0]]);
                endcase
            pc = pc + 1;
        end
function [12:0] multiply;
    input [12:0] a;
```

```
     input [15:0] b;
          begin: serialMult
               reg [5:0] mcnd, mpy;
               mpy = b[5:0];
               mcnd = a[5:0];
               multiply = 0;
               repeat (6) begin
                    if (mpy[0])
                         multiply = multiply + {mcnd, 6'b000000};
                    multiply = multiply >> 1;
                    mpy = mpy >> 1;
                    end
          end
     endfunction
endmodule
```

Exercise: **2, 3, 4, 7, 8, 10, 11, 12, 14, 15, 16**