Chapter- 7

Samir Palnitkar
Structural Procedures

1. always
2. initial

- Basic statements in behavioral modeling
- Other statements can appear only inside these blocks
- always and initial represent separate activity flow
- Both blocks begins at 0 simulation time
initial

- Starts at zero simulation time
- **executes only once** during a simulation
- All initial blocks executed concurrently at time 0
- begin-end is used to build initial blocks (similar to `{}` in C)
- Used mainly for initialization, monitoring and waveforms and other processes that must be executed only once during the entire simulation run
Example-7.1 : initial block

module stimulus;
reg x,y, a,b, m;

initial
  m = 1'b0; //single statement; does not
  //need to be grouped

initial begin
  #5   a  = 1'b1; //multiple statements;
  //need to be grouped
  #25 b = 1'b0;
end

initial begin
  #10 x = 1'b0;
  #25 y = 1'b1;
end

initial
  #50 $finish;
endmodule
Example-7.1 : initial block

module stimulus;
reg x, y, a, b, m;

initial
  m = 1'b0; //single statement; does not need to be grouped

initial begin
  #5 a = 1'b1; //multiple statements; need to be grouped
  #25 b = 1'b0;
end

initial begin
  #10 x = 1'b0;
  #25 y = 1'b1;
end

initial
  #50 $finish;
endmodule

Three initial blocks execute in parallel at time 0

If a delay #<delay> is seen before a statement, the statement is executed <delay> time units after the current simulation time.

Results:

<table>
<thead>
<tr>
<th>time</th>
<th>statement executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>m = 1'b0;</td>
</tr>
<tr>
<td>5</td>
<td>a = 1'b1;</td>
</tr>
<tr>
<td>10</td>
<td>x = 1'b0;</td>
</tr>
<tr>
<td>30</td>
<td>b = 1'b0;</td>
</tr>
<tr>
<td>35</td>
<td>y = 1'b1;</td>
</tr>
<tr>
<td>50</td>
<td>$finish;</td>
</tr>
</tbody>
</table>
always

- Starts at zero simulation time
- **executes continuously** in a looping fashion
- begin-end is used to build initial blocks (similar to `{}` in C)
- Models a block of activity that is repeated continuously in a digital circuit
Example-7.2 : always block

module clock_gen;

reg clock;

// Initialize clock at time zero
initial
    clock = 1'b0;

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

initial
    #1000 $finish;

endmodule
Procedural Assignment

► Update values of reg, integer, real or time variables

\[ \text{lvalue} = \text{expression} \]

► The left-hand side of a procedural assignment

\[ \text{lvalue} \] can be one of the following:

- A \text{reg}, integer, real or time register variable or a memory element
- A bit select of these variables (e.g., addr[0])
- A part select of these variables (e.g., addr[31:16])
- A concatenation of any of the above (e.g., \{a, b[3:0]\})
Blocking assignments

► Executed in the order they are specified in a sequential block
► Will not block execution of statements that follow in a parallel block
► The “ = “ operator is used to specify blocking assignments
► Use blocking statements to build combinational logic block
Example-7.3 : Blocking Statements

module dummy;

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

//All behavioral statements must be inside an initial or always block
initial
begin
    x = 0; y = 1; z = 1; //Scalar assignments (@ time 0)
    count = 0; //Assignment to integer variables (@ time 0)
    reg_a = 16'b0; reg_b = reg_a; //initialize vectors (@ time 0)
    #15 reg_a[2] = 1'b1; //Bit select assignment with delay (@ time 15)
    #10 reg_b[15:13] = {x, y, z}; //Assign result of concatenation to
        //part select of a vector (@ time 25)
    count = count + 1; //Assignment to an integer increment (@ time 25)
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
Non-blocking assignments

► Allows scheduling of assignments without blocking execution of the statements that follow in a sequential block

► A “<=” operator is used to specify non-blocking assignments

► Executed last in the time step in which they are scheduled, that is, after all the blocking assignments in that time step are executed

► Use non-blocking statements to build Sequential logic block
Example-7.4 : Non-blocking Statements

module dummy;

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

//All behavioral statements must be inside an initial or always block
//Don’t mix blocking and non-blocking statements in one block
initial
begin
  x = 0; y = 1; z = 1;  //Scalar assignments (@ time 0)
  count = 0;            //Assignment to integer variables (@ time 0)
  reg_a = 16'b0; reg_b = reg_a;  //initialize vectors (@ time 0)
  #15 reg_a[2] <= 1'b1;        //Bit select assignment with delay (@ time 15)
  #10 reg_b[15:13] <= {x, y, z}; //Assign result of concatenation to
                                 //part select of a vector (@ time 10)
  count <= count + 1;         //Assignment to an integer increment (@ time 0)
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
Application of non-blocking assignments

- They are used as a method to model several concurrent data transfers that take place after a common event.

- Disadvantage: Degradation in the simulator performance and increase in memory usage.
Example: application of non-blocking assignment

always @(posedge clock)
Begin
    reg1 <= #1 in1;
    reg2 <= @(negedge clock) in2 ^ in3;
    reg3 <= #1 reg1; //The old value of reg1
end

►► At each positive edge of clock following sequence takes place for the non-blocking assignments.

1. Read operation is performed on each right-hand-side variable (in1, in2, in3 and reg1), at the positive edge of clock and expressions are evaluated
2. Write operations to the left-hand-side variables are based on the intra-assignment delay in each assignment
3. Write operations are executed at the scheduled time steps.

►► The order in which the write operations are executed is not important.
Example - 7.5 : non-blocking statements to eliminate race conditions

//Illustration 1: Two concurrent always blocks with blocking statements

always @(posedge clock)
    a = b;

always @(posedge clock)
    b = a;

//Illustration 2: Two concurrent always blocks with nonblocking statements

always @(posedge clock)
    a <= b;

always @(posedge clock)
    b <= a;
Example 7.6 : processing of non-blocking assignment

always @(posedge clock)
begin
  //Read operation
  //store values of right-hand-side expressions in temporary variable
  temp_a = a;
temp_b = b;

  //Write operation
  //Assign values of temporary variables to left-hand-side variables
  a = temp_b;
b = temp_a;
end
Timing Control for Simulation

- If no timing control statement, then the simulation time does not advance
  1. Delay based timing control
  2. Event based timing control
Delay based timing control

- Time duration between when the statement is encountered and when it is executed

- Syntax
  
  \#<number>
  
  \#<identifier>
  
  #(minimum: typical: maximum)
Example 7.7 : Regular delay control

module regular_delay;

//define parameters
parameter latency = 20;
parameter delta = 2;

//define register variables
reg x, y, z, p, q;

initial
begin
  x = 0;
  #10 y = 1;
  #latency z = 0;
  #(latency + delta) p = 1;
  #y x = x + 1;
  #(4:5:6) q = 0;
end

endmodule

Regular delay control

- Delay is always relative to when the statement is encountered.
- Specified to the left of a procedural assignment.

// no delay control
// delay control with a number. Delay execution of y = 1 by 10 units
// Delay control with identifier. Delay of 20 units
// Delay control with expression
// Delay control with identifier. Take value of y.
// Minimum, typical and maximum delay values. Discussed earlier in gate level modeling chapter.
Example 7.8: Intra-assignment delay control

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

// intra assignments delays
initial
begin
    x = 0; z = 0;
    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

// Equivalent method with temporary variables and regular delay control
initial
begin
    x = 0; z = 0;
    temp_xz = x + z;
    #5 y = temp_xz;  // Take value of x + z at the current time and store it in
                     // a temporary variable. Even though x and z
                     // might change between 0 and 5, the value assigned to
                     // at time 5 is unaffected.
end
```

Intra-assignment delay control

- Specified to the right of the assignment operator
Zero delay control

- Procedural statements in different always or initial blocks may be evaluated at the same simulation time.
- Order of execution of these statements is non-deterministic.
- Zero delay ensures that a statement is executed last in a simulation time.
Example 7.9: zero delay control

initial
begin
  x = 0;
  y = 0;
end

initial
begin
  #0 x = 1;  //zero delay control
  #0 y = 1;
end
Event based timing control

► What is an event…..?  
  ▪ The change in the value of a register or a net

► Four types of event control
  ▪ Regular event control
  ▪ Named event control
  ▪ Event or control
  ▪ Level-sensitive timing
Regular event control

- change in signal value
- positive/negative transition of a signal value

- \@ (clock) \ q = d; //q = d is executed whenever signal clock
  // changes value
- \@ (posedge clock) \ q = d; //q = d is executed whenever signal clock
  // does a positive transition (0 to 1, x or z,
  // x to 1, z to 1)
- \@ (negedge clock) \ q = d; //q = d is executed whenever signal clock
  // does a negative transition (1 to 0, x to z,
  // x to 0, z to 0)
- q = \@ (posedge clock) \ d; //d is evaluated immediately and assigned
  // to q at the positive edge of clock
Named event control

Example 7.11: named event control

// This is an example of a data buffer storing data after the last packet of data has arrived.

event received_data; // Define an event called received_data

always @ (posedge clock) // check at each positive clock edge
begin
  if(last_data_packet) // If this is the last data packet
    -> received_data; // trigger the event received_data
end

always @(received_data) // Await triggering of event received_data
begin
  data_buf = {data_pkt[0], data_pkt[2], data_pkt[3]}; // When event is triggered, store all four
  // packets of received data in data buffer
  // use concatenation operator {
end
event OR control

► Transition on any one of multiple signals or events

always @(reset or clock or D)
begin
  if(reset)
    q = 1'b0;
  else if(clock)
    q = D;
end
Level sensitive timing

► @ is edge sensitive
► wait is level sensitive
  ▪ Waits for a certain condition to be true before a block is executed

always

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

// monitors continuously
Conditional statements

►► if statement

- if (<expression>)
  true_statement;
- else
  false_statement;
- if (<expression>)
  true_statement1;
  else if (<expression>)
  true_statement2;
  ...
  else (<expression>)
  default;
Example 7.13 : conditional statement example

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

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

//Type 3 statements
//Execute statements based on ALU control signal.
if (alu_control == 0)
  y = x + z;
else if (alu_control == 1)
  y = x - z;
else
  $display ("Invalid ALU control signal");
case statement

- case (<expression>)

  alternative1 : statement1;  // block or statement
  alternative2 : statement2;

  default : default_statement;  // optional

endcase

- case statement compares 0, 1, x and z values in the expression and the alternative bit for bit. If unequal bit width, they are filled with zero to match the widest width.
Example 7.14: multiplexer with case statement

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

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;

Always @(s1 or s0 or i0 or i1 or i2 or i3)
Case ({s1, s0}) // Switch based on concatenation of control signals
  2’d0 : out = i0;
  2’d1 : out = i1;
  2’d2 : out = i2;
  2’d3 : out = i3;
  default: $display("Invalid control signals");
endcase

endmodule
Example 7.15: case statement with x and z

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

// Port declarations from the I/O diagram
output out0, out1, out2, out3;
reg  out0, out1, out2, out3;
input in;
input s1, s0;

always @(s1 or s0 or in)
    // Switch based on control signals
    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'b10 : begin  out0 = 1'bz;  out1 = 1'bz;  out2 = in;  out3 = 1'bz; end
        2'b11 : begin  out0 = 1'bz;  out1 = 1'bz;  out2 = 1'bz;  out3 = in; end
    // 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.
        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
Example 7.15: case statement with x and z continue....

module stimulus;

wire OUT0, OUT1, OUT2, OUT3;
reg IN, S1, S0;

// instantiate the decoder

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

initial
$monitor($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 = 1; S0 = 1;
  #5 IN = 1; S1 = 1'bwx; S0 = 0;
  #5 IN = 1; S1 = 1'bwx; S0 = 1'bwx;
  #5 IN = 1; S1 = 1'bwx; S0 = 0;
  #5 IN = 1; S1 = 0; S0 = 1'bwx;
end

endmodule
Example 7.16: casex use

```verilog
reg [3:0] encoding;
integer state;

casex (encoding) //logic value x represents a don’t care bit.
4’b1xxx : next_state = 3;
4’bx1xx : next_state = 2;
4’bxx1x : next_state = 1;
4’bxxx1 : next_state = 0;
default : next_state = 0;
endcode
```

casez: treats all z values in the case alternatives as the case expression as don’t care

casex: treats all z & x values as don’t care
Loops

- `while(<expression>)` // any logical expression

  - Example 7.17

    ```
    module count_mod;

    // Illustration 1: Increment count from 0 to 127.
    // Exit at count 128. Display the count variable.

    integer count;

    initial
    begin
      count = 0;
      while (count < 128) // Execute loop till count is 127. 
        // exit at count 128
        begin
          $display("Count = %d", count);
          count = count + 1;
        end
    end
    endmodule
    ```
For (initial condition; condition; change of value)

- Example 7.18

```
module counter;

integer count;

initial
  for (count=0; count < 128; count = count + 1)
    $display("Count = %d", count);
endmodule
```
repeat

- Executes the loop a fixed number of times
- Must contain a number (constant, variable, signal value)
- Variable or signal value evaluated only when the loop starts

Example 7.19

```vhdl
module counter;
// Illustration 1: increment and display count from 0 to 127

integer count;

initial
begin
  count = 0;
  repeat(128)
  begin
    $display("Count = %d", count);
    count = count + 1;
  end
end
endmodule
```
forever

- No expression
- Executes forever until $finish is encountered
- Can also be exited using disable
- Without a timing constrain this statement is executed indefinitely
- Useful to generate a clock in test bench

Example 7.20

```verilog
module clock_gen;
    //Example 1: Clock generation
    //Use forever loop instead of always block

    reg clock;

    initial
    begin
        clock = 1'b0;
        forever #10 clock = ~clock; //Clock with period of 20 units
    end

    initial
    #100000 $finish;

endmodule
```
Sequential blocks

begin.........end

- Statements are processed in the order they are specified
- Delay/event control relative to the simulation time when the
  previous statement in the block completed execution

Example 7.21

//Illustration 1: Sequential block without delay

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

initial
begin
  x = 1'b0;
y = 1'b1;
z = {x, y};
w = {y, x};
end
```
Example 7.21: sequential block continue...

module sequential;

// Illustration 2: Sequential blocks with delay.

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;        // completes at simulation time 0
    #5  y   = 1'b1;        // completes at simulation time 5
    #10 z  = {x, y};       // completes at simulation time 15
    #20 w = {y, x};w = {y, x}; // completes at simulation time 35
end

endmodule
Parallel blocks

Example 7.22

module parallel;
    // Illustration 1: Parallel Blocks with delay
    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;       // completes at simulation time 0
            #5 y = 1'b1;    // completes at simulation time 5
            #10 z = {x, y}; // completes at simulation time 15
            #20 w = {y, x}; // completes at simulation time 35
        join
    endmodule
Nested Blocks

►► Example 7.23 :

// Nested blocks
initial
begin
    x = 1'b0;
    fork
        #5 y = 1'b1;
        #10 z = {x, y};
    join
        #20 w = {y, x};
end
endmodule
Named Block

Named blocks can be disabled, i.e. their execution can be stopped.

module find_true_bit;
//Illustration: Find the first bit with a value 1 in flag (vector variable)
reg [15:0] flag;
integer i; //integer to keep count

initial
begin
  flag = 16'b 0010_0000_0000_0000;   i = 0;
begin: block1 //The main block inside while is named block1
  while(i < 16)
  begin
    if (flag[i])
    begin
      $display("Encountered a TRUE bit at element number %d", i);
      disable block1; //disable block1 because you found true bit.
    end
    i = i + 1;
  end
end
endmodule
Example (traffic signal controller)

Specifications:

- The traffic signal for the main highway gets highest priority because cars are continuously present on the main highway. Thus, 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 and the traffic signal on the main highway turns green again.

- 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 $S_1$ to $S_2$, from $S_2$ to $S_3$, and from $S_4$ to $S_0$. The delays must be controllable.
State Machine

<table>
<thead>
<tr>
<th>State</th>
<th>Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>Hwy=G, Cntry=R</td>
</tr>
<tr>
<td>S1</td>
<td>Hwy=Y, Cntry=R</td>
</tr>
<tr>
<td>S2</td>
<td>Hwy=R, Cntry=R</td>
</tr>
<tr>
<td>S3</td>
<td>Hwy=R, Cntry=G</td>
</tr>
<tr>
<td>S4</td>
<td>Hwy=R, Cntry=Y</td>
</tr>
</tbody>
</table>
Verilog description

`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; //declare output signals are registers
input X; //if TRUE, indicates that there is car on the country road, otherwise FALSE
input clock, clear;

//Status of lights
parameter RED = 2'd0, YELLOW = 2'd1, GREEN = 2'd2;

//State definition
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;
Verilog description (cont...) 

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

//Compute values of main signal and country signal 
always @(state) 
begin 
    hwy = GREEN; //Default light assignment 
    cntry = RED; //Default light assignment 
    case (state) 
        S0: ; 
        S1: hwy = YELLOW; 
        S2: hwy = RED; 
        S3: begin 
            hwy = RED; 
            cntry = GREEN; 
        end 
        S4: begin 
            hwy = RED; 
            cntry = YELLOW; 
        end 
    endcase 
end
Verilog description (cont…)

//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
Verilog description (Stimulus)

//Stimulus Module
module stimulus;
wire [1:0] MAIN_SIG, CNTRY_SIG;
reg CAR_ON_CNTRY_RD;       //if TRUE, indicates that there is car on the country road
reg CLOCK, CLEAR;

//Instantiate signal controller
sig_control SC(MAIN_SIG, CNTRY_SIG, CAR_ON_CNTRY_RD, CLOCK, CLEAR);

//Setup monitor
initial
$monitor($time, " Main Sig = %b Country Sig = %b Car_on_cntry = %b",
           MAIN_SIG, CNTRY_SIG, CAR_ON_CNTRY_RD);

//setup clock
initial begin
  CLOCK = `FALSE;
  forever #5 CLOCK = ~CLOCK;
end
Verilog description (Stimulus)

//control clear signal
initial
begin
    CLEAR = `TRUE;
    repeat (5) @(negedge CLOCK);
    CLEAR = `FALSE;
end

//apply stimulus
initial
begin
    CAR_ON_CNTRY_RD = `FALSE;
    repeat (20) @(negedge CLOCK); CAR_ON_CNTRY_RD = `TRUE;
    repeat (10) @(negedge CLOCK); CAR_ON_CNTRY_RD = `FALSE;
    repeat (20) @(negedge CLOCK); CAR_ON_CNTRY_RD = `TRUE;
    repeat (10) @(negedge CLOCK); CAR_ON_CNTRY_RD = `FALSE;
    repeat (20) @(negedge CLOCK); CAR_ON_CNTRY_RD = `TRUE;
    repeat (10) @(negedge CLOCK); CAR_ON_CNTRY_RD = `FALSE;
    repeat (20) @(negedge CLOCK); CAR_ON_CNTRY_RD = `TRUE;
    repeat (10) @(negedge CLOCK); CAR_ON_CNTRY_RD = `FALSE;
end
endmodule