שפת תכנון חומרה Verilog - ורילוג

Verilog – Behavioral Modeling

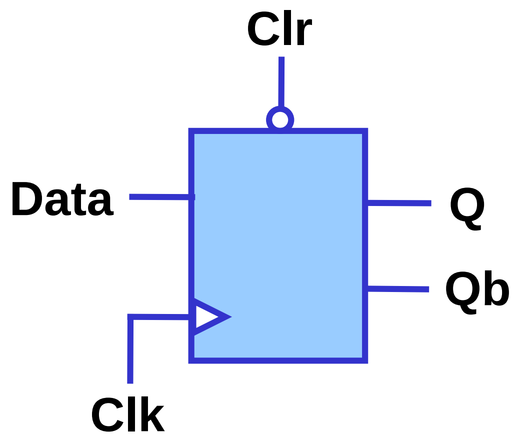
Dr. Avihai Aharon

Objectives

- Explain the significance of structured procedures always and initial in behavioral modeling
- Define blocking and non-blocking procedural assignments
- Understand delay-based timing control mechanism in behavioral modeling. Use regular delays, intra-assignment delays, and zero delays
- Use level-sensitive timing control mechanism in behavioral modeling
- Explain conditional statements using if and else
- Describe multi-way branching, using case, casex, and casez statements
- Understand looping statements such a while, for, repeat, and forever
- Define sequential and parallel blocks
- Understand naming of blocks and disabling of named blocks

Behavioral Modeling

- Behavioral modeling enables us to describe the system at a high level of abstraction
- Design at this level resembles C programming more than it resembles digital circuit design
- Verilog is rich in behavioral constructs that provide the designer with a great amount of flexibility



```
At every positive edge of Clk
   If Clr is not low
      Set Q to the value of Data
      Set Qb to inverse of Data

Whenever Clr goes low
      Set Q to 0
      Set Qb 1
```

Structured Procedures (Procedural Blocks)

- There are two structured procedure statements in Verilog: always and initial
- These statements are the two most basic statements in behavioral modeling (all over behavioral statements can appear only inside these procedural blocks)
- Procedural blocks have the following components
  - ✓ Procedural assignment statement
  - ✓ High-level constructs (loops, conditional statements)
  - ✓ Timing controls
- Each always and initial statement represents a separate activity flow in Verilog that can run in parallel rather than in sequence
  - ✓ Each activity flow starts at simulation time 0
  - ✓ The statements always and initial cannot be nested

“initial” Statement

- All statements inside an initial statement constitute an initial block
  - ✓ An initial block starts at time 0
  - ✓ An initial block executes once during a simulation and does not execute again
- If there are multiple initial blocks
  - ✓ Each block starts to execute concurrently in time 0
  - ✓ Each block finishes execution independently of over blocks
- Multiple behavioral statement must be grouped, typically using the keywords begin and end
  - ✓ If there is only one behavioral statement, grouping is not necessary
- The initial blocks are typically used for initialization, monitoring, waveforms and other processes that must be executed only once during the entire simulation run

“initial” Statement (example)

```
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
```

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

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

“always” Statement

- **All behavioral statements inside an always statement constitute an always block**
  - ✓ **The always statement starts at time 0**
  - ✓ **An always statement executes the statements in the always block continuously in a looping fashion (its stopped only by power off ($finish) or by an interrupt ($stop))**
- **If there are multiple always blocks**
  - ✓ **Each block starts to execute concurrently in time 0**
  - ✓ **Each block continue execution independently of over blocks**
- **Multiple behavioral statement must be grouped using the keywords begin and end**
  - ✓ **If there is only one behavioral statement, grouping is not necessary**
- **The always block is used to model a block of activity that is repeated continuously in a digital circuit (a clock generator module)**

“always” Statement (example)

```
module clock_gen;

reg clock;

// Initialize a 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 Assignments

- ***Procedural assignments*** **updates 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***
- ***Procedural assignment*** **syntax:**

```
<assignment>
        ::= <LHS_Value> = <RHS_Expression>
```

  - ✓ ***The LHS of a procedural assignment must be of a register-class data type (reg, integer, real or time register variable or memory element)***
  - ✓ ***The RHS of a procedural assignment can be any valid expression (the data types used here are not restricted)***
- **There are two types of procedural assignment statements: *blocking* and *non-blocking***

Blocking Assignment

- **Blocking assignment statements are executed in the order they are specified in a sequential block**
  - **A blocking assignment will not block execution of statements that follow in a parallel block**
  - **The "=" operator is used to specify blocking assignment**
- **Procedural assignments to registers:**
  - **In the RHS has more bits than the register variable, the RHS is truncated to mach the width of the register variable. The least significant bits are selected and the most significant bits are discarded**
  - **If the RHS has fewer bits, zeros are filled in the most significant bits of the register variable**

Blocking Assignment (example)

```
reg x, y,z;
reg [15:0] reg_a, reg_b;
integer count;
initial
    begin
        // Scalar assignments
        x = 1'b0; y = 1'b1; z = 1'b1;
        // Assignment to integer variable
        count = 0;
        // Initialize vectors
        reg_a = 16'b0; reg_b = reg_a;
        // Bit select assignment with delay
        #15 reg_a[2] = 1'b1;
        // Assign result of concatenation to part select of  a vector
        #10 reg_b[15:13] = {x, y, z};
        // Assignment to  an integer (increment)
        count = count + 1;
    end
```

time	*statement executed*
0	x = 1'b0; y = 1'b1; z = 1'b1;
	reg_a = 16'b0; reg_b = reg_a;
	count = 0;
15	reg_a[2] = 1'b1;
25	reg_b[15:13] = {x, y, z};
	count = count + 1;

Non-blocking Assignment

- **Non-blocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block**
  - ✓ **The "<=" operator is used to specify non-blocking assignment (this operator interpreted as a relational operator in an expressions)**
- **The simulator schedules a non-blocking assignment statement to execute and continues to the next statement in the block without waiting for the non-blocking statement to complete execution**
- **Typically (but not for any simulator), non-blocking assignment statement are executed last in the time step in which they are scheduled, that is, after all the blocking assignments in that time step are executed**

Non-blocking Assignment (example)

```
reg x, y,z;
reg [15:0] reg_a, reg_b;
integer count;
initial
    begin
        // Scalar assignments
        x = 1'b0; y = 1'b1; z = 1'b1;
        // Assignment to integer variable
        count = 0;
        // Initialize vectors
        reg_a = 16'b0; reg_b = reg_a;
        // Bit select assignment with delay
        reg_a[2] <= #15 1'b1;
        // Assign result of concatenation to part select of  a vector
        reg_b[15:13] <= #10 {x, y, z};
        // Assignment to  an integer (increment)
        count = count + 1;
    end
```

time	*statement executed*
0	x = 1'b0; y = 1'b1; z = 1'b1;
	reg_a = 16'b0; reg_b = reg_a;
	count = count + 1;
10	reg_b[15:13] = {x, y, z};
15	reg_a[2] = 1'b1;

Non-blocking Assignment Application

- **Non-blocking assignments used as a method to model several concurrent data transfers that take place after a common event**
  - ✓ **In such cases, blocking assignment can potentially cause race conditions because the final result depends on the order in which the assignments are evaluated (see examples)**
  - ✓ **When using non-blocking assignment, the final result is not dependent on the order in which the assignments are evaluated**

```
// Illustration 1: Two concurrent always blocks with blocking statements
// Potentially race condition (depending on simulator implementation)
always @(posedge clock)             a = b;
always @(posedge clock)             b = a;

// Illustration 2: Two concurrent always blocks with non-blocking statements
// Eliminate the race condition, values of registers a and b are swapped correctly
always @(posedge clock)             a <= b;
always @(posedge clock)             b <= a;
```

Timing Control

- Various behavioral timing control constructs are available in Verilog
  - ✓ If there are no timing control statements – the simulation time does not advance
- Timing controls provide a way to specify the simulation time at which procedural statements will execute
- There are three methods of timing controls
  - ✓ Delay-based timing control
  - ✓ Event-based timing control
  - ✓ Level-sensitive timing control

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 (delays are specified by the symbol "#")**
- **Syntax for the delay-based timing control statements:**

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

- **There are three types of delay control for procedural assignment**
  - ✓ **Regular delay control**
  - ✓ **Intra-assignment delay control**
  - ✓ **Zero delay control**

Regular Delay Control

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

```
initial
    begin
         x =0;                          // no delay control

         #10 y = 1;                     // delay control with a number

         #latency z = 0;                       // delay control with identifier (parameter)

         #(latency + delay) p = 1;           // delay control with expression

         #y x = x + 1;                  // delay control with a identifier (value of y)

         #(4:5:6) q = 0;                // minimal, typical and maximum delay values
                                   // (discussed in gate-level modeling chapter)
    end
```

Intra-Assignment Delay Control

- **Regular delay control** is used when a non-zero delay is specified to the right of assignment operator

```
initial                          // Intra-assignment delay
    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

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

Zero Delay Control

- **Zero delay control is a method to ensure that a statement is executed last, after all other statements in that simulation time are executed (used to eliminate race condition)**
  - ✓ **Procedural statements in different always-initial blocks may be evaluated at the same simulation time**
  - ✓ **The order of execution of these statements in different always-initial blocks is non-deterministic**
  - ✓ **If there are multiple zero delay statements, the order between them is non-deterministic**

```
…
     initial      x = 0;
     initial      y = 0;
     initial      #0 x = 1;        // Zero delay control
     initial      #0 y = 1;        // Zero delay control
…
```

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***
- There are four types of *event-based timing control*
  - ✓ ***Regular event control***
  - ✓ ***Named event control***
  - ✓ ***Event OR control***
  - ✓ ***Level-sensitive event control***

Regular Event Control

- A "@" 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
  - ✓ The keywords "posedge" and "negedge" are used for a positive and negative transitions

```
@(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 -> 1,x, or z; x -> 1; z -> 1)

@(negedge clock) q = d;      // q = d is executed whenever signal clock does a negative
                             // transition (1 -> 0,x, or z; x -> 0; z -> 0)

q = @(posedge clock) d;      // d is evaluated immediately and assigned to q at the
                                     // positive edge of clock
```

Named Event Control

- Verilog provide the capability to declare an event and then trigger and recognize the occurrence of that event
  - ✓ The event does not hold any data
  - ✓ A named event is declared by the keyword event
  - ✓ An event is triggered by the symbol "->"
  - ✓ The triggering of the event is recognized by the symbol "@"

```
event received_data;                  // Define an event called receive data

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

// Await triggering of event received_data and store all four received packets in data buffer
always @(received_data) data_buff = {data_pkt[0], data_pkt[1], data_pkt[2], data_pkt[2]};
```

Event OR Control

- A transition of any one of multiple signals or events can trigger the execution of a statement or a block of statements
  - ✓ This is expressed as an OR of events or signals
  - ✓ The list of events or signals expressed as an OR is also known as a sensitivity list
  - ✓ The keyword “or” is used to specify multiple triggers

```
// A level-sensitive latch with asynchronous reset
always @(reset or clock or d)                       // wait for reset or clock or d to change
    begin
        if(reset)
                    q = 1’b0;                       // if reset signal is high, set q to 0
        else
            if (clock)
                    q = d;                          // if clock is high, latch output
    end
```

Level-Sensitive Timing Control

- Verilog allows level-sensitive timing control – the ability to wait for a certain condition to be true before a statement or a block of statements is executed
  - ✓ The keyword “wait” is used for level-sensitive construct

```
initial received_data = 1’b0;                    // initialize flag register called receive_data

always @(posedge clock)                          // check at each positive clock edge
    begin
        if(last_data_packet)                     // if this is the last data packet
                received_data = 1’b1;            // Set received_data flag
    end

 // Await setting of flag received_data and store all four received packets in data buffer
always
        wait (received_data) #20 data_buff = {data_pkt[0], data_pkt[1], data_pkt[2], data_pkt[2]};
```

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 executed
- Keywords “if” and “else” are used for conditional statements
- There are three types of conditional statements
  - ✓ No “else” statement. Statement execute or not execute
  - ✓ One “else” statement. Either true statement and false statement is evaluated
  - ✓ Nested if-else-if. Choice of multiple statements. Only one executed

Conditional Statement Types

```
// Type 1 conditional statement. No “else” statement.
// Statement execute or not execute.
If (<expression>) true_statement;
--------------------------------------------------------------------------------
// Type 2 conditional statement. One “else” statement.
// Either true_statement and false_statement is evaluated.
If (<expression>) true_statement;
else false_statement;
--------------------------------------------------------------------------------
// Type 3 conditional statement. Nested if-else-if.
// Choice of multiple statements. Only one executed.
If (<expression1>) true_statement1;
else if (<expression2>) true_statement2;
else if (<expression3>) true_statement3;
else default_statement;
```

Multi-way Branching

- The nested if-else-if (conditional statements type 3) can become unwieldy if there are too many alternatives.
- A shortcut to achieve the same result is to use the “case” statement
  - ✓ The keywords “case”, “endcase”, and “default” are used in the case statement
  - ✓ A block of multiple statements can be grouped by keywords “begin” and “end”
  - ✓ The expression is compared to alternatives in the order they are written. For the first alternative that matches, the corresponding statement or block is executed
  - ✓ The case statement compares 0,1, x, and z values in the expression and the alternatives bit for bit
  - ✓ If the expression and the alternative are of unequal bit width, they are zero filled to match the bit width of the widest of the expression and the alternative

“case” Statement

- It is a good programming practice to always use the default statement, especially to check for x or z
  - ✓ The default_statement is optional
  - ✓ Placing of multiple default statements in one case statement are not allowed
- More than one *case item* can be specified at a time
- The *case statements* can be nested
- The *case statement* can also act like a many-to-one multiplexer *(see example)*

```
case (expression)
      alternative1: statement1;
      alternative2: statement2;
      alternative3: statement3;
      alternative4,
      alternative5: statement5;
      …
      default: default_statement;
endcase
```

4-to-1 Multiplexer with “case” Statement

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
        // port declaration from the I/O diagram
        output out;
        input i0, i1, i2, i3, s0, s1;
        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 signal”);
        endcase
endmodule
```

“casex” & “casez” Statements

There are two variations of the case statement. They are denoted by keywords, casex and casez .

- ✓ casex treats all x and z values in the case item or the case expression as don’t cares.
- ✓ casez treats all z values in the case alternatives or the case expression as don’t cares. All bit positions with z can also represented by “?” in that position.
- ✓ The use of casex and casez allows comparison of only non-x or -z positions in the case expression and the case alternatives.

```
casex (encoding)          // logic value x represents a don’t
                          // care bits
    3’b1xx : next_state = 2;
    3’bx1x : next_state = 1;
    3’bxx1 : next_state = 0;
    default: next_state = 0;
endcase
```

Loops in Verilog

- There are four types of looping statements in Verilog:
  - ✓ "while" looping statements
  - ✓ "for" looping statements
  - ✓ "repeat" looping statements
  - ✓ "forever" looping statements
- The syntax of these loops is very similar to the syntax of loops in the C programming language
- All looping statements can appear only inside an initial or always block
- Loops may contain delay expressions

Loops in Verilog: “while” Loop Structure

- The keyword “while” is used to specify this loop
  - ✓ The while loop executes until the while_expression becomes false (logical “0”)
  - ✓ If the loop is entered when the while_expression is false, the loop is not executed at all
  - ✓ If multiple statements are to be executed in the loop, they must be grouped typically using keywords begin and end

```
    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
```

Loops in Verilog: “for” Loop Structure

- The keyword “for” is used to specify this loop
  - ✓ This loop provides a more compact loop structure than the while loop
  - ✓ The for loop cannot be used in place of the while loop in all situations
  - ✓ For loops are generally used when there is a fixed beginning and end to the loop. If the loop is simply looping on a certain condition, it is better to use the while 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

```
integer count;
initial
        for ( count = 0; count < 128; count = count + 1) $display(“count = %d”,
   count);
```

Loops in Verilog: “repeat” Loop Structure

- The keyword “repeat” is used to specify 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 (a while loop is used for this purpose)
  - ✓ A repeat construct must contain a number, which can be a constant, a variable, or a signal value
  - ✓ If the number is variable or signal value, it is evaluated only when the loop starts and not during the loop execution

```
integer count;
initial
    begin count = 0;
          repeat (128) begin
                $display(“count = %d”, count);
                count = count + 1;
          end
    end
```

Loops in Verilog: “forever” Loop Structure

- The keyword “forever” is used to specify 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 (e.g., while (1))
  - ✓ A forever loop can be exited by use of the disable statement
  - ✓ A forever loop is typically used in conjunction with timing control constructs (if timing control construct is not used, the Verilog simulator would execute this statement infinitely without advancing simulation time and the rest of the design would never be executed)

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

Sequential & Parallel Blocks

- Block statements are used to group multiple statements to act together as one
  - ✓ We already used keywords begin and end to group multiple statements
- There are two types of blocks
  - ✓ Sequential blocks
  - ✓ Parallel blocks
- There are three special features of blocks
  - ✓ Named blocks
  - ✓ Disabling named blocks
  - ✓ Nested blocks

Block Types: Sequential Blocks

- The keywords begin and end are used to group statements into sequential blocks, that have the following characteristics
  - ✓ The statements in a sequential block are processed in the order they are specified. A statement is executed only after its preceding statement completes execution (except for non-blocking assignment with intra-assignment timing control)
  - ✓ If delay or event control is specified, it is relative to the simulation time when the previous statement in the block completed execution

Block Types: Parallel Blocks

- Parallel blocks, specified by keywords "fork" and "join", provide interesting simulation feature and have the following characteristics
  - ✓ Statements in a parallel block are executed concurrently
  - ✓ Ordering of statements is controlled by the delay or event control assigned to each statement
  - ✓ I delay or event control is specified, it is relative to the time the block was entered

- All statements in a parallel block start at the time when the block was entered.
  - ✓ The order in which the statements are written in the block is not important
  - ✓ It is important to be careful with parallel blocks because of implicit race condition that might arise if two statements that effect the same variable complete at the same time

Block Types: Parallel Blocks

- The keyword fork can be viewed as splitting a single flow into independent flows
- The keyword join can be seen as joining the independent flows into a single flow
- Independent flows operate concurrently

```
// Parallel blocks with delay
reg x, y;
reg [1:0] z, w;
initial
        fork
                x = 1'b0;
                #5 y = 1'b1;
                #10 z = {x, y};
                #20 w = {y, x};
        join
```

time	*statement executed*
0	x = 1'b0;
5	y = 1'b1;
10	z = {0, 1};
20	w = {1,0};

Special Features: Nested Blocks

- Blocks can be nested
- Sequential and parallel blocks can be mixed

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

time	*statement executed*
0	x = 1' b0;
5	y = 1' b1;
10	z = {0, 1};
30	w = {1,0};

Special Features: Named Blocks

- Blocks can be given names
  - ✓ Local variables can be declared for the named block
  - ✓ Named blocks are a part of the design hierarchy. Variables in a named block can be accessed by using hierarchical name referencing
  - ✓ Named blocks can be disabled, i.e., their execution can be stopped

```
module top;
initial
        begin: block1         // sequential named block
        integer i; // static and local to block 1 (top.block1.i)
                    …
        end
initial
        fork:  block2         // parallel named block
        reg i;      // static and local to block 2 (top.block2.i)
                    …
        join
```

Special Features: Disabling Named Blocks

- The keyword "disable" provides a way to terminate the execution of a block
  - ✓ "disable" can be used to get out of loops, handle error conditions, or control execution of pieces of code, based on control signal
  - ✓ Disabling a block causes the execution control to be passed to the statement immediately succeeding the block

```
initial
        begin/fork: block_name       // sequential/parallel named block
                …
                if (<condition>)
                        …
                else
                        disable block_name;     // Disabling of sequential/parallel
                                                // named block
        end/join
```

Verilog “Stratified Event Queue”

- Active events (these events may be scheduled in any order)
  - Blocking assignments
  - Evaluate RHS of non-blocking assignments
  - Continuous assignments
  - $display command execution
  - Evaluate inputs and change outputs of primitives
- Inactive events
  - #0 blocking assignments
- Non-blocking events
  - Update LHS of non-blocking assignments
- Monitor events
  - $monitor command execution
  - $strobe command execution
- Other specific PLI commands

Behavioral Modeling Summary

- A behavioral description expresses a digital circuit in terms of the algorithms it implements. A behavioral description does not necessarily include the hardware implementation details. Behavioral modeling is used in the initial stages of design process to evaluate various design-related trade-offs.

- Structured procedures initial and always form the basis of behavioral modeling. All other behavioral statements can appear only inside initial or always blocks. An initial block executes once; an always block executes continuously until simulation ends

- Procedural assignment are used in behavioral modeling to assign values to register variables. Blocking assignments must be complete before the succeeding statement can execute. Non-blocking assignments schedule assignments to be executed and continue processing to the succeeding statement

Behavioral Modeling Summary (cont.)

- Delay-based timing control, event-based timing control, and level-sensitive timing control are there three ways to control timing and execution order of statements in Verilog. Regular delay, zero delay, and intra-assignment delay are three types of delay-based timing control. Regular event, named event, and event OR are three types of event-based timing control. The wait statement is used to model level-sensitive timing control
- Conditional statements are modeled in behavioral Verilog with if and else statements. If there are multiple branches, use of case statements is recommended. casex and casez are special cases of the case statement
- Keywords while, for, repeat, and forever are used for four types of looping statements in Verilog
- Sequential and parallel are two types of blocks. Sequential blocks are specified by keywords begin and end. Parallel blocks are expressed by keywords fork and join. Blocks can be nested and named. Named blocks can be referenced by hierarchical names. Named blocks can be disabled