Textbook: Verilog® HDL 2nd. Edition
Samir Palnitkar
Prentice-Hall, Inc.

INSTRUCTOR: CHING-LUNG SU
E-mail: kevinsu@yuntech.edu.tw
Chapter 7

Behavioral Modeling
Outline of Chapter 6

7.1 Structured Procedures
7.2 Procedural Assignments
7.3 Timing Controls
7.4 Conditional Statements
7.5 Multiway Branching
7.6 Loops
7.7 Sequential and Parallel Blocks
7.8 Generate Blocks
7.9 Examples
7.1 Structured Procedures
7.2 Procedural Assignments
7.3 Timing Controls
7.4 Conditional Statements
7.5 Multiway Branching
7.6 Loops
7.7 Sequential and Parallel Blocks
7.8 Generate Blocks
7.9 Examples
7.1 Structured Procedures

- **Behavioral Modeling**
  - Designers need to **early evaluate** the trade-offs of various architectures and algorithms before they decide on the optimum architecture and algorithm.
  - More design considerations focus in the behavior of the algorithm and its performance.
  - Verilog provides designers the ability to describe design functionality in an algorithmic manner.
  - Designers describes the behavior of the circuit in Verilog.
7.1 Structured Procedures

- **Behavioral Modeling (Cont.)**

  - Behavioral modeling represents the circuit at a very high level of abstraction.
  - Behavioral Verilog constructs are similar to C language constructs in many ways and the behavioral constructs that provide the designer with a great amount of flexibility.
There are two structured procedure statements in Verilog: `always` and `initial`.

Behavioral statements can appear only inside these structured procedure statements.

Verilog is a concurrent programming language unlike the C programming language, which is sequential in nature.

Activity flows in Verilog run in parallel rather than in sequence.

“always” and “initial” statement represents a separate activity flow in Verilog.

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, 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.
- Multiple behavioral statements must be grouped, typically using the keywords begin and end.
- If there is only one behavioral statement, grouping is not necessary.
## Example for initial Statement

```verilog
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
```

**Parallel Start at Time 0**

<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>
7.1 Structured Procedures

- **initial Statement** (Cont.)
  - The initial blocks are typically used for initialization, monitoring, waveforms and other processes that must be executed only once during the entire simulation run.
  - The following subsections discuss how to initialize values using alternate shorthand syntax.
  - The use of such shorthand syntax has the same effect as an initial block combined with a variable declaration.
Combined Variable Declaration and Initialization

// The clock variable is defined first
reg clock;
// The value of clock is set to 0
initial clock = 0;

// Instead of the above method, clock variable
// can be initialized at the time of declaration
// This is allowed only for variables declared
// at module level.
reg clock = 0;
module adder (sum, co, a, b, ci);

output reg [7:0] sum = 0; // Initialize 8 bit output sum
output reg          co  = 0; // Initialize 1 bit output co
input   [7:0] a, b;
input   ci;

endmodule
Combined ANSI C Style Port Declaration and Initialization

module adder (output reg [7:0] sum = 0, //Initialize 8 bit output
    output reg          co  = 0, //Initialize 1 bit output co
    input   [7:0] a, b,
    input   ci
);
--
--
endmodule
7.1 Structured Procedures

- **always Statement**
  - All behavioral statements inside an always statement constitute an always block.
  - The always statement starts at time 0 and executes the statements in the always block continuously in a looping fashion.
  - This statement is used to model a block of activity that is repeated continuously in a digital circuit.
  - An example is a clock generator module that toggles the clock signal every half cycle.
  - In real circuits, the clock generator is active from time 0 to as long as the circuit is powered on.
Example for always Statement

module clock_gen (output 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
7.2 Procedural Assignments

7.1 Structured Procedures
7.2 Procedural Assignments
7.3 Timing Controls
7.4 Conditional Statements
7.5 Multiway Branching
7.6 Loops
7.7 Sequential and Parallel Blocks
7.8 Generate Blocks
7.9 Examples
Procedural Assignments

- 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.
- These are unlike continuous assignments discussed in Chapter 6, Dataflow Modeling, where one assignment statement can cause the value of the right-hand-side expression to be continuously placed onto the left-hand-side net.
7.2 Procedural Assignments

Syntax for Procedural Assignment

\[
\text{assignment ::= variable\_lvalue = [ delay\_or\_event\_control ] expression}
\]

- The left-hand side of a procedural assignment \texttt{<lvalue>} can be one of the following:
  1. A \texttt{reg, integer, real, or time} register variable or a memory element
  2. A bit select of these variables (e.g., \texttt{addr[0]})
  3. A part select of these variables (e.g., \texttt{addr[31:16]})
  4. A concatenation of any of the above

- There are two types of procedural assignment statements: blocking and nonblocking.
7.2 Procedural Assignments

- **Blocking Assignments**

  - 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.
  - Both parallel and sequential blocks are discussed in Section 7.7, Sequential and Parallel Blocks.
  - The = operator is used to specify blocking assignments.
Example for Blocking Assignments

```verilog
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
  count = 0; //Assignment to integer variables
  reg_a = 16'b0; reg_b = reg_a; //initialize vectors
  #15 reg_a[2] = 1'b1; //Bit select assignment with delay
  #10 reg_b[15:13] = {x, y, z} //Assign result of concatenation to
    // part select of a vector
  count = count + 1; //Assignment to an integer (increment)
end
```
Example for Blocking Assignments (Cont.)

- In Example of P-20, the statement $y = 1$ is executed only after $x = 0$ is executed.
- The behavior in a particular block is sequential in a `begin-end` block if blocking statements are used, because the statements can execute only in sequence.
- The statement `count = count + 1` is executed last.
7.2 Procedural Assignments

- Example for Blocking Assignments (Cont.)

- The simulation times at which the statements are executed are as follows:
  - All statements $x = 0$ through $\text{reg}_b = \text{reg}_a$ are executed at time 0
  - Statement $\text{reg}_a[2] = 0$ at time = 15
  - Statement $\text{reg}_b[15:13] = \{x, y, z\}$ at time = 25
  - Statement $\text{count} = \text{count} + 1$ at time = 25
  - Since there is a delay of 15 and 10 in the preceding statements, $\text{count} = \text{count} + 1$ will be executed at time = 25 units
Blocking Assignments (Cont.)

- Note that for procedural assignments to registers, if the right-hand side has more bits than the register variable, the right-hand side is truncated to match the width of the register variable.

- The least significant bits are selected and the most significant bits are discarded.

- If the right-hand side has fewer bits, zeros are filled in the most significant bits of the register variable.
Nonblocking Assignments

- Nonblocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block.
- A <= operator is used to specify nonblocking assignments.
- The same symbol as a relational operator, less_than_equal_to.
### Example for Nonblocking Assignments

```vhdl
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
  count = 0; //Assignment to integer variables
  reg_a = 16'b0; reg_b = reg_a; //Initialize vectors
  reg_a[2] <= #15 1'b1; //Bit select assignment with delay
  reg_b[15:13] <= #10 {x, y, z}; // Assign result of concatenation
  count <= count + 1; //Assignment to an integer (increment)
end
```
Example for Nonblocking Assignments

- In example of P-25, the statements \( x = 0 \) through \( \text{reg}_b = \text{reg}_a \) are executed sequentially at time 0.
- Then the three nonblocking assignments are processed at the same simulation time.
  - \( \text{reg}_a[2] = 0 \) is scheduled to execute after 15 units (i.e., \( \text{time} = 15 \))
  - \( \text{reg}_b[15:13] = \{x, y, z\} \) is scheduled to execute after 10 time units (i.e., \( \text{time} = 10 \))
  - \( \text{count} = \text{count} + 1 \) is scheduled to be executed without any delay (i.e., \( \text{time} = 0 \))
Nonblocking Assignments (Cont.)

- The simulator schedules a nonblocking assignment statement to execute and continues to the next statement in the block without waiting for the nonblocking statement to complete execution.
- Nonblocking assignment statements 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.
- In the example of P-25, we mixed blocking and nonblocking assignments to illustrate their behavior.
- It is recommended that blocking and nonblocking assignments not be mixed in the same always block.
7.2 Procedural Assignments

Application of Nonblocking Assignments

- Having described the behavior of nonblocking assignments, it is important to understand why they are used in digital design.
- They are used as a method to model several concurrent data transfers that take place after a common event.
Example for Application of Nonblocking Assignments

Consider the following example where three concurrent data transfers take place at the positive edge of clock.

```verilog
always @(posedge clock)
begin
  reg1 <= #1 in1;
  reg2 <= @(negedge clock) in2 ^ in3;
  reg3 <= #1 reg1; // The old value of reg1
end
```
1. A read operation is performed on each right-hand-side variable, in1, in2, in3, and reg1, at the positive edge of clock. The right-hand-side expressions are evaluated, and the results are stored internally in the simulator.

2. The write operations to the left-hand-side variables are scheduled to be executed at the time specified by the intra-assignment delay, i.e., schedule "write" to reg1 after 1 time unit, to reg2 at the next negative edge of clock, and to reg3 after 1 time unit.

3. The order in which the write operations are executed is not important. Reg3 is assigned the old value of reg1 that was stored after the read operation, even if the write operation wrote a new value to reg1 before the write operation to reg3 was executed.
7.2 Procedural Assignments

Nonblocking 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;
Implementing Nonblocking Assignments using Blocking Assignments

// Emulate the behavior of nonblocking assignments by using temporary variables and blocking assignments

always @(posedge clock)
begin
    // Read operation
    // store values of right-hand-side expressions in temporary variables
    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
7.1 Structured Procedures
7.2 Procedural Assignments
7.3 Timing Controls
7.4 Conditional Statements
7.5 Multiway Branching
7.6 Loops
7.7 Sequential and Parallel Blocks
7.8 Generate Blocks
7.9 Examples
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 control:

1. Delay-based timing control
2. Event-based timing control
3. Level-sensitive timing control
7.3 Timing Controls

- **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.
  - Delays are specified by the symbol #.
Syntax for the delay-based timing control statement is shown below.

```
delay3 ::= # delay_value | #( delay_value [ , delay_value [ , delay_value ] ] )
delay2 ::= # delay_value | #( delay_value [ , delay_value ] )
delay_value ::= unsigned_number | parameter_identifier | specparam_identifier | mintypmax_expression
```

- Delay-based timing control can be specified by a number, identifier, or a mintypmax_expression.
- There are three types of delay control for procedural assignments: regular delay control, intra-assignment delay control, and zero delay control.
7.3 Timing Controls

- Regular Delay Control

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

- Usage of regular delay control is shown in P-38.
### Example for Regular Delay Control

```verilog
//define parameters
parameter latency = 20;
parameter delta = 2;
//define register variables
reg x, y, z, p, q;

initial
begin
    x = 0; // no delay control
    #10 y = 1; // delay control with a number. Delay execution of
               // y = 1 by 10 units
    #latency z = 0; // Delay control with identifier. Delay of 20 units
    #(latency + delta) p = 1; // Delay control with expression
    #y x = x + 1; // Delay control with identifier. Take value of y.
    #(4:5:6) q = 0; // Minimum, typical and maximum delay values.
                     //Discussed in gate-level modeling chapter.
end
```
In P-38, the execution of a procedural assignment is delayed by the number specified by the delay control.

For begin-end groups, delay is always relative to time when the statement is encountered.

Thus, \( y = 1 \) is executed 10 units after it is encountered in the activity flow.
7.3 Timing Controls

- **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.
  
  - P-41 shows the contrast between intra-assignment delays and regular delays.
Example for Intra-assignment Delay Control

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

//intra assignment 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 y at time 5 is unaffected.
end
Note the difference between intra-assignment delays and regular delays.

Regular delays defer the execution of the entire assignment.

Intra-assignment delays compute the right-hand-side expression at the current time and defer the assignment of the computed value to the left-hand-side variable.

Intra-assignment delays are like using regular delays with a temporary variable to store the current value of a right-hand-side expression.
Zero Delay Control

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 nondeterministic.

Zero delay control is a method to ensure that a statement is executed last, after all other statements in that simulation time are executed.

This is used to eliminate race conditions.

However, if there are multiple zero delay statements, the order between them is nondeterministic.

P-44 illustrates zero delay control.
7.3 Timing Controls

Example for Zero Delay Control

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

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

First Execution

Last Execution
Example for Zero Delay Control (Cont.)

- In P-44, four statements $x = 0, y = 0, x = 1, y = 1$ are to be executed at simulation time 0.
- Since $x = 1$ and $y = 1$ have #0, they will be executed last.
- Thus, at the end of time 0, $x$ will have value 1 and $y$ will have value 1.
- The order in which $x = 1$ and $y = 1$ are executed is not deterministic.
- However, using #0 is not a recommended practice.
7.3 Timing Controls

- **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:
    1. Regular event control
    2. Named event control
    3. Event OR control
    4. Level-sensitive timing control
7.3 Timing Controls

- **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.
  - The keyword `posedge` is used for a positive transition, as shown in P-48.
7.3 Timing Controls

Example for Regular Event Control

@\(\text{clock}\) q = d; //q = d is executed whenever signal clock changes value

@\(\text{posedge clock}\) 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 )

@\(\text{negedge clock}\) q = d; //q = d is executed whenever signal clock does

    // a negative transition ( 1 to 0, x or z,
    // x to 0, z to 0)

q = @\(\text{posedge clock}\) d; //d is evaluated immediately and assigned

    // to q at the positive edge of clock
Named Event Control

- Verilog provides the capability to declare an event and then trigger and recognize the occurrence of that event (see P-50).
- 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 `@`. 
Example for 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
    // When event is triggered, store all four packets of received data in data buffer
    // use concatenation operator { }
    data_buf = {data_pkt[0], data_pkt[1], data_pkt[2], data_pkt[3]};
### Event OR Control

- Sometimes a transition on 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, as shown in P-52.
Example for Event OR Control (Sensitivity List)

//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)               //if reset signal is high, set q to 0.
    q = 1'b0;
  else if (clock)       //if clock is high, latch input
    q = d;
end
■ Event OR Control (Cont.)

- Sensitivity lists can also be specified using the "," (comma) operator instead of the or operator.
- P-54 shows how the above example can be rewritten using the comma operator.
- Comma operators can also be applied to sensitivity lists that have edge-sensitive triggers.
Example for Sensitivity List with Comma Operator

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

// A positive edge triggered D flipflop with asynchronous falling
// reset can be modeled as shown below
always @(posedge clk, negedge reset) // Note use of comma operator
    if(!reset)
        q <=0;
    else
        q <=d;
When the number of input variables to a combination logic block are very large, sensitivity lists can become very cumbersome to write.

Moreover, if an input variable is missed from the sensitivity list, the block will not behave like a combinational logic block.

To solve this problem, Verilog HDL contains two special symbols: @* and @(*), both symbols exhibit identical behavior.

These special symbols are sensitive to a change on any signal that may be read by the statement group that follows this symbol.

P-56 shows an example of this special symbol.
Example for Use of @* Operator

//Combination logic block using the or operator
//Cumbersome to write and it is easy to miss one input to the block
always @(a or b or c or d or e or f or g or h or p or m)
begin
  out1 = a ? b+c : d+e;
  out2 = f ? g+h : p+m;
end

//Instead of the above method, use @(*) symbol
//Alternately, the @* symbol can be used
//All input variables are automatically included in the
//sensitivity list.
always @(*)
begin
  out1 = a ? b+c : d+e;
  out2 = f ? g+h : p+m;
end
7.3 Timing Controls

- **Level-Sensitive Timing Control**
  - Event control discussed earlier waited for the change of a signal value or the triggering of an event.
  - 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.
  - The keyword `wait` is used for level-sensitive constructs.
Example for Level-Sensitive Timing Control

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

- In the above example, the value of count_enable is monitored continuously.
- If count_enable is 0, the statement is not entered.
- If it is logical 1, the statement count = count + 1 is executed after 20 time units.
- If count_enable stays at 1, count will be incremented every 20 time units.
7.4 Conditional Statements

7.1 Structured Procedures
7.2 Procedural Assignments
7.3 Timing Controls
7.4 Conditional Statements
7.5 Multiway Branching
7.6 Loops
7.7 Sequential and Parallel Blocks
7.8 Generate Blocks
7.9 Examples
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.
- For formal syntax, see Appendix D, Formal Syntax Definition.
Three Type of Conditional Statements

// Type 1 conditional statement. No else statement. Statement executes or does not execute.
if (<expression>) true_statement ;

// Type 2 conditional statement. One else statement. Either true_statement or 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 is executed.
if (<expression1>) true_statement1 ;
else if (<expression2>) true_statement2 ;
else if (<expression3>) true_statement3 ;
else default_statement ;
Three Type of Conditional Statements

- The `<expression>` is evaluated.
- If it is true (1 or a non-zero value), the `true_statement` is executed.
- If it is false (zero) or ambiguous (x), the `false_statement` is executed.
- The `<expression>` can contain any operators mentioned in Table 6-1 on page 96.
- Each `true_statement` or `false_statement` can be a single statement or a block of multiple statements.
- A block must be grouped, typically by using keywords `begin` and `end`.
- A single statement need not be grouped.
7.4 Conditional Statements

Conditional Statement Examples

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

//Type 2 statements
if (number_queued < MAX_QDEPTH)
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 if(alu_control == 2)
  y = x * z;
else
  $display("Invalid ALU control signal");
7.5 Multiway Branching

7.1 Structured Procedures
7.2 Procedural Assignments
7.3 Timing Controls
7.4 Conditional Statements
7.5 Multiway Branching
7.6 Loops
7.7 Sequential and Parallel Blocks
7.8 Generate Blocks
7.9 Examples
7.5 Multiway Branching

- Multiway Branching
  - Conditional Statements, there were many alternatives, from which one was chosen.
  - The nested 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.
The keywords **case**, **endcase**, and **default** are used in the case statement.

**Syntax:**

```plaintext
case (expression)
  alternative1: statement1;
  alternative2: statement2;
  alternative3: statement3;
  ...
  ...
  default: default_statement;
endcase
```
Case Statement (Cont.)

- Each of statement1, statement2 ..., default_statement can be a single statement or a block of multiple statements.
- A block of multiple statements must be grouped by keywords begin and end.
- For the first alternative that matches, the corresponding statement or block is executed.
- If none of the alternatives matches, the default_statement is executed.
- The default_statement is optional.
- Placing of multiple default statements in one case statement is not allowed.
- The case statements can be nested.
7.5 Multiway Branching

- Case Statement Equivalent to P-63

//Execute statements based on the ALU control signal
reg [1:0] alu_control;
...
...
case (alu_control)
  2'd0 : y = x + z;
  2'd1 : y = x - z;
  2'd2 : y = x * z;
  default : $display("Invalid ALU control signal");
endcase
7.5 Multiway Branching

- **Case Statement** (Cont.)

  - The case statement can also act like a many-to-one multiplexer.
  - The I/O ports are unchanged (Ref Text Book P-106).
  - Notice that an 8-to-1 or 16-to-1 multiplexer can also be easily implemented by case statements.
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
The case statement compares 0, 1, x, and z values in the expression and the alternative 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. In P-72, we will define a 1-to-4 demultiplexer for which outputs are completely specified, that is, definitive results are provided even for x and z values on the select signal.
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)
  case ({s1, s0}) //Switch based on control signals
      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
  endcase

//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'bxz, 2'bxz, 2'bxz, 2'bxz, 2'bxz :
  begin
    out0 = 1'bx; out1 = 1'bx; out2 = 1'bx; out3 = 1'bx;
  end
2'b0z, 2'b1z, 2'b0z, 2'b1z, 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
7.5 Multiway Branching

- 4-to-1 Multiplexer with Case Statement (Cont.)

In the demultiplexer shown in P-72, multiple input signal combinations such as 2'bz0, 2'bz1, 2'bzz, 2'b0z, and 2'b1z that cause the same block to be executed are put together with a comma (,) symbol.
There are two variations of the case statement. They are denoted by keywords, casex and casez.

1. casez treats all z values in the case alternatives or the case expression as don't cares. All bit positions with z can also be represented by ? in that position.

2. casex treats all x and z values in the case item or the case expression as don't cares.

The use of casex and casez allows comparison of only non-x or -z positions in the case expression and the case alternatives.

P-75 presents its example.

Only one bit is considered to determine the next state and the other bits are ignored.
### 7.5 Multiway Branching

#### casex, casez Keywords

```
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;
endcase
```

- Thus, an input encoding = 4'b10xz would cause next_state = 3 to be executed.
7.6 Loops

7.1 Structured Procedures
7.2 Procedural Assignments
7.3 Timing Controls
7.4 Conditional Statements
7.5 Multiway Branching
7.6 Loops
7.7 Sequential and Parallel Blocks
7.8 Generate Blocks
7.9 Examples
There are four types of looping statements in Verilog: while, for, repeat, and forever.

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.
7.6 Loops

- **While Loop**
  - The keyword `while` is used to specify this loop.
  - The while loop executes until the while-expression is not true.
  - If the loop is entered when the while-expression is not true, the loop is not executed at all.
  - Any logical expression can be specified with these operators.
  - If multiple statements are to be executed in the loop, they must be grouped typically using keywords begin and end.
  - P-79 illustrates the use of the while loop.
Example 1 for While Loop

//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
Example 2 for While Loop

//Illustration 2: Find the first bit with a value 1 in flag (vector variable)
'define TRUE 1'b1;
'define FALSE 1'b0;
reg [15:0] flag;
integer i; //integer to keep count
reg continue;

initial
begin
  flag = 16'b 0010_0000_0000_0000;
  i = 0;
  continue = 'TRUE;

while((i < 16) && continue ) //Multiple conditions using operators.
begin
  if (flag[i])
  begin
    $display("Encountered a TRUE bit at element number %d", i);
    continue = 'FALSE;
  end
  i = i + 1;
end
7.6 Loops

- **For Loop**
  - The keyword `for` is used to specify this loop. The for loop contains three parts:
    1. An initial condition
    2. A check to see if the terminating condition is true
    3. A procedural assignment to change value of the control variable
  - The for loop provides a more compact loop structure than the while loop.
  - The while loop is more general-purpose than the for loop.
  - The for loop cannot be used in place of the while loop in all situations.
For Loop Example

integer count;

initial
  for (count=0; count < 128; count = count + 1)
    $display("Count = %d", count);
7.6 Loops

- For Loop Example (Cont.)

- **for** loops can also be used to initialize an array or memory.

```plaintext
//Initialize array elements
'define MAX_STATES 32
integer state [0: 'MAX_STATES-1];  //Integer array state with elements 0:31
integer i;

initial
begin
    for(i = 0; i < 32; i = i + 2) //initialize all even locations with 0
        state[i] = 0;
    for(i = 1; i < 32; i = i + 2) //initialize all odd locations with 1
        state[i] = 1;
end
```

- **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
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 (while loop is used for that purpose).
- A repeat construct must contain a number, which can be a constant, a variable or a signal value.
- If the number is a variable or signal value, it is evaluated only when the loop starts and not during the loop execution.
Example 1 for Repeat Loop

- The counter in P-79 can be expressed with the repeat loop, as shown in the following.
- Illustration 2 in P-86 shows how to model a data buffer that latches data at the positive edge of clock for the next eight cycles after it receives a data start signal.

```verbatim
//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
```
Illustration 2: Data buffer module example
After it receives a data_start signal.
Reads data for next 8 cycles.

module data_buffer(data_start, data, clock);

parameter cycles = 8;
input data_start;
input [15:0] data;
input clock;

reg [15:0] buffer [0:7];
integer i;

always @(posedge clock)
begin
if(data_start) //data start signal is true
begin
i = 0;
repeat(cycles) //Store data at the posedge of next 8 clock cycles
begin
@ (posedge clock) buffer[i] = data; //waits till next posedge to latch data
i = i + 1;
end
end
endmodule
7.6 Loops

<table>
<thead>
<tr>
<th>Forever Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>The keyword <code>forever</code> is used to express this loop.</td>
</tr>
<tr>
<td>The loop does not contain any expression and executes forever until the <code>$finish</code> task is encountered.</td>
</tr>
<tr>
<td>The loop is equivalent to a while loop with an expression that always evaluates to true, e.g., while (1).</td>
</tr>
<tr>
<td>A <code>forever</code> loop can be exited by use of the <code>disable</code> statement.</td>
</tr>
<tr>
<td>A <code>forever</code> loop is typically used in conjunction with timing control constructs.</td>
</tr>
<tr>
<td>If timing control constructs are not used, the Verilog simulator would execute this statement infinitely without advancing simulation time and the rest of the design would never be executed.</td>
</tr>
</tbody>
</table>
Example for Forever Loop

// 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

// Example 2: Synchronize two register values at every positive edge of clock
reg clock;
reg x, y;

initial
  forever @(posedge clock) x = y;
7.1 Structured Procedures
7.2 Procedural Assignments
7.3 Timing Controls
7.4 Conditional Statements
7.5 Multiway Branching
7.6 Loops
7.7 Sequential and Parallel Blocks
7.8 Generate Blocks
7.9 Examples
Sequential and Parallel Blocks

- Block statements (begin and end) are used to group multiple statements to act together as one.
- In this section we discuss the block types: sequential blocks and parallel blocks.
- We also discuss three special features of blocks: named blocks, disabling named blocks, and nested blocks.
7.7 Sequential and Parallel Blocks

- **Block Types**

- There are two types of blocks: sequential blocks and parallel blocks.
### 7.7 Sequential and Parallel Blocks

#### Sequential Block

- The keywords `begin` and `end` are used to group statements into sequential blocks.
- Sequential blocks have the following characteristics:
  1. 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 nonblocking assignments with intra-assignment timing control).
  2. If delay or event control is specified, it is relative to the simulation time when the previous statement in the block completed execution.
Example 1 for Sequential Block

- Statements in the sequential block execute in order.
- Example 1, the final values are $x = 0$, $y = 1$, $z = 1$, $w = 2$ at simulation time 0.

```verbatim
// Example 1: Sequential block without delay
reg x, y;
reg [1:0] z, w;

initial
begin
    x = 1'b0;
y = 1'b1;
z = {x, y};
w = {y, x};
end
```
7.7 Sequential and Parallel Blocks

Example 2 for Sequential Block

Example 2, the final values are the same except that the simulation time is 35 at the end of the block.

```verilog
// Example 2: Sequential blocks with delay.
reg x, y;
reg [1:0] 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}; // completes at simulation time 35
end
```
Parallel Blocks

Parallel blocks, specified by keywords *fork* and *join*, provide interesting simulation features. Parallel blocks have the following characteristics:

1. Statements in a parallel block are executed concurrently.

2. Ordering of statements is controlled by the delay or event control assigned to each statement.

3. If 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.
Example 1 for Parallel Blocks

- Conversion P-94 Example 2 to a parallel block.
- The result of simulation remains the same except that all statements start in parallel at time 0.
- The block finishes at time 20 instead of time 35.

```verilog
//Example 1: Parallel blocks with delay.
reg x, y;
reg [1:0] 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 10
  #20 w = {y, x}; //completes at simulation time 20
join
```
Example 2 for Parallel Blocks

- Race conditions have been deliberately introduced in this example.
- All statements start at simulation time 0.
- The order in which the statements will execute is not known.
- Variables z and w will get values 1 and 2 if $x = 1'b0$ and $y = 1'b1$ execute first.
- Variables z and w will get values $2'bxx$ and $2'bxx$ if $x = 1'b0$ and $y = 1'b1$ execute last.
- The result of z and w is nondeterministic and dependent on the simulator implementation.
- In simulation time, all statements in the fork-join block are executed at once.
In reality, CPUs running simulations can execute only one statement at a time.

Different simulators execute statements in different order.

The race condition is a limitation of today's simulators, not of the fork-join block.
Example 2 for Parallel Blocks (Cont.)

// Parallel blocks with deliberate race condition
reg x, y;
reg [1:0] z, w;

initial
fork
  x = 1'b0;
y = 1'b1;
z = {x, y};
w = {y, x};
join

- 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 back into a single flow.
- Independent flows operate concurrently.
Special Features of Blocks

- Three special features available with block statements:
  1. Nested blocks
  2. Named blocks
  3. Disabling of named blocks
7.7 Sequential and Parallel Blocks

**Nested Blocks**

- Blocks can be nested.
- Sequential and parallel blocks can be mixed, as shown in the follows:

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

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

initial
begin: block1 //sequential block named block1
integer i; //integer i is static and local to block1
    // can be accessed by hierarchical name, top.block1.i
    ...
    ...
end

initial
fork: block2 //parallel block named block2
reg i; // register i is static and local to block2
    // can be accessed by hierarchical name, top.block2.i
    ...
    ...
join
```
7.7 Sequential and Parallel Blocks

- **Disabling Named Blocks**

  - The keyword **disable** provides a way to terminate the execution of a named block.
  
  - Disable can be used to get out of loops, handle error conditions, or control execution of pieces of code, based on a control signal.
  
  - Disabling a block causes the execution control to be passed to the statement immediately succeeding the block.
  
  - For C programmers, this is very similar to the break statement used to exit a loop.
  
  - The difference is that a break statement can break the current loop only, whereas the keyword **disable** allows disabling of any named block in the design.
Example for Disabling Named Blocks

// 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
end