MODULE -4

BEHAVIORAL MODELING

4.1 Objectives

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

4.2 Structured Procedures

There are two structured procedure statements in Verilog: always and initial. These statements are the two most basic statements in behavioral modeling. All other behavioral statements can appear only inside these structured procedure statements. 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. Each 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. The fundamental difference between the two statements is explained in the following sections

4.2.1 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. Each block finishes execution independently of other blocks.

Multiple behavioral statements must be grouped, typically using the keywords begin and end. If there is only one behavioral statement, grouping is not necessary. This is similar to the begin-end blocks in Pascal programming language or the { } grouping in the C programming language. Example 4.1 illustrates the use of the initial statement.

Example 4.1:Initial Statement

```
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

128

#50 $finish;

endmodule
```

In the above example, the three initial statements start to 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. Thus, the execution sequence of the statements inside the initial blocks will be as follows.

```
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;
```

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 discussion 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

Variables can be initialized when they are declared. Example 4-2 shows such a declaration.

Example 4-2 Initial Value Assignment

```
//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;
```

Combined Port/Data Declaration and Initialization

The combined port/data declaration can also be combined with an initialization. Example 4-3 shows such a declaration.

Example 4-3 Combined Port/Data Declaration and Variable Initialization

```
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

ANSI C style port declaration can also be combined with an initialization. Example 4-4 shows such a declaration.

Example 4-4 Combined ANSI C Port Declaration and Variable 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
```

4.2.2 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 4-5 illustrates one method to model a clock generator in Verilog.

Example 4-5 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
```

In Example 4-5, the always statement starts at time 0 and executes the statement clock = ~clock every 10 time units. Notice that the initialization of clock has to be done inside a separate initial statement. If we put the initialization of clock inside the always block, clock will be initialized every time the always is entered. Also, the simulation must be halted inside an initial statement. If there is no $stop or $finish statement to halt the simulation, the clock generator will run forever. C programmers might draw an analogy between the always block and an infinite loop.

But hardware designers tend to view it as a continuously repeated activity in a digital circuit starting from power on. The activity is stopped only by power off ($finish) or by an interrupt ($stop).

4.3 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, 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. The

syntax for the simplest form of procedural assignment is shown below.

assignment ::= variable_lvalue = [delay_or_event_control] expression

The left-hand side of a procedural assignment <lvalue> can be one of the following:

• A 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

The right-hand side can be any expression that evaluates to a value. In behavioral modeling, all operators can be used in behavioral expressions.

There are two types of procedural assignment statements: blocking and nonblocking.

4.3.1 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. The = operator is used to specify blocking assignments.

Example 4-6 Blocking Statements

```
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
```

In Example 4-6, 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. The simulation times at which the statements are executed are as follows:

• All statements x = 0 through reg_b = reg_a are executed at time 0

• Statement reg_a[2] = 0 at time = 15

• Statement reg_b[15:13] = {x, y, z} at time = 25

• Statement count = count + 1 at time = 25

• Since there is a delay of 15 and 10 in the preceding statements, count = count + 1 will be executed at time = 25 units

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.

4.3.2 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. Note that this operator has the same symbol as a relational operator, less_than_equal_to. The operator <= is interpreted as a relational operator in an expression and as an assignment operator in the context of a nonblocking assignment. To illustrate the behavior of nonblocking statements and its difference from blocking statements, let us consider Example 4-7, where we convert some blocking assignments to nonblocking assignments, and observe the behavior.

Example 4-7 Nonblocking Assignments

```
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
//to part select of a vector
count <= count + 1; //Assignment to an integer (increment)
end
```

In this example, the statements x = 0 through reg_b = reg_a are executed sequentially at time 0. Then the three nonblocking assignments are processed at the same simulation time.

1. reg_a[2] = 0 is scheduled to execute after 15 units (i.e., time = 15)

2. reg_b[15:13] = {x, y, z} is scheduled to execute after 10 time units (i.e., time = 10)

3. count = count + 1 is scheduled to be executed without any delay (i.e., time = 0) Thus, 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. Typically, 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 above, we mixed blocking and nonblocking assignments to illustrate their behavior. However, it is recommended that blocking and nonblocking assignments not be mixed in the same always block.

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

```
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, the following sequence takes place for the nonblocking assignments.

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 in each assignment, that is, 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 write operations are executed at the scheduled time steps. The order in which the write operations are executed is not important because the internally stored right-hand-side expression values are used to assign to the left-hand-side values. For example, note that 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.

Thus, the final values of reg1, reg2, and reg3 are not dependent on the order in which the assignments are processed.

To understand the read and write operations further, consider Example 4-8, which is intended to swap the values of registers a and b at each positive edge of clock, using two concurrent always blocks.

Example 4-8 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;
135
//Illustration 2: Two concurrent always blocks with nonblocking
//statements
always @(posedge clock)
a <= b;
always @(posedge clock)
b <= a;
```

In Example 4-8, in Illustration 1, there is a race condition when blocking statements are used. Either a = b would be executed before b = a, or vice versa, depending on the simulator implementation. Thus, values of registers a and b will not be swapped. Instead, both registers will get the same value (previous value of a or b), based on the Verilog simulator implementation.

However, nonblocking statements used in Illustration 2 eliminate the race condition. At the positive edge of clock, the values of all right-hand-side variables are "read," and the right-hand-side expressions are evaluated and stored in temporary variables. During the write operation, the values stored in the temporary variables are

assigned to the left-handside variables. Separating the read and write operations ensures that the values of registers a and b are swapped correctly, regardless of the order in which the write operations are performed. Example 4-9 shows how nonblocking assignments shown in Illustration 2 could be emulated using blocking assignments.

Example 4-9 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
```

For digital design, use of nonblocking assignments in place of blocking assignments is highly recommended in places where concurrent data transfers take place after a common event. In such cases, blocking assignments can potentially cause race conditions because the final result depends on the order in which the assignments are evaluated. Nonblocking assignments can be used effectively to model concurrent data transfers because the final result is not dependent on the order in which the assignments are evaluated. Typical applications of nonblocking assignments include pipeline modeling and modeling of several mutually exclusive data transfers. On the downside, nonblocking assignments can potentially cause degradation in the simulator performance and increase in memory usage.

4.4 Timing Controls

Various behavioral timing control constructs are available in Verilog. 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: delay-based timing control, event-based timing control, and level-sensitive timing control.

4.4.1 Delay-Based Timing Control

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

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 Example 4-10.

Example 4-10 Regular Delay Control

```
//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 Example 4-10, 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.

Intra-assignment delay control

Instead of specifying delay control to the left of the assignment, it is possible to assign a delay to the right of the assignment operator. Such delay specification alters the flow of activity in a different manner. Example 4-11 shows the contrast between intra-assignment delays and regular delays.

Example 4-11 Intra-assignment Delays

```
//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 righthand- 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. Example 4-12 illustrates zero delay control.

Example 4-12 Zero Delay Control

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

```
#0 y = 1;
end
```

In Example 4-12, four statements?x = 0, y = 0, x = 1, y = 1?are to be executed at simulation time 0. However, 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. The above example was used as an illustration. However, using #0 is not a recommended practice.

4.4.2 Event-Based Timing Control

An event is the change in the value on a register or a net. Events can be utilized to trigger execution of a statement or a block of statements. There are four types of event-based timing control: regular event control, named event control, event OR control, and level sensitive timing control.

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 Example 4-13.

Example 4-13 Regular Event Control

```
@(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 or 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

Verilog provides the capability to declare an event and then trigger and recognize the occurrence of that event (see Example 4-14). 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 4-14 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 Example 4-15.

Example 4-15 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
```

Sensitivity lists can also be specified using the "," (comma) operator instead of the or operator. Example 4-16 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 4-16 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

Example 4-17 shows an example of this special symbol for combinational logic sensitivity lists.

IEEE Standard Verilog Hardware Description Language document for details and restrictions on the @* and @(*) symbols.

Example 4-17 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
```

4.4.3 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.

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.

4.5 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. Usage of conditional statements is shown below.

//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 ;

The <expression> is evaluated. If it is true (1 or a non-zero value), the true_statement is executed. However, if it is false (zero) or ambiguous (x), the false_statement is executed. The <expression> can contain any operators. 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.

Example 4-18 Conditional Statement Examples

```
//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 if(alu_control == 2)
y = x * z;
else
$display("Invalid ALU control signal");
```

4.6 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.

4.6.1 case Statement

The keywords case, endcase, and default are used in the case statement..

case (expression)

alternative1: statement1;

alternative2: statement2;

alternative3: statement3;

...
...
default: default_statement;

endcase

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. The expression is compared to the alternatives in the order they are written. 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. The following Verilog code implements the type 3 conditional statement in Example 4-18.

```
//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
```

The case statement can also act like a many-to-one multiplexer. To understand this, let us model the 4-to-1 multiplexer, using case statements. The I/O ports are unchanged. Notice that an 8-to-1 or 16-to-1 multiplexer can also be easily implemented by case statements.

Example 4-19 4-to-1 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
```

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 Example 4- 20, 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.

Example 4-20 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)

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

//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
```

In the demultiplexer shown above, 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.

4.6.2 casex, casez Keywords

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

• 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.

• 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. Example 4-21 illustrates the decoding of state bits in a finite state machine using a casex statement. The use of casez is similar. Only one bit is considered to determine the next state and the other bits are ignored.

Example 4-21 casex Use

```
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.

4.7 Loops

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.

4.7.1 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. Each expression can contain the operators. 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. Example 4-22 illustrates the use of the while loop.

Example 4-22 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

//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;

148

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
end
```

4.7.2 for Loop

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

• An initial condition

• A check to see if the terminating condition is true

• A procedural assignment to change value of the control variable

The counter described in Example 4-22 can be coded as a for loop (Example 4-23). The initialization condition and the incrementing procedural assignment are included in the for loop and do not need to be specified separately. Thus, the for loop provides a more compact loop structure than the while loop. Note, however, that 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.

Example 4-23 For Loop

```
integer count;

initial

for ( count=0; count < 128; count = count + 1)

$display("Count = %d", count);

for loops can also be used to initialize an array or memory, as shown below.

//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.

4.7.3 Repeat Loop

The keyword repeat is used for this loop. The repeat construct executes the loop a fixed number of times. A repeat construct cannot be used to loop on a general logical expression. A 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. However, if the number is a variable or signal value, it is evaluated only when the loop starts and not during the loop execution.

The counter in Example 4-22 can be expressed with the repeat loop, as shown in

Illustration 1 in Example 4-24. Illustration 2 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.

Example 4-24 Repeat Loop

```
//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;

150

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

end

endmodule
```

4.7.4 Forever loop

The keyword forever is used to express this loop. The loop does not contain any expression and executes forever until the $finish task is encountered. The loop is equivalent to a while loop with an expression that always evaluates to true, 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 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. Example 4-25 explains the use of the forever statement.

Example 4-25 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;
```

4.8 Sequential and Parallel Blocks

Block statements are used to group multiple statements to act together as one. In previous examples, we used keywords begin and end to group multiple statements. Thus, we used sequential blocks where the statements in the block execute one after another. 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.

4.8.1 Block Types

There are two types of blocks: sequential blocks and parallel blocks.

Sequential blocks

The keywords begin and end are used to group statements into sequential blocks.

Sequential blocks 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 nonblocking assignments 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.

We have used numerous examples of sequential blocks in this book. Two more examples of sequential blocks are given in Example 4-26. Statements in the sequential block execute in order. In Illustration 1, the final values are x = 0, y= 1, z = 1, w = 2 at simulation time 0. In Illustration 2, the final values are the same except that the simulation time is 35 at the end of the block.

Example 4-26 Sequential Blocks

```
//Illustration 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
//Illustration 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:

• Statements in a parallel block are executed concurrently.

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

• If delay or event control is specified, it is relative to the time the block was entered.

Notice the fundamental difference between sequential and parallel blocks. All statements in a parallel block start at the time when the block was entered. Thus, the order in which the statements are written in the block is not important.

Let us consider the sequential block with delay in Example 4-26 and convert it to a parallel block. The converted Verilog code is shown in Example 4-27. The result of simulation remains the same except that all statements start in parallel at time 0. Hence, the block finishes at time 20 instead of time 35.

Example 4-27 Parallel Blocks

```
//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
```

Parallel blocks provide a mechanism to execute statements in parallel. However, it is important to be careful with parallel blocks because of implicit race conditions that might arise if two statements that affect the same variable complete at the same time. Shown below is the parallel version of Illustration 1 from Example 4-26. 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. Thus, 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. However, in reality, CPUs running simulations can execute only one statement at a time. Different simulators execute statements in different order. Thus, the race condition is a limitation of today's simulators, not of the fork-join block.

```
//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.

4.8.2 Special Features of Blocks

We discuss three special features available with block statements: nested blocks, named blocks, and disabling of named blocks.

Nested blocks

Blocks can be nested. Sequential and parallel blocks can be mixed, as shown in Example 4-28.

Example 4-28 Nested Blocks

```
//Nested blocks

initial

begin

x = 1'b0;

154

fork

#5 y = 1'b1;

#10 z = {x, y};

join

#20 w = {y, x};

end
```

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.

Example 4-29 shows naming of blocks and hierarchical naming of blocks.

Example 4-29 Named Blocks

```
//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
```

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.

4.9: Outcomes

After completion of the module the students are able to:

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

4.10: Recommended Questions

1. Describe the following statements with an example: initial and always
2. What are blocking and non-blocking assignment statements? Explain with examples.
3. With syntax explain conditional, branching and loop statements available in Verilog HDL behavioural description.
4. Describe sequential and parallel blocks of Verilog HDL.
5. Write Verilog HDL program of 4:1 mux using CASE statement.
6. Write Verilog HDL program of 4:1 mux using If-else statement.
7. Write Verilog HDL program of 4-bit synchronous up counter.
8. Write Verilog HDL program of 4-bit asynchronous down counter.
9. Write Verilog HDL program to simulate traffic signal controller