8

Behavioral Modeling

The language constructs introduced so far allow hardware to be described at a relatively detailed level. Modeling a circuit with logic gates and continuous assignments reflects quite closely the logic structure of the circuit being modeled; however, these constructs do not provide the power of abstraction necessary for describing complex high level aspects of a system. The procedural constructs described in this chapter are well suited to tackling problems such as describing a microprocessor or implementing complex timing checks.

This chapter covers the following topics:

- Section 8.1 provides a brief overview of a behavioral model to provide a context for understanding the behavioral constructs detailed in this chapter.
- Section 8.2 discusses the two statements used to describe procedures (`always` and `initial`).
- Section 8.3 discusses procedural assignment statements.
- Section 8.4 describes conditional statements (`if` and `if-else`).
- Section 8.5 describes multi-way decision statements (`if-else-if` and `case`).
- Section 8.6 describes loop statements (`forever`, `repeat`, `while`, and `for`).
- Section 8.7 discusses procedural timing controls.
- Section 8.8 describes sequential block statements (`begin-end`) and parallel block statements (`fork-join`).
- Section 8.9 contains two complete behavioral model examples that illustrate the behavioral constructs introduced in this chapter.

8.1 Behavioral Model Overview

Verilog behavioral models contain procedural statements that control the simulation and manipulate variables of the data types previously described. These statements are contained within procedures. Each procedure has an activity flow associated with it.

The activity starts at the control constructs `initial` and `always`. Each `initial` statement and each `always` statement starts a separate activity flow. All of the activity flows are concurrent, allowing you to model the inherent concurrence of hardware.

Example 8-1 is a complete Verilog behavioral model.

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

Example 8-1: Example of a behavioral model

During simulation of this model, all of the flows defined by the `initial` and `always` statements start together at simulation time zero. The `initial` statements execute once, and the `always` statements execute repetitively.

In this model, the register variables `a` and `b` initialize to binary 1 and 0 respectively at simulation time zero. The `initial` statement is then complete and does not execute again during this simulation run. This `initial` statement contains a `begin-end` block (also called a sequential block) of statements. In this `begin-end` block, `a` is initialized first, followed by `b`.

The `always` statements also start at time zero, but the values of the variables do not change until the times specified by the delay controls (introduced by #) have gone by. Thus, register `a` inverts after 50 time units, and register `b` inverts after 100 time units. Since the `always` statements repeat, this model produces two square waves. Register `a` toggles with a period of 100 time units, and register `b` toggles with a period of 200 time units. The two `always` statements proceed concurrently throughout the entire simulation run.

8.2 Structured Procedures

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

- `always` statement
- `initial` statement
- task
- function

Tasks and functions are procedures that are enabled from one or more places in other procedures. Tasks and functions are covered in detail in Chapter 9, *Tasks and Functions*.

The `initial` and `always` statements are enabled at the beginning of simulation. The `initial` statement executes only once and its activity dies when the statement has finished. The `always` statement executes repeatedly. Its activity dies only when the simulation is terminated. There is no limit to the number of `initial` and `always` blocks that can be defined in a module.

8.2.1 `always` Statement

The `always` statement repeats continuously throughout the whole simulation run. Syntax 8-1 shows the syntax for the `always` statement.

```
<always_statement>
        ::= always <statement>
```

Syntax 8-1: Syntax for `always` statement

The `always` statement, because of its looping nature, is only useful when used in conjunction with some form of timing control. If an `always` statement provides no means for time to advance, the `always` statement creates a simulation deadlock condition. The following code, for example, creates an infinite zero-delay loop:

```
always areg = ~areg;
```

Providing a timing control to this code creates a potentially useful description—as in the following example:

```
always #half_period areg = ~areg;
```

8.2.2 `initial` Statement

The `initial` statement is similar to the `always` statement, except that it is executed only once. The syntax for the `initial` statement is as follows:

```
<initial_statement>
        ::= initial <statement>
```

Syntax 8-2: Syntax for `initial` statement

Example 8-2 illustrates use of the `initial` statement for initialization of variables at the start of simulation.

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

Example 8-2: Use of `initial` statement

Another typical usage of the `initial` statement is specification of waveform descriptions that execute once to provide stimulus to the main part of the circuit being simulated. Example 8-3 illustrates this usage.

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

Example 8-3: Another use for `initial` statement

8.3 Procedural Assignments

As described in Chapter 5, *Assignments*, procedural assignments are for updating `reg`, `integer`, `time`, and memory variables.

There is a significant difference between procedural assignments and continuous assignments, as described below:

- Continuous assignments drive net variables and are evaluated and updated whenever an input operand changes value.
- Procedural assignments update the value of register variables under the control of the procedural flow constructs that surround them.

The right-hand side of a procedural assignment can be any expression that evaluates to a value. However, part-selects on the right-hand side must have constant indices. The left-hand side indicates the variable that receives the assignment from the right-hand side. The left-hand side of a procedural assignment can take one of the following forms:

- **register, integer, real, or time variable:**
  An assignment to the name reference of one of these data types.
- **bit-select of a register, integer, real, or time variable:**
  An assignment to a single bit that leaves the other bits untouched.
- **part-select of a register, integer, real, or time variable:**
  A part-select of two or more contiguous bits that leaves the rest of the bits untouched. For the part-select form, only *constant* expressions are legal.
- **memory element:**
  A single word of a memory. Note that bit-selects and part-selects are illegal on memory element references.
- **concatenation of any of the above:**
  A concatenation of any of the previous four forms can be specified, which effectively partitions the result of the right-hand side expression and assigns the partition parts, in order, to the various parts of the concatenation.

Note: Assignment to a register differs from assignment to a `real`, `time`, or `integer` variable when the right-hand side evaluates to fewer bits than the left-hand side. *Assignment to a register does not sign-extend.* Registers are unsigned; if you assign a register to an integer, real, or time variable, the variable will not sign-extend.

The Verilog HDL contains two types of procedural assignment statements:

- blocking procedural assignment statements
- non-blocking procedural assignment statements

Blocking and non-blocking procedural assignment statements specify different procedural flow in sequential blocks.

8.3.1 Blocking Procedural Assignments

A blocking procedural assignment statement must be executed before the execution of the statements that follow it in a sequential block (see Section 8.8.1 on page 8-39). A blocking procedural assignment statement does not prevent the execution of statements that follow it in a parallel block (see Section 8.8.2 on page 8-41).

Syntax:

The syntax for a blocking procedural assignment is as follows:

```
<lvalue> = <timing_control> <expression>
```

Where `lvalue` is a data type that is valid for a procedural assignment statement, = is the assignment operator, and `timing_control` is the optional intra-assignment delay. The `timing_control` delay can be either a delay control (for example, `#6`) or an event control (for example, `@(posedge clk)`). The `expression` is the right-hand side value the simulator assigns to the left-hand side.

Example 8-4 shows examples of blocking procedural assignments.

```
rega = 0;
rega[3] = 1;            // a bit-select
rega[3:5] = 7;          // a part-select
mema[address] = 8'hff;  // assignment to a memory
// element
{carry, acc} = rega + regb; // a concatenation
```

Example 8-4: Examples of blocking procedural assignments

8.3.2 The Non-Blocking Procedural Assignment

The non-blocking procedural assignment allows you to schedule assignments without blocking the procedural flow. You can use the non-blocking procedural statement whenever you want to make several register assignments within the same time step without regard to order or dependance upon each other.

Syntax:

The syntax for a non-blocking procedural assignment is as follows:

```
<lvalue> <= <timing_control> <expression>
```

Where `lvalue` is a data type that is valid for a procedural assignment statement, `<=` is the non-blocking assignment operator, and `timing_control` is the optional intra-assignment timing control. The `timing_control` delay can be either a delay control (for example, `#6`) or an event control (for example, `@(posedge clk)`). The `expression` is the right-hand side value the simulator assigns to the left-hand side.

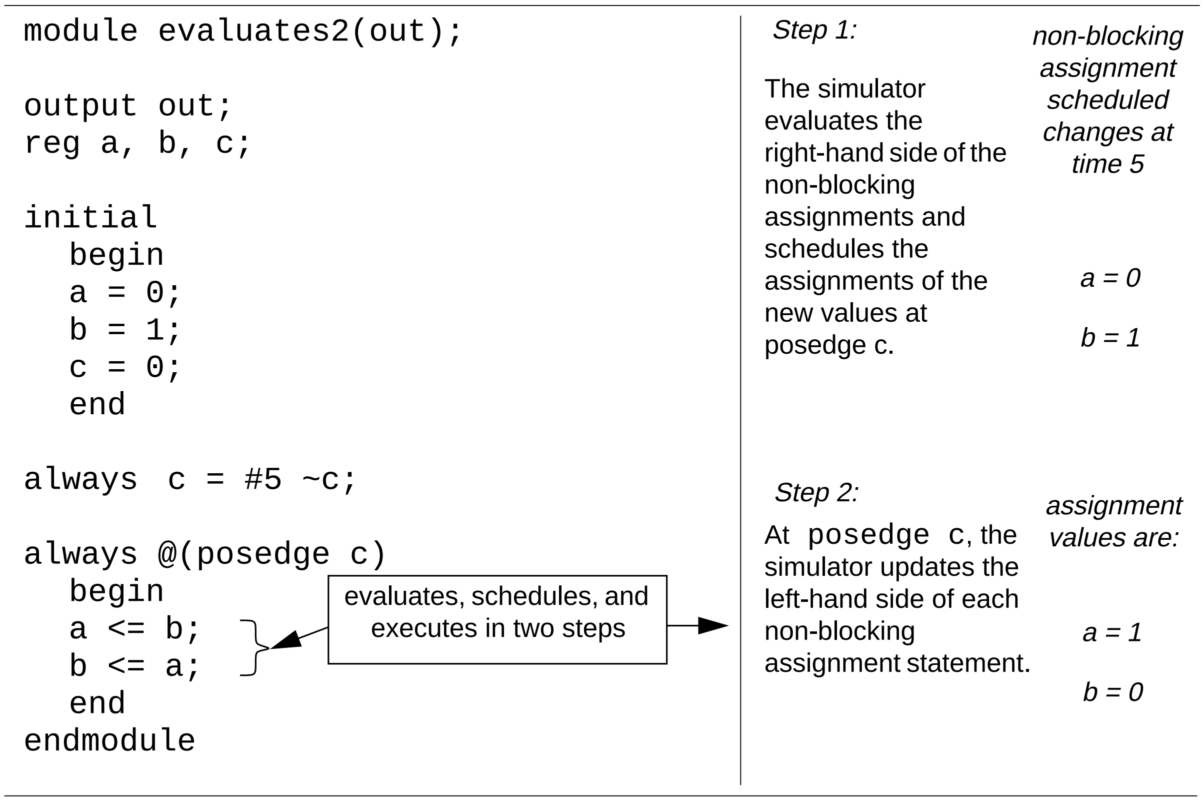
The non-blocking assignment operator is the same operator the simulator uses for the less-than-or-equal relational operator. The simulator interprets the <= operator to be a relational operator when you use it in an expression, and interprets the <= operator to be an assignment operator when you use it in a non-blocking procedural assignment construct.

How the simulator evaluates non-blocking procedural assignments

When the simulator encounters a non-blocking procedural assignment, the simulator evaluates and executes the non-blocking procedural assignment in two steps as follows:

1. The simulator evaluates the right-hand side and schedules the assignment of the new value to take place at a time specified by a procedural timing control.
2. At the end of the time step, in which the given delay has expired or the appropriate event has taken place, the simulator executes the assignment by assigning the value to the left-hand side.

These two steps are shown in Example 8-5.



```
module evaluates2(out);

output out;
reg a, b, c;

initial
  begin
  a = 0;
  b = 1;
  c = 0;
  end

always  c = #5 ~c;

always @(posedge c)
  begin
  a <= b;
  b <= a;
  end
endmodule
```

Step 1:

The simulator evaluates the right-hand side of the non-blocking assignments and schedules the assignments of the new values at posedge c.

non-blocking assignment scheduled changes at time 5

a = 0

b = 1

Step 2:

At `posedge c`, the simulator updates the left-hand side of each non-blocking assignment statement.

assignment values are:

a = 1

b = 0

Example 8-5: How the simulator evaluates non-blocking procedural assignments

At the end of the time step means that the non-blocking assignments are the last assignments executed in a time step—with one exception. Non-blocking assignment events can create blocking assignment events. The simulator processes these blocking assignment events after the scheduled non-blocking events.

Unlike a regular event or delay control, the non-blocking assignment does not block the procedural flow. The non-blocking assignment evaluates and schedules the assignment, but does not block the execution of subsequent statements in a begin-end block, as shown in Example 8-6.

```
//non_block1.v
module non_block1(out,);
//input
output out;
reg a, b, c, d, e, f;

//blocking assignments
initial begin
   a = #10 1;
   b = #2 0;
   c = #4 1;
   end
//non-blocking assignments
   initial begin
   d <= #10 1;
   e <= #2 0;
   f <= #4 1;
   end

   initial begin
$monitor ($time, ,"a = %b b = %b c = %b
 d = %b e = %b f = %b", a,b, c, d,e, f);
#100 $finish;
   end
endmodule
```

The simulator assigns `1` to register `a` at simulation time `10`, assigns `0` to register `b` at simulation time `12`, and assigns `1` to register `c` at simulation time `16`.

The simulator assigns `1` to register `d` at simulation time `10`, assigns `0` to register `e` at simulation time `2`, and assigns `1` to register `f` at simulation time `4`.

non-blocking assignment lists

scheduled changes at time 2

e = 0

scheduled changes at time 4

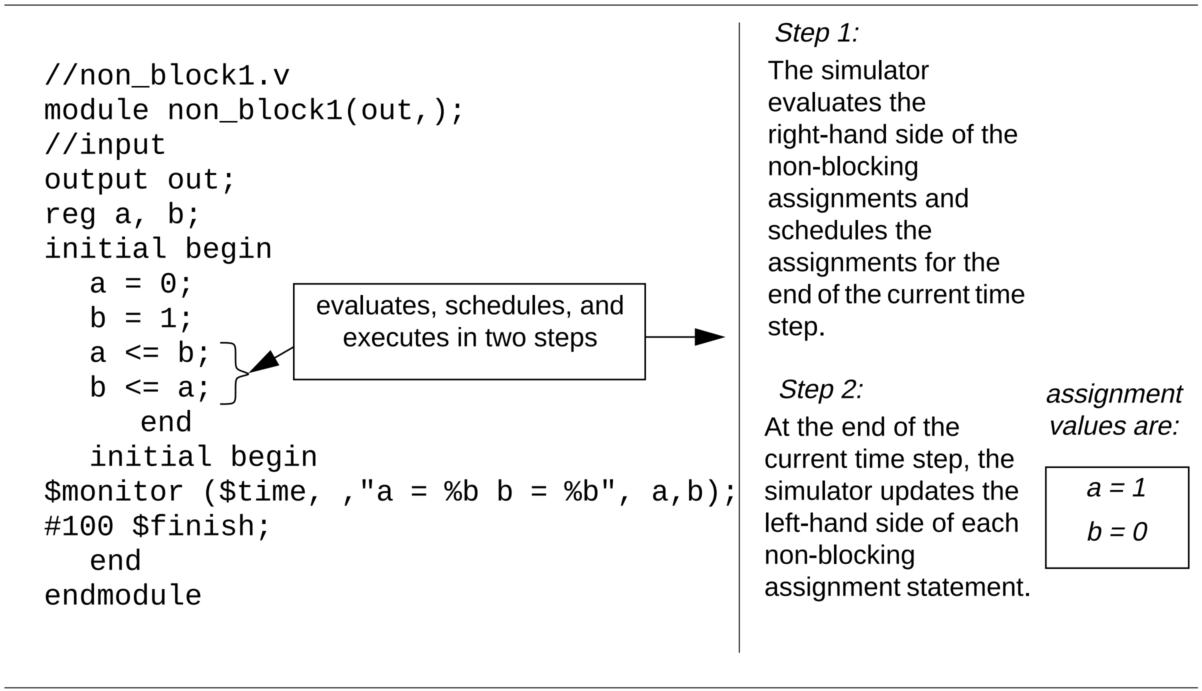
f = 1

scheduled changes at time 10

d = 1

Example 8-6: Non-blocking assignments do not block execution of sequential statements

Note: As shown in Example 8-7, the simulator evaluates and schedules assignments for the end of the current time step and can perform swapping operations with non-blocking procedural assignments.



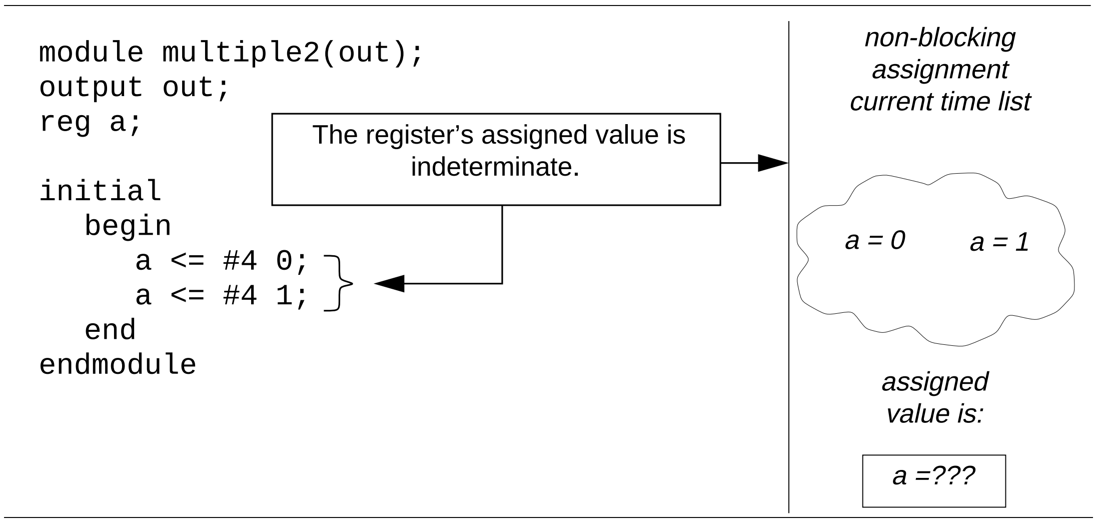
```
//non_block1.v
module non_block1(out,);
//input
output out;
reg a, b;
initial begin
   a = 0;
   b = 1;
   a <= b;
   b <= a;
      end
   initial begin
$monitor ($time, ,"a = %b b = %b", a,b);
#100 $finish;
   end
endmodule
```

Step 1:
The simulator evaluates the right-hand side of the non-blocking assignments and schedules the assignments for the end of the current time step.

Step 2:
At the end of the current time step, the simulator updates the left-hand side of each non-blocking assignment statement.

Example 8-7: Non-blocking procedural assignments used for swapping operations

When you schedule multiple non-blocking assignments to occur in the same register in a particular time slot, the simulator cannot guarantee the order in which it processes the assignments—the final value of the register is indeterminate. As shown in Example 8-8, the value of register a is not known until the end of time step 4.



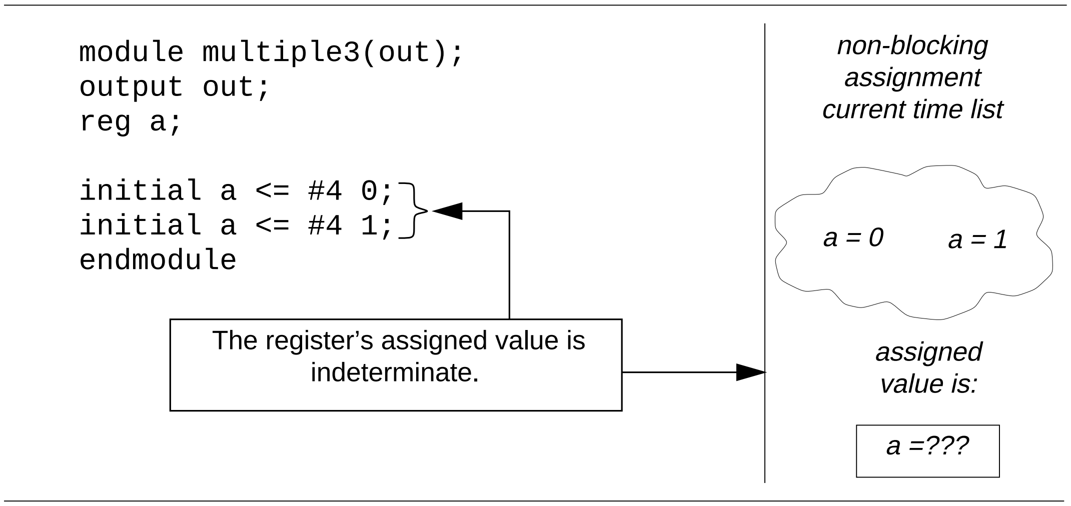
```
module multiple2(out);
output out;
reg a;

initial
   begin
      a <= #4 0;
      a <= #4 1;
   end
endmodule
```

The register's assigned value is indeterminate.
non-blocking assignment current time list
a = 0
a = 1
assigned value is:
a =???

Example 8-8: Multiple non-blocking assignments made in a single time step

If the simulator executes two procedural blocks concurrently, and these procedural blocks contain non-blocking assignment operators, the final value of the register is indeterminate. For example, in Example 8-9 the value of register `a` is indeterminate.



```
module multiple3(out);
output out;
reg a;

initial a <= #4 0;
initial a <= #4 1;
endmodule
```

non-blocking assignment current time list
a = 0
a = 1
The register's assigned value is indeterminate.
assigned value is:
a =???

Example 8-9: Processing two procedural assignments concurrently

When multiple non-blocking assignments with *timing controls* are made to the same register, the assignments can be made without cancelling previous non-blocking assignments. In Example 8-10, the simulator evaluates the value of `i[0]` to `r1` and schedules the assignments to occur after each time delay.

```
module multiple;
reg r1;
reg [2:0] i;

initial
   begin
// starts at time 0 doesn't hold the block
      for (i = 0; i <= 5; i = i+1)
      r1 <= # (i*10) i[0];
   end
endmodule
```

Make the assignments to `r1` without cancelling previous non-blocking assignments.

scheduled changes at time0

r1 = 0

scheduled changes at time10

r1 = 1

scheduled changes at time 20

r1 = 0

scheduled changes at time 30

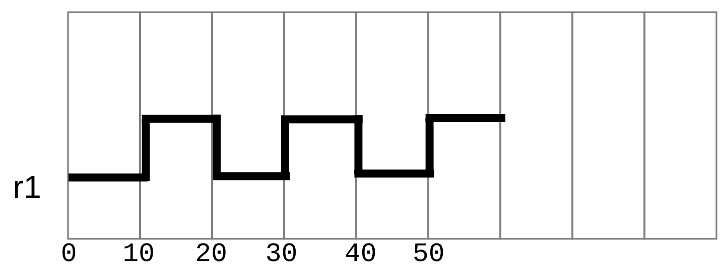
r1 = 1

scheduled changes at time 40

r1 = 0

scheduled changes at time 50

r1 = 1



Example 8-10: Multiple non-blocking assignments with timing controls

8.3.3 How the Simulator Processes Blocking and Non-Blocking Procedural Assignments

For each time slot during simulation, blocking and non-blocking procedural assignments are processed in the following way:

1. Evaluate the right-hand side of all assignment statements in the current time slot.
2. Execute all blocking procedural assignments and non-blocking procedural assignments that have no timing controls. At the same time, non-blocking procedural assignments with timing controls are set aside for processing.
3. Check for procedures that have timing controls and execute if timing control is set for the current time unit.
4. Advance the simulation clock.

8.4 Conditional Statement

The conditional statement (or `if-else` statement) is used to make a decision as to whether a statement is executed or not. Formally, the syntax is as follows:

```
<statement>
        ::= if ( <expression> ) <statement_or_null>
        ||= if ( <expression> ) <statement_or_null>
                        else <statement_or_null>
<statement_or_null>
          ::= <statement>
          ||= ;
```

Syntax 8-3: Syntax of `if` statement

The <expression> is evaluated; if it is true (that is, has a non-zero known value), the first statement executes. If it is false (has a zero value or the value is `x` or `z`), the first statement does not execute. If there is an `else` statement and <expression> is false, the `else` statement executes.

Since the numeric value of the `if` expression is tested for being zero, certain shortcuts are possible. For example, the following two statements express the same logic:

```
if (expression)
if (expression != 0)
```

Because the `else` part of an `if-else` is optional, there can be confusion when an `else` is omitted from a nested `if` sequence. This is resolved by always associating the `else` with the closest previous `if` that lacks an `else`. In Example 8-11, the `else` goes with the inner `if`, as we have shown by indentation.

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

Example 8-11: Association of `else` in nested `if`

If that association is not what you want, use a begin-end block statement to force the proper association, as shown in Example 8-12.

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

Example 8-12: Forcing correct association of `else` with `if`

Begin-end blocks left out inadvertently can change the logic behavior being expressed, as shown in Example 8-13.

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

Example 8-13: Erroneous association of `else` with `if`

The indentation in Example 8-13 shows unequivocally what you want, but the compiler does not get the message and associates the `else` with the inner `if`. This kind of bug can be very hard to find. (One way to find this kind of bug is to use the `$list` system task, which indents according to the logic of the description).

Notice that in Example 8-14, there is a semicolon after `result = rega`. This is because a `<statement>` follows the `if`, and a semicolon is an essential part of the syntax of a `<statement>`.

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

Example 8-14: Use of semicolon in `if` statement

For Verilog-XL to behave predictably in interactive mode, each conditional statement must conform to one or both of the following rules:

- The conditional statement must be in a sequential (`begin-end`) procedural block or a parallel (`fork-join`) procedural block.
- The conditional statement must include an `else` statement.

8.5 Multi-Way Decision Statements

There are two statements that you can use to specify one or more actions to be taken based on specified conditions: `if-else-if` and `case`.

8.5.1 `if-else-if` Statement

The sequence of `if` statements known as an `if-else-if` construct is the most general way of writing a multi-way decision. The syntax is shown in in Syntax 8-4.

```
if (<expression>)
    <statement>
else if (<expression>)
    <statement>
else if (<expression>)
    <statement>
else
    <statement>
```

Syntax 8-4: Syntax of `if-else-if` construct

The expressions are evaluated in order; if any expression is true, the statement associated with it is executed, and this terminates the whole chain. Each statement is either a single statement or a block of statements.

The last `else` part of the `if-else-if` construct handles the default case where none of the other conditions was satisfied. Sometimes there is no explicit action for the default; in that case the trailing `else` can be omitted or it can be used for error checking to catch an impossible condition.

8.5.2 Example

The module fragment in Example 8-15 uses the `if-else` statement to test the variable `index` to decide whether one of three `modify_segn` registers must be added to the memory address, and which increment is to be added to the `index` register. The first ten lines declare the registers and parameters.

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

Example 8-15: Use of `if-else-if` construct

8.5.3 case Statement

The `case` statement is a special multi-way decision statement that tests whether an expression matches one of several other expressions, and branches accordingly. For example, the `case` statement is useful for describing the decoding of a microprocessor instruction. The syntax of the `case` statement is shown in Syntax 8-5.

```
<statement>
        ::= case ( <expression> ) <case_item>+ endcase
        ||= casez ( <expression> ) <case_item>+ endcase
        ||= casex ( <expression> ) <case_item>+ endcase
<case_item>
        ::= <expression> <,<expression>>* : <statement_or_null>
        ||= default : <statement_or_null>
        ||= default <statement_or_null>
```

Syntax 8-5: Syntax for `case` statement

The default statement is optional. Use of multiple default statements in one `case` statement is illegal syntax.

A simple example of the `case` statement is the decoding of register `rega` to produce a value for `result` as shown in Example 8-16.

```
reg [15:0] rega;
reg [9:0] result;
          .
          .
          .
case (rega)
    16'd0: result = 10'b0111111111;
    16'd1: result = 10'b1011111111;
    16'd2: result = 10'b1101111111;
    16'd3: result = 10'b1110111111;
    16'd4: result = 10'b1111011111;
    16'd5: result = 10'b1111101111;
    16'd6: result = 10'b1111110111;
    16'd7: result = 10'b1111111011;
    16'd8: result = 10'b1111111101;
    16'd9: result = 10'b1111111110;
    default result = 'bx;
endcase
```

Example 8-16: Use of the `case` statement

The case expressions are evaluated and compared in the exact order in which they are given. During the linear search, if one of the case item expressions matches the expression in parentheses, then the statement associated with that case item is executed. If all comparisons fail, and the default item is given, then the default item statement is executed. If the default statement is not given, and all of the comparisons fail, then none of the `case` item statements is executed.

Apart from syntax, the `case` statement differs from the multi-way `if-else-if` construct in two important ways:

1. The conditional expressions in the `if-else-if` construct are more general than comparing one expression with several others, as in the `case` statement.
2. The `case` statement provides a definitive result when there are `x` and `z` values in an expression.

In a case comparison, the comparison only succeeds when each bit matches exactly with respect to the values `0`, `1`, `x`, and `z`. As a consequence, care is needed in specifying the expressions in the `case` statement. The bit length of all the expressions must be equal so that exact bit-wise matching can be performed. The length of all the case item expressions, as well as the controlling expression in the parentheses, will be made equal to the length of the longest <case_item> expression. The most common mistake made here is to specify ´bx or ´bz instead of n'bx or n'bz, where n is the bit length of the expression in parentheses. The default length of x and z is the word size of the host machine, usually 32 bits.

The reason for providing a case comparison that handles the x and z values is that it provides a mechanism for detecting such values and reducing the pessimism that can be generated by their presence. Example 8-17 illustrates the use of a case statement to properly handle x and z values.

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

Example 8-17: Detecting x and z values with the case statement

Example 8-17 contains a robust case statement used to trap x and z values. Notice that if select[1] is 0 and flaga is 0, then no matter what the value of select[2] is, the result is set to 0. The first, second, and third case items cause this assignment.

Example 8-18 shows another way to use a case statement to detect x and z values.

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

Example 8-18: Another example of detecting `x` and `z` with `case`

8.5.4 case Statement with Don't-Cares

Two other types of `case` statements are provided to allow handling of don't-care conditions in the case comparisons. One of these treats high-impedance values (`z`) as don't-cares, and the other treats both high-impedance and unknown (`x`) values as don't-cares.

These `case` statements are used in the same way as the traditional `case` statement, but they begin with new keywords—`casez` and `casex`, respectively.

Don't-care values (`z` values for `casez`, `z` and `x` values for `casex`), in any bit of either the case expression or the case items, are treated as don't-care conditions during the comparison, and that bit position is not considered.

Note that allowing don't-cares in the case items means that you can dynamically control which bits of the case expression are compared during simulation.

The syntax of literal numbers allows the use of the question mark `(?)` in place of `z` in these `case` statements. This provides a convenient format for specification of don't-care bits in `case` statements.

Example 8-19 is an example of the `casez` statement. It demonstrates an instruction decode, where values of the most significant bits select which task should be called. If the most significant bit of `ir` is a `1`, then the task `instruction1` is called, regardless of the values of the other bits of `ir`.

```
reg [7:0] ir;
          •
          •
          •

casez (ir)
   8'b1???????: instruction1(ir);
   8'b01??????: instruction2(ir);
   8'b00010???: instruction3(ir);
   8'b000001??: instruction4(ir);
endcase
```

Example 8-19: Using the `casez` statement

Example 8-20 is an example of the `casex` statement. It demonstrates an extreme case of how don't-care conditions can be dynamically controlled during simulation. In this case, if `r = 8´b01100110`, then the task `stat2` is called.

```
reg [7:0] r, mask;
          •
          •
          •

mask = 8'bx0x0x0x0;
casex (r ^ mask)
   8'b001100xx: stat1;
   8'b1100xx00: stat2;
   8'b00xx0011: stat3;
   8'bxx001100: stat4;
endcase
```

Example 8-20: Using the `casex` statement

8.6 Looping Statements

There are four types of looping statements. They provide a means of controlling the execution of a statement zero, one, or more times.

- `forever` continuously executes a statement.
- `repeat` executes a statement a fixed number of times.
- `while` executes a statement until an expression becomes false. If the expression starts out false, the statement is not executed at all.
- `for` controls execution of its associated statement(s) by a three-step process, as follows:

    1. executes an assignment normally used to initialize a variable that controls the number of loops executed
    2. evaluates an expression—if the result is zero, the `for` loop exits, and if it is not zero, the `for` loop executes its associated statement(s) and then performs step 3
    3. executes an assignment normally used to modify the value of the loop-control variable, then repeats step 2

The following are the syntax rules for the looping statements:

```
<statement>
        ::= forever <statement>
        ||=forever
                begin
                    <statement>+
                end
<statement>
        ::= repeat ( <expression> ) <statement>
        ||=repeat ( <expression> )
                begin
                    <statement>+
                end
<statement>
        ::= while ( <expression> ) <statement>
        ||=while ( <expression> )
                begin
                    <statement>+
                end
<statement>
        ::= for ( <assignment> ; <expression> ; <assignment> )
                      <statement>
        ||=for ( <assignment> ; <expression> ; <assignment> )
                begin
                    <statement>+
                end
```

Syntax 8-6: Syntax for the looping statements

The rest of this section presents examples for three of the looping statements.

8.6.1 `forever` Loop

The `forever` loop should only be used in conjunction with the timing controls or the disable statement. An example is presented in Section 8.7.3 on page 8-32.

8.6.2 repeat Loop Example

In the following example of a `repeat` loop, add and shift operators implement a multiplier.

```
parameter size = 8, longsize = 16;
reg [size:1] opa, opb;
reg [longsize:1] result;
begin :mult
    reg [longsize:1] shift_opa, shift_opb;

    shift_opa = opa;
    shift_opb = opb;
    result = 0;

    repeat (size)
        begin
           if (shift_opb[1])
              result = result + shift_opa;
           shift_opa = shift_opa << 1;
           shift_opb = shift_opb >> 1;
        end
end
```

Example 8-21: Use of the `repeat` loop to implement a multiplier

8.6.3 while Loop Example

An example of the `while` loop follows. It counts up the number of logic `1` values in `rega`.

```
begin :count1s
    reg [7:0] tempreg;
    count = 0;
    tempreg = rega;
    while(tempreg)
        begin
            if (tempreg[0]) count = count + 1;
            tempreg = tempreg >> 1;
        end
end
```

Example 8-22: Use of the `while` loop to count logic values

8.6.4 `for` Loop Examples

The `for` loop construct accomplishes the same results as the following pseudocode that is based on the `while` loop:

```
begin
   initial_assignment;
   while (condition)
   begin
      statement
      step_assignment;
   end
end
```

Example 8-23: Pseudocode equivalent of a `for` loop

The `for` loop implements the logic in the preceding 8 lines while using only two lines, as shown in the pseudocode in Example 8-24.

```
for (initial_assignment; condition; step_assignment)
       statement
```

Example 8-24: Pseudocode for a `for` loop

Example 8-25 uses a `for` loop to initialize a memory.

```
begin :init_mem
   reg [7:0] tempi;
   for (tempi = 0; tempi < memsize; tempi = tempi + 1)
         memory[tempi] = 0;
end
```

Example 8-25: Use of the `for` loop to initialize a memory

Here is another example of a `for` loop statement. It is the same multiplier that was described in Example 8-21 using the `repeat` loop.

```
parameter size = 8, longsize = 16;
reg [size:1] opa, opb;
reg [longsize:1] result;

begin :mult
    integer bindex;
    result = 0;
    for (bindex = 1; bindex <= size; bindex = bindex + 1)
        if (opb[bindex])
          result = result + (opa << (bindex - 1));
end
```

Example 8-26: Use of the `for` loop to implement a multiplier

Note that the `for` loop statement can be more general than the normal arithmetic progression of an index variable, as in Example 8-27. This is another way of counting the number of logic `1` values in `rega` (see Example 8-22).

```
begin :count1s
    reg [7:0] tempreg;
    count = 0;
    for (tempreg = rega; tempreg; tempreg = tempreg >
> 1)
        if (tempreg[0]) count = count + 1;
end
```

Example 8-27: Use of the `for` loop to count logic values

8.7 Procedural Timing Controls

The Verilog language provides two types of explicit timing control over when in simulation time procedural statements are to occur. The first type is a delay control in which an expression specifies the time duration between initially encountering the statement and when the statement actually executes. The delay expression can be a dynamic function of the state of the circuit, but is usually a simple number that separates statement executions in time. The delay control is an important feature when specifying stimulus waveform descriptions. It is described in Sections 8.7.1, 8.7.2, and 8.7.7.

The second type of timing control is the event expression, which allows statement execution to wait for the occurrence of some simulation event occurring in a procedure executing concurrently with this procedure. A simulation event can be a change of value on a net or register (an implicit event), or the occurrence of an explicitly named event that is triggered from other procedures (an explicit event). Most often, an event control is a positive or negative edge on a clock signal. Sections 8.7.3 through 8.7.7 discuss event control.

In Verilog, actions are scheduled in the future through the use of delay controls. A general principle of the Verilog language is that “where you do not see a timing control, simulation time does not advance”—if you specify no timing delays, the simulation completes at time zero. To schedule activity for the future, use one of the following methods of timing control:

- a `delay` control, which is introduced by the number symbol (#)
- an `event` control, which is introduced by the at symbol (@)
- the `wait` statement, which operates like a combination of the event control and the `while` loop

The next sections discuss these three methods.

8.7.1 Delay Control

The execution of a procedural statement can be delay-controlled by using the following syntax:

<statement>
    ::= <delay_control> <statement_or_null>
<delay_control>
    ::= # <NUMBER>
    ||= # <identifier>
    ||= # (<mintypmax_expression>)

Syntax 8-7: Syntax for delay_control

The following example delays the execution of the assignment by 10 time units:

```
#10 rega = regb;
```

The next three examples provide an expression following the number sign (#). Execution of the assignment delays by the amount of simulation time specified by the value of the expression.

```
#d rega = regb;      // d is defined as a parameter
#((d+e)/2) rega = regb;// delay is the average of d and e
#regr regr = regr + 1;// delay is the value in regr
```

8.7.2 Zero-Delay Control

A special case of the delay control is the zero-delay control, as in the following example:

```
forever
   #0 a = ~a;
```

This type of delay control has the effect of moving the assignment statement to the end of the list of statements to be evaluated at the current simulation time unit. Note that if there are several such delay controls encountered at the same simulation time, the order of evaluation of the statements which they control cannot be predicted.

8.7.3 Event Control

The execution of a procedural statement can be synchronized with a value change on a net or register, or the occurrence of a declared event, by using the following event control syntax:

```
<statement>
        ::= <event_control> <statement_or_null>
<event_control>
        ::= @ <identifier>
        ||= @ ( <event_expression> )
<event_expression>
        ::= <expression>
        ||= posedge <SCALAR_EVENT_EXPRESSION>
        ||= negedge <SCALAR_EVENT_EXPRESSION>
        ||= <event_expression> <or <event_expression>>*
<SCALAR_EVENT_EXPRESSION> is an expression that resolves
                    to a one bit value.
```

Syntax 8-8: Syntax for `event_control`

Value changes on nets and registers can be used as events to trigger the execution of a statement. This is known as detecting an implicit event. See item 1 in Example 8-28 for a syntax example of a wait for an implicit event. Verilog syntax also allows you to detect change based on the direction of the change—that is, toward the value 1 (`posedge`) or toward the value 0 (`negedge`). The behavior of `posedge` and `negedge` for unknown expression values is as follows:

- a `negedge` is detected on the transition from 1 to unknown and from unknown to 0
- a `posedge` is detected on the transition from 0 to unknown and from unknown to 1

Items 2 and 3 in Example 8-28 show illustrations of edge-controlled statements.

```
❶ @r rega = regb;  // controlled by any value changes
                   // in the register r

❷ @(posedge clock) rega = regb; // controlled by positive
                                // edge on clock

❸ forever @(negedge clock) rega = regb;  // controlled by
                                         // negative edge
```

Example 8-28: Event controlled statements

8.7.4 Named Events

Verilog also provides syntax to name an event and then to trigger the occurrence of that event. A model can then use an event expression to wait for the triggering of this explicit event. Named events can be made to occur from a procedure. This allows control over the enabling of multiple actions in other procedures. Named events and event control give a powerful and efficient means of describing the communication between, and synchronization of, two or more concurrently active processes. A basic example of this is a small waveform clock generator that synchronizes control of a synchronous circuit by signalling the occurrence of an explicit event periodically while the circuit waits for the event to occur.

An event name must be declared explicitly before it is used. The following is the syntax for declaring events.

```
<event_declaration>
        ::= event <name_of_event> <,<name_of_event>>* ;
<name_of_event>
        ::= <IDENTIFIER> - the name of an explicit event
```

Syntax 8-9: Syntax for event_declaration

Note that an event does not hold any data. The following are the characteristics of a Verilog event:

- it can be made to occur at any particular time
- it has no time duration
- its occurrence can be recognized by using the <event_control> syntax described in Section 8.7.3

The power of the explicit event is that it can represent any general happening. For example, it can represent a positive edge of a clock signal, or it can represent a microprocessor transferring data down a serial communications channel. A declared event is made to occur by the activation of an event-triggering statement of the following syntax:

```
-> <name_of_event> ;
```

An event-controlled statement (for example, `@trig rega = regb;`) causes simulation of its containing procedure to wait until some other procedure executes the appropriate event-triggering statement (for example, `->trig;`).

8.7.5 Event OR Construct

The ORing of any number of events can be expressed such that the occurrence of any one will trigger the execution of the statement. The next two examples show the ORing of two and three events respectively.

```
@(trig or enable) rega = regb;// controlled by trig or enable
   @(posedge clock_a or posedge clock_b or trig) rega = regb;
```

8.7.6 Level-Sensitive Event Control

The execution of a statement can also be delayed until a condition becomes true. This is accomplished using the `wait` statement, which is a special form of event control. The nature of the `wait` statement is level-sensitive, as opposed to basic event control (specified by the `@` character), which is edge-sensitive. The `wait` statement checks a condition, and, if it is false, causes the procedure to pause until that condition becomes true before continuing. The `wait` statement has the following form:

```
wait(condition_expression) statement
```

Example 8-29 shows the use of the `wait` statement to accomplish level-sensitive event control.

```
begin
   wait(!enable) #10 a = b;
   #10 c = d;
end
```

Example 8-29: Use of `wait` statement

If the value of `enable` is one when the block is entered, the `wait` statement delays the evaluation of the next statement (`#10 a = b;`) until the value of `enable` changes to zero. If `enable` is already zero when the begin-end block is entered, then the next statement is evaluated immediately and no delay occurs.

8.7.7 Intra-Assignment Timing Controls

The delay and event control constructs previously described precede a statement and delay its execution. The intra-assignment delay and event controls are contained within an assignment statement and modify the flow of activity in a slightly different way.

Encountering an intra-assignment delay or event control delays the assignment just as a regular delay or event control does, but the right-hand side expression is evaluated before the delay, instead of after the delay. This allows data swap and data shift operations to be described without the need for temporary variables. This section describes the purpose of intra-assignment timing controls and the `repeat` timing control that can be used in intra-assignment delays.

Figure 8-1 illustrates the philosophy of intra-assignment timing controls by showing the code that could accomplish the same timing effect without using intra-assignment.

Intra-assignment timing control	
with intra-assignment construct	without intra-assignment construct

`a = #5 b;`	`begin`
	   `temp = b;`
	   `#5 a = temp;`
	`end`

`a = @(posedge clk) b;`	`begin`
	   `temp = b;`
	   `@(posedge clk) a = temp;`
	`end`

`a = repeat(3)@(posedge clk) b;`	`begin`
	   `temp = b;`
	   `@(posedge clk;`
	   `@(posedge clk;`
	   `@(posedge clk) a = temp;`
	`end`

Figure 8-1: Equivalents to intra-assignment timing controls

The next three examples use the `fork-join` behavioral construct. All statements between the keywords `fork` and `join` execute concurrently. Section 8.8.2 on page 8-41 describes this construct in more detail.

The following example shows a race condition that could be prevented by using intra-assignment timing control:

```
fork
   #5 a = b;
   #5 b = a;
join
```

The code in the previous example samples the values of both `a` and `b` at the same simulation time, thereby creating a race condition. The intra-assignment form of timing control used in the following example prevents this race condition:

```
fork                    // data swap
   a = #5 b;
   b = #5 a;
join
```

Intra-assignment timing control works because the intra-assignment delay causes the values of `a` and `b` to be evaluated *before* the delay, and the assignments to be made *after* the delay. Verilog-XL and other tools that implement intra-assignment timing control use temporary storage in evaluating each expression on the right-hand side.

Intra-assignment waiting for *events* is also effective. In the example below, the right-hand-side expressions are evaluated when the assignment statements are encountered, but the assignments are delayed until the rising edge of the clock signal.

```
fork                 // data shift
   a = @(posedge clk) b;
   b = @(posedge clk) c;
join
```

The `repeat` event control

The `repeat` event control specifies an intra-assignment delay of a specified number of occurrences of an event. This construct is convenient when events must be synchronized with counts of clock signals.

Syntax 8-10 presents the `repeat` event control syntax:

```
<repeat_event _controlled_assignment>
        ::=<lvalue> = <repeat_event_control><expression>;
        ||=<lvalue> <= <repeat_event_control><expression>;

<repeat_event_control>
        ::=repeat(<expression>)@(<identifier>)
        ||=repeat(<expression>)@(<event_expression>)

<event_expression>
        ::=<expression>
        ||=posedge<SCALAR_EVENT_EXPRESSION>
        ||=negedge<SCALAR_EVENT_EXPRESSION>
        ||=<event_expression>or<event_expression>
```

Syntax 8-10: Syntax of the `repeat` event control

The event expression must resolve to a one bit value. A scalar event expression is an expression which resolves to a one bit value.

The following is an example of a `repeat` event control as the intra-assignment delay of a non-blocking assignment:

```
a<=repeat(5)@(posedge clk)data;
```

Figure 8-2 illustrates the activities that result from this `repeat` event control:

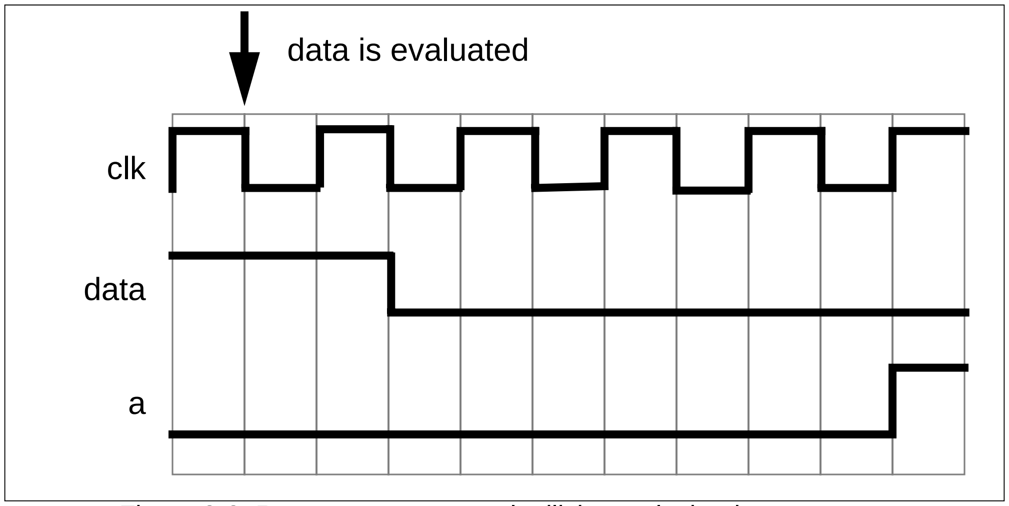


Figure 8-2: Repeat `event` control utilizing a clock edge

In this example, the value of `data` is evaluated when the assignment is encountered. After five occurrences of `posedge clk`, `a` is assigned the previously evaluated value of `data`.

The following is an example of a `repeat` event control as the intra-assignment delay of a procedural assignment:

```
a = repeat(num)@(clk)data;
```

In this example, the value of `data` is evaluated when the assignment is encountered. After the number of transitions of `clk` equals the value of `num`, `a` is assigned the previously evaluated value of `data`.

The following is an example of a `repeat` event control with expressions containing operations to specify both the number of event occurrences and the event that is counted:

```
a <= repeat(a+b)@(posedge phi1 or negedge phi2)data;
```

In the example above, the value of `data` is evaluated when the assignment is encountered. After the positive edges of `phi1`, the negative edges of `phi2`, or the combination of these two events occurs a total of `(a+b)` times, `a` is assigned the previously evaluated value of `data`.

8.8 Block Statements

The block statements are a means of grouping two or more statements together so that they act syntactically like a single statement. We have already introduced and used the sequential block statement which is delimited by the keywords `begin` and `end`. Section 8.8.1 discusses sequential blocks in more detail.

A second type of block, delimited by the keywords `fork` and `join`, is used for executing statements in parallel. A `fork-join` block is known as a parallel block, and enables procedures to execute concurrently through time. Section 8.8.2 discusses parallel blocks.

8.8.1 Sequential Blocks

A sequential block has the following characteristics:

- Statements execute in sequence, one after another.
- The delays are cumulative; each statement executes after all the delays preceding it have elapsed.
- Control passes out of the block after the last statement executes.

The following is the formal syntax for a sequential block:

```
<seq_block>
        ::= begin <statement>* end
        ||= begin : <name_of_block>
                        <block_declaration>*
                        <statement>*
        end
<name_of_block>
        ::= <IDENTIFIER>
<block_declaration>
        ::= <parameter_declaration>
        ||= <reg_declaration>
        ||= <integer_declaration>
        ||= <real_declaration>
        ||= <time_declaration>
        ||= <event_declaration>
```

Syntax 8-11: Syntax for the sequential block

A sequential block enables the following two assignments to have a deterministic result:

```
begin
   areg = breg;
   creg = areg; // creg becomes the value of breg
end
```

Here the first assignment is performed and `areg` is updated before control passes to the second assignment.

Delay control can be used in a sequential block to separate the two assignments in time.

```
begin
   areg = breg;
   #10 creg = areg; // this gives a delay of 10 time
end                 // units between assignments
```

Example 8-30 shows how the combination of the sequential block and delay control can be used to specify a time-sequenced waveform.

```
parameter d = 50; // d declared as a parameter
reg [7:0] r;      // and r declared as an 8-bit register
begin     // a waveform controlled by sequential
          // delay
   #d r = 'h35;
   #d r = 'hE2;
   #d r = 'h00;
   #d r = 'hF7;
   #d -> end_wave;// trigger the event called
end_wave
```

Example 8-30: A waveform controlled by sequential delay

Example 8-31 shows three examples of sequential blocks.

```
❶  begin
      @trig r = 1;
      #250 r = 0; // a 250 delay monostable
   end
❷  begin
      @(posedge clock) q = 0;
      @(posedge clock) q = 1;
   end
❸  begin // a waveform synchronized by the event c
      @c r = 'h35;
      @c r = 'hE2;
      @c r = 'h00;
      @c r = 'hF7;
      @c -> end_wave;
   end
```

Example 8-31: Three examples of sequential blocks

8.8.2 Parallel Blocks

A parallel block has the following characteristics:

- statements execute concurrently
- delay values for each statement are relative to the simulation time when control enters the block
- delay control is used to provide time-ordering for assignments
- control passes out of the block when the last time-ordered statement executes or a `disable` statement executes

Syntax 8-12 gives the formal syntax for a parallel block.

```
<par_block>
        ::= fork <statement>* join
        ||= fork : <name_of_block>
                  <block_declaration>*
                  <statement>*
             join
<name_of_block>
          ::= <IDENTIFIER>
<block_declaration>
          ::= <parameter_declaration>
          ||= <reg_declaration>
          ||= <integer_declaration>
          ||= <real_declaration>
          ||= <time_declaration>
          ||= <event_declaration>
```

Syntax 8-12: Syntax for the parallel block

Example 8-32 codes the waveform description shown in Example 8-30 by using a parallel block instead of a sequential block. The waveform produced on the register is exactly the same for both implementations.

```
fork
   #50 r = 'h35;
   #100 r = 'hE2;
   #150 r = 'h00;
   #200 r = 'hF7;
   #250 -> end_wave;
join
```

Example 8-32: Use of the `fork-join` construct

8.8.3 Block Names

Note that blocks can be named by adding: `name_of_block` after the keywords `begin` or `fork`. The naming of blocks serves several purposes:

- It allows local variables to be declared for the block.
- It allows the block to be referenced in statements like the `disable` statement (as discussed in Chapter 10, *Disabling of Named Blocks and Tasks*).
- In the Verilog language, all variables are static—that is, a unique location exists for all variables and leaving or entering blocks does not affect the values stored in them.

Thus, block names give a means of uniquely identifying all variables at any simulation time. This is very important for debugging purposes, where it is necessary to be able to reference a local variable inside a block from outside the body of the block.

8.8.4 Start and Finish Times

Both forms of blocks have the notion of a start and finish time. For sequential blocks, the start time is when the first statement is executed, and the finish time is when the last statement has finished. For parallel blocks, the start time is the same for all the statements, and the finish time is when the last time-ordered statement has finished executing. When blocks are embedded within each other, the timing of when a block starts and finishes is important. Execution does not continue to the statement following a block until the block's finish time has been reached—that is, until the block has completely finished executing.

Moreover, the timing controls in a `fork-join` block do not have to be given sequentially in time. Example 8-33 shows the statements from Example 8-32 written in the reverse order and still producing the same waveform.

```
fork
  #250 -> end_wave;
  #200 r = 'hF7;
  #150 r = 'h00;
  #100 r = 'hE2;
  #50 r = 'h35;
join
```

Example 8-33: Timing controls in a parallel block

Sequential and parallel blocks can be embedded within each other allowing complex control structures to be expressed easily and with a high degree of structure.

One simple example of this is when an assignment is to be made after two separate events have occurred. This is known as the “joining” of events.

```
begin
    fork
        @Aevent;
        @Bevent;
    join
    areg = breg;
end
```

Example 8-34: The joining of events

Note that the two events can occur in any order (or even at the same time), the `fork-join` block will complete, and the assignment will be made. In contrast to this, if the `fork-join` block was a `begin-end` block and the `Bevent` occurred before the `Aevent`, then the block would be deadlocked waiting for the `Bevent`.

Example 8-35 shows two sequential blocks, each of which will execute when its controlling event occurs. Because the `wait` statements are within a `fork-join` block, they execute in parallel and the sequential blocks can therefore also execute in parallel.

```
fork
    @enable_a
        begin
            #ta wa = 0;
            #ta wa = 1;
            #ta wa = 0;
        end
    @enable_b
        begin
            #tb wb = 1;
            #tb wb = 0;
            #tb wb = 1;
        end
join
```

Example 8-35: Enabling sequential blocks to execute in parallel

8.9 Examples

This section contains two behavioral model examples. These examples are given as complete descriptions enclosed in modules—such that they can be put directly through the Verilog-XL compiler, simulated and the results observed.

Example 8-36 shows a simple traffic light sequencer described with its own clock generator.

```
module traffic_lights;
    reg
        clock,
        red,
        amber,
        green;
    parameter
        on = 1,
        off = 0,
        red_tics = 350,
        amber_tics = 30,
        green_tics = 200;
    // the sequence to control the lights
    always
        begin
          red = on;
          amber = off;
          green = off;
          repeat (red_tics) @(posedge clock);
          red = off;
          green = on;
          repeat (green_tics) @(posedge clock);
          green = off;
          amber = on;
          repeat (amber_tics) @(posedge clock);
        end
    // waveform for the clock
    always
        begin
            #100 clock = 0;
            #100 clock = 1;
        end
    // simulate for 10 changes on the red light
    initial
        begin
            repeat (10) @red;
            $finish;
        end
    // display the time and changes made to the lights
    always
        @(red or amber or green)
        $display("%d red=%b amber=%b green=%b",
                $time, red, amber, green);
endmodule
```

Example 8-36: Behavioral model of traffic light sequencer

Example 8-37 shows a use of variable delays. The module has a clock input and produces two synchronized clock outputs. Each output clock has equal mark and space times, is out of phase from the other by 45 degrees, and has a period half that of the input clock. Note that the clock generation is independent of the simulation time unit, except as it affects the accuracy of the divide operation on the input clock period.

```
module synch_clocks;
   reg
       clock,
       phase1,
       phase2;
   time clock_time;
   initial clock_time = 0;
   always @(posedge clock)
       begin :phase_gen
           time d; // a local declaration is possible
                   // because the block is named
           d = ($time - clock_time) / 8;
           clock_time = $time;
           phase1 = 0;
           #d phase2 = 1;
           #d phase1 = 1;
           #d phase2 = 0;
           #d phase1 = 0;
           #d phase2 = 1;
           #d phase1 = 1;
           #d phase2 = 0;
       end
   // set up a clock waveform, finish time,
   // and display
   always
       begin
           #100 clock = 0;
           #100 clock = 1;
       end
   initial #1000 $finish; //end simulation at time 1000
   always
       @(phase1 or phase2)
           $display($time,,
               "clock=%b phase1=%b phase2=%b",
               clock, phase1, phase2);
endmodule
```

Example 8-37: Behavioral model with variable delays