5

Functional Descriptions

A Verilog functional description defines a circuit in terms of what it does.

This chapter describes the construction and use of functional descriptions, in the following major sections:

- Sequential Constructs
- Function Declarations
- Function Statements
- task Statements
- always Blocks

Sequential Constructs

Although many Verilog constructs appear sequential in nature, they describe combinational circuitry. A simple description that appears to be sequential is shown in Example 5-1.

Example 5-1 Sequential Statements

```
x = b;
if (y)
     x = x + a;
```

HDL Compiler determines the combinational equivalent of this description. In fact, it treats the statements in Example 5-1 exactly as it treats the statements in Example 5-2.

Example 5-2 Equivalent Combinational Description

```
if (y)
     x = b + a;
else
     x = b;
```

To describe combinational logic, you write a sequence of statements and operators to generate the outputs you want. For example, suppose the addition operator (+) is not supported and you want to create a combinational ripple carry adder. The easiest way to describe this circuit is as a cascade of full adders, as in Example 5-3. The example has eight full adders, with each adder following the one before. From this description, HDL Compiler generates a fully combinational adder.

Example 5-3 Combinational Ripple Carry Adder

```
function [7:0] adder;
input [7:0] a, b;
     reg c;
     integer i;
     begin
          c = 0;
          for (i = 0; i <= 7; i = i + 1) begin
               adder[i] = a[i] ^ b[i] ^ c;
               c = a[i] & b[i] | a[i] & c | b[i] & c;
          end
     end
endfunction
```

Function Declarations

Using a function declaration is one of three methods for describing combinational logic. The other two methods are to use the `always` block, described in “always Blocks” on page 5-33, and to use the continuous assignment, described in “Continuous Assignment” on page 3-15. You must declare and use Verilog functions within a module. You can call functions from the structural part of a Verilog description by using them in a continuous assignment statement or as a terminal in a module instantiation. You can also call functions from other functions or from `always` blocks.

HDL Compiler supports the following Verilog function declarations:

- Input declarations
- Output from a function
- Register declarations
- Memory declarations

- Parameter declarations
- Integer declarations

Functions begin with the keyword `function` and end with the keyword `endfunction`. The width of the function's return value (if any) and the name of the function follow the `function` keyword, as the following syntax shows.

```
function [range] name_of_function ;
              [func_declaration]
              statement_or_null
endfunction
```

Defining the bit range of the return value is optional. Specify the *range* inside square brackets (`[]`). If you do not define the range, a function returns a 1-bit quantity by default. You set the function's output by assigning it to the function name. A function can contain one or more statements. If you use multiple statements, enclose the statements inside a `begin...end` pair.

A simple `function` declaration is shown in Example 5-4.

Example 5-4 Simple Function Declaration

```
function [7:0] scramble;
input [7:0] a;
input [2:0] control;
integer i;
     begin
          for (i = 0; i <= 7; i = i + 1)
               scramble[i] = a[ i ^ control ];
     end
endfunction
```

The function statements HDL Compiler supports are discussed in "Function Statements" on page 5-9.

Input Declarations

The input declarations specify the input signals for a function. You must declare the inputs to a Verilog function immediately after you declare the function name. The syntax of input declarations for a function is the same as the syntax of input declarations for a module:

```
input [range] list_of_variables ;
```

The optional range specification declares an input as a vector of signals. Specify *range* inside square brackets ([]).

Note:

The order in which you declare the inputs must match the order of the inputs in the function call.

Output From a Function

The output from a function is assigned to the function name. A Verilog function has only one output, which can be a vector. For multiple outputs from a function, use the concatenation operation to bundle several values into one return value. This single return value can then be unbundled by the caller. Example 5-5 shows how unbundling is done.

Example 5-5 Many Outputs From a Function

```
function [9:0] signed_add;
input [7:0] a, b;
    reg [7:0] sum;
    reg carry, overflow;

    begin
        ...
        signed_add = {carry, overflow, sum};
    end
endfunction
...
assign {C, V, result_bus} = signed_add(busA, busB);
```

The `signed_add` function bundles the values of `carry`, `overflow`, and `sum` into one value. This new value is returned in the `assign` statement following the function. The original values are then unbundled by the function that called the `signed_add` function.

Register Declarations

A register represents a variable in Verilog. The syntax for a register declaration is

`reg [`*`range`*`]` *`list_of_register_variables`* `;`

A `reg` can be a single-bit quantity or a vector of bits. The *range* specifies the most significant bit (`msb`) and the least significant bit (`lsb`) of the vector enclosed in square brackets ([ ]). Both bits must be nonnegative constants, parameters, or constant-valued expressions. Example 5-6 shows some `reg` declarations.

Example 5-6 Register Declarations

```
reg x;              //single bit
reg a, b, c;        //3 single-bit quantities
reg [7:0] q;        //an 8-bit vector
```

The Verilog language allows you to assign a value to a `reg` variable only within a function or an `always` block.

In the Verilog simulator, `reg` variables can hold state information. A `reg` can hold its value across separate calls to a function. In some cases, HDL Compiler emulates this behavior by inserting flow-through latches. In other cases, it emulates this behavior without a latch. The concept of holding state is elaborated on in “Inferring Latches” on page 6-10 and in several examples in Appendix A, “Examples.”

Memory Declarations

The memory declaration models a bank of registers or memory. In Verilog, the memory declaration is a two-dimensional array of `reg` variables. Sample memory declarations are shown in Example 5-7.

Example 5-7 Memory Declarations

```
reg [7:0] byte_reg;
reg [7:0] mem_block [255:0];
```

In Example 5-7, byte_reg is an 8-bit register and mem_block is an array of 256 registers, each of which is 8 bits wide. You can index the array of registers to access individual registers, but you cannot access individual bits of a register directly. Instead, you must copy the appropriate register into a temporary one-dimensional register. For example, to access the fourth bit of the eighth register in mem_block, enter

```
byte_reg = mem_block [7];
individual_bit = byte_reg [3];
```

Parameter Declarations

Parameter variables are local or global variables that hold values. The syntax for a parameter declaration is

```
parameter [range] identifier = expression,
     identifier = expression;
```

The range specification is optional.

You can declare parameter variables as being local to a function. However, you cannot use a local variable outside that function. Parameter declarations in a function are identical to parameter declarations in a module. The function in Example 5-8 contains a parameter declaration.

Example 5-8 Parameter Declaration in a Function

```
function gte;
     parameter width = 8;
     input [width-1:0] a,b;
     gte = (a >= b);
endfunction
```

Integer Declarations

Integer variables are local or global variables that hold numeric values. The syntax for an integer declaration is

```
integer identifier_list;
```

You can declare integer variables locally at the function level or globally at the module level. The default size for integers is 32 bits. HDL Compiler determines bit-widths, except in the case of a don't care condition resulting during compile.

Example 5-9 illustrates integer declarations.

Example 5-9 Integer Declarations

```
integer a;        //single 32-bit integer
integer b, c;     //two integers
```

Function Statements

The function statements HDL Compiler supports are

- Procedural assignments
- RTL assignments
- `begin...end` block statements
- `if...else` statements
- `case`, `casex`, and `casez` statements
- `for` loops
- `while` loops

- `forever` loops
- `disable` statements

Procedural Assignments

Procedural assignments are assignment statements used inside a function. They are similar to the continuous assignment statements described in “Continuous Assignment” on page 3-15, except that the left side of a procedural assignment can contain only `reg` variables and integers. Assignment statements set the value of the left side to the current value of the right side. The right side of the assignment can contain any arbitrary expression of the data types described in “Structural Data Types” on page 3-7, including simple constants and variables.

The left side of the procedural assignment statement can contain only the following data types:

- `reg` variables
- Bit-selects of `reg` variables
- Part-selects of `reg` variables (must be constant-valued)
- Integers
- Concatenations of the previous data types

HDL Compiler assigns the low bit on the right side to the low bit on the left side. If the number of bits on the right side is greater than the number on the left side, the high-order bits on the right side are discarded. If the number of bits on the left side is greater than the number on the right side, the right-side bits are zero-extended. HDL Compiler allows multiple procedural assignments.

Example 5-10 shows some examples of procedural assignments.

Example 5-10 Procedural Assignments

```
sum = a + b;
control[5] = (instruction == 8'h2e);
{carry_in, a[7:0]} = 9'h 120;
```

RTL Assignments

HDL Compiler handles variables driven by an RTL (nonblocking) assignment differently than those driven by a procedural (blocking) assignment.

In procedural assignments, a value passed along from variable A to variable B to variable C results in all three variables having the same value in every clock cycle. In the netlist, procedural assignments are indicated when the input net of one flip-flop is connected to the input net of another flip-flop. Both flip-flops input the same value in the same clock cycle.

In RTL assignments, however, values are passed on in the next clock cycle. Assignment from variable A to variable B occurs after one clock cycle, if variable A has been a previous target of an RTL assignment. Assignment from variable B to variable C always takes place after one clock cycle, because B is the target when RTL assigns variable A's value to B. In the netlist, an RTL assignment shows flip-flop B receiving its input from the output net of flip-flop A. It takes one clock cycle for the value held by flip-flop A to propagate to flip-flop B.

A variable can follow only one assignment method and therefore cannot be the target of RTL as well as procedural assignments.

Example 5-11 is a description of a serial register implemented with RTL assignments. Figure 5-1 shows the resulting schematic for Example 5-11.

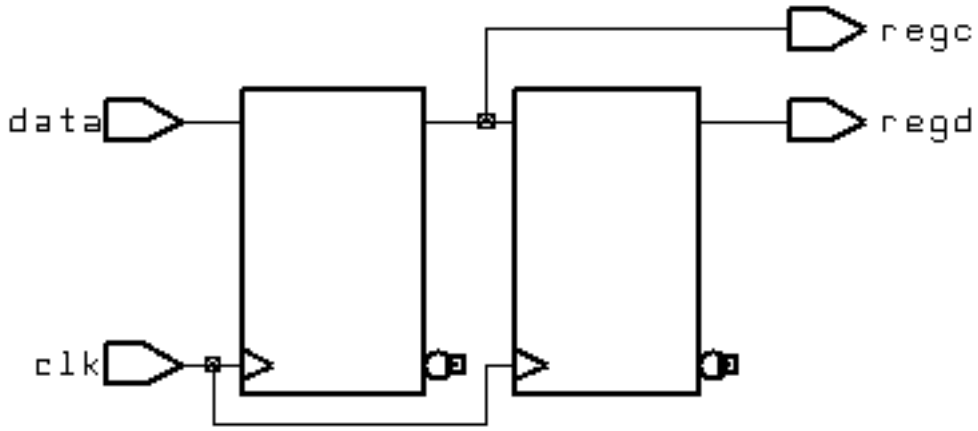
Example 5-11 RTL Nonblocking Assignments

```
module rtl (clk, data, regc, regd);
input data, clk;
output regc, regd;

reg regc, regd;

always @(posedge clk)
begin
     regc <= data;
     regd <= regc;
end
endmodule
```

Figure 5-1 Schematic of RTL Nonblocking Assignments



If you use a procedural assignment, as in Example 5-12, HDL Compiler does not synthesize a serial register. Therefore, the recently assigned value of `rega`, which is data, is assigned to `regb`, as the schematic in Figure 5-2 indicates.

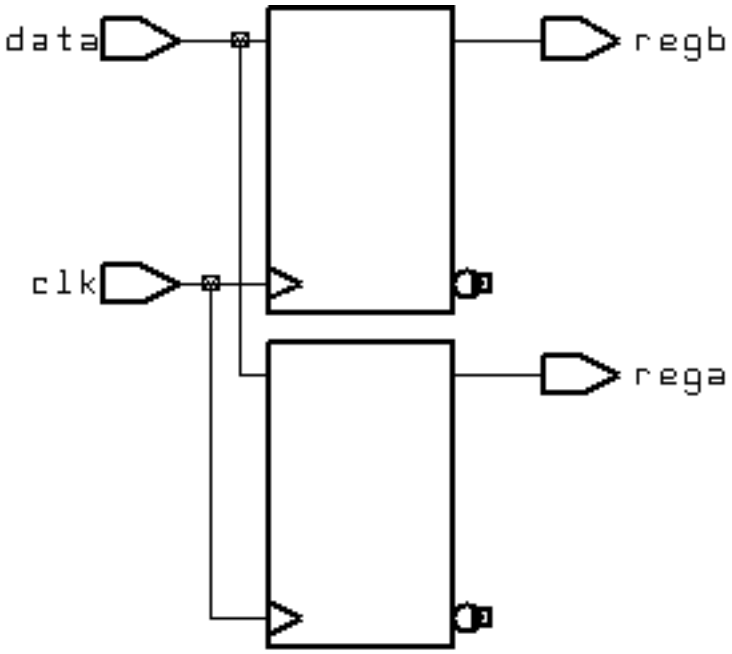
Example 5-12 Blocking Assignment

```
module rtl (clk, data, rega, regb);
input data, clk;
output rega, regb;

reg rega, regb;

always @(posedge clk)
begin
     rega = data;
     regb = rega;
end
endmodule
```

Figure 5-2 Schematic of Blocking Assignment



begin...end Block Statements

Using block statements is a way of syntactically grouping several statements into a single statement.

In Verilog, sequential blocks are delimited by the keywords `begin` and `end`. These `begin...end` pairs are commonly used in conjunction with `if`, `case`, and `for` statements to group several statements. Functions and `always` blocks that contain more than one statement require a `begin...end` pair to group the statements. Verilog also provides a construct called a named block, as in Example 5-13.

Example 5-13 Block Statement With a Named Block

```
begin : block_name
   reg local_variable_1;
     integer local_variable_2;
     parameter local_variable_3;
     ... statements ...
end
```

In Verilog, no semicolon (`;`) follows the `begin` or `end` keywords. You identify named blocks by following the `begin` with a colon (`:`) and a `block_name`, as shown. Verilog syntax allows you to declare variables locally in a named block. You can include `reg`, `integer`, and `parameter` declarations within a named block but not in an unnamed block. Named blocks allow you to use the `disable` statement.

if...else Statements

The `if...else` statements execute a block of statements according to the value of one or more expressions.

The syntax of `if...else` statements is

```
if ( expr )
     begin
     ... statements ...
     end
else
     begin
     ... statements ...
     end
```

The `if` statement consists of the keyword `if` followed by an expression in parentheses. The `if` statement is followed by a statement or block of statements enclosed by `begin` and `end`. If the value of the expression is nonzero, the expression is true and the statement block that follows is executed. If the value of the expression is zero, the expression is `false` and the statement block that follows is not executed.

An optional `else` statement can follow an `if` statement. If the expression following `if` is false, the statement or block of statements following `else` is executed.

The `if...else` statements can cause synthesis of registers. Registers are synthesized when you do not assign a value to the same `reg` in all branches of a conditional construct. Information on registers is in “Register Inference” on page 6-2.

HDL Compiler synthesizes multiplexer logic (or similar select logic) from a single `if` statement. The conditional expression in an `if` statement is synthesized as a control signal to a multiplexer, which determines the appropriate path through the multiplexer. For example, the statements in Example 5-14 create multiplexer logic controlled by c and place either a or b in the variable x.

Example 5-14 if Statement That Synthesizes Multiplexer Logic

```
if (c)
     x = a;
else
     x = b;
```

Example 5-15 illustrates how `if` and `else` can be used to create an arbitrarily long `if...else if...else` structure.

Example 5-15 if...else if...else Structure

```
if (instruction == ADD)
    begin
        carry_in = 0;
        complement_arg = 0;
    end
else if (instruction == SUB)
    begin
        carry_in = 1;
        complement_arg = 1;
    end
else
    illegal_instruction = 1;
```

Example 5-16 shows how to use nested `if` and `else` statements.

Example 5-16 Nested if and else Statements

```
if (select[1])
    begin
        if (select[0]) out = in[3];
        else out = in[2];
    end
else
    begin
        if (select[0]) out = in[1];
        else out = in[0];
    end
```

Conditional Assignments

HDL Compiler can synthesize a latch for a conditionally assigned variable. A variable is conditionally assigned if there is a path that does not explicitly assign a value to that variable.

In Example 5-17, the variable `value` is conditionally driven. If `c` is not true, `value` is not assigned and retains its previous value.

Example 5-17 Synthesizing a Latch for a Conditionally Driven Variable

```
always begin
  if ( c ) begin
    value = x;
  end
  Y = value; //causes a latch to be synthesized for value
end
```

case Statements

The `case` statement is similar in function to the `if...else` conditional statement. The `case` statement allows a multipath branch in logic that is based on the value of an expression. One way to

describe a multicycle circuit is with a `case` statement (see Example 5-18). Another way is with multiple @ (clock edge) statements, which are discussed in the subsequent sections on loops.

The syntax for a case statement is

```
case ( expr )
     case_item1: begin

     ... statements ...

     end
     case_item2: begin

     ... statements ...

     end
     default: begin

     ... statements ...
     end
endcase
```

The `case` statement consists of the keyword `case`, followed by an expression in parentheses, followed by one or more case items (and associated statements to be executed), followed by the keyword `endcase`. A case item consists of an expression (usually a simple constant) or a list of expressions separated by commas, followed by a colon (`:`).

The expression following the `case` keyword is compared with each case item expression, one by one. When the expressions are equal, the condition evaluates to true. Multiple expressions separated by commas can be used in each case item. When multiple expressions are used, the condition is said to be true if any of the expressions in the case item match the expression following the `case` keyword.

The first case item that evaluates to true determines the path. All subsequent case items are ignored, even if they are true. If no case item is true, no action is taken.

You can define a default case item with the expression `default`, which is used when no other case item is true.

An example of a `case` statement is shown in Example 5-18.

Example 5-18 case Statement

```
case (state)
    IDLE: begin
        if (start)
            next_state = STEP1;
        else
            next_state = IDLE;
    end
    STEP1: begin
        //do first state processing here
        next_state = STEP2;
    end
    STEP2: begin
        //do second state processing here
        next_state = IDLE;
    end
endcase
```

Full Case and Parallel Case

HDL Compiler automatically determines whether a `case` statement is full or parallel. A `case` statement is full if all possible branches are specified. If you do not specify all possible branches but you know that one or more branches can never occur, you can declare a `case` statement as full-case with the `// synopsys full_case` directive.

Otherwise, HDL Compiler synthesizes a latch. See “parallel_case Directive” on page 9-8 and “full_case Directive” on page 9-10 for more information.

HDL Compiler synthesizes optimal logic for the control signals of a `case` statement. If HDL Compiler cannot determine that branches are parallel, it synthesizes hardware that includes a priority encoder. If HDL Compiler can determine that no cases overlap (parallel case), it synthesizes a multiplexer, because a priority encoder is not necessary. You can also declare a `case` statement as parallel case with the `//synopsys parallel_case` directive. See “full_case Directive” on page 9-10. Example 5-19 does not result in either a latch or a priority encoder.

Example 5-19 A case Statement That Is Both Full and Parallel

```
input [1:0] a;
always @(a or w or x or y or z) begin
     case (a)
          2’b11:
              b = w ;
          2’b10:
              b = x ;
          2’b01:
              b = y ;
          2’b00:
              b = z ;
     endcase
end
```

Example 5-20 shows a case statement that is missing branches for the cases 2’b01 and 2’b10. Example 5-20 infers a latch for b.

Example 5-20 A case Statement That Is Parallel but Not Full

```
input [1:0] a;
always @(a or w or z) begin
    case (a)
        2'b11:
            b = w ;
        2'00:
            b = z ;
    endcase
end
```

The `case` statement in Example 5-21 is not parallel or full, because the values of inputs `w` and `x` cannot be determined. However, if you know that only one of the inputs equals `2'b11` at a given time, you can use the `// synopsys parallel_case` directive to avoid synthesizing a priority encoder. If you know that either `w` or `x` always equals `2'b11` (a situation known as a one-branch tree), you can use the `// synopsys full_case` directive to avoid synthesizing a latch.

Example 5-21 A case Statement That Is Not Full or Parallel

```
always @(w or x) begin
    case (2'b11)
        w:
            b = 10 ;
        x:
            b = 01 ;
    endcase
end
```

casex Statements

The `casex` statement allows a multipath branch in logic, according to the value of an expression, just as the `case` statement does. The differences between the `case` statement and the `casex` statement are the keyword and the processing of the expressions.

The syntax for a `casex` statement is

```
casex ( expr )
     case_item1: begin
     ... statements ...
     end
     case_item2: begin
     ... statements ...
     end
     default: begin
     ... statements ...
     end
endcase
```

A case item can have expressions consisting of

- A simple constant
- A list of identifiers or expressions separated by commas, followed by a colon (`:`)
- Concatenated, bit-selected, or part-selected expressions
- A constant containing `z`, `x`, or `?`

When a `z`, `x`, or `?` appears in a case item, it means that the corresponding bit of the `casex` expression is not compared. Example 5-22 shows a case item that includes an `x`.

Example 5-22 casex Statement With x

```
reg [3:0] cond;
casex (cond)
    4'b100x: out = 1;
    default: out = 0;
endcase
```

In Example 5-22, out is set to 1 if cond is equal to 4'b1000 or 4'b1001, because the last bit of cond is defined as x.

Example 5-23 shows a complicated section of code that can be simplified with a casex statement that uses the ? value.

Example 5-23 Before Using casex With ?

```
if (cond[3]) out = 0;
else if (!cond[3] & cond[2] ) out = 1;
else if (!cond[3] & !cond[2] & cond[1] ) out = 2;
else if (!cond[3] & !cond[2] & !cond[1] & cond[0] ) out = 3;
else if (!cond[3] & !cond[2] & !cond[1] & !cond[0] ) out = 4;
```

Example 5-24 shows the simplified version of the same code.

Example 5-24 After Using casex With ?

```
casex (cond)
    4'b1???: out = 0;
    4'b01??: out = 1;
    4'b001?: out = 2;
    4'b0001: out = 3;
    4'b0000: out = 4;
endcase
```

HDL Compiler allows ?, z, and x bits in case items but not in *casex* expressions. Example 5-25 shows an invalid *casex* expression.

Example 5-25 Invalid casex Expression

```
express = 3'bxz?;
    ...
casex (express) //illegal testing of an expression
    ...
endcase
```

casez Statements

The `casez` statement allows a multipath branch in logic according to the value of an expression, just like the `case` statement. The differences between the `case` statement and the `casez` statement are the keyword and the way the expressions are processed. The `casez` statement acts exactly the same as `casex`, except that `x` is not allowed in case items; only `z` and `?` are accepted as special characters.

The syntax for a `casez` statement is

```
casez ( expr )
     case_item1: begin
     ... statements ...
     end
     case_item2: begin
     ... statements ...
     end
     default: begin
     ... statements ...
     end
endcase
```

A case item can have expressions consisting of

- A simple constant
- A list of identifiers or expressions separated by commas, followed by a colon (:)

- Concatenated, bit-selected, or part-selected expressions
- A constant containing z or ?

When a `casez` statement is evaluated, the value z in the case item is ignored. An example of a `casez` statement with z in the case item is shown in Example 5-26.

Example 5-26 casez Statement With z

```
casez (what_is_it)
  2'bz0: begin
      //accept anything with least significant bit zero
     it_is = even;
  end
  2'bz1: begin
      //accept anything with least significant bit one
     it_is = odd;
  end
endcase
```

HDL Compiler allows ? and z bits in case items but not in `casez` expressions. Example 5-27 shows an invalid expression in a `casez` statement.

Example 5-27 Invalid casez Expression

```
express = 1'bz;
    ...
casez (express)  //illegal testing of an expression
    ...
endcase
```

for Loops

The `for` loop repeatedly executes a single statement or block of statements. The repetitions are performed over a range determined by the range expressions assigned to an index. Two range

expressions appear in each `for` loop: `low_range` and `high_range`. In the syntax lines that follow, `high_range` is greater than or equal to `low_range`. HDL Compiler recognizes incrementing as well as decrementing loops. The statement to be duplicated is surrounded by `begin` and `end` statements.

Note:

HDL Compiler allows four syntax forms for a `for` loop. They are

```
for (index = low_range;index < high_range;index = index + step)
for (index = high_range;index > low_range;index = index - step)
for (index = low_range;index <= high_range;index = index + step)
for (index = high_range;index >= low_range;index = index - step)
```

Example 5-28 shows a simple `for` loop.

Example 5-28 A Simple for Loop

```
for (i = 0; i <= 31; i = i + 1) begin
    s[i] = a[i] ^ b[i] ^ carry;
    carry = a[i] & b[i]  |  a[i] & carry  | b[i] & carry;
end
```

The `for` loops can be nested, as shown in Example 5-29.

Example 5-29 Nested for Loops

```
for (i = 6; i >= 0; i = i - 1)
    for (j = 0; j <= i; j = j + 1)
        if (value[j] > value[j+1]) begin
            temp = value[j+1];
            value[j+1] = value[j];
            value[j] = temp;
        end
```

You can use `for` loops as duplicating statements. Example 5-30 shows a `for` loop that is expanded into its longhand equivalent in Example 5-31.

Example 5-30 Example for Loop

```
for ( i=0; i < 8; i=i+1 )
    example[i] = a[i] & b[7-i];
```

Example 5-31 Expanded for Loop

```
example[0] = a[0] & b[7];
example[1] = a[1] & b[6];
example[2] = a[2] & b[5];
example[3] = a[3] & b[4];
example[4] = a[4] & b[3];
example[5] = a[5] & b[2];
example[6] = a[6] & b[1];
example[7] = a[7] & b[0];
```

while Loops

The `while` loop executes a statement until the controlling expression evaluates to false. A `while` loop creates a conditional branch that must be broken by one of the following statements to prevent combinational feedback.

```
@ (posedge clock)
```

or

```
@ (negedge clock)
```

HDL Compiler supports `while` loops if you insert one of these expressions in every path through the loop:

```
@ (posedge clock)
```

or

```
@ (negedge clock)
```

Example 5-32 shows an unsupported `while` loop that has no event expression.

Example 5-32 Unsupported while Loop

```
always
     while (x < y)
          x = x + z;
```

If you add `@ (posedge clock)` expressions after the `while` loop in Example 5-32, you get the supported version shown in Example 5-33.

Example 5-33 Supported while Loop

```
always
     begin @ (posedge clock)
          while (x < y)
          begin
               @ (posedge clock);
               x = x + z;
     end
end
```

forever Loops

Infinite loops in Verilog use the keyword `forever`. You must break up an infinite loop with an `@ (posedge clock)` or `@ (negedge clock)` expression to prevent combinational feedback, as shown in Example 5-34.

Example 5-34 Supported forever Loop

```
always
     forever
     begin
          @ (posedge clock);
          x = x + z;
     end
```

You can use `forever` loops with a `disable` statement to implement synchronous resets for flip-flops. The `disable` statement is described in the next section. See “Register Inference” on page 6-2 for more information on synchronous resets.

Using the style illustrated in Example 5-34 is not a good idea, because you cannot test it. The synthesized state machine does not reset to a known state; therefore, it is impossible to create a test program for it. Example 5-36 on page 5-31 illustrates how a synchronous reset for the state machine can be synthesized.

disable Statements

HDL Compiler supports the `disable` statement when you use it in `named` blocks. When a `disable` statement is executed, it causes the `named` block to terminate. A comparator description that uses `disable` is shown in Example 5-35.

Example 5-35 Comparator Using disable

```
begin : compare
     for (i = 7; i >= 0; i = i - 1) begin
     if (a[i] != b[i]) begin
          greater_than = a[i];
          less_than = ~a[i];
          equal_to = 0;
          //comparison is done so stop looping
          disable compare;
       end
     end

// If we get here a == b
// If the disable statement is executed, the next three
// lines will not be executed
   greater_than = 0;
   less_than = 0;
   equal_to = 1;
end
```

Example 5-35 describes a combinational comparator. Although the description appears sequential, the generated logic runs in a single clock cycle.

You can also use a `disable` statement to implement a synchronous reset, as shown in Example 5-36.

Example 5-36 Synchronous Reset of State Register Using disable in a forever Loop

```
always
     forever
     begin: Block
          @ (posedge clk)
          if (Reset)
               begin
                    z <= 1'b0;
                    disable Block;
               end
                    z <= a;
     end
```

The `disable` statement in Example 5-36 causes the block `Block` to terminate immediately and return to the beginning of the block.

task Statements

In Verilog, `task` statements are similar to functions, but `task` statements can have output and inout ports. You can use the `task` statement to structure your Verilog code so that a portion of code is reusable.

In Verilog, tasks can have timing controls and can take a nonzero time to return. However, HDL Compiler ignores all timing controls, so synthesis might disagree with simulation if timing controls are critical to the function of the circuit.

Example 5-37 shows how a `task` statement is used to define an adder function.

Example 5-37 Using the task Statement

```
module task_example (a,b,c);
     input [7:0] a,b;
     output [7:0] c;
     reg [7:0] c;

task adder;
     input [7:0] a,b;
     output [7:0] adder;
     reg c;
     integer i;

     begin
          c = 0;
          for (i = 0; i <= 7; i = i+1) begin
               adder[i] = a[i] ^ b[i] ^ c;
               c = (a[i] & b[i]) | (a[i] & c) | (b[i] & c);
          end
     end
endtask
     always
          adder (a,b,c);  //c is a reg

endmodule
```

Note:

Only `reg` variables can receive output values from a task; `wire` variables cannot.

always Blocks

An `always` block can imply latches or flip-flops, or it can specify purely combinational logic. An `always` block can contain logic triggered in response to a change in a level or the rising or falling edge of a signal. The syntax of an `always` block is

```
always @ ( event-expression [or event-expression*] )   begin
    ... statements ...
end
```

Event Expression

The event expression declares the triggers or timing controls. The word *or* groups several triggers. The Verilog language specifies that if triggers in the event expression occur, the block is executed. Only one trigger in a group of triggers needs to occur for the block to be executed. However, HDL Compiler ignores the event expression unless it is a synchronous trigger that infers a register. See Chapter 6, “Register, Multibit, Multiplexer, and Three-State Inference,” for details.

Example 5-38 shows a simple example of an always block with triggers.

Example 5-38 A Simple always Block

```
always @ ( a or b or c ) begin
    f = a & b & c
end
```

In Example 5-38, a, b, and c are asynchronous triggers. If any triggers change, the simulator resimulates the `always` block and recalculates the value of `f`. HDL Compiler ignores the triggers in this example, because they are not synchronous. However, you must indicate all

variables that are read in the a`lways` block as triggers. If you do not indicate all the variables as triggers, HDL Compiler gives a warning message similar to the following:

```
Warning: Variable 'foo' is being read in block 'bar' declared
on line 88 but does not occur in the timing control of the
block.
```

For a synchronous a`lways` block, HDL Compiler does not require listing of all variables.

Any of the following types of event expressions can trigger an a`lways` block:

- A change in a specified value. For example,

  ```
  always @ ( identifier ) begin
     ...  statements ...
  end
  ```

  In the previous example, HDL Compiler ignores the trigger.

- The rising edge of a clock. For example,

  ```
  always @ ( posedge event ) begin
     ... statements ...
  end
  ```

- The falling edge of a clock. For example,

  ```
  always @ ( negedge event ) begin
     ... statements ...
  end
  ```

- A clock or an asynchronous preload condition. For example,

```
always @ ( posedge CLOCK or negedge reset ) begin
   if !reset begin
    ... statements ...
   end
   else begin
    ... statements ...
   end
end
```

- An asynchronous preload that is based on two events joined by the word *or*. For example,

```
always @ ( posedge CLOCK or posedge event1 or
           negedge event2 ) begin
    if ( event1 ) begin
     ... statements ...
    end
    else if ( !event2 ) begin
     ... statements ...
    end
    else begin
     ... statements ...
    end
end
```

When the event expression does not contain posedge or negedge, combinational logic (no registers) is usually generated, although flow-through latches can be generated.

Note:

The statements `@ (posedge clock)` and `@ (negedge clock)` are not supported in functions or tasks.

Incomplete Event Specification

You risk misinterpretation of an `always` block if you do not list all the signals entering an `always` block in the event specification. Example 5-39 shows an incomplete event list.

Example 5-39 Incomplete Event List

```
always @(a or b) begin
     f = a & b & c;
end
```

HDL Compiler builds a 3-input `AND` gate for the description in Example 5-39, but in simulation of this description, `f` is not recalculated when c changes, because c is not listed in the event expression. The simulated behavior is not that of a 3-input `AND` gate.

The simulated behavior of the description in Example 5-40 is correct, because it includes all the signals in the event expression.

Example 5-40 Complete Event List

```
always @(a or b or c) begin
   f = a & b & c;
end
```

In some cases, you cannot list all the signals in the event specification. Example 5-41 illustrates this problem.

Example 5-41 Incomplete Event List for Asynchronous Preload

```
always @ (posedge c or posedge p)
     if (p)
          z = d;
     else
          z = a;
```

In the logic synthesized for Example 5-41, if d changes while p is high, the change is reflected immediately in the output, z. However, when this description is simulated, z is not recalculated when d changes, because d is not listed in the event specification. As a result, synthesis might not match simulation.

Asynchronous preloads can be correctly modeled in HDL Compiler only when you want changes in the load data to be reflected immediately in the output. In Example 5-41, data d must change to the preload value before preload condition p transits from low to high. If you attempt to read a value in an asynchronous preload, HDL Compiler prints a warning similar to the following:

```
Warning:Variable 'd' is being read asynchronously in routine
reset line 21 in file '/usr/tests/hdl/asyn.v'. This may cause
simulation-synthesis mismatches.
```