Table 4.3 Precedence of Verilog operators.

Operator type	Operator symbols	Precedence
Complement	! ~ –	Highest precedence
Arithmetic	* / + –	
Shift	<< >>	
Relational	< <= > >=	
Equality	== !=	
Reduction	& ~& ^ ~^ | ~|	
Logical	&& ||	
Conditional	?:	Lowest precedence

Conditional Operator

The conditional operator is discussed fully in Section 4.6.1.

Operator Precedence

The Verilog operators are assumed to have the precedence indicated in Table 4.3. The order of precedence is from top to bottom; operators in the top row have the highest precedence and those in the bottom row have the lowest precedence. The operators listed in the same row have the same precedence.

The designer can use parentheses to change the precedence of operators in Verilog code or remove any possible misinterpretation. It is a good practice to use parentheses to make the code unambiguous and easy to read.

4.6.6 THE GENERATE CONSTRUCT

In Section 3.5.4 we introduced the **generate** loop capability which can be used to create multiple instances of subcircuits. A subcircuit may be defined in a block of statements delineated by the **generate** and **endgenerate** keywords. The subcircuit is instantiated multiple times using a generate-index variable. This variable is defined using the **genvar** keyword and it can have only positive integer values. It is not possible to use an index declared as a normal **integer** variable.

Example 4.21 Figure 4.41 shows how the **generate** construct can be used to specify an n-bit ripple-carry adder. The subcircuit is a full-adder defined structurally in terms of primitive gates as introduced in Figure 3.18. The **for** loop causes the full-adder block to be instantiated n times.

```
module addern (carryin, X, Y, S, carryout);
   parameter n = 32;
   input carryin;
   input [n–1:0] X, Y;
   output [n–1:0] S;
   output carryout;
   wire [n:0] C;

   genvar k;
   assign C[0] = carryin;
   assign carryout = C[n];
   generate
      for (k = 0; k < n; k = k+1)
      begin: fulladd_stage
         wire z1, z2, z3;  //wires within full-adder
         xor (S[k], X[k], Y[k], C[k]);
         and (z1, X[k], Y[k]);
         and (z2, X[k], C[k]);
         and (z3, Y[k], C[k]);
         or (C[k+1], z1, z2, z3);
      end
   endgenerate

endmodule
```

Figure 4.41 Using the **generate** loop to define an n-bit ripple-carry adder.

In this example, the **for** statement is used in the **generate** block to control the selection of the generated objects. The **generate** block can also contain **if-else** and **case** statements to determine which objects are generated.

4.6.7 TASKS AND FUNCTIONS

In high-level programming languages it is possible to use subroutines and functions to avoid replicating specific routines that may be needed in several places of a given program. Verilog provides similar capabilities, known as tasks and functions. They can be used to modularize large designs and make the Verilog code easier to understand.

Verilog Task

A task is declared by the keyword **task** and it comprises a block of statements that ends with the keyword **endtask**. The task must be included in the module that calls it. It may

have input and output ports. These are not the ports of the module that contains the task, which are used to make external connections to the module. The task ports are used only to pass values between the module and the task.

Example 4.22 In Figure 4.29 we showed the Verilog code for a 16-to-1 multiplexer that instantiates five copies of a 4-to-1 multiplexer circuit given in a separate module named *mux4to1*. The same circuit can be specified using the task approach as shown in Figure 4.42. Observe the key differences. The task *mux4to1* is included in the module *mux16to1*. It is called from an **always** block by means of an appropriate **case** statement. The output of a task must be a variable, hence *g* is of **reg** type.

```
module mux16to1 (W, S16, f);
   input [0:15] W;
   input [3:0] S16;
   output reg f;

   always @(W, S16)
      case (S16[3:2])
         0: mux4to1 (W[0:3], S16[1:0], f);
         1: mux4to1 (W[4:7], S16[1:0], f);
         2: mux4to1 (W[8:11], S16[1:0], f);
         3: mux4to1 (W[12:15], S16[1:0], f);
      endcase

   // Task that specifies a 4-to-1 multiplexer
   task mux4to1;
      input [0:3] X;
      input [1:0] S4;
      output reg g;

      case (S4)
         0: g = X[0];
         1: g = X[1];
         2: g = X[2];
         3: g = X[3];
      endcase
   endtask

endmodule
```

Figure 4.42 Use of a task in Verilog code.

Verilog Function

A function is declared by the keyword **function** and it comprises a block of statements that ends with the keyword **endfunction**. The function must have at least one input and it returns a single value that is placed where the function is invoked.

Example 4.23

Figure 4.43 shows how the code in Figure 4.42 can be written to use a function. The Verilog compiler essentially inserts the body of the function at each place where it is called. Hence the clause

```
0: f = mux4to1 (W[0:3], S16[1:0]);
```

becomes

```
module mux16to1 (W, S16, f);
    input [0:15] W;
    input [3:0] S16;
    output reg f;

    // Function that specifies a 4-to-1 multiplexer
    function mux4to1;
        input [0:3] X;
        input [1:0] S4;

        case (S4)
            0: mux4to1 = X[0];
            1: mux4to1 = X[1];
            2: mux4to1 = X[2];
            3: mux4to1 = X[3];
        endcase
    endfunction

    always @(W, S16)
        case (S16[3:2])
            0: f = mux4to1 (W[0:3], S16[1:0]);
            1: f = mux4to1 (W[4:7], S16[1:0]);
            2: f = mux4to1 (W[8:11], S16[1:0]);
            3: f = mux4to1 (W[12:15], S16[1:0]);
        endcase

endmodule
```

Figure 4.43 The code from Figure 4.42 using a function.

```
0: case (S16[1:0])
       0: f = W[0];
       1: f = W[1];
       2: f = W[2];
       3: f = W[3];
   endcase
```

The function serves as a convenience that makes the *mux16to1* module more compact.

A Verilog function can invoke another function but it cannot call a Verilog task. A task may call another task and it may invoke a function. In Figure 4.42 we defined the task after the **always** block that calls it. In contrast, in Figure 4.43 we defined the function before the **always** block that invokes it. Both possibilities are allowed in the Verilog standard for both tasks and functions. However, some tools require functions to be defined before the statements that invoke them.

4.7 CONCLUDING REMARKS

This chapter has introduced a number of circuit building blocks. Examples using these blocks to construct larger circuits will be presented in later chapters. To describe the building block circuits efficiently, several Verilog constructs have been introduced. In many cases a given circuit can be described in various ways, using different constructs. A circuit that can be described using an **if-else** statement can also be described using a **case** statement or perhaps a **for** loop. In general, there are no strict rules that dictate when one style should be preferred over another. With experience the user develops a sense for which types of statements work well in a particular design situation. Personal preference also influences how the code is written.

Verilog is not a programming language, and Verilog code should not be written as if it were a computer program. The statements discussed in this chapter can be used to create large, complex circuits. A good way to design such circuits is to construct them using well-defined modules, in the manner that we illustrated for the multiplexers, decoders, encoders, and so on. Additional examples using the Verilog statements introduced in this chapter are given in Chapters 5 and 6. In Chapter 7 we provide a number of examples of using Verilog code to describe larger digital systems. For more information on Verilog, the reader can consult more specialized books [2–8].

In the next chapter we introduce logic circuits that include the ability to store logic signal values in memory elements.