appendix

A

VERILOG REFERENCE

This appendix describes the features of Verilog used in this book. It is meant to serve as a convenient reference for the reader, hence only brief descriptions are provided, along with examples. The reader is encouraged to first study the introduction to Verilog in Section 2.10.

This appendix is not meant to be a comprehensive Verilog manual. While we discuss almost all the features of Verilog that are useful in the synthesis of logic circuits, we do not discuss many of the features that are useful only for simulation of circuits. Although the omitted features are not needed for any of the examples used in this book, a reader who wishes to learn more about Verilog can refer to specialized texts [1–7].

How to Write Verilog Code

The tendency for the novice is to write Verilog code that resembles a computer program, containing many variables and loops. It is difficult to determine what logic circuit the CAD tools will produce when synthesizing such code. The task of a synthesis tool is to analyze a piece of Verilog code and determine, according to the semantics of the language, what circuit can be used to implement the code. Consider a code fragment such as

```
if (s == 0)
    f = w0;
else
    f = w1;
```

We can understand the semantics by considering each statement in sequence, in the way that a simulation tool would. The code results in f being assigned the value of either w_0 or w_1, depending on the value of s. A synthesis tool would usually implement this behavior using a multiplexer circuit.

In general, synthesis tools have to recognize certain structures in code, like the above multiplexer. From the practical point of view, this will work only if users write code that conforms to a commonly used style. The beginning Verilog user should therefore adopt the style of code recommended by experienced designers. This book contains more than 120 examples of Verilog code that represent a wide range of logic circuits. In all of these examples the code is easily related to the described logic circuit. The reader is encouraged to adopt the same style of code. A good approach is to "write Verilog code that obviously represents the intended circuit."

Although Verilog is a fairly straightforward language to learn and use, the novice designer will tend to make some common errors in syntax and semantics. A list of typical errors is given in Section A.15, as well as a set of guidelines that expert Verilog coders recommend as good style for writing clear and effective code.

Once complete Verilog code is written for a particular design, it is useful to analyze the resulting circuit synthesized by the CAD tools. Much can be learned about Verilog, logic circuits, and logic synthesis through this process.

A.1 DOCUMENTATION IN VERILOG CODE

Documentation can be included in Verilog code by writing a comment. A *short* comment begins with the double slash, //, and continues to the end of the line. A long comment can span multiple lines and is contained inside the delimiters /* and */. Examples of comments are

```
// This is a short comment
/*This is a long Verilog comment
  that spans two lines */
```

A.2 WHITE SPACE

White space characters, such as SPACE and TAB, and blank lines are ignored by the Verilog compiler. Multiple statements can be written on a single line, such as

```
f = w0; if (s == 1) f = w1;
```

Although legal, this code is hard to read and uses poor style. Placing each statement on a separate line, and using indentation within blocks of code, such as an **if-else** statement, are good ways to increase the readability of code.

A.3 SIGNALS IN VERILOG CODE

In Verilog, a signal in a circuit is represented as a *net* or a *variable* with a specific type. The term *net* is derived from the electrical jargon, where it refers to the interconnection of two or more points in a circuit. A net or variable declaration has the form

type [range] signal_name{, signal_name};

The square brackets indicate an optional field, and the curly brackets indicate that additional entries are permitted. We will use this syntax throughout the appendix. The *signal_name* is an identifier, as defined in the next section. Without the *range* field the declared net or variable is scalar and represents a single-bit signal. The range is used to specify vectors that correspond to multibit signals, as explained in Section A.6.

A.4 IDENTIFIER NAMES

Identifiers are the names of variables and other elements in Verilog code. The rules for specifying identifiers are simple: any letter or digit may be used, as well as the _ underscore and $ characters. There are two caveats: an identifier must not begin with a digit and it should not be a Verilog keyword. Examples of legal identifiers are *f*, *x*1, *x_y*, and *Byte*. Some examples of illegal names are 1*x*, +*y*, *x***y*, and *258*. Verilog is case sensitive, hence k is not the same as K, and BYTE is not the same as Byte.

For special purposes Verilog allows a second form of identifier, called an *escaped* identifier. Such identifiers begin with the (\) backslash character, which can then be followed by any printable ASCII characters except white spaces. Examples of escaped identifiers are \123, *sig-name*, and *a*+*b*. Escaped identifiers should not be used in normal Verilog code; they are intended for use in code produced automatically when other languages are translated into Verilog.

A.5 SIGNAL VALUES, NUMBERS, AND PARAMETERS

Verilog supports scalar nets and variables that represent individual signals, and vectors that correspond to multiple signals. Each individual signal can have four possible values:

0 = logic value 0
1 = logic value 1
z = tri-state (high impedance)
x = unknown value

The z and x values can also be denoted by the capital letters Z and X. The value x can be used to denote a don't-care condition in Verilog code; the symbol ? can also be used for this purpose. The value of a vector variable is specified by giving a constant of the form

[size]['radix]constant

where *size* is the number of bits in the constant, and *radix* is the number base. Supported radices are

d = decimal
b = binary
h = hexadecimal
o = octal

When no radix is specified, the default is decimal. If size specifies more bits than are needed to represent the given constant, then in most cases the constant is padded with zeros. The exceptions to this rule are when the first character of the constant is either x or z, in which case the padding is done using that value. Some examples of constants include

0	the number 0
10	the decimal number 10
'b10	the binary number $10 = (2)_{10}$
'h10	the hex number $10 = (16)_{10}$
4'b100	the binary number $0100 = (4)_{10}$
4'bx	an unknown 4-bit value xxxx
8'b1000_0011	_ can be inserted for readability
8'hfx	equivalent to 8'b1111_xxxx

A.5.1 PARAMETERS

A *parameter* associates an identifier name with a constant. Let the Verilog code include the following declarations:

```
parameter n = 4;
parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;
```

Then the identifier *n* can be used in place of the number 4, the name *S0* can be substituted for the value 2'b00, and so on. An important use of parameters is in the specification of parameterized subcircuits, which is described in Section A.12.

A.6 NET AND VARIABLE TYPES

Verilog defines a number of types of nets and variables. These types are defined by the language itself, and user-defined types are not permitted.

A.6.1 NETS

A *net* represents a node in a circuit. To distinguish between different types of circuit nodes there exist several types of nets, called *wire*, *tri*, and a number of others that are not needed for synthesis, and are not used in this book.

The **wire** type is employed to connect an output of one logic element in a circuit to an input of another logic element. The following are examples of scalar **wire** declarations.

```
wire x;
wire Cin, AddSub;
```

A vector **wire** represents multiple nodes, such as

```
wire [3:0] S;
wire [1:2] Array;
```

The square brackets are the syntax for specifying a vector's range. The range [R_a:R_b] can be either increasing or decreasing, as shown. In either case, R_a is the index of the most-significant (leftmost) bit in the vector, and R_b is the index of the least-significant (rightmost) bit. The indices R_a and R_b can be either positive or negative integers.

The net S can be used as a four-bit quantity, or each bit can be referred to individually as $S[3]$, $S[2]$, $S[1]$, and $S[0]$. If a value is assigned to S such as S = 4'b0011, the result is $S[3] = 0$, $S[2] = 0$, $S[1] = 1$, and $S[0] = 1$. The assignment of a single bit in a vector to another net, such as $f = S[0]$, is called a *bit-select* operation. A range of values from one vector can be assigned to another vector, which is called a *part-select* operation. If we assign Array = S[2:1], this produces $Array[1] = S[2]$ and $Array[2] = S[1]$. The index used in a bit-select operation can involve a variable, such as $S[i]$, while the indices used with a part-select operation have to be constant expressions, such as S[2:1].

The *tri* type denotes circuit nodes that are connected in a tri-state fashion. Examples of **tri** nets are

```
tri z;
tri [7:0] DataOut;
```

These nets are treated in the same manner as the **wire** type, and they are used only to enhance the readability of code that includes tri-state gates.

A.6.2 VARIABLES

Nets provide a means for interconnecting logic elements, but they do not allow a circuit to be described in terms of its *behavior*. For this purpose, Verilog provides *variables*. A variable can be assigned a value in one Verilog statement, and it retains this value until it is overwritten in a subsequent assignment statement. There are two types of variables, *reg* and *integer*. Consider the code fragment

```
Count = 0;
for (k = 0; k < 4; k = k + 1)
   if (S[k])
      Count = Count + 1;
```

The **for** and **if** statements are described in Section A.11. This code stores in *Count* the number of bits in S that have the value 1. Since it models the behavior of a circuit, *Count* has to be declared as a variable, rather than a simple **wire**. If *Count* has three bits, then the declaration is

```
reg [2:0] Count;
```

The keyword **reg** does *not* denote a storage element, or register. In Verilog code, **reg** variables can be used to model either combinational or sequential parts of a circuit. In our example, the variable k serves as a loop index. Such variables are declared as type **integer** in the statement

```
integer k;
```

Integer variables are useful for describing the behavior of a module, but they do not directly correspond to nodes in a circuit. In this book we use **integers** as loop control variables.

A.6.3 MEMORIES

A memory is a two-dimensional array of bits. Verilog allows such a structure to be declared as a variable (**reg** or **integer**) that is an array of vectors, such as

reg [7:0] R [3:0];

This statement defines R as four eight-bit variables named $R[3]$, $R[2]$, $R[1]$, and $R[0]$.

Two-level indexing, such as R[3][7], can be used. There is also support for higher dimension arrays. A three-dimensional array may be declared as

reg [7:0] R [3:0] [1:0];

This statement defines a three-dimensional array of bits.

A.7 OPERATORS

Verilog has a large number of operators, as shown in Table A.1. The first column gives the category, the second column indicates how each operator is used, and the third column specifies the number of bits produced in the result. To aid in describing the table, we use

Table A.1 Verilog operators and bit lengths.

Category	Examples	Bit Length
Bitwise	$\sim A, \quad +A, \quad -A$ $A \,\&\, B, \quad A \mid B, \quad A \sim\hat{}\, B, \quad A \,\hat{}\sim B$	L(A) MAX (L(A), L(B))
Logical	$!A, \quad A\&\&B, \quad A \parallel B$	1 bit
Reduction	$\&A, \quad \sim \&A, \quad \mid A, \quad \sim \mid A, \quad \hat{}\sim A, \quad \sim\hat{}\, A$	1 bit
Relational	$A == B, \quad A! = B, \quad A > B, \quad A < B$ $A >= B, \quad A <= B$ $A === B, \quad A! == B$	1 bit
Arithmetic	$A + B, \quad A - B, \quad A * B, \quad A/B$ $A \,\%\, B$	MAX (L(A), L(B))
Shift	$A << B, \quad A >> B$	L(A)
Concatenate	$\{A, \ldots, B\}$	L(A) + $\cdots$ + L(B)
Replication	$\{B\{A\}\}$	$B *$ L(A)
Condition	$A \,?\, B : C$	MAX (L(B), L(C))

operands named A, B, and C, which may be either vectors or scalars. The syntax $\sim A$ means that the $\sim$ operator is applied to the variable A, and the syntax $L(A)$ means that the result has the same number of bits (length) as in A.

Most of the operators in Table A.1 are also listed in Table 4.2 and described in detail in Section 4.6.5. The bitwise operators are 1's complement ($\sim$), unary plus ($+$), 2's complement ($-$), AND (&), OR (|), XOR ($\hat{}$), and XNOR ($\sim\hat{}$ or $\hat{}\sim$). The bitwise operators produce multibit results, usually with the same number of bits as the operands. For example, if $A = a_1a_0$, $B = b_1b_0$, and $C = c_1c_0$, then the operation $C = A \,\&\, B$ results in $c_1 = a_1 \,\&\, b_1$ and $c_0 = a_0 \,\&\, b_0$. Note that unary plus just denotes a positive number; it has no effect.

The logical operators generate a one-bit result. They are NOT (!), AND (&&), and OR (||). If the operand of ! is a vector, then $!A$ produces 1 (True) only if all bits in A are 0; otherwise $!A$ gives 0 (False). The result of $A \,\&\&\, B$ is 1 if both A and B are nonzero, while $A \;||\; B$ produces 1 unless both A and B are zero. If an operand is ambiguous (contains an x), the result is x. The logical operators are normally used in conditional statements such as **if** $((A < B) \,\&\&\, (B < C))$.

The reduction operators use the same symbols as some of the bitwise operators, but have only one operand. The reduction $\&A$ produces the AND of all of the bits in A, while $\sim\&A$ produces the NAND. Similarly, the other reduction operators produce single-bit Boolean results.

The relational operators give a 1 (True) or 0 (False) result based on the specified comparison of A and B. For synthesis of logic circuits A and B are usually **wire** or **reg** types, and Verilog treats them as unsigned numbers. If **integer** variables are supported, they may be treated as signed numbers. The result of a relational operation is ambiguous (x) if either operand has any unspecified digits. An exception is for the $===$ and $!==$ operators, which check for equality and inequality, respectively. These operators compare equality of x and z digits in addition to 0 and 1 digits.

Verilog includes the normal arithmetic operators +, −, *, and /. The modulus operator (%) is also included, but it is usually not supported for synthesis, except for use in calculating a compile-time constant. The operation $A \,\%\, B$ returns the remainder of the integer division $A \div B$. Arithmetic operands of type **wire** and **reg** are treated as unsigned numbers. If the two operands are of unequal size, zero digits are padded on the left, and bits are truncated if the result has fewer digits than the largest operand. **Integer** variables are considered as 2's complement numbers.

The $<<$ and $>>$ operators perform logical shifts to the left and right, respectively. For a left shift, zeros are shifted into the LSB, while for a right shift, zeros are shifted into the MSB. For synthesis, the operand B should be a constant.

The {,} concatenate operator allows vectors to be combined to produce a larger resulting vector. Any operand that is a constant must have a specified size, as in 4'b0011. Operands can be repeated multiple times by using the replication operator. The operation {{3{A}}, {2{B}}} is equivalent to $\{A, A, A, B, B\}$. The replication operator can be used to form an n-bit vector of digits: the operation {n{1'b1}} represents n ones.

The last item in Table A.1 is the ? : conditional operator. The result of A ? B : C is equal to B if operand A evaluates to 1 (True); otherwise, the result is C. In the case that A evaluates to x, the conditional operator generates a bitwise output; each bit in the result is 1 if the corresponding bits in both B and C are 1, 0 if these bits are 0, and x otherwise.

The precedence of Verilog operators follows similar rules as in arithmetic and Boolean algebra. For example, * has precedence over +, and & has precedence over |. A complete listing of the precedence rules is given in Table 4.3.

A.8 VERILOG MODULE

A circuit or subcircuit described with Verilog code is called a *module*. Figure A.1 gives the general structure of a **module** declaration. The module has a name, *module_name*, which can be any valid identifier, followed by a list of ports. The term *port* is adopted from the electrical jargon, in which it refers to an input or output connection in an electrical circuit. The ports can be of type **input**, **output**, or **inout** (bidirectional), and can be either scalar or vector. Examples of ports are

```
input Cin, x, y;
input [3:0] X, Y;
output Cout, s;
inout [7:0] Bus;
output [3:0] S;

wire Cout, s;
wire [7:0] Bus;
reg [3:0] S;
```

As shown, output and inout ports have an associated type. We assume that *Cout*, *s*, and *Bus* are nets in this example, while *S* is a variable. The **wire** declarations can actually be

```
module module_name [(port_name{, port_name})];
   [parameter declarations]
   [input declarations]
   [output declarations]
   [inout declarations]
   [wire or tri declarations]
   [reg or integer declarations]
   [function or task declarations]
   [assign continuous assignments]
   [initial block]
   [always blocks]
   [gate instantiations]
   [module instantiations]
endmodule
```

Figure A.1 The general form of a module.

omitted, because Verilog assumes that signals are nets by default. However, any port used as a variable must be explicitly declared as such.

Instead of using a separate statement to declare that the variable *S* is of **reg** type, we can include this declaration in the statement that specifies *S* to be an **output**, as follows:

output reg [3:0] S;

This is the style we use throughout the book.

As Figure A.1 indicates, a module may contain any number of net (**wire** or **tri**) or variable (**reg** or **integer**) declarations, and a variety of other types of statements that are described later in this appendix.

Figure A.2 gives the Verilog code for a module *fulladd*, which represents a full-adder circuit. (The full-adder is discussed in Section 3.2.) The input port *Cin* is the carry-in, and the bits to be added are the input ports *x* and *y*. The output ports are the sum, *s*, and the carry-out, *Cout*. The functionality of the full-adder is described with logic equations preceded by the keyword **assign**, which is discussed in Section A.10.

There is usually more than one way to describe a given circuit using Verilog. Figure A.3 gives another version of the *fulladd* module, in which the functionality is specified by using the concatenate and addition operators. The statement

assign {Cout, s} = x + y + Cin;

assigns the least-significant bit in the result $x + y + Cin$ to the output *s* and the most-significant bit to *Cout*. The circuits generated from the modules in Figures A.2 and A.3 are the same.

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin;
   assign Cout = (x & y) | (Cin & x) | (Cin & y);

endmodule
```

Figure A.2 A full-adder module.

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign {Cout, s} = x + y + Cin;

endmodule
```

Figure A.3 A full-adder module defined using the + operator.

A.9 GATE INSTANTIATIONS

Verilog includes predefined modules that implement basic logic gates. These gates allow a circuit's structure to be described using *gate instantiation* statements of the form

gate_name [instance_name] (output_port, input_port{, input_port});

The *gate_name* specifies the desired type of gate, and the *instance_name* is any unique identifier. Each gate may have a different number of ports, with the output port listed first, followed by a variable number of input ports. An example of using gates to realize a full-adder is given in Figure A.4. The code defines four **wire** nets, $z1$ to $z4$, that connect the gates together, and each gate has a specified instance name. Figure A.5 shows a simpler version, in which instance names are not included and the declarations of $z1$ to $z4$ are omitted. Since the nets are not explicitly declared, they are implicitly assumed to be of type **wire**.

The logic gates supported in Verilog are summarized in Table A.2. The second column describes the function of each gate, and the rightmost column gives an example of instantiating the gate. Verilog allows gates with any number of inputs to be specified, but some CAD systems set practical limits. The *notif* and *bufif* gates represent tri-state buffers (drivers). The gate *notif0* is an inverting tri-state buffer with active-low enable, and *notif1* provides the same functionality with an active-high enable. The *bufif0* and *bufif1* gates are tri-state buffers that do not invert the output.

For simulation purposes, it is possible to set a parameter of the gate that represents its propagation delay. As an example, the following statement instantiates a three-input AND gate with a delay of five time units (the units of time are determined by the simulator being used):

and #(5) And3 (z, x1, x2, x3);

This type of delay parameter has no meaning when using Verilog for synthesis of logic circuits.

```
// Structural specification of a full-adder
module  fulladd (Cin, x, y, s, Cout);
    input  Cin, x, y;
    output  s, Cout;
    wire  z1, z2, z3, z4;

    and  And1 (z1, x, y);
    and  And2 (z2, x, Cin);
    and  And3 (z3, y, Cin);
    or  Or1 (Cout, z1, z2, z3);
    xor  Xor1 (z4, x, y);
    xor  Xor2 (s, z4, Cin);

endmodule
```

Figure A.4 A full-adder described using gate instantiation.

```
// Structural specification of a full-adder
module fulladd (Cin, x, y, s, Cout);
    input Cin, x, y;
    output s, Cout;

    and (z1, x, y);
    and (z2, x, Cin);
    and (z3, y, Cin);
    or (Cout, z1, z2, z3);
    xor (z4, x, y);
    xor (s, z4, Cin);

endmodule
```

Figure A.5 A simplified version of Figure A.4.

Table A.2 Verilog gates.

Name	Description	Usage
and	$f = (a \cdot b \cdot \cdots)$	**and** $(f, a, b, \ldots)$
nand	$f = \overline{(a \cdot b \cdot \cdots)}$	**nand** $(f, a, b, \ldots)$
or	$f = (a + b + \cdots)$	**or** $(f, a, b, \ldots)$
nor	$f = \overline{(a + b + \cdots)}$	**nor** $(f, a, b, \ldots)$
xor	$f = (a \oplus b \oplus \cdots)$	**xor** $(f, a, b, \ldots)$
xnor	$f = (a \odot b \odot \cdots)$	**xnor** $(f, a, b, \ldots)$
not	$f = \overline{a}$	**not** (f, a)
buf	$f = a$	**buf** (f, a)
notif0	$f = (!e \ ? \ \overline{a} : \text{'}bz)$	**notif0** (f, a, e)
notif1	$f = (e \ ? \ \overline{a} : \text{'}bz)$	**notif1** (f, a, e)
bufif0	$f = (!e \ ? \ a : \text{'}bz)$	**bufif0** (f, a, e)
bufif1	$f = (e \ ? \ a : \text{'}bz)$	**bufif1** (f, a, e)

A.10 CONCURRENT STATEMENTS

In any hardware description language, including Verilog, the concept of a *concurrent statement* means that the code may include a number of such statements, and each represents a part of the circuit. We use the word *concurrent* because the statements are considered in parallel and the ordering of statements in the code does not matter. Gate instantiations are one type of concurrent statements. This section introduces another type of concurrent statement, called the *continuous assignment*.

A.10.1 CONTINUOUS ASSIGNMENTS

While gate instantiations allow the description of a circuit's structure, continuous assignments permit the description of a circuit's function. The general form of this statement is

assign net_assignment{, net_assignment};

The *net_assignment* can be any expression involving the operators listed in Table A.1. Examples of continuous assignments are

```
assign Cout = (x & y) | (x & Cin) | (y & Cin);
assign s = x ^ y ^ z;
```

Although they are not needed in terms of operator precedence, the parentheses in the expression for *Cout* are included for clarity. Multiple assignments can be specified in one **assign** statement, using commas to separate the assignments, as in

```
assign Cout = (x & y) | (x & Cin) | (y & Cin),
       s = x ^ y ^ z;
```

An example of a multibit assignment is

```
wire [1:3] A, B, C;
     ⋮
assign C = A & B;
```

This results in $c_1 = a_1 b_1$, $c_2 = a_2 b_2$, and $c_3 = a_3 b_3$.

The arithmetic assignment

```
wire [3:0] X, Y, S;
     ⋮
assign S = X + Y;
```

represents a four-bit adder without carry-in and carry-out. If we declare a carry-in and carry-out,

wire carryin, carryout;

then the statement

```
module adder_sign (X, Y, S, S2s);
   input  [3:0] X, Y;
   output [7:0] S, S2s;

   assign  S = X + Y,
           S2s = {{4{X[3]}}, X} + {{4{Y[3]}}, Y};

endmodule
```

Figure A.6 An example of arithmetic assignments and sign extension.

```
assign {carryout, S} = X + Y + carryin;
```

represents the four-bit adder with carry-in and carry-out. We mentioned in Section A.7 that Verilog treats the **wire** type as an unsigned number. Since a five-bit result is needed in {carryout, S}, each operand is padded with a zero. When using Verilog for synthesis, it is up to the compiler to determine, or *infer*, that a four-bit adder with carry-out is needed and to recognize the carry-in.

A complete example of arithmetic assignments is given in Figure A.6. There are two four-bit inputs, X and Y, and two eight-bit outputs, S and $S2s$. To produce the eight-bit sum $S = X + Y$ the Verilog compiler automatically pads X and Y with four zeros. The assignment to $S2s$ shows how a signed (2's complement) result can be generated. Recall from Section 3.3 that the leftmost bit in a 2's complement number is the sign bit. The assignment to $S2s$ uses the concatenate and replication operators to pad X and Y with four copies of their most-significant bit, thereby performing sign extensions.

As an example, assume that $X = 0011$ and $Y = 1101$. The unsigned result is $S = 0011 + 1101 = 00010000$, or $S = 3 + 13 = 16$. The signed result is $S2s = 0011 + 1101 = 00000000$, or $S2s = 3 + (-3) = 0$.

A.10.2 USING PARAMETERS

Figure A.6 specifies an adder for four-bit numbers. We can make this code more general by introducing a parameter that sets the number of bits in the adder. Figure A.7 gives the code for an n-bit adder module, *addern*. The number of bits to be added is defined with the **parameter** keyword, introduced in Section A.5. The value of n defines the bit widths of X, Y, S, and $S2s$.

It is possible to combine a continuous assignment with a **wire** declaration. For example, the sum, s, and carry-out, c, of a half-adder could be defined as

```
wire s = x ^ y,
     c = x & y;
```

Verilog allows parameters, such as delays, to be associated with continuous assignments. These parameters have no meaning for synthesis, but we mention them for completeness. Consider the statement

```
module addern (X, Y, S, S2s);
   parameter n = 4;
   input [n−1:0] X, Y;
   output [2*n−1:0] S, S2s;

   assign S = X + Y,
          S2s = {{n{X[n−1]}}, X} + {{n{Y[n−1]}}, Y};

endmodule
```

Figure A.7 Using a parameter in an *n*-bit adder.

```
wire #8 s = x ^ y,
     #5 c = x & y;
```

These assignments specify that the operation $x \wedge y$ has an associated propagation delay of eight time units, and $x \& y$ has a delay of five units. Such delays, which are useful only for simulation purposes, can also be associated with **wires**, as in

```
wire   #2 c;
assign #5 c = x & y;
```

This code specifies that two time units of delay are incurred on the **wire** *c* in addition to the five time units for the AND gate that produces $x \& y$.

A.11 PROCEDURAL STATEMENTS

In addition to the concurrent statements described in the previous section, Verilog also provides *procedural statements* (also called *sequential statements*). Whereas concurrent statements are executed in parallel, procedural statements are evaluated in the order in which they appear in the code. Verilog syntax requires that procedural statements be contained inside an **always** block.

A.11.1 ALWAYS AND INITIAL BLOCKS

An **always** block is a construct that contains one or more procedural statements. It has the form

always @(sensitivity_list)
[begin]
  [procedural assignment statements]
  [**if-else** statements]
  [**case** statements]
  [**while**, **repeat**, and **for** loops]
  [**task** and **function** calls]
[end]

Verilog includes several types of procedural statements. These statements permit the description of a circuit in terms of its *behavior* in a much more powerful way than is possible with continuous assignments or gate instantiations.

When multiple statements are included in an **always** block, the **begin** and **end** keywords are needed; otherwise, these keywords can be omitted. The **begin** and **end** keywords are also used with other Verilog constructs. We refer to the statements delimited by **begin** and **end** as a *begin-end block*.

The *sensitivity_list* is a list of signals that directly affect the output results generated by the **always** block. A simple example of an **always** block is

```
always @(x, y)
begin
  s = x ^ y;
  c = x & y;
end
```

Since the output variables *s* and *c* depend on *x* and *y*, these signals are included in the sensitivity list, separated by a comma or by the keyword **or**. We use the comma in this book, but it should be noted that in the original version of Verilog it was necessary to use the keyword **or**. When specifying a combinational circuit by using an **always** block, it is possible to write simply

```
always @*
```

which indicates that all input signals used in the **always** block are included in the sensitivity list. In the examples in this book we explicitly show the signals in the sensitivity list to make it easier to understand the Verilog code.

The semantics of the **always** block are as follows: if the value of a signal in the sensitivity list changes, then the statements inside the **always** block are evaluated in the order presented.

For simulation purposes, Verilog also provides the *initial* construct. The **initial** and **always** constructs have the same form, but the statements inside the **initial** construct are executed only once, at the start of a simulation. **Initial** constructs are not meaningful for synthesis, hence we do not discuss them further.

A Verilog module may include several **always** blocks, each representing a part of the circuit being modeled. While the statements inside each **always** block are evaluated in order, there is no meaningful order among the different **always** blocks. In this sense, each entire **always** block can be considered as a concurrent statement, because a Verilog compiler evaluates all **always** blocks concurrently.

Procedural Assignment Statements

Any signal assigned a value inside an **always** block has to be a variable of type **reg** or **integer**. A value is assigned to a variable with a *procedural assignment statement*. There are two kinds of assignments: *blocking* assignments, denoted by the = symbol, and *non-blocking* assignments, denoted by the <= symbol. The term *blocking* means that the assignment statement completes and updates its left-hand side before the subsequent statement is evaluated. This concept is best explained in the context of simulation. Consider the blocking assignments

```
S = X + Y;
p = S[0];
```

At simulation time t_i the statements are evaluated, in order. The first statement sets S using the current values of X and Y, and then the second statement sets p according to this new value of S. Verilog also provides *non-blocking* assignments, specified as

```
S <= X + Y;
p <= S[0];
```

In this case, at simulation time t_i the statements are still evaluated in order, but they both use the values of variables that exist at the start of the simulation time t_i. The first statement determines a new value for S based on the current values of X and Y, but S is not actually changed to this value until all statements in the associated **always** block have been evaluated. Therefore, the value of p at time t_i is based on the value of S at time t_{i-1}. We can summarize the difference between blocking and non-blocking assignments as follows. For blocking assignments, the values of variables seen at time t_i by each statement are the new values set in t_i by any preceding statements in the **always** block. For non-blocking assignments, the values of variables seen at time t_i are the values set in time t_{i-1}.

Although we introduced the concepts of blocking and non-blocking assignments in the context of simulation, the semantics are the same for synthesis. For combinational circuits, only blocking assignments should be used, as we will explain in Section A.11.7. We give a number of examples of combinational circuits in the following sections and also introduce the **if-else**, **case**, and loop statements. Section A.14 focuses on sequential circuits and explains that they should be designed with non-blocking assignments.

A.11.2 THE IF-ELSE STATEMENT

The general form of the **if-else** statement is given in Figure A.8. If *expression*1 is True, then the first statement is evaluated. When multiple statements are involved, they have to be included inside a **begin**-**end** block.

The **else if** and **else** clauses are optional. Verilog syntax specifies that when **else if** or **else** are included, they are paired with the most recent unfinished **if** or **else if**.

```
if (expression1)
begin
    statement;
end
else if (expression2)
begin
    statement;
end
else
begin
    statement;
end
```

Figure A.8 The form of the **if-else** statement.

An example of an **if-else** statement used for combinational logic is

```
always @(w0, w1, s)
    if (s == 0)
        f = w0;
    else
        f = w1;
```

which defines a 2-to-1 multiplexer with data inputs w_0 and w_1, select input s, and output f.

A.11.3 STATEMENT ORDERING

Another way of describing the 2-to-1 multiplexer with an **if-else** statement is presented in Figure A.9. Instead of using an **else** clause, this code first makes the default assignment $f = w_0$ and then changes this assignment to $f = w_1$ if s has the value 1. The Verilog semantics specify that a signal assigned multiple values in an **always** construct retains the last assignment. This example highlights the importance of the ordering of statements in an **always** block. If the statements are reversed, as in

```
always @(w0, w1, s)
begin
    if ( s == 1 )
        f = w1;
    f = w0;
end
```

then the **if** statement would be evaluated first and the assignment $f = w_0$ would be performed last. Hence, the code would always result in f being set to the value of w_0.

```
module mux2to1 (w0, w1, s, f);
   input w0, w1, s;
   output reg f;

   always @(w0, w1, s)
   begin
      f = w0;
      if (s == 1)
         f = w1;
   end

endmodule
```

Figure A.9 Code for a 2-to-1 multiplexer.

Implied Memory

Consider the **always** block

```
always @( w0, w1, s )
begin
   if (s == 1)
      f = w1;
end
```

This is the same as the code in Figure A.9, except that the default assignment f = w0; has been removed. Since the code does not specify a value for the variable f when s is 0, the Verilog semantics specify that f must retain its current value. A synthesized circuit has to implement the functionality

$$f = s \cdot w_1 + \bar{s} \cdot f$$

Hence, when $s = 0$, the value of w_1 is "remembered" by a latch at the output f. This effect is called *implied memory*. We will show shortly that implied memory is the key concept used to describe sequential circuits.

A.11.4 THE CASE STATEMENT

The form of a **case** statement is illustrated in Figure A.10. The bits in *expression*, called the *controlling expression*, are checked for a match with each *alternative*. The first successful match causes the associated statements to be evaluated. Each digit in each alternative is compared for an exact match of the four values 0, 1, x, and z. A special case is the **default** clause, which takes effect if no other alternative matches. When using Verilog for simulation, an alternative can be a general expression, but for synthesis these items are

```
case (expression)
   alternative1: begin
                    statement;
                 end
   alternative2: begin
                    statement;
                 end
   [default:     begin
                    statement;
                 end]
endcase
```

Figure A.10 The general form of the **case** statement.

restricted to a single constant, such as 1'b0:, or a list of constants separated by commas, such as 1, 2, 3:.

An example of a **case** statement is

```
always @(w0, w1, s)
   case (s)
      1'b0:  f = w0;
      1'b1:  f = w1;
   endcase
```

This code represents the same 2-to-1 multiplexer described in Section A.11.2 using the **if-else** statement. When using Verilog for simulation, it is necessary to give alternatives for all possible valuations of the controlling expression. A default has to be included for any valuations not explicitly covered by the listed alternatives. In this example, s can have the four values 0, 1, x, or z; hence, we could include a default to handle the cases $s = x$ and $s = z$. We have not included the default clause here because we are concerned only with synthesis, and the synthesis tools require only the bit values 0 and 1 to be considered.

Figure A.11 demonstrates the use of a **case** statement to specify truth tables. This code represents the same full-adder that is described in Figures A.2 and A.3. The controlling expression in the **case** statement is the concatenated bits $\{Cin, x, y\}$, and the alternatives correspond to the rows of the truth table in Figure 3.3*a*.

The **case** statement is also important for representing some types of sequential circuits, such as finite state machines, which are discussed in Section A.14.

A.11.5 CASEZ AND CASEX STATEMENTS

In the **case** statement, the values x or z in an alternative are checked for an exact match with the same values in the controlling expression. The **casez** statement adds more flexibility, by treating a z digit in an alternative as a don't-care condition. The **casex** statement treats

```
// Full adder
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output reg s, Cout;

   always @(Cin, x, y)
   begin
      case ( {Cin, x, y} )
         3'b000: {Cout, s} = 'b00;
         3'b001: {Cout, s} = 'b01;
         3'b010: {Cout, s} = 'b01;
         3'b011: {Cout, s} = 'b10;
         3'b100: {Cout, s} = 'b01;
         3'b101: {Cout, s} = 'b10;
         3'b110: {Cout, s} = 'b10;
         3'b111: {Cout, s} = 'b11;
      endcase
   end

endmodule
```

Figure A.11 Using a **case** statement to specify a truth table.

both x and z as don't cares. The alternatives do not have to be mutually exclusive. If they are not, then the first matching item has priority. Figure A.12 shows how **casex** can be used to describe a priority encoder with the 4-bit input W and the outputs Y and f. The priority encoder is defined in Figure 4.20 (the output z in Figure 4.20 is named f in the Verilog code in Figure A.12). The first alternative, 1xxx, specifies that if w_3 has the value 1, then the other inputs are treated as don't cares, hence the output is set to $Y = 3$. Similarly, the other alternatives describe the desired priority scheme.

A.11.6 LOOP STATEMENTS

Verilog includes four types of loop statements: **for**, **while**, **repeat**, and **forever**. Synthesis tools typically support the **for** loop, which has the general form

```
for (initial_index; terminal_index; increment)
begin
   statement;
end
```

This syntax is very similar to the **for** loop in the C programming language. The *initial_index* is evaluated once, before the first loop iteration, and typically performs the initialization of the **integer** loop control variable, such as $k = 0$. In each loop iteration, the **begin-end**

```
module priority (W, Y, f);
   input [3:0] W;
   output reg [1:0] Y;
   output f;

   assign f = (W != 0);
   always @(W)
   begin
      casex (W)
         'b1xxx: Y = 3;
         'b01xx: Y = 2;
         'b001x: Y = 1;
         default: Y = 0;
      endcase
   end

endmodule
```

Figure A.12 A priority encoder described using a **casex** statement.

block is performed, and then the *increment* statement is evaluated. A typical increment statement is $k = k + 1$. Finally, the *terminal_index* condition is checked, and if it is True (1), then another loop iteration is done. For synthesis, the terminal_index condition has to compare the loop index to a constant value, such as $k < 8$.

An example of using a **for** loop to describe an n-bit ripple-carry adder is presented in Figure A.13. The effect of the loop is to repeat its **begin**-**end** block for the specified values of k. In this example, each loop iteration, k, defines a full-adder with the inputs x_k, y_k, and c_k, and the outputs s_k and c_{k+1}. It is possible to define the **integer** k (parameters can also be defined in this way) inside the **always** block if the **begin**-**end** block has a label. For example,

```
always @(X, Y, carryin)
begin: fulladders
   integer k;
   C[0] = carryin;
   for (k = 0; k <= n-1; k = k+1)
   begin
      S[k] = X[k] ^ Y[k] ^ C[k];
      C[k+1] = (X[k] & Y[k]) | (C[k] & X[k]) | (C[k] & Y[k]);
   end
   carryout = C[n];
end
```

```
module ripple (carryin, X, Y, S, carryout);
   parameter n = 4;
   input carryin;
   input [n–1:0] X, Y;
   output reg [n–1:0] S;
   output reg carryout;
   reg [n:0] C;
   integer k;

   always @(X, Y, carryin)
   begin
      C[0] = carryin;
      for (k = 0; k <= n–1; k = k+1)
      begin
         S[k] = X[k] ^ Y[k] ^ C[k];
         C[k+1] = (X[k] & Y[k]) | (C[k] & X[k]) | (C[k] & Y[k]);
      end
      carryout = C[n];
   end

endmodule
```

Figure A.13 An n-bit ripple-carry adder using a **for** loop.

A second **for**-loop example is given in Figure A.14. This code produces a count of the number of bits in the n-bit input X that have the value 1. Unrolling the loop, the first two iterations give

$$\begin{aligned} &\text{Count} = \text{Count} + \text{X[0]};\\ &\text{Count} = \text{Count} + \text{X[1]}; \end{aligned}$$

The first statement produces the value $Count = 0 + X[0] = X[0]$. The second assignment then gives $Count = X[0] + X[1]$, and so on for the other loop iterations. At the end of the loop, we have

$$\text{Count} = \text{X[0]} + \text{X[1]} + \cdots + \text{X[n–1]}$$

A synthesis tool generates a circuit that has a cascade of adders to implement this expression. For example, for $n = 3$ a possible circuit that employs two-bit adders is shown in Figure A.15.

Figure A.16 shows the general forms of the **while** and **repeat** loops. The **while** loop has the same structure as the corresponding statement in the C language, and the **repeat** loop simply specifies a number of times to repeat its **begin**-**end** block. The **forever** loop, not shown in the figure, loops endlessly.

```
module bit_count (X, Count);
    parameter n = 4;
    parameter logn = 2;
    input [n–1:0] X;
    output reg [logn:0] Count;
    integer k;

    always @(X)
    begin
        Count = 0;
        for (k = 0; k < n; k = k+1)
            Count = Count + X[k];
    end

endmodule
```

Figure A.14 A bit-counting example.

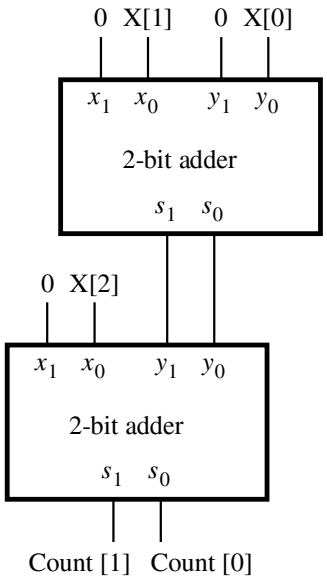

Figure A.15 A circuit that implements the code in Figure A.14.

```
while (condition)
begin
   statement;
end

⋮

repeat (constant_value)
begin
   statement;
end
```

Figure A.16 The general forms of the **while** and **repeat** statements.

A.11.7 BLOCKING VERSUS NON-BLOCKING ASSIGNMENTS FOR COMBINATIONAL CIRCUITS

All our previous examples of combinational circuits used blocking assignments, which is a good way to design such circuits. A natural question is whether combinational circuits can be described using non-blocking assignments. The answer is that this would work in many cases, but if subsequent assignments depend on the results of preceding assignments, non-blocking assignments can produce nonsensical combinational circuits. As an example, consider changing the **for** loop in Figure A.14 to use non-blocking assignments, as indicated in Figure A.17. For simplicity assume that $n = 3$, so that the unrolled loop is

```
Count <= Count + X[0];
Count <= Count + X[1];
Count <= Count + X[2];
```

Since non-blocking assignments are involved, each subsequent assignment statement sees the starting value of *Count* = 0 rather than a new *Count* value produced by the previous statements. The **for** loop thus degenerates to

```
Count <= 0 + X[0];
Count <= 0 + X[1];
Count <= 0 + X[2];
```

When there are multiple assignments to the same variable in an **always** block, Verilog semantics specify that the variable retains its last assignment. Therefore, the code produces the wrong result $Count = X[2]$.

```
always @(X)
begin
    Count = 0;
    for (k = 0; k < n; k = k+1)
        Count <= Count + X[k];
end
```

Figure A.17 Using non-blocking assignments for a combinational circuit.

A.12 USING SUBCIRCUITS

A Verilog module can be included as a subcircuit in another module. For this to work, both modules must be defined in the same file or else the Verilog compiler must be told where each module is located (the mechanism for doing this varies from one compiler to the next). The general form of a *module instantiation* statement is similar to a gate instantiation statement

```
module_name [#(parameter overrides)] instance_name (
            .port_name ( [expression] ) {, .port_name ( [expression] )} );
```

The *instance_name* can be any legal Verilog identifier and the port connections specify how the module is connected to the rest of the circuit. The same module can be instantiated multiple times in a given design provided that each instance name is unique. The #(parameter overrides) can be used to set the values of parameters defined inside the *module_name* module. We discuss this feature in the next section. Each *port_name* is the name of a port in the subcircuit, and each *expression* specifies a connection to that port. The syntax *.port_name* is provided so that the order of signals listed in the instantiation statement does not have to be the same as the order of the ports given in the **module** statement of the subcircuit. In Verilog jargon, this is called *named* port connections. If the port connections are given in the same order as in the subcircuit, then *.port_name* is not needed. This format is called *ordered* port connections.

An example is presented in Figure A.18. It gives the code for a four-bit ripple-carry adder built using four instances of the *fulladd* subcircuit in Figure A.2. The inputs to the adder are *carryin* and two four-bit numbers X and Y. The output is the four-bit sum, S, and *carryout*. A three-bit signal, C, represents the carries from stages 0, 1, and 2. This signal is declared as a **wire** vector with three bits.

The *adder4* module instantiates four copies of the full-adder subcircuit. In the first three instantiation statements, we use ordered port connections because the signals are listed in the same order as given in the declaration of the *fulladd* module in Figure A.2. The last instantiation statement gives an example of named port connections. The port connections used in the instantiation statements specify how the *fulladd* instances are interconnected by nets to create the adder module.

Figure A.19 gives an example of a hierarchical Verilog file containing two modules. The bottom module, *seg*7, specifies the BCD-to-7-segment code converter shown in Figure 2.63.

```
module adder4 (carryin, X, Y, S, carryout);
   input carryin;
   input [3:0] X, Y;
   output [3:0] S;
   output carryout;
   wire [3:1] C;

   fulladd stage0 (carryin, X[0], Y[0], S[0], C[1]);
   fulladd stage1 (C[1], X[1], Y[1], S[1], C[2]);
   fulladd stage2 (C[2], X[2], Y[2], S[2], C[3]);
   fulladd stage3 (.Cout(carryout), .s(S[3]), .y(Y[3]), .x(X[3]), .Cin(C[3]));

endmodule
```

Figure A.18 An example of module instantiation.

It has the four-bit *bcd* input, which represents a binary-coded-decimal digit, and the seven-bit *leds* output, which is intended to drive the seven segments *a* to *g* on a digit-oriented display. Three copies of the seven-segment decoder are instantiated in the top module, *group*. It has a 12-bit input, *Digits*, and a 21-bit output, *Lights*, that are connected to the three instantiated subcircuits.

A.12.1 SUBCIRCUIT PARAMETERS

When a subcircuit includes parameters, their default values can be changed in an instantiation statement. Figure A.20 gives a module with two eight-bit inputs, X and Y, and a four-bit output, C. The module determines the number of bits in X and Y that are identical. The code first uses the bitwise XNOR operation to generate a signal T that has 1s in the bit positions where X and Y have identical values. Then it instantiates the subcircuit specified in Figure A.14 to count the number of 1s in T. The syntax #(8,3) overrides the default values of the subcircuit parameters in the order given in the code. Since Figure A.14 first defines the parameter n, followed by *logn*, then #(8,3) sets $n = 8$ and $logn = 3$. If only one parameter is specified, such as #(8), it overrides the first parameter given in the subcircuit. The parameters can also be referred to by name, as in #(.n(8), .logn(3)).

An alternative syntax for overriding parameter values is shown in Figure A.21. Here, the subcircuit parameters are specified by the separate statement

```
defparam cbits.n = 8, cbits.logn = 3;
```

This statement is not a part of the instantiation statement; hence, it can appear anywhere in the code. The intended subcircuit is identified uniquely by its instance name, *cbits*. If the **defparam** statement is in a separate module from the corresponding instantiation statement, the subcircuit module name should be specified in addition to the instance name. An example is

```
defparam bit_count.cbits.n = 8, bit_count.cbits.logn = 3;
```

```
module group (Digits, Lights);
   input [11:0] Digits;
   output [1:21] Lights;

   seg7 digit0 (Digits[3:0], Lights[1:7]);
   seg7 digit1 (Digits[7:4], Lights[8:14]);
   seg7 digit2 (Digits[11:8], Lights[15:21]);

endmodule

module seg7(bcd, leds);
   input [3:0] bcd;
   output reg [1:7] leds;

   always @(bcd)
      case (bcd)   //abcdefg
         0: leds = 7'b1111110;
         1: leds = 7'b0110000;
         2: leds = 7'b1101101;
         3: leds = 7'b1111001;
         4: leds = 7'b0110011;
         5: leds = 7'b1011011;
         6: leds = 7'b1011111;
         7: leds = 7'b1110000;
         8: leds = 7'b1111111;
         9: leds = 7'b1111011;
         default: leds = 7'bx;
      endcase

endmodule
```

Figure A.19 An example of a hierarchical design file.

```
module common (X, Y, C);
   input [7:0] X, Y;
   output [3:0] C;
   wire [7:0] T;

   // Make T[i] = 1 if X[i] == Y[i]
   assign T = X ~^ Y;

   bit_count #(8,3) cbits (T, C);

endmodule
```

Figure A.20 Overriding module parameters using #.

```
module common (X, Y, C);
   input [7:0] X, Y;
   output [3:0] C;
   wire [7:0] T;

   // Make T[i] = 1 if X[i] == Y[i]
   assign T = X ~^ Y;

   bit_count cbits (T, C);
      defparam cbits.n = 8, cbits.logn = 3;

endmodule
```

Figure A.21 Overriding module parameters using the **defparam** statement.

A.12.2 THE GENERATE CAPABILITY

Figure A.18 instantiates four copies of the *fulladd* subcircuit to form a four-bit ripple-carry adder. A natural extension of this code is to add a parameter that sets the number of bits needed and then use a loop to instantiate the required subcircuits. This can be achieved with the **generate** construct.

The **generate** construct has the simplified form

```
generate
   [for loops]
   [if-else statements]
   [case statements]
   [instantiation statements]
endgenerate
```

This construct enhances the flexibility of Verilog modules, because it allows instantiation statements to be included inside **for** loops and **if-else** statements. If a **for** loop is included in the **generate** block, the loop index variable has to be declared of type **genvar**. A **genvar** variable is similar to an **integer** variable, but it can have only positive values and it can be used only inside **generate** blocks.

Figure A.22 shows the *ripple_g* module, which instantiates *n fulladd* modules. Each instance generated in the **for** loop will have a unique instance name produced by the compiler based on the **for** loop label. The generated names are *addbit*[0].*stage*, . . . , *addbit*[$n-1$].*stage*. This code produces the same result as the code in Figure A.13.

The **generate** construct can include concurrent statements and procedural statements, but its main advantage is in the instantiation of gates and modules inside **for** loops and **if-else** statements.

```
module ripple_g (carryin, X, Y, S, carryout);
   parameter n = 4;
   input carryin;
   input [n–1:0] X, Y;
   output [n–1:0] S;
   output carryout;
   wire [n:0] C;

   genvar i;
   assign C[0] = carryin;
   assign carryout = C[n];

   generate
      for (i = 0; i <= n–1; i = i+1)
      begin:addbit
         fulladd stage (C[i], X[i], Y[i], S[i], C[i+1]);
      end
   endgenerate

endmodule
```

Figure A.22 An *n*-bit ripple-carry adder using the **generate** statement.

A.13 FUNCTIONS AND TASKS

Figure 4.4 shows how a 16-to-1 multiplexer module can be created by instantiating five 4-to-1 multiplexers. Another way of modeling this circuit is to use a Verilog *function* which has the general form

```
function [range | integer] function_name;
   [input declarations]
   [parameter, reg, integer declarations]
   begin
      statement;
   end
endfunction
```

The purpose of a **function** is to allow the code to be written in a modular fashion without defining separate modules. A **function** is defined within a module, and it is called either in a continuous assignment statement or in a procedural assignment statement inside that module. A **function** can have more than one input, but it does not have an output, because the **function** name itself serves as the output variable. Figure A.23 shows how the

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

   function mux4to1;
      input [0:3] W;
      input [1:0] S;

      if (S == 0) mux4to1 = W[0];
      else if (S == 1) mux4to1 = W[1];
      else if (S == 2) mux4to1 = W[2];
      else if (S == 3) mux4to1 = W[3];
   endfunction

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

endmodule
```

Figure A.23 An example of a function.

16-to-1 multiplexer module can be written using a function. To see how this code works, consider the assignment

```
f = mux4to1(M[0:3], S16[3:2]);
```

The effect of the **function** call is to have the Verilog compiler replace the assignment statement with the **function** body. The in-line equivalent of the above **function** call is

```
if (S16[3:2] == 0)  f = M[0];
else if (S16[3:2] == 1)  f = M[1];
else if (S16[3:2] == 2)  f = M[2];
else if (S16[3:2] == 3)  f = M[3];
```

Similarly, each **function** call is inserted in-line, with the substitution of appropriate bits from *S16*, *W*, and *M*.

A second example of a **function** is shown in Figure A.24. The *group_ f* module is equivalent to the *group* module in Figure A.19. Where the *group* module instantiates three copies of the *seg*7 subcircuit, *group_ f* achieves the same effect by using a **function**. Since it returns a seven-bit value, the **function** is defined with the syntax

function [1:7] leds;

Consider again the 16-to-1 multiplexer example in Figure A.23. Another method of writing this code appears in Figure A.25. This code uses a Verilog *task*, which is similar to a **function**. While a **function** returns a value, a **task** does not; it has input and output variables, like a module. A **task** can be called only from inside an **always** (or **initial**) block.

```
module group_f (Digits, Lights);
   input [11:0] Digits;
   output reg [1:21] Lights;

   function [1:7] leds;
      input [3:0] bcd;
      begin
         case (bcd)   //  abcdefg
            0: leds = 7'b1111110;
            1: leds = 7'b0110000;
            2: leds = 7'b1101101;
            3: leds = 7'b1111001;
            4: leds = 7'b0110011;
            5: leds = 7'b1011011;
            6: leds = 7'b1011111;
            7: leds = 7'b1110000;
            8: leds = 7'b1111111;
            9: leds = 7'b1111011;
            default: leds = 7'bx;
         endcase
      end
   endfunction

   always @(Digits)
   begin
      Lights[1:7] = leds(Digits[3:0]);
      Lights[8:14] = leds(Digits[7:4]);
      Lights[15:21] = leds(Digits[11:8]);
   end

endmodule
```

Figure A.24 Using a function to implement the *group* module.

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

    task mux4to1;
        input [0:3] W;
        input [1:0] S;
        output Result;
        begin
            if (S == 0) Result = W[0];
            else if (S == 1) Result = W[1];
            else if (S == 2) Result = W[2];
            else if (S == 3) Result = W[3];
        end
    endtask

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

endmodule
```

Figure A.25 An example of a **task**.

In a similar way as described above for the Verilog **function**, the compiler essentially inserts the code for the **task** body at the point in the code where it is called.

Functions and **tasks** are not crucial for designing Verilog code, but they facilitate the writing of modular code without using separate modules. One advantage of **functions** and **tasks** is that they can be called from an **always** block, whereas these blocks are not allowed to contain instantiation statements. These features of Verilog become increasingly important as the size of the code being developed increases.

A.14 SEQUENTIAL CIRCUITS

While combinational circuits can be modeled with either continuous assignment or procedural assignment statements, sequential circuits can be described only with procedural statements. We now give some examples of sequential circuits.

A.14.1 A GATED D LATCH

Figure A.26 gives the code for a gated D latch. The sensitivity list for the **always** block includes both the data input, *D*, and clock, *clk*. The **if** statement specifies that Q should be set to the value of *D* whenever *clk* is 1. There is no **else** clause in the **if** statement. As we explained in Section A.11.3, this situation implies that Q should retain its present value when the **if** condition is not met.

A.14.2 D FLIP-FLOP

Figure A.27 shows how flip-flops are described in Verilog. The **always** construct uses the special sensitivity list @(**posedge** Clock). This event expression tells the Verilog compiler that any **reg** variable assigned a value in the **always** construct is the output of a D flip-flop. The code in the figure generates a flip-flop with the input *D* and the output Q that is sensitive to the positive clock edge. A negative-edge sensitive flip-flop is specified by @(**negedge** Clock).

We said in Section A.11.1 that sequential circuits should be described with non-blocking assignments, and we used this type of assignment in Figure A.27. The behavior of blocking and non-blocking assignments for sequential circuits is discussed in Section A.14.5.

```
module latch (D, clk, Q);
   input D, clk;
   output reg Q;

   always @(D, clk)
      if (clk)
         Q = D;

endmodule
```

Figure A.26 A gated D latch.

```
module flipflop (D, Clock, Q);
   input D, Clock;
   output reg Q;

   always @(posedge Clock)
      Q <= D;

endmodule
```

Figure A.27 A D flip-flop.

A.14.3 FLIP-FLOPS WITH RESET

Figure A.28 gives an **always** block that is similar to the one in Figure A.27. It describes a D flip-flop with an asynchronous reset (clear) input, *Resetn*. When *Resetn* = 0, the flip-flop output Q is set to 0. Appending the letter *n* to a signal name is a popular convention to denote an active-low signal.

Verilog syntax requires that a sensitivity list contains either all edge-sensitive events or all level-sensitive events but not a mixture; hence, the reset condition is checked using **negedge** Resetn. An active-high reset would require the event **posedge** Reset and the **if-else** statement would then check for the condition (Reset == 1).

In general, Verilog offers a variety of ways to describe a given circuit. But, for specifying flip-flops, the format of the code is quite strict. Only minor variations of the code in Figures A.27 and A.28 can be made and still infer the desired flip-flops. For instance, the **if-else** statement could alternatively specify **if** (!Resetn), but this has to be the first statement in the **always** block. Note that nothing is special about the variable name *Clock*; the keyword **posedge** and the format of the rest of the **always** block are what allow the Verilog compiler to recognize the flip-flop clock signal.

Figure A.29 shows how a flip-flop with a synchronous reset input can be described. Since only the **posedge** Clock event appears in the sensitivity list of the **always** block, the reset operation has to be synchronized to the clock edge.

A.14.4 REGISTERS

One possible approach for describing a multibit register is to create an entity that instantiates multiple flip-flops. A more convenient method is illustrated in Figure A.30. It gives the same code shown in Figure A.28 but using the four-bit input *D* and the four-bit output Q. The code describes a four-bit register with asynchronous clear.

Figure A.31 shows how the code in Figure A.30 can be extended to represent an *n*-bit register with an enable input, *E*. The number of flip-flops is set by the parameter *n*. When the active clock edge occurs, the flip-flops in the register cannot change their stored values if the enable *E* is 0. If *E* = 1, the register responds to the active clock edge in the normal way.

```
module  flipflop_ar (D, Clock, Resetn, Q);
   input  D, Clock, Resetn;
   output  reg  Q;

   always @(posedge Clock, negedge Resetn)
      if (Resetn == 0)
         Q <= 0;
      else
         Q <= D;

endmodule
```

Figure A.28 A D flip-flop with asynchronous reset.

```
module flipflop_sr (D, Clock, Resetn, Q);
    input D, Clock, Resetn;
    output reg Q;

    always @(posedge Clock)
        if (Resetn == 0)
            Q <= 0;
        else
            Q <= D;

endmodule
```

Figure A.29 A D flip-flop with synchronous reset.

```
module reg4 (D, Clock, Resetn, Q);
    input [3:0] D;
    input Clock, Resetn;
    output reg [3:0] Q;

    always @(posedge Clock, negedge Resetn)
    if (Resetn == 0)
        Q <= 4'b0000;
    else
        Q <= D;

endmodule
```

Figure A.30 A four-bit register with asynchronous clear.

```
module regne (D, Clock, Resetn, E, Q);
    parameter n = 4;
    input [n–1:0] D;
    input Clock, Resetn, E;
    output reg [n–1:0] Q;

    always @(posedge Clock, negedge Resetn)
        if (Resetn == 0)
            Q <= 0;
        else if (E)
            Q <= D;

endmodule
```

Figure A.31 An *n*-bit register with asynchronous clear and enable.

A.14.5 SHIFT REGISTERS

An example of code that defines a three-bit shift register is provided in Figure A.32. The lines of code are numbered for ease of reference. The shift register has a serial input, *w*, and parallel outputs, Q. The right-most bit in the register is Q[3], and the left-most bit is Q[1]. Shifting is performed in the right-to-left direction. All assignments to Q are synchronized to the clock edge by the (**posedge** Clock) event, hence Q represents the outputs of flip-flops. The statement in line 6 specifies that Q[3] is assigned the value of *w*. The semantics of the non-blocking assignments mean that the subsequent statements do not see the new value of Q[3] until the next time the **always** block is evaluated (in the following clock cycle). In line 7, the current value of Q[3], before it is shifted as a result of line 6, is assigned to Q[2]. Line 8 completes the shift operation by assigning the current value of Q[2] to Q[1], before it is changed as a result of line 7.

The key point that has to be appreciated in the code in Figure A.32 is that the assignments in lines 6 to 8 do not take effect until the end of the **always** block. Hence, all flip-flops change their values at the same time, as required in the shift register. We could write the statements in lines 6 to 8 in any order without changing the meaning of the code.

Blocking Assignments for Sequential Circuits

We said previously that blocking assignments should not be used for sequential circuits. As an example of the semantics involved, Figure A.33 gives the **always** block from Figure A.32 with blocking assignments. The first assignment sets Q[3] = *w*. Since blocking assignments are involved, the next statement sees this new value of Q[3]; hence, it produces Q[2] = Q[3] = *w*. Similarly, the final assignment gives Q[1] = Q[2] = *w*. The code does not describe the desired shift register, but rather loads all flip-flops with the value on the input *w*.

For the code in Figure A.33 to correctly describe a shift register, the ordering of the three assignments has to be reversed. Then the first assignment sets Q[1] to the value of Q[2], the second sets Q[2] to the value of Q[3], and so on. Each successive assignment is not affected by the one that precedes it; hence, the semantics of blocking assignments cause no problem.

```
1   module shift3 (w, Clock, Q);
2      input w, Clock;
3      output reg [1:3] Q;

4      always @(posedge Clock)
5      begin
6         Q[3] <= w;
7         Q[2] <= Q[3];
8         Q[1] <= Q[2];
9      end

10  endmodule
```

Figure A.32 A three-bit shift register.

```
module shift3 (w, Clock, Q);
    input w, Clock;
    output reg [1:3] Q;

    always @(posedge Clock)
    begin
        Q[3] = w;
        Q[2] = Q[3];
        Q[1] = Q[2];
    end

endmodule
```

Figure A.33 Wrong code for a three-bit shift register.

```
module count4 (Clock, Resetn, E, Q);
    input Clock, Resetn, E;
    output reg [3:0] Q;

    always @(posedge Clock, negedge Resetn)
        if (Resetn == 0)
            Q <= 0;
        else if (E)
            Q <= Q + 1;

endmodule
```

Figure A.34 Code for a four-bit counter.

To avoid the confusing dependence on the ordering of statements, blocking assignments should be avoided when modeling sequential circuits. Also, because they imply differing semantics, blocking and non-blocking assignments should never be mixed in a single **always** construct.

A.14.6 COUNTERS

Figure A.34 presents the code for a four-bit counter with an asynchronous reset input. The counter also has an enable input, E. On the positive clock edge, if E is 1, the count is incremented. If $E = 0$, the counter holds its current value.

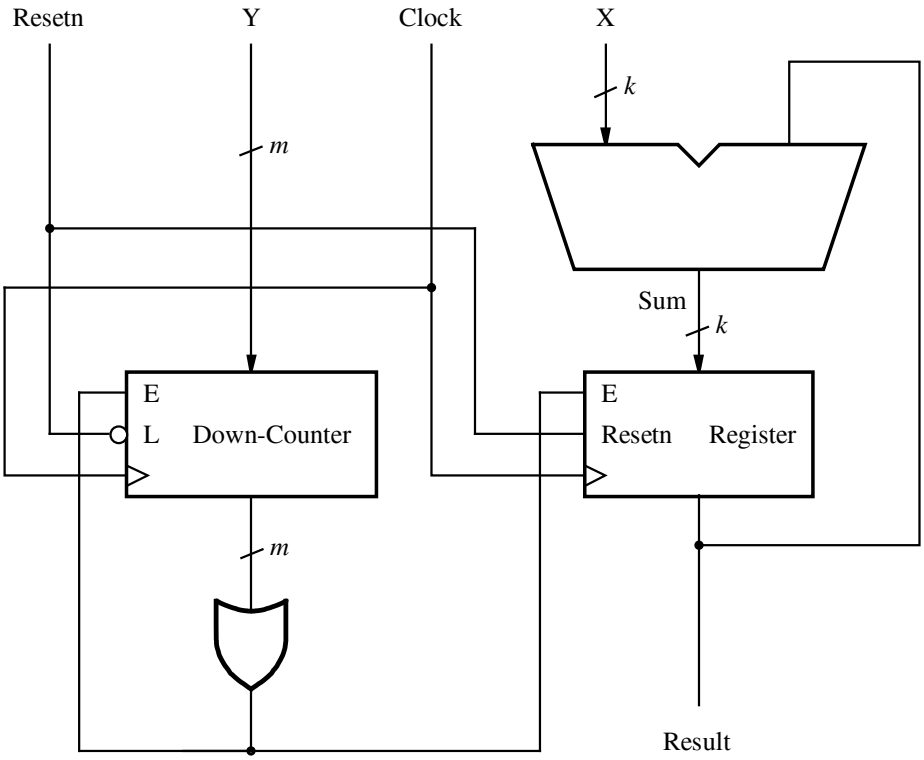

Figure A.35 The accumulator circuit.

A.14.7 AN EXAMPLE OF A SEQUENTIAL CIRCUIT

An example of a sequential circuit is given in Figure A.35. It adds together the values of the k-bit input X over successive clock cycles, and stores the sum of these values into a k-bit register. Such a circuit is often called an *accumulator*. To store the result of each addition operation, the circuit includes a k-bit register with an asynchronous reset input, *Resetn*. It also has an enable input, E, which is controlled by a down-counter. The down-counter has an asynchronous load input and a count enable input. The circuit is operated by first setting *Resetn* to 0, which resets the contents of the k-bit register to 0 and loads the down-counter with an m-bit number on the Y input.

Then, in each clock cycle, the counter is decremented, and the sum outputs from the adder are loaded into the register. When the counter reaches 0, the enable inputs on both the register and counter are set to 0 by the OR gate. The circuit remains in this state until it is reset again. The final value stored in the register is the sum of the values of X in each of the Y clock cycles.

We designed the accumulator circuit by using two subcircuits described in this appendix: *ripple* (Figure A.13) and *regne* (Figure A.31). The complete code is given in Figure A.36. It uses the parameter k to set the number of bits in the input X, and parameter m to specify the number of bits in the counter. Using these parameters in the code makes

```
module accum (X, Y, Clock, Resetn, Result);
    parameter k = 8;
    parameter m = 4;
    input [k–1:0] X;
    input [m–1:0] Y;
    input Clock, Resetn;
    output [k–1:0] Result;
    wire [k–1:0] Sum;
    wire Cout, Go;
    reg [m–1:0] C;

    ripple u1 (.carryin(0), .X(X), .Y(Result), .S(Sum), .carryout(Cout));
        defparam u1.n = k;
    regne u2 (.D(Sum), .Clock(Clock), .Resetn(Resetn), .E(Go), .Q(Result));
        defparam u2.n = 8;

    always @(posedge Clock, negedge Resetn)
        if (Resetn == 0)
            C <= Y;
        else if (Go)
            C <= C – 1;

    assign Go = | C;

endmodule
```

Figure A.36 Code for a *k*-bit accumulator circuit.

it easy to change the bit width at a later time if desired. The code defines the signal *Sum* to represent the outputs of the adder; for simplicity, we ignore the possibility of arithmetic overflow and assume that the sum will fit into k bits. The m-bit signal C represents the outputs from the down-counter. The *Go* signal is connected to the enable inputs on the register and counter.

A.14.8 MOORE-TYPE FINITE STATE MACHINES

Figure A.37 gives the state diagram of a simple Moore machine. Verilog code for this machine is shown in Figure A.38. The two-bit vector y represents the present state of the machine, and the state codes are defined as parameters. Some CAD synthesis systems provide a means of requesting that the state assignment be chosen automatically, but we have specified the assignment manually in this example.

The present state signal y corresponds to the outputs of the state flip-flops, and the signal Y represents the inputs to the flip-flops, which define the next state. The code has two

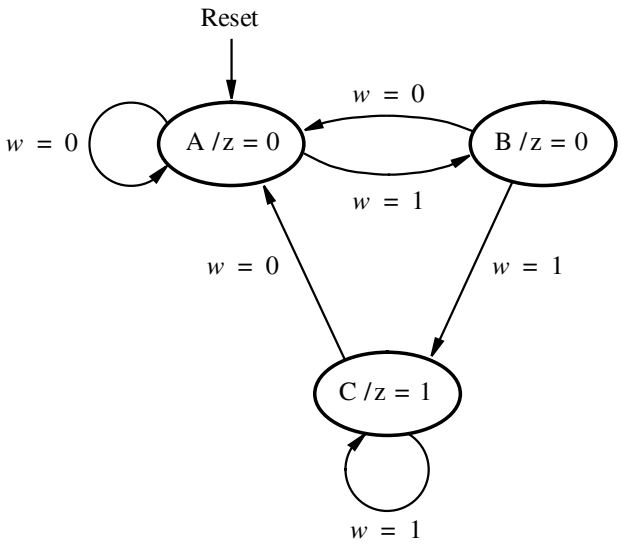

Figure A.37 State diagram of a simple Moore-type FSM.

always blocks. The top one describes a combinational circuit and uses a **case** statement to specify the values that *Y* should have for each value of *y*. The other **always** block represents a sequential circuit, which specifies that *y* is assigned the value of *Y* on the positive clock edge. The **always** block also specifies that *y* should take the value *A* when *Resetn* is 0, which provides the asynchronous reset.

Since the machine is of the Moore type, the output *z* can be defined using the assignment statement z = (y == C) that depends only on the present state of the machine. This statement is provided as a continuous assignment at the end of the code, but it could alternatively have been given inside the top **always** block that represents the combinational part of the FSM. This assignment statement cannot be placed inside the bottom **always** block. Doing so would cause *z* to be the output of a separate flip-flip, rather than a combinational function of *y*. This circuit would set *z* to 1 one clock cycle later than required when the machine enters state *C*.

An alternative version of the code for the Moore machine is given in Figure A.39. This code uses a single **always** block to define both the combinational and sequential parts of the finite state machine. In practice, the code in Figure A.38 is used more commonly.

A.14.9 MEALY-TYPE FINITE STATE MACHINES

A state diagram for a simple Mealy machine is shown in Figure A.40, and the corresponding code is given in Figure A.41. The code has the same structure as in Figure A.38 except that the output *z* is defined within the top **always** block. The **case** statement specifies that, when the FSM is in state *A*, *z* should be 0, but when in state *B*, *z* should take the value of *w*. Since the top **always** block represents a combinational circuit, the output *z* can change value as soon as the input *w* changes, as required for the Mealy machine.

```
module moore (Clock, w, Resetn, z);
    input Clock, w, Resetn;
    output z;
    reg [1:0] y, Y;
    parameter A = 2'b00, B = 2'b01, C = 2'b10;

    always @(w, y)
    begin
        case (y)
            A:  if (w == 0)  Y = A;
                else         Y = B;
            B:  if (w == 0)  Y = A;
                else         Y = C;
            C:  if (w == 0)  Y = A;
                else         Y = C;
            default:         Y = 2'bxx;
        endcase
    end

    always @(posedge Clock, negedge Resetn)
    begin
        if (Resetn == 0)
            y <= A;
        else
            y <= Y;
    end

    assign z = (y == C);

endmodule
```

Figure A.38 Code for a Moore machine.

A.15 GUIDELINES FOR WRITING VERILOG CODE

Modern digital systems are large and complex. A good approach in designing such systems is to decompose them into smaller, manageable parts. Each part can then be designed by making use of the popular subcircuit building blocks that we describe in this book.

When using Verilog for synthesis, it is essential to write code such that the compiler will generate the intended circuit. For example, code for combinational circuits like adders, multiplexers, encoders, and decoders should be written as illustrated in Chapters 3 and 4. Flip-flops, registers, and counters should be described with the style of code presented

```
module moore (Clock, w, Resetn, z);
   input Clock, w, Resetn;
   output z;
   reg [1:0] y;
   parameter A = 2'b00, B = 2'b01, C = 2'b10;

   always @(posedge Clock, negedge Resetn)
   begin
      if (Resetn == 0)
         y <= A;
      else
         case (y)
            A:   if (w == 0)  y <= A;
                 else         y <= B;
            B:   if (w == 0)  y <= A;
                 else         y <= C;
            C:   if (w == 0)  y <= A;
                 else         y <= C;
            default:          y <= 2'bxx;
         endcase
   end

   assign z = (y == C);

endmodule
```

Figure A.39 Alternative version of the code for a Moore machine.

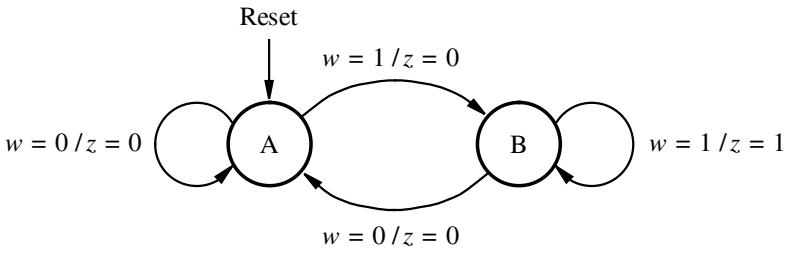

Figure A.40 State diagram of a Mealy-type FSM.

in Chapter 5; and finite-state machine code should be expressed in the manner shown in Chapter 6. Chapter 7 gives some examples of how larger circuits can be constructed by employing these common building blocks. This approach of writing HDL code such that the system is built by interconnecting common, relatively simple subcircuits is often referred to as the *register-transfer level* (RTL) style of code. It is the most popular design method used in practice. The rest of this section lists some common errors found in Verilog code and gives some useful guidelines.

```
module mealy (Clock, w, Resetn, z);
    input Clock, w, Resetn;
    output reg z;
    reg y, Y;
    parameter A = 1'b0, B = 1'b1;

    always @(w, y)
        case (y)
            A: if (w == 0)
                begin
                    Y = A;
                    z = 0;
                end
                else
                begin
                    Y = B;
                    z = 0;
                end
            B: if (w == 0)
                begin
                    Y = A;
                    z = 0;
                end
                else
                begin
                    Y = B;
                    z = 1;
                end
        endcase

    always @(posedge Clock, negedge Resetn)
        if (Resetn == 0)
            y <= A;
        else
            y <= Y;

endmodule
```

Figure A.41 Code for a Mealy machine.

Missing Begin-End

The **always** construct requires **begin** and **end** delimiters if there are multiple statements in the block. The compiler does not take indentation into consideration! For example the **always** block

```
always @(w0, w1, s)
   if ( s == 1 )
      f = w1;
   f = w0;
```

has one statement in it, not two.

Missing Semicolon

Every Verilog statement must end with a semicolon.

Missing { }

The replication operator requires a lot of brackets. A common mistake is to write {3{A}, 2{B}} instead of {{3{A}}, {2{B}}}.

Accidental Assignment

The statement

```
always @(w0, w1, s)
begin
   if ( s = 1 )
      f = w1;
   f = w0;
end
```

does not test the value of *s*. It sets *s* to the value 1.

Incomplete Sensitivity List

Consider the **always** block

```
always @(x)
begin
   s = x ^ y;
   c = x & y;
end
```

When using Verilog for synthesis, this code produces the desired circuit, which is a half-adder. However, if this code is used for simulation of circuits, the values of *s* and *c* are updated only as a result of changes in the value of *x*. Changes in *y* will have no effect, because *y* is not included in the sensitivity list. To avoid mismatches between simulation results and synthesized circuits, **always** blocks for combinational circuits should include all signals used on the right-hand side of assignments in the block.

Variables versus Nets

Only nets can serve as the targets of continuous assignment statements. Variables assigned values inside an **always** block have to be of type **reg** or **integer**. It is not possible to make an assignment to a signal both inside an **always** block and via a continuous assignment statement.

Assignments in Multiple always Blocks

In a module that has multiple **always** blocks, all the **always** blocks are concurrent with respect to one another. Therefore, a given variable should never be assigned a value in more than one **always** block. Doing so would mean that there exist multiple concurrent assignments to this variable, which makes no sense.

Blocking versus Non-blocking Assignments

When describing a combinational circuit in an **always** construct, it is best to use only blocking assignments (see Section A.11.7). For sequential circuits, non-blocking assignments should be used (see Section A.14.5). Blocking and non-blocking assignments should not be mixed in a single **always** block.

It is not possible to model both a combinational output and a sequential output in a single **always** block. The sequential output requires an edge-triggered event control, such as @(**posedge** Clock), and this means that all variables assigned a value in the **always** block will be implemented as the outputs of flip-flops.

Module Instantiation

A **defparam** statement must reference the instance name of a module, but not just the subcircuit's module name. The code

```
bit_count cbits (T, C);
   defparam bit_count.n = 8, bit_count.logn = 3;
```

is illegal, while the code

```
bit_count cbits (T, C);
   defparam cbits.n = 8, cbits.logn = 3;
```

is syntactically correct.

Label, Net, and Variable Names

It is illegal to use any Verilog keyword as a label, net, or variable name. For example, it is illegal to name a signal *input* or *output*.

Labeled Begin-End Blocks

It is legal to define a variable or parameter inside a **begin-end** block, but only if the block has a label. The code

```
always @(X)
begin
   integer k;
   Count = 0;
   for (k = 0; k < n; k = k+1)
      Count = Count + X[k];
end
```

is illegal, while the code

```
always @(X)
begin: label
   integer k;
   Count = 0;
   for (k = 0; k < n; k = k+1)
      Count = Count + X[k];
end
```

is syntactically correct.

Implied Memory

As shown in Section A.14.1, implied memory is used to describe storage elements. Care must be taken to avoid unintentional implied memory. The code

```
always @(LA)
   if (LA == 1)
      EA = 1;
```

results in implied memory for the *EA* variable. If this is not desired, then the code can be fixed by writing

```
always @(LA)
   if (LA == 1)
      EA = 1;
   else
      EA = 0;
```

Implied memory also applies to **case** statements. The code

```
always @(W)
   case (W)
      2'b01:  EA = 1;
      2'b10:  EB = 1;
   endcase
```

does not specify the value of the *EA* variable when *W* is not equal to 01, and it does not specify the value of *EB* when *W* is not equal to 10. To avoid having implied memory for both *EA* and *EB*, these variables should be assigned default values, as in the code

```
always @(W)
begin
    EA = 0; EB = 0;
    case (W)
        2'b01:  EA = 1;
        2'b10:  EB = 1;
    endcase
end
```

A.16 CONCLUDING REMARKS

This appendix describes the important features of Verilog that are useful for the synthesis of logic circuits. As mentioned earlier, we do not discuss many features of Verilog which are useful only for simulation of circuits or for other purposes. A reader who wishes to learn more about using Verilog can refer to specialized books [1–7].

REFERENCES

1. D. A. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*, 5th ed., (Kluwer: Norwell, MA, 2002).
2. Z. Navabi, *Verilog Digital System Design*, 2nd ed., (McGraw-Hill: New York, 2006).
3. S. Palnitkar, *Verilog HDL—A Guide to Digital Design and Synthesis*, 2nd ed., (Prentice-Hall: Upper Saddle River, NJ, 2003).
4. D. R. Smith and P. D. Franzon, *Verilog Styles for Synthesis of Digital Systems*, (Prentice-Hall: Upper Saddle River, NJ, 2000).
5. J. Bhasker, *Verilog HDL Synthesis—A Practical Primer*, (Star Galaxy Publishing: Allentown, PA, 1998).
6. D. J. Smith, *HDL Chip Design*, (Doone Publications: Madison, AL, 1996).
7. S. Sutherland, *Verilog 2001. A Guide to the New Features of the Verilog Hardware Description Language*, (Kluwer: Hingham, MA, 2001).