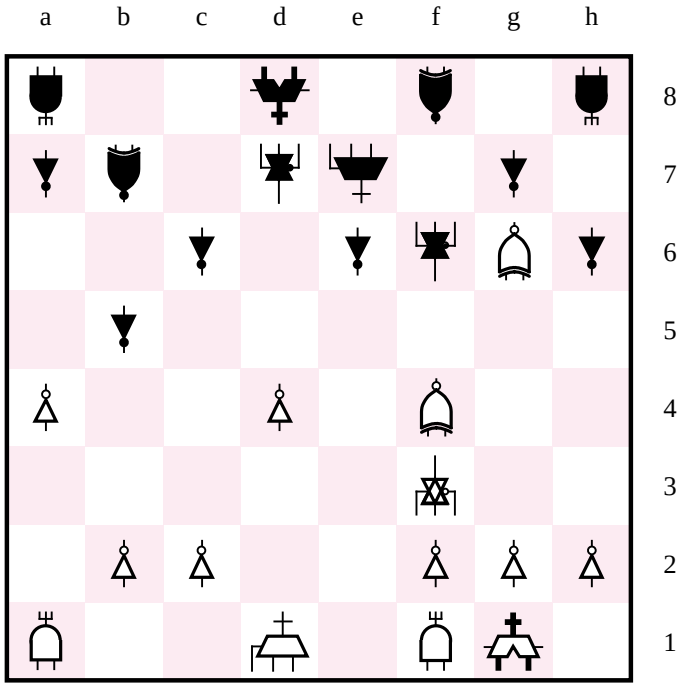
appendix

A

VHDL Reference



12. a2–a4, Bc8–b7

This appendix describes the features of VHDL that are 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 VHDL in sections 2.9 and 4.12.5.

Another useful source of information on VHDL is the MAX+plusII CAD system that accompanies the book. The on-line help included with the software describes how to use VHDL with MAX+plusII, and the "templates" provided with the Text Editor tool are a convenient guide to VHDL syntax. We describe how to access these features of the CAD tools in Appendix B.

In some ways VHDL uses an unusual syntax for describing logic circuits. The prime reason is that VHDL was originally intended to be a language for documenting and simulating circuits, rather than for describing circuits for synthesis. This appendix is not meant to be a comprehensive VHDL manual. While we discuss almost all the features of VHDL that are useful in the synthesis of logic circuits, we do not discuss any of the features that are useful only for simulation of circuits or for other purposes. Although the omitted features are not needed for any of the examples used in this book, a reader who wishes to learn more about using VHDL can refer to specialized books [1–7].

How *Not* to Write VHDL Code

In section 2.9 we mentioned the most common problem encountered by designers who are just beginning to write VHDL code. The tendency for the novice is to write 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. This book contains more than 100 examples of complete VHDL code that represents 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 general guideline is to assume that if the designer cannot readily determine what logic circuit is described by the VHDL code, then the CAD tools are not likely to synthesize the circuit that the designer is trying to describe.

Since VHDL is a complex language, errors in syntax and usage are quite common. Some problems encountered by our students, as novice designers, are listed at the end of this appendix in section A.11. The reader may find it useful to examine these errors in an effort to avoid them when writing code.

Once complete VHDL 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 VHDL, logic circuits, and logic synthesis by studying the circuits that are produced automatically by the CAD tools.

A.1 DOCUMENTATION IN VHDL CODE

Documentation can be included in VHDL code by writing a comment. The two characters '-', '-' denote the beginning of the comment. The VHDL compiler ignores any text on a line after the '--'.

Example A.1

```
- - this is a VHDL comment
```

A.2 DATA OBJECTS

Information is represented in VHDL code as data objects. Three kinds of data objects are provided: signals, constants, and variables. For describing logic circuits, the most important data objects are signals. They represent the logic signals (wires) in the circuit. The constants and variables are also sometimes useful for describing logic circuits, but they are used infrequently.

A.2.1 DATA OBJECT NAMES

The rules for specifying data object names are simple: any alphanumeric character may be used in the name, as well as the '_' underscore character. There are four caveats. A name cannot be a VHDL keyword, it must begin with a letter, it cannot end with an '_' underscore, and it cannot have two successive '_' underscores. Thus examples of legal names are *x*, *x*1, *x_y*, and *Byte*. Some examples of illegal names are 1*x*, *_y*, *x_ _y*, and *entity*. The latter name is not allowed because it is a VHDL keyword. We should note that VHDL is not case sensitive. Hence x is the same as X, and ENTITY is the same as entity. To make the examples of VHDL code in this book more readable, we use uppercase letters in all keywords.

To avoid confusion when using the word signal, which can mean either a VHDL data object or a logic signal in a circuit, we sometimes write the VHDL data object as SIGNAL.

A.2.2 DATA OBJECT VALUES AND NUMBERS

We use SIGNAL data objects to represent individual logic signals in a circuit, multiple logic signals, and binary numbers (integers). The value of an individual SIGNAL is specified using apostrophes, as in '0' or '1'. The value of a multibit SIGNAL is given with double quotes. An example of a four-bit SIGNAL value is "1001", and an eight-bit value is "10011000". Double quotes can also be used to denote a binary number. Hence while "1001" can represent the four SIGNAL values '1', '0', '0', '1', it can also mean the integer $(1001)_2 = (9)_{10}$. Integers can alternatively be specified in decimal by not using quotes, as in 9 or 152. The values of CONSTANT or VARIABLE data objects are specified in the same way as for SIGNAL data objects.

A.2.3 SIGNAL DATA OBJECTS

SIGNAL data objects represent the logic signals, or wires, in a circuit. There are three places in which signals can be declared in VHDL code: in an entity declaration (see section

A.4.1), in the declarative section of an architecture (see section A.4.2), and in the declarative section of a package (see section A.5). A signal has to be declared with an associated *type*, as follows:

```
SIGNAL signal_name : type_name ;
```

The signal's *type_name* determines the legal values that the signal can have and its legal uses in VHDL code. In this section we describe 10 signal types: BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, STD_ULOGIC, SIGNED, UNSIGNED, INTEGER, ENUMERATION, and BOOLEAN.

A.2.4 BIT AND BIT_VECTOR TYPES

These types are predefined in the VHDL Standards IEEE 1076 and IEEE 1164. Hence no library is needed to use these types in the code. Objects of BIT type can have the values '0' or '1'. An object of BIT_VECTOR type is a linear array of BIT objects.

Example A.2

```
SIGNAL  x1   : BIT ;
SIGNAL  C    : BIT_VECTOR (1 TO 4) ;
SIGNAL  Byte : BIT_VECTOR (7 DOWNTO 0) ;
```

The signals *C* and *Byte* illustrate the two possible ways of defining a multibit data object. The syntax "lowest_index TO highest_index" is useful for a multibit signal that is simply an array of bits. In the signal *C* the most-significant (left-most) bit is referenced using lowest_index, and the least-significant (right-most) bit is referenced using highest_index. The syntax "highest_index DOWNTO lowest_index" is useful if the signal represents a binary number. In this case the most-significant (left-most) bit has the index highest_index, and the least-significant (right-most) bit has the index lowest_index.

The multibit signal *C* represents four BIT objects. It can be used as a single four-bit quantity, or each bit can be referred to individually. The syntax for referring to the signals individually is $C(1)$, $C(2)$, $C(3)$, or $C(4)$. An assignment statement such as

```
C  <= "1010" ;
```

results in $C(1) = 1$, $C(2) = 0$, $C(3) = 1$, and $C(4) = 0$.

The signal *Byte* comprises eight BIT objects. The assignment statement

```
Byte <= "10011000" ;
```

results in *Byte*(7) = 1, *Byte*(6) = 0, and so on to *Byte*(0) = 0.

A.2.5 STD_LOGIC AND STD_LOGIC_VECTOR TYPES

The STD_LOGIC type was added to the VHDL Standard in IEEE 1164. It provides more flexibility than the BIT type. To use this type, we must include the two statements

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
```

These statements provide access to the *std_logic_1164* package, which defines the STD_LOGIC type. We describe VHDL packages in section A.5. In general, they are used as a place to store VHDL code, such as the code that defines a type, which can then be used in other source code files. The following values are legal for a STD_LOGIC data object: 0, 1, Z, –, L, H, U, X, and W. Only the first four are useful for synthesis of logic circuits. The value Z represents high impedance, and – stands for "don't care." The value L stands for "weak 0," H means "weak 1," U means "uninitialized," X means "unknown," and W means "weak unknown." The STD_LOGIC_VECTOR type represents an array of STD_LOGIC objects.

Example A.3

```
SIGNAL x1, x2, Cin, Cout, Sel  : STD_LOGIC ;
SIGNAL C                       : STD_LOGIC_VECTOR (1 TO 4) ;
SIGNAL X, Y, S                 : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
```

STD_LOGIC objects are often used in logic expressions in VHDL code. STD_LOGIC_VECTOR signals can be used as binary numbers in arithmetic circuits by including in the code the statement

```
USE ieee.std_logic_signed.all ;
```

The *std_logic_signed* package specifies that it is legal to use the STD_LOGIC_VECTOR signals with arithmetic operators, like + (see section A.7.1). The VHDL compiler should generate a circuit that works for signed numbers. An alternative is to use the package *std_logic_unsigned*. In this case the compiler should generate a circuit that works for unsigned numbers.

A.2.6 STD_ULOGIC TYPE

In this book we use the STD_LOGIC type in most examples of VHDL code. This type is actually a *subtype* of the STD_ULOGIC type. Signals that have the STD_ULOGIC type can take the same values as the STD_LOGIC signals that we have been using. The only difference between STD_ULOGIC and STD_LOGIC has to do with the concept of a *resolution function*. In VHDL a resolution function is used to determine what value a signal should take if there are two sources for that signal. For example, two tri-state buffers could both have their outputs connected to a signal, *x*. At some given time one buffer might produce the output value 'Z' and the other buffer might produce the value 1. A resolution function is used to determine that the value of *x* should be 1 in this case. The STD_LOGIC type allows multiple sources for a signal; it resolves the correct value using a resolution function that is provided as part of the *std_logic_1164* package. The STD_ULOGIC type

does not permit signals to have multiple sources. We have introduced STD_ULOGIC for completeness only; it is not used in this book.

A.2.7 SIGNED AND UNSIGNED TYPES

The *std_logic_signed* and *std_logic_unsigned* packages mentioned in section A.2.5 make use of another package, called *std_logic_arith*. This package defines the type of circuit that should be used to implement the arithmetic operators, such as +. The *std_logic_arith* package defines two signal types, SIGNED and UNSIGNED. These types are identical to the STD_LOGIC_VECTOR type because they represent an array of STD_LOGIC signals. The purpose of the SIGNED and UNSIGNED types is to allow the user to indicate in the VHDL code what kind of number representation is being used. The SIGNED type is used in code for circuits that deal with signed (2's complement) numbers, and the UNSIGNED type is used in code that deals with unsigned numbers.

Example A.4 Assume that A and B are signals with the SIGNED type. Assume that A is assigned the value "1000", and B is assigned the value "0001". VHDL provides relational operators (see Table A.1 in section A.3) that can be used to compare the values of two signals. The comparison $A < B$ evaluates to true because the signed values are $A = -8$ and $B = 1$. On the other hand, if A and B are defined with the UNSIGNED type, then $A < B$ evaluates to false because the unsigned values are $A = 8$ and $B = 1$.

The *std_logic_signed* package specifies that STD_LOGIC_VECTOR signals should be treated like SIGNED signals. Similarly, *std_logic_unsigned* specifies that STD_LOGIC_VECTOR signals should be treated like UNSIGNED signals. It is an arbitrary choice whether code is written using STD_LOGIC_VECTOR signals in conjunction with the *std_logic_signed* or *std_logic_unsigned* packages or using SIGNED and UNSIGNED signals with the *std_logic_arith* package.

The *std_logic_arith* package, and hence the *std_logic_signed* and *std_logic_unsigned* packages, are not actually a part of the VHDL standards. They are provided by Synopsys Inc., which is a vendor of CAD software. However, these packages are included with most CAD systems that support VHDL, and they are widely used in practice.

A.2.8 INTEGER TYPE

The VHDL standard defines the INTEGER type for use with arithmetic operators. In this book the STD_LOGIC_VECTOR type is usually preferred in code for arithmetic circuits, but the INTEGER type is used occasionally. An INTEGER signal represents a binary number. The code does not specifically give the number of bits in the signal, as it does for STD_LOGIC_VECTOR signals. By default, an INTEGER signal has 32 bits and can represent numbers from $-(2^{31} - 1)$ to $2^{31} - 1$. This is one number less than the normal 2's complement range; the reason is simply that the VHDL standard specifies an equal number of negative and positive numbers. Integers with fewer bits can also be declared, using the RANGE keyword.

Example A.5

```
SIGNAL X : INTEGER RANGE −127 TO 127 ;
```

This defines *X* as an eight-bit signed number.

A.2.9 BOOLEAN TYPE

An object of type BOOLEAN can have the values TRUE or FALSE, where TRUE is equivalent to 1 and FALSE is 0.

Example A.6

```
SIGNAL Flag : Boolean ;
```

A.2.10 ENUMERATION TYPE

A SIGNAL of ENUMERATION type is one for which the possible values that the signal can have are user specified. The general form of an ENUMERATION type is

```
TYPE enumerated_type_name IS (name {, name}) ;
```

The curly brackets indicate that one or more additional items can be included. We use these brackets in this manner in several places in the appendix. The most common example of using the ENUMERATION type is for specifying the states for a finite-state machine.

Example A.7

```
TYPE State_type IS (stateA, stateB, stateC) ;
SIGNAL y : State_type ;
```

This declares a signal named *y*, for which the legal values are *stateA*, *stateB*, and *stateC*. When the code is translated by the VHDL compiler, it automatically assigns bit patterns (codes) to represent *stateA*, *stateB*, and *stateC*.

A.2.11 CONSTANT DATA OBJECTS

A CONSTANT is a data object whose value cannot be changed. Unlike a SIGNAL, a CONSTANT does not represent a wire in a circuit. The general form of a CONSTANT declaration is

```
CONSTANT constant_name : type_name := constant_value ;
```

The purpose of a constant is to improve the readability of code, by using the name of the constant in place of a value or number.

Example A.8

```
CONSTANT Zero : STD_LOGIC_VECTOR (3 DOWNTO 0) := "0000" ;
```

Then the word *Zero* can be used in the code to indicate the constant value "0000".

A.2.12 VARIABLE DATA OBJECTS

A VARIABLE, unlike a SIGNAL, does not necessarily represent a wire in a circuit. VARIABLE data objects are sometimes used to hold the results of computations and for the index variables in loops. We will give some examples in section A.9.7.

A.2.13 TYPE CONVERSION

VHDL is a strongly type-checked language, which means that it does not permit the value of a signal of one type to be assigned to another signal that has a different type. Even for signals that intuitively seem compatible, such as BIT and STD_LOGIC, using the two types together is not permitted. To avoid this problem, we generally use only the STD_LOGIC and STD_LOGIC_VECTOR types in this book. When it is necessary to use code that has a mixture of types, type-conversion functions can be used to convert from one type to another.

Assume that *X* is defined as an eight-bit STD_LOGIC_VECTOR signal and *Y* is an INTEGER signal defined with the range from 0 to 255. An example of a conversion function that allows the value of *Y* to be assigned to *X* is

```
X <= CONV_STD_LOGIC_VECTOR(Y, 8) ;
```

This conversion function has two parameters: the name of the signal to be converted and the number of bits in *X*. The function is provided as part of the *std_logic_arith* package; hence that package must be included in the code using the appropriate LIBRARY and USE clauses. Other conversion functions are described in the MAX+plusII on-line help.

A.2.14 ARRAYS

We said above that the BIT_VECTOR and STD_LOGIC_VECTOR types are arrays of BIT and STD_LOGIC signals, respectively. The definitions of these arrays, which are provided as part of the VHDL standards, are

```
TYPE BIT_VECTOR IS ARRAY (NATURAL RANGE < >) OF BIT ;
TYPE STD_LOGIC_VECTOR IS ARRAY (NATURAL RANGE < >) OF STD_LOGIC ;
```

The sizes of the arrays are not set in the definitions; the syntax (NATURAL RANGE < >) has the effect of allowing the user to set the size of the array when a data object of either type is declared. Arrays of any type can be defined by the user. For example

```
TYPE Byte IS ARRAY (7 DOWNTO 0) OF STD_LOGIC ;
SIGNAL X : Byte ;
```

declares the signal *X* with the type *Byte*, which is an eight-element array of STD_LOGIC data objects.

An example that defines a two-dimensional array is

```
TYPE RegArray IS ARRAY(3 DOWNTO 0) OF STD_LOGIC_VECTOR(7 DOWNTO 0) ;
SIGNAL R : RegArray ;
```

This code defines *R* as an array with four elements. Each element is an eight-bit STD_LOGIC_VECTOR signal. The syntax $R(i)$, where $3 \geq i \geq 0$, is used to refer to element i of the array. The syntax $R(i)(j)$, where $7 \geq j \geq 0$, is used to refer to one bit in the array $R(i)$. This bit has the type STD_LOGIC. An example using the *RegArray* type is given in section 10.2.6.

A.3 OPERATORS

VHDL provides Boolean operators, arithmetic operators, and relational operators. They are categorized in an unusual way, shown in Table A.1, according to the precedence of the operators. Note that operators in the same category do not have precedence over one another. There is no precedence among any Boolean operators. Thus a logic expression

```
x1 AND x2 AND x3 OR x4
```

does not have the $x_1x_2x_3 + x_4$ meaning that would be expected because AND does not have precedence over OR. In fact, this expression is not even legal in VHDL as written above. To be both legal and have the desired meaning, it must be written as

```
(x1 AND x2 AND x3) OR x4
```

For the relational operators, /= means *not equal*, <= means *less than or equal*, and >= means *greater than or equal*.

Table A.1 The VHDL operators.

	Operator Class	Operator
Highest precedence	Miscellaneous	**, ABS, NOT
	Multiplying	*, /, MOD, REM
	Sign	+, –
	Adding	+, –, &
	Relational	=, /=, <, <=, >, >=
Lowest precedence	Logical	AND, OR, NAND, NOR, XOR, XNOR

A.4 VHDL DESIGN ENTITY

A circuit or subcircuit described with VHDL code is called a *design entity*, or just *entity*. Figure A.1 shows the general structure of an entity. It has two main parts: the *entity declaration*, which specifies the input and output signals for the entity, and the *architecture*, which gives the circuit details.

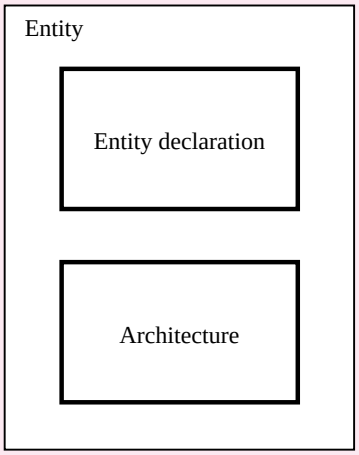


Figure A.1 The general structure of a VHDL design entity.

A.4.1 ENTITY DECLARATION

The input and output signals in an entity are specified using the ENTITY declaration, as indicated in Figure A.2. The name of the entity can be any legal VHDL name. The square brackets indicate an optional item. The input and output signals are specified using the keyword PORT. Whether each port is an input, output, or bidirectional signal is specified by the *mode* of the port. The available modes are summarized in Table A.2. If the mode of a port is not specified, it is assumed to have the mode IN.

Table A.2 The possible modes for signals that are entity ports.

Mode	Purpose
IN	Used for a signal that is an input to an entity.
OUT	Used for a signal that is an output from an entity. The value of the signal can not be used inside the entity. This means that in an assignment statement, the signal can appear only to the left of the <= operator.
INOUT	Used for a signal that is both an input to an entity and an output from the entity.
BUFFER	Used for a signal that is an output from an entity. The value of the signal can be used inside the entity, which means that in an assignment statement, the signal can appear both on the left and right sides of the <= operator.

A.4.2 ARCHITECTURE

An ARCHITECTURE provides the circuit details for an entity. The general structure of an architecture is shown in Figure A.3. It has two main parts: the *declarative region* and the *architecture body*. The declarative region appears preceding the BEGIN keyword. It can be used to declare signals, user-defined types, and constants. It can also be used to declare components and to specify attributes; we discuss the COMPONENT and ATTRIBUTE keywords in sections A.6 and A.10.13, respectively.

The functionality of the entity is specified in the architecture body, which follows the BEGIN keyword. This specification involves statements that define the logic functions in the circuit, which can be given in a variety of ways. We will discuss a number of possibilities in the sections that follow.

```
ENTITY entity_name IS
    PORT ( [SIGNAL] signal_name {, signal_name} : [mode] type_name {;
           SIGNAL] signal_name {, signal_name}  : [mode] type_name } ) ;
END entity_name ;
```

Figure A.2 The general form of an entity declaration.

```
ARCHITECTURE architecture_name OF entity_name IS
    [SIGNAL declarations]
    [CONSTANT declarations]
    [TYPE declarations]
    [COMPONENT declarations]
    [ATTRIBUTE specifications]
BEGIN
    {COMPONENT instantiation statement ;}
    {CONCURRENT ASSIGNMENT statement ;}
    {PROCESS statement ;}
    {GENERATE statement ;}
END [architecture_name] ;
```

Figure A.3 The general form of an architecture.

Example A.9 Figure A.4 gives the VHDL code for an entity named *fulladd*, which represents a full-adder circuit. (The full-adder is discussed in section 5.2.) The entity declaration specifies the input and output signals. 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 input and output signals are called the *ports* of the entity. This term is adopted from the electrical jargon in which it refers to an input or output connection in an electrical circuit.

The architecture defines the functionality of the full-adder using logic equations. The name of the architecture can be any legal VHDL name. We chose the name LogicFunc for this simple example. In terms of the general form of the architecture in Figure A.3, a logic equation is a type of concurrent assignment statement. These statements are described in section A.7.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY fulladd IS
    PORT ( Cin, x, y  : IN   STD_LOGIC ;
           s, Cout    : OUT  STD_LOGIC ) ;
END fulladd ;

ARCHITECTURE LogicFunc OF fulladd IS
BEGIN
    s <= x XOR y XOR Cin ;
    Cout <= (x AND y) OR (x AND Cin) OR (y AND Cin) ;
END LogicFunc ;
```

Figure A.4 Code for a full-adder.

A.5 PACKAGE

A VHDL package serves as a repository. It is used to hold VHDL code that is of general use, like the code that defines a type. The package can be included for use in any number of other source code files, which can then use the definitions provided in the package. Like an architecture, introduced in section A.4.2, a package can have two main parts: the *package declaration* and the *package body*. The *package_body* is an optional part, which we do not use in this book; one use of a package body is to define VHDL functions, such as the conversion functions introduced in section A.2.13.

The general form of a package declaration is depicted in Figure A.5. Definitions provided in the package, such as the definition of a type, can be used in any source code file that includes the statements

```
LIBRARY library_name ;
USE library_name.package_name.all ;
```

The *library_name* represents the location in the computer file system where the package is stored. A library can either be provided as part of a CAD system, in which case it is termed a *system library*, or be created by the user, in which case it is called a *user library*. An example of a system library is the *ieee* library. We discussed four packages in that library in section A.2: *std_logic_1164*, *std_logic_signed*, *std_logic_unsigned*, and *std_logic_arith*.

A special case of a user library is represented by the file-system directory where the VHDL source code file that declares a package is stored. This directory can be referred to by the library name *work*, which stands for *working directory*. Hence, if a source code file that contains a package declaration called *user_package_name* is compiled, then the package can be used in another source code file (which is stored in the same file-system directory) by including the statements

```
LIBRARY work ;
USE work.user_package_name.all ;
```

Actually, for the special case of the *work* library, the LIBRARY clause is not required, because the work library is always accessible.

Figure A.5 shows that the package declaration can be used to declare signals and components. Components are discussed in the next section. A signal declared in a package can be used by any design entity that accesses the package. Such signals are similar in

```
PACKAGE package_name IS
    [TYPE declarations]
    [SIGNAL declarations]
    [COMPONENT declarations]
END package_name ;
```

Figure A.5 The general form of a PACKAGE declaration.

concept to global variables used in computer programming languages. In contrast, a signal declared in an architecture can be used only inside that architecture. Such signals are analogous to local variables in a programming language.

A.6 USING SUBCIRCUITS

A VHDL entity defined in one source code file can be used as a subcircuit in another source code file. In VHDL jargon the subcircuit is called a *component*. A subcircuit must be declared using a *component declaration*. This statement specifies the name of the subcircuit and gives the names of its input and output ports. The component declaration can appear either in the declaration region of an architecture or in a package declaration. The general form of the statement is shown in Figure A.6. The syntax used is similar to the syntax in an entity declaration.

Once a component declaration is given, the component can be *instantiated* as a subcircuit. This is done using a *component instantiation* statement. It has the general form

```
instance_name : component_name PORT MAP (
    formal_name => actual_name {, formal_name => actual_name} ) ;
```

Each *formal_name* is the name of a port in the subcircuit. Each *actual_name* is the name of a signal in the code that instantiates the subcircuit. The syntax "formal_name =>" is provided so that the order of the signals listed after the PORT MAP keywords does not have to be the same as the order of the ports in the corresponding COMPONENT declaration. In VHDL jargon this is called the *named association*. If the signal names following the PORT MAP keywords are given in the same order as in the COMPONENT declaration, then "formal_name =>" is not needed. This is called the *positional association*.

An example using a component (subcircuit) is shown in Figure A.7. It gives the code for a four-bit ripple-carry adder built using four instances of the *fulladd* subcircuit. The inputs to the adder are the carry-in, *Cin*, and the 2 four-bit numbers *X* and *Y*. The output is the four-bit sum, *S*, and the carry-out, *Cout*. We have chosen the name Structure in the architecture because the hierarchical style of code that uses subcircuits is often called the *structural* style. Observe that a three-bit signal, C, is declared to represent the carry-outs from stages 0, 1, and 2. This signal is declared in the architecture, rather than in the entity declaration, because it is used internally in the circuit and is not an input or output port.

```
COMPONENT component_name
    [GENERIC (parameter_name : integer := default_value {;
                parameter_name : integer := default_value} ) ;]
    PORT ( [SIGNAL] signal_name {, signal_name} : [mode] type_name {;
            SIGNAL] signal_name {, signal_name}  : [mode] type_name } ) ;
END COMPONENT ;
```

Figure A.6 The general form of a component declaration.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder IS
    PORT ( Cin    : IN     STD_LOGIC ;
           X, Y   : IN     STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S      : OUT    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout   : OUT    STD_LOGIC ) ;
END adder ;

ARCHITECTURE Structure OF adder IS
    SIGNAL C : STD_LOGIC_VECTOR(1 TO 3) ;
    COMPONENT fulladd
        PORT ( Cin, x, y : IN     STD_LOGIC ;
               s, Cout   : OUT STD_LOGIC) ;
    END COMPONENT ;
BEGIN
    stage0: fulladd PORT MAP ( Cin , X(0), Y(0), S(0), C(1) ) ;
    stage1: fulladd PORT MAP ( C(1), X(1), Y(1), S(1), C(2) ) ;
    stage2: fulladd PORT MAP ( C(2), X(2), Y(2), S(2), C(3) ) ;
    stage3: fulladd PORT MAP (
        x => X(3), y => Y(3), Cin => C(3), s => S(3), Cout => Cout ) ;
END Structure ;
```

Figure A.7 Code for a four-bit adder, using component instantiation.

The next statement in the architecture gives the component declaration for the *fulladd* subcircuit. The architecture body instantiates four copies of the full-adder subcircuit. In the first three instantiation statements, we have used positional association because the signals are listed in the same order as given in the declaration for the fulladd component in Figure A.4. The last instantiation statement gives an example of named association. Note that it is legal to use the same name for a signal in the architecture that is used for a port name in a component. An example of this is the *Cout* signal. The signal names used in the instantiation statements implicitly specify how the component instances are interconnected to create the adder entity.

A second example of component instantiation is shown in Figure A.8. A package called *lpm_components* in the library named *lpm* is included in the code. This package represents a collection of components called the *Library of Parameterized Modules (LPM)*, which is a standardized library of circuit building blocks that are generally useful for implementing logic circuits. The MAX+plusII CAD system includes the LPM components as standard building blocks for creating logic circuits. Information about the components in the library can be found in the MAX+plusII on-line help. We describe how to access this information in Tutorial 3.

The code in Figure A.8 instantiates the LPM component called *lpm_add_sub*, which is introduced in section 5.5.1. It represents an adder/subtractor circuit. The GENERIC

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
LIBRARY lpm ;
USE lpm.lpm_components.all ;

ENTITY adderLPM IS
    PORT ( Cin   : IN   STD_LOGIC ;
           X, Y  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout  : OUT  STD_LOGIC ) ;
END adderLPM ;

ARCHITECTURE Structure OF adderLPM IS
BEGIN
    instance: lpm_add_sub
        GENERIC MAP (LPM_WIDTH => 4)
        PORT MAP (
            dataa => X, datab => Y, Cin => Cin, result => S, Cout => Cout ) ;
END Structure ;
```

Figure A.8 Instantiating a four-bit adder from the LPM library.

keyword is used to set the number of bits in the adder/subtractor to 4. We discuss generics in section A.8. The function of each PORT on the *lpm_add_sub* component is self-evident from the port names used in the instantiation statement.

A.6.1 DECLARING A COMPONENT IN A PACKAGE

Figure A.5 shows that a component declaration can be given in a package. An example is shown in Figure A.9. It defines the package named *fulladd_package*, which provides the component declaration for the *fulladd* entity. This package can be stored in a separate

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE fulladd_package IS
    COMPONENT fulladd
        PORT ( Cin, x, y  : IN   STD_LOGIC ;
               s, Cout    : OUT  STD_LOGIC ) ;
    END COMPONENT ;
END fulladd_package ;
```

Figure A.9 An example of a package declaration.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY adder IS
    PORT ( Cin   : IN   STD_LOGIC ;
           X, Y  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout  : OUT  STD_LOGIC ) ;
END adder ;

ARCHITECTURE Structure OF adder IS
    SIGNAL C : STD_LOGIC_VECTOR(1 TO 3) ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, X(0), Y(0), S(0), C(1) ) ;
    stage1: fulladd PORT MAP ( C(1), X(1), Y(1), S(1), C(2) ) ;
    stage2: fulladd PORT MAP ( C(2), X(2), Y(2), S(2), C(3) ) ;
    stage3: fulladd PORT MAP ( C(3), X(3), Y(3), S(3), Cout ) ;
END Structure ;
```

Figure A.10 Using a component defined in a package.

source code file or can be included at the end of the file that defines the *fulladd* entity (see Figure A.4). Any source code that includes the statement "USE work.fulladd_package.all" can use the *fulladd* component as a subcircuit. Figure A.10 shows how a four-bit ripple-carry adder entity can be written to use the package. The code is the same as that in Figure A.7 except that it includes the extra USE clause for the package and deletes the component declaration statement from the architecture.

A.7 CONCURRENT ASSIGNMENT STATEMENTS

A concurrent assignment statement is used to assign a value to a signal in an architecture body. An example was given in Figure A.4, in which the logic equations illustrate one type of concurrent assignment statement. VHDL provides four different types of concurrent assignment statements: simple signal assignment, selected signal assignment, conditional signal assignment, and generate statements.

A.7.1 SIMPLE SIGNAL ASSIGNMENT

A simple signal assignment statement is used for a logic or an arithmetic expression. The general form is

```
signal_name <= expression ;
```

where <= is the VHDL *assignment operator*. The following examples illustrate its use.

```
SIGNAL x1, x2, x3, f : STD_LOGIC ;
      .
      .
      .
f <= (x1 AND x2) OR x3 ;
```

This defines *f* in a logic expression, which involves single-bit quantities. VHDL also supports multibit logic expressions, as in

```
SIGNAL A, B, C : STD_LOGIC_VECTOR (1 TO 3) ;
      .
      .
      .
C <= A AND B ;
```

This results in $C(1)=A(1)\cdot B(1)$, $C(2)=A(2)\cdot B(2)$, and $C(3)=A(3)\cdot B(3)$.

An example of an arithmetic expression is

```
SIGNAL X, Y, S : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
      .
      .
      .
S <= X + Y ;
```

This represents a four-bit adder, without carry-in and carry-out. We can alternatively declare a carry-in signal, *Cin*, and a five-bit signal, *Sum*, as follows

```
SIGNAL Cin : STD_LOGIC ;
SIGNAL Sum : STD_LOGIC_VECTOR (4 DOWNTO 0) ;
```

Then the statement

```
Sum <= ('0' & X) + Y + Cin ;
```

represents the four-bit adder with carry-in and carry-out. The four sum bits are *Sum*(3) to *Sum*(0), while the carry-out is the bit *Sum*(4). The syntax ('0' & X) uses the VHDL *concatenate operator*, &, to put a 0 on the left end of the signal *X*. The reader should not confuse this use of the & symbol with the logical AND operation, which is the usual meaning of this symbol; in VHDL the logical AND is indicated by the word AND, and & means concatenate. The concatenate operation prepends a 0 digit onto *X*, creating a five-bit number. VHDL requires at least one of the operands of an arithmetic expression to have the

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY adder IS
    PORT ( Cin   : IN   STD_LOGIC ;
           X, Y  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout  : OUT  STD_LOGIC ) ;
END adder ;

ARCHITECTURE Behavior OF adder IS
    SIGNAL Sum : STD_LOGIC_VECTOR(4 DOWNTO 0) ;
BEGIN
    Sum <= ('0' & X) + Y + Cin ;
    S <= Sum(3 DOWNTO 0) ;
    Cout <= Sum(4) ;
END Behavior ;
```

Figure A.11 Code for a four-bit adder, using arithmetic expressions.

same number of bits as the signal used to hold the result. The complete code for the four-bit adder with carry signals is given in Figure A.11. We should note that this is a different way (it is actually a better way) to describe a four-bit adder, in comparison with the structural code in Figure A.7. Observe that the statement "S <= Sum(3 DOWNTO 0)" assigns the lower four bits of the *Sum* signal, which are the four sum bits, to the output *S*.

A.7.2 ASSIGNING SIGNAL VALUES USING OTHERS

Assume that we wish to set all bits in the signal *S* to 0. As we already know, one way to do so is to write "S <= "0000" ;". If the number of bits in *S* is large, a more convenient way of expressing the assignment statement is to use the OTHERS keyword, as in

```
S <= (OTHERS => '0') ;
```

This statement also sets all bits in *S* to 0. But it has the benefit of working for any number of bits, not just four. In general, the meaning of (OTHERS => *Value*) is to set each bit of the destination operand to *Value*. An example of code that uses this construct is shown in Figure A.28.

A.7.3 SELECTED SIGNAL ASSIGNMENT

A selected signal assignment statement is used to set the value of a signal to one of several alternatives based on a selection criterion. The general form is

```
[label:] - - an optional label can be placed here
WITH expression SELECT
      signal_name <= expression WHEN constant_value{,
                     expression WHEN constant_value} ;
```

Example A.10

```
SIGNAL x1, x2, Sel, f : STD_LOGIC ;
        .
        .
        .
WITH Sel SELECT
    f <=  x1 WHEN '0',
          x2 WHEN OTHERS ;
```

This code describes a 2-to-1 multiplexer with *Sel* as the select input. In a selected signal assignment, all possible values of the select input, *Sel* in this case, must be explicitly listed in the code. The word OTHERS provides an easy way to meet this requirement. OTHERS represents all possible values not already listed. In this case the other possible values are 1, Z, –, and so on. Another requirement for the selected signal assignment is that each WHEN clause must specify a criterion that is mutually exclusive of the criteria in all other WHEN clauses.

A.7.4 CONDITIONAL SIGNAL ASSIGNMENT

Similar to the selected signal assignment, the conditional signal assignment is used to set a signal to one of several alternative values. The general form is

```
[label:]
signal_name <=  expression WHEN logic_expression ELSE
                {expression WHEN logic_expression ELSE}
                expression ;
```

An example is

```
f <= '1' WHEN x1 = x2 ELSE '0' ;
```

One key difference in comparison with the selected signal assignment has to be noted. The conditions listed after each WHEN clause need not be mutually exclusive, because the conditions are given a priority from the first listed to the last listed. This is illustrated by the example in Figure A.12. The code represents a priority encoder in which the highest-priority request is indicated as the output of the circuit. (Encoder circuits are described in Chapter 6.) The output, *f*, of the priority encoder comprises two bits whose values depend on the three inputs, *req*1, *req*2, and *req*3. If *req*1 is 1, then *f* is set to 01. If *req*2 is 1, then *f* is set to 10, but only if *req*1 is not also 1. Hence *req*1 has higher priority than *req*2. Similarly, *req*1 and *req*2 have higher priority than *req*3. Thus if *req*3 is 1, then *f* is 11, but only if neither *req*1 nor *req*2 is also 1. For this priority encoder, if none of the three inputs is 1, then *f* is assigned the value 00.

A.7.5 GENERATE STATEMENT

There are two variants of the GENERATE statement: the FOR GENERATE and the IF GENERATE. The general form of both types is shown in Figure A.13. The IF GENERATE statement is seldom needed, but FOR GENERATE is often used in practice. It provides a convenient way of repeating either a logic equation or a component instantiation. Figure A.14 illustrates its use for component instantiation. The code in the figure is equivalent to the code given in Figure A.7.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY priority IS
    PORT ( req1, req2, req3   : IN     STD_LOGIC ;
           f                  : OUT  STD_LOGIC_VECTOR(1 DOWNTO 0) ) ;
END priority ;

ARCHITECTURE Behavior OF priority IS
BEGIN
    f <= "01" WHEN req1 = '1' ELSE
         "10" WHEN req2 = '1' ELSE
         "11" WHEN req3 = '1' ELSE
         "00" ;
END Behavior;
```

Figure A.12 A priority encoder described with a conditional signal assignment.

```
generate_label:
FOR index_variable IN range GENERATE
      statement ;
      {statement ;}
END GENERATE ;
```

```
generate_label:
IF expression GENERATE
      statement ;
      {statement ;}
END GENERATE ;
```

Figure A.13 The general forms of the GENERATE statement.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY adder IS
    PORT ( Cin   : IN    STD_LOGIC ;
           X, Y  : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           S     : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Cout  : OUT   STD_LOGIC ) ;
END adder ;

ARCHITECTURE Structure OF adder IS
    SIGNAL C : STD_LOGIC_VECTOR(0 TO 4) ;
BEGIN
    C(0) <= Cin ;
    Generate_label:
    FOR i IN 0 TO 3 GENERATE
        bit: fulladd PORT MAP ( C(i), X(i), Y(i), S(i), C(i+1) ) ;
    END GENERATE ;
    Cout <= C(4) ;
END Structure ;
```

Figure A.14 An example of component instantiation with FOR GENERATE.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY addern IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( Cin   : IN   STD_LOGIC ;
           X, Y  : IN   STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
           S     : OUT  STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
           Cout  : OUT  STD_LOGIC ) ;
END addern ;

ARCHITECTURE Structure OF addern IS
    SIGNAL C : STD_LOGIC_VECTOR(0 TO n) ;
BEGIN
    C(0) <= Cin ;
    Generate_label:
    FOR i IN 0 TO n−1 GENERATE
        stage: fulladd PORT MAP ( C(i), X(i), Y(i), S(i), C(i+1) ) ;
    END GENERATE ;
    Cout <= C(4) ;
END Structure ;
```

Figure A.15 An *n*-bit adder.

A.8 DEFINING AN ENTITY WITH GENERICS

The code in Figure A.14 represents an adder for four-bit numbers. It is possible to make this code more general by introducing a parameter in the code that represents the number of bits in the adder. In VHDL jargon such a parameter is called a GENERIC. Figure A.15 gives the code for an *n*-bit adder entity, named *addern*. The GENERIC keyword is used to define the number of bits, *n*, to be added. This parameter is used in the code, both in the definitions of the signals *X*, *Y*, and *S* and in the FOR GENERATE statement that instantiates the *n* full-adders.

It is possible to use the GENERIC feature with components that are instantiated as subcircuits in other code. In section A.10.10 we give an example that uses the *addern* entity as a subcircuit.

A.9 SEQUENTIAL ASSIGNMENT STATEMENTS

The order in which the concurrent assignment statements in an architecture body appear does not affect the meaning of the code. Many types of logic circuits can be described

using these statements. However, VHDL also provides another type of statements, called *sequential assignment statements*, for which the order of the statements in the code can affect the semantics of the code. There are three variants of the sequential assignment statements: IF statement, CASE statement, and loop statements.

A.9.1 PROCESS STATEMENT

Since the order in which the sequential statements appear in VHDL code is significant, whereas the ordering of concurrent statements is not, the sequential statements must be separated from the concurrent statements. This is accomplished using a PROCESS statement. The PROCESS statement appears inside an architecture body, and it encloses other statements within it. The IF, CASE, and LOOP statements can appear only inside a process. The general form of a PROCESS statement is shown in Figure A.16. Its structure is somewhat similar to an architecture. VARIABLE data objects can be declared (only) inside the process. Any variable declared can be used only by the code within the process; we say that the *scope* of the variable is limited to the process. To use the value of such a variable outside the process, the variable's value can be assigned to a signal. The various elements of the process are best explained by giving some examples. But first we need to introduce the IF, CASE, and LOOP statements.

The IF, CASE, and LOOP statements can be used to describe either combinational or sequential circuits. We will introduce these statements by giving some examples of combinational circuits because they are easier to understand. Sequential circuits are described in section A.10.

A.9.2 IF STATEMENT

The general form of an IF statement is given in Figure A.17. An example using an IF statement for combinational logic is

```
[process_label:]
PROCESS [( signal name {, signal name} )]
    [VARIABLE declarations]
BEGIN
    [WAIT statement]
    [Simple Signal Assignment Statements]
    [Variable Assignment Statements]
    [IF Statements]
    [CASE Statements]
    [LOOP Statements]
END PROCESS [process_label] ;
```

Figure A.16 The general form of a PROCESS statement.

```
IF expression THEN
      statement ;
      {statement ;}
ELSIF expression THEN
      statement ;
      {statement ;}
ELSE
      statement ;
      {statement ;}
END IF ;
```

Figure A.17 The general form of an IF statement.

```
IF Sel = '0' THEN
    f <= x1 ;
ELSE
    f <= x2 ;
END IF ;
```

This code defines the 2-to-1 multiplexer that was used as an example of a selected signal assignment in the previous section. Examples of sequential logic described with IF statements are given in section A.10.

A.9.3 CASE STATEMENT

The general form of a CASE statement is shown in Figure A.18. The *constant_value* can be a single value, such as 2, a list of values separated by the | pipe, such as 2|3, or a range, such as 2 to 4. An example of a CASE statement used to describe combinational logic is

```
CASE expression IS
      WHEN constant_value =>
            statement ;
            {statement ;}
      WHEN constant_value =>
            statement ;
            {statement ;}
      WHEN OTHERS =>
            statement ;
            {statement ;}
END CASE ;
```

Figure A.18 The general form of a CASE statement.

```
CASE Sel IS
    WHEN '0' =>
        f <= x1 ;
    WHEN OTHERS =>
        f <= x2 ;
END CASE ;
```

This code represents the same 2-to-1 multiplexer described in section A.9.2 using the IF statement. Similar to a selected signal assignment, all possible valuations of the expression used for the WHEN clauses must be listed; hence the OTHERS keyword is needed. Also, all WHEN clauses in the CASE statement must be mutually exclusive. Examples of sequential circuits described with the CASE statement are given in section A.10.11.

A.9.4 LOOP STATEMENTS

VHDL provides two types of loop statements: the FOR-LOOP statement and the WHILE-LOOP statement. Their general forms are shown in Figure A.19. These statements are used to repeat one or more sequential assignment statements in much the same way as a FOR GENERATE statement is used to repeat concurrent assignment statements. Examples of the FOR-LOOP are given in section A.9.7.

A.9.5 USING A PROCESS FOR A COMBINATIONAL CIRCUIT

An example of a PROCESS statement is shown in Figure A.20. It includes the code for the IF statement from section A.9.2. The signals *Sel*, *x*1, and *x*2 are shown in parentheses after the PROCESS keyword. They indicate which signals the process depends on and are called the *sensitivity list* of the process. For a process that describes combinational logic, as in this example, the sensitivity list includes all input signals used inside the process.

```
[loop_label:]
FOR variable_name IN range LOOP
    statement ;
    {statement ;}
END LOOP [loop_label] ;
```

```
[loop_label:]
WHILE boolean_expression LOOP
    statement ;
    {statement ;}
END LOOP [loop_label] ;
```

Figure A.19 The general forms of FOR-LOOP and WHILE-LOOP statements.

```
PROCESS ( Sel, x1, x2 )
BEGIN
    IF Sel = '0' THEN
        f <= x1
    ELSE
        f <= x2 ;
    END IF ;
END PROCESS ;
```

Figure A.20 A PROCESS statement.

In VHDL jargon a process is described as follows. When the value of a signal in the sensitivity list changes, the process becomes *active*. Once active, the statements inside the process are "evaluated" in sequential order. Any signal assignments made in the process take effect only after all the statements inside the process have been evaluated. We say that the signal assignment statements inside the process are *scheduled* and will take effect at the end of the process.

The process describes a logic circuit and is translated into logic equations in the same manner as the concurrent assignment statements in an architecture body. The concept of the process statements being evaluated in sequence provides a convenient way of understanding the semantics of the code inside a process. In particular, a key concept is that if multiple assignments are made to a signal inside a process, only the last one to be evaluated has any effect. This is illustrated in the next example.

A.9.6 STATEMENT ORDERING

The IF statement in Figure A.20 describes a multiplexer that assigns either of two inputs, $x1$ or $x2$, to the output f. Another way of describing the multiplexer with an IF statement is shown in Figure A.21. The statement "f <= x1 ;" is evaluated first. However, the signal f may not actually be changed to the value of $x1$, because there may be a subsequent assignment to f in the code inside the process statement. At this point in the process, $x1$ represents the *default* value for f if no other assignment to f is evaluated. If we assume

```
PROCESS ( Sel, x1, x2 )
BEGIN
    f <= x1 ;
    IF Sel = 1 THEN
        f <= x2 ;
    END IF ;
END PROCESS ;
```

Figure A.21 An example illustrating the ordering of statements within a PROCESS.

that *Sel* = 1, then the statement "f <= x2 ;" will be evaluated. The effect of this second assignment to *f* is to override the default assignment. Hence the result of the process is that *f* is set to the value *x*2 when *Sel* = 1. If we assume that *Sel* = 0, then the IF condition fails and *f* is assigned its default value, *x*1.

This example illustrates the effect of the ordering of statements inside a process. If the two statements were reversed in order, then the IF statement would be evaluated first and the statement "f <= x1 ;" would be evaluated last. Hence the process would always result in *f* being set to the value of *x*1.

Implied Memory

Consider the process in Figure A.22. It is the same as the process in Figure A.21 except that the default assignment statement "f <= x1 ;" has been removed. Because the process does not specify a default value for *f*, and there is no ELSE clause in the IF statement, the meaning of the process is that *f* should retain its present value when the IF condition is not satisfied. The following expression is generated by the VHDL compiler for this process

$$f = Sel \cdot x2 + \overline{Sel} \cdot f$$

Hence when *Sel* = 0, the value of *x*2 is "remembered" at the output *f*. In VHDL jargon this is called *implied memory* or *implicit memory*. Although it is rarely useful for combinational circuits, we will show shortly that implied memory is the key concept used to describe sequential circuits.

A.9.7 USING A VARIABLE IN A PROCESS

We mentioned earlier that VHDL provides VARIABLE data objects, in addition to SIGNAL data objects. Unlike a signal, a variable data object does not represent a wire in a circuit. Therefore, a variable can be used to describe the functionality of a logic circuit in ways that are not possible using a signal. This concept is illustrated in Figure A.23. The intent of the code is to describe a logic circuit that counts the number of bits in the three-bit signal *X* that are equal to 1. The count is output using the signal called *Count*, which is a two-bit unsigned integer. Notice that *Count* is declared with the mode *Buffer* because it is used in the architecture body on both the left and right sides of an assignment operator. Table A.2 explains the meaning of the *Buffer* mode.

```
PROCESS ( Sel, x2 )
BEGIN
    IF Sel = 1 THEN
        f <= x2 ;
    END IF ;
END PROCESS ;
```

Figure A.22 An example of implied memory.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY numbits IS
    PORT ( X      : IN     STD_LOGIC_VECTOR(1 TO 3) ;
           Count  : BUFFER INTEGER RANGE 0 TO 3 ) ;
END numbits ;

ARCHITECTURE Behavior OF numbits IS
BEGIN
    PROCESS ( X )   -- count the number of bits in X with the value 1
    BEGIN
        Count <= 0 ;   -- the 0 with no quotes is a decimal number
        FOR i IN 1 TO 3 LOOP
            IF X(i) = '1' THEN
                Count <= Count + 1 ;
            END IF ;
        END LOOP ;
    END PROCESS ;
END Behavior ;
```

Figure A.23 A FOR-LOOP that does not represent a sensible circuit.

Inside the process, *Count* is initially set to 0. No quotes are used for the number 0 in this case, because VHDL allows a decimal number, which we said in section A.2.2 is denoted with no quotes, to be assigned to an INTEGER signal. The code gives a FOR-LOOP with the loop index variable *i*. For the values of *i* from 1 to 3, the IF statement inside the FOR-LOOP checks the value of bit X(i); if it is 1, then the value of *Count* is incremented. The code given in the figure is legal VHDL code and can be compiled without generating any errors. However, it will not work as intended, and it does not represent a sensible logic circuit.

There are two reasons why the code in Figure A.23 will not work as intended. First, there are multiple assignment statements for the signal *Count* within the process. As explained for the previous example, only the last of these assignments will have any effect. Hence if any bit in *X* is 1, then the statement "Count <= '0' ;" will not have the desired effect of initializing *Count* to 0, because it will be overridden by the assignment statement in the FOR-LOOP. Also, the FOR-LOOP will not work as desired, because each iteration for which *X*(1) is 1 will override the effect of the previous iteration. The second reason why the code is not sensible is that the statement "Count <= Count + '1' ;" describes a circuit with feedback. Since the circuit is combinational, such feedback will result in oscillations and the circuit will not be stable.

The desired behavior of the VHDL code in Figure A.23 can be achieved using a variable, instead of a signal. This is illustrated in Figure A.24, in which the variable *Tmp* is used instead of the signal *Count* inside the process. The value of *Tmp* is assigned to *Count* at the

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY Numbits IS
    PORT ( X      : IN  STD_LOGIC_VECTOR(1 TO 3) ;
           Count  : OUT INTEGER RANGE 0 TO 3 ) ;
END Numbits ;

ARCHITECTURE Behavior OF Numbits IS
BEGIN
    PROCESS ( X )  -- count the number of bits in X equal to 1
        VARIABLE TMP : INTEGER ;
    BEGIN
        Tmp := 0 ;
        FOR i IN 1 TO 3 LOOP
            IF X(i) = '1' THEN
                Tmp := Tmp + 1 ;
            END IF ;
        END LOOP ;
        Count <= Tmp ;
    END PROCESS ;
END Behavior ;
```

Figure A.24 The FOR-LOOP from Figure A.23 using a variable.

end of the process. Observe that the assignment statements to *Tmp* are indicated with the := operator, as opposed to the <= operator. The := is called the *variable assignment operator*. Unlike <=, it does not result in the assignment being *scheduled* until the end of the process. The variable assignment takes place immediately. This *immediate* assignment solves the first of the two problems with the code in Figure A.23. The second problem is also solved by using a variable instead of a signal. Because the variable does not represent a wire in a circuit, the FOR-LOOP need not be literally interpreted as a circuit with feedback. By using the variable, the FOR-LOOP represents only a desired *behavior*, or *functionality*, of the circuit. When the code is translated, the VHDL compiler will generate a combinational circuit that implements the functionality expressed in the FOR-LOOP.

When the code in Figure A.24 is translated by the VHDL compiler, it produces the circuit with 2 two-bit adders shown in Figure A.25. It is possible to see how this circuit corresponds to the FOR-LOOP in the code. The result of the first iteration of the loop is that *Count* is set to the value of $X(1)$. The second iteration then adds $X(1)$ to $X(2)$. This is realized by the top adder in the figure. The third iteration adds $X(3)$ to the sum produced from the second iteration. This corresponds to the bottom adder. When this circuit is optimized by the logic synthesis algorithms, the resulting expressions for *Count* are

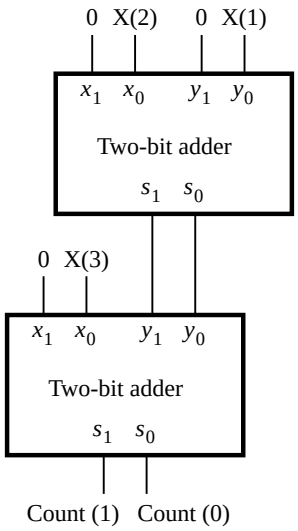


Figure A.25 The circuit generated from the code in Figure A.24.

$$\text{Count(1)} = \text{X(1)X(2)} + \text{X(1)X(3)} + \text{X(2)X(3)}$$
$$\text{Count(0)} = \text{X(1)} \oplus \text{X(2)} \oplus \text{X(3)}$$

These expressions represent a full-adder circuit, with *Count*(0) as the sum output and *Count*(1) as the carry-out. It is interesting to note that even though the VHDL code describes the desired behavior of the circuit in an abstract way, using a FOR-LOOP, in this example the logic synthesis algorithms produce the most efficient circuit, which is the full-adder. As we said at the beginning of this appendix and in section 2.9, the style of code in Figure A.24 should be avoided, because it is often difficult for the designer to envisage what logic circuit the code represents.

As another example of the use of a variable, Figure A.26 gives the code for an *n*-bit NAND gate entity, named *NANDn*. The number of inputs to the NAND gate is set by the GENERIC parameter *n*. The inputs are the *n*-bit signal *X*, and the output is *f*. The variable *Tmp* is defined in the architecture and originally set to the value of the input signal *X*(1). In the FOR LOOP, *Tmp* is ANDed successively with input signals *X* (2) to *X* (*n*). Since *Tmp* is a variable data object, assignments to it take effect immediately; they are not scheduled to take effect at the end of the process. The complement of *Tmp* is assigned to *f*, thus completing the description of the *n*-input NAND operation.

Figure A.27 shows the same code given in Figure A.26 but with the data object *Tmp* defined as a signal, instead of as a variable. This code gives a wrong result, because only the last statement included in the process has any effect on *Tmp*. The code results in $Tmp = Tmp \cdot X(4)$, as determined by the last iteration of the FOR LOOP. Also, since *Tmp* is never initialized, its value is unknown. Hence the value of the output $f = \overline{Tmp}$ is unknown.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY NANDn IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( X  : IN   STD_LOGIC_VECTOR(1 TO n) ;
           f  : OUT  STD_LOGIC ) ;
END NANDn ;

ARCHITECTURE Behavior OF NANDn IS
BEGIN
    PROCESS ( X )
        VARIABLE Tmp : STD_LOGIC ;
    BEGIN
        Tmp := X(1) ;
        AND_bits: FOR i IN 2 TO n LOOP
            Tmp := Tmp AND X(i) ;
        END LOOP AND_bits ;
        f <= NOT Tmp ;
    END PROCESS ;
END Behavior ;
```

Figure A.26 Using a variable to describe an *n*-input NAND gate.

Figure A.28 shows one way to describe the *n*-input NAND gate using signals. Here *Tmp* is defined as an *n*-bit signal, which is set to contain *n* 1s using the (OTHERS => '1') construct. The conditional signal assignment specifies that *f* is 0 only if all bits in the input *X* are 1, thus describing the NAND operation.

A final example of variables used in a sequential circuit is given in section A.10.8. In general, using both variables and signals in VHDL code can lead to confusion because they imply different semantics. Since variables do not necessarily represent wires in a circuit, the meaning of code that uses variables is sometimes ill defined. To avoid confusion, in this book we use variables only for the loop indices in FOR GENERATE and FOR LOOP statements. Except for similar purposes, the reader should avoid using variables because they are not needed for describing logic circuits.

A.10 SEQUENTIAL CIRCUITS

Although combinational circuits can be described using either concurrent or sequential assignment statements, sequential circuits can be described only with sequential assignment statements. We now give some representative examples of sequential circuits.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY NANDn IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( X   : IN     STD_LOGIC_VECTOR(1 TO n) ;
           f   : OUT  STD_LOGIC ) ;
END NANDn ;

ARCHITECTURE Behavior OF NANDn IS
    SIGNAL Tmp : STD_LOGIC ;
BEGIN
    PROCESS ( X )
    BEGIN
        Tmp <= X(1) ;
        AND_bits: FOR i IN 2 TO n LOOP
            Tmp <= Tmp AND X(i) ;
        END LOOP AND_bits ;
        f <= NOT Tmp ;
    END PROCESS ;
END Behavior ;
```

Figure A.27 The code from Figure A.26 using a signal.

A.10.1 A GATED D LATCH

Figure A.29 gives the code for a gated D latch. The process sensitivity list includes both the latch's data input, *D*, and clock, *clk*. Hence whenever a change occurs in the value of either *D* or *clk*, the process becomes active. The IF statement specifies that Q should be set to the value of *D* whenever the clock is 1. There is no ELSE clause in the IF statement. As we explained for Figure A.22, this implies that Q should retain its present value when the IF condition is not met.

A.10.2 D FLIP-FLOP

Figure A.30 gives a process that is slightly different from the one in Figure A.29. The sensitivity list includes only the *Clock* signal, which means that the process is active only when the value of *Clock* changes. The condition in the IF statement looks unusual. The syntax Clock'EVENT represents a *change* in the value of the clock signal. In VHDL jargon 'EVENT is called an *attribute*, and combining 'EVENT with a signal name, such as *Clock*, yields a logical condition. The combination in the IF statement of the two conditions Clock'EVENT and Clock = '1' specifies that Q should be assigned the value of *D* when "a change occurs in the value of *Clock*, and *Clock* is now 1". This describes a low-to-high transition of the clock signal; hence the code describes a positive-edge-triggered D flip-flop.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY NANDn IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( X  : IN     STD_LOGIC_VECTOR(1 TO n) ;
           f  : OUT  STD_LOGIC ) ;
END NANDn ;

ARCHITECTURE Behavior OF NANDn IS
    SIGNAL Tmp : STD_LOGIC_VECTOR(1 TO n) ;
BEGIN
    Tmp <= (OTHERS => '1') ;
    f <= '0' WHEN X = Tmp ELSE '1' ;
END Behavior ;
```

Figure A.28 Using a signal to describe an *n*-input NAND gate.

The *std_logic_1164* package defines the two functions named *rising_edge* and *falling_edge*. They can be used as a short-form notation for the condition that checks for the occurrence of a clock edge. In Figure A.30 we could replace the line "IF Clock'EVENT AND Clock = '1' THEN" with the equivalent line "IF rising_edge(Clock) THEN". We do not use *rising_edge* or *falling_edge* in this book; they are mentioned for completeness.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY latch IS
    PORT ( D, clk  : IN     STD_LOGIC ;
           Q       : OUT  STD_LOGIC ) ;
END latch ;

ARCHITECTURE Behavior OF latch IS
BEGIN
    PROCESS ( D, clk )
    BEGIN
        IF clk = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.29 A gated D Latch.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Clock  : IN   STD_LOGIC ;
           Q         : OUT  STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Clock )
    BEGIN
        IF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.30 D flip-flop.

A.10.3 USING A WAIT UNTIL STATEMENT

The process in Figure A.31 uses a different syntax to describe a D flip-flop. Synchronization with the clock edge is specified using the statement "WAIT UNTIL Clock = '1' ;". This statement should be read as "wait for the next positive-edge of the clock signal." A process that uses a WAIT UNTIL statement is a special case because the sensitivity list is omitted. Use of this WAIT UNTIL statement implicitly specifies that the sensitivity list includes only *Clock*. For our purposes, which is using VHDL for synthesis of circuits, a process can include a WAIT UNTIL statement only if it is the first statement in the process.

A.10.4 A FLIP-FLOP WITH ASYNCHRONOUS RESET

Figure A.32 gives a process that is similar to the one in Figure A.30. It describes a D flip-flop with an asynchronous reset, or clear, input. The reset signal has the name *Resetn*. When $Resetn = 0$, the flip-flop output Q is set to 0. Appending the letter *n* to a signal name is a widely used convention to denote an active-low signal.

A.10.5 SYNCHRONOUS RESET

Figure A.33 shows how a flip-flop with a synchronous reset input can be described.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Clock  : IN     STD_LOGIC ;
           Q         : OUT  STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock = '1' ;
        Q <= D ;
    END PROCESS ;
END Behavior ;
```

Figure A.31 Equivalent code to Figure A.30, using a WAIT UNTIL statement.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Resetn, Clock  : IN     STD_LOGIC ;
           Q                 : OUT  STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.32 D flip-flop with asynchronous reset.

A.10.6 INSTANTIATING A FLIP-FLOP FROM A LIBRARY

Because flip-flops are widely used in logic circuits, most CAD systems provide an assortment of flip-flop components that can be instantiated in VHDL code. An example of this is provided in Figure A.34. It uses a package named *maxplus2* in the library called *altera*. The maxplus2 package is part of the MAX+plusII system and includes many types of basic circuit elements. Figure A.34 instantiates the component named *dff*, which is a D flip-flop declared in the *maxplus2* package. The documentation provided in MAX+plusII specifies that the *dff* component has active-low asynchronous reset and preset inputs.

A.10.7 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.35. It gives the same code shown in Figure A.32 but using the four-bit STD_LOGIC_VECTOR input *D* and the four-bit output Q. The code describes a four-bit register with asynchronous clear.

Figure A.36 gives the code for an entity named *regn*. It shows how the code in Figure A.35 can be extended to represent an *n*-bit register. The number of flip-flops is set by the generic parameter *n*.

The code in Figure A.37 shows how an enable input can be added to the *n*-bit register from Figure A.36. When the active clock edge occurs, the flip-flops in the register cannot

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT ( D, Resetn, Clock  : IN   STD_LOGIC ;
           Q                 : OUT  STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock = '1' ;
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSE
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.33 D flip-flop with synchronous reset.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
LIBRARY altera ;
USE altera.maxplus2.all ;

ENTITY flipflop IS
    PORT ( D, Clock          : IN   STD_LOGIC ;
           Resetn, Presetn   : IN   STD_LOGIC ;
           Q                 : OUT  STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    Dff_instance: Dff PORT MAP (
        D, Clock, Resetn, Presetn, Q ) ;
END Behavior ;
```

Figure A.34 Instantiating a D flip-flop component.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY reg4 IS
    PORT ( D              : IN   STD_LOGIC_VECTOR(3 DOWNTO 0) ;
           Resetn, Clock  : IN   STD_LOGIC ;
           Q              : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END reg4 ;

ARCHITECTURE Behavior OF reg4 IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= "0000" ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.35 Code for a four-bit register with asynchronous clear.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regn IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( D              : IN   STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
           Resetn, Clock  : IN   STD_LOGIC ;
           Q              : OUT  STD_LOGIC_VECTOR(n−1 DOWNTO 0) ) ;
END regn ;

ARCHITECTURE Behavior OF regn IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= (OTHERS => '0') ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.36 Code for an *n*-bit register with asynchronous clear.

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.

A.10.8 SHIFT REGISTERS

An example of code that defines a four-bit shift register is shown in Figure A.38. 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 (4), and the left-most bit is Q (1); shifting is performed in the right-to-left direction. The architecture declares the signal *Sreg*, which is used to describe the shift operation. All assignments to *Sreg* are synchronized to the clock edge by the IF condition; hence *Sreg* represents the outputs of flip-flops. The statement in line 13 specifies that *Sreg*(4) is assigned the value of *w*. As we explained previously, this assignment does not take effect immediately but is scheduled to occur at the end of the process. In line 14 the current value of *Sreg*(4), before it is shifted as a result of line 13, is assigned to *Sreg*(3). Lines 15 and 16 complete the shift operation. They assign the current values of *Sreg*(3) and *Sreg*(2), before they are changed as a result of lines 14 and 15, to *Sreg*(2) and *Sreg*(1), respectively. Finally, *Sreg* is assigned to the Q outputs.

The key point that has to be appreciated in the code in Figure A.38 is that the assignment statements in lines 13 to 16 do not take effect until the end of the process. Hence all flip-

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regne IS
    GENERIC ( n : INTEGER := 4 ) ;
    PORT ( D          : IN   STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
           Resetn     : IN   STD_LOGIC ;
           E, Clock   : IN   STD_LOGIC ;
           Q          : OUT  STD_LOGIC_VECTOR(n−1 DOWNTO 0) ) ;
END regne ;

ARCHITECTURE Behavior OF regne IS
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= (OTHERS => '0') ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            IF E = '1' THEN
                Q <= D ;
            END IF ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Figure A.37 VHDL code for an *n*-bit register with an enable input.

flops change their values at the same time, as required in the shift register. We could write the statements in lines 13 to 16 in any order without changing the meaning of the code.

In section A.9.7 we introduced variables and showed how they differ from signals. As another example of the semantics involved using variables, Figure A.39 gives the code from Figure A.38 but with *Sreg* declared as a variable, instead of as a signal. The statement in line 13 assigns the value of *w* to *Sreg* (4). Since *Sreg* is a variable, the assignment takes effect immediately. In line 14 the value of *Sreg* (4), which has already been changed to *w*, is assigned to *Sreg* (3). Hence line 14 results in *Sreg* (3) = *w*. Similarly, lines 15 and 16 set *Sreg* (2) and *Sreg* (1) to the value of *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.39 to correctly describe a shift register, the ordering of lines 13 to 16 has to be reversed. Then the first assignment sets *Sreg* (1) to the value of *Sreg* (2), the second sets *Sreg* (2) to the value of *Sreg* (3), and so on. Each successive assignment is not affected by the one that precedes it; hence the semantics of using variables does not cause a problem. As we said in section A.9.7, it can be confusing to use both signals and variables at the same time because they imply different semantics.

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY shift4 IS
4       PORT ( w, Clock  : IN   STD_LOGIC ;
5              Q         : OUT  STD_LOGIC_VECTOR(1 TO 4) ) ;
6   END shift4 ;

7   ARCHITECTURE Behavior OF shift4 IS
8       SIGNAL Sreg : STD_LOGIC_VECTOR(1 TO 4) ;
9   BEGIN
10      PROCESS ( Clock )
11      BEGIN
12          IF Clock'EVENT AND Clock = '1' THEN
13              Sreg(4) <= w ;
14              Sreg(3) <= Sreg(4) ;
15              Sreg(2) <= Sreg(3) ;
16              Sreg(1) <= Sreg(2) ;
17          END IF ;
18      END PROCESS ;
19      Q <= Sreg ;
20  END Behavior ;
```

Figure A.38 Code for a four-bit shift register.

A.10.9 COUNTERS

Figure A.40 shows the code for a four-bit counter with an asynchronous reset input. The counter also has an enable input. On the positive clock edge, if the enable E is 1, the count is incremented. If $E = 0$, the counter holds its current value. Because counters are commonly needed in logic circuits, most CAD systems provide a selection of counters that can be instantiated in a design. For example, MAX+plusII provides the counter defined by the LPM standard, which is a variable-width counter with options for enabling the counter, resetting the count to 0, and presetting the count to a specific number.

A.10.10 USING SUBCIRCUITS WITH GENERIC PARAMETERS

We have shown several examples of VHDL entities that include generic parameters. When these subcircuits are used as components in other code, the generic parameters can be set to whatever values are needed. To give an example of component instantiation using generics, consider the circuit shown in Figure A.41. The circuit adds the binary number represented by the k-bit input X to itself a number of times. Such a circuit is often called an *accumulator*. To store the result of each addition operation, the circuit includes a k-bit register. The register has an asynchronous reset input, *Resetn*. It also has an enable input,

```
1   LIBRARY ieee ;
2   USE ieee.std_logic_1164.all ;

3   ENTITY shift4 IS
4       PORT ( w, Clock  : IN     STD_LOGIC ;
5              Q         : OUT  STD_LOGIC_VECTOR(1 TO 4) ) ;
6   END shift4 ;

7   ARCHITECTURE Behavior OF shift4 IS
8   BEGIN
9       PROCESS ( Clock )
10          VARIABLE Sreg : STD_LOGIC_VECTOR(1 TO 4) ;
11      BEGIN
12          IF Clock'EVENT AND Clock = '1' THEN
13              Sreg(4) := w ;
14              Sreg(3) := Sreg(4) ;
15              Sreg(2) := Sreg(3) ;
16              Sreg(1) := Sreg(2) ;
17          END IF ;
18          Q <= Sreg ;
19      END PROCESS ;
20  END Behavior ;
```

Figure A.39 The code from Figure A.38, using a variable.

E, which is controlled by a four-bit counter. The counter has an asynchronous clear input and a count enable input. The circuit operates by first clearing all bits in the register and counter to 0. Then in each clock cycle, the counter is incremented, and the sum outputs from the adder are stored in the register. When the counter reaches the value 1111, the enable inputs on both the register and counter are set to 0 by the NAND gate. Hence the circuit remains in this state until it is reset again. The final value stored in the register is equal to 15*X*.

We can represent the accumulator circuit using several subcircuits described in this appendix: *addern* (Figure A.15), *NANDn* (Figure A.28), *regne*, and *count4*. We placed the component declaration statements for all of these subcircuits in one package, named *components*, which is shown in Figure A.42.

Complete code for the accumulator is given in Figure A.43. It uses the generic parameter *k* to represent the number of bits in the input *X*. Using this parameter in the code makes it easy to change the bit-width at a later time if desired. The architecture 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 can be represented using *k* bits. The four-bit signal *C* represents the outputs from the counter. The *Stop* signal is connected to the enable inputs on the register and counter.

The statement labeled *adder* instantiates the *addern* subcircuit. The GENERIC MAP keywords are used to specify the value of the adder's generic parameter, *n*. The syntax

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY count4 IS
    PORT ( Resetn     : IN     STD_LOGIC ;
           E, Clock   : IN     STD_LOGIC ;
           Q          : OUT  STD_LOGIC_VECTOR (3 DOWNTO 0) ) ;
END count4 ;

ARCHITECTURE Behavior OF count4 IS
    SIGNAL Count : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
BEGIN
    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            Count <= "0000" ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            IF E = '1' THEN
                Count <= Count + 1 ;
            END IF ;
        END IF ;
    END PROCESS ;
    Q <= Count ;
END Behavior ;
```

Figure A.40 An example of a counter.

(n => k) sets the number of bits in the adder to *k*. We do not need the carry-in port on the adder, but a signal must be connected to it. The signal *Zero_bit*, which is set to '0' in the code, is used as a placeholder for the carry-in port (the VHDL syntax does not permit a constant value, such as '1', to be associated directly with a port; hence a signal must be defined for this purpose). The *k*-bit data inputs to the adder are *X* and the output of the register, which is named *Result*. The sum output from the adder is named *Sum*, and the carry-out, which is not used in the circuit, is named *Cout*.

The *regne* subcircuit is instantiated in the statement labeled *reg*. GENERIC MAP is used to set the number of bits in the register to *k*. The *k*-bit register input is provided by the *Sum* output from the adder. The register's output is named *Result*; this signal represents the output of the accumulator circuit. It has the mode BUFFER in the entity declaration. This is required in the VHDL syntax for the signal to be connected to a port on an instantiated component.

The *count4* and *NANDn* components are instantiated in the statements labeled *Counter* and *NANDgate*. We do not have to use the GENERIC MAP keyword for *NANDn*, because the default value of its generic parameter is 4, which is the value needed in this application.

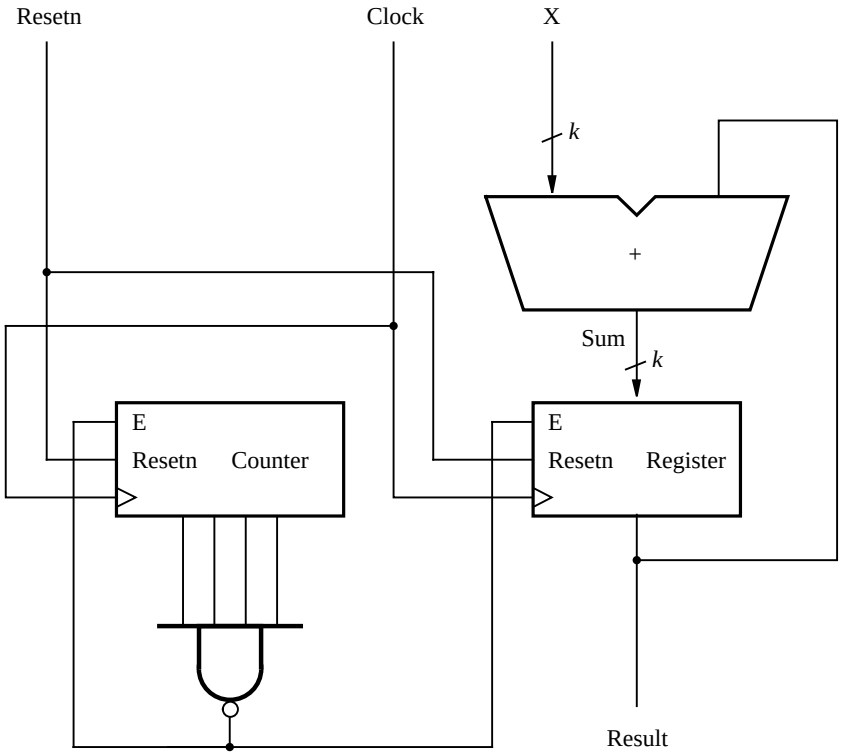


Figure A.41 The accumulator circuit.

A.10.11 A MOORE-TYPE FINITE STATE MACHINE

Figure A.44 shows the state diagram of a simple Moore machine. The code for this machine is shown in Figure A.45. The signal named *y* represents the state of the machine. It is declared with an enumerated type, *State_type*, that has the three possible values A, B, and C. When the code is compiled, the VHDL compiler automatically performs a state assignment to select appropriate bit patterns for the three states. The behavior of the machine is defined by the process with the sensitive list that comprises the reset and clock signals.

The VHDL code includes an asynchronous reset input that puts the machine in state A. The state table for the machine is defined using a CASE statement. Each WHEN clause corresponds to a present state of the machine, and the IF statement inside the WHEN clause specifies the next state to be reached after the next positive edge of the clock signal. Since the machine is of the Moore type, the output *z* can be defined as a separate concurrent assignment statement that depends only on the present state of the machine. Alternatively, the appropriate value for *z* could have been specified within each WHEN clause of the CASE statement.

An alternative way to describe a Moore-type finite state machine is given in the architecture in Figure A.46. Two signals are used to describe how the machine moves from one

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE components IS

    COMPONENT addern - - n-bit adder
        GENERIC ( n : INTEGER := 4 ) ;
        PORT ( Cin    : IN     STD_LOGIC ;
               X, Y   : IN     STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
               S      : OUT    STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
               Cout   : OUT    STD_LOGIC ) ;
    END COMPONENT ;

    COMPONENT regne - - n-bit register with enable
        GENERIC ( n : INTEGER := 4 ) ;
        PORT ( D          : IN     STD_LOGIC_VECTOR(n−1 DOWNTO 0) ;
               Resetn     : IN     STD_LOGIC ;
               E, Clock   : IN     STD_LOGIC ;
               Q          : OUT    STD_LOGIC_VECTOR(n−1 DOWNTO 0) ) ;
    END COMPONENT ;

    COMPONENT count4 - - 4-bit counter with enable
        PORT ( Resetn     : IN     STD_LOGIC ;
               E, Clock   : IN     STD_LOGIC ;
               Q          : OUT    STD_LOGIC_VECTOR (3 DOWNTO 0) ) ;
    END COMPONENT ;

    COMPONENT NANDn - - n-bit AND gate
        GENERIC ( n : INTEGER := 4 ) ;
        PORT ( X   : IN     STD_LOGIC_VECTOR(1 TO n) ;
               f   : OUT    STD_LOGIC ) ;
    END COMPONENT ;

END components ;
```

Figure A.42 Component declarations for the accumulator circuit.

state to another state. The signal *y_present* represents the outputs of the state flip-flops, and the signal *y_next* represents the inputs to the flip-flops. The code has two processes. The top process describes a combinational circuit. It uses a CASE statement to specify the values that *y_next* should have for each value of *y_present*. The other process represents a sequential circuit, which specifies that *y_present* is assigned the value of *y_next* on the positive clock edge. The process also specifies that *y_present* should take the value *A* when *Resetn* is 0, which provides the asynchronous reset.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.components.all ;

ENTITY accum IS
    GENERIC ( k : INTEGER := 8 ) ;
    PORT ( Resetn, Clock  : IN      STD_LOGIC ;
           X              : IN      STD_LOGIC_VECTOR(k-1 DOWNTO 0) ;
           Result         : BUFFER  STD_LOGIC_VECTOR(k-1 DOWNTO 0) ) ;
END accum ;

ARCHITECTURE Structure OF accum IS
    SIGNAL Sum : STD_LOGIC_VECTOR(k-1 DOWNTO 0) ;
    SIGNAL C : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
    SIGNAL Zero_bit, Cout, Stop : STD_LOGIC ;
BEGIN
    Zero_bit <= '0' ;
    adder: addern
        GENERIC MAP ( n => k )
        PORT MAP ( Zero_bit, X, Result, Sum, Cout ) ;
    reg: regne
        GENERIC MAP ( n => k )
        PORT MAP ( Sum, Resetn, Stop, Clock, Result ) ;
    Counter: count4
        PORT MAP ( Clock, Resetn, Stop, C ) ;
    NANDgate: NANDn
        PORT MAP ( C, Stop ) ;
END Structure ;
```

Figure A.43 Code for the accumulator circuit.

Although Figures A.45 and A.46 provide functionally equivalent code, when using the MAX+plusII CAD system, the code in Figure A.45 is preferable. MAX+plusII recognizes the code in Figure A.45 as a finite state machine. It reports all results of synthesizing or simulating the code in terms of the states of the machine. For example, when using the simulator CAD tool, the value of the *y* signal is reported using the names A, B, and C. If the code in Figure A.46 is used instead, then MAX+plusII reports only the logic values of the signals. For example, the value of the *y_present* signal is shown by the simulator as 00, or 01, and so on.

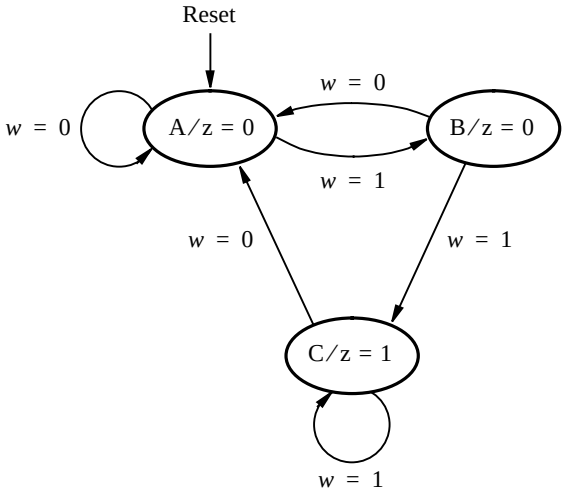


Figure A.44 State diagram of a simple Moore-type FSM.

A.10.12 A MEALY-TYPE FINITE STATE MACHINE

A state diagram for a simple Mealy machine is shown in Figure A.47. The corresponding code is given in Figure A.48. The code is the same as in Figure A.45 except that the output *z* is specified using a separate CASE statement. The CASE statement states that when the FSM is in state *A*, *z* should be 0, but when in state *B*, *z* should take the value of *w*. This CASE statement properly describes the logic needed for *z*. However, it is not obvious why we have used a second CASE statement in the code, rather than specify the value of *z* inside the CASE statement that defines the state table for the machine. This approach would not work properly because the CASE statement for the state table is nested inside the IF statement that waits for a clock edge to occur. Hence if we placed the code for *z* inside this CASE statement, then the value of *z* could change only as a result of a clock edge. This does not meet the requirements of the Mealy-type FSM, because the value of *z* depends not only on the state of the machine but also on the value of the input *w*.

A.10.13 MANUAL STATE ASSIGNMENT FOR A FINITE STATE MACHINE

Instead of having the VHDL compiler determine the state assignment, it is possible to encode the state bits manually. One way to do this in the MAX+plusII system is to use an ATTRIBUTE specification. An attribute provides information about a VHDL element, such as a type. An example showing how an attribute is used for a finite state machine is given in Figure A.49. The code represents the Moore machine from Figure A.45 with the addition of two ATTRIBUTE specifications. The attributes specify that the state encoding should be 00 for state A, 01 for state B, and 11 for state C.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY moore IS
    PORT ( Clock   : IN   STD_LOGIC ;
           w       : IN   STD_LOGIC ;
           Resetn  : IN   STD_LOGIC ;
           z       : OUT  STD_LOGIC ) ;
END moore ;

ARCHITECTURE Behavior OF moore IS
    TYPE State_type IS (A, B, C) ;
    SIGNAL y : State_type ;
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN A =>
                    IF w = '0' THEN
                        y <= A ;
                    ELSE
                        y <= B ;
                    END IF ;
                WHEN B =>
                    IF w = '0' THEN
                        y <= A ;
                    ELSE
                        y <= C ;
                    END IF ;
                WHEN C =>
                    IF w = '0' THEN
                        y <= A ;
                    ELSE
                        y <= C ;
                    END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;

    z <= '1' WHEN y = C ELSE '0' ;
END Behavior ;
```

Figure A.45 An example of a Moore-type finite state machine.

```
ARCHITECTURE Behavior OF moore IS
    TYPE State_type IS (A, B, C) ;
    SIGNAL y_present, y_next : State_type ;
BEGIN
    PROCESS ( w, y_present )
    BEGIN
        CASE y_present IS
            WHEN A =>
                IF w = '0' THEN
                    y_next <= A ;
                ELSE
                    y_next <= B ;
                END IF ;
            WHEN B =>
                IF w = '0' THEN
                    y_next <= A ;
                ELSE
                    y_next <= C ;
                END IF ;
            WHEN C =>
                IF w = '0' THEN
                    y_next <= A ;
                ELSE
                    y_next <= C ;
                END IF ;
        END CASE ;
    END PROCESS ;

    PROCESS ( Clock, Resetn )
    BEGIN
        IF Resetn = '0' THEN
            y_present <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            y_present <= y_next ;
        END IF ;
    END PROCESS ;

    z <= '1' WHEN y_present = C ELSE '0' ;
END Behavior ;
```

Figure A.46 Code equivalent to Figure A.45, using two processes.

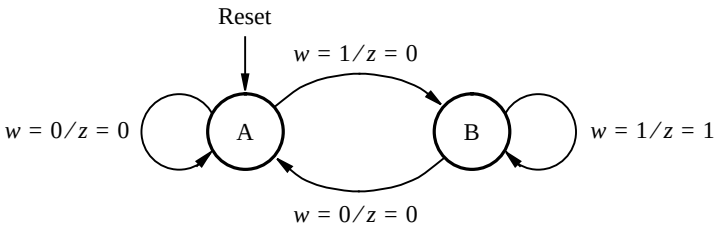


Figure A.47 State diagram of a Mealy-type FSM.

A.11 COMMON ERRORS IN VHDL CODE

This section lists some common errors that our students have made when writing VHDL code.

ENTITY and ARCHITECTURE Names

The name used in an ENTITY declaration and the corresponding ARCHITECTURE must be identical. The code

```
ENTITY adder IS
.
.
.
END adder ;

ARCHITECTURE Structure OF adder4 IS
.
.
.
END Structure ;
```

is erroneous because the ENTITY declaration uses the name *adder*, whereas the architecture uses the name *adder4*.

Missing Semicolon

Every VHDL statement must end with a semicolon.

Use of Quotes

Single quotes are used for single-bit data, double quotes for multibit data, and no quotes are used for integer data. Examples are given in section A.2.

Combinational versus Sequential Statements

Combinational statements include simple signal assignments, selected signal assignments, and generate statements. Simple signal assignments can be used either outside or inside a PROCESS statement. The other types of combinational statements can be used only outside a PROCESS statement.

Sequential statements include IF, CASE, and LOOP statements. Each of these types of statements can be used only inside a process statement.

Component Instantiation

The following statement contains two errors

```
control: shiftr GENERIC MAP ( K => 3 ) ;
                PORT MAP ( '1', Clock, w, Q ) ;
```

There should be no semicolon at the end of the first line, because the two lines represent a single VHDL statement. Also, it is illegal to associate a constant value ('1') with a port on a component. The following code shows how the two errors can be fixed

```
SIGNAL High ;
:
High <= '1' ;
control: shiftr GENERIC MAP ( K => 3 )
                PORT MAP ( High, Clock, w, Q ) ;
```

Label, Signal, and Variable Names

It is illegal to use any VHDL keyword as a label, signal, or variable name. For example, it is illegal to name a signal *In* or *Out*. Also, it is illegal to use the same name multiple times for any label, signal, or variable in a given VHDL design. A common error is to use the same name for a signal and a variable used as the index in a generate or loop statement. For instance, if the code uses the generate statement

```
Generate_label:
FOR i IN 0 TO 3 GENERATE
    bit: fulladd PORT MAP ( C(i), X(i), Y(i), S(i), C(i+1) ) ;
END GENERATE ;
```

then it is illegal to define a signal named *i* (or *I*, because VHDL does not distinguish between lower and uppercase letters).

Implied Memory

As shown in section A.10, implied memory is used to describe storage elements. Care must be taken to avoid unintentional implied memory. The code

```
IF LA = '1' THEN
    EA <= '1' ;
END IF ;
```

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

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mealy IS
    PORT ( Clock, Resetn  : IN    STD_LOGIC ;
           w              : IN    STD_LOGIC ;
           z              : OUT   STD_LOGIC ) ;
END mealy ;

ARCHITECTURE Behavior OF mealy IS
    TYPE State_type IS (A, B) ;
    SIGNAL y : State_type ;
BEGIN
    PROCESS ( Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            y <= A ;
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            CASE y IS
                WHEN A =>
                    IF w = '0' THEN y <= A ;
                    ELSE y <= B ;
                    END IF ;
                WHEN B =>
                    IF w = '0' THEN y <= A ;
                    ELSE y <= B ;
                    END IF ;
            END CASE ;
        END IF ;
    END PROCESS ;

    PROCESS ( y, w )
    BEGIN
        CASE y IS
            WHEN A =>
                z <= '0' ;
            WHEN B =>
                z <= w ;
        END CASE ;
    END PROCESS ;
END Behavior ;
```

Figure A.48 An example of a Mealy-type machine.

```
ARCHITECTURE Behavior OF moore IS
    TYPE State_type IS (A, B, C) ;
    ATTRIBUTE ENUM_ENCODING                 : STRING ;
    ATTRIBUTE ENUM_ENCODING OF State_type  : TYPE IS "00 01 11" ;
    SIGNAL y_present, y_next                : State_type ;
BEGIN
    · · · etc.
```

Figure A.49 An example of specifying the state assignment manually.

```
IF LA = '1' THEN
    EA <= '1' ;
ELSE
    EA <= '0' ;
END IF ;
```

Implied memory also applies to CASE statements. The statement

```
CASE y IS
    WHEN S1 =>
        EA <= '1' ;
    WHEN S2 =>
        EB <= '1' ;
END CASE ;
```

does not specify the value of the *EA* signal when *y* is not equal to *S*1, and it does not specify the value of *EB* when *y* is not equal to *S*2. To avoid having implied memory for both *EA* and *EB*, these signals should be assigned default values, as in the code

```
EA <= '0' ; EB <= '0' ;
CASE y IS
    WHEN S1 =>
        EA <= '1' ;
    WHEN S2 =>
        EB <= '1' ;
END CASE ;
```

In general, the designer should attempt to write VHDL code that contains as few errors as possible because finding the source of an error can often be difficult.

A.12 CONCLUDING REMARKS

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

REFERENCES

1. Institute of Electrical and Electronics Engineers, "1076-1993 IEEE Standard VHDL Language Reference Manual," 1993.
2. D. L. Perry, *VHDL*, 3rd ed. (McGraw-Hill: New York, 1998).
3. Z. Navabi, *VHDL—Analysis and Modeling of Digital Systems* (McGraw-Hill: New York, 1993).
4. J. Bhasker, *A VHDL Primer* (Prentice-Hall: Englewood Cliffs, NJ, 1995).
5. K. Skahill, *VHDL for Programmable Logic* (Addison-Wesley: Menlo Park, CA, 1996).
6. A. Dewey, *Analysis and Design of Digital Systems with VHDL* (PWS Publishing Co.: Boston, MA, 1997).
7. S. Yalamanchili, *VHDL Starter's Guide* (Prentice-Hall: Upper Saddle River, NJ, 1998).