(d) If A is a 6-bit std_logic_vector and B is a 4-bit std_logic_vector, write concurrent VHDL statements that will add A and B to result in a 6-bit sum and a carry.

(e) Draw a circuit that implements the following VHDL code:

```
signal A, B, C, D: std_logic_vector(1 to 3);
signal E, F, G: std_logic;
-------------------------------------------------------
D <= A when E = '1' else "ZZZ";
D <= B when F = '1' else "ZZZ";
D <= C when G = '1' else "ZZZ";
```

(f) Work Problems 10.6(b), 10.7, and 10.8.

8. Before you take the test on Unit 10, pick up a lab assignment sheet and work the assigned lab problems. Turn in your VHDL code and simulation results.

Introduction to VHDL

As integrated circuit technology has improved to allow more and more components on a chip, digital systems have continued to grow in complexity. As digital systems have become more complex, detailed design of the systems at the gate and flip-flop level has become very tedious and time consuming. For this reason, the use of hardware description languages in the digital design process continues to grow in importance. A hardware description language allows a digital system to be designed and debugged at a higher level before implementation at the gate and flip-flop level. The use of computer-aided design tools to do this conversion is becoming more widespread. This is analogous to writing software programs in a high-level language such as C and then using a compiler to convert the programs to machine language. The two most popular hardware description languages are VHDL and Verilog.

VHDL is a hardware description language that is used to describe the behavior and structure of digital systems. The acronym VHDL stands for VHSIC Hardware Description Language, and VHSIC in turn stands for Very High Speed Integrated Circuit. However, VHDL is a general-purpose hardware description language which can be used to describe and simulate the operation of a wide variety of digital systems, ranging in complexity from a few gates to an interconnection of many complex integrated circuits. VHDL was originally developed to allow a uniform method for specifying digital systems. The VHDL language became an IEEE standard in 1987, and it is widely used in industry. IEEE published a revised VHDL standard in 1993, and the examples in this text conform to that standard.

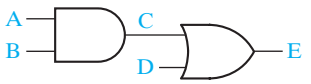
VHDL can describe a digital system at several different levels—behavioral, data flow, and structural. For example, a binary adder could be described at the behavioral level in terms of its function of adding two binary numbers, without giving any implementation details. The same adder could be described at the data flow level by giving the logic equations for the adder. Finally, the adder could be described at the structural level by specifying the interconnections of the gates which make up the adder.

VHDL leads naturally to a top-down design methodology in which the system is first specified at a high level and tested using a simulator. After the system is debugged at this level, the design can gradually be refined, eventually leading to a structural description which is closely related to the actual hardware implementation. VHDL was designed to be technology independent. If a design is described in VHDL and implemented in today's technology, the same VHDL description could be used as a starting point for a design in some future technology.

In this chapter, we introduce VHDL and illustrate how we can describe simple combinational circuits using VHDL. We will use VHDL in later units to design sequential circuits and more complex digital systems. In Unit 17, we introduce the use of CAD software tools for automatic synthesis from VHDL descriptions. These synthesis tools will derive a hardware implementation from the VHDL code.

10.1 VHDL Description of Combinational Circuits

We begin by describing a simple gate circuit using VHDL. A VHDL signal is used to describe a signal in a physical system. (Section 10.4 contains a summary of signals, constants, and types. The VHDL language also includes variables similar to variables in programming languages, but to obtain synthesizable code for hardware, signals should be used to represent hardware signals. VHDL variables are not used in this text.) The gate circuit of Figure 10-1 has five signals: A, B, C, D, and E. The symbol "<=" is the signal assignment operator which indicates that the value computed on




```
C <= A and B after 5 ns;
E <= C or D after 5 ns;
```

FIGURE 10-1
Gate Circuit

the right-hand side is assigned to the signal on the left side. A *behavioral* description of the circuit in Figure 10-1 is

```
E <= D or (A and B);
```

Parentheses are used to specify the order of operator execution.

The two assignment statements in Figure 10-1 give a *dataflow* description of the circuit where it is assumed that each gate has a 5-ns propagation delay. When the statements in Figure 10-1 are simulated, the first statement will be evaluated any time A or B changes, and the second statement will be evaluated any time C or D changes. Suppose that initially A = 1, and B = C = D = E = 0. If B changes to 1 at time 0, C will change to 1 at time = 5 ns. Then, E will change to 1 at time = 10 ns.

The circuit of Figure 10-1 can also be described using *structural* VHDL code. To do so requires that a two-input AND-gate component and a two-input OR-gate component be declared and defined. Components may be declared and defined either in a library or within the architecture part of the VHDL code. (VHDL architectures are discussed in Section 10.3, and packages and libraries are discussed in Section 10.7.) Instantiation statements are used to specify how components are connected. Each copy of a component requires a separate instantiation statement to specify how it is connected to other components and to the port inputs and outputs. An instantiation statement is a concurrent statement that executes anytime one of the input signals in its port map changes. The circuit of Figure 10-1 is described by instantiating the AND gate and the OR gate as follows:

```
Gate1: AND2 port map (A, B, C);
Gate2: OR2 port map (C, D, E);
```

The port map for Gate1 connects A and B to the AND-gate inputs, and it connects D to the AND-gate output. Since an instantiation statement is concurrent, whenever A or B changes, these changes go to the Gate1 inputs, and then the component computes a new value of C. Similarly, the second statement passes changes in C or D to the Gate2 inputs, and then the component computes a new value of E. This is exactly how the real hardware works. (The order in which the instantiation statements appear is irrelevant.) Instantiating a component is different than calling a function in a computer program. A function returns a new value whenever it is called, but an instantiated component computes a new output value whenever its input changes.

VHDL signal assignment statements, such as the ones in Figure 10-1, are examples of concurrent statements. The VHDL simulator monitors the right side of each concurrent statement, and any time a signal changes, the expression on the right side is immediately re-evaluated. The new value is assigned to the signal on the left side

after an appropriate delay. This is exactly the way the hardware works. Any time a gate input changes, the gate output is recomputed by the hardware, and the output changes after the gate delay.

When we initially describe a circuit, we may not be concerned about propagation delays. If we write

```
C <= A and B;
E <= C or D;
```

this implies that the propagation delays are 0 ns. In this case, the simulator will assume an infinitesimal delay referred to as Δ (delta). Assume that initially A = 1 and B = C = D = E = 0. If B is changed to 1 at time = 1 ns, then C will change at time 1 + Δ and E will change at time 1 + 2Δ.

Unlike a sequential program, the order of the above concurrent statements is unimportant. If we write

```
E <= C or D;
C <= A and B;
```

the simulation results would be exactly the same as before.

In general, a signal assignment statement has the form

```
signal_name <= expression [after delay];
```

The expression is evaluated when the statement is executed, and the signal on the left side is scheduled to change after delay. The square brackets indicate that after delay is optional; they are not part of the statement. If after delay is omitted, then the signal is scheduled to be updated after a delta delay. Note that the time at which the statement executes and the time at which the signal is updated are not the same.

Even if a VHDL program has no explicit loops, concurrent statements may execute repeatedly as if they were in a loop. Figure 10-2 shows an inverter with the output connected back to the input. If the output is '0', then this '0' feeds back to the input and the inverter output changes to '1' after the inverter delay, assumed to be 10 ns. Then, the '1' feeds back to the input, and the output changes to '0' after the inverter delay. The signal CLK will continue to oscillate between '0' and '1', as shown in the waveform. The corresponding concurrent VHDL statement will produce the same result. If CLK is initialized to '0', the statement executes and CLK changes to '1' after 10 ns. Because CLK has changed, the statement executes again, and CLK will change back to '0' after another 10 ns. This process will continue indefinitely.

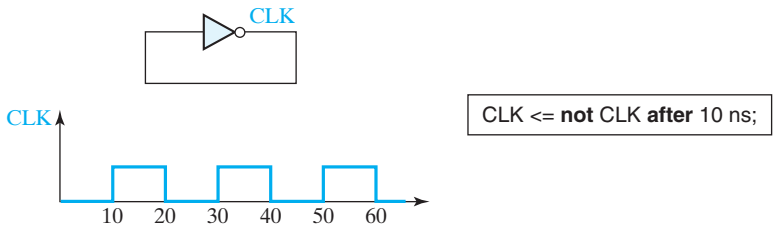


FIGURE 10-2
Inverter with Feedback

The statement in Figure 10-2 generates a clock waveform with a half period of 10 ns. On the other hand, the concurrent statement

```
CLK <= not CLK;
```

will cause a run-time error during simulation. Because there is 0 delay, the value of CLK will change at times 0 + Δ, 0 + 2Δ, 0 + 3Δ, etc. Because Δ is an infinitesimal time, time will never advance to 1 ns.

In general, VHDL is not case sensitive, that is, capital and lower case letters are treated the same by the compiler and the simulator. Thus, the statements

```
Clk <= NOT clk After 10 NS;
```

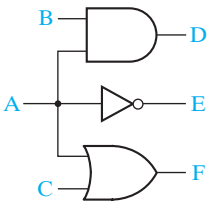
and

```
CLK <= not CLK after 10 ns;
```

would be treated exactly the same. Signal names and other VHDL identifiers may contain letters, numbers, and the underscore character (_). An identifier must start with a letter, and it cannot end with an underscore. Thus, C123 and ab_23 are legal identifiers, but 1ABC and ABC_ are not. Every VHDL statement must be terminated with a semicolon. Spaces, tabs, and carriage returns are treated in the same way. This means that a VHDL statement can be continued over several lines, or several statements can be placed on one line. In a line of VHDL code, anything following a double dash (--) is treated as a comment. Words such as **and**, **or**, and **after** are reserved words (or keywords) which have a special meaning to the VHDL compiler. In this text, we will put all reserved words in boldface type.

Figure 10-3 shows three gates that have the signal A as a common input and the corresponding VHDL code. The three concurrent statements execute simultaneously whenever A changes, just as the three gates start processing the signal change at the same time. However, if the gates have different delays, the gate outputs can change at different times. If the gates have delays of 2 ns, 1 ns, and 3 ns, respectively, and A changes at time 5 ns, then the gate outputs D, E, and F can change at times 7 ns, 6 ns, and 8 ns, respectively. The VHDL statements work in the same way. Even though the statements execute simultaneously, the signals D, E, and F are updated at times 7 ns, 6 ns, and 8 ns. However, if no delays were specified, then D, E, and F would all be updated at time 5 + Δ.

In these examples, every signal is of type bit, which means it can have a value of '0' or '1'. (Bit values in VHDL are enclosed in single quotes to distinguish them from integer values.) In digital design, we often need to perform the same operation on a group of signals. A one-dimensional array of bit signals is referred to as a bit-vector. If a 4-bit vector named B has an index range 0 through 3, then the four elements of the bit-vector are designated B(0), B(1), B(2), and B(3). The statement `B <= "0110"` assigns '0' to B(0), '1' to B(1), '1' to B(2), and '0' to B(3).




```
-- when A changes, these concurrent
-- statements all execute at the same time
D <= A and B after 2 ns;
E <= not A after 1 ns;
F <= A or C after 3 ns;
```

FIGURE 10-3
Three Gates with a Common Input and Different Delays

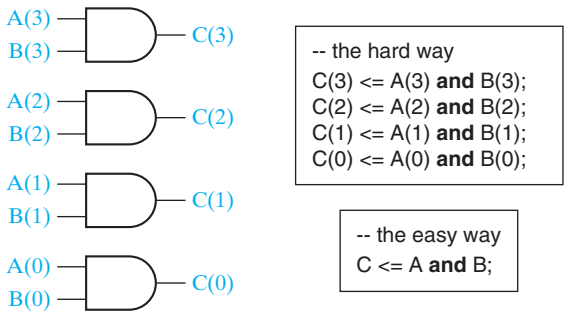


FIGURE 10-4
Array of AND Gates

Figure 10-4 shows an array of four AND gates. The inputs are represented by bit-vectors A and B, and the outputs by bit-vector C. Although we can write four VHDL statements to represent the four gates, it is much more efficient to write a single VHDL statement that performs the **and** operation on the bit-vectors A and B. When applied to bit-vectors, the **and** operator performs the **and** operation on corresponding pairs of elements.

The preceding signal assignment statements containing "**after delay**" create what is called an **inertial** delay model. Consider a device with an inertial delay of D time units. If an input change to the device will cause its output to change, then the output changes D time units later. However, this is not what happens if the device receives two input changes within a period of D time units and both input changes should cause the output to change. In this case the device output does not change in response to either input change. As an example, consider the signal assignment

```
C <= A and B after 10 ns;
```

Assume A and B are initially 1, and A changes to 0 at 15 ns, to 1 at 30 ns, and to 0 at 35 ns. Then C changes to 1 at 10 ns and to 0 at 25 ns, but C does not change in response to the A changes at 30 ns and 35 ns because these two changes occurred less than 10 ns apart. A device with an inertial delay of D time units filters out output changes that would occur in less than or equal to D time units.

VHDL can also model devices with an **ideal** (**transport**) delay. Output changes caused by input changes to a device exhibiting an ideal (transport) delay of D time units are delayed by D time units, and the output changes occur even if they occur within D time units. The VHDL signal assignment statement that models ideal (transport) delay is

```
signal_name <= transport expression after delay
```

As an example, consider the signal assignment

```
C <= transport A and B after 10 ns;
```

Assume A and B are initially 1 and A changes to 0 at 15 ns, to 1 at 30 ns, and to 0 at 35 ns. Then C changes to 1 at 10 ns, to 0 at 25 ns, to 1 at 40 ns, and to 0 at 45 ns. Note that the last two changes are separated by just 5 ns.

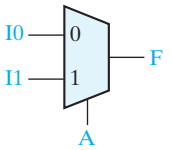
10.2 VHDL Models for Multiplexers

Figure 10-5 shows a 2-to-1 multiplexer (MUX) with two data inputs and one control input. The MUX output is F = A′·I0 + A·I1. The corresponding VHDL statement is

```
F <= (not A and I0) or (A and I1);
```

Alternatively, we can represent the MUX by a conditional signal assignment statement, as shown in Figure 10-5. This statement executes whenever A, I0, or I1 changes. The MUX output is I0 when A = '0', and else it is I1. In the conditional statement, I0, I1, and F can either be bits or bit-vectors.

FIGURE 10-5
2-to-1 Multiplexer



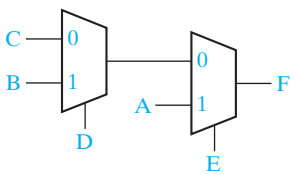

```
-- conditional signal assignment statement
F <= I0 when A = '0' else I1;
```

The general form of a conditional signal assignment statement is

```
signal_name <= expression1 when condition1
               else expression2 when condition2
               [else expressionN];
```

This concurrent statement is executed whenever a change occurs in a signal used in one of the expressions or conditions. If condition1 is true, signal_name is set equal to the value of expression1, or else if condition2 is true, signal_name is set equal to the value of expression2, etc. The line in square brackets is optional. Figure 10-6 shows how two cascaded MUXes can be represented by a conditional signal assignment statement. The output MUX selects A when E = '1'; or else it selects the output of the first MUX, which is B when D = '1', or else it is C.

FIGURE 10-6
Cascaded 2-to-1 MUXes




```
F <= A when E = '1'
     else B when D = '1'
     else C;
```

Figure 10-7 shows a 4-to-1 MUX with four data inputs and two control inputs, A and B. The control inputs select which one of the data inputs is transmitted to the output. The logic equation for the 4-to-1 MUX is

$$F = A'B'I_0 + A'BI_1 + AB'I_2 + ABI_3$$

Thus, one way to model the MUX is with the VHDL statement

```
F <= (not A and not B and I0) or (not A and B and I1) or
     (A and not B and I2) or (A and B and I3);
```

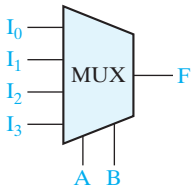


FIGURE 10-7
4-to-1 Multiplexer


```
sel <= A&B;
-- selected signal assignment statement
with sel select
  F <= I0 when "00",
       I1 when "01",
       I2 when "10",
       I3 when "11";
```

Another way to model the 4-to-1 MUX is to use a conditional assignment statement:

```
F <= I0 when A&B = "00"
else I1 when A&B = "01"
else I2 when A&B = "10"
else I3;
```

The expression A&B means A concatenated with B, that is, the two bits A and B are merged together to form a 2-bit vector. This bit vector is tested, and the appropriate MUX input is selected. For example, if A = '1' and B = '0', A&B = "10" and I2 is selected. Instead of concatenating A and B, we could use a more complex condition:

```
F <= I0 when A = '0' and B = '0'
  else I1 when A = '0' and B = '1'
  else I2 when A = '1' and B = '0'
  else I3;
```

A third way to model the MUX is to use a selected signal assignment statement, as shown in Figure 10-7. A&B cannot be used in this type of statement, so we first set Sel equal to A&B. The value of Sel then selects the MUX input that is assigned to F.

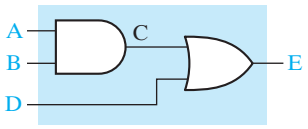
The general form of a selected signal assignment statement is

```
with expression_s select
    signal_s <= expression1 [after delay-time] when choice1,
                expression2 [after delay-time] when choice2,
                . . .
                [expression_n [after delay-time] when others];
```

This concurrent statement executes whenever a signal changes in any of the expressions. First, expression_s is evaluated. If it equals choice1, signal_s is set equal to expression1; if it equals choice2, signal_s is set equal to expression2; etc. If all possible choices for the value of expression_s are given, the last line should be omitted; otherwise, the last line is required. When it is present, if expression_s is not equal to any of the enumerated choices, signal_s is set equal to expression_n. The signal_s is updated after the specified delay-time, or after Δ if the "**after** delay-time" is omitted.

10.3 VHDL Modules

To write a complete VHDL module, we must declare all of the input and output signals using an **entity** declaration, and then specify the internal operation of the module using an **architecture** declaration. As an example, consider Figure 10-8. The entity declaration gives the name "two_gates" to the module. The port declaration specifies the inputs and outputs to the module. A, B, and D are input signals of type bit, and E is an output signal of type bit. The architecture is named "gates". The signal C is declared within the architecture because it is an internal signal. The two concurrent statements that describe the gates are placed between the keywords **begin** and **end**.




```
entity two_gates is
    port (A,B,D: in bit; E: out bit);
end two_gates;
architecture gates of two_gates is
    signal C: bit;
begin
    C <= A and B; -- concurrent
    E <= C or D; -- statements
end gates;
```

FIGURE 10-8
VHDL Module with Two Gates

When we describe a system in VHDL, we must specify an entity and an architecture at the top level, and also specify an entity and architecture for each of the component modules that are part of the system (see Figure 10-9). Each entity declaration includes a list of interface signals that can be used to connect to other modules or to the outside world. We will use entity declarations of the form:

```
entity entity-name is
    [port(interface-signal-declaration);]
end [entity] [entity-name];
```

The items enclosed in square brackets are optional. The interface-signal-declaration normally has the following form:

```
list-of-interface-signals: mode type [: = initial-value]
{; list-of-interface-signals: mode type [: = initial-value]};
```

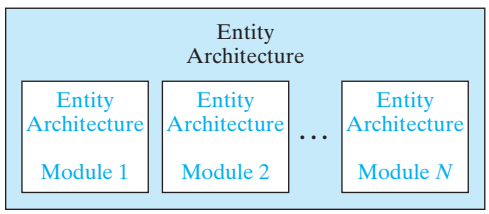


FIGURE 10-9
VHDL Program Structure

The curly brackets indicate zero or more repetitions of the enclosed clause. Input signals are of mode **in**, output signals are of mode **out**, and bi-directional signals (see Figure 9-16) are of mode **inout**.

So far, we have only used type bit and bit_vector; other types are described in Section 10.4. The optional initial-value is used to initialize the signals on the associated list; otherwise, the default initial value is used for the specified type. For example, the port declaration

```
port(A, B: in integer := 2; C, D: out bit);
```

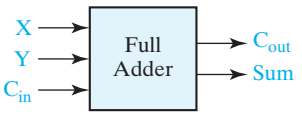
indicates that A and B are input signals of type integer that are initially set to 2, and C and D are output signals of type bit that are initialized by default to '0'.

Associated with each entity is one or more architecture declarations of the form

```
architecture architecture-name of entity-name is
    [declarations]
begin
    architecture body
end [architecture] [architecture-name];
```

In the declarations section, we can declare signals and components that are used within the architecture. The architecture body contains statements that describe the operation of the module.

Next, we will write the entity and architecture for a full adder module (refer to Section 4.7 for a description of a full adder). The entity specifies the inputs and outputs of the adder module, as shown in Figure 10-10. The port declaration specifies that X, Y and Cin are input signals of type bit, and that Cout and Sum are output signals of type bit.




```
entity FullAdder is
    port (X,Y,Cin: in bit;          -- Inputs
          Cout, Sum: out bit);  -- Outputs
end FullAdder;
```

FIGURE 10-10
Entity Declaration for a Full Adder Module

The operation of the full adder is specified by an architecture declaration:

```
architecture Equations of FullAdder is
begin           -- concurrent assignment statements
    Sum <= X xor Y xor Cin after 10 ns;
    Cout <= (X and Y) or (X and Cin) or (Y and Cin) after 10 ns;
end Equations;
```

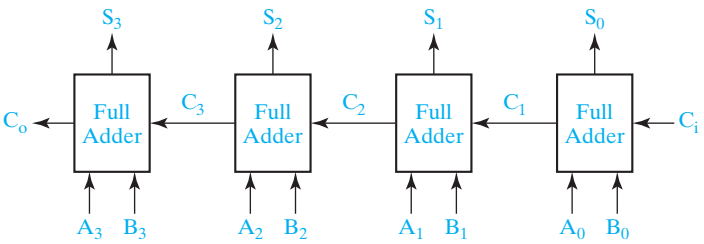
In this example, the architecture name (Equations) is arbitrary, but the entity name (FullAdder) must match the name used in the associated entity declaration.

The VHDL assignment statements for Sum and Cout represent the logic equations for the full adder. Several other architectural descriptions such as a truth table or an interconnection of gates could have been used instead. In the Cout equation, parentheses are required around (X **and** Y) because VHDL does not specify an order of precedence for the logic operators.

Four-Bit Full Adder

Next, we will show how to use the FullAdder module defined above as a component in a system which consists of four full adders connected to form a 4-bit binary adder (see Figure 10-11). We first declare the 4-bit adder as an entity (see Figure 10-12). Because the inputs and the sum output are four bits wide, we declare them as bit_vectors which are dimensioned 3 **downto** 0. (We could have used a range 1 **to** 4 instead.)

FIGURE 10-11
4-Bit Binary Adder



Next, we specify the FullAdder as a component within the architecture of Adder4 (Figure 10-12). The component specification is very similar to the entity declaration for the full adder, and the input and output port signals correspond to those declared for the full adder. Following the component statement, we declare a 3-bit internal carry signal C.

In the body of the architecture, we create several instances of the FullAdder component. (In CAD jargon, we *instantiate* four copies of the FullAdder.) Each copy of FullAdder has a name (such as FA0) and a port map. The signal names following the port map correspond one-to-one with the signals in the component port. Thus, A(0), B(0), and Ci correspond to the inputs X, Y, and Cin, respectively. C(1) and S(0) correspond to the Cout and Sum outputs. Note that the order of the signals in the port map must be the same as the order of the signals in the port of the component declaration.

In preparation for simulation, we can place the entity and architecture for the FullAdder and for Adder4 together in one file and compile. Alternatively, we could compile the FullAdder separately and place the resulting code in a library which is linked in when we compile Adder4.

All of the simulation examples in this text use the ModelSim simulator from Model Tech. Most other VHDL simulators use similar command files and can

FIGURE 10-12
Structural Description of 4-Bit Adder


```
entity Adder4 is
    port (A, B: in bit_vector(3 downto 0); Ci: in bit;    -- Inputs
          S: out bit_vector(3 downto 0); Co: out bit);    -- Outputs
end Adder4;
architecture Structure of Adder4 is
component FullAdder
    port (X, Y, Cin: in bit;    -- Inputs
          Cout, Sum: out bit);    -- Outputs
end component;
signal C: bit_vector(3 downto 1);
begin    -- instantiate four copies of the FullAdder
    FA0: FullAdder port map (A(0), B(0), Ci, C(1), S(0));
    FA1: FullAdder port map (A(1), B(1), C(1), C(2), S(1));
    FA2: FullAdder port map (A(2), B(2), C(2), C(3), S(2));
    FA3: FullAdder port map (A(3), B(3), C(3), Co, S(3));
end Structure;
```

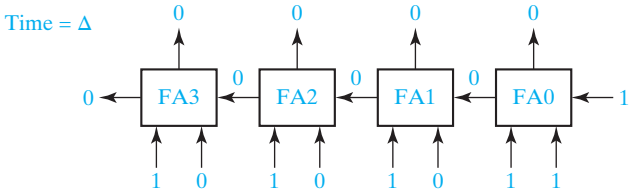
produce output in a similar format. We will use the following simulator commands to test Adder4:

```
add list A B Co C Ci S          -- put these signals on the output list
force A 1111                    -- set the A inputs to 1111
force B 0001                    -- set the B inputs to 0001
force Ci 1                      -- set Ci to 1
run 50 ns                       -- run the simulation for 50 ns
force Ci 0
force A 0101
force B 1110
run 50 ns
```

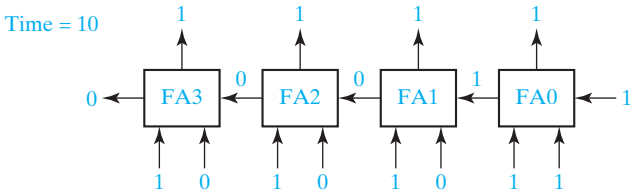
We have chosen to run the simulation for 50 ns because this is more than enough time for the carry to propagate through all of the full adders. The simulation results for the above command list are:

ns	delta	a	b	co	c	ci	s
0	+0	0000	0000	0	000	0	0000
0	+1	1111	0001	0	000	1	0000
10	+0	1111	0001	0	001	1	1111
20	+0	1111	0001	0	011	1	1101
30	+0	1111	0001	0	111	1	1001
40	+0	1111	0001	1	111	1	0001
50	+0	0101	1110	1	111	0	0001
60	+0	0101	1110	1	110	0	0101
70	+0	0101	1110	1	100	0	0111
80	+0	0101	1110	1	100	0	0011

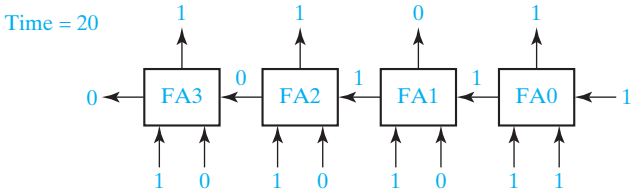
The listing shows how the carry propagates one position every 10 ns. The full adder inputs change at time = Δ:



The sum and carry are computed by each FA and appear at the FA outputs 10 ns later:



Because the inputs to FA1 have changed, the outputs change 10 ns later:



The final simulation results are:

1111 + 0001 + 1 = 0001 with a carry of 1 (at time = 40 ns) and
0101 + 1110 + 0 = 0011 with a carry of 1 (at time = 80 ns)

The simulation stops at 80 ns because no further changes occur after that time. For more details on how the simulator handles Δ delays, refer to Section 10.9.

In this section we have shown how to construct a VHDL module using an entity-architecture pair. The 4-bit adder module demonstrates the use of VHDL components to write structural VHDL code. Components used within the architecture are declared at the beginning of the architecture, using a component declaration of the form

```
component component-name
    port (list-of-interface-signals-and-their-types);
end component;
```

The port clause used in the component declaration has the same form as the port clause used in an entity declaration. The connections to each component used in a circuit are specified by using a component instantiation statement of the form

```
label: component-name port map (list-of-actual-signals);
```

The list of actual signals must correspond one-to-one to the list of interface signals specified in the component declaration.

10.4 Signals and Constants

Input and output signals for a module are declared in a port. Signals internal to a module are declared at the start of an architecture, before **begin**, and can be used only within that architecture. Port signals have an associated mode (usually in or out), but internal signals do not. A signal used within an architecture must be declared either in a port or in the declaration section of an architecture, but it cannot be declared in both places. A signal declaration has the form

```
signal list_of_signal_names: type_name [constraint] [:= initial_value];
```

The constraint can be an index range like (0 **to** 5) or (4 **downto** 1), or it can be a range of values such as **range** 0 to 7. Examples:

```
signal A, B, C: bit_vector(3 downto 0):= "1111";
```

A, B, and C are 4-bit vectors dimensioned 3 downto 0 and initialized to 1111.

```
signal E, F: integer range 0 to 15;
```

E and F are integers in the range 0 to 15, initialized by default to 0. The compiler or simulator will flag an error if we attempt to assign a value outside the specified range to E or F.

Constants declared at the start of an architecture can be used anywhere within that architecture. A constant declaration is similar to a signal declaration:

```
constant constant_name: type_name [constraint] [:= constant_value];
```

A constant named limit of type integer with a value of 17 can be defined as

```
constant limit : integer := 17;
```

A constant named delay1 of type time with the value of 5 ns can be defined as

```
constant delay1 : time := 5 ns;
```

This constant could then be used in an assignment statement

```
A <= B after delay1;
```

Once the value of a constant is defined in a declaration statement, unlike a signal, the value cannot be changed by using an assignment statement.

Signals and constants can have any one of the predefined VHDL types, or they can have a user-defined type. Some of the predefined types are

Definition

bit	'0' or '1'
boolean	FALSE or TRUE
integer	an integer in the range $-(2^{31} - 1)$ to $+ (2^{31} - 1)$ (some implementations support a wider range)
positive	an integer in the range 1 to $2^{31} - 1$ (positive integers)
natural	an integer in the range 0 to $2^{31} - 1$ (positive integers and zero)
real	floating-point number in the range –1.0E38 to + 1.0E38
character	any legal VHDL character including upper- and lower case letters, digits, and special characters; each printable character must be enclosed in single quotes, e.g., 'd', '7', '+'
time	an integer with units fs, ps, ns, us, ms, sec, min, or hr

Note that the integer range for VHDL is symmetrical even though the range for a 32-bit 2's complement integer is -2^{31} to $+ (2^{31} - 1)$.

A common user-defined type is the enumeration type in which all of the values are enumerated. For example, the declarations

```
type state_type is (S0, S1, S2, S3, S4, S5);
signal state : state_type := S1;
```

define a signal called state which can have any one of the values S0, S1, S2, S3, S4, or S5 and which is initialized to S1. If no initialization is given, the default initialization is the left most element in the enumeration list, S0 in this example. If we declare the signal state as shown, the following assignment statement sets state to S3:

```
state <= S3;
```

VHDL is a strongly typed language so signals of different types generally cannot be mixed in the same assignment statement, and no automatic type conversion is performed. Thus the statement A <= B **or** C is only valid if A, B, and C all have the same type or closely related types.

10.5 Arrays

In order to use an array in VHDL, we must first declare an array type, and then declare an array object. For example, the following declaration defines a one-dimensional array type named SHORT_WORD:

```
type SHORT_WORD is array (15 downto 0) of bit;
```

An array of this type has an integer index with a range from 15 downto 0, and each element of the array is of type bit.

Next, we will declare array objects of type SHORT_WORD:

```
signal DATA_WORD: SHORT_WORD;
signal ALT_WORD: SHORT_WORD := "0101010101010101";
constant ONE_WORD: SHORT_WORD := (others => '1');
```

DATA_WORD is a signal array of 16 bits, indexed 15 downto 0, which is initialized (by default) to all '0' bits. ALT_WORD is a signal array of 16 bits which is initialized to alternating 0's and 1's. ONE_WORD is a constant array of 16 bits; all bits are set to '1' by **others** => '1'). Because none of the bits have been set individually,[1] in this case **others** applies to all of the bits.

We can reference individual elements of the array by specifying an index value. For example, ALT_WORD(0) accesses the far right bit of ALT_WORD. We can also specify a portion of the array by specifying an index range: ALT_WORD(5 **downto** 0) accesses the low order six bits of ALT_WORD, which have an initial value of 010101.

The array type and array object declarations illustrated above have the general forms:

```
type array_type_name is array index_range of element_type;
signal array_name: array_type_name [ := initial_values ];
```

In this declaration, **signal** may be replaced with **constant**.

Multidimensional array types may also be defined with two or more dimensions. The following example defines a two-dimensional array signal which is a matrix of integers with four rows and three columns:

```
type matrix4x3 is array (1 to 4, 1 to 3) of integer;
signal matrixA: matrix4x3 := ((1,2,3),(4,5,6),(7,8,9),(10,11,12));
```

The signal matrixA, will be initialized to

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

The array element matrixA(3,2) references the element in the third row and second column, which has a value of 8. The statement B <= matrixA (2,3) assigns a value of 6 to B.

When an array type is declared, the dimensions of the array may be left undefined. This is referred to as an unconstrained array type. For example,

```
type intvec is array (natural range <>) of integer;
```

declares intvec as an array type which defines a one-dimensional array of integers with an unconstrained index range of natural numbers. The default type for array indices is integer, but another type may be specified. Because the index range is not specified in the unconstrained array type, the range must be specified when the array object is declared. For example,

```
signal intvec5: intvec(1 to 5) := (3,2,6,8,1);
```

defines a signal array named intvec5 with an index range of 1 to 5, which is initialized to 3, 2, 6, 8, 1. The following declaration defines matrix as a two-dimensional array with unconstrained row and column index ranges:

```
type matrix is array (natural range <>, natural range <>) of integer;
```

[1] See Reference [1, p. 86] for information on how to set individual bits.

Predefined unconstrained array types in VHDL include bit_vector and string, which are defined as follows:

```
type bit_vector is array (natural range <>) of bit;
type string is array (positive range <>) of character;
```

The characters in a string literal must be enclosed in double quotes. For example, "This is a string." is a string literal. The following example declares a constant string1 of type string:

```
constant string1: string(1 to 29) := "This string is 29 characters."
```

A bit_vector literal may be written either as a list of bits separated by commas or as a string. For example, ('1','0','1','1','0') and "10110" are equivalent forms. The following declares a constant A which is a bit_vector with a range 0 to 5.

```
constant A : bit_vector(0 to 5) := "101011";
```

A truth table can be implemented using a ROM (read-only memory) as illustrated in Figure 9-21. If we represent the ROM outputs by a bit_vector, F(0 **to** 3), we can represent the truth table that is stored in the ROM by an array of bit_vectors. The VHDL code for this ROM is given in Figure 10-13. The port declaration (line 4) defines the inputs and outputs for the ROM. The type declaration (line 7) defines an array with 8 rows where each row is 4 bits wide. Line 8 declares ROM1 to be an array of this type with binary data stored in each row. Line 9 declares an integer called index. This index will be used to select one of the 8 rows in the ROM1 array. In line 11, this index is formed by concatenating the three input bits to form a 3-bit vector, and this vector is converted to an integer. The data is read from the ROM1 array in line 13. For example, if A = '1', B = '0', and C = '1', index = 5, and "0001" is read from the ROM. Lines 1 and 2 allow us to use the vec2int function, which is defined in a library named BITLIB.

FIGURE 10-13 VHDL Description of a ROM

```
1   library BITLIB;
2   use BITLIB.bit_pack.all;
3   entity ROM9_17 is
4       port (A, B, C: in bit; F: out bit_vector(0 to 3));
5   end entity;
6   architecture ROM of ROM9_17 is
7   type ROM8X4 is array (0 to 7) of bit_vector(0 to 3);
8   constant ROM1: ROM8X4 := ("1010", "1010", "0111", "0101", "1100", "0001", "1111", "0101");
9   signal index: Integer range 0 to 7;
10  begin
11      index <= vec2int(A&B&C);      -- A&B&C Is a 3-bit vector
12          -- vec2int is a function that converts this vector to an integer
13      F <= ROM1 (index);
14          -- this statement reads the output from the ROM
15  end ROM;
```

10.6 VHDL Operators

Predefined VHDL operators can be grouped into seven classes:

1. binary logical operators: **and or nand nor xor xnor**
2. relational operators: = /= < <= > >=
3. shift operators: **sll srl sla sra rol ror**
4. adding operators: + − & (concatenation)
5. unary sign operators: + −
6. multiplying operators: * **/ mod rem**
7. miscellaneous operators: **not abs** **

When parentheses are not used, operators in class 7 have highest precedence and are applied first, followed by class 6, then class 5, etc. Class 1 operators have lowest precedence and are applied last. Operators in the same class have the same precedence and are applied from left to right in an expression. The precedence order can be changed by using parentheses. In the following expression, A, B, C, and D are bit_vectors:

```
not A or B and not C & D
```

In this expression, **not** is performed first, then **&** (concatenation), then **or**, and finally **and**. The equivalent expression using parentheses is

```
((not A) or B) and ((not C) &D)
```

The binary logical operators (class 1) as well as **not** can be applied to bits, booleans, bit_vectors, and boolean_vectors. The class 1 operators require two operands of the same type and size, and the result is of that type and size.

Relational operators (class 2) are used to compare two expressions and return a value of FALSE or TRUE. The two expressions must be of the same type and size. Equal(=) and not equal (/=) apply to any type, but the application of the other relational operators is more restricted. Note that "=" is always a relational operator, but "<=" also serves as an assignment operator. Example: If A = 5, B = 4, and C = 3 the expression

```
(A >= B) and (B <= C) evaluates to FALSE.
```

Figure 10-14 shows a comparator for two integers with a restricted range. C must be of type Boolean since the condition A <= B evaluates to TRUE or FALSE. If we implement the comparator in hardware, each integer would be represented by a 4-bit signal because the range is restricted to 0 to 15. C, D, and E would each be one bit (0 for FALSE or 1 for TRUE).

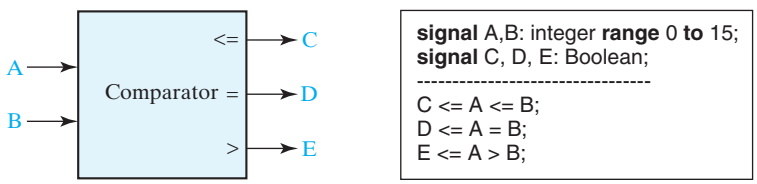


FIGURE 10-14
Comparator for Integers

The shift operators are used to shift or rotate a bit_vector. In the following examples, A is an 8-bit vector equal to "10010101":

A **sll** 2	is "01010100" (shift left logical, filled with '0')
A **srl** 3	is "00010010" (shift right logical, filled with '0')
A **sla** 3	is "10101111" (shift left arithmetic, filled with rightmost bit)
A **sra** 2	is "11100101" (shift right arithmetic, filled with leftmost bit)
A **rol** 3	is "10101100" (rotate left)
A **ror** 5	is "10101100" (rotate right)

We will not utilize these shift operators because some software used for synthesis uses different shift operators. Instead, we will do shifting using the concatenation operator. For example, if A in the above listing is dimensioned 7 downto 0, we can implement shift right arithmetic two places as follows:

A(7)&A(7)&A(7 **downto** 2) = '1'&'1'&"100101" = "11100101"

This makes two copies of the sign bit followed by the left 6 bits of A, which gives the same result as A **sra** 2.

The + and − operators can be applied to integer or real numeric operands. The & operator can be used to concatenate two vectors (or an element and a vector, or two elements) to form a longer vector. For example, "010" & '1' is "0101" and "ABC" & "DEF" is "ABCDEF."

The * and / operators perform multiplication and division on integer or floating-point operands. The **rem** and **mod** operators calculate the remainder and modulus for integer operands. (We will not use rem and mod; for further discussion of these operators see Reference [1].) The ** operator raises an integer or floating-point number to an integer power, and **abs** finds the absolute value of a numeric operand.

10.7 Packages and Libraries

Packages and libraries provide a convenient way of referencing frequently used functions and components. A package consists of a package declaration and an optional package body. The package declaration contains a set of declarations which may be shared by several design units. For example, it may contain type, signal, component, function, and procedure declarations. The package body usually contains component descriptions and the function and procedure bodies. The package and its associated compiled VHDL models may be placed in a library, so they can be accessed as required by different VHDL designs. A package declaration has the form:

package package-name **is**
    package declarations
end [**package**][package-name];

A package body has the form

```
package body package-name is
    package body declarations
end [package body][package name];
```

We have developed a package called bit_pack which is used in a number of examples in this book. This package contains commonly used components and functions which use signals of type bit and bit_vector. A complete listing of this package and associated component models is included on the CD-ROM that accompanies this text. Most of the components in this package have a default delay of 10 ns, but this delay can be changed by the use of generics. For an explanation of generics, refer to one of the VHDL references. We have compiled this package and the component models and placed the result in a library called BITLIB.

One of the components in the library is a two-input NOR gate named Nor2, which has default delay of 10 ns. The package declaration for bit_pack includes the component declaration

```
component Nor2
    port (A1, A2: in bit; Z: out bit);
end component;
```

The NOR gate is modeled using a concurrent statement. The entity-architecture pair for this component is

```
-- two-input NOR gate
entity Nor2 is
    port (A1, A2: in bit; Z: out bit);
end Nor2;

architecture concur of Nor2 is
begin
    Z <= not(A1 or A2) after 10 ns;
end concur;
```

To access components and functions within a package requires a **library** statement and a **use** statement. The statement

```
library BITLIB;
```

allows your design to access the BITLIB. The statement

```
use BITLIB.bit_pack.all;
```

allows your design to use the entire bit_pack package. A statement of the form

```
use BITLIB.bit_pack.Nor2;
```

may be used if you want to use a specific component (in this case Nor2) or function in the package.

When components from a library package are used, component declarations are not needed. Figure 10-15 shows a NOR-NOR circuit and the corresponding structural VHDL code. This code instantiates three copies of the Nor2 gate component from the package bit_pack and connects the gate inputs and outputs.

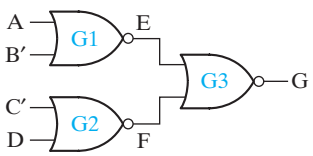


FIGURE 10-15
NOR-NOR Circuit and Structural VHDL Code Using Library Components


```
library BITLIB;
use BITLIB.bit_pack.all;
entity nor_nor is
    port (A,B,C,D: in bit; G: out bit);
end nor_nor;
architecture structural of nor_nor is
signal E,F,BN,CN: bit;  -- internal signals
begin
    BN <= not B;  CN <= not C;
    G1: Nor2 port map (A, BN, E);
    G2: Nor2 port map (CN, D, F);
    G3: Nor2 port map (E, F, G);
end structural;
```

10.8 IEEE Standard Logic

Use of two-valued logic (bits and bit vectors) is generally not adequate for simulation of digital systems. In addition to '0' and '1', values of 'Z' (high-impedance or no connection) and 'X' (unknown) are frequently used in digital system simulation. The IEEE Standard 1164 defines a std_logic type that actually has nine values ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', and '–'). We will only be concerned with the first five values in this text. 'U' stands for uninitialized. When a logic circuit is first turned on and before it is reset, the signals will be uninitialized. If these signals are represented by std_logic, they will have a value of 'U' until they are changed. Just as a group of bits is represented by a bit_vector, a group of std_logic signals is represented by a std_logic_vector.

Figure 10-16 shows how a tri-state buffer can be represented by a concurrent statement. When the buffer is enabled (B = '1'), the output is A, or else it is high impedance ('Z'). A and C could be std_logic_vectors instead of std_logic bits.

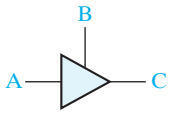


FIGURE 10-16
Tri-State Buffer


```
signal A,B,C: std_logic;
--------------------------------
C <= A when B = '1' else 'Z';
```

Figure 10-17 shows two tri-state buffers with their outputs connected together by a tri-state bus. If buffer 1 has an output of '1' and buffer 2 has a hi-Z output, the bus value is '1'. When both buffers are enabled, if buffer 1 drives '0' onto the bus and buffer 2 drives '1' onto the bus, the result is a bus conflict. In this case, the bus value is unknown, which we represent by an 'X'.

In the VHDL code, A, C, and F are std_logic_vectors and F represents the tri-state bus. The signal F is driven from two different sources. If the two concurrent

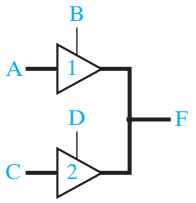


FIGURE 10-17
Tri-State Buffers Driving a Bus


```
signal A,C,F: std_logic_vector(3 downto 0);
signal B,D: std_logic;
-------------------------------------------
-- concurrent statements
F <= A when B = '1' else "ZZZZ";
F <= C when D = '1' else "ZZZZ";
```

statements assign different values to F, VHDL automatically calls a *resolution function* to determine the resulting value. This is similar to the way the hardware works—if the two buffers have different output values, the hardware resolves the values and comes up with an appropriate value on the bus. VHDL uses the table of Figure 10-18 to resolve the bus value when two different std_logic signals, S1 and S2, drive the bus. (Only signal values 'U', 'X', '0', '1', and 'Z' are considered here.) This table is similar to Figure 9-14, which is used for four-valued logic simulation, except for the addition of a row and a column corresponding to 'U'. When an uninitialized signal is connected to any other signal, VHDL considers that the result is uninitialized.

FIGURE 10-18
Resolution Function for Two Signals

S1 \ S2	U	X	0	1	Z
U	U	U	U	U	U
X	U	X	X	X	X
0	U	X	0	X	0
1	U	X	X	1	1
Z	U	X	0	1	Z

If A, B, and F are bits (or bit_vectors) and we write the concurrent statements

```
F <= A;   F <= not B;
```

the compiler will flag an error because no resolution function exists for signals of type bit. If A, B, and F are std_logic bits or vectors, the compiler will generate a call to the resolution function and not report an error. If F is assigned conflicting values during simulation, then F will be set to 'X' (unknown).

In order to use signals of type std_logic and std_logic_vector in a VHDL module, the following declarations must be placed before the entity declaration:

```
library ieee;
use ieee.std_logic_1164.all;
```

The IEEE std_logic_1164 package defines std_logic and related types, logic operations on these types, and functions for working with these types.

The original IEEE standards for VHDL do not define arithmetic operations on bit_vectors or on std_logic vectors. Based on these standards, we cannot add, subtract, multiply, or divide bit_vectors or std_logic_vectors without first converting them to other types. For example, if A and B are bit_vectors, the expression A + B is not allowed. However, VHDL libraries and packages are available that define arithmetic and comparison operations on std_logic_vectors. The operators defined in these packages are referred to as *overloaded* operators. This means that the compiler will automatically use the proper definition of the operator depending on its context. For example, when evaluating the expression A + B, if A and B are integers, the compiler will use the integer arithmetic routine to do the addition. On the other hand, if A and B are of type std_logic_vector, the compiler will use the addition routine for standard logic vectors. In order to use overloaded operators, the appropriate library and use statements must be included in the VHDL code so that the compiler can locate the definitions of these operators.

In this text, we will use the std_logic_unsigned package, originally developed by Synopsys and now widely available. This package treats std_logic_vectors as unsigned

numbers. The std_logic_unsigned package defines arithmetic operators (+, –, *) and comparison operators (<, <=, =, /=, >, >=) that operate on std_logic_vectors. For +, –, and comparison operators, if the two operands are of different length, the shorter operand is filled on the left end with zeros.

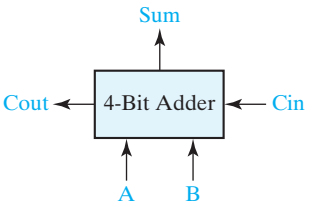
These operations can also be applied when the left operand is a std_logic_vector and the right operand is an integer. The arithmetic operations return a std_logic_vector, and the comparison operations return a Boolean. For example, if A is "10011", A + 7 returns a value of "11010", and A >= 5 returns TRUE. In these examples, + and >= are overloaded operators, and the compiler automatically calls the appropriate routine to add an integer to a std_logic_vector or to compare an integer with a std_logic_vector.

If A and B are 4-bit std_logic vectors, A + B gives their sum as a 4-bit vector, and any carry is lost. If the carry is needed, then A must be extended to five-bits before addition. This is accomplished by concatenating a '0' in front of A. Then '0' & A + B gives a 5-bit sum that can be split into a carry and a 4-bit sum.

Figure 10-19 shows a binary adder and its VHDL representation using the std_logic_unsigned package. Addout is a 5-bit sum that is split into Sum and Cout. For example, if A = "1011", B = "1001", and Cin = '1', Addout evaluates to "10101", which is then split into a sum "0101" with a carry out of '1'.

Figure 10-20 shows how to implement the bi-directional input-output pin and tri-state buffer of Figure 9-16 using IEEE std_logic. The I/O pin declared in the port is

FIGURE 10-19
VHDL Code for Binary Adder




```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
-----------------------------------------------
signal A,B,Sum: std_logic_vector(3 downto 0);
signal Addout: std_logic_vector(4 downto 0);
signal Cin,Cout: std_logic;
-----------------------------------------
Addout <= '0'&A + B + Cin;
Sum <= Addout(3 downto 0);
Cout <= Addout(4);
```

FIGURE 10-20
VHDL Code for Bi-Directional I/O Pin


```
entity IC_pin is
    port(IO_pin: inout std_logic);
end entity;
architecture bi_dir of IC_pin is
    component IC
        port(input: in std_logic; output: out std_logic);
    end component;
    signal input, output, en: std_logic;
begin           -- connections to bi-directional I/O pin
    IO_pin <= output when en = '1' else 'Z';
    input <= IO_pin;
    IC1: IC port map (input, output);
end bi_dir;
```

of mode **inout**. The concurrent statements in the architecture connect the IC output to the pin via a tri-state buffer and also connect the pin to the IC input.

10.9 Compilation and Simulation of VHDL Code

After describing a digital system in VHDL, simulation of the VHDL code is important for two reasons. First, we need to verify the VHDL code correctly implements the intended design, and second, we need to verify that the design meets its specifications. Before the VHDL model of a digital system can be simulated, the VHDL code must first be compiled (see Figure 10-21). The VHDL compiler, also called an analyzer, first checks the VHDL source code to see that it conforms to the syntax and semantic rules of VHDL. If there is a syntax error such as a missing semicolon or a semantic error such as trying to add two signals of incompatible types, the compiler will output an error message. The compiler also checks to see that references to libraries are correct. If the VHDL code conforms to all of the rules, the compiler generates intermediate code which can be used by a simulator or by a synthesizer.

In preparation for simulation, the VHDL intermediate code must be converted to a form which can be used by the simulator. This step is referred to as *elaboration*. During elaboration, ports are created for each instance of a component, memory storage is allocated for the required signals, the interconnections among the port signals are specified, and a mechanism is established for executing the VHDL statements in the proper sequence. The resulting data structure represents the digital system being simulated. After an initialization phase, the simulator enters the execution phase. The simulator accepts simulation commands which control the simulation of the digital system and specify the desired simulator output.

Understanding the role of the delta (Δ) time delays is important when interpreting output from a VHDL simulator. Although the delta delays do not show up on waveform outputs from the simulator, they show up on listing outputs. The simulator uses delta delays to make sure that signals are processed in the proper sequence. Basically, the simulator works as follows: Whenever a component input changes, the output is scheduled to change after the specified delay or after Δ if no delay is specified. When all input changes have been processed, the simulated time is advanced to the next time at which an output change is specified. When time is advanced by a finite amount (1 ns for example), the Δ counter is reset, and simulation resumes. Real time does not advance again until all Δ delays associated with the current simulation time have been processed.

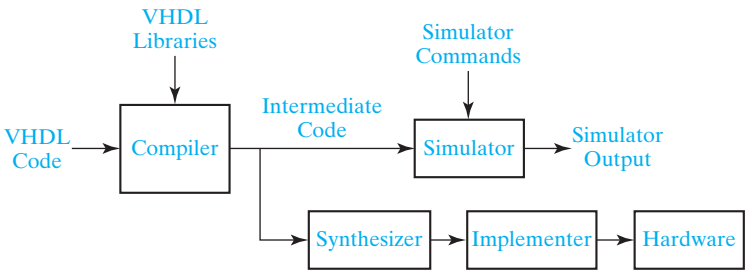


FIGURE 10-21
Compilation, Simulation, and Synthesis of VHDL Code