CHAPTER 5

SEQUENTIAL STATEMENTS OF VHDL

As the name suggests, sequential statements are executed in sequence. The semantics of these statements is more like that of a traditional programming language. Since they are not compatible with the general concurrent execution model of VHDL, sequential statements have to be enclosed inside a construct known as a *process*. The main purpose of sequential statements is to describe and model a circuit's "abstract behavior." Unlike concurrent signal assignment statements, there is no clear mapping between sequential statements and hardware components. Some statements and coding styles are difficult or even impossible to synthesize. To use processes and sequential statements for synthesis, the VHDL description has to be coded in a disciplined way so that the code can be faithfully mapped into the intended hardware configuration.

5.1 VHDL PROCESS

5.1.1 Introduction

A *process* is a VHDL construct that contains a set of actions to be executed sequentially. These actions are known as *sequential statements*. The process itself is a concurrent statement. It can be interpreted as a circuit part enclosed inside a black box whose behavior is described by the sequential statements. We may or may not be able to construct physical hardware that exhibits the desired behavior.

Sequential statements include a rich variety of constructs, and they can exist only inside a process. The execution inside a process is sequential, and thus the order of the statements is important. Many sequential constructs don't have clear counterparts in hardware implementation, and are difficult, if not impossible, to synthesize. We examine the use and synthesis of the following sequential statements in this chapter:

- ~~*wait statement*~~
- *sequential signal assignment statement*
- *variable assignment statement*
- *if statement*
- *case statement*
- simple *for loop statement*

More sophisticated loop statements as well as two other sequential statements, the *exit* and *next statements*, are discussed in Chapter 14. Note that we should not confuse sequential statements with sequential circuits. *Sequential statements* are VHDL statements inside a process, and *sequential circuits* are circuits with internal states. A process and its internal sequential statements can be used to describe a combinational or sequential circuit. As in Chapter 4, our discussion in this chapter is limited to combinational circuits.

The process has two basic forms. The first form has a *sensitivity list* but no wait statement inside the process. The second form has one or more wait statements but no sensitivity list. Because of its clarity, we use mainly the first form in this book. The second form is examined briefly in Section 5.1.3.

5.1.2 Process with a sensitivity list

The syntax of a process with a sensitivity list is

```
process(sensitivity_list)
   declarations;
begin
   sequential statement;
   sequential statement;
   . . .
end process;
```

The `sensitivity_list` is a list of signals to which the process responds (i.e., is "sensitive to"). The `declarations` part consists of various declarations that are local to the process.

Whereas the appearance of a VHDL process is like a function or procedure of a traditional programming language, the behavior of the process is very different. A VHDL process is not invoked (or called) by another routine. It acts like a circuit part, which is either active (known as *activated*) or inactive (known as *suspended*). A VHDL process is activated when a signal in the sensitivity list changes its value, like a circuit responding to an input signal. Once a process is activated, its statements will be executed sequentially until the end of the process. The process is then suspended until the next change of signal. A simple process with a single sequential signal assignment is

```
signal a,b,c,y: std_logic;  -- in architecture declaration
. . .
process(a,b,c)
begin
   y <= a and b and c;
end process;
```

When any input (i.e., a, b or c) changes, the process is activated and its statement is executed. The statement evaluates the expression and assigns the result to the y signal. This process simply describes a three-input and circuit with a, b and c inputs and y output.

One tricky issue about the process is the *incomplete sensitivity list*, which is a list with one or more input signals missing. For example, the b and c signals are omitted from the sensitivity list of the previous example:

```
signal a,b,c,y: std_logic;
. . .
process(a)
begin
   y <= a and b and c;
end process;
```

When the a signal changes, the process is activated and the circuit acts as expected. On the other hand, when the b or c signal changes, the process remains suspended and the y signal keeps its previous value. This implies that the circuit has some sort of memory element that is triggered at both positive and negative edges of the a signal. When the a signal changes, the expression is evaluated and the result is stored in the memory element. This is not the circuit behavior we expected, and it cannot be synthesized by regular hardware components.

For a combinational circuit, the output is a function of input. This implies that the circuit responds to any input change. Thus, *all* input signals of a combinational circuit should be included in the sensitivity list. A process with incomplete sensitivity can be used to describe a circuit with internal memory and thus infer a memory element. This is discussed in Chapter 8.

5.1.3 Process with a wait statement

A process with wait statements has one or more wait statements but no sensitivity list. The wait statement has several forms:

```
wait on signals;
wait until boolean_expression;
wait for time_expression;
```

Use of the wait statement can best be explained by an example. The code segment of Section 5.1.2 can be rewritten as

```
process
begin
   y <= a and b and c;
   wait on a, b, c;
end process;
```

Note that there is no sensitivity list. The process starts automatically after the system initialization. It continues the execution until a wait statement is reached and then becomes suspended. The statement **wait on** a, b, c means that the process waits for a change in the a or b or c signal. When one of them changes value, the process is activated. It executes to the end of the process, then returns to the beginning of the process and continues execution. It becomes suspended again when reaching the wait statement. The overall effect of this process describes a three-input and gate, as in the previous example.

The behavior of the two other types of wait statements is similar except that the process waits until a special Boolean condition is asserted or waits for a specific amount of time.

Since multiple wait statements are allowed, a process with wait statements can be used to model complex timing behavior and sequential events. However, in synthesis, only few well-defined forms of wait statements can be used, and normally only one wait statement is allowed in a process. Since a process with a sensitivity list can clearly show the input signals and make the code clearer and more descriptive, we prefer this form and normally don't use the wait statement in this book.

5.2 SEQUENTIAL SIGNAL ASSIGNMENT STATEMENT

The syntax of a sequential signal assignment is identical to that of the simple concurrent signal assignment of Chapter 4 except that the former is inside a process. It can be written as

```
signal_name <= projected_waveform;
```

The `projected_waveform` clause consists of a value expression and a time expression, which is generally used to represent the propagation delay. As in the concurrent signal assignment statement, the delay specification cannot be synthesized and we always use the default δ-delay. The syntax becomes

```
signal_name <= value_expression;
```

Note that the concurrent conditional and selected signal assignment statements cannot be used inside the process.

For a signal assignment with δ-delay, the behavior of a sequential signal assignment statement is somewhat different from that of its concurrent counterpart. If a process has a sensitivity list, the execution of sequential statements is treated as a "single abstract evaluation," and the actual value of an expression will not be assigned to a signal until the end of the process. This is consistent with the black box interpretation of the process; that is, the entire process is treated as one indivisible circuit part, and the signal is assigned a value only after the completion of all sequential statements.

Inside a process, a signal can be assigned multiple times. If all assignments are with δ-delays, only the last assignment takes effect. Because the signal is not updated until the end of the process, it *never* assumes any "intermediate" value. For example, consider the following code segment:

```
a,b,c,d,y: std_logic;
. . .
process(a,b,c,d)
begin
   y <= a or c;
   y <= a and b;
   y <= c and d;
end process;
```

It is the same as

```
process(a,b,c,d)
begin
   y <= c and d;
end process;
```

Although this segment is easy to understand, multiple assignments may introduce subtle mistakes in a more complex code and make synthesis very difficult. Unless there is a

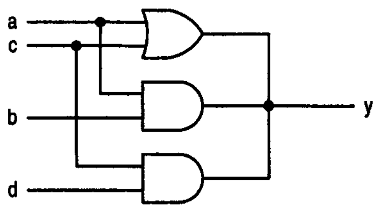

Figure 5.1 Conceptual diagram of multiple concurrent signal assignments.

compelling reason, it is a good idea to avoid assigning a signal multiple times. The only exception is the assignment of a default value in the if and case statements. This is discussed in Sections 5.4.3 and 5.5.3.

The result will be very different if the multiple assignments are the concurrent signal assignment statements. Assume that the previous three assignment statements are concurrent signal assignment statements (i.e., not inside a process). The code segment becomes

```
a,b,c,d,y: std_logic;
. . .
-- the statements are not inside a process
y <= a or c;
y <= a and b;
y <= c and d;
```

The code is syntactically correct since multiple assignments are allowed for a signal with the `std_logic` data type (since it is a "resolved" data type). The corresponding circuit is shown in Figure 5.1. Although the syntax is fine, the design is incorrect because of the potential output conflict. The y signal may get a value of `'X'` in simulation if any two of the output values of the three gates are different.

5.3 VARIABLE ASSIGNMENT STATEMENT

The syntax of a variable assignment statement is

```
variable_name := value_expression;
```

The immediate assignment notion, :=, is used for the variable assignment. There is no time dimension (i.e., no propagation delay) and the assignment takes effect *immediately*. The behavior of the variable assignment is just like that of a regular variable assignment used in a traditional programming language. For example, consider the code segment

```
signal a, b, y: std_logic;
. . .
process(a,b)
  variable tmp: std_logic;
begin
   tmp := '0';
   tmp := tmp or a;
   tmp := tmp or b;
   y <= tmp;
end process;
```

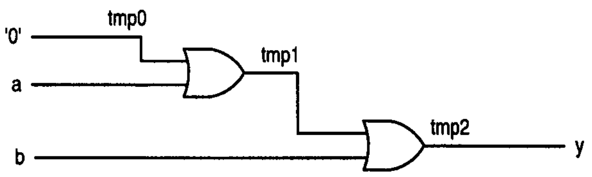

Figure 5.2 Conceptual implementation of simple variable assignments.

The tmp variable assumes the value immediately in each sequential statement and assigns its value, which is equal to $a + b$, to the y signal. Note that the variables are "local" to the process and have to be declared inside the process.

Although the behavior of a variable is easy to understand, mapping it into hardware is difficult. For example, to realize the previous process in hardware, we have to rename the variables tmp0, tmp1 and tmp2 and change the process to

```
process(a,b)
  variable tmp0, tmp1, tmp2: std_logic;
begin
   tmp0 := '0';
   tmp1 := tmp0 or a;
   tmp2 := tmp1 or b;
   y <= tmp2;
end process;
```

For synthesis purposes, we can now interpret the variables as signals or nets. The corresponding diagram is shown in Figure 5.2. Because of the lack of clear hardware mapping, we should try to use signals in code in general and resort to variables only for the characteristics that cannot be described by signals.

For comparison purposes, let us repeat the previous segment by replacing the variables with signals:

```
signal a, b, y, tmp: std_logic;
. . .
process(a,b,tmp)
begin
   tmp <= '0';
   tmp <= tmp or a;
   tmp <= tmp or b;
   y <= tmp;
end process;
```

Note that the signals have to be "global" and declared outside the process, and the tmp signal has to be included in the sensitivity list. This code is the same as

```
process(a,b,tmp)
begin
   tmp <= tmp or b;
   y <= tmp;
end process;
```

This code implies a combinational loop with an or gate, as shown in Figure 5.3.

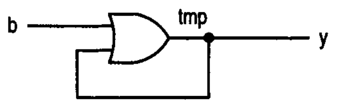

Figure 5.3 Conceptual implementation of erroneous signal assignments.

5.4 IF STATEMENT

5.4.1 Syntax and examples

The simplified syntax of an if statement is

```
if boolean_expr_1 then
   sequential_statements;
elsif boolean_expr_2 then
   sequential_statements;
elsif boolean_expr_3 then
   sequential_statements;
. . .
else
   sequential_statements;
end if;
```

An if statement has one *then branch*, one or more optional *elsif branches* and one optional *else branch*. The `boolean_expr_i` term is a Boolean expression that returns `true` or `false`. These Boolean expressions are evaluated sequentially. When an expression is evaluated as `true`, the statements in the corresponding branch will be executed and the remaining branches will be skipped. If none of the expressions is `true` and the else branch exists, the statements in the else branch will be executed.

We use the same circuit examples as in Chapter 4, which include a multiplexer, a decoder, a priority encoder and a simple ALU, to illustrate use of an if statement. The if statement description of an 8-bit 4-to-1 multiplexer is shown in Listing 5.1. Since the multiplexer is a combinational circuit, all input signals, including `a`, `b`, `c`, `d` and `s`, are in the sensitivity list. Note that the signals used in the Boolean expressions are also the input signals.

Listing 5.1 4-to-1 multiplexer based on an if statement

```
architecture if_arch of mux4 is
begin
   process(a,b,c,d,s)
   begin
      if (s="00") then
         x <= a;
      elsif (s="01")then
         x <= b;
      elsif (s="10")then
         x <= c;
      else
         x <= d;
      end if;
   end process;
end if_arch;
```

The if statement versions of binary decoder, priority encoder and simple ALU are shown in Listings 5.2, 5.3 and 5.4 respectively.

Listing 5.2 2-to-4 decoder based on an if statement

```
architecture if_arch of decoder4 is
begin
   process(s)
   begin
      if (s="00") then
         x <= "0001";
      elsif (s="01")then
         x <= "0010";
      elsif (s="10")then
         x <= "0100";
      else
         x <= "1000";
      end if;
   end process;
end if_arch;
```

Listing 5.3 4-to-2 priority encoder based on an if statement

```
architecture if_arch of prio_encoder42 is
begin
   process(r)
   begin
      if (r(3)='1') then
         code <= "11";
      elsif (r(2)='1')then
         code <= "10";
      elsif (r(1)='1')then
         code <= "01";
      else
         code <= "00";
      end if;
   end process;
   active <= r(3) or r(2) or r(1) or r(0);
end if_arch;
```

Listing 5.4 Simple ALU based on an if statement

```
architecture if_arch of simple_alu is
   signal src0s, src1s: signed(7 downto 0);
begin
   src0s <= signed(src0);
   src1s <= signed(src1);
   process(ctrl,src0,src1,src0s,src1s)
   begin
     if (ctrl(2)='0') then
        result <= std_logic_vector(src0s + 1);
     elsif (ctrl(1 downto 0)="00")then
        result <=  std_logic_vector(src0s + src1s);
     elsif (ctrl(1 downto 0)="01")then
```

```
            result <= std_logic_vector(src0s - src1s);
         elsif (ctrl(1 downto 0)="10")then
            result <= src0 and src1 ;
         else
            result <= src0 or src1;
         end if;
      end process;
   end if_arch;
```

5.4.2 Comparison to a conditional signal assignment statement

An if statement is somewhat like a concurrent conditional signal assignment statement. If the sequential statements inside an if statement consist of only the signal assignment of a single signal, as in previous examples, the two statements are equivalent. Consider the following conditional signal assignment statement:

```
sig <= value_expr_1 when boolean_expr_1 else
       value_expr_2 when boolean_expr_2 else
       value_expr_3 when boolean_expr_3 else
          . . .
       value_expr_n;
```

It can be written as

```
process(...)
begin
   if boolean_expr_1 then
      sig <= value_expr_1;
   elsif boolean_expr_2 then
      sig <= value_expr_2;
   elsif boolean_expr_3 then
      sig <= value_expr_3;
   . . .
   else
      sig <= value_expr_n;
   end if;
end process;
```

Thus, our discussion in Chapter 4 regarding the conditional signal assignment statement can also be applied to the if statement.

The equivalency, however, is true only for this simple scenario. An if statement is much more general since a branch of the if statement can be a *sequence* of sequential statements. Proper and disciplined use of an if statement can make code more descriptive and sometimes even more efficient. For example, an if statement is a sequential statement, and thus it can be nested in a branch of another if statement. Assume that we want to find the maximum value of three signals, a, b and c. One way to do it is by using nested if statements:

```
process(a,b,c)
begin
   if (a > b) then
      if (a > c) then
         max <= a;   -- a>b and a>c
      else
```

```
            max <= c;   -- a>b and c>=a
         end if;
      else
         if (b > c) then
            max <= b;   -- b>=a and b>c
         else
            max <= c;   -- b>=a and c>=b
         end if;
      end if;
   end process;
```

We have to use three conditional signal assignment statements to achieve the same task:

```
signal ac_max, bc_max: std_logic;
. . .
ac_max <= a when (a > c) else c;
bc_max <= b when (b > c) else c;
max <= ac_max when (a > b) else bc_max;
```

We can also convert code using one conditional signal assignment statement. Since it cannot be nested, we have to "flatten" the Boolean conditions of the if statements. The following code follows the pattern of the previous nested Boolean conditions:

```
max <= a when ((a > b) and (a > c)) else
       c when (a > b) else
       b when (b > c) else
       c;
```

Although the code is shorter, it is not very descriptive and is difficult to understand.

Another situation suitable for the if statement is when many operations are controlled by the same Boolean conditions. For example, consider the following code segment:

```
process(a,b)
begin
   if (a > b and op="00") then
      y <= a - b;
      z <= a - 1;
      status <= '0';
   else
      y <= b - a;
      z <= b - 1;
      status <= '1';
   end if;
end process;
```

The Boolean conditions and the if-then-else structure is shared by three signals. On the other hand, we need three conditional signal assignment statements to describe the same circuit:

```
y <= a-b when (a > b and op="00") else
     b-a;
z <= a-1 when (a > b and op="00") else
     b-1;
status <= '0' when (a > b and op="00") else
          '1';
```

5.4.3 Incomplete branch and incomplete signal assignment

In Section 5.1.2, we learned that an incomplete sensitivity list, in which one or more input signals are omitted, may lead to unexpected circuit behavior. This may also happen to the *incomplete branch* and *incomplete signal assignment*. According to VHDL semantics, the else branch is optional and a signal does not need to be assigned in all branches. Although syntactically correct, the omissions introduce unwanted memory elements (i.e., latches).

Incomplete branch In VHDL, only the then branch is mandatory and the other branches can be omitted. For example, the following statement is an attempt to code a comparator that compares the a and b inputs and asserts the eq output when a and b are equal:

```
process(a,b)
begin
   if (a=b) then
      eq <= '1';
   end if;
end process;
```

The code is syntactically correct. When a is equal to b, the eq signal becomes '1'. When a is not equal to b, there is no else branch and thus no action is taken. VHDL semantics specifies that the eq signal does not change and *keeps its previous value*. Thus, the previous statement is the same as

```
process(a,b)
begin
   if (a=b) then
      eq <= '1';
   else
      eq <= eq;
   end if;
end process;
```

This implies a circuit with a closed feedback loop, which constitutes internal states or memory. Clearly, this description does not meet the intended specification. The correct code should be

```
process(a,b)
begin
   if (a=b) then
      eq <= '1';
   else
      eq <= '0';
   end if ;
end process;
```

For a combinational circuit, the else branch should always be included to avoid the unwanted memory or latch.

Incomplete signal assignment An if statement has several branches. It is possible that a signal is assigned only in some, but not all branches. Although syntactically correct, the incomplete signal assignment infers unwanted memory. For example, the following statement attempts to describe a comparator with three outputs, gt, lt and eq, which indicate the conditions "a is greater than b," "a is less than b" and "a is equal to b" respectively:

```
process(a,b)
begin
   if (a>b) then
      gt <= '1';
   elsif (a=b) then
      eq <= '1';
   else
      lt <= '1';
   end if;
end process;
```

The VHDL semantics specifies that a signal will *keep its previous value* if it is not assigned. When a is greater than b, the first branch is taken and the gt signal becomes '1'. The eq and lt signals keep their previous values since they are not assigned. A similar situation occurs in two other branches since only one output signal is assigned. This implies that three unwanted memory elements are inferred from the code. The correct code should have the signals assigned in all branches:

```
process(a,b)
begin
   if (a>b) then
      gt <= '1';
      eq <= '0';
      lt <= '0';
   elsif (a=b) then
      gt <= '0';
      eq <= '1';
      lt <= '0';
   else
      gt <= '0';
      eq <= '0';
      lt <= '1';
   end if;
end process;
```

One way to make the code compact and clear is to assign a default value for each signal in the beginning of the process:

```
process(a,b)
begin
   gt <= '0';
   eq <= '0';
   lt <= '0';
   if (a>b) then
      gt <= '1';
   elsif (a=b) then
      eq <= '1';
   else
      lt <= '1';
   end if;
end process;
```

Recall that, in a process, only the last signal assignment takes effect. If a signal is assigned in a branch of the if statement, that assignment takes effect. If it is not assigned in any branch, the default assignment takes effect. The output signals are therefore always assigned. We can treat the assignment of a default value as shorthand for the previous code segment.

For a combinational circuit, an output signal should be assigned in all branches of an if statement. It is a good practice to assign a default value at the beginning of the process to cover the unassigned branches.

5.4.4 Conceptual implementation

An if statement evaluates a set of Boolean expressions in sequential order and takes action when the first Boolean condition is met. To achieve this in hardware, we need a priority routing network similar to that of a conditional signal assignment statement.

Discussion in Section 5.4.2 shows that a simple one-output-signal if statement is equivalent to a conditional signal assignment statement. We can apply the same procedure as that described in Section 4.3.2 to derive the conceptual block diagram for the simple if statement. Consider an if statement with four branches:

```
if boolean_expr_1 then
   sig <= value_expr_1;
elsif boolean_expr_2 then
   sig <= value_expr_2;
elsif boolean_expr_3 then
   sig <= value_expr_3;
else
   sig <= value_expr_4;
end if;
```

We first derive the circuit for the first branch by constructing the rightmost 2-to-1 multiplexing circuit and the boolean_expr_1 and value_expr_1 circuits. We can then repeat the process and complete the implementation branch by branch. The finished diagram is identical to Figure 4.4.

An if statement is more flexible and can accommodate more than one statement in each branch. The following examples illustrate the construction of two more complex forms. The first form is an if statement with multiple statements in each branch. The following code shows an if statement with two output signals:

```
if boolean_expr then
   sig_a <= value_expr_a_1;
   sig_b <= value_expr_b_1
else
   sig_a <= value_expr_a_2;
   sig_b <= value_expr_b_2;
end if;
```

Since there are two signals, two separating routing networks are needed. When each routing network has its own multiplexer, the two networks use the same Boolean expressions to control the selection signals of the multiplexers. Thus, the boolean_exp circuit is actually shared. The conceptual diagram is shown in Figure 5.4. We can apply the same idea to derive a conceptual diagram for an if statement with more output signals.

The second form is a nested if statement; that is, one or more if statements is used inside the branches of an if statement. The following code shows a two-level nested if statement:

```
if boolean_expr_1 then
   if boolean_expr_2 then
```

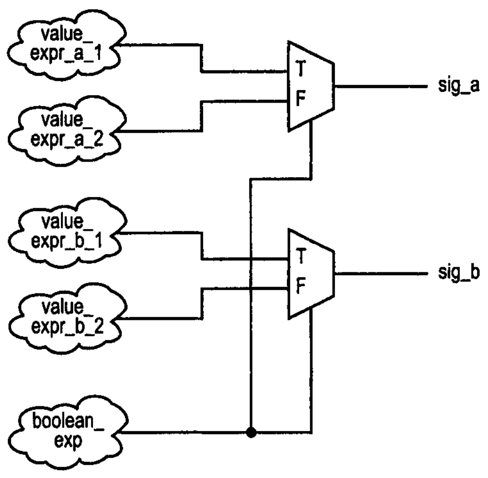

Figure 5.4 Conceptual implementation of an if statement with multiple signal assignments.

```
      signal_a <= value_expr_1;
   else
      signal_a <= value_expr_2;
   end if;
else
   if boolean_expr_3 then
      signal_a <= value_expr_3;
   else
      signal_a <= value_expr_4;
   end if;
end if;
```

The conceptual diagram can be constructed in a hierarchal manner, and the derivation process is shown in Figure 5.5. We first derive the routing structure for the outer if statement, as in Figure 5.5(a), and then realize the two inner if statements inside the then and else branches of the outer if statement, as in Figure 5.5(b). We can apply this procedure repeatedly if the code consists of more nested levels.

5.4.5 Cascading single-branched if statements

Because of the sequential semantics, a signal can be assigned multiple times inside a process and only the last assignment takes effect. We can use this property to construct a priority circuit using a sequence of single-branched if statements (i.e., if statements with only a then branch). For example, the previous priority encoder can be rewritten using three if statements, as shown in Listing 5.5. The `code` signal is first assigned with `"00"`. If the `r(1)` request is asserted, the `code` signal will be reassigned with `"01"`. This procedure continues until the end of the process. Clearly, the Boolean conditions in the later if statements have the higher priority and can override the earlier conditions. Thus, this sequence of if statements implicitly forms a priority circuit.

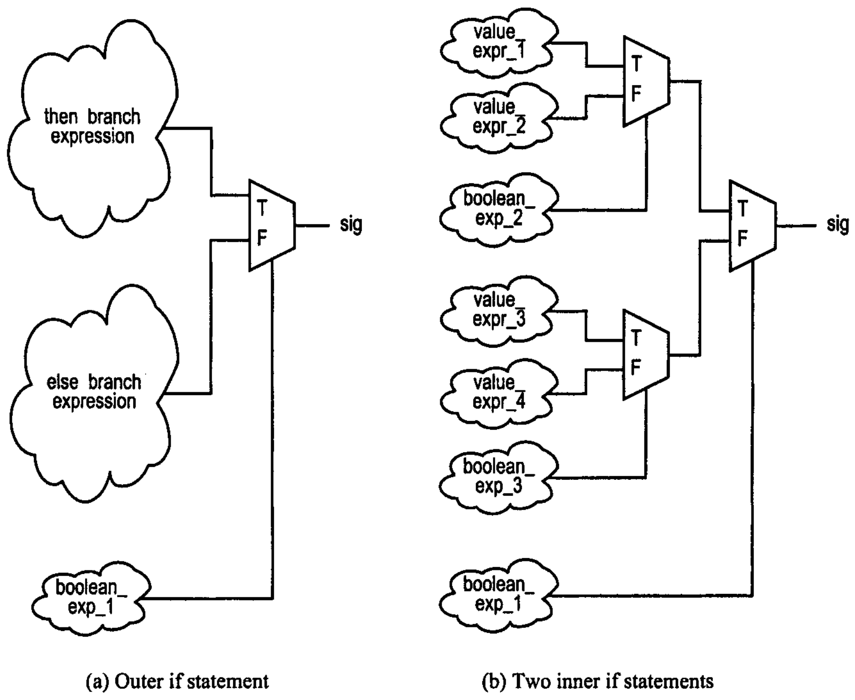

Figure 5.5 Conceptual implementation of a nested if statement.

Listing 5.5 Priority encoder based on cascading if statements

```
architecture cascade_if_arch of prio_encoder42 is
begin
   process(r)
   begin
      code <="00";
      if (r(1)='1') then
         code <= "01";
      end if;
      if (r(2)='1') then
         code <= "10";
      end if;
      if (r(3)='1') then
         code <= "11";
      end if;
   end process;
   active <= r(3) or r(2) or r(1) or r(0);
end cascade_if_arch;
```

We can generalize this idea and replace an if statement with multiple elsif branches with a sequence of simple cascading single-branched if statements. For example, consider the following code segment:

```
if boolean_expr_1 then
   sig <= value_expr_1;
```

```
elsif boolean_expr_2 then
   sig <= value_expr_2;
elsif boolean_expr_3 then
   sig <= value_expr_3;
else
   sig <= value_expr_4;
end if;
```

It can be rewritten as

```
sig <= value_expr_4;
if boolean_expr_3 then
   sig <= value_expr_3;
end if;
if boolean_expr_2 then
   sig <= value_expr_2;
end if;
if boolean_expr_1 then
   sig <= value_expr_1;
end if;
```

Although inferring the same priority configuration, VHDL code in this form is less clear and more difficult for software to synthesize. We should avoid this style in general. However, due to its repetitive nature, this form is sometimes useful to describe a replicated structure of parameterized design. This aspect is discussed in detail in Chapter 14.

5.5 CASE STATEMENT

5.5.1 Syntax and examples

The simplified syntax of a case statement is

```
case case_expression is
   when choice_1 =>
      sequential statements;
   when choice_2 =>
      sequential statements;
   . . .
   when choice_n =>
      sequential statements;
end case;
```

A case statement uses the value of `case_expression` to select a set of sequential statements. The `case_expression` term functions just as the `select_expression` term of the concurrent selected signal assignment statement. Its data type must be a discrete type or one-dimensional array. The `choice_i` term is a value or a set of values that can be assumed by `case_expression`. The choices have to be *mutually exclusive* (i.e., no value can be used more than once) and *all-inclusive* (i.e., all values must be included). The keyword **others** can be used in `choice_n` in the end to cover all unused values.

Again, we use the same multiplexer, binary decoder, priority encoder and ALU circuits to show the use of the case statement. The VHDL code of the multiplexer is shown in Listing 5.6. Note that there are 81 (9*9) possible combinations for the 2-bit `s` signal, including the normal `"00"`, `"01"`, `"10"` and `"11"` combinations as well as 77 other metavalue combinations. This issue was examined in the selected signal assignment statement in Section 4.4.1, and the discussion can be applied to the case statement as well. In the code, we use the **when others** clause to cover "11" and all unused combinations. The signals used in `case_expression` are the inputs to the circuit and thus should be included in the sensitivity list.

Listing 5.6 4-to-1 multiplexer based on a case statement

```
architecture case_arch of mux4 is
begin
   process(a,b,c,d,s)
   begin
      case s is
         when "00" =>
            x <= a;
         when "01" =>
            x <= b;
         when "10" =>
            x <= c;
         when others =>
            x <= d;
      end case;
   end process;
end case_arch;
```

The VHDL codes for the other three examples are shown in Listings 5.7, 5.8 and 5.9.

Listing 5.7 2-to-4 decoder based on a case statement

```
architecture case_arch of decoder4 is
begin
   process(s)
   begin
      case s is
         when "00" =>
            x <= "0001";
         when "01" =>
            x <= "0010";
         when "10" =>
            x <= "0100";
         when others =>
            x <= "1000";
      end case;
   end process;
end case_arch;
```

Listing 5.8 4-to-2 priority encoder based on a case statement

```
architecture case_arch of prio_encoder42 is
begin
   process(r)
   begin
      case r is
         when "1000"|"1001"|"1010"|"1011"|
              "1100"|"1101"|"1110"|"1111" =>
```

```
            code <= "11";
         when "0100"|"0101"|"0110"|"0111" =>
            code <= "10";
         when "0010"|"0011" =>
            code <= "01";
         when others =>
            code <= "00";
      end case;
   end process;
   active <= r(3) or r(2) or r(1) or r(0);
end case_arch;
```

Listing 5.9 Simple ALU based on a case statement

```
architecture case_arch of simple_alu is
   signal src0s, src1s: signed(7 downto 0);
begin
   src0s <= signed(src0);
   src1s <= signed(src1);
   process(ctrl,src0,src1,src0s,src1s)
   begin
      case ctrl is
         when "000"|"001"|"010"|"011" =>
            result <=  std_logic_vector(src0s + 1);
         when "100" =>
            result <=  std_logic_vector(src0s + src1s);
         when "101" =>
            result <= std_logic_vector(src0s - src1s);
         when "110" =>
            result <= src0 and src1;
         when others =>     -- "111"
            result <= src0 or src1;
      end case;
   end process;
end case_arch ;
```

5.5.2 Comparison to a selected signal assignment statement

A case statement is somewhat like a concurrent selected signal assignment statement. If each when branch of a case statement consists only of the assignment of a single signal, the two statements are equivalent. Consider a selected signal assignment statement:

```
with sel_exp select
   sig <= value_expr_1 when choice_1,
          value_expr_2 when choice_2,
          value_expr_3 when choice_3,
          . . .
          value_expr_n when choice_n;
```

It can be rewritten as

```
process(...)
begin
```

```
      case sel_exp is
         when choice_1 =>
            sig <= value_expr_1;
         when choice_2 =>
            sig <= value_expr_2;
         when choice_3 =>
            sig <= value_expr_3;
         . . .
         when choice_n =>
            sig <= value_expr_n;
      end case;
   end process;
```

Thus, the discussion in Chapter 4 regarding the selected signal assignment statement can also be applied to a case statement. Again, the equivalence is limited to this simple scenario. The case statement is much more flexible and general since each when branch can consist of a sequence of sequential statements. The comparison between the if statement and the conditional signal assignment statement in Section 5.4.2 can be applied here as well.

5.5.3 Incomplete signal assignment

Unlike an if statement, the choices of a case statement have to be inclusive, and thus no omitted when clause is allowed. Any "incomplete when clause" will lead to a syntax error and thus be detected when the VHDL code is analyzed. However, incomplete signal assignment can still occur and infer unwanted memory. For example, the following statement attempts to describe a priority encoder with a 3-bit input request signal, a, and three output signals, `high`, `middle` and `low`. The `a(3)` signal has the highest priority. When it is `'1'`, the `high` signal will be asserted. The two other output signals are for two other lower requests. The code uses a case statement:

```
   process(a)
   begin
      case a is
         when "100"|"101"|"110"|"111" =>
            high <= '1';
         when "010"|"011" =>
            middle <= '1';
         when others =>
            low <='1';
      end case;
   end process;
```

Again, the VHDL semantics specifies that a signal will keep its previous value if it is unassigned. If the a signal is `"111"`, the first when clause is taken and the `high` signal is assigned a `'1'`. Since the `middle` and `low` signals are unspecified, they keep their previous values. A similar situation occurs in other when clauses, and therefore three unwanted memory elements are inferred. To fix the problem, we must make sure to have the signals assigned in all when clauses:

```
   process(a)
   begin
      case a is
         when "100"|"101"|"110"|"111" =>
```

```
            high <= '1';
            middle <= '0';
            low <= '0';
         when "010"|"011" =>
            high <= '0';
            middle <= '1';
            low <= '0';
         when others =>
            high <= '0';
            middle <= '0';
            low <= '1';
      end case;
   end process;
```

As in the if statement discussion, we can also use a default assignment to make the code clearer and more compact:

```
   process(a)
   begin
      high <= '0';
      middle <= '0';
      low <= '0';
      case a is
         when "100"|"101"|"110"|"111" =>
            high <= '1';
         when "010"|"011" =>
            middle <= '1';
         when others =>
            low <='1';
      end case;
   end process;
```

5.5.4 Conceptual implementation

A case statement utilizes the value of `case_expression` to select a set of sequential statements to execute. Conceptually, it can be thought of as an abstract multiplexing circuit that utilizes `case_expression` as the selection signal to route the results of designated expressions to output signals. A case statement with a single output signal can be implemented by an abstract multiplexer identical to the one used in the selected signal assignment statement in Section 4.4.2. Consider the following case statement:

```
   case case_exp is
      when c0 =>
         sig <= value_expr_0;
      when c1 =>
         sig <= value_expr_1;
      when others =>
         sig <= value_expr_n;
   end case;
```

We assume that `case_exp` may result in one of five possible values: `c0`, `c1`, `c2`, `c3` and `c4`. The **when others** clause implicitly covers `c2`, `c3` and `c4`. The conceptual diagram is identical to Figure 4.11 except that the `selection_expression` circuit is replaced by the `case_exp` circuit.

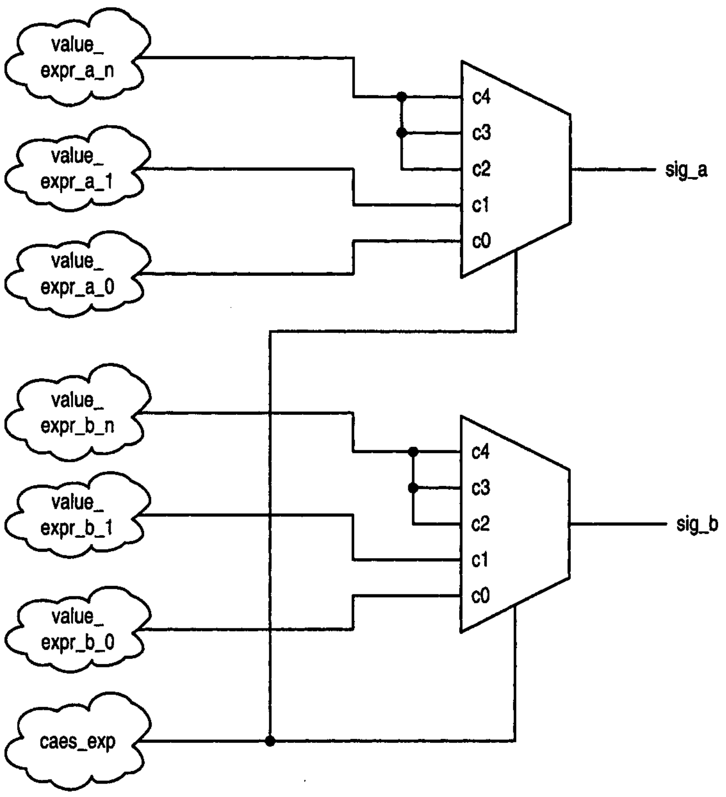

Figure 5.6 Conceptual implementation of a case statement with multiple output signals.

The previous scheme can easily be extended for a case statement with multiple output signals. We simply duplicate the abstract multiplexer for each signal and connect the case_exp to the selection signals of all multiplexers. For example, the following case statement has two output signals:

```
case case_exp is
   when c0 =>
      sig_a <= value_expr_a_0;
      sig_b <= value_expr_b_0;
   when c1 =>
      sig_a <= value_expr_a_1;
      sig_b <= value_expr_b_1;
   when others =>
      sig_a <= value_expr_a_n;
      sig_b <= value_expr_b_n;
end case;
```

The corresponding conceptual diagram is shown in Figure 5.6. As an if statement, a case statement is very general. Any valid sequence of sequential statements can be included inside a when branch. We can derive the conceptual diagram from the outermost level and iterate through the inner levels, as in the nested if statement example in Section 5.4.4.

5.6 SIMPLE FOR LOOP STATEMENT

5.6.1 Syntax

VHDL provides a variety of loop constructs, including *simple infinite loop*, *for loop* and *while loop*, as well as mechanisms to terminate a loop, including the *exit* statement, which skips the remaining iterations of the loop, and the *next* statement, which skips the remaining part of the current iteration. These constructs are mainly for modeling. Only few, very restricted forms of loop can be realized by hardware and synthesized automatically. In this section, we limit the discussion to the simple for loop statement and use it as shorthand for repetitive statements. A more general application of loops is discussed in Chapter 14.

The simplified syntax of the for loop statement is:

```
for index in loop_range loop
   sequential statements;
end loop;
```

The for loop repeats the loop body of sequential statements for a fixed number of iterations. The `loop_range` term specifies a range of values between the left and right bounds. A loop index, `index`, is used to keep track of the iteration and takes a successive value from `loop_range` in each iteration, starting with the leftmost value. The loop index automatically takes the data type of `loop_range`'s element and does not need to be declared. For synthesizable code, `loop_range` must be determined at the time of synthesis (i.e., be *static*) and cannot change with the input signal. The loop body is a sequence of sequential statements. It is very flexible and versatile but can be difficult or impossible to synthesize. In this chapter, we limit it to sequential signal assignment statements.

5.6.2 Examples

Use of the for loop statement is demonstrated by two examples. The first example is a 4-bit xor circuit and its code is shown in Listing 5.10. The for loop performs bitwise xor operation on two 4-bit signals. The operation is done one bit at a time. The loop range is `WIDTH-1` **downto** `0`. We use a symbolic constant here to make the code more readable and to facilitate future modification. The loop index is `i`. It is local to the loop and does not need to be declared. The index assumes a value of 3, the leftmost value in the range, in the first iteration, and then assumes a value of 2 in the second iteration. The iteration continues until the value of the rightmost value, 0, is used.

Listing 5.10 Bitwise xor operation using a for loop statement

```
library ieee;
use ieee.std_logic_1164.all;
entity bit_xor is
   port(
      a, b: in std_logic_vector(3 downto 0);
      y: out std_logic_vector(3 downto 0)
   );
end bit_xor;

architecture demo_arch of bit_xor  is
   constant WIDTH: integer := 4;
begin
```

```
      process(a,b)
      begin
         for i in (WIDTH-1) downto 0 loop
            y(i) <= a(i) xor b(i);
         end loop;
      end process;
   end demo_arch;
```

The code here is just for demonstration purposes. The same operation can actually be achieved by a single statement:

```
y <= a xor b;
```

The second example is a *reduced-xor* circuit, which performs the xor operation over a group of signals. For example, consider a group of four signals, a_3, a_2, a_1 and a_0. The reduced-xor operation of the four signals is $a_3 \oplus a_2 \oplus a_1 \oplus a_0$. The for-loop VHDL description of this circuit is shown in Listing 5.11.

Listing 5.11 Reduced-xor operation using a for loop statement

```
library ieee;
use ieee.std_logic_1164.all;
entity reduced_xor_demo is
   port(
      a: in std_logic_vector(3 downto 0);
      y: out std_logic
   );
end reduced_xor_demo;

architecture demo_arch of reduced_xor_demo is
   constant WIDTH: integer := 4;
   signal tmp: std_logic_vector(WIDTH-1 downto 0);
begin
   process(a,tmp)
   begin
      tmp(0) <= a(0);   -- boundary bit
      for i in 1 to (WIDTH-1) loop
         tmp(i) <= a(i) xor tmp(i-1);
      end loop;
   end process;
   y <= tmp(WIDTH-1);
end demo_arch;
```

5.6.3 Conceptual implementation

The basic way to realize a for loop in hardware is to *unroll* or *flatten* the loop and convert it into code that contains no loop constructs. The flattened code can then be constructed accordingly. This implies that we replicate the hardware described by the loop body for each iteration. To unroll a loop, the range has to be constant and has to be known at the time of synthesis. That is why the range has to be static. We cannot, for example, use the value of an input signal to set the range's right boundary.

Let us first consider the bitwise xor code. The for loop can be unrolled by manually substituting index i into the loop body for four iterations. The flattened code becomes

```
y(3) <= a(3) xor b(3);
y(2) <= a(2) xor b(2);
y(1) <= a(1) xor b(1);
y(0) <= a(0) xor b(0);
```

We can derive the conceptual implantation accordingly. Similarly, the reduced-xor code can be unrolled and the flattened code is

```
tmp(0) <= a(0);
tmp(1) <= a(1) xor tmp(0);
tmp(2) <= a(2) xor tmp(1);
tmp(3) <= a(3) xor tmp(2);
y <= tmp(3);
```

Since we limit the loop body to sequential signal assignment statements, the implementation is straightforward. The for loop can be thought of as shorthand for repetitive statements. A unique property of the for loop is that we can use the range to control hardware replication. It is very useful for the development of parameterized design, in which the "width" of the circuit (e.g., the input width of an adder) can be adjusted to match a specific need. For example, we can change the value of the WIDTH constant of the reduced-xor code to accommodate different input widths. The implementation and synthesis of more versatile loop structure and the parameterized design are examined in Chapters 14 and 15.

5.7 SYNTHESIS OF SEQUENTIAL STATEMENTS

The nature of concurrent and sequential statements is very different. Concurrent statements are modeled after hardware, and thus there is a clear, direct mapping between a concurrent statement and a hardware structure. On the other hand, sequential statements are intended to describe the abstract behavior of a system, and some constructs cannot be easily realized by hardware. Sequential statements are more flexible and versatile than concurrent statements. For synthesis, this is a mixed blessing. On the positive side, the flexibility allows us to specify the desired design in a compact, clear and descriptive manner and to explore more design alternatives. On the negative side, the flexibility can easily be abused. It may make us think falsely that we can synthesize hardware directly from sequential descriptions. This usually leads to unnecessarily complex or unsynthesizable implementation.

Our goal is to develop code for synthesis. When we use sequential statements, we should think in terms of hardware rather than treating them as a way to describe a sequential algorithm. This helps us to focus on the underlying hardware complexity and the efficiency of the design. One good way to check sequential statements is to ask ourselves whether we can derive the conceptual diagram manually. If we cannot, the description is probably also too difficult for synthesis software to synthesize.

5.8 SYNTHESIS GUIDELINES

5.8.1 Guidelines for using sequential statements

- Variables should be used with care. A signal is generally preferred. A statement like n:=n+1 can cause great confusion for synthesis.

- Except for the default value, avoid overriding a signal multiple times in a process.

- Think of the if and case statements as routing structures rather than as sequential control constructs.
- An if statement infers a priority routing structure, and a larger number of elsif branches leads to a long cascading chain.
- A case statement infers a multiplexing structure, and a large number of choices leads to a wide multiplexer.
- Think of a for loop statement as a mechanism to describe the replicated structure.
- Avoid "innovative" use of language constructs. We should be innovative about the hardware architecture but not about the description of the architecture. Synthesis software may not be able to interpret the intention of the code.

5.8.2 Guidelines for combinational circuits

- For a combinational circuit, include all input signals in the sensitivity list to avoid unexpected behavior.
- For a combinational circuit, include all branches of an if statement to avoid unwanted latch.
- For a combinational circuit, an output signal should be assigned in every branch of the if and case statements to avoid unwanted latch.
- For a combinational circuit, it is a good practice to assign a default value to each signal at the beginning of the process.

5.9 BIBLIOGRAPHIC NOTES

The bibliographic information for this chapter is similar to that of Chapter 4.

Problems

5.1 Consider a circuit described by the following code segment:

```
process(a)
begin
  q <= d;
end process;
```

(a) Describe the operation of this circuit.
(b) Does this circuit resemble any real physical component?

5.2 Consider the following code segment:

```
process(a,b)
begin
  if a='1' then
     q <= b;
  end if;
end process;
```

(a) Describe the operation of this circuit.
(b) Draw the conceptual diagram of this circuit.

5.3 Add an enable signal, en, to the 2-to-4 decoder discussed in Section 5.4.1. When en is '1', the decoder functions as usual. When en is '0', the decoder is disabled and the output becomes "0000". Use an if statement to derive this circuit and draw the conceptual diagram.

5.4 Repeat Problem 5.3, but use a case statement to derive the circuit.

5.5 Derive the conceptual diagram for the following code segment:

```
if (a > b and op="00") then
   y <= a - b;
   z <= a - 1;
   status <= '0';
else
   y <= b - a;
   z <= b - 1;
   status <= '1';
end if;
```

5.6 Consider the 2-by-2 switch discussed in Problem 4.3. Its inputs are x1, x0 and ctrl, and its outputs are y1 and y0. The functional table is shown below. Use one if statement to derive the circuit.

ctrl	y1	y0
0 0	x1	x0
0 1	x0	x1
1 0	x0	x0
1 1	x1	x1

5.7 Repeat Problem 5.6, but use a case statement to derive the circuit.

5.8 Consider the following code segment:

```
if (a > b) then
   y <= a - b;
else
   if (a > c) then
      y <= a - c;
   else
      y <= a + 1;
   end if;
end if;
```

(a) Draw the conceptual diagram.
(b) Rewrite the code using two concurrent conditional signal assignment statements.
(c) Rewrite the code using one concurrent conditional signal assignment statement.
(d) Rewrite the code using one case statement.

5.9 Consider the following code segment:

```
y <= (others=>'0');
if (a > b) then
   y <= a - b;
end if;
if (crtl='1') then
   y <= c;
end if;
```

(a) Rewrite the code using one if statement.

(b) Draw the conceptual diagram.

5.10 Assume that op is a 2-bit signal with the `std_logic_vector` data type. Consider the following code segment:

```
case op is
   when "00" =>
      y <= (others => '0');
   when "01" =>
      if (a > 0) then
         y <= a - 1;
      else
         y <= a + 1;
      end if;
   when others =>
      y <= a + b;
end case;
```

(a) Draw the conceptual diagram.

(b) Rewrite the code using concurrent conditional and selected signal assignment statements.

5.11 Consider the shift-left circuit discussed in Problem 4.6. The inputs include a, which is an 8-bit signal to be shifted, and `ctrl`, which is a 3-bit signal specifying the amount to be shifted. Both are with the `std_logic_vector` data type. The output y is an 8-bit signal with the `std_logic_vector` data type. Use an if statement to derive the circuit and draw the conceptual diagram.

5.12 Repeat Problem 5.11, but use a case statement.