10 Concurrent Code

A review of *combinational circuits* was seen in chapter 1. As we know, a circuit is said to be combinational when its output depends uniquely on its preset input, so the circuit has no clock or memory. Regular arithmetic circuits are examples of combinational circuits because the present computation (say, a multiplication) is not affected by previous computations. The material in that chapter will serve as basis for the examples and exercises in this chapter and in the next.

Purely concurrent VHDL code (i.e., without processes) is proper for implementing only combinational circuits, for which the statements *when*, *select*, and *generate* are commonly used. Digital systems, however, usually include sequential circuits, for which sequential code (and therefore sequential statements, like *if*, *case*, and *loop*) is needed. Concurrent code is studied in chapters 10 and 11, while sequential code (which can infer both sequential and combinational circuits) is studied in chapters 12 and 13.

It is also important to emphasize that many combinational circuits can be constructed without any formal statement thanks to the large collection of predefined operators (which are just function calls, as seen in chapter 9) and to the easiness with which data arrays can be constructed and manipulated in VHDL (as seen in chapter 8). As an example of the former, see example 6.1; for the latter, see example 8.1.

10.1 Concurrent Statements

In figure 6.1 we had a global view of a VHDL design code's structure. The architecture body, being the region where the circuits are constructed, is naturally the most complex part, so it is in that region that we concentrate our efforts in this chapter and the next three.

Figure 10.1a shows what can be used to construct an architecture body. It can contain only *concurrent* statements (recall that VHDL is inherently concurrent rather than sequential), which are the following: *when* statements; *select* statements; *generate* statements; *process* statements; *component instantiation* statements; and *procedure call* statements (*block* and *assertion* statements were left out—the first, because of its rare usage; the second,

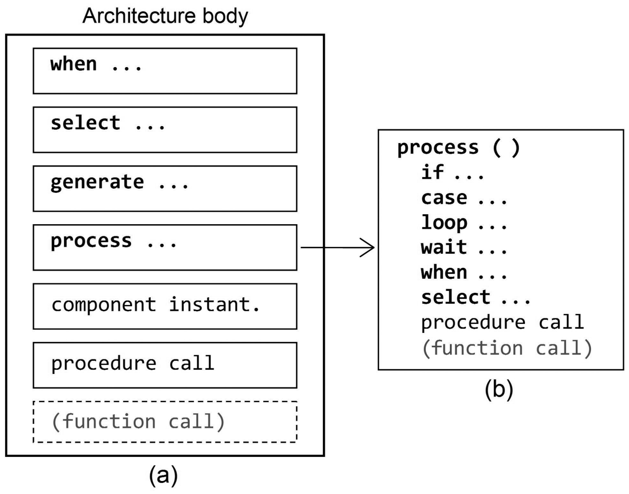

Figure 10.1
(a) Concurrent and (b) sequential VHDL statements, all synthesizable (function call is not a statement but was included for clarity; see the text).

because it is not used to generate hardware but rather to test it, so it will be seen in chapter 14). Because these statements are concurrent, their relative positions in the code do not matter.

Note that *function call* was also included in the list of figure 11.1a. The reason for its being between parentheses is because function calls are not statements but rather part of expressions (chapter 14). For instance, the predefined operators (chapter 9) are function calls.

A special statement in this list is the *process* statement. Although as a whole it is concurrent with respect to all other statements, *internally* it is *sequential*, so only sequential statements (listed in figure 10.1b) can be used inside a process. Note that some statements appear in both (concurrent and sequential) lists.

The formal designations for the concurrent statements of figure 10.1a, according with the IEEE 1076 standard, are shown in table 10.1. Brief comments for each case follow.

Concurrent signal assignment statements: This category contains two very useful statements, called *when* and *select*. They are referred to as *conditional signal assignment* and *selected signal assignment* statements, respectively. Their original (concurrent) version is proper only for the construction of combinational circuits.

Generate statements: As shown in table 10.1, there are three versions of *generate*. The most common is *for-generate*, which acts as a loop that replicates a number of times a section of

Table 10.1
Main concurrent statements

Category/Subcategory		Statements	Studied in
Concurrent signal assignment statements	Conditional assignment	**`when`**	Section 10.2
	Selected assignment	**`select`**	Section 10.3
Generate statements		**`for … generate`**	Section 10.4
		`if … generate`	
		`case … generate`	
Process statement		**`process`**	Section 12.3
Component instantiation statement		`component instant.`	Section 10.5
Concurrent procedure call statement		`procedure call`	Section 14.4

code containing only concurrent statements. In summary, a loop is called *generate* in concurrent code (figure 10.1a) and *loop* in sequential code (figure 10.1b).

Process statement: The only VHDL units that are not (internally) concurrent are *process* and subprograms (the latter consist of *functions* and *procedures*). However, of these, only processes are built directly in the statement part of an architecture body, and, as already mentioned, any process is concurrent with respect to all other statements as a whole. As will be seen in chapter 12, despite being sequential, a process can be used to construct both sequential and combinational circuits.

Component instantiation statement: This statement allows previous designs (which can be combinational or sequential) to be included as part of a new design, hence allowing code reusability and IP inclusion. It also allows the construction of hierarchical (structural) code. Component instantiations are always concurrent, so they cannot be done in sequential units (process and subprograms).

Concurrent procedure call statement: As already mentioned, VHDL subprograms consist of functions and procedures. However, while a function is called as part of an expression (for example, "`if rising_edge(clk) then…`", where *rising_edge* is a function, or "`y <= a + b;`", where "+" is a function), a procedure call is a statement on its own (for example, "`add(a, b, cin, sum, cout);`"). Both (procedure and function) calls are allowed in concurrent (and sequential) code.

The main goal of this chapter is to describe and emphasize all that is needed to design *combinational* circuits using *concurrent* VHDL code. In the same way, chapter 12 will describe and emphasize all that is needed to design both *combinational* and *sequential* circuits using *sequential* VHDL code. This separation is important because, as seen in chapters 1 and 2, such circuits are analyzed and designed differently.

In summary, this chapter describes the following concurrent statements: *when*, *select*, *generate*, and component instantiation. The others (*process* and subprogram calls), since they are internally sequential, are studied separately, in chapters 12–13 and 14, respectively. Two special cases of concurrent code are also discussed in this chapter: (*1*) how to avoid assigning a value to a signal more than once and (*2*) how to implement arithmetic circuits properly in VHDL.

10.2 The *when* Statement

As seen in table 10.1, an assignment using the *when* statement is called a *conditional assignment*. Its syntax is presented below in two versions; note that the first ends with "else value," while the second ends with "when condition."

```
target <= value when condition else
          value when condition else
          value;
```

```
target <= value when condition else
          value when condition else
          value when condition;
```

The target in the syntax above is a signal (VHDL-2008 allows *when* to be used also in sequential code, so there the target can be also a variable). The value can range from a simple static value up to elaborate expressions involving several values. Any number of tests is allowed, but only a few are usually employed (straightforward truth tables, which can be long, should be entered using the *select* statement as described in the next section). Indeed, note in the syntaxes above that the *when* statement has a priority-encoding nature (for any given line to be executed, the tests in all preceding lines must return false) and hence are definitely not tailored for entering straightforward truth tables (though that is not illegal).

A typical use for *when* is shown below (see the complete code in example 7.1). Note that it contains only one test, and the version ending in "else value" is employed.

```
outp <= inp when ena else (others => 'Z');  --multi-bit tri-state buffer (example 7.1)
```

Another example is shown below, using again the syntax ending in "else value." The option on the left (VHDL-2008) is slightly less verbose, while that on the right is slightly clearer. Observe again the priority-encoding nature of *when*; for example, if a = `'1'`, the output is `"01"`, regardless of b, c, and d, so the second line is equivalent to *outp* = `"10"` if $a \neq$ `'1'` and b = `'0'`. Note also that it would take a substantial effort to explicitly describe all possible conditions for a, b, c, and d.

```
outp <= "01" when a else                     outp <= "01" when a='1' else
        "10" when not b else                         "10" when b='0' else
        "11" when c xor d else                       "11" when (c xor d)='1' else
        "--";                                        "--";
```

The advantage of ending with "else value" is that it guarantees complete input-output mapping coverage, so the compiler will not infer latches (explained shortly). On the other hand, the statement ending with "when condition" is more informative because all conditions are shown explicitly. In practice, explicit full-mapping descriptions are often not viable, particularly when using standard-logic types.

An example where explicit full description is viable is shown below and implements the multiplexer of figure 1.3a. The option on the left (ending with "when condition") is obviously more informative than that on the right (ending in "else value"). (Recall, however, that because this is just a straightforward truth table, *select* is the recommended statement to implement it.)

Ending in *when condition*:

```
y <= a when sel=0 else
     b when sel=1 else
     c when sel=2 else
     d when sel=3;
```

Ending in *else value*:

```
y <= a when sel=0 else
     b when sel=1 else
     c when sel=2 else
     d;
```

The next example illustrates incomplete versus complete in-out mapping coverage. If the code on the left is employed, what should the output be, for example, when *rst*, *hold*, and *run* are all low? Since the compiler will execute anyway, it must make a decision that typically is to infer latches to hold the output's current value. Such latches are highly undesirable because they add delays (which, by the way, are poorly predictable in FPGAs because latches are not built-in circuits), besides wasting hardware and power resources. Contrary to that, the code on the right provides a full in-out mapping description (it ends in "else value"), so latch inference does not occur.

Bad (infers latches):

```
outp <= "00" when rst else
        "01" when hold else
        "11" when run;
```

Fine (complete truth table coverage):

```
outp <= "00" when rst else
        "01" when hold else
        "11" when run else
        "--";
```

Note: When the output of a truth table is registered (which can occur in *sequential* circuits), a full truth table description is not necessary (latches will not be inferred because the result is already stored in memory anyway).

The *unaffected* keyword (section 12.5) or, equivalently, the *null* statement can be used with *when*. However, in combinational circuits that causes the inference of latches, so neither should be employed in codes that are for synthesis.

10.3 The *select* Statement

As seen in table 10.1, an assignment using the *select* statement is called a *selected assignment*. Its main use is to enter truth tables, which *select* does better than *when* for two reasons: first, *select* does not posses the priority-encoding nature of *when*; second, full truth table coverage is checked automatically by the compiler, which is not running the compilation if the table is not completely described (hence preventing the inference of latches always).

The syntax for *select* statements is presented below, in two versions. Note that the first ends with "when others," while the second ends with "when choice."

```
with expression select
   target <= value when choice,
             value when choice,
             value when others;
```

```
with expression select
   target <= value when choice,
             value when choice,
             value when choice;
```

The version on the left above employs the keyword *others* to cover all remaining cases, while the version on the right describes the entire truth table explicitly, leading to a more informative code. However, as mentioned previously, in most cases only the former is viable.

The target in the syntax above is a signal (VHDL-2008 allows *select* to be used also in sequential code, so there the target can be also a variable). The value can range from a simple static value up to elaborate expressions involving several values. In the choice expressions, the following can be used: *to* (for ascending index direction), *downto* (descending direction), "|" (interpreted as *or*), and the keyword *others*. Regarding the *unaffected* keyword or, equivalently, the *null* statement, see the comment at the end of section 10.2.

The example below shows an implementation for the multiplexer of figure 1.3a. Since in this case expressing all choices explicitly takes the same effort as using the *others* keyword, the code on the right, being more informative, is preferred.

```
with sel select                    with sel select
   y <= a when 0,                     y <= a when 0,
        b when 1,                          b when 1,
        c when 2,                          c when 2,
        d when others;                     d when 3;
```

The *select?* statement

The matching select statement (*select?*) employs the matching equality comparator (`?=`), which, as seen in section 9.1.3, assumes the following for `std_ulogic` values: `'0'='L'`, `'1'='H'`, and `'-'`=any value. Any other combination returns `'U'` or `'X'` or `'0'`. This statement is particularly useful when the truth table contains "don't care" values at the input, as illustrated in the example below (for "don't care" values at the output, see example 7.2).

Example 10.1. Priority encoder

The code below implements the priority encoder of figure 1.6.a. As usual, it starts with a packages list (lines 2–3, here with just one package), followed by the entity declaration (lines 5–9, using only type `std_logic_vector` for the circuit ports) and the architecture body (lines 11–19). Note the use of *select?* (lines 13–18) because "don't care" inputs are involved. Observe also that the statement ends with "when others," so all truth table entries are covered, allowing *select?* to be synthesized.

```
1   ---------------------------------------------
2   library ieee;
3   use ieee.std_logic_1164.all;
4
5   entity priority_encoder is
6      port (
7         inp: in std_logic_vector(3 downto 0);
8         outp: out std_logic_vector(3 downto 0));
9   end entity;
10
11  architecture lut of priority_encoder is
12  begin
13     with inp select?
14        outp <= "1000" when "1---",
15                "0100" when "01--",
16                "0010" when "001-",
17                "0001" when "0001",
18                "0000" when others;
19  end architecture;
20  ---------------------------------------------
```

10.4 The *generate* Statement

As seen in table 10.1, there are three versions for this statement: *for-generate*, *if-generate*, and *case-generate*. They are described below.

for-generate:

```
label: for identifier in generate_range generate
   [generate_declarative_part
begin]
   concurrent_statements
end generate [label];
```

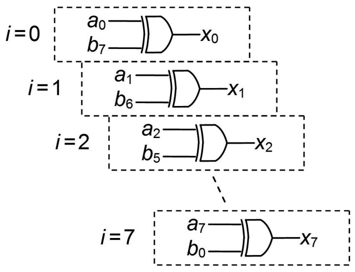

Figure 10.2
Hardware produced by a *generate* statement.

This is the most frequently used form of *generate*. It acts as a loop, but because it is a concurrent statement, a piece of hardware is inferred every time the loop goes around (hence *generate* is indeed a well-chosen name).

An example is shown below, with two equivalent versions (that on the right uses the *range* and *left* attributes, seen in section 9.3.2, which help in the construction of parameterized code). Each time the generate loop goes around, an XOR gate is inferred, so the circuit of figure 10.2 is produced.

```
signal a, b, x: std_ulogic_vector(7 downto 0);
...                                    ...
gen: for i in 0 to 7 generate          gen: for i in x'range generate
   x(i) <= a(i) xor b(7-i);               x(i) <= a(i) xor b(b'left-i);
end generate;                          end generate;
```

The hardware instantiated by the *generate* statement can be an entire design that we want to include as part (i.e., a *component*, in VHDL language) of a new, larger design. That will be seen in section 10.5.

The other two forms of *generate* are shown below.

if-generate:

```
label: if condition generate
  concurrent_statements;
[elsif condition generate
  concurrent_statements;]
[else generate
  concurrent_statements;]
end generate [label];
```

case-generate:

```
label: case expression generate
   when choice =>
      concurrent_statements;
   when choice =>
      concurrent_statements;
    ...
end generate [label];
```

These are conditional forms of *generate*. The syntaxes for *if* and *case* are similar to those for the sequential *if* and *case* statements (chapter 12). An application for these *generate* statements is to choose between different hardware specifications determined, for example, by a generic constant, as in the example below.

Example 10.2. Conditional adder instantiation
Say that we have a library of standard cells, among which are unsigned and signed adders of generic size. The unsigned version, called *adder_unsigned*, is shown in the first of the two codes below (to obtain the other version, just replace the word "unsigned" with "signed"). The number of bits is determined by the generic constant *WIDTH* (line 8), whose value can be left unspecified because it can/will be overwritten by the *generic map* association during instantiation.

The second code below is the main code, in which one of these adders is instantiated. The selection is made by a generic constant, called *POLARITY* (line 7), which points to the unsigned version when low (lines 18–19) or the signed version otherwise (lines 20–21). Note that the instantiation is done using the conditional *case-generate* statement (lines 17–22); another option would be to use the *if-generate* statement, but for simple cases the former is preferred. (Component instantiation details are presented in the next section.)

```
 1  --------------------------------------------------------
 2  library ieee;
 3  use ieee.std_logic_1164.all;
 4  use ieee.numeric_std.all;
 5
 6  entity adder_unsigned is
 7     generic (
 8        WIDTH: natural);
 9     port (
10        in1, in2: in std_logic_vector(WIDTH-1 downto 0);
11        sum: out std_logic_vector(WIDTH-1 downto 0));
12  end entity adder_unsigned;
13
14  architecture adder_unsigned of adder_unsigned is
15  begin
16     sum <= std_logic_vector(unsigned(in1) + unsigned(in2));
17  end architecture adder_unsigned;
18  --------------------------------------------------------
```

```
 1  -------------------------------------------------------------------------------
 2  library ieee;
 3  use ieee.std_logic_1164.all;
 4
 5  entity adder is
```

```
   generic (
      POLARITY: std_logic := '1';
      NUM_BITS: natural := 32);
   port (
      a, b: in std_logic_vector(NUM_BITS-1 downto 0);
      sum: out std_logic_vector(NUM_BITS-1 downto 0));
end entity adder;

architecture with_std_cell of adder is
begin

   gen_adder: case POLARITY generate
      when '0' =>
        adder: entity work.adder_unsigned generic map (NUM_BITS) port map (a, b, sum);
      when others =>
        adder: entity work.adder_signed generic map (NUM_BITS) port map (a, b, sum);
   end generate;

end architecture with_std_cell;
----------------------------------------------------------------------------------
```

10.5 Component Instantiation Statements

As seen in figure 10.1, *component instantiation statements* are also concurrent statements, so they are not allowed in sequential units (i.e., process and subprograms). A single instantiation can be done directly, but looped instantiations require the *generate* statement, which, as we saw, plays the role of loop in concurrent code.

A previous design (i.e., a complete VHDL code) can be instantiated as part of another design in two equivalent ways: using a *component* instantiation or using a *design entity* instantiation. Both cases are described next.

10.5.1 Component Instantiation

To instantiate a component, a component declaration plus a component instantiation statement are needed, as shown below.

```
component component_name [is]
   [generic (...);]
   port (...);
end component [component_name];
```

```
label: [component] component_name
   [generic map (generic_association_list)]
   port map (port_association_list);
```

The component declaration (on the above left) consists simply of a copy of the entity declaration of the design to be instantiated with the word *entity* replaced with the word *component*. The component name is the same as its entity's name. The typical location for this declaration is the declarative part of architecture, package, or generate statement.

The component instantiation statement (syntax on the above right) provides the (possibly new) names and values for the generic parameters by means of *generic map*, and the port mapping between the instantiated design and the current design by means of *port map*. The mapping associations can be *named* or *positional*; in the former, each association is explicitly named, while in the latter each position of the instantiated design is associated to the same position in the current design. Finally, if a port must be left unconnected in the instantiation, the keyword *open* should be used for it in the port map.

The example below shows, on the left, an entity called *brick*, of a design that is going to be instantiated by another design, whose entity, called *wall*, is shown on the right:

```
entity brick is                      entity wall is
   generic (                            generic (
      NUM_BITS: positive);                 WIDTH: positive := 32);
   port (                               port (
      a, b: in ...;                        x, y: in ...;
      c: out ...);                         z: out ...);
end entity brick;                    end entity wall;
```

Corresponding component instantiations, labeled *comp*, are shown below:

```
--Component instantiation with named association:
comp: brick generic map (NUM_BITS => WIDTH) port map (a => x, b => y, c => z);

--Component instantiation with positional association:
comp: brick generic map (WIDTH) port map (x, y, z);
```

Signal expressions in port map are allowed after VHDL-2008, as illustrated below:

```
... port map (x1 => y1, x2 => y2 and y3, x3 => to_unsigned(y4));
```

A complete component instantiation will be presented in example 10.3.

10.5.2 Design Entity Instantiation

This is another way of instantiating one design as part of another. As shown in the syntax below, an advantage here is that the component declaration is not needed. The instantiated code is allowed to have multiple architectures because that of interest to the present design can be specified in the optional *architecture_name* field. The *work.entity_name* declaration assumes that the *entity_name.vhd* file is present in the project directory.

```
label: entity work.entity_name [(architecture_name)]
  [generic map (generic_association_list)]
  port map (port_association_list);
```

The instantiations below are again for the *brick* and *wall* circuits, so they can be compared to the previous component instantiations.

```
--Design entity instantiation with named association:
comp: entity work.brick generic map (NUM_BITS => WIDTH) port map (a => x, b => y, c => z);

--Design entity instantiation with positional association:
comp: entity work.brick generic map (WIDTH) port map (x, y, z);
```

Example 10.3. Carry-ripple adder built with full-adder components

The full-adder unit and the carry-ripple adder were reviewed in sections 1.5.1 and 1.5.2, respectively. In this example we use the full adder of figure 1.12a to build the carry-ripple adder of figure 1.13b.

Solution 1: Using design entity instantiation

The first code below is for the full-adder unit, with the outputs calculated using boolean equations (lines 13–14). Note that *std_logic_1164* is the only package needed in the packages list (lines 2–3). The second code is for the carry-ripple adder. In the entity declaration (lines 5–13), a generic constant called *NUM_BITS* (line 7) is used to specify the number of bits (which is also the number of full-adder units) in this circuit. In the architecture body (lines 15–24), a signal called *carry* is declared (line 16) to represent the internal circuit nodes (carry interface between the full-adder units). Next, a *for-generate* statement (lines 19–22) is used to do the instantiations (recall that a loop is called *loop* in sequential code, but it is called *generate* in concurrent code). Finally, observe that the instantiation (lines 20–21) was done using the *design entity instantiation* option (seen in this section), with positional association in the port map.

```
1  ---------------------------------------------------------------
2  library ieee;
3  use ieee.std_logic_1164.all;
4
5  entity full_adder_unit is
6     port (
7        in1, in2, cin: in std_logic;
8        sum, cout: out std_logic);
9  end entity;
```

```

architecture boolean of full_adder_unit is
begin
   sum <= in1 xor in2 xor cin;
   cout <= (in1 and in2) or (in1 and cin) or (in2 and cin);
end architecture;
--------------------------------------------------------------
```

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

entity carry_ripple_adder is
   generic (
      NUM_BITS: natural := 8);
   port (
      a, b: in std_logic_vector(NUM_BITS-1 downto 0);
      cin: in std_logic;
      sum: out std_logic_vector(NUM_BITS-1 downto 0);
      cout: out std_logic);
end entity;

architecture structural of carry_ripple_adder is
   signal carry: std_logic_vector(0 to NUM_BITS);
begin
   carry(0) <= cin;
   gen_adder: for i in 0 to NUM_BITS-1 generate
      adder: entity work.full_adder_unit
         port map (a(i), b(i), carry(i), sum(i), carry(i+1));
   end generate;
   cout <= carry(NUM_BITS);
end architecture;
--------------------------------------------------------------
```

Solution 2: Using component instantiation

The architecture below shows the modifications needed in the code above to use the *component instantiation* option, studied in the previous section. It requires a component declaration (lines 17–21) plus a component instantiation statement (line 25). In this case, the component declaration (which is just a copy of the full-adder's entity) is located in the declarative part of the architecture. Another option is to locate it in the declarative part of the generate statement (the keyword *begin* is then required), as shown in the subsequent code. Recall that still another popular place for component declarations is a package.

```
architecture structural of carry_ripple_adder is
   signal carry: std_logic_vector(0 to NUM_BITS);
   component full_adder_unit is
      port (
         in1, in2, cin: in std_logic;
         sum, cout: out std_logic);
   end component;
begin
   carry(0) <= cin;
   gen_adder: for i in 0 to NUM_BITS-1 generate
      adder: full_adder_unit port map (a(i), b(i), carry(i), sum(i), carry(i+1));
   end generate;
   cout <= carry(NUM_BITS);
end architecture;
---------------------------------------------------------------------------------
```

```
architecture structural of carry_ripple_adder is
   signal carry: std_logic_vector(0 to NUM_BITS);
begin
   carry(0) <= cin;
   gen_adder: for i in 0 to NUM_BITS-1 generate
      component full_adder_unit is
         port (
            in1, in2, cin: in std_logic;
            sum, cout: out std_logic);
      end component;
   begin
      adder: full_adder_unit port map (a(i), b(i), carry(i), sum(i), carry(i+1));
   end generate;
   cout <= carry(NUM_BITS);
end architecture;
---------------------------------------------------------------------------------
```

10.6 Avoiding Multiple Assignments to the Same Signal

This section and the next discuss two special cases related to concurrent code. The first (in this section) introduces a way to circumvent the fact that a signal cannot receive multiple assignments in concurrent code. The second (next section) shows recommendations regarding the implementation of arithmetic circuits in VHDL.

As we know, VHDL code is inherently concurrent, so any distribution of the statements (listed in figure 10.1) must lead to the same result. Consequently, we cannot assign a value to a signal somewhere in the code and then assign another value to it later, believing that the compiler should simply consider the last value as the valid one because that could lead to different circuits depending on the relative positions of the statements (that would be fine only in regions of *sequential* code, studied in chapters 12–13).

A solution when multiple assignments are necessary is to create an *internal* signal with an *extra* dimension (for example, with dimension 1D × 1D if the target signal is 1D–see figure 8.1) and use it to do the computations, passing then the last element value of that signal to the target signal. This approach is illustrated in the example below.

Note: It will be shown in chapter 12 (example 12.11, process P1) that the technique above is not necessary when using sequential code and a variable to do the computations, which simplifies the solution.

Example 10.4. Hamming-weight calculator

The code below implements a circuit that determines the Hamming weight (HW) of a vector, which is the number of 1s in it. The input and output are *inp_vector* and *hamm_weight*, respectively.

Solution 1: As usual, the code starts with declarations regarding the packages needed in the design (lines 2–4); *std_logic_1164* is needed because the type `std_logic_vector` is used in the circuit ports (lines 11–12), while *numeric_std* is needed because the type `unsigned` is employed for type conversion (line 23—see details about `integer` to `std_logic_vector` conversion in section 7.10.3).

The entity declaration (lines 6–13) starts with a generic list (lines 8–9), which allows the construction of a parameterized code. Note that if the minimum number of bits is wanted for *BITS_OUT*, then *BITS_OUT* is no longer an independent constant but rather a function of *BITS_IN*—that is, $BITS\_OUT = \lceil log_2(BITS\_IN + 1) \rceil$ (see comments after the code). Next come the circuit ports (lines 11–12) using only the type `std_logic_vector`.

The architecture body (lines 15–24) starts with type and signal declarations (lines 16–17). This 1D × 1D signal (an array of integers) is needed because, to follow the strategy just described, a signal with an extra dimension with respect to the signal to be measured (an integer, hence 1D, according to figure 8.1) is needed. The statement part employs a loop (lines 20–22—recall that loops are produced by the *generate* statement in concurrent code) to produce the internal values, the last of which is passed to the output (line 23).

```
1  ---------------------------------------------------------------------------
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.numeric_std.all;
5
6  entity hamming_weight_calculator is
7     generic (
8        BITS_IN: positive := 16;
```

```
      BITS_OUT: positive := 5);   --calculated by user as ceil(log2(BITS_IN+1))
   port (
      inp_vector: in std_logic_vector(BITS_IN-1 downto 0);
      hamm_weight: out std_logic_vector(BITS_OUT-1 downto 0));
end entity;

architecture concurrent of hamming_weight_calculator is
   type integer_array is array (0 to BITS_IN) of integer range 0 to BITS_IN;
   signal internal: integer_array;
begin
   internal(0) <= 0;
   gen: for i in 1 to BITS_IN generate
      internal(i) <= internal(i-1) + 1 when inp_vector(i-1) else internal(i-1);
   end generate;
   hamm_weight <= std_logic_vector(to_unsigned(internal(BITS_IN), BITS_OUT));
end architecture;
---------------------------------------------------------------------------------
```

Solution 2: An option to improve the code above is to employ an expression in the generic list, as shown below, but check restrictions and other comments as noted in section 6.7.

```
use ieee.math_real.all;
...
   generic (
      BITS_IN: positive := 16;
      BITS_OUT: positive := integer(ceil(log2(real(BITS_IN+1)))); --a dependent constant
   port (...
```

Solution 3: The most formal solution is to list only the truly independent constants in the generic list, as shown below, leaving the log2 computation for the range specifications (hence with increased verbosity):

```
entity hamming_weight_calculator is
   generic (
      BITS_IN: positive := 16);
   port (
      inp_vector: in std_logic_vector(BITS_IN-1 downto 0);
      hamm_weight: out std_logic_vector(integer(ceil(log2(real(BITS_IN+1))))-1 downto 0)));
end entity;

architecture concurrent of hamming_weight_calculator is
   constant BITS_OUT: positive := integer(ceil(log2(real(BITS_IN+1))));
   ...
```

10.7 Suggested Approaches for Arithmetic Circuits

This is the second of the two special cases mentioned previously regarding concurrent code. It deals with arithmetic circuits, for which implementation suggestions are presented below.

Arithmetic circuits, reviewed in sections 1.5 and 1.6, are those for which sign matters. Therefore, the use of the type `integer`, for example, is not a good idea because the code then will not show explicitly whether the system is signed or unsigned (that is determined by the compiler upon inspecting the range specified for the involved integers).

The types recommended for arithmetic circuits, with the respective packages of origin (which must then be included in the code's packages list—the second option for some of the packages below is for default parameters) are the following:

- For integer arithmetic: `unsigned` or `signed` (package *numeric_std*).
- For fixed-point arithmetic: `ufixed` or `sfixed` (package *fixed_generic_pkg* or *fixed_pkg*).
- For floating-point arithmetic: `float` (package *float_generic_pkg* or *float_pkg*).

For conciseness, in this section we will refer to these types as "arithmetic" types.

It is important, however, to try to always use the same types for the interface signals (circuit ports) to allow direct connection between system blocks (in large projects) and help reusability. The standard-logic types (`std_ulogic`, `std_logic`, `std_ulogic_vector`, and `std_logic_vector`) are here considered the default types for that role. Only in particular cases should the arithmetic types be used directly in the circuit ports (for example, when it is a stand-alone design and reusability is not an issue or when a port type must be a user-defined type).

Suggested procedure

Before the VHDL code:

1) Make sure that the circuit is arithmetic and decide which "arithmetic" type to use (see comment on integer versus floating point, presented next).

2) List all arithmetic operations involved (as seen in section 9.1.2, the predefined arithmetic operators are `+`, `–`, `*`, `/`, `**`, *rem*, *mod*, and *abs*). Then check, using table 9.4, the constraints for each operator; for example, for types `unsigned` and `signed`, the "+" and "–" operators require the result to have the same number of bits as the largest operand, while the "*" operator requires the number of bits in the result to be equal to the sum of the bits in the operands.

3) Decide how to deal with overflow (section 1.6.1) in case the constraints above might cause the circuit to be subject to overflow. If overflow is not acceptable, one solution is to extend the operands and the result (the *resize* function is in section 7.9.3); if it is acceptable but must be flagged, decide how that flag will be produced.

In the VHDL code:

1) As a general rule, use only standard-logic types for the circuit ports (see possible exceptions listed above).

2) In the architecture, convert the `std_ulogic_vector` or `std_logic_vector` inputs to one of the arithmetic types. Recall that single-bit standard-logic types (`std_ulogic` and `std_logic`) do not need any conversion.

3) Do the computations.

4) Convert the multi-bit results to standard-logic vector types and send them out (along with all single-bit standard-logic results, of course).

5) Carefully simulate your design.

Examples of arithmetic circuits implemented using the suggestions above are presented after the following comment.

Integer (or fixed-point) versus floating-point Floating point should be avoided whenever possible because of its high consumption of resources (hardware, primarily, but also some power; speed might also be impaired). One procedure that might help that decision is to define some error parameter for the target application and establish a maximum acceptable value for it and then run a corresponding analysis tool (Matlab, for example). Say that the analysis starts with 32-bit floating point, which passes the maximum-error test; the number of bits in the exponent and fraction should then be reduced gradually until the error limit is reached. This should then be repeated for integer (or fixed point) until the minimum number of bits is again obtained. Both should then be implemented in VHDL to check the amount of hardware consumed for the same device or technology in each case (plus other parameters, like maximum speed). As a practical example, this procedure could be applied to the sine calculator of example 11.3, for which the total harmonic distortion (THD) could be used as error parameter.

Example 10.5. Signed integer adder

Figure 10.3a shows a fully equipped *N*-bit signed integer adder, where *a* and *b* are the addends, *cin* and *cout* are the carry-in and carry-out bits, *sum* is the core result, and *oflow* is an overflow flag.

Note that this adder contains all optional outputs (*cout*, *oflow*, and *sum*(*N*)), as seen in section 1.6.3; even though this kind of cell is usually not built with all three (see figure 1.21b), the purpose here is to practice with all possibilities.

The time behavior is illustrated in figure 10.3b, where the values of *a* and *b* are {–8, 0, 7, –8, 0, 7} and {–8, –7, 5}, respectively. It is left to the reader to complete it for later comparison against simulation results obtained after synthesizing the code that follows. (Signed integer addition was reviewed in section 1.6.3, and signed/unsigned types were studied in section 7.6.3).

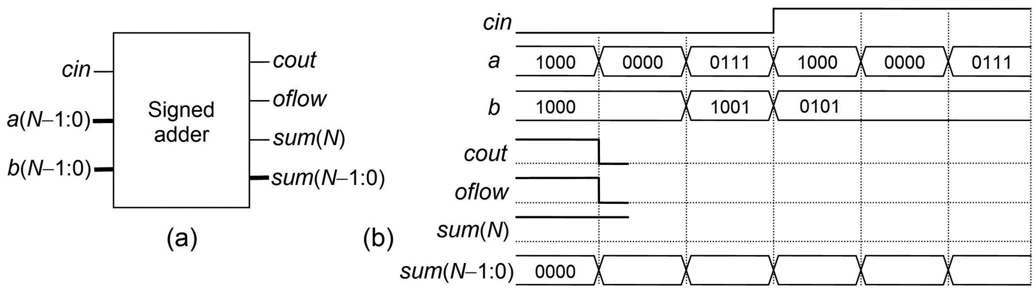

Figure 10.3
Signed adder of example 10.5.

Since this is an arithmetic circuit, we will follow the suggestions presented above. The right type is `signed`, and the only operator to be used is "+". Consulting table 9.4, we observe in comment (8) that the size of the output must be the same as that of the largest input.

A VHDL solution is presented below. To make it clear, a step-by-step code is shown. The packages list (lines 2–4) contains the packages *std_logic_1164* (because standard-logic types are employed for the circuit ports) and *numeric_std* (because type `signed` is used in the computations). In the entity declaration (lines 6–14), a generic constant (line 8; *N* is called *NUM_BITS* in the code) defines the number of bits in the multi-bit signals. Also, to make it clear that *sum*(*N*) is an optional output, it is called *sumMSB* (sum's new most significant bit, if used) in the code.

The architecture body (lines 16–30) starts with the declaration of a signal called *sum_sig* (line 17), with *NUM_BITS* + 1 bits, which will be used to hold the sum temporarily. The sum (line 21) employs sign-extension and conversion to `signed`. The MSB of this extended sum (with *NUM_BITS* + 1 bits) is *sumMSB* (line 26), while all other bits constitute the original sum (line 25). Notice that the sum is converted to `std_logic_vector` before being sent out. Finally, the *cout* (line 27) and *oflow* (line 28) outputs are calculated using the equations of figure 1.22a.

```
 1  ---------------------------------------------------------------------
 2  library ieee;
 3  use ieee.std_logic_1164.all;
 4  use ieee.numeric_std.all;
 5
 6  entity adder_signed is
 7     generic (
 8        NUM_BITS: integer := 4);
 9     port (
10        a, b: in std_logic_vector(NUM_BITS-1 downto 0);
11        cin: in std_logic;
12        sum: out std_logic_vector(NUM_BITS-1 downto 0);
13        cout, oflow, sumMSB: out std_logic);
14  end entity;
```

```
15
16  architecture suggested of adder_signed is
17     signal sum_sig: signed(NUM_BITS downto 0);
18  begin
19
20     --Sign-extension, conversion to signed, and addition:
21     sum_sig <= signed(a(NUM_BITS-1) & a) +  signed(b) + cin;
22     --sum_sig <= resize(signed(a), NUM_BITS+1) +  signed(b) + cin;
23
24     --Conversion to std_logic_vector plus single-bit calculations:
25     sum <= std_logic_vector(sum_sig(NUM_BITS-1 downto 0));
26     sumMSB <= sum_sig(NUM_BITS);
27     cout <= a(NUM_BITS-1) xor b(NUM_BITS-1) xor sumMSB;
28     oflow <= sumMSB xor sum(NUM_BITS-1);
29
30  end architecture;
31  ----------------------------------------------------------------------
```

Four additional observations about line 21 follow. First, an equivalent construction using the *resize* function (section 7.9.3) is shown in line 22. Second, table 9.4 tells us that a `signed` value can be added to another `signed`, `integer`, or `std_ulogic` value, so the fact that the sum includes a single-bit value is fine, but that was not so before VHDL-2008; if your compiler does not support that feature yet, a solution is to extend that bit by using `('0' & cin)` or `('0', cin)`. The third observation is that only one of the inputs needs sign-extension because the output of addition for `signed` is required to have the same number of bits as the largest input. The final observation regards the number of adders: even though two sums appear in line 21, the compiler will understand that the last addend is just a carry bit, so a single adder with carry-in port will be inferred. Just out of curiosity, below is a solution using a single "+" operator, where an extra bit is appended at the right end of *a* and *b*, with *cin* in one addend and a `'1'` in the other (*cin* in both would also do). Observe below that *sum_sig* (line 17) has now *NUM_BITS* + 2 bits and that the sum's LSB is ignored in all output equations.

```
16  architecture with_one_sum of adder_signed is
17     signal sum_sig: signed(NUM_BITS+1 downto 0);
18  begin
19     sum_sig <= signed(a(NUM_BITS-1) & a & cin) + signed(b & '1');
20     sum <= std_logic_vector(sum_sig(NUM_BITS downto 1));
21     sumMSB <= sum_sig(NUM_BITS);
22     cout <= a(NUM_BITS-1) xor b(NUM_BITS-1) xor sumMSB;
23     oflow <= sumMSB xor sum(NUM_BITS-1);
24  end architecture;
```

Example 10.6. Floating-point adder and multiplier

This example shows the implementation of a floating-point (FP) adder/multiplier. Since this too is an arithmetic circuit, the suggestions presented above will again be followed. (FP arithmetic was reviewed in section 1.6.5 and FP types were seen in section 7.6.5.)

The data type now is `float`, and the operators involved in this circuit are "`+`" and "`*`". Consulting table 9.4, we see in comment (28) that for both operators the output's upper and lower bounds must be equal to the inputs' largest and smallest bounds, respectively (though the compiler might require only the resulting vector length to be obeyed).

The first solution below employs the type `float` directly in the circuit ports (lines 7–8). The only package then needed in the packages list (lines 2–3) is *float_pkg*, which employs the default FP parameters (for example, the rounding style is *roundTiesToEven*, explained in section 1.6.5; as seen in section 7.6.5, the generic package is called *float_generic_pkg*). The inputs and outputs range is "5 downto –3" (lines 7–8), so the data representation is (S)(EEEEE)(FFF) for all, including the sum and multiplication (computed in lines 13–14). Because the exponent uses $E_{width} = 5$ bits, this circuit parameters are (see section 1.6.5) $E_{min} = 1$ (always), $E_{max} = 30$, $MAX = 31$, $BIAS = 15$, and $2^{-14} \le dec \le (2 - 2^{-3})2^{15}$.

In the second solution below, standard-logic types are employed for the circuit ports (lines 8–9), which are generally preferred. Note the inclusion of the *std_logic_1164* package (line 3) in the packages list.

It is also important to mention that free compiler versions usually have less VHDL support than their paid counterparts, so FP support might be available only in the latter.

```
 1  ---------------------------------------------------------------------------
 2  library ieee;
 3  use ieee.float_pkg.all;
 4
 5  entity fp_adder_multiplier is
 6     port (
 7        a, b: in float(5 downto -3);
 8        sum, prod: out float(5 downto -3));
 9  end entity;
10
11  architecture fp_arithmetic of fp_adder_multiplier is
12  begin
13     sum <= a + b;
14     prod <= a * b;
15  end architecture;
16  ---------------------------------------------------------------------------
```

```
 1  ---------------------------------------------------------------------------
 2  library ieee;
 3  use ieee.std_logic_1164.all;
```

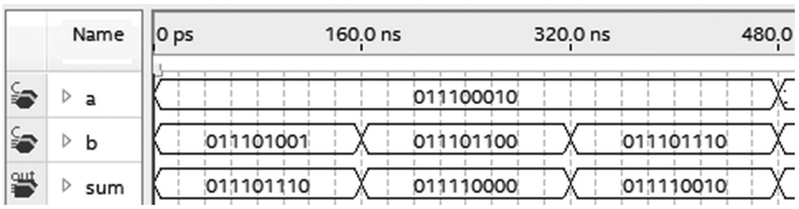

Figure 10.4
Simulation results from the adder part of example 10.6.

```
use ieee.float_pkg.all;

entity fp_adder_multiplier is
   port (
      a, b: in std_logic_vector(8 downto 0);         --for float(5 downto -3)
      sum, prod: out std_logic_vector(8 downto 0));  --for float(5 downto -3)
end entity;

architecture fp_arithmetic of fp_adder_multiplier is
begin
   sum <= to_std_logic_vector(to_float(a, 5, 3) + to_float(b, 5, 3));
   prod <= to_std_logic_vector(to_float(a, 5, 3) * to_float(b, 5, 3));
end architecture;
---------------------------------------------------------------------------------
```

Simulation results for the adder part of the code above are shown in figure 10.4. The inputs are: $a = (0)(11100)(010)$, so $E = 28$ and $F = 1/4$; $b_1 = (0)(11101)(001)$, so $E = 29$ and $F = 1/8$; $b_2 = (0)(11101)(100)$, so $E = 29$ and $F = 1/2$; and $b_3 = (0)(11101)(110)$, so $E = 29$ and $F = 3/4$. *BIAS* is 15 in all cases. Note that these values are precisely those employed in the last example of section 1.6.5, from which the expected results, after truncation and rounding (using the *roundTiesToEven* style, which is the default and recommended style for FP arithmetic), are: $sum_1 = a + b_1 = 1.110{\cdot}2^{14} = (0)(11101)(110)$; $sum_2 = a + b_2 = 1.000{\cdot}2^{15} = (0)(11110)(000)$; and $sum_3 = a + b_3 = 1.010{\cdot}2^{15} = (0)(11110)(010)$. These values are in perfect agreement with the simulation results of figure 10.4.

10.8 Additional Examples and Exercises

These are in chapter 11, which is dedicated entirely to practicing with concurrent code. The list of all enumerated examples and exercises in this edition of the book is in appendix M.