VHDL Summary

19

Objective: This chapter concisely describes the VHDL language and presents some introductory circuit synthesis examples. Its purpose is to lay the fundamentals for the many designs that follow in Chapters 20 and 21, for *combinational* circuits, and in Chapters 22 and 23, for *sequential* circuits. The use of VHDL is concluded in Chapter 24, which introduces simulation techniques with VHDL *testbenches*. The descriptions below are very brief; for additional details, books written specifically for VHDL ([Pedroni04a] and [Ashenden02], for example) should be consulted.

Chapter Contents

The summary presented in this chapter can be divided into two parts [Pedroni04a]. The first part, which encompasses Sections 19.1 to 19.11, plus Section 19.16, describes the VHDL statements and constructs that are intended for the main code, hence referred to as *circuit-level design*. The second part, covered by Sections 19.12 to 19.15, presents the VHDL units that are intended mainly for libraries and code partitioning, so it is referred to as *system-level design*.

19.1 About VHDL

VHDL is a technology and vendor independent *hardware description language*. The code describes the behavior or structure of an electronic circuit from which a compliant physical circuit can be inferred by a compiler. Its main applications include synthesis of digital circuits onto CPLD/FPGA chips and layout/mask generation for ASIC (application-specific integrated circuit) fabrication.

VHDL stands for VHSIC hardware description language and resulted from an initiative funded by the U.S. Department of Defense in the 1980s. It was the first hardware description language standardized by the IEEE, through the 1076 and 1164 standards.

VHDL allows circuit *synthesis* as well as circuit *simulation*. The former is the translation of a source code into a hardware structure that implements the specified functionalities; the latter is a testing procedure to ensure that such functionalities are achieved by the synthesized circuit. In the descriptions that follow, the synthesizable constructs are emphasized, but an introduction to circuit simulation with VHDL is also included (Chapter 24).

The following are examples of EDA (electronic design automation) tools for VHDL synthesis and simulation: Quartus II from Altera, ISE from Xilinx, FPGA Advantage, Leonardo Spectrum (synthesis), and ModelSim (simulation) from Mentor Graphics, Design Compiler RTL Synthesis from Synopsys, Synplify Pro from Synplicity, and Encounter RTL from Cadence.

All design examples presented in this book were synthesized and simulated using Quartus II Web Edition, version 6.1 or higher, available free of charge at www.altera.com. The designs simulated using testbenches (Chapter 24) were processed with ModelSim-Altera Web Edition 6.1g, also available free of charge at the same site.

19.2 Code Structure

This section describes the basic structure of VHDL code, which consists of three parts: library declarations, entity, and architecture.

Library declarations

Library declarations is a list of all libraries and corresponding packages that the compiler will need to process the design. Two of them (*std* and *work*) are made visible by default. The *std* library contains definitions for the standard data types (`BIT`, `BOOLEAN`, `INTEGER`, `BIT`, `BIT_VECTOR`, etc.), plus information for text and file handling, while the *work* library is simply the location where the design files are saved.

A package that often needs to be included in this list is *std_logic_1164*, from the *ieee* library, which defines a nine-value logic type called `STD_ULOGIC` and its resolved subtype, `STD_LOGIC` (the latter is the industry standard). The main advantage of `STD_LOGIC` over `BIT` is that it allows high-impedance (`'Z'`) and "don't care" ('–') specifications.

To declare the package above (or any other) two lines of code are needed, one to *declare* the library and the other a *use* clause pointing to the specific package within it, as illustrated below.

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

Entity

Entity is a list with specifications of all I/O ports of the circuit under design. It also allows generic parameters to be declared, as well as several other declarations, subprogram bodies, and concurrent statements. Its syntax is shown below.

```
ENTITY entity_name IS
   GENERIC (constant_name: constant_type := constant_value;
            constant_name: constant_type := constant_value;
            ...);
   PORT (port_name: signal_mode signal_type;
         port_name: signal_mode signal_type;
         ...);
  [declarative part]
  [BEGIN]
     [statement part]
END entity_name;
```

- GENERIC: Allows the declaration of generic constants, which can then be used anywhere in the code, including in PORT. Example: GENERIC (number_of_bits: INTEGER := 16).
- PORT, *signal mode*: IN, OUT, INOUT, BUFFER. The first two are truly unidirectional, the third is bidirectional, and the fourth is needed when an output signal must be read internally.
- PORT, *signal type*: BIT, BIT_VECTOR, INTEGER, STD_LOGIC, STD_LOGIC_VECTOR, BOOLEAN, etc.
- Declarative part: Can contain TYPE, SUBTYPE, CONSTANT, SIGNAL, FILE, ALIAS, USE, and ATTRIBUTE declarations, plus FUNCTION and PROCEDURE bodies. Rarely used in this way.
- Statement part: Can contain concurrent statements (rarely used).

Architecture

This part contains the code proper (the intended circuit's structural or behavioral description). It can be *concurrent* or *sequential*. The former is adequate for the design of *combinational* circuits, while the latter can be used for *sequential* as well as *combinational* circuits. Its syntax is shown below.

```
ARCHITECTURE architecture_name OF entity_name IS
  [declarative part]
BEGIN
  (code)
END architecture_name;
```

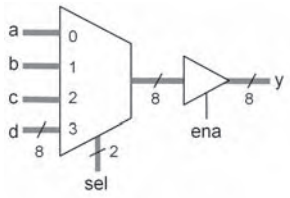
- Declarative part: Can contain the same items as the entity's declarative part, plus COMPONENT and CONFIGURATION declarations.
- Code: Can be concurrent, sequential, or mixed. To be sequential, the statements must be placed inside a PROCESS. However, as a whole, VHDL code is always concurrent, meaning that all of its parts are treated in "parallel" with no precedence. Consequently, any process is compiled concurrently with any other statements located outside it. The only other option to construct sequential VHDL code is by means of *subprograms* (FUNCTION and PROCEDURE), described in Sections 19.14 and 19.15.

To write purely concurrent code, *operators* (Section 9.6) can be used as well as the `WHEN` and `GENERATE` statements. To write *sequential* code (inside a `PROCESS`, `FUNCTION`, or `PROCEDURE`), the allowed statements are `IF`, `CASE`, `LOOP`, and `WAIT` (plus operators).

The code structure described above is illustrated in the example that follows.

EXAMPLE 19.1 BUFFERED MULTIPLEXER

This introductory example shows the design of a 4×8 multiplexer (Section 11.6) whose output passes through a tri-state buffer (Section 4.8) controlled by an output-enable (*ena*) signal. The circuit is depicted in Figure 19.1(a), and the desired functionality is expressed in the truth table of Figure 19.1(b).



(a)

ena	sel	y
0	X	"ZZZZZZZZ"
1	0	a
	1	b
	2	c
	3	d

(b)

FIGURE 19.1. Buffered multiplexer of Example 19.1.

SOLUTION

A VHDL code for this example is presented in Figure 19.2. Note that it contains all three code sections described above. The additional package is precisely *std_logic_1164* (lines 2 and 3), which is needed because the high-impedance state (`'Z'`) is employed in this design.

The entity is in lines 5–10, under the name *buffered_mux* (any name can be chosen) and contains six input signals and one output signal (note the modes `IN` and `OUT`). Signals *a* to *d* are 8-bits wide and of type `STD_LOGIC_VECTOR`, and so is *y* (otherwise none of the inputs *a*–*d* could be passed to it); in *a* to *d* and *y* the indexing is from 7 down to 0 (in VHDL, the leftmost bit is the MSB). The type of *sel* was declared as `INTEGER`, though it could also be `STD_LOGIC_VECTOR(1 DOWNTO 0)`, among other options. Finally, *ena* was declared as `STD_LOGIC` (single bit), but `BIT` would also do.

The architecture is in lines 12–21, with the name *myarch* (can be basically any name, including the same name as the entity's). In its declarative part (between the words `ARCHITECTURE` and `BEGIN`) an internal signal (*x*) was declared, which plays the role of multiplexer output. The `WHEN` statement (described later) was employed in lines 15–18 to implement the multiplexer and in lines 19 and 20 to implement the tri-state buffer. Because all eight wires that feed *y* must go to a high-impedance state when *ena* = '0', the keyword `OTHERS` was employed to avoid repeating `'Z'` eight times (that is, `y <="ZZZZZZZZ"`).

Note that single quotes are used for single-bit values, while double quotes are used for multibit values. Observe also that lines 1, 4, 11, and 22 were included only to improve code organization and readability ("--" is used for comments). Finally, VHDL is not case sensitive, but to ease visualization capital letters were employed for reserved VHDL words.

```
Library declarations
1  ----------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------------------
Entity
5  ENTITY buffered_mux IS
6    PORT (a, b, c, d: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
7          sel: IN INTEGER RANGE 0 TO 3;
8          ena: IN STD_LOGIC;
9          y: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
10 END buffered_mux;
11 ----------------------------------------------------
Architecture
12 ARCHITECTURE myarch OF buffered_mux IS
13   SIGNAL x: STD_LOGIC_VECTOR(7 DOWNTO 0);
14 BEGIN
15   x <= a WHEN sel=0 ELSE --Mux
16        b WHEN sel=1 ELSE
17        c WHEN sel=2 ELSE
18        d;
19   y <= x WHEN ena='1' ELSE --Tristate buffer
20        (OTHERS => 'Z');
21 END myarch;
22 ----------------------------------------------------
```

FIGURE 19.2. VHDL code for the circuit of Figure 19.1 (Example 19.1).

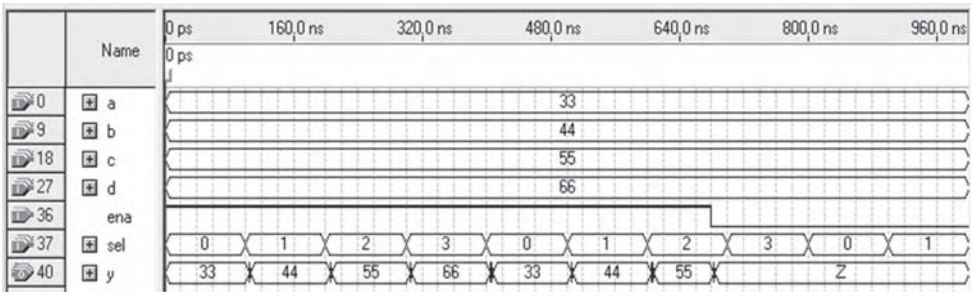


FIGURE 19.3. Simulation results from the circuit (buffered mux) inferred with the code of Figure 19.2.

Simulation results obtained with the code above are shown in Figure 19.3. Note that the input signals are preceded by an arrow with an "I" (Input) marked inside, while the arrow for the output signal has an "O" (Output) marked inside. The inputs can be chosen feely, while the output is calculated and plotted by the simulator. As can be seen, the circuit does behave as expected. ■

19.3 Fundamental VHDL Packages

The list below shows the main VHDL packages along with their libraries of origin. These packages can be found in the libraries that accompany your synthesis/simulation software.

Library *std*

- Package *standard*: Defines the basic VHDL types (`BOOLEAN`, `BIT`, `BIT_VECTOR`, `INTEGER`, `NATURAL`, `POSITIVE`, `REAL`, `TIME`, `DELAY_LENGTH`, etc.) and related *logical*, *arithmetic*, *comparison*, and *shift* operators.

Library *ieee*

- Package *std_logic_1164*: Defines the nine-valued type `STD_ULOGIC` and its *resolved* subtype `STD_LOGIC` (industry standard). Only *logical* operators are included, along with some type-conversion functions.
- Package *numeric_std*: Defines the numeric types `SIGNED` and `UNSIGNED`, having `STD_LOGIC` as the base type. Includes also *logical*, *arithmetic*, *comparison*, and *shift* operators, plus some type-conversion functions.

Nonstandard packages

Both libraries above (*std*, *ieee*) are standardized by the IEEE. The next packages are common sharewares provided by EDA vendors.

- Package *std_logic_arith*: Defines the numeric types `SIGNED` and `UNSIGNED`, having `STD_LOGIC` as the base type. Includes some *arithmetic*, *comparison*, and *shift* operators, plus some type-conversion functions. This package is only *partially* equivalent to *numeric_std*.
- Package *std_logic_signed*: Defines *arithmetic*, *comparison*, and some *shift* operators for the `STD_LOGIC_VECTOR` type as if it were `SIGNED`.
- Package *std_logic_unsigned*: Defines *arithmetic*, *comparison*, and some *shift* operators for the `STD_LOGIC_VECTOR` type as if it were `UNSIGNED`.

19.4 Predefined Data Types

The predefined data types are from the libraries/packages listed above. Those that are synthesizable are listed in Figure 19.4, which shows their names, library/package of origin, and synthesizable values.

Predefined data types	Library / Package of origin	Synthesizable values without restrictions
BIT, BIT_VECTOR	std / standard	'0', '1'
BOOLEAN	std / standard	TRUE, FALSE
INTEGER	std / standard	$-(2^{31}-1)$ to $+(2^{31}-1)$
NATURAL	std / standard	0 to $+(2^{31}-1)$
POSITIVE	std / standard	1 to $+(2^{31}-1)$
CHARACTER	std / standard	256-symbol alphabet (1 byte/symbol)
STRING	std / standard	Set of characters
REAL	std / standard	Little or no synthesis support
STD_(U)LOGIC, STD_(U)LOGIC_VECTOR	ieee / std_logic_1164	Input: '0' or 'L', '1' or 'H' Output: '0' or 'L', '1' or 'H', '-' or 'X', and 'Z'
UNSIGNED, SIGNED	ieee / numeric_std, ( ) / std_logic_arith	Same as STD_LOGIC

FIGURE 19.4. Predefined synthesizable data types with respective library/package of origin and synthesizable values.

EXAMPLE 19.2 DATA-TYPE USAGE

Consider the following signal definitions:

```
SIGNAL a: BIT;
SIGNAL b: BIT_VECTOR(7 DOWNTO 0);
SIGNAL c: BIT_VECTOR(1 TO 16);
SIGNAL d: STD_LOGIC;
SIGNAL e: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL f: STD_LOGIC_VECTOR(1 TO 16);
SIGNAL g: INTEGER RANGE -35 TO 35;
SIGNAL h: INTEGER RANGE 0 TO 255;
SIGNAL i: NATURAL RANGE 0 TO 255;
```

a. How many bits will the compiler assign to each of these signals?

b. Explain why the assignments below are legal.

```
a<=b(5);
c(16)<=b(0);
c(1)<=a;
b(5 DOWNTO 1)<=c(8 TO 12);
e(1)<=d;
e(2)<=f(16);
f(1 TO 8)<=e(7 DOWNTO 0);
b<="11110000";
a<='1';
d<='Z';
e<=(0=>'1', 7=>'0', OTHERS=>'Z');
e<="0ZZZZZZ1"; --same as above
```

c. Explain why the assignments below are illegal.

```
a<='Z';
a<=d;
c(16)<=e(0);
e(5 DOWNTO 1)<=c(8 TO 12);
f(5 TO 9)<=e(7 DOWNTO 2);
```

SOLUTION

Part (a):
1 bit for *a* and *d*, 7 bits for *g*, 8 bits for *b*, *e*, *h*, and *i*, and 16 bits for *c* and *f*.

Part (b):
All data types and ranges match.

Part (c):
There are type and range mismatches.

```
a<='Z'; --'Z' not available for BIT
a<=d; --type mismatch (BIT x STD_LOGIC)
c(16)<=e(0); --type mismatch
e(5 DOWNTO 1)<=c(8 TO 12); --type mismatch
f(5 TO 9)<=e(7 DOWNTO 2); --range mismatch
```

19.5 User Defined Data Types

Data types can be created using the keyword TYPE. Such declarations can be done in the declarative part of ENTITY, ARCHITECTURE, FUNCTION, PROCEDURE, or PACKAGE. Roughly speaking, they can be divided into the three categories below.

Integer-based data types Subsets of INTEGER, declared using the syntax below.

Syntax: TYPE type_name IS RANGE type_range;

Examples:

```
TYPE bus_size IS RANGE 8 TO 64;
TYPE temperature IS RANGE -5 TO 120;
```

Enumerated data types Employed in the design of finite state machines.

Syntax: TYPE type_name IS (state_names);

Examples:

```
TYPE machine_state IS (idle, forward, backward);
TYPE counter IS (zero, one, two, three);
```

Array-based data types The keywords TYPE and ARRAY are now needed, as shown in the syntax below. They allow the creation of multi-dimensional data sets (1D, 1D×1D, 2D, and generally also 3D are synthesizable without restrictions).

Syntax: TYPE type_name IS ARRAY (array_range) OF data_type;

Examples of 1D arrays (single row):

```
TYPE vector IS ARRAY (7 DOWNTO 0) OF STD_LOGIC;          --1×8 array
TYPE BIT_VECTOR IS ARRAY (NATURAL RANGE <>) OF BIT;      --unconstrained array
```

Examples of 1D×1D arrays (4 rows with 8 elements each):

```
TYPE array1D1D IS ARRAY (1 TO 4) OF BIT_VECTOR(7 DOWNTO 0);     --4×8 array
TYPE vector_array IS ARRAY (1 TO 4) OF vector;                  --4×8 array
```

Example of 2D array (8×16 matrix):

```
TYPE array2D IS ARRAY (1 TO 8, 1 TO 16) OF STD_LOGIC;           --8×16 array
```

19.6 Operators

VHDL provides a series of operators that are divided into the six categories below. The last four are summarized in Figure 19.5, along with the allowed data types.

Assignment operators "<=", ":=", "=>"

"<=" Used to assign values to signals.
":=" Used to assign values to variables, constants, or initial values.
"=>" Used with the keyword OTHERS to assign values to arrays.

Examples:

```
sig1<='1';                      --assignment to a single-bit signal
sig2<="00001111";               --assignment to a multibit signal
sig3<=(OTHERS=>'0');            --result is sig3<="00...0"
VARIABLE var1: INTEGER:=0;      --assignment of initial value
var2:="0101";                   --assignment to a multibit variable
```

Concatenation operators "&" and ","

These operators are employed to group values.

Example: The assignments to *x*, *y*, and *z* below are equivalent.

```
k: CONSTANT BIT_VECTOR(1 TO 4):="1100";
x<=('Z', k(2 TO 3), "11111");       -- result: x<="Z1011111"
y<='Z' & k(2 TO 3) & "11111";       -- result: y<="Z1011111"
z<=(7=>'Z', 5=>'0', OTHERS=>'1'); -- result: z<="Z1011111"
```

Logical operators NOT, AND, NAND, OR, NOR, XOR, XNOR

The only logical operator with precedence over the others is NOT.

Examples:

```
x<=a NAND b;            -- result: x=(a.b)'
y<=NOT(a AND b);        -- result: same as above
z<=NOT a AND b;         -- result: x=a'.b
```

Arithmetic operators +, -, *, /, **, ABS, REM, MOD

These are the classical operators: *addition*, *subtraction*, *multiplication*, *division*, *exponentiation*, *absolute-value*, *remainder*, and *modulo*. They are also summarized in Figure 19.5 along with the allowed data types.

Examples:

```
x<=(a+b)**N;
y<=ABS(a)+ABS(b);
z<=a/(a+b);
```

Shift operators SLL, SRL, SLA, SRA, ROL, ROR

Shift operators are shown in Figure 19.5 along with the allowed data types. Their meanings are described below.

SLL (shift left logical): Data are shifted to the left with '0's in the empty positions.

SRL (shift right logical): Data are shifted to the right with '0's in the empty positions.

SLA (shift left arithmetic): Data are shifted to the left with the rightmost bit in the empty positions.

SRA (shift right arithmetic): Data are shifted to the right with the leftmost bit in the empty positions.

ROL (rotate left): Circular shift to the left.

ROR (rotate right): Circular shift to the right.

Examples:

```
a<="11001";
x<=a SLL 2;  --result: x<="00100"
```

Operator type	Predefined operators	Supported predefined data types (1)
Logical	NOT, AND, NAND, OR, NOR, XOR, XNOR	BOOLEAN, BIT, BIT_VECTOR, STD_(U)LOGIC, STD_LOGIC_(U)VECTOR, (UN)SIGNED(2)
Arithmetic	+, -, *, /, **, ABS, REM, MOD	INTEGER, NATURAL, POSITIVE, SIGNED, UNSIGNED, REAL(3)
Shift	SLL, SRL, SLA, SRA, ROL, ROR	BIT_VECTOR, (UN)SIGNED(4)
Comparison	=, /=, >, <, >=, <=	BOOLEAN, BIT, BIT_VECTOR, INTEGER, NATURAL, POSITIVE, (UN)SIGNED, CHARACTER, STRING, REAL(3)
(1) As defined in the *original* package. (2) Depends on the package. (3) Limited or no synthesis support. (4) Partial set.		

FIGURE 19.5. Predefined synthesizable operators and allowed predefined data types.

```
y<=a SLA 2;  --result: y<="00111"
z<=a SLL -3; --result: z<="00011"
```

Comparison operators =, /=, >, <, >=, <=

Comparison operators are also shown in Figure 19.5 along with the allowed data types.

Example:

```
IF a>=b THEN x<='1';
```

19.7 Attributes

The main purposes of VHDL attributes are to allow the construction of generic (flexible) codes as well as event-driven codes. They also serve for communicating with the synthesis tool to modify synthesis directives.

The predefined VHDL attributes can be divided into the following three categories: (i) *range related*, (ii) *position related*, and (iii) *event related*. All three are summarized in Figure 19.6, which also lists the allowed data types.

Range-related attributes Return parameters regarding the range of a data array.

Example:

For the signal *x* specified below, the range-related attributes return the values listed subsequently (note that $m > n$).

```
SIGNAL x: BIT_VECTOR(m DOWNTO n);
x'LOW → n
x'HIGH → m
x'LEFT → m
x'RIGHT → n
x'LENGTH → m-n+1
x'RANGE → m DOWNTO n
```

Predefined attributes		Supported predefined data types
Range related	'LOW, 'HIGH, 'LEFT, 'RIGHT, 'LENGTH, 'RANGE, 'ASCENDING, 'REVERSE_RANGE	BIT_VECTOR, STD_LOGIC_(U)VECTOR, INTEGER, NATURAL, POSITIVE, (UN)SIGNED
Position related	'POS, 'VAL, 'LEFTOF, 'RIGHTOF, 'PRED, 'SUCC	Enumerated data types
Event related	'EVENT, 'STABLE, 'LAST_VALUE	BIT, STD_(U)LOGIC, BOOLEAN

FIGURE 19.6. Predefined synthesizable attributes and allowed predefined data types.

```
x'REVERSE_RANGE → n TO m
x'ASCENDING → FALSE (because the range of x is descending)
```

Position-related attributes Return positional information regarding *enumerated* data types. For example, `x'POS(value)` returns the position of the specified value, while `x'VAL(position)` returns the value in the specified position. These attributes are also included in Figure 19.6.

Event-related attributes Employed for monitoring signal changes (like clock transitions). The most common of these is `x'EVENT`, which returns `TRUE` when an event (positive or negative edge) occurs in *x*. The main (synthesizable) attributes in this category are also included in Figure 19.6.

Other attributes

`GENERIC:` This attribute was described in Section 19.2. It allows the specification of arbitrary constants in the code's entity.

`ENUM_ENCODING:` This is a very important attribute for state-encoding in finite-state-machine-based designs. Its description will be seen in Section 19.16.

19.8 Concurrent versus Sequential Code

Contrary to computer programs, which are *sequential* (serial), VHDL code is inherently *concurrent* (parallel). Therefore, all statements have the same precedence.

Though this is fine for the design of combinational circuits, it is not for sequential ones. To circumvent this limitation, `PROCESS`, `FUNCTION`, or `PROCEDURE` can be used, which are the only pieces of VHDL code that are interpreted sequentially.

In the IEEE 1076 standard, `FUNCTION` and `PROCEDURE` are collectively called *subprograms*. To ease any reference to sequential code, we will use the word *subprogram* in a wider sense, including `PROCESS` in it as well.

Regarding sequential code, it is important to remember, however, that each subprogram, as a whole, is still interpreted concurrently to any other statements or subprograms that the code might contain.

The VHDL statements intended for concurrent code (therefore located *outside* subprograms) are `WHEN` and `GENERATE` (plus a less common statement called `BLOCK`), while those for sequential code (thus allowed only *inside* subprograms) are `IF`, `CASE`, `LOOP`, and `WAIT`. *Operators* (seen in Section 19.6) are allowed anywhere.

19.9 Concurrent Code (WHEN, GENERATE)

As mentioned above, concurrent code can be constructed with the statements WHEN and GENERATE, plus *operators*.

The WHEN statement This statement is available in two forms as shown in the syntaxes below. Two equivalent examples (a multiplexer) are also depicted. The keyword OTHERS is useful to specify all remaining input values, while the keyword UNAFFECTED (not employed in the examples below) can be used when no action is to take place. (You can now go back and inspect Example 19.1.)

Syntax (WHEN-ELSE)

```
assignment WHEN conditions ELSE
assignment WHEN conditions ELSE
...;
```

Example

```
x <= a WHEN sel=0 ELSE
     b WHEN sel=1 ELSE
     c;
```

Syntax (WITH-SELECT-WHEN)

```
WITH identifier SELECT
   assignment WHEN conditions ELSE
   assignment WHEN conditions ELSE
   ...;
```

Example

```
WITH sel SELECT
   x <= a WHEN 0,
        b WHEN 1,
        c WHEN OTHERS;
```

The GENERATE statement This statement is equivalent to the LOOP statement. However, the latter is for sequential code, while the former is for concurrent code.

Syntax

```
label: FOR identifier IN range GENERATE
    [declarations
BEGIN]
    (concurrent assignments)
END GENERATE [label];
```

Example

```
G1: FOR i IN 0 TO M GENERATE
    b(i) <= a(M-i);
END GENERATE G1;
```

EXAMPLE 19.3 PARITY DETECTOR

Parity detectors were studied in Section 11.7 (see Figure 11.20). Design a circuit of this type with a *generic* number of inputs.

SOLUTION

A VHDL code for this problem is presented below. Note that this code has only two sections because additional libraries/packages are not needed (the data types employed in the design are all from the package *standard*, which is made visible by default).

The entity, called *parity_detector*, is in lines 2–6. As requested, *N* is entered using the GENERIC attribute (line 3), so this code can implement any size parity detector (the only change needed is in the value of *N* in line 3). The input is called *x* (mode IN, type BIT_VECTOR), while the output is called *y* (mode OUT, type BIT). The architecture is in lines 8–16. Note that GENERATE was employed (lines

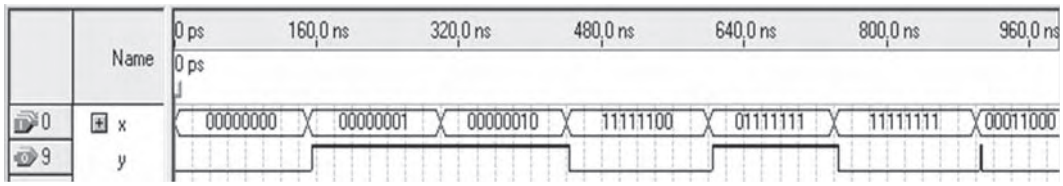


FIGURE 19.7. Simulation results from the code (parity detector) of Example 19.3.

12–14) along with the logical operator XOR (line 13). An internal signal, called *internal*, was created in line 9 to hold the value of the XOR operations (line 13). Observe that this signal has multiple bits because multiple assignments to the same bit are not allowed (if it were a single-bit signal, then N assignments to it would occur, that is, one in line 11 and $N-1$ in line 13). Simulation results are depicted in Figure 19.7.

```
1  ------------------------------------------------
2  ENTITY parity_detector IS
3      GENERIC (N: INTEGER:=8); --number of bits
4      PORT (x: IN BIT_VECTOR(N-1 DOWNTO 0);
5            y: OUT BIT);
6  END parity_detector;
7  ------------------------------------------------
8  ARCHITECTURE structural OF parity_detector IS
9      SIGNAL internal: BIT_VECTOR(N-1 DOWNTO 0);
10 BEGIN
11     internal(0)<=x(0);
12     gen: FOR i IN 1 TO N-1 GENERATE
13         internal(i)<=internal(i-1) XOR x(i);
14     END GENERATE;
15     y<=internal(N-1);
16 END structural;
17 ------------------------------------------------ ■
```

19.10 Sequential Code (IF, CASE, LOOP, WAIT)

As mentioned in Section 19.8, VHDL code is inherently *concurrent*. To make it *sequential*, it has to be written inside a *subprogram* (that is, PROCESS, FUNCTION, or PROCEDURE, in our broader definition). The first is intended for use in the main code (so it will be seen in this section), while the other two are intended mainly for libraries (code-sharing and reusability) and will be seen in Sections 19.14 and 19.15.

PROCESS

Allows the construction of *sequential* code in the *main* code (recall that sequential code can implement sequential and combinational circuits). Because its code is sequential, only IF, CASE, LOOP, and WAIT are allowed (plus *operators*, of course). As shown in the syntax below, a process is always accompanied by a sensitivity list (except when WAIT is employed); the process runs every time a signal included in the

sensitivity list changes (or the condition related to WAIT is fulfilled). In the declarative part (between the words PROCESS and BEGIN), variables can be specified. The label is optional.

Syntax

```
[label:] PROCESS (sensitivity list)
    [declarative part]
BEGIN
    (sequential code)
END PROCESS [label];
```

Example

```
PROCESS (clk)
BEGIN
    IF clk'EVENT AND clk='1' THEN
        q <= d;
    END IF;
END PROCESS;
```

The IF statement This is the most commonly used of all VHDL statements. Its syntax is shown below.

Syntax

```
IF conditions THEN
    assignments;
ELSIF conditions THEN
    assignments; ...
ELSE
    assignments;
END IF;
```

Example

```
IF (x=a AND y=b) THEN
    output <= '0';
ELSIF (x=a AND y=c) THEN
    output <= '1';
ELSE
    output <= 'Z';
END IF;
```

The WAIT statement This statement is somewhat similar to IF, with more than one form available. Contrary to when IF, CASE, or LOOP are used, the process cannot have a sensitivity list when WAIT is employed. The WAIT UNTIL statement accepts only one signal, while WAIT ON accepts several. WAIT FOR is for simulations only. All three syntaxes are shown below.

Syntax (WAIT UNTIL)

```
WAIT UNTIL signal_condition;
```

Example

```
WAIT UNTIL clk'EVENT AND clk='1';
```

Syntax (WAIT ON)

```
WAIT ON signal1 [, signal2, ...];
```

Example

```
WAIT ON clk, rst;
```

Syntax (WAIT FOR)

```
WAIT FOR time;
```

Example

```
WAIT FOR 5 ns;
```

The CASE statement Even though CASE can only be used in *sequential* code, its fundamental role is to allow the easy creation of *combinational* circuits (more specifically, of truth tables), so it is the sequential counterpart of the concurrent statement WHEN. When CASE is employed, all input values must be tested, so the keyword OTHERS can be helpful, as shown in the example below (multiplexer).

Syntax

```
CASE identifier IS
   WHEN value => assignments;
   WHEN value => assignments;
   ...
END CASE;
```

Example

```
CASE sel IS
    WHEN 0 => y<=a;
    WHEN 1 => y<=b;
    WHEN OTHERS => y<=c;
END CASE;
```

The LOOP statement Allows the creation of multiple instances of the same assignments. It is the sequential counterpart of the concurrent statement GENERATE. However, as shown below, there are four options involving LOOP.

Syntax (FOR-LOOP)

```
[label:] FOR identifier IN range LOOP
    (sequential statements)
END LOOP [label];
```

Example

```
FOR i IN x'RANGE LOOP
    x(i) <= a(M-i) AND b(i);
END LOOP;
```

Syntax (WHILE-LOOP)

```
[label:] WHILE condition LOOP
    (sequential statements)
END LOOP [label];
```

Example

```
WHILE i<M LOOP
    ...
END LOOP;
```

Syntax (LOOP with EXIT)

```
... LOOP
    ...
    [label:] EXIT [loop_label]
    [WHEN condition];
    ...
END LOOP;
```

Example

```
temp := 0;
FOR i IN N-1 DOWNTO 0 LOOP
    EXIT WHEN x(i)='1';
    temp := temp +1;
END LOOP;
```

Syntax (LOOP with NEXT)

```
... LOOP

    [label:] NEXT [loop_label]
    [WHEN condition];
    ...
END LOOP;
```

Example

```
temp := 0;
FOR i IN N-1 DOWNTO 0 LOOP
    NEXT WHEN x(i)='1';
    temp := temp +1;
END LOOP;
```

Note in the example with EXIT above that the code counts the number of *leading* zeros in an *N*-bit vector (starting with the MSB), while in the example with NEXT it counts the *total* number of zeros in the vector.

EXAMPLE 19.4 LEADING ZEROS

As mentioned above, the code presented in the example with EXIT counts the total number of leading zeros in an *N*-bit vector, starting from the left (MSB). Write the complete code for that problem.

SOLUTION

The corresponding code is shown below. Again additional library/package declarations are not needed. The entity is in lines 2–6, and the name chosen for it is *leading_zeros*. Note that the number of bits was again declared using the GENERIC statement (line 3), thus causing this solution to be fine for any vector size. The architecture is in lines 8–20 under the name *behavioral*. Note that a process was needed to use a variable (*temp*), which, contrary to a signal, does accept multiple assignments (lines 13 and 16). The EXIT statement (line 15) was employed to quit the loop when a '1' is found. The value of *temp* (a variable) is eventually passed to *y* (a signal) at the end of the process (line 18). Simulation results are depicted in Figure 19.8.

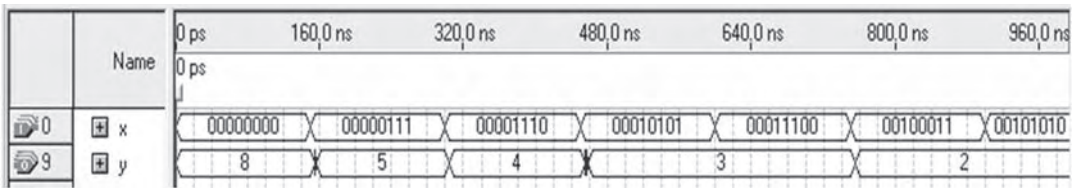


FIGURE 19.8. Simulation results from the code (leading zeros) of Example 19.4.

```
1  -----------------------------------------
2  ENTITY leading_zeros IS
3       GENERIC (N: INTEGER:=8);
4       PORT (x: IN BIT_VECTOR(N-1 DOWNTO 0);
5             y: OUT INTEGER RANGE 0 TO N);
6  END leading_zeros;
7  -----------------------------------------
8  ARCHITECTURE behavioral OF leading_zeros IS
9  BEGIN
10      PROCESS (x)
11          VARIABLE temp: INTEGER RANGE 0 TO N;
12      BEGIN
13          temp:=0;
14          FOR i IN x'RANGE LOOP
15               EXIT WHEN x(i)='1';
16               temp:=temp+1;
17          END LOOP;
18          y<=temp;
19      END PROCESS;
20 END behavioral;
21 ----------------------------------------- ■
```

19.11 Objects (CONSTANT, SIGNAL, VARIABLE)

There are three kinds of objects in VHDL: `CONSTANT`, `SIGNAL`, and `VARIABLE`.

`CONSTANT` A constant can be declared and used basically anywhere (entity, architecture, package, component, block, configuration, and subprograms). Its syntax is shown below. (Constants can also be declared using the `GENERIC` statement seen in Section 19.2.2.)

Syntax: `CONSTANT constant_name: constant_type := constant_value;`

Examples:

```
CONSTANT number_of_bits: INTEGER:=16;
CONSTANT mask: BIT_VECTOR(31 DOWNTO 0):=(OTHERS=>'1');
```

SIGNAL Signals define circuit I/Os and internal wires. They can be declared in the same places as constants, with the exception of subprograms (though they can be *used* there). When used in a subprogram, its value is only updated at the conclusion of the subprogram run. Moreover, it does not accept multiple assignments. Its syntax is shown below. The initial value is ignored during synthesis.

Syntax: `SIGNAL signal_name: signal_type [range] [:= initial_value];`

Examples:

```
SIGNAL seconds: INTEGER RANGE 0 TO 59;
SIGNAL enable: BIT;
SIGNAL my_data: STD_LOGIC_VECTOR(1 TO 8):="00001111";
```

VARIABLE Variables can only be declared and used in subprograms (PROCESS, FUNCTION, or PROCEDURE in our broader definition), so it represents only *local* information (except in the case of *shared* variables). On the other hand, its update is immediate, and multiple assignments are allowed. Its syntax is shown below. The initial value is again only for simulations.

Syntax: `VARIABLE variable_name: variable_type [range] [:= initial_value];`

Examples:

```
VARIABLE seconds: INTEGER RANGE 0 TO 59;
VARIABLE enable: BIT;
VARIABLE my_data: STD_LOGIC_VECTOR(1 TO 8):="00001111";
```

SIGNAL ***versus*** *VARIABLE* The distinction between signals and variables and their correct usage are fundamental to the writing of efficient (and correct) VHDL code. Their differences are summarized in Figure 19.9 by means of six fundamental rules [Pedroni04a] and are illustrated in the example that follows.

Rule	SIGNAL	VARIABLE
1. Local of declaration	In any VHDL unit, except subprograms	Only in subprograms (PROCESS, FUNCTION, or PROCEDURE)
2. Scope	Can be global (available to the whole code)	Always local (visible only inside the subprogram), except for shared variables
3. Update	New value available only at the end of the subprogram run	Updated immediately (new value can be used in the next line of code)
4. Assignment operator	Values are assigned using "<=" (example: sig<=5;)	Values are assigned using ":=" (example: var:=5;)
5. Multiple assignments	Only one assignment is allowed	Accepts multiple assignments (because update is immediate)
6. Inference of registers	Flip-flops are inferred when an assignment to a signal occurs *at the transition* of another signal.	Flip-flops are inferred when an assignment to a variable occurs at the transition of another signal *and* this value is eventually passed to a signal

FIGURE 19.9. Comparison between SIGNAL and VARIABLE.

EXAMPLE 19.5 COUNTER (FLIP-FLOP INFERENCE)

This example illustrates the use of the rules presented in Figure 19.9, particularly rule 6, which deals with the inference of flip-flops. Design a 0-to-9 counter, then simulate it and also check the number of flip-flops inferred by the compiler. Recall from Section 14.2 that four DFFs are expected in this case.

SOLUTION

The corresponding VHDL code is shown below. Note the use of rule 6 in lines 12, 13, and 18. Because a value is assigned to a variable (*temp*, lines 13) at the transition of another signal (*clk*, line 12) and that variable's value is eventually passed to a signal (*count*, line 18), flip-flops are expected to be inferred. Looking at the compilation reports one will find that four registers were inferred. Note also the use of other rules from Figure 19.9, like rule 3; the test in line 14 is fine only because the update of a variable is immediate, so the value assigned in line 13 is ready for testing in the next line of code. Simulation results are displayed in Figure 19.10.

```
1  -----------------------------------------
2  ENTITY counter IS
3       PORT (clk: IN BIT;
4             count: OUT INTEGER RANGE 0 TO 9);
5  END counter;
6  -----------------------------------------
7  ARCHITECTURE counter OF counter IS
8  BEGIN
9       PROCESS (clk)
10          VARIABLE temp: INTEGER RANGE 0 TO 10;
11      BEGIN
12          IF (clk'EVENT AND clk='1') THEN
13              temp:=temp+1;
14              IF (temp=10) THEN
15                  temp:=0;
16              END IF;
17          END IF;
18          count<=temp;
19      END PROCESS;
20 END counter;
21 -----------------------------------------
```

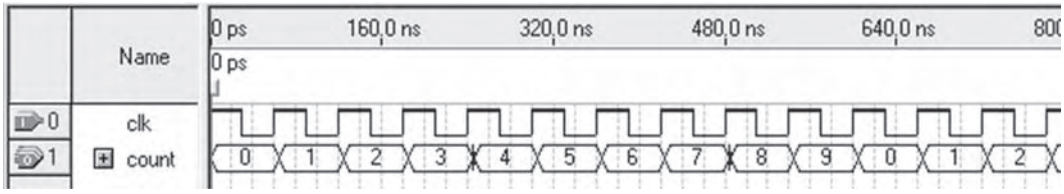


FIGURE 19.10. Simulation results from the code (0-to-9 counter) of Example 19.5.

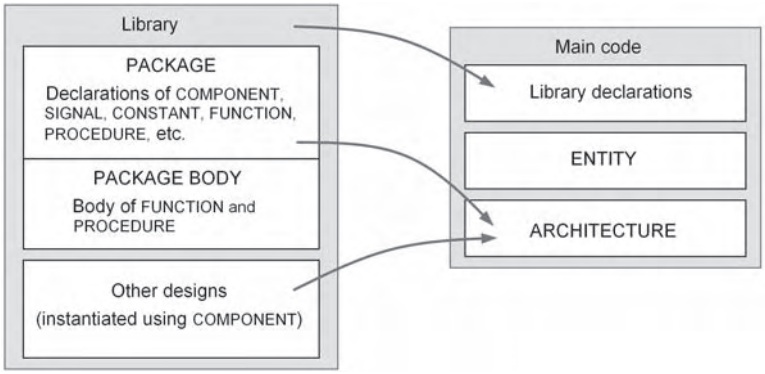


FIGURE 19.11. Relationship between the main code and the units intended mainly for system-level design (located in libraries).

19.12 Packages

We have concluded the description of the VHDL units that are intended for the main code, and we turn now to those that are intended mainly for libraries (*system-level* design); these are `PACKAGE`, `COMPONENT`, `FUNCTION`, and `PROCEDURE`. The relationship between them and the main code is illustrated in Figure 19.11 [Pedroni04a].

A package can be used for two purposes: (i) to make declarations and (ii) to describe global functions and procedures. To construct it, two sections of code might be needed, called `PACKAGE` and `PACKAGE BODY` (see syntax below). The former contains only declarations, while the latter is needed when a function or procedure is declared in the former, in which case it must contain the full description (body) of the declared subprogram(s). The two parts must have the same name.

```
PACKAGE package_name IS
  (declarations)
END package_name;
--------------------------------------
[PACKAGE BODY package_name IS
  (FUNCTION and PROCEDURE descriptions)
END package_name;]
```

EXAMPLE 19.6 PACKAGE WITH A FUNCTION

The `PACKAGE` below (called *my_package*) contains three declarations (one constant, one type, and one function). Because a function declaration is present, a `PACKAGE BODY` is needed, in which the whole function is described (details on how to write functions will be seen in Section 19.14).

```
1  ------------------------------------------------------------------
2  PACKAGE my_package IS
3      CONSTANT carry: BIT:='1';
4      TYPE machine_state IS (idle, forward, backward);
5      FUNCTION convert_integer (SIGNAL S: BIT_VECTOR) RETURN INTEGER;
6  END my_package;
```

```
7  ------------------------------------------------------------------
8  PACKAGE BODY my_package IS
9      FUNCTION convert_integer (SIGNAL S: BIT_VECTOR) RETURN INTEGER IS
10     BEGIN
11         ...(function body)...
12     END convert_integer;
13 END my_package;
14 ------------------------------------------------------------------ ■
```

19.13 Components

COMPONENT is simply a piece of *conventional* code (that is, library declarations, entity, and architecture). However, the declaration of a code as a component allows reusability and also the construction of *hierarchical* designs. Commonly used digital subsystems (adders, multipliers, multiplexers, etc.) are often compiled using this technique.

A component can be instantiated in an ARCHITECTURE, PACKAGE, GENERATE, or BLOCK. Its syntax contains two parts, one for the *declaration* and another for the *instantiation*, as shown below.

```
COMPONENT component_name IS
   PORT (port_name: signal_mode signal_type;
         port_name: signal_mode signal_type;
         ...);
END COMPONENT;

label: [COMPONENT] component_name PORT MAP (port list);
```

As can be seen above, a COMPONENT declaration is similar to an ENTITY declaration. The second part (component instantiation) requires a label, followed by the (optional) word COMPONENT, then the component's name, and finally a PORT MAP declaration, which is simply a list relating the ports of the actual circuit to the ports of the predesigned component that is being instantiated. This mapping can be *positional* or *nominal*, as illustrated below.

```
-------- Component declaration: ------------------------------------
COMPONENT nand_gate IS
    PORT (a, b: IN BIT; c: OUT BIT);
END COMPONENT;

-------- Component instantiations: ---------------------------------
nand1: nand_gate PORT MAP (x, y, z);              --positional mapping
nand2: nand_gate PORT MAP (a=>x, b=>y, c=>z);   --nominal mapping
```

The two traditional ways of using a component (which, as seen above, is a conventional piece of VHDL code that has been compiled into a certain library) are:

i. With the component declared in a package (also located in a library) and instantiated in the main code;

ii. With the component declared and instantiated in the main code.

An example using method (ii) is shown below.

EXAMPLE 19.7 CARRY-RIPPLE ADDER WITH COMPONENT

Adders were studied in Sections 12.2–12.4. The carry-ripple adder of Figure 12.3(b) was repeated in Figure 19.12 with a generic number of stages (*N*). Design this adder using `COMPONENT` to instantiate the *N* full-adder units.

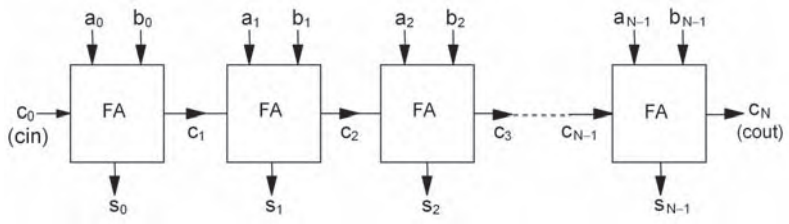


FIGURE 19.12. Carry-ripple adder of Example 19.7.

SOLUTION

A VHDL code for this problem is shown below. Note that the full-adder unit, which will be instantiated using the keyword `COMPONENT` in the main code, was designed separately (it is a conventional piece of VHDL code). In this example, the component was declared in the main code (in the architecture's declarative part, lines 13–15; note that it is simply a copy of the component's entity). The instantiation occurs in line 20. Because *N* is generic, the `GENERATE` statement was employed to create multiple instances of the component. The label chosen for the component is FA, and the mapping is *positional*. Simulation results are shown in Figure 19.13.

```
1  -------- The component: ----------------------------
2  ENTITY full_adder IS
3      PORT (a, b, cin: IN BIT;
4            s, cout: OUT BIT);
5  END full_adder;
6  -------------------------------------------------
7  ARCHITECTURE full_adder OF full_adder IS
8  BEGIN
9      s<=a XOR b XOR cin;
10     cout<=(a AND b) OR (a AND cin) OR (b AND cin);
11 END full_adder;
12 -------------------------------------------------
```

```
1  -------- Main code: --------------------------------
2  ENTITY carry_ripple_adder IS
3      GENERIC (N : INTEGER:=8); --number of bits
4      PORT (a, b: IN BIT_VECTOR(N-1 DOWNTO 0);
5            cin: IN BIT;
6            s: OUT BIT_VECTOR(N-1 DOWNTO 0);
7            cout: OUT BIT);
8  END carry_ripple_adder;
```

```
 9  -------------------------------------------------
10 ARCHITECTURE structural OF carry_ripple_adder IS
11      SIGNAL carry: BIT_VECTOR(N DOWNTO 0);
12      -----------------------------------------------
13      COMPONENT full_adder IS
14          PORT (a, b, cin: IN BIT; s, cout: OUT BIT);
15      END COMPONENT;
16      -----------------------------------------------
17  BEGIN
18      carry(0)<=cin;
19      gen_adder: FOR i IN a'RANGE GENERATE
20          FA: full_adder PORT MAP (a(i), b(i), carry(i), s(i), carry(i+1));
21      END GENERATE;
22      cout<=carry(N);
23  END structural;
24  ---------------------------------------------------------------------------
```

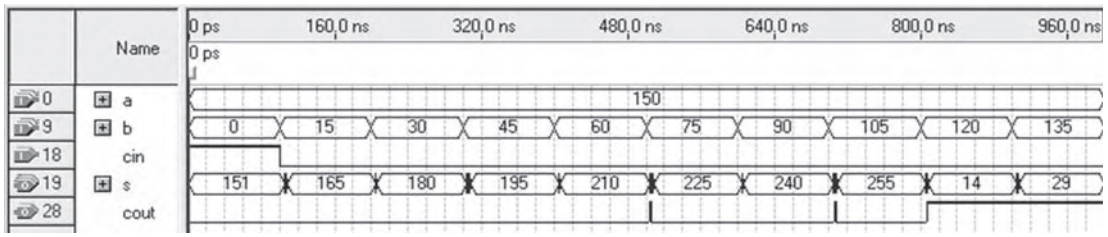


FIGURE 19.13. Simulation results from the code (carry-ripple adder) of Example 19.7.

GENERIC MAP Completely generic code (for libraries) can be attained using the GENERIC attribute (Section 19.2). When a component containing such an attribute is instantiated, the value originally given to the generic parameter can be overwritten by including a GENERIC MAP declaration in the component instantiation. The new syntax for the component instantiation is shown below.

```
label: comp_name [COMPONENT] GENERIC MAP (parameter list) PORT MAP (port list);
```

Example:

```
----- Component declaration: -----------------------
COMPONENT xor_gate IS
    GENERIC (N: INTEGER:=8);
    PORT (inp: IN BIT_VECTOR(1 TO N); outp: OUT BIT);
END COMPONENT;
-------------------------------------------------------

----- Component instantiation: ---------------------------------------------
gate1: xor_gate GENERIC MAP (N=>16) PORT MAP (inp=>x, outp=>y); --Nominal map.
gate2: xor_gate GENERIC MAP (16) PORT MAP (x, y); --Positional mapping
----------------------------------------------------------------------------
```

19.14 Functions

PROCESS, FUNCTION, and PROCEDURE are the three types of VHDL subprograms (in our broader definition). The first is intended mainly for the main code, while the other two are generally located in packages/libraries (for reusability and code sharing).

The code inside a function is *sequential*, so only the sequential VHDL statements (IF, CASE, LOOP, and WAIT) can be used. However, WAIT is generally not supported in functions. Other prohibitions are signal declarations and component instantiations. The syntax of FUNCTION is shown below.

```
FUNCTION function_name [parameters] RETURN data_type IS
   [declarative part]
BEGIN
   (sequential code)
END function_name;
```

Zero or more parameters can be passed to a function. However, it must always return a single value. The parameters, when passed, can be only CONSTANT (default) or SIGNAL (VARIABLE is not allowed), declared in the following way:

```
[CONSTANT] constant_name: constant_type;
SIGNAL signal_name: signal_type;
```

A function can be *called* basically anywhere (in combinational as well as sequential code, inside subprograms, etc.). Its *construction* (using the syntax above), on the other hand, can be done in the following places: (i) in a package, (ii) in the declarative part of an entity, (iii) in the declarative part of an architecture, (iv) in the declarative part of a subprogram. Because of reusability and code sharing, option (i) is by far the most popular (illustrated in the example below).

EXAMPLE 19.8 FUNCTION *SHIFT_INTEGER*

Write a function capable of logically shifting an INTEGER to the left or to the right. Two signals should be passed to the function, called *input* and *shift*, where the former is the signal to be shifted, while the latter is the desired amount of shift. If $shift > 0$, then the vector should be shifted *shift* positions to the left; otherwise, if $shift < 0$, then the vector should be shifted $|shift|$ positions to the right. Test your function with it located in a PACKAGE (plus PACKAGE BODY, of course).

SOLUTION

A VHDL code for this problem is shown below. The function (called *shift_integer*) is declared in line 3 of a PACKAGE (called *my_package*) and constructed in lines 7–10 of the respective PACKAGE BODY. The inputs to the function are signals *a* and *b*, both of type INTEGER, which also returns an INTEGER. Note that only one line of actual code (line 9) is needed to create the desired shifts.

The main code is also shown below. Note that the package described above must be included in the library declarations portion of the main code (see line 2). As can be seen in the entity, 6-bit signals were employed to illustrate the function operation. A call is made in line 12 with *input* and *shift* passed to the function, which returns a value for *output* (observe the operation of this code in Figure 19.14).

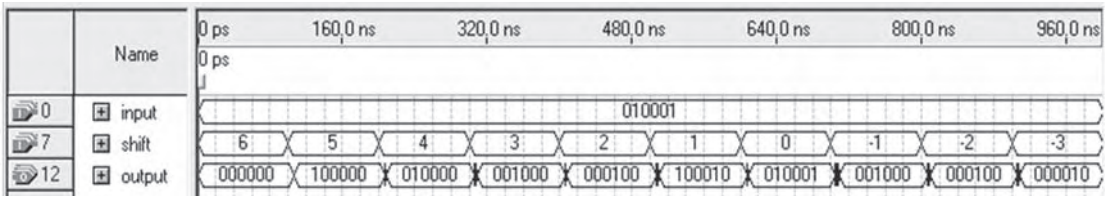


FIGURE 19.14. Simulation results from the code (function *shift_integer*) of Example 19.8.

```
1  -------- Package: ----------------------------------------------------
2  PACKAGE my_package IS
3      FUNCTION shift_integer (SIGNAL a, b: INTEGER) RETURN INTEGER;
4  END my_package;
5  ------------------------------------------------------------------------
6  PACKAGE BODY my_package IS
7      FUNCTION shift_integer (SIGNAL a, b: INTEGER) RETURN INTEGER IS
8      BEGIN
9          RETURN a*(2**b);
10     END shift_integer;
11 END my_package;
12 ------------------------------------------------------------------------
```

```
1  ----------------------- Main code: ---------------------------------
2  USE work.my_package.all;
3  ------------------------------------------------------------------------
4  ENTITY shifter IS
5      PORT (input: IN INTEGER RANGE 0 TO 63;
6            shift: IN INTEGER RANGE -6 TO 6;
7            output: OUT INTEGER RANGE 0 TO 63);
8  END shifter;
9  ------------------------------------------------------------------------
10 ARCHITECTURE shifter OF shifter IS
11 BEGIN
12     output<=shift_integer(input, shift);
13 END shifter;
14 ------------------------------------------------------------------------ ■
```

19.15 Procedures

The purpose, construction, and usage of `PROCEDURE` are similar to those of `FUNCTION`. Its syntax is shown below.

```
PROCEDURE procedure_name [parameters] IS
   [declarative part]
BEGIN
   (sequential code)
END procedure_name;
```

The parameters in the syntax above can contain `CONSTANT`, `SIGNAL`, or `VARIABLE`, accompanied by their respective mode, which can be only `IN`, `OUT`, or `INOUT`. Their full specification is as follows.

```
CONSTANT constant_name: constant_mode constant_type;
SIGNAL signal_name: signal_mode signal_type;
VARIABLE variable_name: variable_mode variable_type;
```

The fundamental differences between FUNCTION and PROCEDURE are the following: (i) a procedure can return more than one value, whereas a function must return exactly one; (ii) variables can be passed to procedures, which are forbidden for functions; and (iii) while a function is called as part of an expression, a procedure call is a statement on its own.

EXAMPLE 19.9 PROCEDURE *SORT_DATA*

Write a procedure (called *sort_data*) that sorts two signed decimal values. Test it with the procedure located in a PACKAGE.

SOLUTION

A VHDL code for this problem is shown below. As requested, the procedure (*sort_data*) is located in a package (called *my_package*). This procedure (which returns two values) is declared in lines 3–4 of a PACKAGE and is constructed in lines 8–18 of the corresponding PACKAGE BODY. In the main code, note the inclusion of *my_package* in line 2. Observe also in line 11 that the procedure call is a statement on its own. Simulation results are depicted in Figure 19.15.

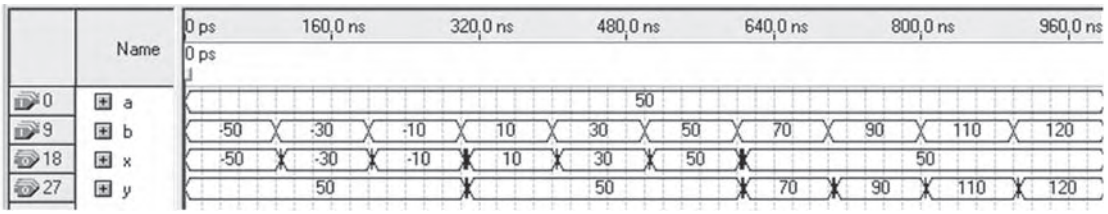


FIGURE 19.15. Simulation results from the code (procedure *sort_data*) of Example 19.9.

```
1  -------- Package: ------------------------------------
2  PACKAGE my_package IS
3       PROCEDURE sort_data (SIGNAL in1, in2: IN INTEGER;
4           SIGNAL out1, out2: OUT INTEGER);
5  END my_package;
6  ----------------------------------------------------
7  PACKAGE BODY my_package IS
8       PROCEDURE sort_data (SIGNAL in1, in2: IN INTEGER;
9           SIGNAL out1, out2: OUT INTEGER) IS
10      BEGIN
11        IF (in1<in2) THEN
12            out1<=in1;
13            out2<=in2;
14        ELSE
15            out1<=in2;
16            out2<=in1;
17        END IF;
```

```
18     END sort_data;
19 END my_package;
20 ------------------------------------------------

1  ----------- Main code: -------------------------
2  USE work.my_package.all;
3  ------------------------------------------------
4  ENTITY sorter IS
5      PORT (a, b: IN INTEGER RANGE -128 TO 127;
6            x, y: OUT INTEGER RANGE -128 TO 127);
7  END sorter;
8  ------------------------------------------------
9  ARCHITECTURE sorter OF sorter IS
10 BEGIN
11     sort_data (a, b, x, y);
12 END sorter;
13 ------------------------------------------------ ■
```

19.16 VHDL Template for FSMs

As seen in Chapter 15, there are two fundamental aspects that characterize an FSM, one related to its specifications and the other related to its physical structure. The specifications are normally translated by means of a *state transition diagram*, like that in Figure 19.16(a), which says that the machine has three states (*A*, *B*, and *C*), one output (*y*), and one input (*x*). Regarding the hardware, it can be modeled as in Figure 19.16(b), which shows the system split into two sections, one *sequential* (contains the flip-flops) and one *combinational* (contains the combinational circuits). The signal presently stored in the DFFs is called *pr_state*, while that to be stored at the next (positive) clock transition is called *nx_state*.

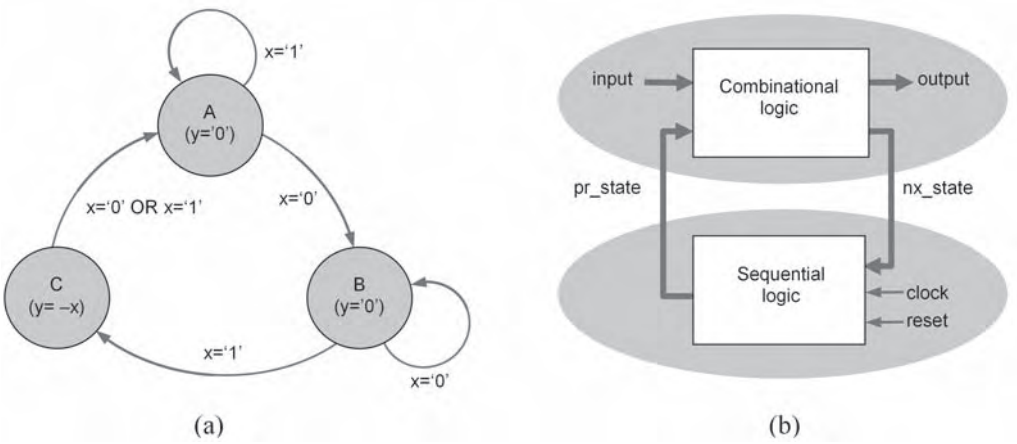


FIGURE 19.16. (a) Example of *state transition diagram*; (b) Simplified FSM model (for the hardware).

A VHDL template, resembling the diagram of Figure 19.16(b), is shown below [Pedroni04a]. First, observe the block formed by lines 12–15. An enumerated data type is created in line 12, which contains all machine states; next, the (optional) ENUM_ENCODING attribute is declared in lines 13–14, which allows the user to choose the encoding scheme for the machine states (explained shortly); finally, a signal whose type is that defined in line 12 is declared in line 15. Observe now that the code proper has three parts. The first part (lines 18–25) creates the lower section of the FSM, which contains the flip-flops, so the clock is connected to it and a process is needed. The second part (lines 27–44) creates the upper section of the FSM; because it is combinational, the clock is not employed and the code can be concurrent or sequential (because sequential code allows the construction of both types of circuits); the latter was employed in the template, and CASE was used. Finally, the third part (lines 46–51) is optional; it can be used to store ("clean") the output when it is subject to glitches but glitches are not acceptable in the design (like in signal generators).

```
1  ------------------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------------------
5  ENTITY <entity_name> IS
6      PORT (input: IN <data_type>;
7            clock, reset: IN STD_LOGIC;
8            output: OUT <data_type>);
9  END <entity_name>;
10 ------------------------------------------------------------
11 ARCHITECTURE <arch_name> OF <entity_name> IS
12     TYPE state IS (A, B, C, ...);
13     [ATTRIBUTE ENUM_ENCODING: STRING;
14     ATTRIBUTE ENUM_ENCODING OF state: TYPE IS "sequential";]
15     SIGNAL pr_state, nx_state: state;
16 BEGIN
17     ----------- Lower section: -----------
18     PROCESS (reset, clock)
19     BEGIN
20         IF (reset='1') THEN
21             pr_state<=A;
22         ELSIF (clock'EVENT AND clock='1') THEN
23             pr_state<=nx_state;
24         END IF;
25     END PROCESS;
26     ----------- Upper section: -----------
27     PROCESS (input, pr_state)
28     BEGIN
29         CASE pr_state IS
30             WHEN A =>
31                 IF (input=<value>) THEN
32                 output<=<value>;
33                 nx_state<=B;
34                 ELSE ...
35                 END IF;
36             WHEN B =>
37                 IF (input=<value>) THEN
```

```
38                      output<=<value>;
39                      nx_state<=C;
40                  ELSE ...
41                  END IF;
42              WHEN ...
43          END CASE;
44      END PROCESS;
45      ------ Output section (optional): -------
46      PROCESS (clock)
47      BEGIN
48          IF (clock'EVENT AND clock='1') THEN
49              new_output<=old_output;
50          END IF;
51      END PROCESS;
52  END <arch_name>;
53  ------------------------------------------------------------
```

`ENUM_ENCODING` This attribute allows the user to choose the encoding scheme for any enumerated data type. Its set of options includes the following (see Section 15.9):

- Sequential binary encoding (regular binary code, Section 2.1)
- Gray encoding (see Section 2.3)
- One-hot encoding (codewords with only one '1')
- Default encoding (defined by the compiler, generally sequential binary or one-hot or a combination of these)
- User-defined encoding (any other encoding)

Example:

```
TYPE state IS (A, B, C);
ATTRIBUTE ENUM_ENCODING: STRING;
ATTRIBUTE ENUM_ENCODING OF state: TYPE IS "11 00 10";
```

The following encoding results in this example: *A*="11", *B*="00", *C*="10".

Example:

```
TYPE state IS (red, green, blue, white, black);
ATTRIBUTE ENUM_ENCODING: STRING;
ATTRIBUTE ENUM_ENCODING OF state: TYPE IS “sequential”;
```

The following encoding results in this case: *red*="000", *green*="001", *blue*="010", *white*="011", *black*="100".

Note: When using Quartus II, the `ENUM_ENCODING` attribute causes the State Machine Viewer to be turned off. To keep it on and still choose the encoding style, instead of employing the attribute above, set up the compiler using Assignments > Settings > Analysis & Synthesis Settings > More Settings and choose “minimal bits” to have an encoding similar to “sequential” or “one-hot” for one-hot encoding (the latter is Quartus II default). This solution, however, is not portable and does not allow encodings like “gray,” for example.

EXAMPLE 19.10 BASIC STATE MACHINE

The code below implements the FSM of Figure 19.16(a). As can be seen, it is a straightforward application of the VHDL template described above. Note that the enumerated data type is called *state* (line 8) and that the optional attribute `ENUM_ENCODING` was employed (lines 9–10) to guarantee that the states are represented using sequential (regular) binary code.

```
1  ------------------------------------------------------------
2  ENTITY fsm IS
3      PORT (x, clk: IN BIT;
4            y: OUT BIT);
5  END fsm;
6  ------------------------------------------------------------
7  ARCHITECTURE fsm OF fsm IS
8      TYPE state IS (A, B, C);
9      ATTRIBUTE ENUM_ENCODING: STRING;
10     ATTRIBUTE ENUM_ENCODING OF state: TYPE IS "sequential";
11     SIGNAL pr_state, nx_state: state;
12 BEGIN
13     -------- Lower section: --------
14     PROCESS (clk)
15     BEGIN
16         IF (clk'EVENT AND clk='1') THEN
17             pr_state<=nx_state;
18         END IF;
19     END PROCESS;
20     -------- Upper section: --------
21     PROCESS (x, pr_state)
22     BEGIN
23         CASE pr_state IS
24             WHEN A =>
25                 y<='0';
26                 IF (x='0') THEN nx_state<=B;
27                 ELSE nx_state<=A;
28                 END IF;
29             WHEN B =>
30                 y<='0';
31                 IF (x='1') THEN nx_state<=C;
32                 ELSE nx_state<=B;
33                 END IF;
34             WHEN C =>
35                 y<=NOT x;
36                 nx_state<=A;
37         END CASE;
38     END PROCESS;
39 END fsm;
40 ------------------------------------------------------------ ■
```

19.17 Exercises

1. VHDL packages

Examine in the libraries that accompany your synthesis software the packages listed in Section 19.3. Write down at least the following:

a. Data types and operators defined in the package *standard*.

b. Data types and operators (if any) defined in the package *std_logic_1164*.

c. Data types and operators (if any) defined in the package *numeric_std*.

d. Data types and operators defined in the package *std_logic_arith*.

2. Buffered multiplexer

Redo the design in Example 19.1, but for a *generic* number of bits for *a*, *b*, *c*, *d*, and *y* (enter *N* using the `GENERIC` statement).

3. Data-type usage

Consider the following VHDL objects:

```
SIGNAL x1: BIT;
SIGNAL x2: BIT_VECTOR(7 DOWNTO 0);
SIGNAL x3: STD_LOGIC;
SIGNAL x4: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL x5: INTEGER RANGE -35 TO 35;
VARIABLE y1: BIT_VECTOR(7 DOWNTO 0);
VARIABLE y2: INTEGER RANGE -35 TO 35;
```

a. Why are the statements below legal?

```
x2(7)<=x1;
x3<=x4(0);
x2<=y1;
x5<=y2;
x3<='1';
x2<=(OTHERS=>'0');
y2:=35;
y1:="11110000";
```

b. And why are these illegal?

```
x1(0)<=x2(0);
x3<=x1;
x2<=(OTHERS=>'Z');
y2<=-35;
x3:='Z';
x2(7 DOWNTO 5)<=y1(3 DONWTO 0);
y1(7)<='1';
```

4. Logical operators

Suppose that *a* = "11110001", *b* = "11000000", and *c* = "00000011". Determine the values of *x*, *y*, and *z*.

a. `x<=NOT (a XOR b)`

b. `y<=a XNOR b`

c. `z<=(a AND NOT b) OR (NOT a AND c)`

5. Shift operators

Determine the result of each shift operation below.

a. `"11110001" SLL 3`

b. `"11110001" SLA 2`

c. `"10001000" SRA 2`

d. `"11100000" ROR 4`

6. Parity detector

Redo the design of Example 19.3, but use sequential instead of concurrent code (that is, a `PROCESS` with `LOOP`).

7. Hamming weight (HW)

The HW of a vector is the number of '1's in it. Design a circuit that computes the HW of a *generic-length* vector (use `GENERIC` to enter the number of bits, *N*).

8. Flip-flop inference

a. Briefly describe the main differences between `SIGNAL` and `VARIABLE`. When are flip-flops inferred?

b. Write a code from which flip-flops are guaranteed to be inferred, then check the results in the compilation report.

c. If the counter of Example 19.5 were a 0-to-999 counter, how many flip-flops would be required?

d. Modify the code of Example 19.5 for it to be a 0-to-999 counter, then compile and simulate it, finally checking whether the actual number of registers matches your prediction.

9. Shift register

As seen in Section 14.1, a shift register (SR) is simply a string of serially connected flip-flops commanded by a common clock signal and, optionally, also by a reset signal. Design the SR of Figure 14.2(a) using the `COMPONENT` construct to instantiate $N=4$ DFFs. Can you make your code *generic* (arbitrary *N*)?

10. Function *add_bitvector*

Arithmetic operators, as defined in the original packages, do not support the type `BIT_VECTOR` (see Figure 19.5). Write a function (called *add_bitvector*) capable of adding two `BIT_VECTOR` signals and returning a signal of the same type. Develop two solutions:

a. With the function located in a package.

b. With the function located in the main code.