MCA. Eng. K. DAWAH

VHDL

Advanced Digital Electronics

ألألكترونيات الرقميةالمتقدمة

(stage 4^{TH} BRANCH ELECTRONIC)

A U C Almaaref University Collage

كلية المعارف الجامعة-قسم هندسة تقنيات الحاسوب - المرحلةالرا بعة- فرع الالكترونيات
Department of Computer Engineering and Technology
BY:K.DAWAH .ABBAS

MCA. Eng. K. DAWAH

1-Introduction

1.1 About VHDL

VHDL is a hardware description language. It describes the behavior of an electronic circuit or system, from which the physical circuit or system can then be attained (implemented).

VHDL stands for VHSIC Hardware Description Language. VHSIC is itself an Abbreviation for Very High Speed Integrated Circuits, an initiative funded by the United States Department of Defense in the 1980s that led to the creation of VHDL.

Its first version was VHDL 87, later upgraded to the so-called VHDL 93. VHDL was the original and first hardware description language to be standardized by the Institute of Electrical and Electronics Engineers, through the IEEE1076 standard. An additional standard, the IEEE 1164, was later added to introduce a multi-valued logic system

VHDL is a standard, technology/vendor independent language, and is there fore Portable and reusable. The two main immediate applications of VHDL are in the field of Programmable Logic Devices (including CPLDs—Complex Programmable Logic Devices and FPGAs—Field Programmable Gate Arrays) and in the field of ASICs (Application Specific Integrated Circuits). Once the VHDL code has been written, it can be used either to implement the circuit in a programmable device(from Altera, Xilinx, Atmel, etc.) or can be submitted to a foundry for fabrication of an ASIC chip. Currently, many complex commercial chips (microcontrollers, for example) are designed using such an approach.

A final note regarding VHDL is that, contrary to regular computer programs which are sequential, its statements are inherently concurrent (parallel). For that reason, VHDL is usually referred to as a code rather than a program. In VHDL ,only statements placed inside a PROCESS, FUNCTION, or PROCEDURE are executed sequentially

1.2 Design Flow

As mentioned above, one of the major utilities of VHDL is that it allows the synthesis of a circuit or system in a programmable device (PLD or FPGA) or in an ASIC. The steps followed during such a project are summarized in figure1.1.

We start the design by writing the VHDL code, which is saved in a file with the extension

Eng. K. DAWAH

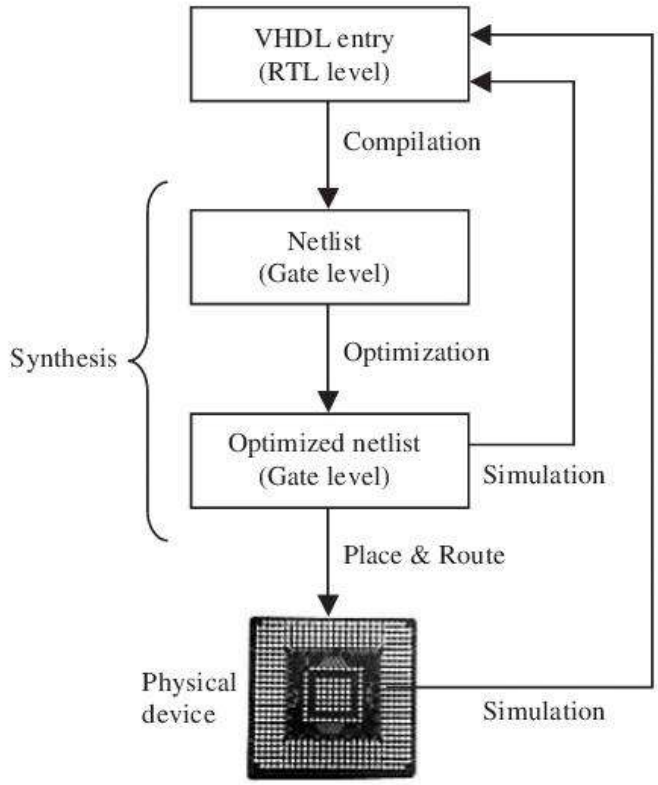


Figure 1.1
Summary of VHDL design flow.

VHDL and the same name as its ENTITY's name. The first step in the synthesis process is compilation. Compilation is the conversion of the high-level VHDL language, which describes the circuit at the Register Transfer Level (RTL), into a net list at the gate level. The second step is optimization, which is performed on the gate-level net list for speed or for area. At this stage, the design can be simulated. Finally, a place and-route (fitter) software will generate the physical layout for a PLD/FPGA chip or will generate the masks for an ASIC.

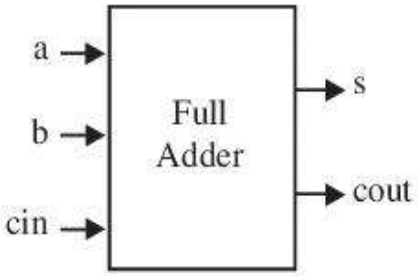
1.3 EDA Tools

There are several EDA (Electronic Design Automation) tools available for circuit synthesis, implementation, and simulation using VHDL. Some tools (place and route, for example) are oered as part of a vendor's design suite
(e.g., Altera's Quartus II, which allows the synthesis of VHDL code onto Altera's CPLD/FPGA chips)

1.4 Translation of VHDL Code into a Circuit

A full-adder unit is depicted in figure 1.2. In it, a and b represent the input bits to be added, cin is the carry-in bit, **s** is the **sum** bit, and **cout** the **carry-out** bit. As shown in the truth table, Smust be high whenever the number of inputs that are high is odd,
while **cout** must be **high** when **two or more inputs are high**. A VHDL code for the full adder of figure 1.2 is shown in figure 1.3. As can be seen, it consists of an ENTITY, which is a description of the pins (PORTS) of the

.



a b cin	s cout
0 0 0	0 0
0 1 0	1 0
1 0 0	1 0
1 1 0	0 1
0 0 1	1 0
0 1 1	0 1
1 0 1	0 1
1 1 1	1 1

Figure 1.2
Full-adder diagram and truth table.

```
ENTITY full_adder IS
PORT (a, b, cin: IN BIT;
      s, cout: OUT BIT);
END full_adder;
-------------------------------------
ARCHITECTURE dataflow OF full_adder IS
BEGIN
   s <= a XOR b XOR cin;
   cout <= (a AND b) OR (a AND cin) OR
           (b AND cin);
END dataflow;
```

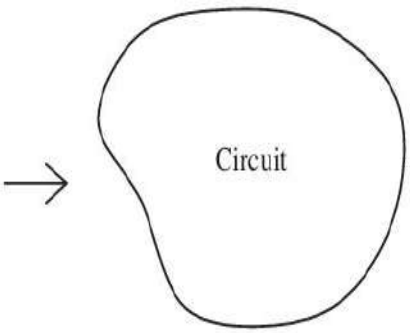


Figure 1.3
Example of VHDL code for the full-adder unit of figure 1.2.

Circuit, and of an ARCHITECTURE, which describes how the circuit should function. We see in the latter that the sum bit is computed as S=aXORbXORcin, while cout is obtained from COUT=a.b+a.cin+b.cin.

2-Code Structure

In this chapter , we describe the **fundamental** sections that comprise a piece of VHDL code: LIBRARY declarations, ENTITY, and ARCHITECTURE.

2.1 Fundamental VHDL Units

As depicted in figure 2.1, a standalone piece of VHDL code is composed of at least three fundamental sections:

1-**LIBRARY** declarations: Contains a list of all libraries to be used in the design. For example: ieee, std, work, etc.

2- ENTITY: Specifies the I/O pins of the circuit.

3-**ARCHITECTURE:** Contains the VHDL code proper, which describes how the circuit should behave (function).

A LIBRARY is a collection of commonly used pieces of code. Placing such pieces inside a library allows them to be reused or shared by other designs.

The typical structure of a library is illustrated in figure 2.2. The code is usually Written in the form of FUNCTIONS, PROCEDURES, or COMPONENTS, which are placed inside PACKAGES, and then compiled into the destination library.

2.2 Library Declarations

To declare a LIBRARY (that is, to make it visible to the design) two lines of codeare needed, one containing the name of the library, and the other a use

clause, as shown in the syntax below. LIBRARY

library_name;
USE library_name.package_name.package_parts;

At least **three** packages, from three different libraries, are usually needed in a design:

1-ieee.std_logic_1164*(from the ieeelibrary),*
2- standard*(from the stdlibrary),*
3-work*(work library).*

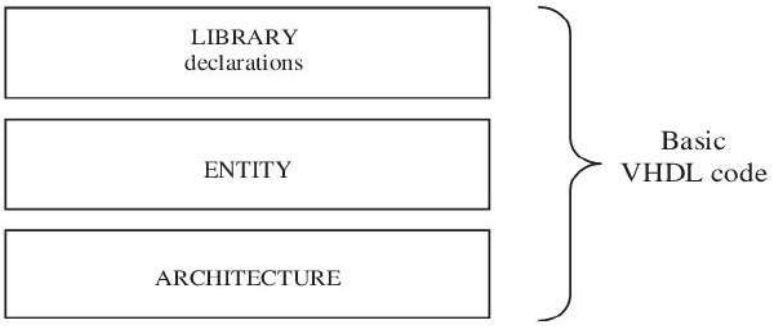


Figure 2.1
Fundamental sections of a basic VHDL code.

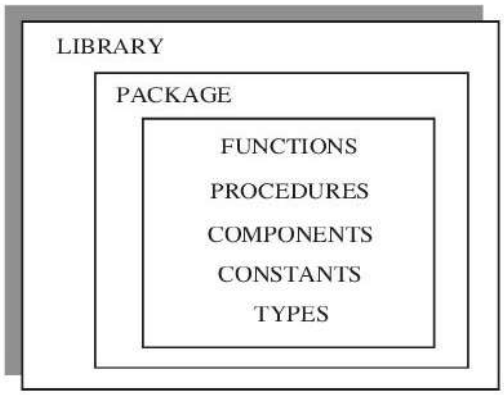


Figure 2.2
Fundamental parts of a LIBRARY.

Their **declarations** are as follows:

LIBRARY ieee; -- A semi-colon (;) indicates
USE ieee.std_logic_1164.all; -- the end of a statement or
LIBRARY std;-- declaration, while a double
USE std. standard. all; -- dash (--) indicates a comment.
LIBRARY work;
USE work.all;

The libraries **std** and **work** shown above are made visible by default, so there is no need to declare them; only the **ieee** library must be explicitly written. However, the latter is only necessary when the STD_LOGIC (or STD_ULOGIC) data type is employed in the design ***The purpose of the three packages/libraries*** mentioned above is the following: the std_logic_1164package of the ieee library specifies a multi-level logic system; std is are source library (data types, text i/o, etc.) for the VHDL design environment; and the work library is where we save our design (the .vhd file, plus all files created by the compiler, simulator, etc.).Indeed, the ieee library contains several packages, including the following:

1-std_logic_1164: Specifies the STD_LOGIC (8 levels) and STD_ULOGIC (9 levels) multi-valued logic systems.

2-std_logic_arith: Specifies the SIGNED and UNSIGNED data types and related arithmetic and comparison operations. It also contains several data conversion functions, which allow one type to be converted into another: conv_integer (p),conv_unsigned (p, b), conv_signed(p, b), conv_std_logic_vector(p, b).

3-std_logic_signed: Contains functions that allow operations with STD_LOGIC_VECTOR data to be performed as if the data were of type SIGNED.

4-std_logic_unsigned: Contains functions that allow operations with STD_LOGIC_VECTOR data to be performed as if the data were of type UNSIGNED.

2.3 ENTITY

An ENTITY is a list with specifications of all input and output pins (PORTS) of the circuit. Its **syntax** is shown below.

```
ENTITY entity_name IS
PORT (
Port_name : signal_modesignal_type;
Port_name : signal_modesignal_type;
...);
END entity_name
```

The mode of the signal can be ***IN***, ***OUT***, ***INOUT***, or ***BUFFER***. As illustrated in figure 2.3, IN and OUT are truly unidirectional pins, while INOUT is bidirectional. BUFFER, on the other hand, is employed when the output signal must be used(read)

internally. The type of the signal can be BIT, STD_LOGIC, INTEGER, etc. Finally, the name of the entity can be basically any name, except VHDL reserved words

Example: Let us consider the NAND gate of figure 2.4. Its ENTITY can be specified as:

```
ENTITY nand_gate IS
PORT (a, b : IN BIT;
x : OUT BIT);
END nand_gate;
```

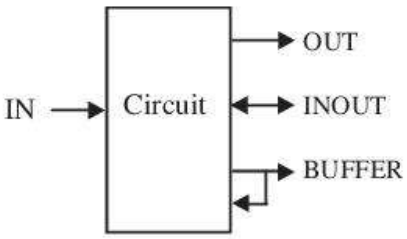


Figure 2.3
Signal *modes*.

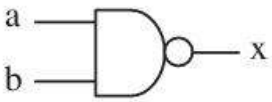


Figure 2.4
NAND gate.

The meaning of the ENTITY above is the following: the circuit has three I/O pins being two inputs (a and b, mode IN) and one output (x, mode OUT). All three signals are of type BIT. The name chosen for the entity was nand_gate.

2.4 ARCHITECTURE

The ARCHITECTURE is a description of how the circuit should behave (function).
Its syntax is the following:

```
ARCHITECTURE architecture_ name OF entity_name
IS[declarations]

BEGIN(code)

END architecture_name;
```

As shown above, an architecture has two parts: a declarative part (optional), where
signals and constants (among others) are declared, and the code part (from BEGIN down). Like in the case of an entity, the name of an architecture can be basically any name (except VHDL reserved words), including the same name as the entity's.

Example: Let us consider the NAND gate of figure 2.4 once again.

```
ARCHITECTURE myarch OF nand_gate IS
BEGIN
x <= a NAND b;
END myarch
```

The meaning of the ARCHITECTURE above is the following: the circuit must perform the NAND operation between the two input signals (a, b) and assign ("<=")
the result to the output pin (x). The name chosen for this architecture was myarch. In this example, there is no declarative part, and the code contains just a single assignment

.Example 2.1: DFF with Asynchronous Reset

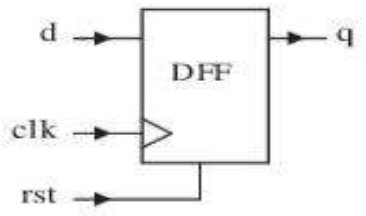


Figure 2.5

Figure 2.5 shows the diagram of a D-type flip-flop (DFF), triggered at the rising edge of the clock signal (clk), and with an asynchronous reset input (rst). When rst='1', the output must be turned low, regardless of clk. Otherwise, the output must copy the input (that is, q <=d) at the moment when clk changes from '0' to '1' (that is, when an upward event occurs on clk). There are several ways of implementing the DFF of figure 2.5, one being the solution presented below.

One thing to remember, however, is that

VHDL is inherently concurrent (contrary to regular computer programs, which are sequential), so to implement any clocked circuit (flip-flops, for example) we have to ''force'' VHDL to be sequential. This can be done using a PROCESS, as shown below

```
1 ---------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ---------------------------------------
5 ENTITY dff IS
6 PORT ( d, clk, rst: IN STD_LOGIC;
7 q: OUT STD_LOGIC);
8 END dff;
9 ---------------------------------------
10 ARCHITECTURE behavior OF dff IS
11 BEGIN
12 PROCESS (rst, clk)
13 BEGIN
14 IF (rst='1') THEN
15 q <= '0';
16 ELSIF (clk'EVENT AND clk='1') THEN
17 q <= d;
18 END IF;
19 END PROCESS;
20 END behavior;
21 ---------------------------------------
```

Comments:

Lines 2–3: Library declaration (library name and library use clause). Recall that the other two indispensable libraries (stdand work) are made visible by default

Lines 5–8: Entity dff.

Lines 10–20: Architecture behavior.

Line 6: Input ports (input mode can only be IN). In this example, all input signals are of type STD_LOGIC.
Line 7: Output port (output mode can be OUT, INOUT, or BUFFER). Here, the output is also of type STD_LOGIC.
Lines 11–19: Code part of the architecture (from word BEGIN on).
Lines 12–19: A PROCESS (inside it the code is executed sequentially).
Line 12: The PROCESS is executed every time a signal declared in its sensitivity list changes. In this example, every time rst or clk changes the PROCESS is run.
Lines 14–15: Every time rst goes to '1' the output is reset, regardless of clk (asynchronous reset).
Lines 16–17: If rst is not active, plus clk has changed (an EVENT occurred on clk),
plus such event was a rising edge (clk='1'), then the input signal (d) is stored in the
flip-flop (q <=d).
Lines 15 and 17: The "<=" operator is used to assign a value to a SIGNAL. In contrast, ":=" would be used for a VARIABLE. All ports in an entity are signals by default.
Lines 1, 4, 9, and 21: Commented out (recall that "- -" indicates a comment).Used only to better organize the design.
Note: VHDL is not case sensitive.

Simulation results:

Figure 2.6 presents simulation results regarding example 2.1. The graphs can be easily interpreted. The first column shows the signal names, as defined in the ENTITY. It also shows the mode (direction) of the signals; notice that the arrows associated

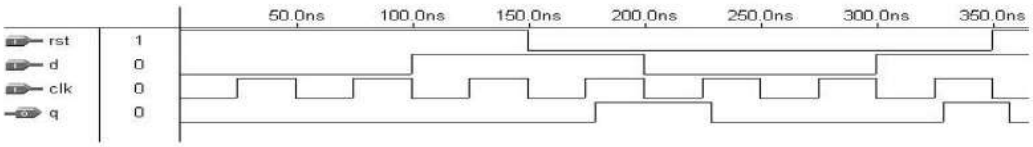


Figure 2.6
Simulation results of example 2.1.

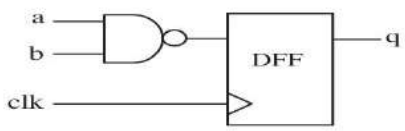


Figure 2.7
DFF plus NAND gate.

With rst, d, and clk are inward, and contain the letter I (input) inside, while that of qis outward and has an O (output) marked inside. The second column has the value of each signal in the position where the vertical cursor is placed. In the present case, the cursor is at 0ns, where the signals have value 1, 0, 0, 0, respectively. In this example, the values are simply '0' or '1', but when vectors are used, the values can be shown in binary, decimal, or hexadecimal form.
The third column shows the simulation proper. The input signals (rst, d, clk) can be chosen freely, and the simulator will determine the corresponding output (q). Comparing the results of figure 2.6 with those expected from the circuit shown previously, we notice that it works properly.
As mentioned earlier, the designs presented in the book were synthesized onto CPLD/FPGA devices

Example 2.2: DFF plus NAND Gate

The circuit of figure 2.4 was purely **combinational,**
While that of figure 2.5 was purely **sequential.**
The circuit of figure 2.7 is a mixture of both (without reset).

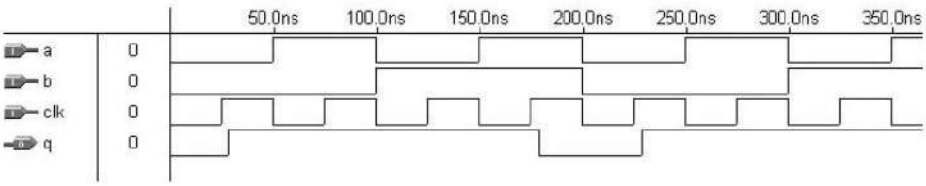


Figure 2.8
Simulation results of example 2.2.

solution that follows, we have purposely introduced an unnecessary signal (temp), just to illustrate how a signal should be declared.
Simulation results from the circuit synthesized with the code below are shown in figure 2.8.

```
1 ---------------------------------------
2 ENTITY example IS
3 PORT ( a, b, clk: IN BIT;
4 q: OUT BIT);
5 END examples;
6 ---------------------------------------
7 ARCHITECTURE example OF example IS
8 SIGNAL temp: BIT;
```

```
9 BEGIN
10 temp <= a NAND b;
11 PROCESS (clk)
12 BEGIN
13 IF (clk'EVENT AND clk='1') THEN q<=temp;
14 END IF;
15 END PROCESS;
16 END examples;
17 ------------------------------------
```

Comments:

Library declarations are not necessary in this case, because the data is of type BIT, which is specified in the library std (recall that the libraries std and work are made visible by default).

Lines 2–5 : Entity example.

Lines7–16: Architecture example.

Line 3: Input ports (all of type BIT).

Line 4: Output port (also of type BIT).

Line 8:Declarative part of the architecture (optional). The signal temp, of type BIT,

Was declared. Notice that there is no mode declaration (mode is only used in entities).

Lines9–15: Code part of the architecture (from word BEGIN on).

Lines 11–15: A PROCESS (sequential statements executed every time the signal clk

changes).

Lines 10 and 11–15: Though within a process the execution is sequential, the process,

as a whole, is concurrent with the other (external) statements; thus line 10 is executed

Concurrently with the block 11–15.

Line 10: Logical NAND operation. Result is assigned to signal temp.

Lines 13–14: IF statement. At the rising edge of clk the value of temp is assigned to q.

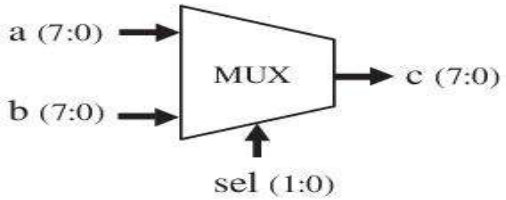
Lines 10 and 13: The "<=" operator is used to assign a value to a SIGNAL. In contrast, ":=" would be used for a VARIABLE.

Lines 8 and 10: Can be eliminated, changing "q <=a NAND b" in line 13.

Lines 1, 6, and 17: Commented out. Used only to better organize the design.

Problem 2.1: Multiplexer

The top-level diagram of a multiplexer is shown in figure P2.1. According to the truth table, the output should be equal to one of the inputs if sel= ‘‘01’’ (c = a) or sel= ‘‘10’’ (c = b), but it should be ‘0’ or Z (high impedance) if sel= ‘‘00’’ or sel= ‘‘11’’, respectively.



sel	c
00	0
01	a
10	b
11	Z

Figure P2.1

a) Complete the VHDL code below.
b) Write relevant comments regarding your solution (as in examples 2.1 and 2.2).
c) Compile and simulate your solution, checking whether it works as expected.
Note: A solution using IF was employed in the code below, because it is more intuitive.
However, as will be seen later, a multiplexer can also be implemented with other statements, like WHEN or CASE.

```
1 ----------------------------------------
2 LIBRARY ieee;
3USE ieee.STD_LOGIC_1164.all;
4 ----------------------------------------
5 ENTITY mux IS
6 PORT ( a , b : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7 sel : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
8c: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
9 END mux ;
10 ----------------------------------------
11 ARCHITECTURE example OF mux IS
12 BEGIN
13 PROCESS (a, b, sel)
14 BEGIN
15 IF (sel = "00") THEN
16 c <= "00000000";
17 ELSIF (sel="01") THEN
18 c <= a;
19ELSE  (sel = "10") THEN
20 c <= b;
21 ELSE
22 c <= (OTHERS => '0');
23 END IF ;
24 END PROCESS ;
25 END mux ;
26 ----------------------------------------
```

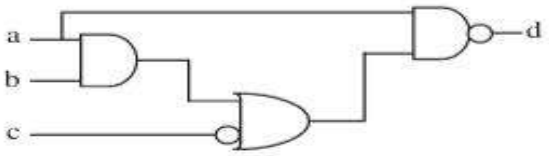


Figure P2.2

Problem 2.2: **Logic Gates**

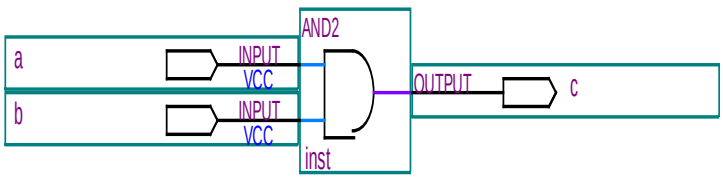
a) Write a VHDL code for the circuit of figure P2.2. Notice that it is purely combinational, so a PROCESS is not necessary. Write an expression for d using only logical operators (AND, OR, NAND, NOT, etc.).

b) Synthesize and simulate your circuit. After assuring that it works properly, open the report file and check the actual expression implemented by the compiler. Compare it with your expression.

```
1-.....;
2-LIBRARY ieee;
3-USE ieee.std_logic_1164.all;
4-.....
5-ENTITY P2-3 IS
6-PORT(
7-     a,b,c :IN STD_LOGIC;
8-     d :OUT STD_LOGIC);
9- END P2-3;
10-.....
11-ARCHITECTURE arc of P2-3 IS
12-BEGIN
13- d<=a NAND((a AND b)OR NOT c);
14-END arc;
15-.....
```

Example -A simple AND gate can be modeled, as follows:

```
ENTITY and2 IS
 PORT ( a, b : IN BIT
        c : OUT BIT );
END and2;
ARCHITECTURE and2_dawah OF and2 IS
BEGIN
c <= a AND b;
END and2_dawah;
```



3Data Types

In order to write VHDL code efficiently, it is essential to know what data types are allowed, and how to specify and use them. In this chapter, all fundamental data types are described, with special emphasis on those that are synthesizable. Discussions on data compatibility and data conversion are also included.

3.1 Pre-Defined Data Types

VHDL contains a series of pre-defined data types, specified through the IEEE 1076 and IEEE 1164 standards. More specifically, such data type definitions can be found in the following packages / libraries:

Package_ standard of library std: Defines BIT, BOOLEAN, INTEGER, and REAL data types.

_ Package std_logic_1164 of library ieee: Defines STD_LOGIC and STD_ULOGIC data types.

_ Package std_logic_arith of library ieee: Defines SIGNED and UNSIGNED data types, plus several data conversion functions, like conv_integer(p), conv_unsigned(p, b), conv_signed(p, b), and conv_std_logic_vector(p, b).

_ Packages std_logic_signed and std_logic_unsigned of library ieee: Contain functions that allow operations with STD_LOGIC_VECTOR data to be performed as if the data were of type SIGNED or UNSIGNED, respectively.

All pre-defined data types (specified in the packages/libraries listed above) are described below.

_ BIT (and BIT_VECTOR): 2-level logic (‘0’, ‘1’).

Examples:

SIGNAL x: BIT;

-- x is declared as a one-digit signal of type BIT.

SIGNAL y: BIT_VECTOR (3 DOWNTO 0);

-- y is a 4-bit vector, with the leftmost bit being the MSB.

SIGNAL w: BIT_VECTOR (0 TO 7);

-- w is an 8-bit vector, with the rightmost bit being the MSB.

Based on the signals above, the following assignments would be ***legal*** (to assign a value to a signal, the ‘‘<=’’ operator must be used):

x <= '1';

-- x is a single-bit signal (as specified above), whose value is

-- '1'. Notice that single quotes (' ') are used for a single bit.

y <= "0111";

-- y is a 4-bit signal (as specified above), whose value is "0111"

-- (MSB='0'). Notice that double quotes (" ") are used for

-- vectors.

w <= "01110001";

-- w is an 8-bit signal, whose value is "01110001" (MSB='1').

_ STD_LOGIC (and STD_LOGIC_VECTOR):8-valued logic system

introduced in the IEEE 1164 standard.

'X' Forcing Unknown (synthesizable unknown)
'0' Forcing Low (synthesizable logic '1')
'1' Forcing High (synthesizable logic '0')
'Z' High impedance (synthesizable tri-state buffer)
'W' Weak unknown
'L' Weak low
'H' Weak high
'–'Don't care

Examples:

SIGNAL x: STD_LOGIC;
-- x is declared as a one-digit (scalar) signal of type STD_LOGIC.
SIGNAL y: STD_LOGIC_VECTOR (3 DOWNTO 0) := "0001";
-- y is declared as a 4-bit vector, with the leftmost bit being
-- the MSB. The initial value (optional) of y is "0001". Notice
-- that the ":=" operator is used to establish the initial value.

Most of the std_logic levels are intended for simulation only. However, '0', '1', and 'Z' are synthesizable with no restrictions. With respect to the ''weak'' values, they are resolved in favor of the ''forcing'' values in multiply-driven nodes (see table 3.1).Indeed, if any two std_logic signals are connected to the same node, then conflicting logic levels are automatically resolved according to table 3.1.

Table 3.1
Resolved logic system (STD_LOGIC).

	X	0	1	Z	W	L	H	-
X	X	X	X	X	X	X	X	X
0	X	0	X	0	0	0	0	X
1	X	X	1	1	1	1	1	X
Z	X	0	1	Z	W	L	H	X
W	X	0	1	W	W	W	W	X
L	X	0	1	L	W	L	W	X
H	X	0	1	H	W	W	H	X
-	X	X	X	X	X	X	X	X

STD_LOGIC system described above is a subtype of STD_ULOGIC. The latter includes an extra logic value, ***'U'***, which stands for unresolved. Thus, contrary to STD_LOGIC, conflicting logic levels are not automatically resolved here, so output wires should never be connected together directly. However, if two output wires are never supposed to be connected together, this logic system can be used to detect design errors.

_ ***BOOLEAN:*** True, False.
_ ***INTEGER: 32-bit integers*** (from _2,147,483,647 to 2,147,483,647).
_ ***NATURAL:*** Non-negative integers (from 0 to 2,147,483,647).

_ ***REAL:*** Real numbers ranging from _1.0E38 to 1.0E38. Not synthesizable.
_ Physical literals: Used to inform physical quantities, like time, voltage, etc. Use fouling simulations. Not synthesizable.
_ ***Character literals***: Single ASCII character or a string of such characters. Not synthesizable
_ ***SIGNED and UNSIGNED***: data types defined in the std_logic_arith package of the ieee library. They have the appearance of STD_LOGIC_VECTOR, but accept arithmetic operations, which are typical of INTEGER data types

Examples:

x0<= '0';-- bit, std_logic, or std_ulogic value '0'
x1<= "00011111";-- bit_vector, std_logic_vector,
-- std_ulogic_vector, signed, or unsigned
x2<= "0001_1111";-- underscore allowed to ease visualization
x3<= "101111"-- binary representation of decimal 47
x4<= B"101111" -- binary representation of decimal 47
x5<= O"57"-- octal representation of decimal 47
x6<= X"2F"-- hexadecimal representation of decimal 47
n <= 1200;-- integer
m <= 1_200;-- integer, underscore allowed
IF ready THEN...-- Boolean, executed if ready=TRUE
y <= 1.2E-5;-- real, not synthesizable
q <= d after 10 ns;-- physical, not synthesizable

Example: ***Legal*** and ***illegal*** operations between data of different types.

```
SIGNAL a: BIT;
SIGNAL b: BIT_VECTOR(7 DOWNTO 0);
SIGNAL c: STD_LOGIC;
SIGNAL d: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL e: INTEGER RANGE 0 TO 255;
SOLUTION:

a <= b(5); -- legal (same scalar type: BIT)
b(0) <= a; -- legal (same scalar type: BIT)
c <= d(5); -- legal (same scalar type: STD_LOGIC)
d(0) <= c; -- legal (same scalar type: STD_LOGIC)
a <= c; -- illegal (type mismatch: BIT x STD_LOGIC)
b <= d;--illegal (type mismatch: BIT_VECTOR x-- STD_LOGIC_VECTOR)
e <= b; -- illegal (type mismatch: INTEGER x BIT_VECTOR)
e <= d; -- illegal (type mismatch: INTEGER x-- STD_LOGIC_VECTOR)
```

3.2 User-Defined Data Types

VHDL also allows the user to define own data types. Two categories of user defined data types are shown below: ***integer*** and ***enumerated.***

User-defined ***integer*** types:

1-TYPE integer IS RANGE -2147483647 TO +2147483647;
-- This is indeed the pre-defined type INTEGER.
2-TYPE natural IS RANGE 0 TO +2147483647;
-- This is indeed the pre-defined type NATURAL.
3-TYPE my_integer IS RANGE -32 TO 32;
-- A user-defined subset of integers.
4-TYPE student_grade IS RANGE 0 TO 100;
-- A user-defined subset of integers or naturals.

User-defined ***enumerated***types:

1-TYPE bit IS ('0', '1');
-- This is indeed the pre-defined type BIT
2-TYPE my_logic IS ('0', '1', 'Z');
-- A user-defined subset of std_logic.
3-TYPE bit_vector IS ARRAY (NATURAL RANGE <>) OF BIT;
-- This is indeed the pre-defined type BIT_VECTOR.
-- RANGE <> is used to indicate that the range is unconstrained.
-- NATURAL RANGE <>, on the other hand, indicates that the only
-- Restriction is that the range must fall within the NATURAL
-- range.
4-TYPE state IS (idle, forward, backward, stop);
-- An enumerated data type, typical of finite state machines.
5-TYPE color IS (red, green, blue, white);
-- Another enumerated data type.

The encoding of enumerated types is done sequentially and automatically (unless specified otherwise by a user-defined attribute, as will be shown in chapter 4). For example, for the type color above, two bits are necessary (there are four states), being ***"00"*** assigned to the first state (***red***), ***"01"*** to the second ***(green),"10"*** to the next ***(blue),*** and finally ***"11"*** to the last state ***(white)***

3.3 *Subtypes*

A SUBTYPE is a TYPE with a constraint. The main reason for using a sub type rather than specifying a new type is that, though operations between data of different types are not allowed, they are allowed between a subtype and its corresponding base type.

Examples: The subtypes below were derived from the types presented in the previous examples.

SUBTYPE natural IS INTEGER RANGE 0 TO INTEGER'HIGH;
-- As expected, NATURAL is a subtype (subset) of INTEGER.
SUBTYPE my_logic IS STD_LOGIC RANGE '0' TO 'Z';
-- Recall that STD_LOGIC=('X','0','1','Z','W','L','H','-').
-- Therefore, my_logic=('0','1','Z').

SUBTYPE my_color IS color RANGE red TO blue;
-- Since color=(red, green, blue, white), then
-- my_color=(red, green, blue).
SUBTYPE small_integer IS INTEGER RANGE -32 TO 32;
-- A subtype of INTEGER.

Example: Legal and illegal operations between types and subtypes.

SUBTYPE my_logic IS STD_LOGIC RANGE '0' TO '1';
SIGNAL a: BIT;
SIGNAL b: STD_LOGIC;
SIGNAL c: my_logic;
SOLUION:
b <= a; --***illegal*** (type mismatch: BIT versus STD_LOGIC)
b <= c;--***legal*** (same "base" type: STD_LOGIC)

3.4 –Arrays--

Arrays are ***collections*** of ***objects*** of the ***same type***. They can be one-dimensional **(1D)**, two-dimensional **(2D),** or one-dimensional-by-one-Dimensional**(1Dx1D).** They can also be of higher dimensions, but then they are generally not synthesizable Figure 3.1 illustrates the construction of data arrays. A single value (scalar) is shown in (a), a vector (1D array) in (b), an array of vectors (1Dx1D array) in (c), and an array of scalars (2D array) in (d).
Indeed, the pre-defined VHDL data types (seen in section 3.1) include only thescalar (single bit) and vector (one-dimensional array of bits) categories.
The predefined synthesizable types in each of these categories are the following:
1-Scalars: BIT, STD_LOGIC, STD_ULOGIC, and BOOLEAN.
2-Vectors: BIT_VECTOR, STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, INTEGER, SIGNED, and UNSIGNED.

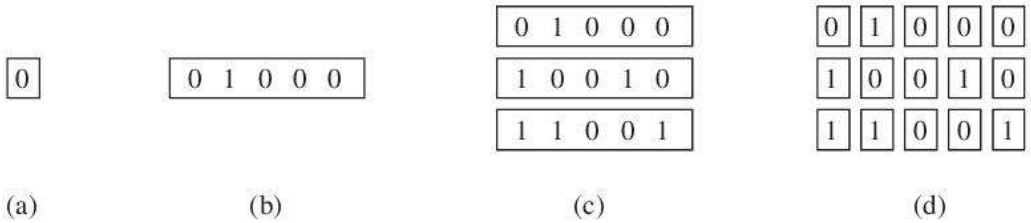


Figure 3.1
Illustration of (a) scalar, (b) 1D, (c) 1Dx1D, and (d) 2D data arrays.

As can be seen, there are no pre-defined 2D or 1Dx1D arrays, which, when necessary must be specified by the user. To do so, the new TYPE must first be defined ,then the new SIGNAL, VARIABLE, or CONSTANT can be declared using that data type. The syntax below should be used.

To specify a new array type:

```
TYPE type_name IS ARRAY (specification) OF data_type;
```

To make use of the new array type:

```
SIGNAL signal_name: type_name [:= initial_value];
```

In the syntax above, a SIGNAL was declared. However, it could also be a CONSTANT or a VARIABLE. Notice that the initial value is optional (for simulation only).

Example: 1Dx1D array.

Say that we want to build an array containing ***four vectors***, each of size **eight** bits .This is then an 1Dx1D array (see figure 3.1). Let us call each vector by row, and the Complete array by matrix. Additionally, say that we want the leftmost bit of each vector to be its MSB (most significant bit), and that we want the top row to be row 0.Then the array implementation would be the following (notice that a signal, called x, of type matrix, was declared as an example):

Example

```
TYPE row IS ARRAY (7 DOWNTO 0) OF STD_LOGIC;-- 1D array
TYPE matrix IS ARRAY (0 TO 3) OF row; -- 1Dx1D array
SIGNAL x: matrix; -- 1Dx1D signal
```

Example: 1Dx1D array.

Another way of constructing the 1Dx1D array above would be the following:

TYPE matrix IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(7 DOWNTO 0);

From a data-compatibility point of view, the latter might be advantageous over that in the previous example (see example 3.1).

Example: 2D array.

The array below is truly two-dimensional. Notice that its construction is not basedon vectors, but rather entirely on scalars.

TYPE matrix2D IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;

-- 2D array

Example: Array initialization.

As shown in the syntax above, the initial value of a SIGNAL or VARIABLE is optional .However, when initialization is required, it can be done as in the **examples** below.

```
.....:="0001";-- for 1D array
... :=('0','0','0','1')-- for 1D array
... :=(('0','1','1','1'), ('1','1','1','0')); -- for 1Dx1D or-- 2D array
```

Example: Legal and illegal array assignments.

The assignments in this example are based on the following type definitions andsignal declarations:

TYPE row IS ARRAY (7 DOWNTO 0) OF STD_LOGIC;-- 1D array
TYPE array1 IS ARRAY (0 TO 3) OF row;-- 1Dx1D array
TYPE array2 IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
-- 1Dx1D
TYPE array3 IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;- 2D array

SIGNAL x: row;
SIGNAL y: array1;
SIGNAL v: array2;
SIGNAL w: array3;
--------- Legal **scalar** assignments: ----------
-- The scalar (single bit) assignments below are all legal,
-- because the "base" (scalar) type is STD_LOGIC for all signals-- (x,y,v,w).
x(0) <= y(1)(2); -- notice two pairs of parenthesis-- (y is 1Dx1D)
x(1) <= v(2)(3); -- two pairs of parenthesis (v is 1Dx1D)
x(2) <= w(2,1); -- a single pair of parenthesis (w is 2D)
y(1)(1) <= x(6);
y(2)(0) <= v(0)(0);
y(0)(0) <= w(3,3);
w(1,1) <= x(7);
w(3,0) <= v(0)(3);
--------- **Vector** assignments: ----------
x <= y(0);-------- ***-- legal*** (same data types: ROW)
x <= v(1);.....***-- illegal*** (type mismatch: ROW x-- STD_LOGIC_VECTOR)
x <= w(2);.....***-- illegal*** (w must have 2D index)
x <= w(2, 2 DOWNTO 0);...***-- illegal*** (type mismatch: ROW x-- STD_LOGIC)
v(0) <= w(2, 2 DOWNTO 0);***-- illegal*** (mismatch: STD_LOGIC_VECTOR-x STD_LOGIC)
v(0) <= w(2);.....***-- illegal*** (w must have 2D index)
y(1) <= v(3);.....***-- illegal*** (type mismatch: ROW x-- STD_LOGIC_VECTOR)
y(1)(7 DOWNTO 3) <= x(4 DOWNTO 0); ***-- legal*** (same type,-- same size)
v(1)(7 DOWNTO 3) <= v(2)(4 DOWNTO 0);***-- legal*** (same type,-- same size)
w(1, 5 DOWNTO 1) <= v(2)(4 DOWNTO 0);***-- illegal*** (type mismatch)

3.5 Port Array

As we have seen, there are no pre-defined data types of more than one dimension .However, in the specification of the input or output pins (PORTS) of a circuit (which is made in the ENTITY), we might need to specify the ports as arrays of vectors .Since User-defined data types in a PACKAGE, which will then be visible to the whole design
(thus including the ENTITY). An example is shown below.
------- Package: ----------
LIBRARY ieee;
USE ieee.std_logic_1164.all;

PACKAGE my_data_types IS
TYPE vector_array IS ARRAY (NATURAL RANGE <>) OF
STD_LOGIC_VECTOR(7 DOWNTO 0);
END my_data_types;

------- Main code: ----------
LIBRARY ieee;
USE ieee.std_logic_1164.all;

USE work.my_data_types.all; -- user-defined package

ENTITY mux IS
PORT (inp: IN VECTOR_ARRAY (0 TO 3);
…);
END mux;

As can be seen in the example above, a user-defined data type, called vector array, was created, which can contain an indefinite number of vectors of size eight bits each (NATURAL RANGE <>signifies that the range is not fixed, with the only restriction that it must fall within the NATURAL range, which goes from 0 to 2,147,483,647). The data type was saved in a PACKAGE called my _data_ types, and later used in an ENTITY to specify a PORT called inp. Notice in the main code the inclusion of an additional USE clause to make the user-defined package my _data _types visible to the design Another option for the PACKAGE above would be that shown below, Where a CONSTANT declaration is included (a detailed study of PACKAGES will be presented in chapter 10).

------- Package: ----------
LIBRARY ieee;
USE ieee.std_logic_1164.all;

PACKAGE my_data_types IS
CONSTANT b: INTEGER:= 7;
TYPE vector_array IS ARRAY (NATURAL RANGE <>) OF
STD_LOGIC_VECTOR(b DOWNTO 0);
END my_data_types;

3.6 Records

Records are similar to arrays, with the only difference that they contain objects of different types

Example:
TYPE birthday IS RECORD
day: INTEGER RANGE 1 TO 31;
month: month_ name;
END RECORD;

3.7 Signed and Unsigned Data Types

As mentioned earlier, these types are defined in the std _logic _arith package of the ieee library. Their syntax is illustrated in the examples below.

Examples:

SIGNAL x: SIGNED (7 DOWNTO 0);
SIGNAL y: UNSIGNED (0 TO 3);
Notice that their syntax is similar to that of STD_LOGIC_VECTOR, not like thatof an INTEGER, as one might have expected.
An UNSIGNED value is a number never lower than zero. For example,

''0101''represents the decimal 5, while ''1101'' signifies 13. If type SIGNED is used instead, the value can be positive or negative (in two's complement format). Therefore,''0101'' would represent the decimal 5, while ''1101'' would mean(-3).
To use SIGNED or UNSIGNED data types, the std_logic_arith package, of The ieee library, must be declared. Despite their syntax, SIGNED and UNSIGNED data types are intended mainly for arithmetic operations, that is, contrary to STD_LOGIC_VECTOR, they accept arithmetic operations. On the other hand ,logical operations are not allowed. With respect to relational (comparison) operations ,there are no restrictions.

Example: **Legal** and **illegal** operations with signed/unsigned data types.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all; -- extra package necessary
...
SIGNAL a: IN SIGNED (7 DOWNTO 0);
SIGNAL b: IN SIGNED (7 DOWNTO 0);
SIGNAL x: OUT SIGNED (7 DOWNTO 0);
SOLUTION
v <= a + b; -- legal (arithmetic operation OK)
w <= a AND b; -- illegal (logical operation not OK)
```

Example: ***Legal*** and ***illegal*** operations with std_logic_vector.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all; -- no extra package required
...
SIGNAL a: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
SOLUTION:
v <= a + b; -- illegal (arithmetic operation not OK)
w <= a AND b; -- legal (logical operation OK)
```

Despite the constraint mentioned above, there is a simple way of allowing data of type STD_LOGIC_VECTOR to participate directly in arithmetic operations. For that, the ieee library provides two packages, std_logic_signed and std_logic_unsigned,which allow operations with STD_LOGIC_VECTOR data to be performed as if the data were of type SIGNED or UNSIGNED, respectively.
Example: Arithmetic operations with std_logic_vector.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all; -- extra package included

SIGNAL a: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
SOLUTION:
v <= a + b; -- legal (arithmetic operation OK), unsigned
w <= a AND b; -- legal (logical operation OK)
```

3.8 Data Conversion

VHDL does not allow direct operations (arithmetic, logical, etc.) between data of different types. Therefore, it is often necessary to convert data from one type to another .This can be done in basically two ways: or we write a piece of VHDL code for that, or we invoke a FUNCTION from a pre-defined PACKAGE which is capable of doing it for us. If the data are closely related (that is, both operands have the same base type ,despite being declared as belonging to two different type classes), then thestd_logic_1164 of the ieee library provides straightforward conversion functions. An example is shown below

Example: Legal and illegal operations with subsets.

```
TYPE long IS INTEGER RANGE -100 TO 100;
TYPE short IS INTEGER RANGE -10 TO 10;
SIGNAL x : short;
SIGNAL y : long;
SOLUTION:
y <= 2*x + 5; -- error, type mismatch
y <= long(2*x + 5); -- OK, result converted into type long
```

Several data conversion functions can be found in the std_logic_arith package ofthe ieee library. They are:

-conv_integer(p) : Converts a parameter p of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to an INTEGER value. Notice that STD_LOGIC_ VECTOR is not included.

-conv_unsigned(p, b): Converts a parameter p of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to an UNSIGNED value with size b bits.

-conv_signed(p, b): Converts a parameter p of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to a SIGNED value with size b bits.

-conv_std_logic_vector(p, b): Converts a parameter p of type INTEGER, UNSIGNED,SIGNED, or STD_LOGIC to a STD_LOGIC_VECTOR value with size bits

Example: Data conversion.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
...
SIGNAL a: IN UNSIGNED (7 DOWNTO 0);
SIGNAL b: IN UNSIGNED (7 DOWNTO 0);
SIGNAL y: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
...
y <= CONV_STD_LOGIC_VECTOR ((a+b), 8);
-- Legal operation: a+b is converted from UNSIGNED to an-- 8-bit
STD_LOGIC_VECTOR value, then assigned to y.
```

Another alternative was already mentioned in the previous section. It consists of using the std_logic_signedor the std_logic_unsigned package from the ieee library Such packages allow operations with STD_LOGIC_VECTOR data to be performed as if the data were of type SIGNED or UNSIGNED, respectively .Besides the data

conversion functions described above, several others are often offered by synthesis tool vendors.

3.9 Summary

The fundamental synthesizable VHDL data types are summarized in table 3.2.

3.10 Additional Examples We close this chapter with the presentation of additional examples illustrating the specification and use of data types. The development of actual designs from scratch will only be possible after we conclude laying out the basic foundations of VHDL(chapters 1 to 4).

Example 3.1: Dealing with Data Types

The legal and illegal assignments presented next are based on the following type definitions and signal declarations:

```
TYPE byte IS ARRAY (7 DOWNTO 0) OF STD_LOGIC; -- 1D-- array
TYPE mem1 IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;--2D-- array
TYPE mem2 IS ARRAY (0 TO 3) OF byte; -- 1Dx1D-- array
TYPE mem3IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(0 TO 7); -- 1Dx1D─ array
SIGNAL a: STD_LOGIC; -- scalar signal
SIGNAL b: BIT; -- scalar signal
SIGNAL x: byte; -- 1D signal
SIGNAL y: STD_LOGIC_VECTOR (7 DOWNTO 0); -- 1D signal
SIGNAL v: BIT_VECTOR (3 DOWNTO 0); -- 1D signal
SIGNAL z: STD_LOGIC_VECTOR (x'HIGH DOWNTO 0); -- 1D signal
SIGNAL w1: mem1; -- 2D signal
SIGNAL w2: mem2; -- 1Dx1D signal
SIGNAL w3: mem3; -- 1Dx1D signal
-------- Legal scalar assignments: ----------
x(2) <= a; -- same types (STD_LOGIC), correct indexing
y(0) <= x(0); -- same types (STD_LOGIC), correct indexing
z(7) <= x(5); -- same types (STD_LOGIC), correct indexing
b <= v(3); -- same types (BIT), correct indexing
w1(0,0) <= x(3); -- same types (STD_LOGIC), correct indexing
w1(2,5) <= y(7); -- same types (STD_LOGIC), correct indexing
w2(0)(0) <= x(2); -- same types (STD_LOGIC), correct indexing
w2(2)(5) <= y(7); -- same types (STD_LOGIC), correct indexing
w1(2,5) <= w2(3)(7); -- same types (STD_LOGIC), correct indexing
------- Illegal scalar assignments: ----------
b <= a; -- type mismatch (BIT x STD_LOGIC)
w1(0)(2) <= x(2); -- index of w1 must be 2D
w2(2,0) <= a; -- index of w2 must be 1Dx1D
------- Legal vector assignments: ----------
x <= "11111110";
y <= ('1','1','1','1','1','1','0','Z');
z <= "11111" & "000";
x <= (OTHERS => '1');
y <= (7 =>'0', 1 =>'0', OTHERS => '1');
z <= y;
y(2 DOWNTO 0) <= z(6 DOWNTO 4);
```

```
w2(0)(7 DOWNTO 0) <= "11110000";
w3(2) <= y;
z <= w3(1);
z(5 DOWNTO 0) <= w3(1)(2 TO 7);
w3(1) <= "00000000";
w3(1) <= (OTHERS => '0');
w2 <= ((OTHERS=>'0'),(OTHERS=>'0'),(OTHERS=>'0'),(OTHERS=>'0'));
w3 <= ("11111100", ('0','0','0','0','Z','Z','Z','Z',),
(OTHERS=>'0'), (OTHERS=>'0'));
w1 <= ((OTHERS=>'Z'), "11110000" ,"11110000", (OTHERS=>'0'));

------ Illegal array assignments: ----------------------
x <= y; -- type mismatch
y(5 TO 7) <= z(6 DOWNTO 0); -- wrong direction of y
w1 <= (OTHERS => '1'); -- w1 is a 2D array
w1(0, 7 DOWNTO 0) <="11111111"; -- w1 is a 2D array
w2 <= (OTHERS => 'Z'); -- w2 is a 1Dx1D array
w2(0, 7 DOWNTO 0) <= "11110000"; -- index should be 1Dx1D

-- Example of data type independent array initialization:
FOR i IN 0 TO 3 LOOP
FOR j IN 7 DOWNTO 0 LOOP
x(j) <= '0';
y(j) <= '0'
z(j) <= '0';
w1(i,j) <= '0';
w2(i)(j) <= '0';
w3(i)(j) <= '0';
END LOOP;
```

Example 3.2: Single Bit Versus Bit Vector

This example illustrates the difference between a single bit assignment and a bit vector assignment (that is, BIT versus BIT_VECTOR, STD_LOGIC versus STD_ LOGIC_VECTOR, or STD_ULOGIC versus STD_ULOGIC_VECTOR).
Two VHDL codes are presented below. Both perform the AND operation between the input signals and assign the result to the output signal. The only difference between them is the number of bits in the input and output ports (one bit in the first, four bits in the second).
The circuits inferred from these codes are shown in figure3.2.

```
----------------------------
ENTITY and2 IS
PORT (a, b: IN BIT;
x: OUT BIT);
END and2;
-----------------------------------
ENTITY and2 IS
PORT (a, b: IN BIT_VECTOR (0 TO 3);
```

```
x: OUT BIT_VECTOR (0 TO 3));
END and2;
----------------------------
ARCHITECTURE and2 OF and2 IS
BEGIN
x <= a AND b;
END and2;
----------------------------------
ARCHITECTURE and2 OF and2 IS
BEGIN
x <= a AND b;
END and2;
```

Example 3.3: Adder

Figure 3.3 shows the top-level diagram of a 4-bit adder. The circuit has two inputs(a, b) and one output (sum). Two solutions are presented. In the first, all signals are of type SIGNED, while in the second the output is of type INTEGER. Notice in solution 2 that a conversion function was used in line 13, for the type of a b does not match that of sum. Notice also the inclusion of the std_logic_arith package (line4 of each solution), which specifies the SIGNED data type. Recall that a SIGNED value is represented like a vector; that is, similar to STD_LOGIC_VECTOR, not like an INTEGER.

```
1 ----- Solution 1: in/out=SIGNED ----------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 USE ieee.std_logic_arith.all;
5 ----------------------------------------
6 ENTITY adder1 IS
7 PORT ( a, b : IN SIGNED (3 DOWNTO 0);
8 sum : OUT SIGNED (4 DOWNTO 0));
9 END adder1;
10 ----------------------------------------
11 ARCHITECTURE adder1 OF adder1 IS
12 BEGIN
13 sum <= a + b;
14 END adder1;
15 ----------------------------------------
```

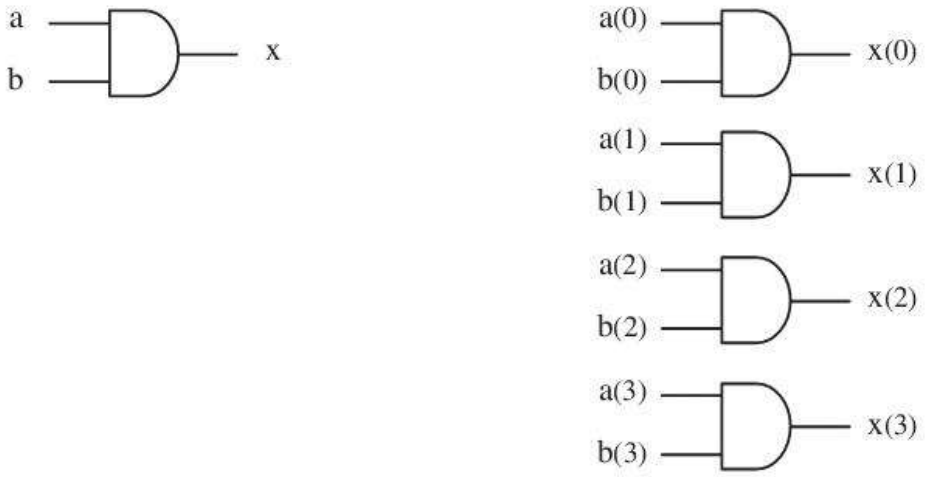


Figure 3.2
Circuits inferred from the codes of example 3.2.

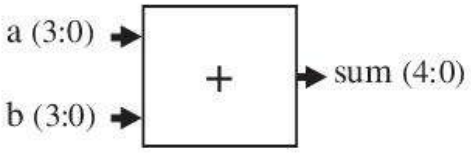


Figure 3.3
4-bit adder of example 3.3.

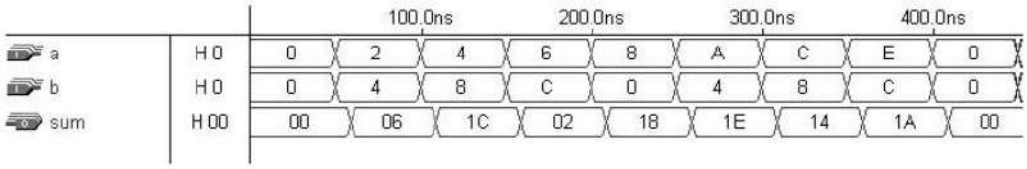


Figure 3.4
Simulation results of example 3.3.

```
1 ------ Solution 2: out=INTEGER----------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 USE ieee.std_logic_arith.all;
5 ---------------------------------------
6 ENTITY adder2 IS
7 PORT ( a, b : IN SIGNED (3 DOWNTO 0);
8 sum : OUT INTEGER RANGE -16 TO 15);
9 END adder2;
10 ---------------------------------------
11 ARCHITECTURE adder2 OF adder2 IS
```

```
12 BEGIN
13 sum <= CONV_INTEGER(a + b);
14 END adder2;
15 ----------------------------------------
```

Simulation results (for either solution) are presented in figure 3.4. Notice that the numbers are represented in hexadecimal 2's complement form. Since the input range is from(- 8 to 7), its representation is(7 →7), 6 →6, . . . ,, 0 →0, -1 →15, -2 →14,. . . , -8 →8. Likewise, the output range is from -16 to 15, so its representation is15 →15, . . . , 0 →0, -1 →31, . . . , -16 →16. Therefore, 2H 4H =06H (that is,2 +4 = 6), 4H +8H = 1CH (that is, 4 + (-8) =-4), etc., where H = Hexadecimal

4 -Operators and Attributes

The purpose of this chapter, along with the preceding chapters, is to lay the basic foundations of VHDL, so in the next chapter we can start dealing with actual circuit designs.
It is indeed impossible—or little productive, at least—to write any code efficiently without undertaking first the sacrifice of understanding data types, operators, and attributes well.
Operators and attributes constitute a relatively long list of general VHDL constructs, which are often examined only sparsely. We have collected them together in

4.1-Operators

***VHDL** provides several kinds of pre-defined operators:*

1- Assignment operators
2- Logical operators
3- Arithmetic operators
4- Relational operators
5- Shift operators
6- Concatenation operators

Each of these categories is described below.

1-Assignment Operators

Are used to assign values to signals, variables, and constants. They are:
<=Used to assign a value to a SIGNAL.
:=Used to assign a value to a VARIABLE, CONSTANT, or GENERIC. Usedalso for establishing initial values.
=>Used to assign values to individual vector elements or with OTHERS.

Example: Consider the following signal and variable declarations:

```
SIGNAL x : STD_LOGIC;
VARIABLE y : STD_LOGIC_VECTOR(3 DOWNTO 0); -- Leftmost bit is MSB
SIGNAL w: STD_LOGIC_VECTOR(0 TO 7); -- Rightmost bit is
-- MSB
```

Then the following assignments are **legal:**

```
x <= '1'; -- '1' is assigned to SIGNAL x using "<="
y := "0000"; -- "0000" is assigned to VARIABLE y using ":="
w <= "10000000"; -- LSB is '1', the others are '0'
w <= (0 =>'1', OTHERS =>'0'); -- LSB is '1', the others are '0'
```

Logical Operators

Used to perform logical operations. The data must be of type BIT, STD_LOGIC,
or STD_ULOGIC (or, obviously, their respective extensions, BIT_VECTOR,
STD_LOGIC_VECTOR, or STD_ULOGIC_VECTOR). ***The logical operators are:***

_ NOT
_ AND
_ OR
_ NAND
_ NOR
_ XOR
_ XNOR

Notes: The NOT operator has precedence over the others. The XNOR operator was
introduced in VHDL93.

Examples:

```
y <= NOT a AND b; -- (a'.b)
y <= NOT (a AND b); -- (a.b)'
y <= a NAND b; -- (a.b)'
```

2-Arithmetic Operators

Used to perform arithmetic operations. The data can be of type INTEGER,
SIGNED, UNSIGNED, or REAL (recall that the last cannot be synthesized directly).
Also, if the std_logic_signed or the std_logic_unsigned package of the ieeelibrary is used, then STD_LOGIC_VECTOR can also be employed directly in addition

+Addition
-Subtraction
*** Multiplication**
/ Division
**** Exponentiation**
MOD Modulus
REM Remainder
ABS Absolute value

There are no synthesis restrictions regarding addition and subtraction, and the same is generally true for multiplication. For division, only power of two dividers are allowed. For exponentiation, only static values of base and exponent are accepted. Regarding the mod and rem operators, y mod x returns the remainder of y/x with the signal of x,
while y rem x returns the remainder of y/x with the signal of y. Finally, abs returns the absolute value. With respect to the last three operators (mod, rem, abs),there generally is little or no synthesis support.

Comparison Operators

Used for making comparisons. The data can be of any of the types listed above. The relational (comparison) operators are:
=Equal to
≠Not equal to
<Less than
>Greater than
<=Less than or equal to
>=Greater than or equal to

Shift Operators

Used for shifting data. They were introduced in VHDL93. Their syntax is the following:
(left operand)(shift operation)(right operand). The left operand must be of type BIT_VECTOR, while the right operand must be an INTEGER (+or -infront of it is accepted).
The shift operators are:
1-sll Shift left logic– positions on the right are filled with ‘0’s
2-srl Shift right logic – positions on the left are filled with ‘0’s

Data Attributes

The pre-defined, synthesizable data attributes are the following:
1-d’LOW: Returns lower array index
2-d’HIGH: Returns upper array index
3-d’LEFT: Returns leftmost array index
4-d’RIGHT: Returns rightmost array index
5-d’LENGTH: Returns vector size
6-d’RANGE: Returns vector range
7d’REVERSE_RANGE: Returns vector range in reverse order

Example: Consider the following signal:
SIGNAL d : STD_LOGIC_VECTOR (7 DOWNTO 0);
Then:
d'LOW=0, d'HIGH=7, d'LEFT=7, d'RIGHT=0, d'LENGTH=8,
d'RANGE=(7 downto 0), d'REVERSE_RANGE=(0 to 7).

Example: Consider the following signal:
SIGNAL x: STD_LOGIC_VECTOR (0 TO 7);
Then all four LOOP statements below are synthesizable and equivalent.
FOR i IN RANGE (0 TO 7) LOOP ...
FOR i IN x'RANGE LOOP ...
FOR i IN RANGE (x'LOW TO x'HIGH) LOOP...
FOR i IN RANGE (0 TO x'LENGTH-1) LOOP...
If the signal is of enumerated type, then:
-d'VAL(pos): Returns value in the position specified
-d'POS(value): Returns position of the value specified
-d'LEFTOF(value): Returns value in the position to the left of the value specified
-d'VAL(row, column): Returns value in the position specified; etc.
There is little or no synthesis support for enumerated data type attributes.

Signal Attributes

Let us consider a signal s. Then:
-s'EVENT: Returns true when an event occurs on s
-s'STABLE: Returns true if no event has occurred on s
-s'ACTIVE: Returns true if **S**='1'
-'QUIET (time): Returns true if no event has occurred during the time specified
-s'LAST_EVENT: Returns the time elapsed since last event
-s'LAST_ACTIVE: Returns the time elapsed since last **S**='1'
-s'LAST_VALUE: Returns the value of s before the last event; etc.
Though most signal attributes are for simulation purposes only, the first two in the list above are synthesizable, s'EVENT being the most often used of them all.
Example: All four assignments shown below are synthesizable and equivalent. They return TRUE when an event (a change) occurs on clk, AND if such event is upward (in other words, when a rising edge occurs on clk).
IF (clk'EVENT AND clk='1')... -- EVENT attribute used-- with IF
IF (NOT clk'STABLE AND clk='1')... -- STABLE attribute used-- with IF
WAIT UNTIL (clk'EVENT AND clk='1'); -- EVENT attribute used-- with WAIT
IF RISING_EDGE(clk)... -- call to a function

4.3 User-Defined Attributes

We saw above attributes of the type HIGH, RANGE, EVENT, etc. Those are all pre-defined in VHDL87. However, VHDL also allows the construction of user defined attributes To employ a user-defined attribute, it must be declared and specified. The syntax is the following

Attribute declaration:

```
ATTRIBUTE attribute_name: attribute_type;
```

Attribute specification:

```
ATTRIBUTE attribute_name OF target_name: class IS value;
```

where:
attribute_type:any data type (BIT, INTEGER, STD_LOGIC_VECTOR, etc.)
class: TYPE, SIGNAL, FUNCTION, etc.
value: '0', 27, ''00 11 10 01'', etc.

4.4 GENERIC

As the name suggests, GENERIC is a way of specifying a generic parameter (that is, a static parameter that can be easily modified and adapted to different applications). The purpose is to confer the code more flexibility and reusability.
A GENERIC statement, when employed, must be declared in the ENTITY. The specified parameter will then be truly global (that is, visible to the whole design, including the ENTITY itself). Its syntax is shown below.

```
GENERIC (parameter_name : parameter_type := parameter_value);
```

Example: The GENERIC statement below specifies a parameter called n, of type INTEGER, whose default value is 8. Therefore, whenever **n** is found in the ENTITY itself or in the ARCHITECTURE (one or more) that follows, its value will be assumed to be 8.

```
ENTITY my_entity IS
GENERIC (n : INTEGER := 8);
PORT (...);
END my_entity;
ARCHITECTURE my_architecture OF my_entity IS
.....
END my_architecture:
```

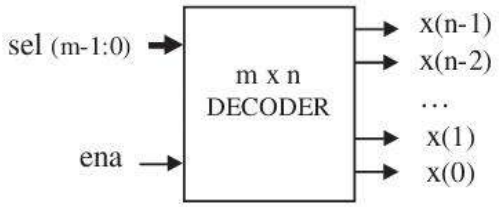
4.5 Examples

We show now a few complete design examples, with the purpose of further illustrating the use of operators, **attributes** and **GENERIC** . Recall, however, that so far we have just worked on establishing the basic foundations of VHDL, with the formal discussion

the student must return and reexamine later

Example 4.1: Generic Decoder

Figure 4.1 shows the top-level diagram of a generic m-by-n decoder. The circuit has two inputs, sel (m bits) and ena (single bit), and one output, x (n bits). We assume that n is a power of two, so m =log2n. If ena =‘0’, then all bits of x should be high;



ena	sel	x
0	00	1111
1	00	1110
	01	1101
	10	1011
	11	0111

Figure 4.1
Decoder of example 4.1.

otherwise, the output bit selected by ***sel*** should be ***low***, as illustrated in the truth table of figure 4.1. The ARCHITECTURE below is totally generic, for the only changes needed to operate with different values of **m** and **n** are in the ENTITY (through sel, line 7, and x, line 8, respectively). In this example, we have used m =3 and n =8. However, though this works fine, the use of GENERIC would have made it clearer that m andn are indeed generic parameters. That is indeed the procedure that we will adopt inthe other examples that follow (please refer to problem 4.4).

Notice in the code below the use of the following operators:"+" (line 22), ‘‘*’’ (lines 22 and 24), ‘‘:=’’ (lines 17, 18, 22, 24, and 27), ‘‘<=’’ (line 29), and ‘‘=>’’ (line 17). Notice also the use of the following attributes: HIGH (lines 14–15) and RANGE (line 20).

```
1 ----------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------------
5 ENTITY decoder IS
6 PORT ( ena : IN STD_LOGIC;
7 sel : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
8 x : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
9 END decoder;
10 ----------------------------------------
11 ARCHITECTURE generic_decoder OF decoder IS
12 BEGIN
13 PROCESS (ena, sel)
```

```
14 VARIABLE temp1 : STD_LOGIC_VECTOR (x'HIGH DOWNTO 0);
15 VARIABLE temp2 : INTEGER RANGE 0 TO x'HIGH;
16 BEGIN

17 temp1 := (OTHERS => '1');
18 temp2 := 0;
19 IF (ena='1') THEN
20 FOR i IN sel'RANGE LOOP -- sel range is 2 downto 0
21 IF (sel(i)='1') THEN -- Bin-to-Integer conversion
22 temp2:=2*temp2+1;
23 ELSE
24 temp2 := 2*temp2;
25 END IF;
26 END LOOP;
27 temp1(temp2):='0';
28 END IF;
29 x <= temp1;
30 END PROCESS;
31 END generic_decoder;
32 ---------------------------------------------
```

The functionality of the encoder above can be verified in the simulation results of figure 4.2. As can be seen, all outputs are high, that is, x =‘‘11111111’’ (decimal 255), when ena =‘0’. After ena has been asserted, only one output bit (that selected by sel) is turned low. For example, when sel =‘‘000’’ (decimal 0), x =‘‘11111110’’ (decimal 254); when sel =‘‘001’’ (decimal 1), x =‘‘11111101’’ (decimal 253); when sel =‘‘010’’ (decimal 2), x =‘‘11111011’’ (decimal 251); and so on.

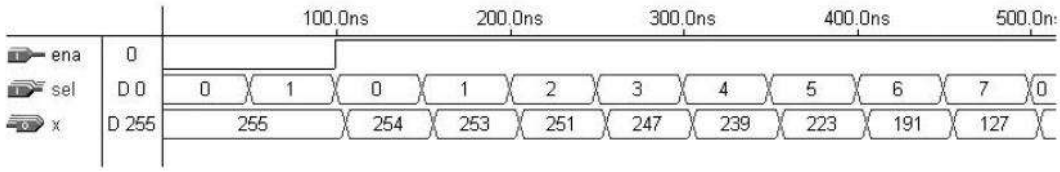


Figure 4.2
Simulation results of example 4.1.

Example 4.2: Generic Parity Detector

Figure 4.3 shows the top-level diagram of a parity detector. The circuit must provide output =‘0’ when the number of ‘1’s in the input vector is even, or output =‘1’ otherwise. Notice in the VHDL code below that the ENTITY contains a GENERIC statement (line 3), which defines n as 7. This code would work for any other vector

size, being only necessary to change the value of n in that line. You are invited to highlight the operators and attributes that appear in this design.

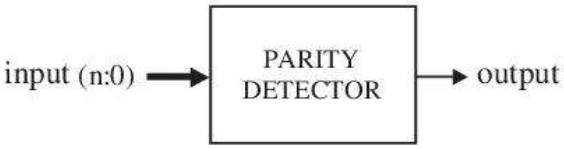


Figure 4.3
Generic parity detector of example 4.2.

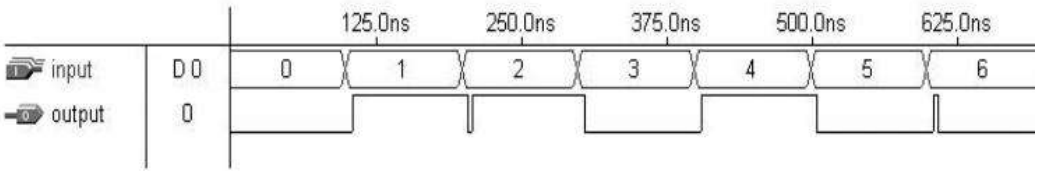


Figure 4.4
Simulation results of example 4.2.

```
1 ---------------------------------------------
2 ENTITY parity_det IS
3 GENERIC (n : INTEGER := 7);
4 PORT ( input: IN BIT_VECTOR (n DOWNTO 0);
5 output: OUT BIT);
6 END parity_det;
7 ---------------------------------------------
8 ARCHITECTURE parity OF parity_det IS
9 BEGIN
10 PROCESS (input)
11 VARIABLE temp: BIT;
12 BEGIN
13 temp := '0';
14 FOR i IN input'RANGE LOOP
15 temp := temp XOR input(i);
16 END LOOP;
17 output <= temp;
18 END PROCESS;
19 END parity;
20 ---------------------------------------------
```

Simulation results from the circuit synthesized with the code above are shown in figure 4.4. Notice that when input =‘‘00000000’’ (decimal 0), the output is ‘0’, because the number of ‘1’s is even; when input =‘‘00000001’’ (decimal 1), the output is‘1’, because the number of ‘1’s is odd; and so on.

Example 4.3: Generic Parity Generator

The circuit of figure 4.5 must add one bit to the input vector (on its left). Such bit must be a ‘0’ if the number of ‘1’s in the input vector is even, or a ‘1’ if it is odd, such that the resulting vector will always contain an even number of ‘1’s (even parity).

A VHDL code for the parity generator is shown below. Once again, you are invited to highlight the operators and attributes used in the design.

```
1 ---------------------------------------------
2 ENTITY parity_gen IS
3 GENERIC (n : INTEGER := 7);
4 PORT ( input: IN BIT_VECTOR (n-1 DOWNTO 0);
5 output: OUT BIT_VECTOR (n DOWNTO 0));
6 END parity_gen;
7 ---------------------------------------------
8 ARCHITECTURE parity OF parity_gen IS
9 BEGIN
10 PROCESS (input)
11 VARIABLE temp1: BIT;
12 VARIABLE temp2: BIT_VECTOR (output' RANGE);
13 BEGIN
14 temp1 := '0';
15 FOR i IN input' RANGE LOOP
16 temp1 := temp1 XOR input(i);
17 temp2(i) := input(i);
18 END LOOP;
19 temp2(output' HIGH) := temp1;
20 output <= temp2;
21 END PROCESS;
22 END parity;
23 ------------------------------------
```

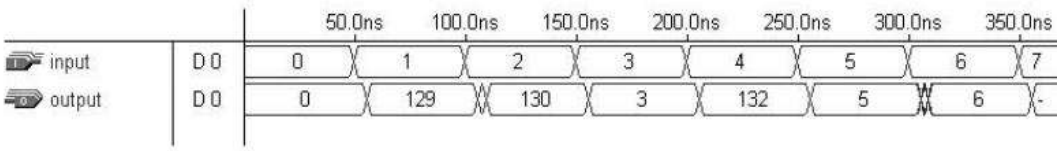


Figure 4.6
Simulation results of example 4.3.

Figure 4.5
Generic parity generator of example 4.3.

input (n-1:0) PARITY GENERATOR
output (n:0)

Simulation results are presented in figure 4.6. As can be seen, when input = ‘‘0000000’’ (decimal 0, with seven bits), output =‘‘00000000’’ (decimal 0, with eight bits); when input =‘‘0000001’’ (decimal 1, with seven bits), output =‘‘10000001’’ (decimal 129, with eight bits); and so on.

Operators.

Operator type	Operators	Data types
Assignment	<=, :=, =>Any	
Logical	NOT, AND, NAND,	BIT, BIT_VECTOR,
	OR, NOR, XOR, XNOR	STD_LOGIC, STD_LOGIC_VECTOR, STD_ULOGIC, STD_ULOGIC_VECTOR
Arithmetic	,- , *, /, ** (mod, rem, abs))	INTEGER, SIGNED, UNSIGNED
Comparison	=, ≠, <, >, <=, >=	All above
Shift	sll, srl, sla, sra, rol, ror	BIT_VECTOR
Concatenation	&, (, , ,)	Same as for logical operators, plus SIGNED and UNSIGNED

5-Concurrent Code

The concurrent statements in VHDL are **WHEN** and **GENERATE**. Besides them, assignments using only operators (AND, NOT,+, *, sll, etc.) can also be used to construct concurrent code. Finally, a special kind of assignment, called BLOCK, can also be employed in this kind of code.

5.1 Concurrent versus Sequential

We start this chapter by reviewing the fundamental differences between **combinational logic** and **sequential logic,** and by contrasting them with the differences between concurrent code and sequential code.

Combinational versus Sequential Logic

By definition, combinational logic is that in which the output of the circuit depends only on the current inputs

(figure 5.1(a))., ***the system requires no memory*** and can be implemented using conventional logic gates.

sequential logic is defined as that in which the output does depend on previous inputs (figure 5.1(b)).*Therefore, storage elements are required,*

the output of the circuit. a common mistake is to think that any circuit that possesses storage elements(flip-flops) is sequential

.A RAM (Random Access Memory) can be modeled as in figure 5.2. Notice that the storage elements appear in a forward path rather than in a feedback loop.

The memory-read operation depends only on the address vector presently applied to the RAM input, with the retrieved value

Concurrent versus Sequential Code

VHDL code is inherently **concurrent (parallel).** Only statements placed inside a PROCESS, FUNCTION, or PROCEDURE are **sequential.** Still, though within these blocks the execution is sequential, the block, as a whole, is concurrent with any other (external) statements. Concurrent code is also called dataflow code.

Example, let us consider a code with three concurrent statements (stat1, stat2, stat3). Then any of the alternatives below will render the same physical circuit:

stat1 stat3 stat1

stat2 ≡stat2 ≡stat3 ≡etc.

stat3stat1 stat2

It is then clear that, since the order does not matter, purely concurrent code can not be used to implement synchronous circuits (the only exception is when a

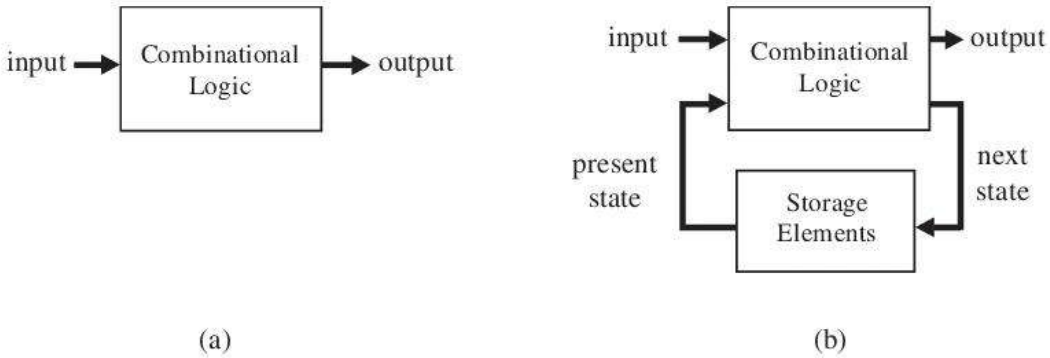


Figure 5.1
Combinational (a) versus sequential (b) logic.

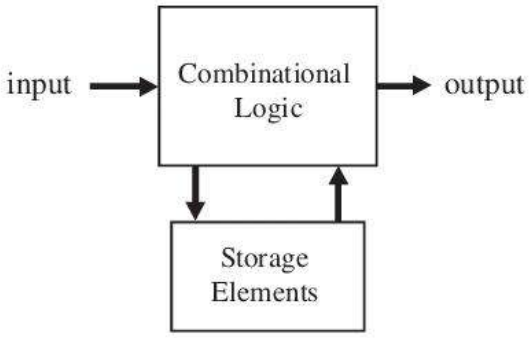


Figure 5.2
RAM model.

Example, let us consider a code with three concurrent statements (stat1, stat2, stat3). Then any of the alternatives below will render the same physical circuit:
stat1 stat3 stat1
stat2 ≡stat2 ≡stat3 ≡etc.
stat3stat1 stat2
In general we can only build combinational logic circuits with concurrent code. obtain
We will discuss concurrent code, that is, we will study the statements
That can only be used outside PROCESSES, FUNCTIONS, or PROCEDURES.
They are the **WHEN** statement and the **GENERATE** statement. Besides them, assignments using only operators (logical, arithmetic, etc) can obviously also be used to create combinational circuits. Finally, a special kind of statement, called **BLOCK**, can also be employed In summary, in concurrent code the following can be used:
1-Operators;
2-The WHEN statement (WHEN/ELSE or WITH/SELECT/WHEN);
3-The GENERATE statement;
4-The BLOCK statement.
Each of these cases is described below.

5.2 Using Operators

This is the **most basic** way of **creating concurrent code.** Operators (AND, OR, , _. *, sll, sra, etc.) were discussed in section 4.1, being a summary repeated in table 5.1 below.

Operators can be used to implement any combinational circuit. However, as will become apparent later, **complex circuits are usually easier to write using sequential code,**
even if the circuit does not contain sequential logic. In the example that follows, a design using only logical operators is presented.

Table 5.1
Operators.

Operator type	Operators	Data types
Logical	NOT, AND, NAND, OR, NOR, XOR, XNOR	BIT, BIT_VECTOR, STD_LOGIC,STD_LOGIC_VECTOR STD_ULOGIC,STD_ULOGIC_VECTOR
Arithmetic	, _, *, /, **	INTEGER, SIGNED, UNSIGNED (mod, rem, abs)
Comparison	=, ≠, <, >, <=, >=	All above
Shift	sll, srl, sla, sra, rol, ror	BIT_VECTOR
Concatenation	&, (, , ,)	Same as for logical operators, plus SIGNED and UNSIGNED

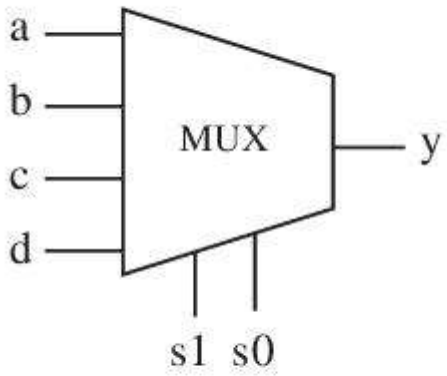


Figure 5.3
Multiplexer of example 5.1.

Example 5.1: Multiplexer #1

Write the VHDL code for the circuit in the figure(5-3)by using the logic ?

Figure 5.3 shows a 4-input, one bit per input multiplexer. The output must be equal to the input selected by the selection bits, s1-s0. Its implementation, using only logical operators, can be done as follows:

```
1 ----------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------------
5 ENTITY mux IS
6 PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
7 y: OUT STD_LOGIC);
8 END mux;
9 ----------------------------------------
10 ARCHITECTURE pure_logic OF mux IS
11 BEGIN
12 y <= (a AND NOT s1 AND NOT s0) OR
13 (b AND NOT s1 AND s0) OR
14 (c AND s1 AND NOT s0) OR
15 (d AND s1 AND s0);
16 END pure_logic;
17 ----------------------------------------
```

Simulation results, confirming the functionality of the circuit, are shown in figure5.4

.

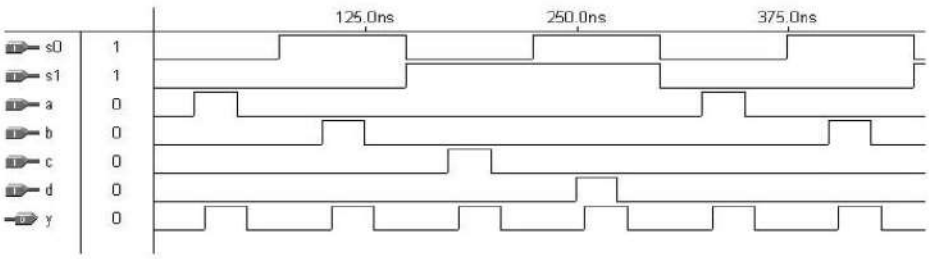


Figure 5.4
Simulation results of example 5.1.

5.3 WHEN (Simple and Selected)

As mentioned above, WHEN is one of the **fundamental concurrent state**ments (along with operators and GENERATE). It appears in two forms: WHEN / ELSE (simple WHEN) and WITH / SELECT / WHEN (selected WHEN). Its syntax is shown below.

WHEN / ELSE:

```
assignment WHEN condition ELSE
assignment WHEN condition ELSE
...;
```

WITH / SELECT / WHEN:

```
WITH identifier SELECT
assignment WHEN value,
assignment WHEN value,
...;
```

Whenever **WITH/ SELECT/WHEN** is used, all permutations must be tested, so the keyword OTHERS is often useful. Another important keyword is UNAFFECTED, which should be used when no action is to take place.

Example

```
:
------ With WHEN/ELSE ------------------------
outp <= "000" WHEN (inp='0' OR reset='1') ELSE
"001" WHEN ctl='1' ELSE
"010";
---- With WITH/SELECT/WHEN --------------------
WITH control SELECT
output <= "000" WHEN reset,
"111" WHEN set,
UNAFFECTED WHEN OTHERS;
-----------------------------------------------
```

Another important aspect related to the WHEN statement is that the ‘‘WHEN value’’ shown in the syntax above can indeed take up three forms:
1-WHEN value -- single value
2-WHEN value1 to value2 -- range, for enumerated data types-- only
3-WHEN value1 | value2 |... -- value1 or value2 or ...

Example 5.2: Multiplexer #2

This example shows the implementation of the same multiplexer of example 5.1, but with a slightly different representation for the sel input (figure 5.5). However, in it WHEN was employed instead of logical operators. Two solutions are presented: one using WHEN/ELSE (simple WHEN) and the other with WITH/SELECT/WHEN (selected WHEN). The experimental results are obviously similar to those obtained in example 5.1.

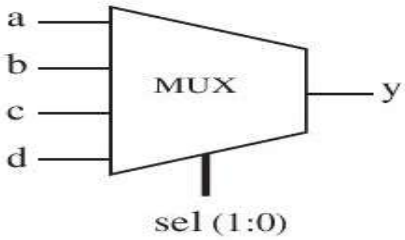


Figure 5.5
Multiplexer of example 5.2.

1 ------- **Solution 1: with WHEN/ELSE**--------
```
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ---------------------------------------
5 ENTITY mux IS
6 PORT ( a, b, c, d: IN STD_LOGIC;
7 sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
8 y: OUT STD_LOGIC);
9 END mux;
10 ---------------------------------------
11 ARCHITECTURE mux1 OF mux IS
12 BEGIN
13 y <= a WHEN sel="00" ELSE
14 b WHEN sel="01" ELSE
15 c WHEN sel="10" ELSE
16 d;
17 END mux1;
18 ---------------------------------------
```

1 --- **Solution 2: with WITH/SELECT/WHEN**-----
```
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ---------------------------------------
5 ENTITY mux IS
6 PORT ( a, b, c, d: IN STD_LOGIC;
7 sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
8 y: OUT STD_LOGIC);
9 END mux;
10 ---------------------------------------
11 ARCHITECTURE mux2 OF mux IS
12 BEGIN
13 WITH sel SELECT
14 y <= a WHEN "00", -- notice "," instead of ";"
15 b WHEN "01",
16 c WHEN "10",
17 d WHEN OTHERS; -- cannot be "d WHEN "11" "
18 END mux2;
19 ---------------------------------------
```
In the solutions above, sel could have been declared as an INTEGER, in which case the code would be the following:
```
1 ---------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ---------------------------------------
5 ENTITY mux IS
6 PORT ( a, b, c, d: IN STD_LOGIC;
7 sel: IN INTEGER RANGE 0 TO 3;
```

```
8 y: OUT STD_LOGIC);
9 END mux;
10 ---- Solution 1: with WHEN/ELSE ----------
11 ARCHITECTURE mux1 OF mux IS
12 BEGIN
13 y <= a WHEN sel=0 ELSE
14 b WHEN sel=1 ELSE
15 c WHEN sel=2 ELSE
16 d;
17 END mux1;

18 -- Solution 2: with WITH/SELECT/WHEN --------
19 ARCHITECTURE mux2 OF mux IS
20 BEGIN
21 WITH sel SELECT
22 y <= a WHEN 0,
23 b WHEN 1,
24 c WHEN 2,
25 d WHEN 3; -- here, 3 or OTHERS are equivalent,
26 END mux2; -- for all options are tested anyway
27 ----------------------------------------------
```

Note: Only one ARCHITECTURE can be synthesized at a time. Therefore, whenever we show more than one solution within the same overall code (like above), it is implicit that all solutions but one must be commented out (with ‘‘- -’’), or a synthesis script must be used, in order to synthesize the remaining solution. In simulations, the CONFIGURATION statement can be used to select a specific architecture.
Note: For a generic mux, please refer to problem 5.1.

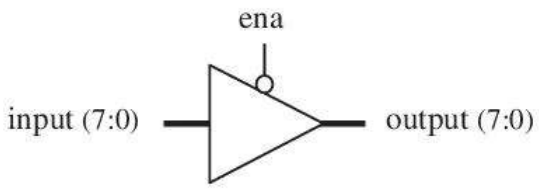


Figure 5.6
Tri-state buffer of example 5.3.

Example 5.3: Tri-state Buffer

This is another example that illustrates the use of WHEN. The 3-state buffer of figure 5.6 must provide output =input when ena (enable) is low, or output = ‘‘ZZZZZZZZ’’(high impedance) otherwise.

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3 ----------------------------------------------
4 ENTITY tri_state IS
5 PORT ( ena: IN STD_LOGIC;
6 input: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
```

```
7 output: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
8 END tri_state;
9 -----------------------------------------
10 ARCHITECTURE tri_state OF tri_state IS
11 BEGIN
12 output <= input WHEN (ena='0') ELSE
13 (OTHERS => 'Z');
14 END tri_state;
15 -----------------------------------------
```

Simulation results from the circuit synthesized with the code above are shown in figure 5.7. As expected, the output stays in the high-impedance state while ena is high, being a copy of the input when ena is turned low.

Example 5.4: Encoder

The top-level diagram of an n-by-m encoder is shown in figure 5.8. We assume that n is a power of two, so m =log2n. One and only one input bit is expected to be high at a time, whose address must be encoded at the output. Two solutions are presented, one using WHEN / ELSE, and the other with WITH / SELECT / WHEN.

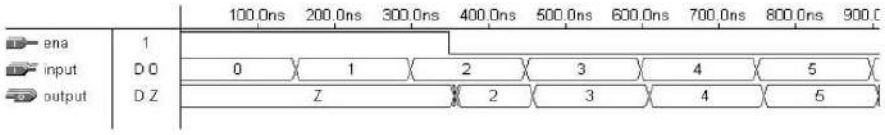


Figure 5.7
Simulation results of example 5.3.

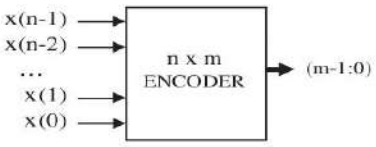


Figure 5.8
Encoder of example 5.4.

```
1 ---- Solution 1: with WHEN/ELSE ------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----------------------------------------
5 ENTITY encoder IS
6 PORT ( x: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7 y: OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
8 END encoder;
9 -----------------------------------------
10 ARCHITECTURE encoder1 OF encoder IS
11 BEGIN
12 y <= "000" WHEN x="00000001" ELSE
```

```
13 "001" WHEN x="00000010" ELSE
14 "010" WHEN x="00000100" ELSE
15 "011" WHEN x="00001000" ELSE
16 "100" WHEN x="00010000" ELSE
17 "101" WHEN x="00100000" ELSE
18 "110" WHEN x="01000000" ELSE
19 "111" WHEN x="10000000" ELSE
20 "ZZZ";
21 END encoder1;
22 -------------------------------------------
```

1 ---- **Solution 2: with WITH/SELECT/WHEN**------

```
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -------------------------------------------
5 ENTITY encoder IS
6 PORT ( x: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7 y: OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
8 END encoder;
9 -------------------------------------------
10 ARCHITECTURE encoder2 OF encoder IS
11 BEGIN
12 WITH x SELECT
13 y <= "000" WHEN "00000001",
14 "001" WHEN "00000010",
15 "010" WHEN "00000100",
16 "011" WHEN "00001000",
17 "100" WHEN "00010000",
18 "101" WHEN "00100000",
19 "110" WHEN "01000000",
20 "111" WHEN "10000000",
21 "ZZZ" WHEN OTHERS;
22 END encoder2;
23 -------------------------------------------
```

Simulation results (from either solution) are shown in figure 5.9

Example 5.5: ALU

An **ALU** (**Arithmetic Logic Unit**) is shown in figure 5.10. As the name says, it is a circuit capable of executing both kinds of operations, arithmetic as well as logical. Its operation is described in the truth table of figure 5.10. The output (arithmetic or logical) is selected by the MSB of sel, while the specific operation is selected by sel's other three bits.

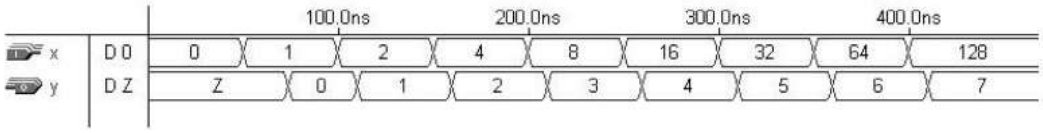
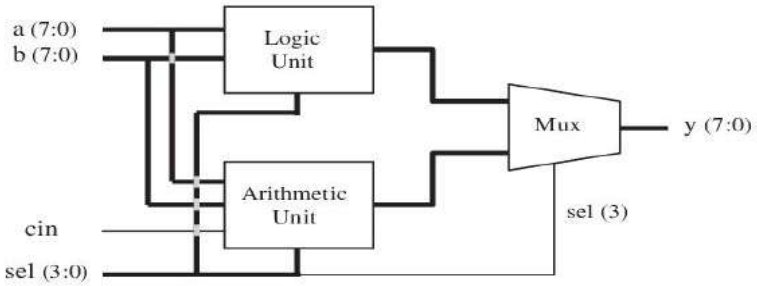


Figure 5.9
Simulation results of example 5.4.



sel	Operation	Function	Unit
0000	y <= a	Transfer a	Arithmetic
0001	y <= a+1	Increment a	
0010	y <= a-1	Decrement a	
0011	y <= b	Transfer b	
0100	y <= b+1	Increment b	
0101	y <= b-1	Decrement b	
0110	y <= a+b	Add a and b	
0111	y <= a+b+cin	Add a and b with carry	
1000	y <= NOT a	Complement a	Logic
1001	y <= NOT b	Complement b	
1010	y <= a AND b	AND	
1011	y <= a OR b	OR	
1100	y <= a NAND b	NAND	
1101	y <= a NOR b	NOR	
1110	y <= a XOR b	XOR	
1111	y <= a XNOR b	XNOR	

Figure 5.10
ALU of example 5.5.

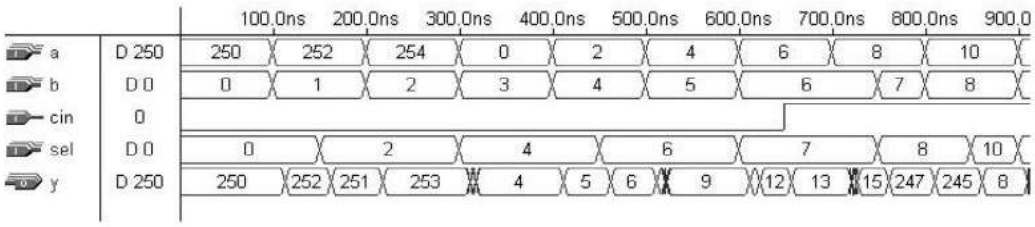


Figure 5.11
Simulation results of example 5.5.

The solution presented below, besides using **only concurrent code**, also illustrates the use of the same data type to perform both **arithmetic** and **logical operations**. That is possible due to the presence of the **std_logic_unsigned** package of the ieee library Two signals, **arith** and **logic**, are used to hold the results
from the arithmetic and logic units, respectively, being the value passed to the output selected by the multiplexer. Simulation results are shown in figure 5.11.

```
1 ----------------------------------------------
2 LIBRARY ieee;
3USE ieee.std_logic_1164.all;
4 USE ieee.std_logic_unsigned.all;
5 ----------------------------------------------
6 ENTITY ALU IS
7 PORT (a, b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
8 sel: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
9 cin: IN STD_LOGIC;
10 y: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
11 END ALU;
12 ----------------------------------------------
13 ARCHITECTURE dataflow OF ALU IS
14 SIGNAL arith, logic: STD_LOGIC_VECTOR (7 DOWNTO 0);
15 BEGIN
16 ----- Arithmetic unit: ------
17 WITH sel(2 DOWNTO 0) SELECT
18 arith <= a WHEN "000",
19 a+1 WHEN "001",
20 a-1 WHEN "010",
21 b WHEN "011",
22 b+1 WHEN "100",
23 b-1 WHEN "101",
24 a+b WHEN "110",
25 a+b+cin WHEN OTHERS;
26 ----- Logic unit: -----------
27 WITH sel(2 DOWNTO 0) SELECT
28 logic <= NOT a WHEN "000",
29 NOT b WHEN "001",
30 a AND b WHEN "010",
31 a OR b WHEN "011",
32 a NAND b WHEN "100",
33 a NOR b WHEN "101",
34 a XOR b WHEN "110",
35 NOT (a XOR b) WHEN OTHERS;
36 -------- Mux: ---------------
37 WITH sel(3) SELECT
38 y <= arith WHEN '0',
39 logic WHEN OTHERS;
40 END dataflow;
41 ----------------------------------------------
```

5.4 GENERATE

GENERATE is another **concurrent** statement (along with operators and **WHEN**). It is equivalent to the **sequential** statement **LOOP**

the same assignments. Its regular form is the FOR / GENERATE construct, with the syntax shown below. Notice that GENERATE must be labeled.

FOR / GENERATE:

```
label: FOR identifier IN range GENERATE(concurrent assignments)

END GENERATE;
```

An irregular form is also available, which uses IF/GENERATE (with an IF equivalent; recall that originally IF is a sequential statement). Here ELSE is not allowed. In the same way that IF/GENERATE can be nested inside FOR/ GENERATE (syntax below), the opposite can also be done.

IF / GENERATE nested inside FOR / GENERATE:

```
label1: FOR identifier IN range GENERATE
...
label2: IF condition GENERATE
(concurrent assignments)
END GENERATE;
...
END GENERATE;
```

Example:

```
SIGNAL x: BIT_VECTOR (7 DOWNTO 0);
SIGNAL y: BIT_VECTOR (15 DOWNTO 0);
SIGNAL z: BIT_VECTOR (7 DOWNTO 0);
...
G1: FOR i IN x'RANGE GENERATE
z(i) <= x(i) AND y(i+8);
END GENERATE;
```

One important remark about GENERATE (and the same is true for LOOP, which will be seen in chapter 6) is that both limits of the range must be static. As an example, let us consider the code below, where choice is an input (non-static) parameter. This kind of code is generally not synthesizable.

```
NotOK: FOR i IN 0 TO choice GENERATE(concurrent statements)
END GENERATE;
```

We also must to be aware of multiply-driven (unresolved) signals. For example,

```
OK: FOR i IN 0 TO 7 GENERATE
output(i)<='1' WHEN (a(i) AND b(i))='1' ELSE '0';
END GENERATE;
```

is fine. However, the compiler will complain that accum is multiply driven (and stop compilation) in either of the following two cases:

```
Not OK: FOR i IN 0 TO 7 GENERATE
accum <="11111111" WHEN (a(i) AND b(i))='1' ELSE "00000000";
END GENERATE;
Not OK: For i IN 0 to 7 GENERATE
accum <= accum + 1 WHEN x(i)='1';
END GENERATE;
```

Example 5.6: Vector Shifter

This example illustrates the use of GENERATE. In it, the output vector must be a shifted version of the input vector, with twice its width and an amount of shift specified by another input. For example, if the input bus has width 4, and the present value is ‘‘1111’’, then the output should be one of the lines of the following matrix (the original vector is underscored):

row(0): 0 0 0 0 1 1 1 1
row(1): 0 0 0 1 1 1 1 0
row(2): 0 0 1 1 1 1 0 0
row(3): 0 1 1 1 1 0 0 0
row(4): 1 1 1 1 0 0 0 0

The **first** row corresponds to the input itself, with **no shift** and the most significant bits filled with ‘0’s. Each successive row is equal to the previous row shifted one position to the left.

The solution below has **input *inp*, output *outp***, and shift selection sel. Each row of the array above (called matrix, line 14) is defined as subtype vector (line 12).

```
1 ------------------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ------------------------------------------------
5 ENTITY shifter IS
6 PORT ( inp: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
7 sel: IN INTEGER RANGE 0 TO 4;
8 outp: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
9 END shifter;
10 ------------------------------------------------
11 ARCHITECTURE shifter OF shifter IS
12 SUBTYPE vector IS STD_LOGIC_VECTOR (7 DOWNTO 0);
13 TYPE matrix IS ARRAY (4 DOWNTO 0) OF vector;
14 SIGNAL row: matrix;
15 BEGIN
16 row(0) <= "0000" & inp;
17 G1: FOR i IN 1 TO 4 GENERATE
18 row(i) <= row(i-1)(6 DOWNTO 0) & '0';
19 END GENERATE;
20 outp <= row(sel);
21 END shifter;
22 ------------------------------------------------
```

Simulation results are presented in figure 5.12. As can be seen, inp =‘‘0011’’ (decimal 3) was applied to the circuit. The result was outp =‘‘00000011’’ (decimal 3) When sel =0 (no shift), outp=‘‘00000110’’ (decimal 6) when sel =1 (one shift to the left), outp =‘‘00001100’’ (decimal 12) when sel =2 (two shifts to the left), and so on.

5.5 BLOCK

There are **two** kinds of BLOCK statements: **Simple** and **Guarded.**

Simple BLOCK

The BLOCK statement, in its simple form, represents only a way of locally partitioning
the code. It allows a set of concurrent statements to be clustered into a BLOCK, with the purpose of turning the overall code more readable and more manageable Its syntax is shown below.

```
label: BLOCK
[declarative part]
BEGIN
(concurrent statements)
END BLOCK label;
```

Example:

```
b1: BLOCK
SIGNAL a: STD_LOGIC;
BEGIN
a <= input_sig WHEN ena='1' ELSE 'Z';
END BLOCK b1;
```

A BLOCK (simple or guarded) can be nested inside another BLOCK. The corresponding
syntax is shown below.

```
label1: BLOCK
[declarative part of top block]
BEGIN
[concurrent statements of top block]
label2: BLOCK
[declarative part nested block]
BEGIN
(concurrent statements of nested block)
END BLOCK label2;
[more concurrent statements of top block]
END BLOCK label1;
```

Guarded BLOCK

A guarded BLOCK is a special kind of BLOCK, which includes an additional expression, called guard expression.
A guarded statement in a guarded BLOCK is executedonly when the guard expression is TRUE.

Guarded BLOCK:

label: BLOCK (guard expression) [declarative part]

BEGIN(concurrent guarded and unguarded statements)

END BLOCK label;

As the examples below illustrate, even though only concurrent statements can be written within a BLOCK, with a guarded BLOCK even sequential circuits can be constructed. This, however, is not a usual design approach.

Example 5.7: Latch Implemented with a Guarded BLOCK

The example presented below implements a transparent latch. In it, clk='1' (line 12) is the guard expression, while q<=GUARDED d (line 14) is a guarded statement. Therefore, q<=d will only occur if clk='1'.

```
1 ------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ------------------------------
5 ENTITY latch IS
6 PORT (d, clk: IN STD_LOGIC;
7 q: OUT STD_LOGIC);
8 END latch;
9 ------------------------------
10 ARCHITECTURE latch OF latch IS
11 BEGIN
12 b1: BLOCK (clk='1')
13 BEGIN
14 q <= GUARDED d;
15 END BLOCK b1;
16 END latch;
17 ------------------------------
```

Example 5.8: DFF Implemented with a Guarded BLOCK

Here, a positive-edge sensitive D-type flip-flop, with synchronous reset, is designed. The interpretation of the code is similar to that in the example above. In it, clk'EVENT AND clk='1' (line 12) is the guard expression, while q <= GUARDED '0' WHEN rst='1' (line 14) is a guarded statement. Therefore, q<='0' will occur when the guard expression is true and rst is '1'.

```
1 ------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ------------------------------
5 ENTITY dff IS
6 PORT ( d, clk, rst: IN STD_LOGIC;
7 q: OUT STD_LOGIC);
8 END dff;
9 ------------------------------
```

```
10 ARCHITECTURE dff OF dff IS
11 BEGIN
12 b1: BLOCK (clk'EVENT AND clk='1')
13 BEGIN
14 q <= GUARDED '0' WHEN rst='1' ELSE d;
15 END BLOCK b1;
16 END dff;
17 ------------------------------
```

6-Sequential Code

As mentioned in chapter 5, VHDL code is inherently concurrent .**PROCESSES, FUNCTIONS,** and **PROCEDURES** are the only sections of code that are executed **sequentially**.

One **important aspect of sequential** code is that it is not limited to sequential logic. Indeed, with it we can build sequential circuits as well as combinational circuits. Sequential code is also called behavioral code.

The statements discussed are all sequential, that is, allowed only inside PROCESSES, FUNCTIONS, or PROCEDURES. They are: **IF**, **WAIT**, **CASE,** and **LOOP.**

VARIABLES are also restricted to be used in sequential code only (that is, inside a PROCESS, FUNCTION, or PROCEDURE). Thus, contrary to a SIGNAL, a VARIABLE can never be global, so its value can not be passed out directly.

We will concentration PROCESSES here. FUNCTIONS and PROCEDURES

6.1 PROCESS

A PROCESS is a sequential section of VHDL code. It is characterized by the presence of IF, WAIT, CASE, or LOOP, and by a sensitivity list (except when WAIT is used).

A PROCESS must be installed in the main code, Itssyntax is shown below.

```
[label:] PROCESS (sensitivity list)
[VARIABLE name type [range] [:= initial_value;]]
BEGIN
(sequential code)
END PROCESS [label];
```

VARIABLES are optional. If used, they must be declared in the declarative part of the PROCESS (before the word BEGIN, as indicated in the syntax above). The initial value is not synthesizable, being only taken into consideration in simulations. The use of a label is also optional. Its purpose is to improve code readability. The label can be any word, except VHDL reserved words .

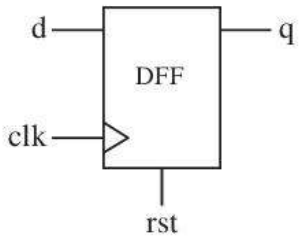


Figure 6.1
DFF with asynchronous reset of example 6.1.

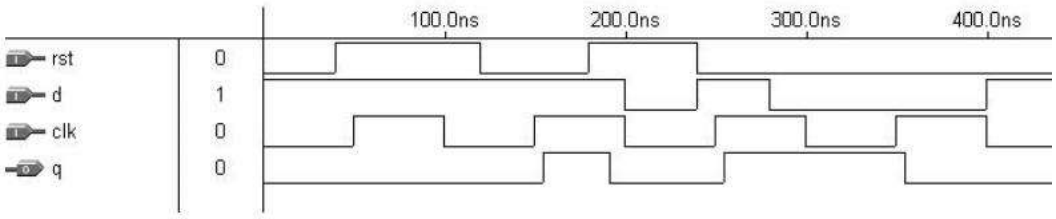


Figure 6.2
Simulation results of example 6.1.

To construct a synchronous circuit, monitoring a signal (clock, for example) is necessary. A common way of detecting a signal change is by means of the EVENT attribute For instance, if clk is a signal to be monitored, then clk'EVENT returns TRUE when a change on clk occurs (rising or falling edge).

Example 6.1: DFF with Asynchronous Reset #1

A D-type flip-flop (DFF, figure 6.1) is the most basic building block in **sequential** logic circuits. In it, **the output** must **copy** the **input** at either the positive or negative transition of the clock signal (rising or falling edge).

In the code presented below, we make use of the **IF** statement to design a DFF with asynchronous reset.

If **rst = '1',** then the output must be **q = '0'** (lines 14–15), **regardless** of the status of **clk** Otherwise, the output must copy the input (that is, **q = d**) at the positive edge of clk (lines 16–17).

The EVENT attribute is used in line 16 to detect a clock transition.

The PROCESS (lines 12–19) is run every time any of the signals that appear in its sensitivity list (clk and rst, line 12)changes.

Simulation results, confirming the functionality of the synthesized circuit, are presented in figure 6.2

```
1 ----------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------
5 ENTITY dff IS
6 PORT (d, clk, rst: IN STD_LOGIC;
7 q: OUT STD_LOGIC);
8 END dff;
9 ----------------------------------
10 ARCHITECTURE behavior OF dff IS
11 BEGIN
12 PROCESS (clk, rst)
13 BEGIN
14 IF (rst='1') THEN
15 q <= '0';
16 ELSIF (clk'EVENT AND clk='1') THEN
17 q <= d;
```

```
18 END IF;
19 END PROCESS;
20 END behavior;
21 -------------------------------------
```

6.2 Signals and Variables

VHDL provides two objects for dealing with non-static data values: SIGNAL and VARIABLE. It also provides means for establishing default (static) values: CONSTANT and GENERIC.

CONSTANT and SIGNAL can be global (that is, seen by the whole code), and can be used in either type of code, concurrent or sequential. A VARIABLE, on the other hand, is local, for it can only be used inside a piece of sequential code (that is, in a PROCESS, FUNCTION, or PROCEDURE) and its value can never be passed out directly.

Constant

CONSTANT serves to establish default values. Its syntax is shown below.

```
CONSTANT name : type := value;
```

Examples:

```
CONSTANT set_bit : BIT := '1';
CONSTANT datamemory : memory := (('0','0','0','0'),
                                 ('0','0','0','1'),
                                 ('0','0','1','1'));
```

A CONSTANT can be declared in a PACKAGE, ENTITY, or ARCHITECTURE. When declared in a package, it is truly global, for the package can be used by several entities. When declared in an entity (after PORT), it is global to all architectures that follow that entity. Finally, when declared in an architecture (in its declarative part), it is global only to that architecture's code. The most common places to find a CONSTANT declaration is in an ARCHITECTURE or in a PACKAGE.

Signal

SIGNAL serves to pass values in and out the circuit, as well as between its internal units. In other words, a signal represents circuit interconnects (wires). For instance, all PORTS of an ENTITY are signals by default. Its syntax is the following:

```
SIGNAL name : type [range] [:= initial_value];
```

Examples:

```
SIGNAL control: BIT := '0';
SIGNAL count: INTEGER RANGE 0 TO 100;
SIGNAL y: STD_LOGIC_VECTOR (7 DOWNTO 0);
```

The declaration of a SIGNAL can be made in the same places as the declaration of a CONSTANT.
A very important aspect of a SIGNAL, when used inside a section of sequential code (PROCESS, for example), is that its update is not immediate. In other words, its new value should not be expected to be ready before the conclusion of the corresponding
PROCESS, FUNCTION or PROCEDURE.
Recall that the assignment operator for a SIGNAL is ‘‘<=’’ (Ex.: count <= 35;).

Example 9.1: Count Ones #1 (not OK)
Design a circuit that counts the number of ‘1’s in a binary vector.

```
1  ---------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------
5  ENTITY count_ones IS
6     PORT ( din: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7            ones: OUT INTEGER RANGE 0 TO 8);
8  END count_ones;
9  ---------------------------------------
10 ARCHITECTURE not_ok OF count_ones IS
11    SIGNAL temp: INTEGER RANGE 0 TO 8;
12 BEGIN
13    PROCESS (din)
14    BEGIN
15       temp <= 0;
16       FOR i IN 0 TO 7 LOOP
17          IF (din(i)='1') THEN
18             temp <= temp + 1;
19          END IF;
20       END LOOP;
21       ones <= temp;
22    END PROCESS;
23 END not_ok;
24 ---------------------------------------
```

VARIABLE

Contrary to CONSTANT and SIGNAL, a VARIABLE represents only local information.
It can only be used inside a PROCESS, FUNCTION, or PROCEDURE (that is, in sequential code), and its value cannot be passed out directly. On the other hand, its update is immediate, so the new value can be promptly used in the next line of code.
To declare a VARIABLE, the following syntax should be used:

```
VARIABLE name : type [range] [:= init_value];
```

Examples:

```
VARIABLE control: BIT := '0';
VARIABLE count: INTEGER RANGE 0 TO 100;
VARIABLE y: STD_LOGIC_VECTOR (7 DOWNTO 0) := "10001000";
```

Since a VARIABLE can only be used in sequential code, its declaration can only be done in the declarative part of a PROCESS, FUNCTION, or PROCEDURE.
Recall that the assignment operator for a VARIABLE is ‘‘ := ’’ (Ex.: count:=35;).

Example 9.2: Count Ones #2 (OK)
Design a circuit that counts the number of ‘1’s in a binary vector

```
1  ----------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY count_ones IS
6     PORT ( din: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7            ones: OUT INTEGER RANGE 0 TO 8);
```

```
8  END count_ones;
9  ---------------------------------------
10 ARCHITECTURE ok OF count_ones IS
11 BEGIN
12    PROCESS (din)
13       VARIABLE temp: INTEGER RANGE 0 TO 8;
14    BEGIN
15       temp := 0;
16       FOR i IN 0 TO 7 LOOP
17          IF (din(i)='1') THEN
18             temp := temp + 1;
19          END IF;
20       END LOOP;
21       ones <= temp;
22    END PROCESS;
23 END ok;
24 ---------------------------------------
```

SIGNAL versus VARIABLE

choosing between a SIGNAL or a VARIABLE is not always straightforward. Their main differences are summarized in table 9.1.

Table 9.1
Comparison between SIGNAL and VARIABLE.

	SIGNAL	VARIABLE
Assignment	<=	:=
Utility	Represents circuit interconnects (wires)	Represents local information
Scope	Can be global (seen by entire code)	Local (visible only inside the corresponding PROCESS, FUNCTION, or PROCEDURE)
Behavior	Update is not immediate in sequential code (new value generally only available at the conclusion of the PROCESS, FUNCTION, or PROCEDURE)	Updated immediately (new value can be used in the next line of code)
Usage	In a PACKAGE, ENTITY, or ARCHITECTURE. In an ENTITY, all PORTS are SIGNALS by default	Only in sequential code, that is, in a PROCESS, FUNCTION, or PROCEDURE

Example 9.3: Bad versus Good Multiplexer

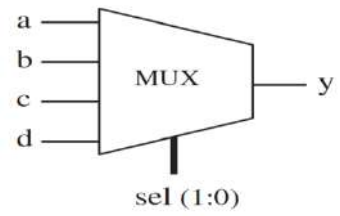


Figure 9.1: Multiplexer of example 9.3.

```
1  -- Solution 1: using a SIGNAL (not ok) --
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------
5  ENTITY mux IS
6     PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
7            y: OUT STD_LOGIC);
8  END mux;
9  ---------------------------------------
10 ARCHITECTURE not_ok OF mux IS
11    SIGNAL sel : INTEGER RANGE 0 TO 3;
12 BEGIN
13    PROCESS (a, b, c, d, s0, s1)
14    BEGIN
15       sel <= 0;
16       IF (s0='1') THEN sel <= sel + 1;
17       END IF;
18       IF (s1='1') THEN sel <= sel + 2;
19       END IF;
20       CASE sel IS
21          WHEN 0 => y<=a;
22          WHEN 1 => y<=b;
23          WHEN 2 => y<=c;
24          WHEN 3 => y<=d;
25       END CASE;
26    END PROCESS;
27 END not_ok;
28 ---------------------------------------
```

```
1  -- Solution 2: using a VARIABLE (ok) ----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------
5  ENTITY mux IS
6     PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
7            y: OUT STD_LOGIC);
8  END mux;
9  ---------------------------------------
```

```
10 ARCHITECTURE ok OF mux IS
11 BEGIN
12    PROCESS (a, b, c, d, s0, s1)
13       VARIABLE sel : INTEGER RANGE 0 TO 3;
14    BEGIN
15       sel := 0;
16       IF (s0='1') THEN sel := sel + 1;
17       END IF;
18       IF (s1='1') THEN sel := sel + 2;
19       END IF;
20       CASE sel IS
21          WHEN 0 => y<=a;
22          WHEN 1 => y<=b;
23          WHEN 2 => y<=c;
24          WHEN 3 => y<=d;
25       END CASE;
26    END PROCESS;
27 END ok;
28 ---------------------------------------
```

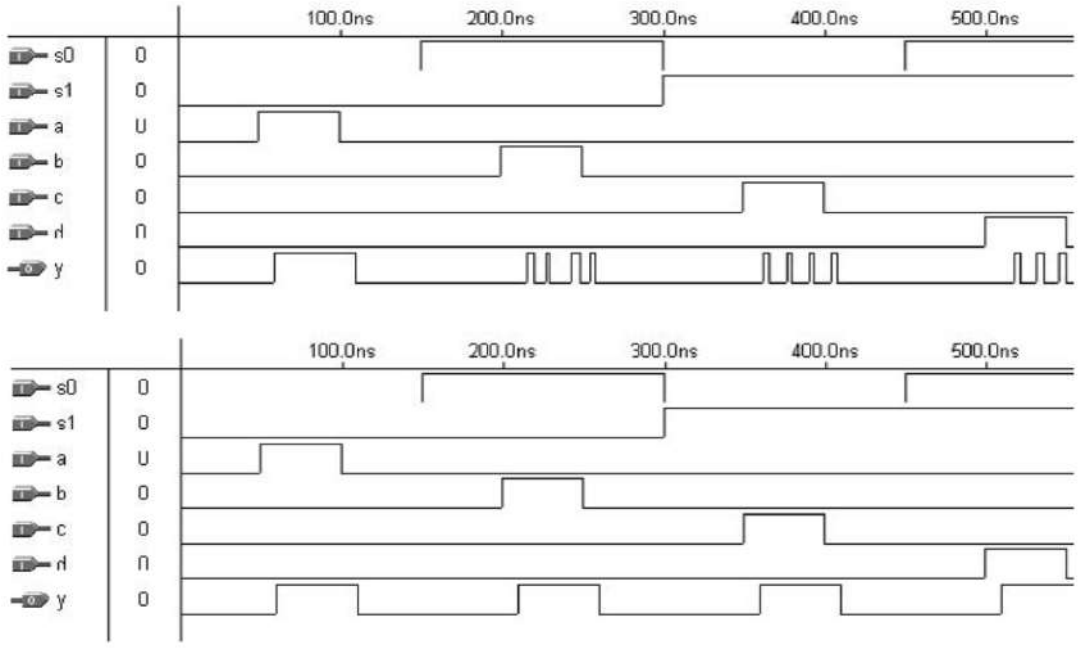


Figure 9.2: Simulation results of example 9.3.

Example 9.4: DFF with q and qbar #1

Implementing the DFF of figure 9.3.

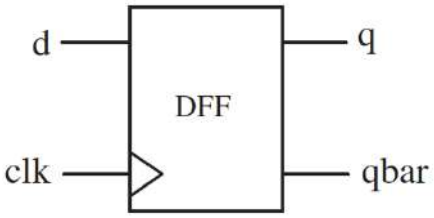


Figure 9.3: DFF of example 9.4.

```
1  ---- Solution 1: not OK ---------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------
5  ENTITY dff IS
6     PORT ( d, clk: IN STD_LOGIC;
7            q: BUFFER STD_LOGIC;
8            qbar: OUT STD_LOGIC);
9  END dff;
10 ---------------------------------------
11 ARCHITECTURE not_ok OF dff IS
12 BEGIN
13    PROCESS (clk)
14    BEGIN
15       IF (clk'EVENT AND clk='1') THEN
16          q <= d;
17          qbar <= NOT q;
18       END IF;
19    END PROCESS;
20 END not_ok;
21 ---------------------------------------
```

```
1  ---- Solution 2: OK -------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------
5  ENTITY dff IS
6     PORT ( d, clk: IN STD_LOGIC;
7            q: BUFFER STD_LOGIC;
8            qbar: OUT STD_LOGIC);
9  END dff;
10 ---------------------------------------
```

```
11 ARCHITECTURE ok OF dff IS
12 BEGIN
13     PROCESS (clk)
14     BEGIN
15        IF (clk'EVENT AND clk='1') THEN
16           q <= d;
17        END IF;
18     END PROCESS;
19     qbar <= NOT q;
20 END ok;
21 ---------------------------------------
```

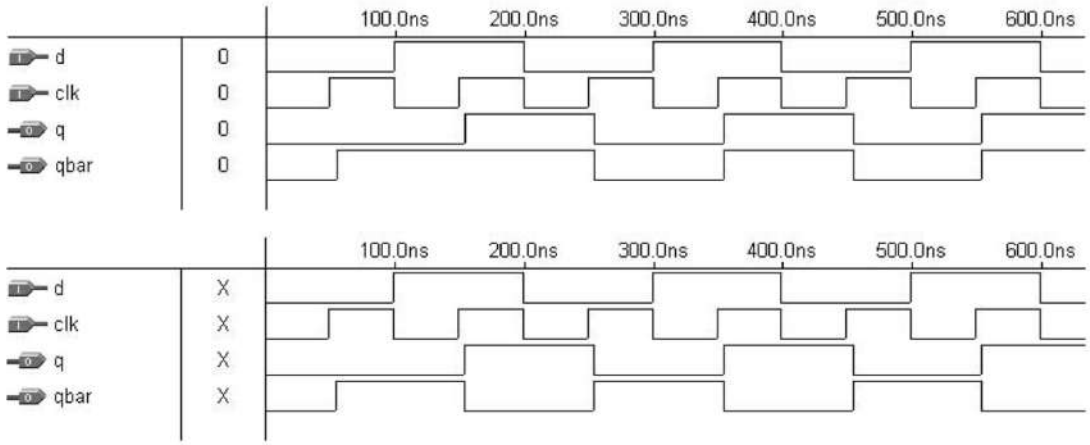


Figure 9.4: Simulation results of example 9.4.

Example 9.5: Frequency Divider

This example implements a circuit that divides the clock frequency by 6.

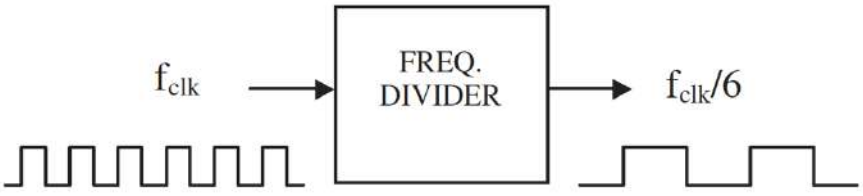


Figure 9.5: Frequency divider of example 9.5.

```
1  ----------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY freq_divider IS
6     PORT ( clk : IN STD_LOGIC;
7            out1, out2 : BUFFER STD_LOGIC);
8  END freq_divider;
9  ----------------------------------------
10 ARCHITECTURE example OF freq_divider IS
11    SIGNAL count1 : INTEGER RANGE 0 TO 7;
12 BEGIN
13    PROCESS (clk)
14       VARIABLE count2 : INTEGER RANGE 0 TO 7;
15    BEGIN
16       IF (clk'EVENT AND clk='1') THEN
17          count1 <= count1 + 1;
18          count2 := count2 + 1;
19          IF (count1 = ? ) THEN
20             out1 <= NOT out1;
21             count1 <= 0;
22          END IF;
23          IF (count2 = ? ) THEN
24             out2 <= NOT out2;
25             count2 := 0;
26          END IF;
27       END IF;
28    END PROCESS;
29 END example;
30 ----------------------------------------
```

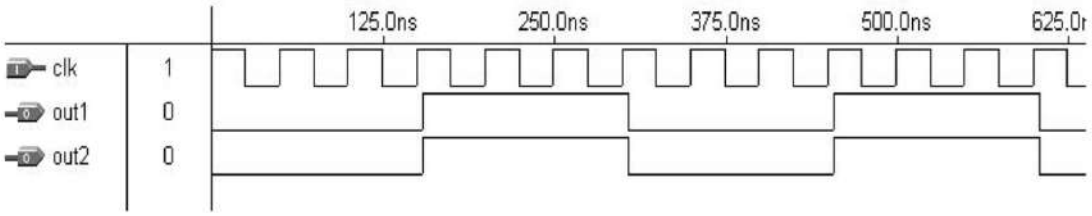


Figure 9.6: Simulation results of example 9.5.

6.3 IF statement

As mentioned earlier, **IF**, **WAIT**, **CASE**, and **LOOP** are the statements intended for

sequential code. Therefore, they can only be **used inside a PROCESS, FUNCTION, or PROCEDURE.**
The syntax of IF isshown below.

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

Example:

```
IF (x<y) THEN temp:="11111111";
ELSIF (x=y AND w='0') THEN temp:="11110000";
ELSE temp:=(OTHERS =>'0');
```

Example 6.2: One-digit Counter #1

The code below implements a progressive 1-digit decimal counter (0 →9 →0). A top-level diagram of the circuit is shown in figure 6.3. It contains a single-bit input (clk) and a 4-bit output (digit). The IF statement is used in this example. A variable, temp, was employed to create the four flip-flops necessary to store the 4-bit output signal. Simulation results, confirming the correct operation of the synthesized circuit, are shown in figure 6.4.

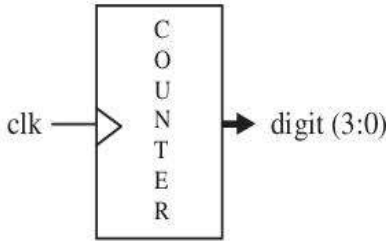


Figure 6.3
Counter of example 6.2.

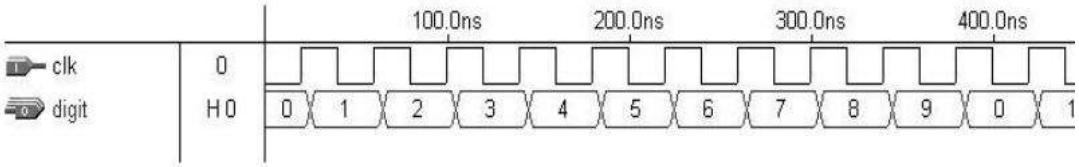


Figure 6.4
Simulation results of example 6.2.

```
1 ------------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ------------------------------------------
5 ENTITY counter IS
6 PORT (clk : IN STD_LOGIC;
7 digit : OUT INTEGER RANGE 0 TO 9);
```

```
8 END counter;
9 ----------------------------------------
10 ARCHITECTURE counter OF counter IS
11 BEGIN
12 count: PROCESS (clk)
13 VARIABLE temp : INTEGER RANGE 0 TO 10;
14 BEGIN
15 IF (clk' EVENT AND clk ='1') THEN
16 temp := temp + 1;
17 IF (temp=10) THEN temp := 0;
18 END IF;
19 END IF;
20 digit <= temp;
21 END PROCESS count;
22 END counter;
23 ----------------------------------------
```

temp in the physical circuit can be any 4-bit value. If such value is below 10 (see line 17), the circuit will count correctly from there. On the other hand, if the value is above 10, a number of clock cycles will be used until temp reaches full count (that is, 15, or ‘‘1111’’), being thus automatically reset to zero, from where the correct operation
then starts. The possibility of wasting a few clock cycles in the beginning is generally not a problem. Still, if one does want to avoid that, temp =10, in line 17, can be changed to temp =>10, but this will increase the hardware. However, if starting exactly from 0 is always necessary, then a reset input should be included (as in example 6.7).Notice in the code above that we increment temp and compare it to 10,with the
purpose of resetting temp once 10 is reached. This is a typical approach used in counters. Notice that 10 is a constant, so a comparator to a constant is inferred by the compiler, which is a relatively simple circuit to construct. However, if instead of a constant we were using a programmable parameter, then a full comparator would need to be implemented, which requires substantially more logic than a comparator to a constant. In this case, a better solution would be to load temp with such a parameter,and then decrement it, reloading temp when the 0 value is reached. In this case,our comparator would compare temp to 0 (a constant), thus avoiding the generationof a full comparator.

Example 6.3: Shift Register

Figure 6.5 shows a 4-bit shift register. The output bit (q) must be four positive clock edges behind the input bit (d). It also contains an asynchronous reset, which must force all flip-flop outputs to ‘0’ when asserted. In this example, the **IF** statement is again employed.

```
1 ----------------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -------------------------
```

```
5 ENTITY shiftreg IS
6 GENERIC (n: INTEGER := 4); -- # of stages
7 PORT (d, clk, rst: IN STD_LOGIC;
8 q: OUT STD_LOGIC);
9 END shiftreg;
10 -------------------------------------------------
11 ARCHITECTURE behavior OF shiftreg IS
12 SIGNAL internal: STD_LOGIC_VECTOR (n-1 DOWNTO 0);
13 BEGIN
14 PROCESS (clk, rst)
15 BEGIN
16 IF (rst='1') THEN
17 internal <= (OTHERS => '0');
18 ELSIF (clk'EVENT AND clk='1') THEN
19 internal <= d & internal(internal'LEFT DOWNTO 1);
20 END IF;
21 END PROCESS;
22 q <= internal(0);
23 END behavior;
24 -------------------------------------------------
```

clock edges behind d.

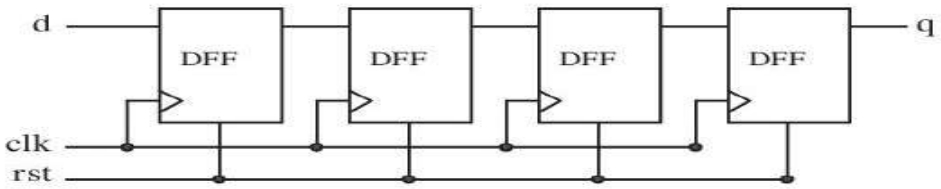


Figure 6.5
Shift register of example 6.3.

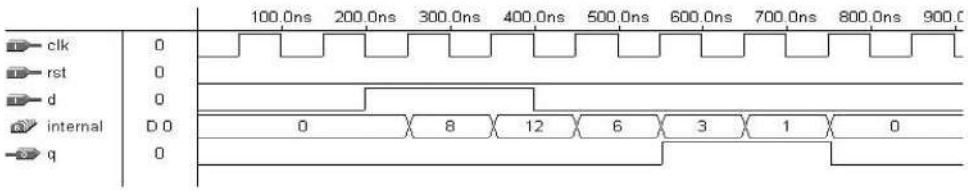


Figure 6.6
Simulation results of example 6.3.

used, the PROCESS cannot have a sensitivity list when **WAIT** is employed. Its syntax(there are three forms of WAIT) is shown below.

WAIT UNTIL signal_condition;

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

WAIT FOR time;

The WAIT UNTIL statement accepts only **one signal**, thus being more appropriate for synchronous code than asynchronous. Since the PROCESS has no sensitivity list in this case, WAIT UNTIL must be the first statement in the PROCESS.

Example: 8-bit register with synchronous reset.

```
PROCESS -- no sensitivity list
BEGIN
WAIT UNTIL (clk'EVENT AND clk='1');
IF (rst='1') THEN
output <= "00000000";
ELSIF (clk'EVENT AND clk='1') THEN
output <= input;
END IF;
END PROCESS;
```

WAIT ON, on the other hand, **accepts multiple signals**. The PROCESS is put on hold until any of the signals listed changes. In the example below, the PROCESS will continue execution whenever a change in rst or clk occurs.

Example: 8-bit register with asynchronous reset.

```
PROCESS
BEGIN
WAIT ON clk, rst;
IF (rst='1') THEN
output <= "00000000";
ELSIF (clk'EVENT AND clk='1') THEN
output <= input;
END IF;
END PROCESS;
```

Finally, WAIT FOR is intended for simulation only (waveform generation for testbenches). Example: **WAIT FOR** 5ns;

Example 6.4: DFF with Asynchronous Reset #2

The code below implements the same DFF of example 6.1 (figures 6.1 and 6.2). However, here WAIT ON is used instead of IF only.

```
1 ----------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------
5 ENTITY dff IS
6 PORT (d, clk, rst: IN STD_LOGIC;
7 q: OUT STD_LOGIC);
```

```
8 END dff;
9 ---------------------------------
10 ARCHITECTURE dff OF dff IS
11 BEGIN
12 PROCESS
13 BEGIN
14 WAIT ON rst, clk;
15 IF (rst='1') THEN
16 q <= '0';
17 ELSIF (clk'EVENT AND clk='1') THEN
18 q <= d;
19 END IF;
20 END PROCESS;
21 END dff;
22 ---------------------------------
```

Example 6.5: One-digit Counter #2

The code below implements the same progressive 1-digit decimal counter of example 6.2 (figures 6.3 and 6.4). However, **WAIT UNTIL** was used **instead** of IF only.

```
1 -----------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----------------------------------------
5 ENTITY counter IS
6 PORT (clk : IN STD_LOGIC;
7 digit : OUT INTEGER RANGE 0 TO 9);
8 END counter;
9 -----------------------------------------
10 ARCHITECTURE counter OF counter IS
11 BEGIN
12 PROCESS -- no sensitivity list
13 VARIABLE temp : INTEGER RANGE 0 TO 10;
14 BEGIN
15 WAIT UNTIL (clk'EVENT AND clk='1');
16 temp := temp + 1;
17 IF (temp=10) THEN temp := 0;
18 END IF;
19 digit <= temp;
20 END PROCESS;
21 END counter;
22 -----------------------------------------
```

6.5 CASE

CASE is another statement intended exclusively for sequential code (along with IF, LOOP, and WAIT). Its syntax is shown below.

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

Example:

```
CASE control IS
WHEN "00" => x<=a; y<=b;
WHEN "01" => x<=b; y<=c;
WHEN OTHERS => x<="0000"; y<="ZZZZ";
END CASE;
```

The CASE statement (sequential) is very similar to WHEN (combinational). should be used when no action is to take place. For example, WHEN OTHERS => NULL;. However, CASE allows multiple assignments for each test condition (as shown in the example above), while WHEN allows only one.
Like in the case of WHEN (section 5.3), here too ‘‘WHEN value’’ can take up three forms:
1-WHEN value -- single value
2-WHEN value1 to value2 -- range, for enumerated data types only
3-WHEN value1 | value2 |... -- value1 or value2 or .
..

Example 6.6: DFF with Asynchronous Reset #3

The code below implements the same DFF of example 6.1 (figures 6.1 and 6.2). However, here CASE was used instead of IF only. Notice that a few unnecessary declarations were intentionally included in the code to illustrate their usage.

```
1 ------------------------------------------
2 LIBRARY ieee; -- Unnecessary declaration,
3 -- because
4 USE ieee.std_logic_1164.all; -- BIT was used instead of
5 -- STD_LOGIC
6 ------------------------------------------
7 ENTITY dff IS
8 PORT (d, clk, rst: IN BIT;
9 q: OUT BIT);
10 END dff;
11 ------------------------------------------
12 ARCHITECTURE dff3 OF dff IS
13 BEGIN
14 PROCESS (clk, rst)
```

```
15 BEGIN
16 CASE rst IS
17 WHEN '1' => q<='0';
18 WHEN '0' =>
19 IF (clk'EVENT AND clk='1') THEN
20 q <= d;
21 END IF;
22 WHEN OTHERS => NULL; -- Unnecessary, rst is of type
23 -- BIT
24 END CASE;
25 END PROCESS;
26 END dff3;
27 ------------------------------------------
```

Example 6.7: Two-digit Counter with SSD Output

The code below implements a progressive 2-digit decimal counter (0 →99 →0), with external asynchronous reset plus binary-coded decimal (**BCD**) to seven-segment display
(SSD) conversion. Diagrams of the circuit and SSD are shown in figure 6.7. The CASE statement (lines 31–56) was employed to determine the output signals that will feed the SSDs. Notice that we have chosen the following connection between the circuit and the SSD: **xabcdefg** (that is, the **MSB** feeds the decimal point, while the **LSB** feeds segment g).
As can be seen, this circuit is a straight extension of that presented in example 6.2, with the differences that now two digits are necessary rather than one, and that the outputs must be connected to SSD displays. The operation of the circuit can be verified in the simulation results of figure 6.8.

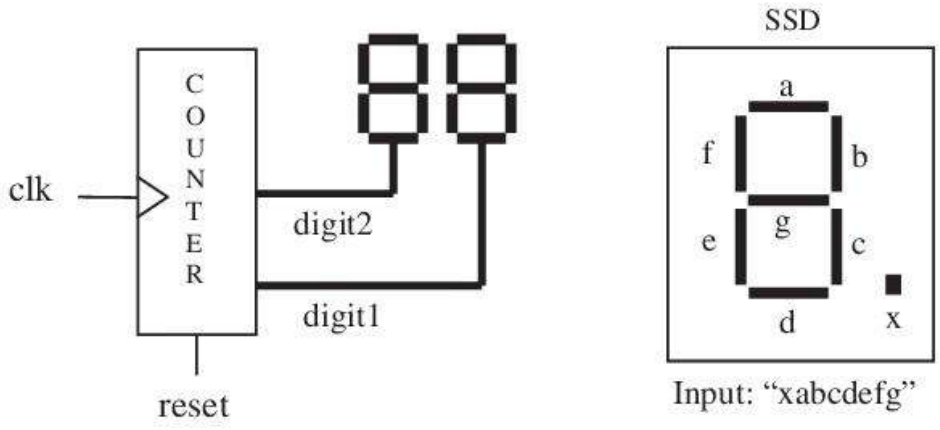


Figure 6.7
2-digit counter of example 6.7.

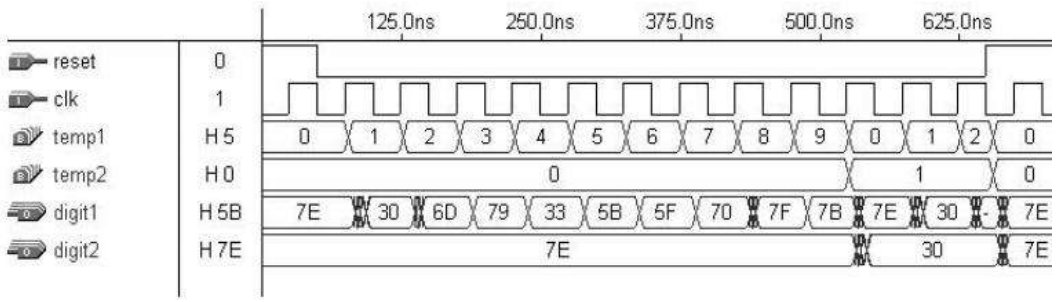


Figure 6.8
Simulation results of example 6.7.

```
1 ------------------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ------------------------------------------------
5 ENTITY counter IS
6 PORT (clk, reset : IN STD_LOGIC;
7 digit1, digit2 : OUT STD_LOGIC_VECTOR (6 DOWNTO 0));
8 END counter;
9 ------------------------------------------------
10 ARCHITECTURE counter OF counter IS
11 BEGIN
12 PROCESS(clk, reset)
13 VARIABLE temp1: INTEGER RANGE 0 TO 10;
14 VARIABLE temp2: INTEGER RANGE 0 TO 10;
15 BEGIN
16 ---- counter: ----------------------
17 IF (reset='1') THEN
18 temp1 := 0;
19 temp2 := 0;
20 ELSIF (clk'EVENT AND clk='1') THEN
21 temp1 := temp1 + 1;
22 IF (temp1=10) THEN
23 temp1 := 0;
24 temp2 := temp2 + 1;
25 IF (temp2=10) THEN
26 temp2 := 0;
27 END IF;
28 END IF;
29 END IF;

30 ---- BCD to SSD conversion: --------
31 CASE temp1 IS
32 WHEN 0 => digit1 <= "1111110"; --7E
33 WHEN 1 => digit1 <= "0110000"; --30
34 WHEN 2 => digit1 <= "1101101"; --6D
```

```
35 WHEN 3 => digit1 <= "1111001"; --79
36 WHEN 4 => digit1 <= "0110011"; --33
37 WHEN 5 => digit1 <= "1011011"; --5B
38 WHEN 6 => digit1 <= "1011111"; --5F
39 WHEN 7 => digit1 <= "1110000"; --70
40 WHEN 8 => digit1 <= "1111111"; --7F
41 WHEN 9 => digit1 <= "1111011"; --7B
42 WHEN OTHERS => NULL;
43 END CASE;
44 CASE temp2 IS
45 WHEN 0 => digit2 <= "1111110"; --7E
46 WHEN 1 => digit2 <= "0110000"; --30
47 WHEN 2 => digit2 <= "1101101"; --6D
48 WHEN 3 => digit2 <= "1111001"; --79
49 WHEN 4 => digit2 <= "0110011"; --33
50 WHEN 5 => digit2 <= "1011011"; --5B
51 WHEN 6 => digit2 <= "1011111"; --5F
52 WHEN 7 => digit2 <= "1110000"; --70
53 WHEN 8 => digit2 <= "1111111"; --7F
54 WHEN 9 => digit2 <= "1111011"; --7B
55 WHEN OTHERS => NULL;
56 END CASE;
57 END PROCESS;
58 END counter;
59 ------------------------------------------------
```

Comment: Notice above that the same routine was repeated twice (using CASE statements).

6.6 LOOP

LOOP is useful when a piece of code must be instantiated several times. Like IF, WAIT, and CASE,*LOOP is intended exclusively for sequential code,* so it too can only be used inside a PROCESS, FUNCTION, or PROCEDURE.

Example of WHILE / LOOP: In this example, LOOP will keep repeating while i <10.

```
WHILE (i < 10) LOOP
WAIT UNTIL clk'EVENT AND clk='1';
(other statements)
END LOOP;
```

Example with EXIT:

```
FOR i IN data'RANGE LOOP
CASE data(i) IS
WHEN '0' => count:=count+1;
WHEN OTHERS => EXIT;
END CASE;
END LOOP;
```

Example with NEXT:

when i =skip.
FOR i IN 0 TO 15 LOOP
NEXT WHEN i=skip; -- jumps to next iteration
(...)
END LOOP;

Example 6.8: Carry Ripple Adder

Figure 6.9 shows an 8-bit unsigned carry ripple adder. The top-level diagram shows the inputs and outputs of the circuit: **a** and **b** are the input vectors to be added, **cin** is the **carry-in**bit, **s** is the **sum** vector, and **cout** is the **carry-out** bit. top diagram shows how the carry bits propagate (ripple).

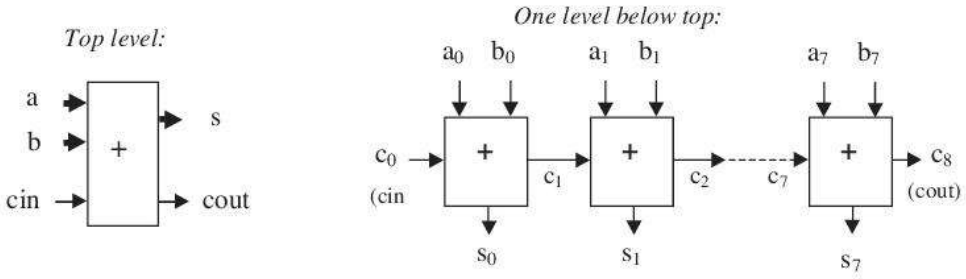


Figure 6.9
8-bit carry ripple adder of example 6.8

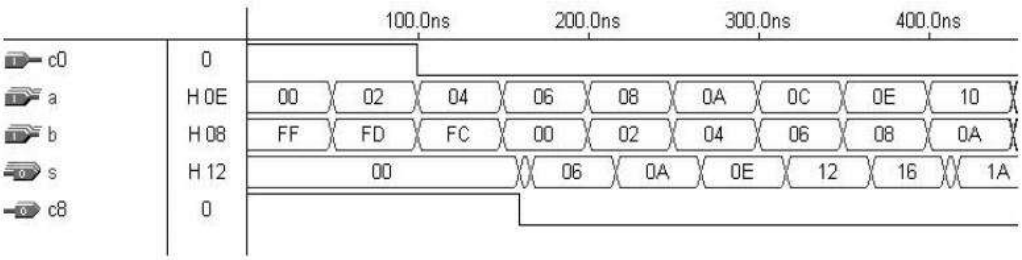


Figure 6.10
Simulation results of example 6.8.

Each section of the latter diagram is a full-adder unit Thus its outputs can be computed by means of:

sj =aj XOR bj XOR cj

cj1 =(aj AND bj) OR (aj AND cj) OR (bj AND cj)

Two solutions are presented, being one generic and the other specific for 8-bit numbers.

Simulation results from either solution are shown in figure 6.10.

1 ----- **Solution 1: Generic, with VECTORS**--------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------

```
5 ENTITY adder IS
6 GENERIC (length : INTEGER := 8);
7 PORT ( a, b: IN STD_LOGIC_VECTOR (length-1 DOWNTO 0);
8 cin: IN STD_LOGIC;
9 s: OUT STD_LOGIC_VECTOR (length-1 DOWNTO 0);
10 cout: OUT STD_LOGIC);
11 END adder;
12 -----------------------------------------------
13 ARCHITECTURE adder OF adder IS
14 BEGIN
15 PROCESS (a, b, cin)
16 VARIABLE carry : STD_LOGIC_VECTOR (length DOWNTO 0);
17 BEGIN
18 carry(0) := cin;
19 FOR i IN 0 TO length-1 LOOP
20 s(i) <= a(i) XOR b(i) XOR carry(i);
21 carry(i+1) := (a(i) AND b(i)) OR (a(i) AND
22 carry(i)) OR (b(i) AND carry(i));
23 END LOOP;
24 cout <= carry(length);
25 END PROCESS;
26 END adder;
27 -----------------------------------------------
```

1 ---- **Solution 2: non-generic, with INTEGERS**----

```
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----------------------------------------------
5 ENTITY adder IS
6 PORT ( a, b: IN INTEGER RANGE 0 TO 255;
7 c0: IN STD_LOGIC;
8 s: OUT INTEGER RANGE 0 TO 255;
9 c8: OUT STD_LOGIC);
10 END adder;
11 -----------------------------------------------
12 ARCHITECTURE adder OF adder IS
13 BEGIN
14 PROCESS (a, b, c0)
15 VARIABLE temp : INTEGER RANGE 0 TO 511;
16 BEGIN
17 IF (c0='1') THEN temp:=1;
18 ELSE temp:=0;
19 END IF;
20 temp := a + b + temp;
21 IF (temp > 255) THEN
22 c8 <= '1';
23 temp := temp---256;
24 ELSE c8 <= '0';
25 END IF;
```

```
26 s <= temp;
27 END PROCESS;
28 END adder;
29 -------------------------------------------
```

Example 6.9: Simple Barrel Shifter

Figure 6.11 shows the diagram of a very simple barrel shifter. In this case, the circuit must **shift** the input vector (of size 8) either 0 or 1 position to the **left**. When actually shifted (shift =1), the LSB bit must be filled with '0' (shown in the bottom left corner of the diagram). If shift =0, then outp =inp; if shift =1, then outp(0) ='0' and outp(i) =inp(i -1), for $1 \leq i \leq 7$.
A complete VHDL code is presented below, which illustrates the use of FOR/ LOOP. Simulation results appear in figure 6.12.

```
1 -------------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -------------------------------------------
5 ENTITY barrel IS
6 GENERIC (n: INTEGER := 8);
7 PORT ( inp: IN STD_LOGIC_VECTOR (n-1 DOWNTO 0);
8 shift: IN INTEGER RANGE 0 TO 1;
9 outp: OUT STD_LOGIC_VECTOR (n-1 DOWNTO 0));
10 END barrel;
11 -------------------------------------------
12 ARCHITECTURE RTL OF barrel IS
13 BEGIN
14 PROCESS (inp, shift)
15 BEGIN
16 IF (shift=0) THEN
17 outp <= inp;
18 ELSE
19 outp(0) <= '0';
20 FOR i IN 1 TO inp'HIGH LOOP
21 outp(i) <= inp(i-1);
22 END LOOP;
23 END IF;
24 END PROCESS;
25 END RTL;
26 -------------------------------------------
```

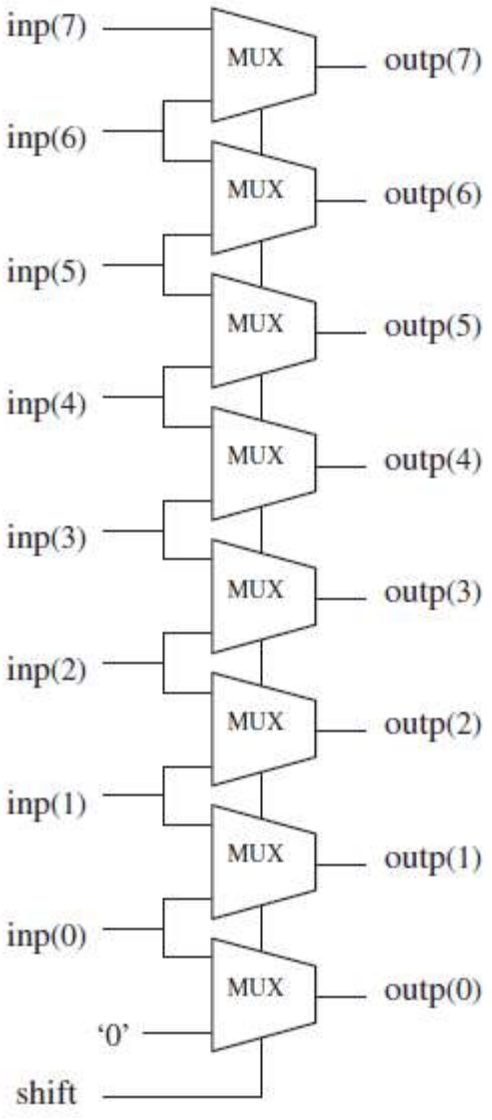


Figure 6-11- simple barrel shifter

Example 6.10: Leading Zeros

The design below counts the number of leading zeros in a binary vector, starting from the left end. The solution illustrates the use of **LOOP / EXIT**. Recall that EXIT implies not a escape from the current iteration of the loop, but rather a definite exit from it (that is, even if i is still within the specified range, the LOOP statement will be considered as concluded). In this example, the loop will end as soon as a '1' is found in the data vector. Therefore, it is appropriate for counting the number of zeros that precedes the first one.

```
1 ----------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------------
5 ENTITY Leading Zeros IS
6 PORT ( data: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
```

```
7 zeros: OUT INTEGER RANGE 0 TO 8);
8 END LeadingZeros;
9 ----------------------------------------
10 ARCHITECTURE behavior OF Leading  Zeros IS
11 BEGIN
12 PROCESS (data)
13 VARIABLE count: INTEGER RANGE 0 TO 8;
14 BEGIN
15 count := 0;
16 FOR i IN data 'RANGE LOOP
17 CASE data(i) IS
18 WHEN '0' => count := count + 1;
19 WHEN OTHERS => EXIT;
20 END CASE;
21 END LOOP;
22 zeros <= count;
23 END PROCESS;
24 END behavior;
25 ----------------------------------------
```

Simulation results, verifying the functionality of the circuit, are shown in figure 6.13. With data =‘‘00000000’’ (decimal 0), eight zeros are detected; when data = ‘‘00000001’’ (decimal 1), seven zeros are encountered; etc.

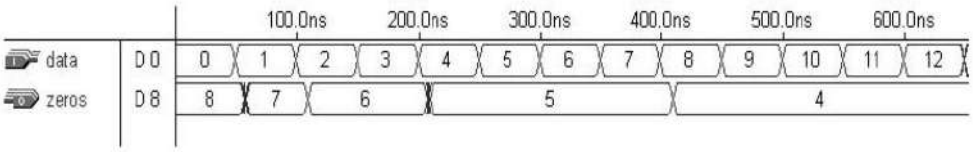


Figure 6.13
Simulation results of example 6.10.

Example 6.11: RAM

Below is another example using **sequential code**, particularly the IF statement. We show the implementation of a RAM (random access memory).
As can be seen in figure 6.14(a), the circuit has a data input bus (data_in), a data output bus (data _out), an address bus (addr), plus clock (clk) and write enable

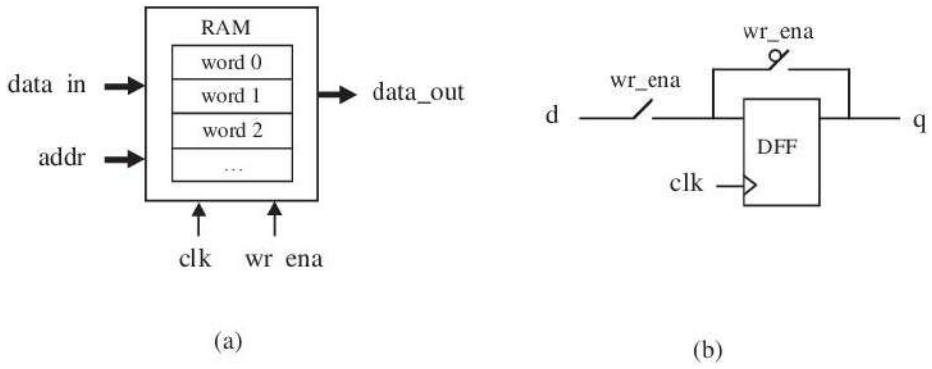


Figure 6.14
RAM circuit of example 6.11.

wr_ena) pins. When wr_ena is asserted, at the next rising edge of clk the vector present at data_in must be stored in the position specified by addr. The output, data_out, on the other hand, must constantly display the data selected by addr.
From the register point-of-view, the circuit can be summarized as in figure 6.14(b). When wr_ena is low, q is connected to the input of the flip-flop, and terminal d is open, so no new data will be written into the memory. However, when wr_ena is turned high, d is connected to the input of the register, so at the next rising edge of clk d will overwrite its previous value.
A VHDL code that implements the circuit of figure 6.14 is shown below. The capacity chosen for the RAM is 16 words of length 8 bits each. Notice that the code is totally generic.
Note: Other memory implementations will be presented in section 9.10 of chapter 9.

```
1 ------------------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ------------------------------------------------
5 ENTITY ram IS
6 GENERIC ( bits: INTEGER := 8; -- # of bits per word
7 words: INTEGER := 16); -- # of words in the memory
8 PORT ( wr_ena, clk: IN STD_LOGIC;
9 addr: IN INTEGER RANGE 0 TO words-1;
10 data_in: IN STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
11 data_out: OUT STD_LOGIC_VECTOR (bits-1 DOWNTO 0));
12 END ram;
13 ------------------------------------------------
14 ARCHITECTURE ram OF ram IS
15 TYPE vector_array IS ARRAY (0 TO words-1) OF
16 STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
17 SIGNAL memory: vector_array;
18 BEGIN
19 PROCESS (clk, wr_ena)
20 BEGIN
21 IF (wr_ena='1') THEN
22 IF (clk'EVENT AND clk='1') THEN
```

```
23 memory(addr) <= data_in;
24 END IF;
25 END IF;
26 END PROCESS;
27 data_out <= memory(addr);
28 END ram;
29 ------------------------------------------------
```

Simulation results from the circuit synthesized with the code above are shown in figure 6.15.

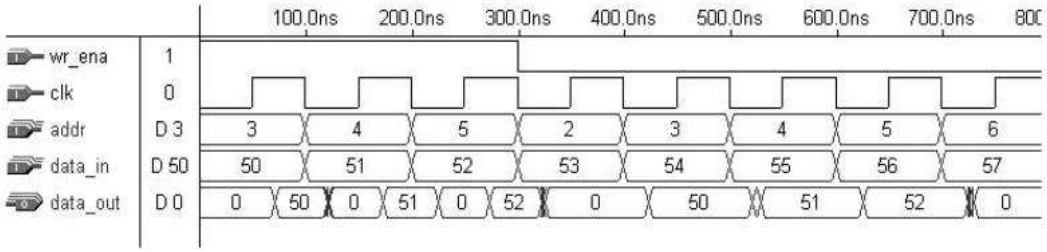


Figure 6.15
Simulation results of example 6.11.

6.10 Using Sequential Code to Design Combinational Circuits

We have already seen that sequential code can be used to implement either sequential or combinational circuits.
In order to satisfy the criteria above, the following rules should be observed:

Rule 1: Make sure that all input signals used (read) in the PROCESS appear in its sensitivity list.

Rule 2: Make sure that all combinations of the input/output signals are included in the code;

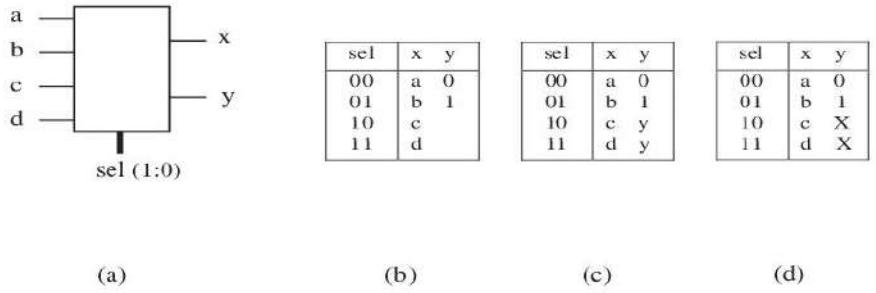


Figure 6.16
Circuit of example 6.12: (a) top-level diagram, (b) specifications provided, (c) implemented truth-table, and (d) the right approach.

With respect to rule 2, however, the consequences can be more serious because incomplete specifications of the output signals might cause the synthesizer to infer latches in order to hold their previous values. This fact is illustrated in the example below.

Example 6.12: Bad Combinational Design

Let us consider the circuit of figure 6.16, for which the following specifications have been provided: x should behave as a multiplexer; that is, should be equal to the input selected by sel; y, on the other hand, should be equal to '0' when sel ="00", or '1' if sel ="01". These specifications are summarized in the truth-table of figure 6.16(b). Notice that this is a combinational circuit. However, the specifications provided for y are incomplete, as can be observed in the truth-table of figure 6.16(b). Using just these specifications, the code could be the following:

```
1 ------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ------------------------------------
5 ENTITY example IS
6 PORT (a, b, c, d: IN STD_LOGIC;
7 sel: IN INTEGER RANGE 0 TO 3;
8 x, y: OUT STD_LOGIC);
9 END example;
10 ------------------------------------
11 ARCHITECTURE example OF example IS
12 BEGIN
13 PROCESS (a, b, c, d, sel)
14 BEGIN
15 IF (sel=0) THEN
16 x<=a;
17 y<='0';
18 ELSIF (sel=1) THEN
19 x<=b;
20 y<='1';
21 ELSIF (sel=2) THEN
22 x<=c;
23 ELSE
24 x<=d;
25 END IF;
26 END PROCESS;
27 END example;
28 ------------------------------------
```

After compiling this code, the report files show that no flip-flops were inferred (as expected). However, when we look at the simulation results (figure 6.17), we notice something peculiar about y. Observe that, for the same value of the input (sel =3 ="11"), two different results are obtained for y (when sel =3 is preceded by sel =0, y ='0' results, while y ='1' is obtained when sel =3 is preceded by sel =1). This signifies that some sort of memory was indeed implemented by the compiler.

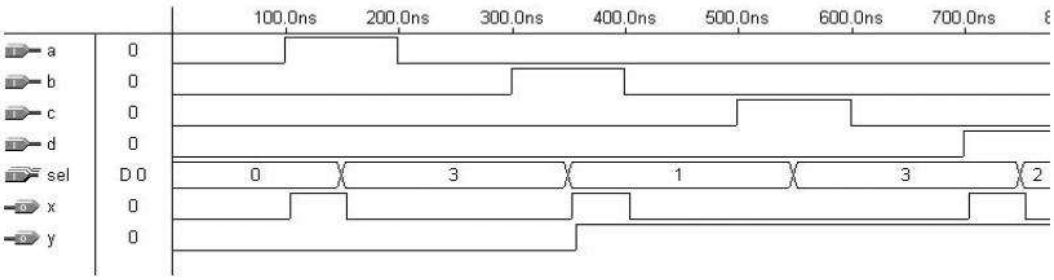


Figure 6.17
Simulation results of example 6.12.

8-State Machines

Finite **s**tate **m**achines (**FSM**) constitute a special modeling technique **for sequential logic circuits**. Such a model can be very helpful in the design of certain types of systems, particularly those whose tasks form a well-defined sequence (digital controllers,

8.1 Introduction

Figure 8.1 shows the block diagram of a **single-phase state machine**. the ***lower section*** contains the ***sequential logic (flip-flops),***the ***upper section*** contains the ***combinational logic.***

The **combinational (upper) section** has two inputs, being one pr***_state (present state)*** and the other the external input proper. It has also two outputs, ***nx_state (next state)*** and the external output proper.

The **sequential (lower) section** has **three inputs (clock, reset, and nx_state)**, and ***one output (pr_state).*** Since all flip-flops are in this part of the system, clock and reset must be connected to it.

If the output of the machine depends not only on the present state but also on the current input, then it is **called a Mealy machine.** Otherwise, **if it depends only on the current state,** it is **called a Moore machine**.

The separation of the circuit into two sections (figure 8.1) allows the part, being sequential, will require a PROCESS, while the upper part, being combinational,

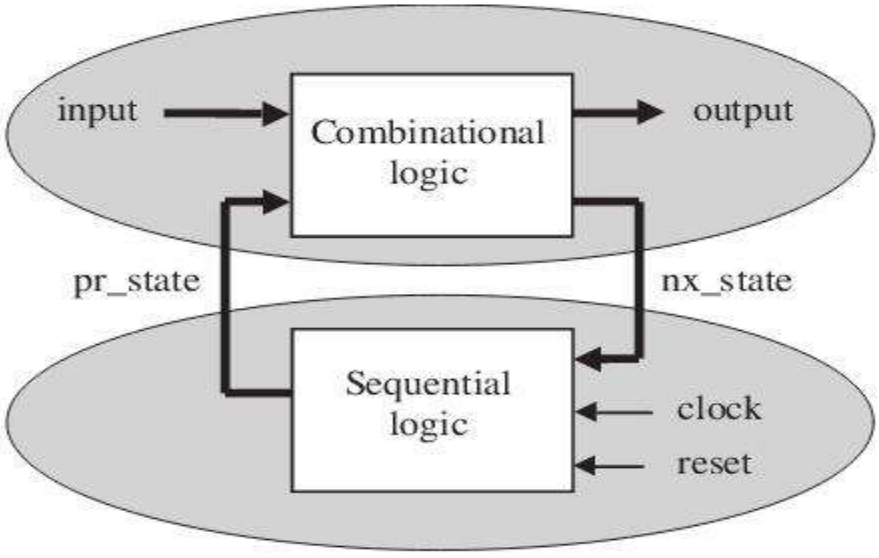


Figure 8.1
Mealy (Moore) state machine diagram.

8.2 Design Style #1

Several approaches can be conceived to design a FSM. We will describe in detail one style that is well structured and easily applicable. In it, **the design of the lower section** of the state machine (figure 8.1) is completely separated from that of the upper section.
storage perspective, in order to further understand and refine its construction, which will lead to design style #2.

Design of the Lower (Sequential) Section

In figure 8.1, the **flip-flops** are in the lower section, so **clock** and **reset** are connected to it. The other lower section's **input** is **nx_state** (next state), while **pr_state** (present state) is its only **output.** Being the circuit of the lower section sequential, a PROCESS is required, in which any of the sequential statements (IF, WAIT, CASE, or LOOP,) can be employed. A typical design template for the lower section is the following:

```
PROCESS (reset, clock)
BEGIN
IF (reset='1') THEN
pr_state <= state0;
ELSIF (clock'EVENT AND clock='1') THEN
pr_state <= nx_state;
END IF;
END PROCESS;
```

The code shown above is very simple. It consists of an **asynchronous reset**, which Determines the initial state of the system (state0), followed by the synchronous storage of nx_state (at the positive transition of clock),which will produce pr_state at the

design of the lower section is basically standard.

Another advantage of this design style is that the **number of registers is minimum**. we know that the number of flip-flops inferred from the code above is simply equal to the number of bits needed to encode all states of the FSM (because the only signal to which a value is assigned at the transition of another signal is pr_state). Therefore, if the default (binary) encoding style (section8.4) is used, just [log2n]flip-flops will then be needed, where n is the number of states.

Design of the *Upper* (Combinational) Section

In figure 8.1, the upper section is fully combinational, so its code does not need to be sequential; concurrent code can be used as well.

```
PROCESS (input, pr_state)
BEGIN
CASE pr_state IS
WHEN state0 =>
IF (input = ...) THEN
output <= <value>;
nx_state <= state1;
ELSE ...
END IF;
WHEN state1 =>
```

```
IF (input = ...) THEN
output <= <value>;
nx_state <= state2;
ELSE ...
END IF;
WHEN state2 =>
IF (input = ...) THEN
output <= <value>;
nx_state <= state2;
ELSE ...
END IF;
...
END CASE;
END PROCESS;
```

As can be seen, **this code is also very simple**, and does two things: (a) it assigns the output value and (b) it establishes the next state. the design of combinational circuits using sequential statements ,for all input signals are present in the sensitivity list and all input/output combinations are specified. Finally, observe that no signal assignment is made at the transition of another signal, so no flip-flops will be inferred

State Machine Template for Design Style #1

A complete template is shown below. Notice that, in addition to the two processes presented above, it also contains a user-defined enumerated data type (here called state), which lists all possible states of the machine.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------------
ENTITY <entity_name> IS
PORT ( input: IN <data_type>;
reset, clock: IN STD_LOGIC;
output: OUT <data_type>);
END <entity_name>;
---------------------------------------------------
ARCHITECTURE <arch_name> OF <entity_name> IS
TYPE state IS (state0, state1, state2, state3, ...);
SIGNAL pr_state, nx_state: state;
BEGIN
---------- Lower section: ----------
PROCESS (reset, clock)
BEGIN
IF (reset='1') THEN
pr_state <= state0;
ELSIF (clock'EVENT AND clock='1') THEN
pr_state <= nx_state;
END IF;
END PROCESS;
---------- Upper section: ----------
```

```
PROCESS (input, pr_state)
BEGIN
CASE pr_state IS
WHEN state0 =>
IF (input = ...) THEN
output <= <value>;
nx_state <= state1;
ELSE ...
END IF;
WHEN state1 =>
IF (input = ...) THEN
output <= <value>;
nx_state <= state2;
ELSE ...
END IF;
WHEN state2 =>
IF (input = ...) THEN
output <= <value>;
nx_state <= state3;
ELSE ...
END IF;
...
END CASE;
END PROCESS;
END <arch_name>;
```

Example 8.1: BCD Counter

A **counter** is an example of **Moore machine**, for the ***output depends*** only on the stored (***present***) ***state***. As a simple registered circuit and as a sequencer, it can be easily implemented in either approach: The problem with the latter is that when the number of states is large it becomes cumbersome to enumerate them all, a problem easily avoided using the LOOP statement in a conventional approach.
The state diagram of a **0-to-9 circular counter** is shown in figure 8.2. The states were called **zero, one, . . . , nine,** each name corresponding to the decimal value of the output.
A VHDL code, directly resembling the design style #1 template, is presented below. An enumerated data type (state) appears in lines 11–12. The design of the lower (clocked) section is presented in lines 16–23, and that of the upper (combinational) section, in lines 25–59. In this example, the number of registers is $\lceil \log_2 10 \rceil = 4$.
Simulation results are shown in figure 8.3. As can be seen, the output (count) grows from 0 to 9, and then restarts from 0 again.

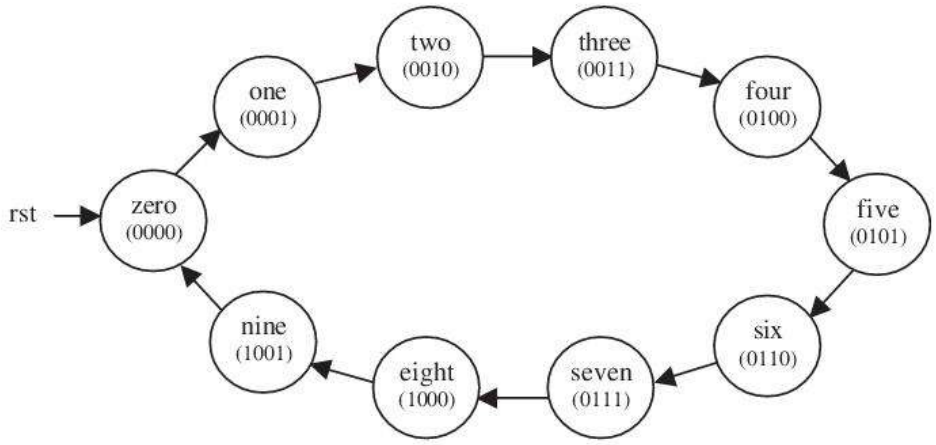


Figure 8.2
States diagram of example 8.1.

```
1 ------------------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ------------------------------------------------
5 ENTITY counter IS
6 PORT ( clk, rst: IN STD_LOGIC;
7 count: OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
8 END counter;
9 ------------------------------------------------
10 ARCHITECTURE state_machine OF counter IS
11 TYPE state IS (zero, one, two, three, four,
12 five, six, seven, eight, nine);
13 SIGNAL pr_state, nx_state: state;
14 BEGIN
15 ---------- Lower section: ----------
16 PROCESS (rst, clk)
17 BEGIN
18 IF (rst='1') THEN
19 pr_state <= zero;
20 ELSIF (clk'EVENT AND clk='1') THEN
21 pr_state <= nx_state;
22 END IF;
23 END PROCESS;
24 ---------- Upper section: ----------
25 PROCESS (pr_state)
26 BEGIN
27 CASE pr_state IS
28 WHEN zero =>
```

```
29 count <= "0000";
30 nx_state <= one;
31 WHEN one =>
32 count <= "0001";
33 nx_state <= two;
34 WHEN two =>
35 count <= "0010";
36 nx_state <= three;
37 WHEN three =>
38 count <= "0011";
39 nx_state <= four;
40 WHEN four =>
41 count <= "0100";
42 nx_state <= five;
43 WHEN five =>
44 count <= "0101";
45 nx_state <= six;
46 WHEN six =>
47 count <= "0110";
48 nx_state <= seven;
49 WHEN seven =>
50 count <= "0111";
51 nx_state <= eight;
52 WHEN eight =>
53 count <= "1000";
54 nx_state <= nine;
55 WHEN nine =>
56 count <= "1001";
57 nx_state <= zero;
58 END CASE;
59 END PROCESS;
60 END state_machine;
61 ------------------------------------------------
```

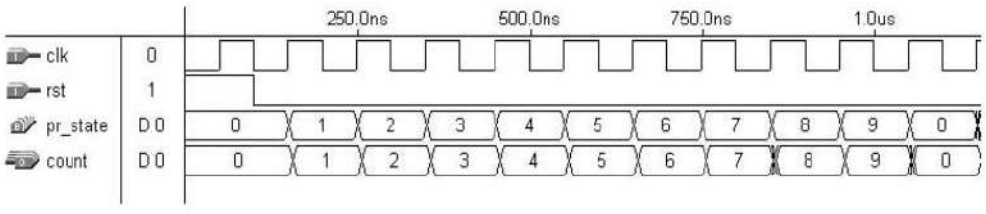


Figure 8.3
Simulation results of example 8.1.

Example 8.2: Simple FSM #1

Figure 8.4 shows the states diagram of a very simple FSM. The system has two states (state A and state B), and must change from one to the other every time d ='1' is received. The desired output is x =a when the machine is in state A, or x =b when in state B. The initial (reset) state is state A.
A VHDL code for this circuit, employing design style #1, is shown below.

```
1 ------------------------------------------
2 ENTITY simple_fsm IS
3 PORT ( a, b, d, clk, rst: IN BIT;
4 x: OUT BIT);
5 END simple_fsm;
6 ------------------------------------------
7 ARCHITECTURE simple_fsm OF simple_fsm IS
8 TYPEstate IS (stateA, stateB);
9 SIGNAL pr_state, nx_state: state;
10 BEGIN
11 ----- Lower section: ----------
12 PROCESS (rst, clk)
13 BEGIN
14 IF (rst='1') THEN
15 pr_state <= stateA;
16 ELSIF (clk'EVENT AND clk='1') THEN
17 pr_state <= nx_state;
18 END IF;
19 END PROCESS;

20 ---------- Upper section: ----------
21 PROCESS (a, b, d, pr_state)
22 BEGIN
23 CASE pr_state IS
24 WHEN stateA =>
25 x <= a;
26 IF (d='1') THEN nx_state <= stateB;
27 ELSE nx_state <= stateA;
28 END IF;
29 WHEN stateB =>
30 x <= b;
31 IF (d='1') THEN nx_state <= stateA;
32 ELSE nx_state <= stateB;
33 END IF;
34 END CASE;
35 END PROCESS;
36 END simple_fsm;
37 ------------------------------------------
```

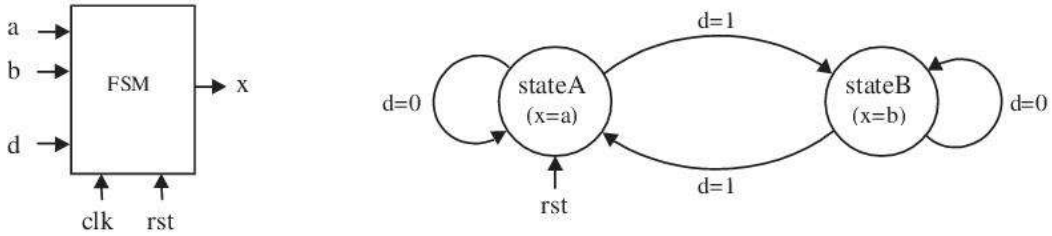


Figure 8.4
State machine of example 8.1.

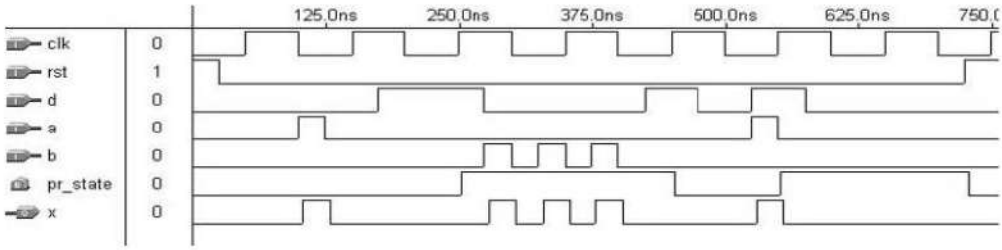


Figure 8.5
Simulation results of example 8.2

Simulation results relative to the code above are shown in figure 8.5. Notice that the circuit works as expected. Indeed, looking at the report files, one will verify that, as expected, only one flip-flop was required to implement this circuit because there are only two states to be encoded. Notice also that the upper section is indeed combinational for the output (x), which in this case does depend on the inputs (a or b, depending on which state the machine is in), varies when a or b vary, regardless of clk. If a synchronous output were required, then design style #2 should be employed.

8.3 Design Style #2 (Stored Output)

As we have seen, in design style #1 only pr-state is stored. Therefore, the overall circuit can be summarized as in figure 8.6(a). Notice that in this case, if it is a **Mealy** machine (**one whose output is dependent on the current input),** the output might change when the input changes (asynchronous output). In many applications, the signals are required to be synchronous, so the output should be updated only when the proper clock edge occurs. To make Mealy machines synchronous, the output must be stored as well, as shown in figure 8.6(b).
This structure is the object of design style #2.
To implement this new structure, very few modifications are needed. For example, we can use an additional signal (say, temp) to compute the output value (upper section), but only pass its value to the actual output signal when a clock event occurs (lower section). These modifications can be observed in the template shown below.

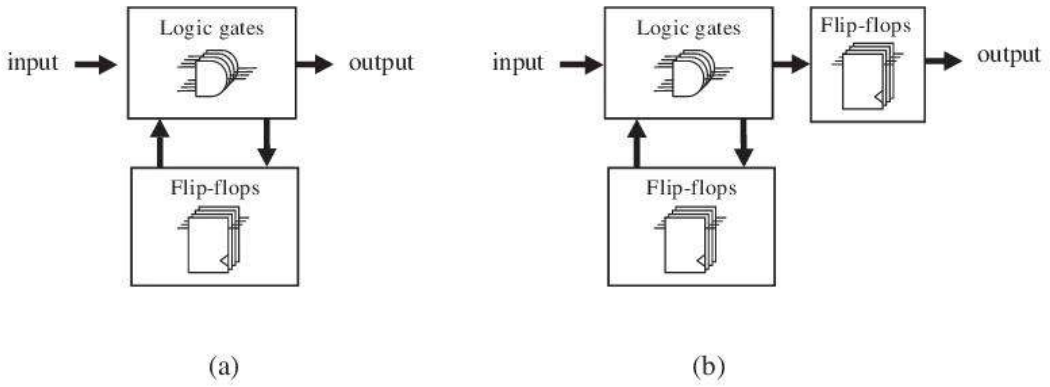


Figure 8.6
Circuit diagrams for (a) Design Style #1 and (b) Design Style #2.

State Machine Template for Design Style #2

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------------
ENTITY <ent_name> IS
PORT (input: IN <data_type>;
reset, clock: IN STD_LOGIC;
output: OUT <data_type>);
END <ent_name>;
-------------------------------------------------------
ARCHITECTURE <arch_name> OF <ent_name> IS
TYPE states IS (state0, state1, state2, state3, ...);
SIGNAL pr_state, nx_state: states;
SIGNAL temp: <data_type>;
BEGIN
---------- Lower section: -------------------------
PROCESS (reset, clock)
BEGIN
IF (reset='1') THEN
pr_state <= state0;
ELSIF (clock'EVENT AND clock='1') THEN
output <= temp;
pr_state <= nx_state;
END IF;
END PROCESS;
---------- Upper section: -------------------------
PROCESS (pr_state)
BEGIN
CASE pr_state IS
WHEN state0 =>
temp <= <value>;
IF (condition) THEN nx_state <= state1;
```

```
...
END IF;
WHEN state1 =>
temp <= <value>;
IF (condition) THEN nx_state <= state2;
...
END IF;
WHEN state2 =>
temp <= <value>;
IF (condition) THEN nx_state <= state3;
...
END IF;
...
END CASE;
END PROCESS;
END <arch_name>;
```

Comparing the template of design style #2 with that of design style #1, we verify that the only differences are those related to the introduction of the internal signal temp. This signal will cause the output of the state machine to be stored, for its value is passed to the output only when clk'EVENT occurs.

Example 8.3: Simple FSM #2

Let us consider the design of example 8.2 once again. However, let us say that now we want the output to be **synchronous** (to **change** only when **clock rises**). Since this isa **Mealy machine**, design style #2 is required.

```
1 ------------------------------------------
2 ENTITY simple_fsm IS
3 PORT ( a, b, d, clk, rst: IN BIT;
4 x: OUT BIT);
5 END simple_fsm;
6 ------------------------------------------
7 ARCHITECTURE simple_fsm OF simple_fsm IS
8 TYPE state IS (stateA, stateB);
9 SIGNAL pr_state, nx_state: state;
10 SIGNAL temp: BIT;
11 BEGIN
12 ----- Lower section: ----------------------
13 PROCESS (rst, clk)
14 BEGIN
15 IF (rst='1') THEN
16 pr_state <= stateA;
17 ELSIF (clk'EVENT AND clk='1') THEN
18 x <= temp;
19 pr_state <= nx_state;
20 END IF;
```

```
21 END PROCESS;
22 ---------- Upper section: ----------
23 PROCESS (a, b, d, pr_state)
24 BEGIN
25 CASE pr_state IS
26 WHEN stateA =>
27 temp <= a;
28 IF (d='1') THEN nx_state <= stateB;
29 ELSE nx_state <= stateA;
30 END IF;
31 WHEN stateB =>
32 temp <= b;
33 IF (d='1') THEN nx_state <= stateA;
34 ELSE nx_state <= stateB;
35 END IF;
36 END CASE;
37 END PROCESS;
38 END simple_fsm;
39 ------------------------------------------------
```

Looking at the report files produced by the compiler, we observe that two flip-flops were now inferred, one to encode the states of the machine, and the other to store the output.

Simulation results are shown in figure 8.7. Recall that when a signal is stored, its value will necessarily remain static between two consecutive clock edges. Therefore, if the input (a or b in the example above) changes during this interval, the change might not be observed by the circuit; moreover, when observed, it will be delayed with respect to the input (which is proper of synchronous circuits).

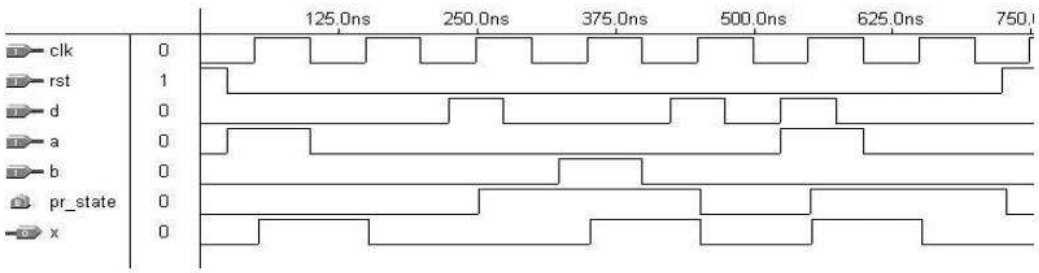


Figure 8.7
Simulation results of example 8.3.

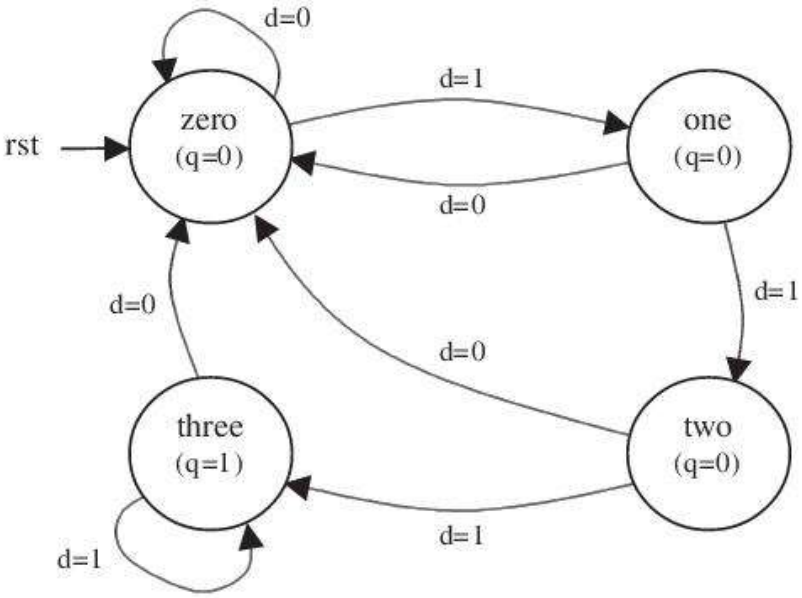


Figure 8.8
States diagram for example 8.4.

Example 8.4: String Detector

We want to design a circuit that takes as input a **serial bit stream and outputs a '1' whenever the sequence "111" occurs.** Overlaps must also be considered, that is, if . . . 0111110 . . . occurs, than the output should remain active for three consecutive clock cycles. The state diagram of our machine is shown in figure 8.8. There are four states, which we called zero, one, two, and three, with the name corresponding to the number of consecutive '1's detected. The solution shown below utilizes design style#1.

```
1 ----------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------------
5 ENTITY string_detector IS
6 PORT ( d, clk, rst: IN BIT;
7 q: OUT BIT);
8 END string_detector;
9 ----------------------------------------
10 ARCHITECTURE my_arch OF string_detector IS
11 TYPE state IS (zero, one, two, three);
12 SIGNAL pr_state, nx_state: state;
13 BEGIN
```

```
14 ----- Lower section: ----------
15 PROCESS (rst, clk)
16 BEGIN
17 IF (rst='1') THEN
18 pr_state <= zero;
19 ELSIF (clk'EVENT AND clk='1') THEN
20 pr_state <= nx_state;
21 END IF;
22 END PROCESS;
23 ---------- Upper section: ----------
24 PROCESS (d, pr_state)
25 BEGIN
26 CASE pr_state IS
27 WHEN zero =>
28 q <= '0';
29 IF (d='1') THEN nx_state <= one;
30 ELSE nx_state <= zero;
31 END IF;
32 WHEN one =>
33 q <= '0';
34 IF (d='1') THEN nx_state <= two;
35 ELSE nx_state <= zero;
36 END IF;
37 WHEN two =>
38 q <= '0';
39 IF (d='1') THEN nx_state <= three;
40 ELSE nx_state <= zero;
41 END IF;
42 WHEN three =>
43 q <= '1';
44 IF (d='0') THEN nx_state <= zero;
45 ELSE nx_state <= three;
46 END IF;
47 END CASE;
48 END PROCESS;
49 END my_arch;
50 ----------------------------------------
```

Notice that in this example the output does not depend on the current input. This fact can be observed in lines 28, 33, 38, and 43 of the code above, which show that all assignments to **q** are unconditional (that is, do not depend on **d**). Therefore, the output is automatically synchronous (a **Moore** machine), so the use of design style #2 is unnecessary. The circuit requires two flip-flops, which encode the four states of the state machine, from which q is computed.

Simulation results are shown in figure 8.9. As can be seen, the data sequence d =‘‘011101100’’ was applied to the circuit, resulting the response q =‘‘000100000’’ at the output.

Example 8.5: Traffic Light Controller (TLC)

As mentioned earlier, digital controllers are good examples of circuits that can be efficiently implemented when modeled as state machines. In the present example, we want to design a **TLC** with the characteristics summarized in the table of figure 8.10, that is:

1-Three modes of operation: Regular, Test, and Standby.

2-Regular mode: four states, each with an independent, programmable time, passed to the circuit by means of a CONSTANT.

3-Test mode: allows all pre-programmed times to be overwritten (by a manual switch) with a small value, such that the system can be easily tested during maintenance
(1 second per state). This value should also be programmable and passed to the circuit using a CONSTANT.

4-Standby mode: if set (by a sensor accusing malfunctioning, for example, or a manual switch) the system should activate the yellow lights in both directions and remain so while the standby signal is active

5-Assume that a 60 Hz clock (obtained from the power line itself) is available.

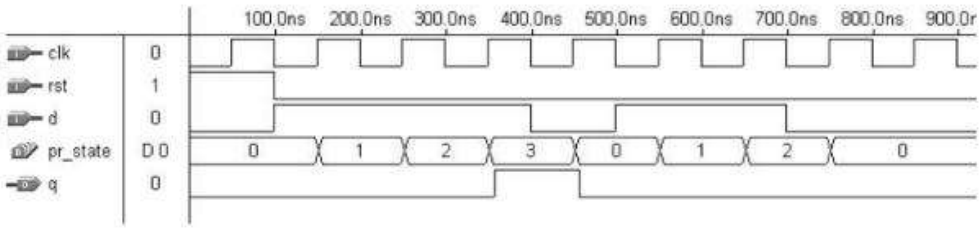
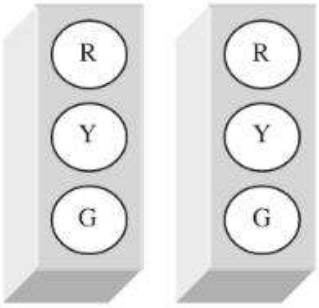


Figure 8.9
Simulation results of example 8.4.



State	Operation Mode		
	REGULAR	TEST	STANDBY
	Time	Time	Time
RG	timeRG (30s)	timeTEST (1s)	---
RY	timeRY (5s)	timeTEST (1s)	---
GR	timeGR (45s)	timeTEST (1s)	---
YR	timeYR (5s)	timeTEST (1s)	---
YY	---	---	Indefinite

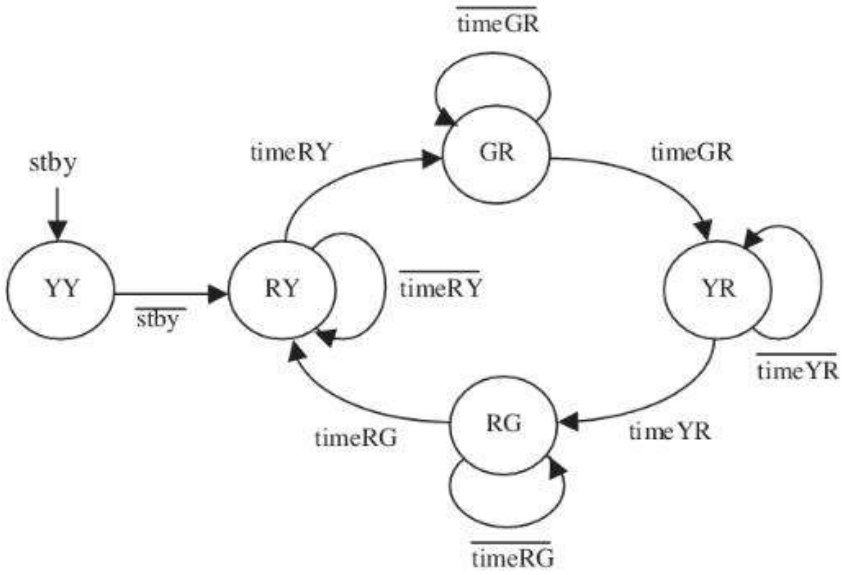


Figure 8.10
Specifications and states diagram (regular mode) for example 8.5.

Here, design style #1 can be employed, as shown in the code below.

```
1 ----------------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------------------
5 ENTITY tlc IS
6 PORT ( clk, stby, test: IN STD_LOGIC;
7 r1, r2, y1, y2, g1, g2: OUT STD_LOGIC);
```

```
8 END tlc;
9 ------------------------------------------------
10 ARCHITECTURE behavior OF tlc IS
11 CONSTANT timeMAX : INTEGER := 2700;
12 CONSTANT timeRG : INTEGER := 1800;
13 CONSTANT timeRY : INTEGER := 300;
14 CONSTANT timeGR : INTEGER := 2700;
15 CONSTANT timeYR : INTEGER := 300;
16 CONSTANT timeTEST : INTEGER := 60;
17 TYPE state IS (RG, RY, GR, YR, YY);
18 SIGNAL pr_state, nx_state: state;
19 SIGNAL time : INTEGER RANGE 0 TO timeMAX;
20 BEGIN
21 -------- Lower section of state machine: ----
22 PROCESS (clk, stby)
23 VARIABLE count : INTEGER RANGE 0 TO timeMAX;
24 BEGIN
25 IF (stby='1') THEN
26 pr_state <= YY;
27 count := 0;
28 ELSIF (clk'EVENT AND clk='1') THEN
29 count := count + 1;
30 IF (count = time) THEN
31 pr_state <= nx_state;
32 count := 0;
33 END IF;
34 END IF;
35 END PROCESS;

36 -------- Upper section of state machine: ----
37 PROCESS (pr_state, test)
38 BEGIN
39 CASE pr_state IS
40 WHEN RG =>
41 r1<='1'; r2<='0'; y1<='0'; y2<='0'; g1<='0'; g2<='1';
42 nx_state <= RY;
43 IF (test='0') THEN time <= timeRG;
44 ELSE time <= timeTEST;
45 END IF;
46 WHEN RY =>
47 r1<='1'; r2<='0'; y1<='0'; y2<='1'; g1<='0'; g2<='0';
48 nx_state <= GR;
49 IF (test='0') THEN time <= timeRY;
50 ELSE time <= timeTEST;
51 END IF;
52 WHEN GR =>
53 r1<='0'; r2<='1'; y1<='0'; y2<='0'; g1<='1'; g2<='0';
54 nx_state <= YR;
```

```
55 IF (test='0') THEN time <= timeGR;
56 ELSE time <= timeTEST;
57 END IF;
58 WHEN YR =>
59 r1<='0'; r2<='1'; y1<='1'; y2<='0'; g1<='0'; g2<='0';
60 nx_state <= RG;
61 IF (test='0') THEN time <= timeYR;
62 ELSE time <= timeTEST;
63 END IF;
64 WHEN YY =>
65 r1<='0'; r2<='0'; y1<='1'; y2<='1'; g1<='0'; g2<='0';
66 nx_state <= RY;
67 END CASE;
68 END PROCESS;
69 END behavior;
70 ------------------------------------------------
```

The expected number of flip-flops required to implement this circuit is 15; three to store pr _state (the machine has five states, so three bits are needed to encode them), plus twelve for the counter (it is a 12-bit counter, for it must count up to time MAX =2700).

Simulation results are shown in figure 8.11. In order for the results to fit properly in the graphs, we adopted small time values, with all CONSTANTS equal to 3 except time TEST, which was made equal to 1. Therefore, the system is expected to change state every three clock cycles when in Regular operation, or every clock cycle if in Test mode. These two cases can be observed in the first two graphs of figure 8.11, respectively. The third graph shows the Standby mode being activated. As expected, stby is asynchronous and has higher priority than test, causing the system to stay in state YY (state 4) while st by is active. The test signal, on the other hand, is synchronous,

but does not need to wait for the current state timing to finish to be activated, as can be observed in the second graph.

Example 8.6: Signal Generator

We want to design a circuit that, from a clock signal clk, gives origin to the signal outp shown in figure 8.12(a). Notice that the circuit must operate at both edges (rising and falling) of clk.

To circumvent the two-edge aspect (section 6.9), one alternative is to implement two machines, one that operates exclusively at the positive transition of clk and another

that operates exclusively at the negative edge, thus generating the intermediate signals out1 and out2 presented in figure 8.12(b). These signals can then be ANDed to give origin to the desired signal outp. Notice that this circuit has no external inputs (except for clk, of course), so the output can only change when clk changes (synchronous output).

```
1 ----------------------------------------
2 ENTITY signal_gen IS
3 PORT ( clk: IN BIT;
4 outp: OUT BIT);
```

```
5 END signal_gen;
6 ---------------------------------------
7 ARCHITECTURE fsm OF signal_gen IS
8 TYPE state IS (one, two, three);
9 SIGNAL pr_state1, nx_state1: state;
10 SIGNAL pr_state2, nx_state2: state;
11 SIGNAL out1, out2: BIT;
12 BEGIN
13 ----- Lower section of machine #1: ---
14 PROCESS(clk)
15 BEGIN
16 IF (clk'EVENT AND clk='1') THEN
17 pr_state1 <= nx_state1;
18 END IF;
19 END PROCESS;
20 ----- Lower section of machine #2: ---
21 PROCESS(clk)
22 BEGIN
23 IF (clk'EVENT AND clk='0') THEN
24 pr_state2 <= nx_state2;
25 END IF;
26 END PROCESS;
27 ---- Upper section of machine #1: -----
28 PROCESS (pr_state1)
29 BEGIN
30 CASE pr_state1 IS
31 WHEN one =>
32 out1 <= '0';
33 nx_state1 <= two;
34 WHEN two =>
35 out1 <= '1';
36 nx_state1 <= three;
37 WHEN three =>
38 out1 <= '1';
39 nx_state1 <= one;
40 END CASE;
41 END PROCESS;
42 ---- Upper section of machine #2: -----
43 PROCESS (pr_state2)
44 BEGIN
45 CASE pr_state2 IS
46 WHEN one =>
47 out2 <= '1';
48 nx_state2 <= two;
49 WHEN two =>
50 out2 <= '0';
51 nx_state2 <= three;
52 WHEN three =>
```

```
53 out2 <= '1';
54 nx_state2 <= one;
55 END CASE;
56 END PROCESS;
57 outp <= out1 AND out2;
58 END fsm;
59 ----------------------------------------
```

Simulation results from the circuit synthesized with the code above are shown in figure 8.13.

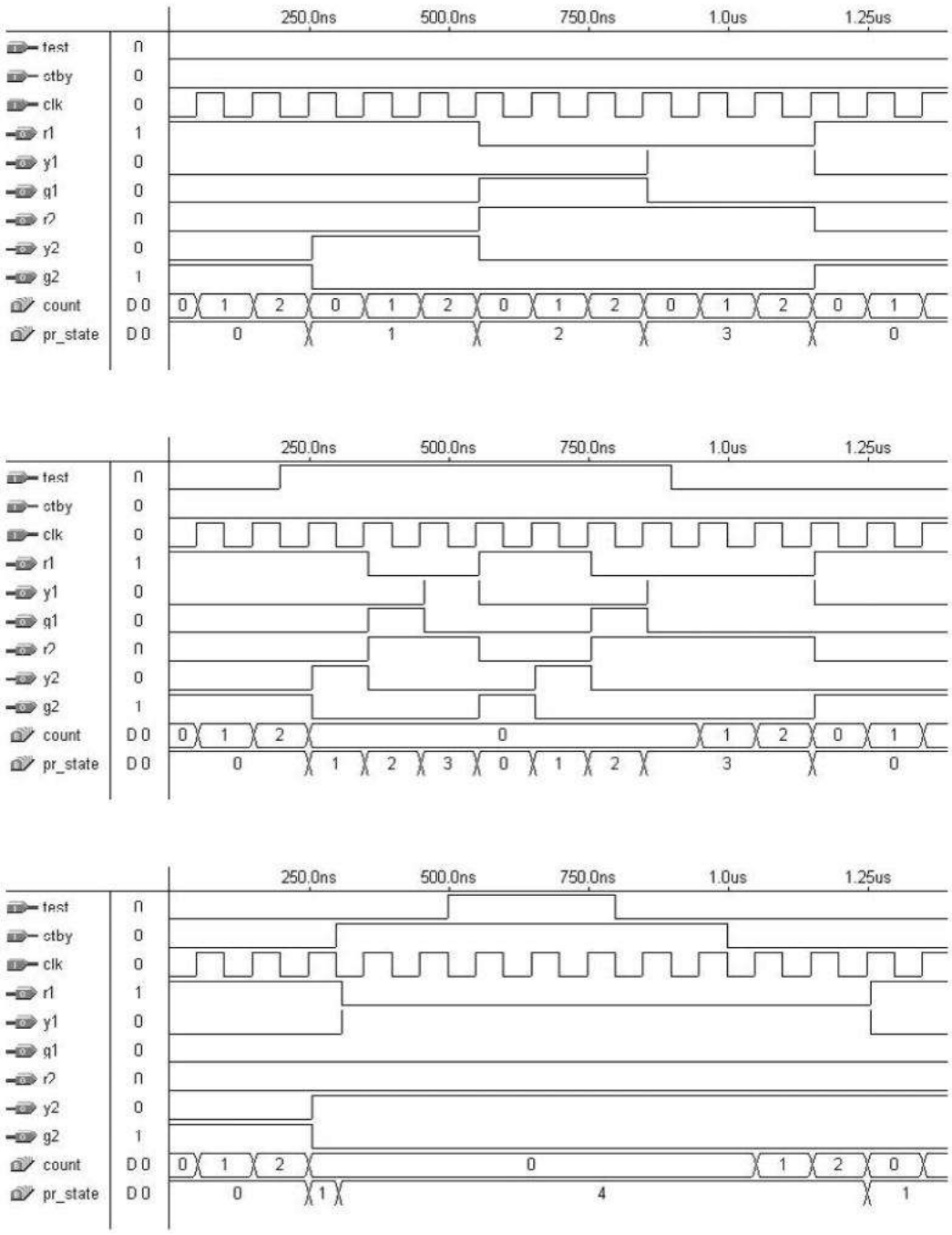


Figure 8.11
Simulation results of example 8.5.

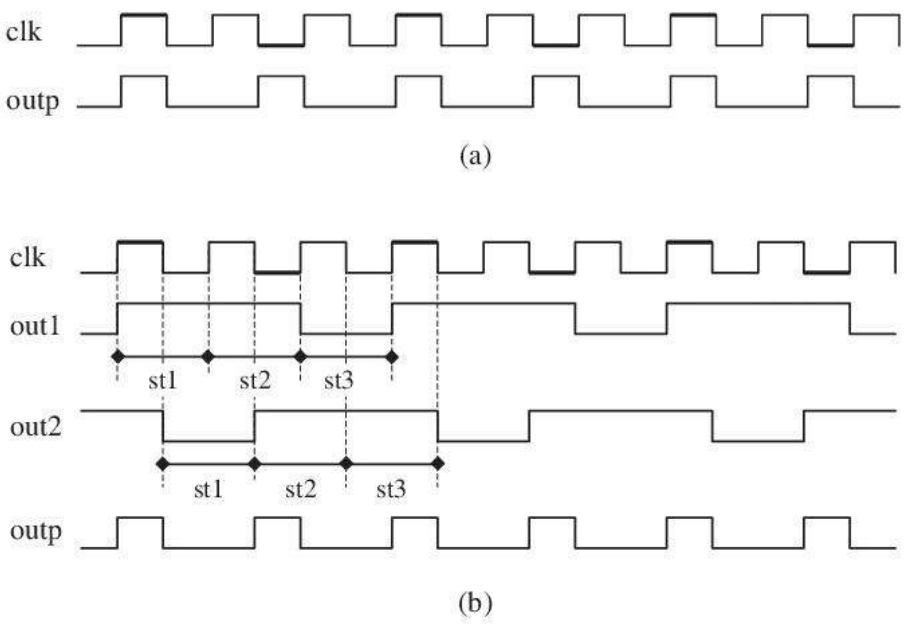


Figure 8.12
Waveforms of example 8.6: (a) signal outp to be generated from clk and (b) intermediate signals out1 and out2 (outp = out1 AND out2).

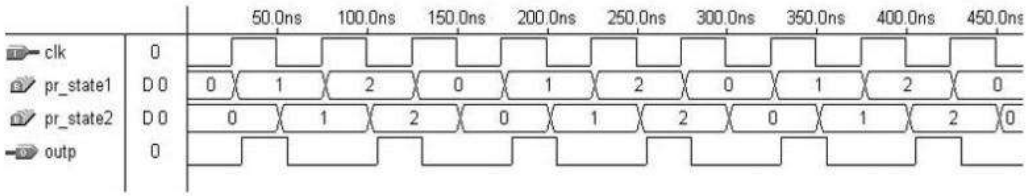


Figure 8.13
Simulation results of example 8.6.

9-AdditionalCircuit Designs

In the preceding chapters, we saw a series of complete design examples utilizing VHDL code. Each design included:
1-Top-level diagram of the circuit, with description;
2-Review of basic concepts whenever necessary;
3-Complete VHDL code;
4-Simulation results; and
5-Additional comments when needed.
The designs presented in this chapter are the following:
1-Barrel shifter (section 9.1)
2-Signed and unsigned comparators (section 9.2)
3-Carry ripple and carry look ahead adders (section 9.3)
4-Fixed-point division (section 9.4)
5-Vending machine controller (section 9.5)
6-Serial data receiver (section 9.6)
7-Parallel-to-serial converter (section 9.7)
8-Playing with a SSD (section 9.8)
9-Signal generators (section 9.9)
10-Memories (section 9.10)

9.1 Barrel Shifter

The diagram of a barrel shifter is shown in figure 9.1. The input is an 8-bit vector. The output is a shifted version of the input, with the amount of shift defined by the ‘‘shift’’ input (from 0 to 7). The circuit consists of three individual barrel shifters, each similar to that seen in example 6.9. Notice that the first barrel has only one ‘0’ connected to one of the multiplexers (bottom left corner), while the second has two, and the third has four. For larger vectors, we would just keep doubling the number of ‘0’ inputs. If shift =‘‘001’’, for example, then only the first barrel should cause a shift; on the other hand, if shift =‘‘111’’, then all barrels should cause a shift.
A VHDL code for the circuit of figure 9.1 is presented below. Simulation results, verifying the functionality of the circuit, are shown in figure 9.2. As can be seen in the latter, the output is equal to the input when shift =0 (that is, shift =‘‘000’’). It can also be seen that, as long as no bit of value ‘1’ is shifted out of the barrel, the output is equal to the input multiplied by 2 (1 shift) when shift =1 (‘‘001’’), multiplied by 4 (2 shifts) when shift =2 (‘‘010’’), multiplied by 8 (3 shifts) when shift =3 (‘‘011’’), and so on.

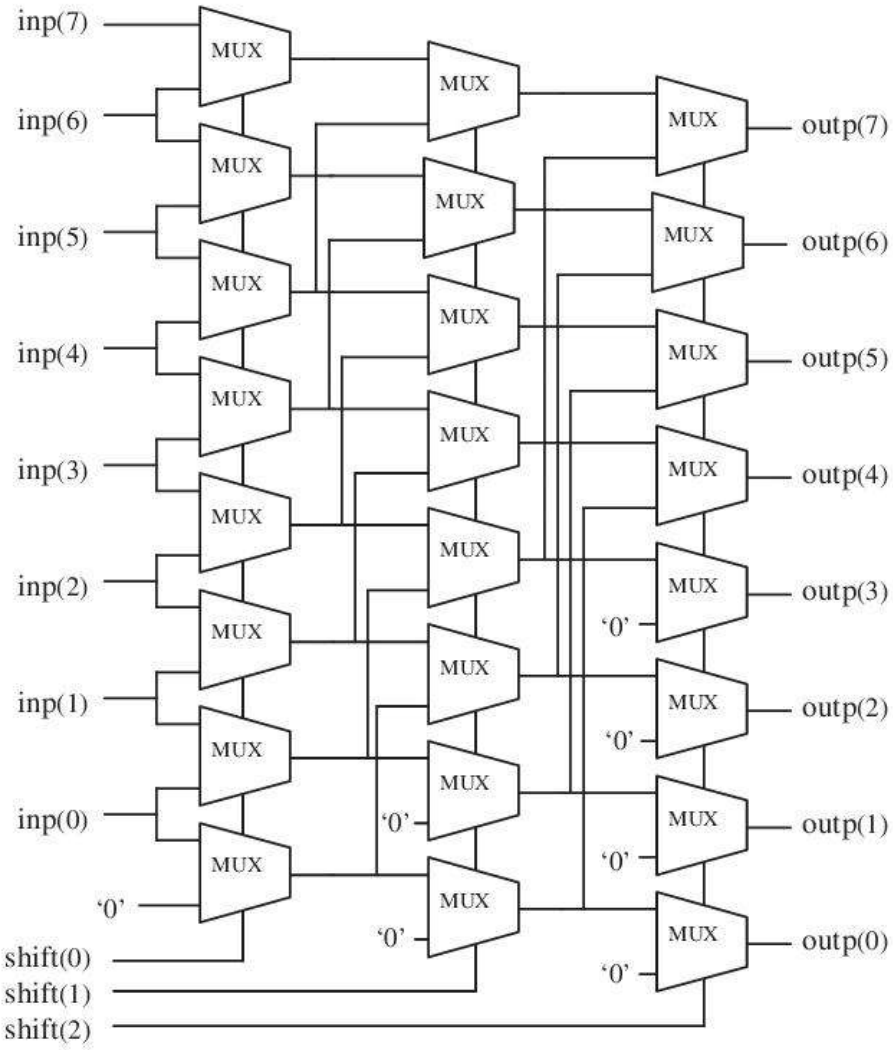


Figure 9.1
Barrel shifter.

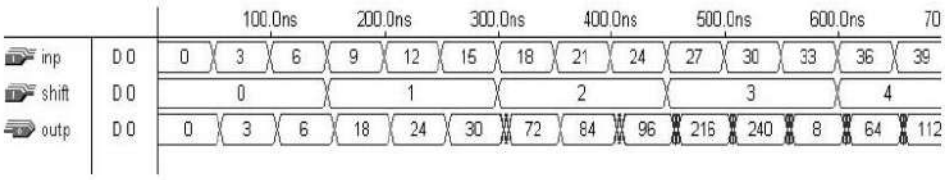


Figure 9.2
Simulation results from barrel shifter of figure 9.1.

```
1 ----------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------------
5 ENTITY barrel IS
6 PORT ( inp: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7 shift: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
8 outp: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
9 END barrel;
10 ----------------------------------------
11 ARCHITECTURE behavior OF barrel IS
12 BEGIN
13 PROCESS (inp, shift)
14 VARIABLE temp1: STD_LOGIC_VECTOR (7 DOWNTO 0);
15 VARIABLE temp2: STD_LOGIC_VECTOR (7 DOWNTO 0);
16 BEGIN
17 ---- 1st shifter -----
18 IF (shift(0)='0') THEN
19 temp1 := inp;
20 ELSE
21 temp1(0) := '0';
22 FOR i IN 1 TO inp'HIGH LOOP
23 temp1(i) := inp(i-1);
24 END LOOP;
25 END IF;
26 ---- 2nd shifter -----
27 IF (shift(1)='0') THEN
28 temp2 := temp1;
29 ELSE
30 FOR i IN 0 TO 1 LOOP
31 temp2(i) := '0';
32 END LOOP;
33 FOR i IN 2 TO inp'HIGH LOOP
34 temp2(i) := temp1(i-2);
35 END LOOP;
36 END IF;
37 ---- 3rd shifter -----
38 IF (shift(2)='0') THEN
39 outp <= temp2;
40 ELSE
41 FOR i IN 0 TO 3 LOOP
42 outp(i) <= '0';
43 END LOOP;
44 FOR i IN 4 TO inp'HIGH LOOP
45 outp(i) <= temp2(i-4);
46 END LOOP;
47 END IF;
```

```
48 END PROCESS;
49 END behavior;
50 ------------------------------------------
```

9.2 Signed and Unsigned Comparators

Figure 9.3 shows the top-level diagram of a comparator. The size of the vectors to be compared is generic (n +1). Three outputs must be provided: one corresponding to a >b, another to a =b, and finally one relative to a <b. Three solutions are presented: the first considers a and b as signed numbers, while the other two consider them as unsigned values. Simulation results are also included.

9.2-1 Signed Comparator

Notice the presence of the **std_logic_arith package** in the code below (line 4), which is necessary to operate with SIGNED (or UNSIGNED) data types (a and b were declared as SIGNED numbers in line 8).

```
1 ---- Signed Comparator: ----------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 USE ieee.std_logic_arith.all; -- necessary!
5 ---------------------------------------
6 ENTITY comparator IS
7 GENERIC (n: INTEGER := 7);
8 PORT (a, b: IN SIGNED (n DOWNTO 0);
9 x1, x2, x3: OUT STD_LOGIC);
10 END comparator;
11 ---------------------------------------
12 ARCHITECTURE signed OF comparator IS
13 BEGIN
14 x1 <= '1' WHEN a > b ELSE '0';
15 x2 <= '1' WHEN a = b ELSE '0';
16 x3 <= '1' WHEN a < b ELSE '0';
17 END signed;
18 ---------------------------------------
```

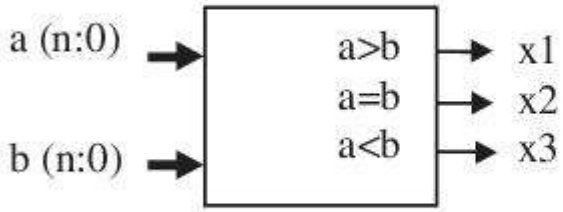


Figure 9.3
Comparator.

Simulation results are shown in figure 9.4. As can be seen, 127 >0, but 128 <0 and also 255 <0 (because in 2's complement notation 127 is the decimal 127 itself, but 128 is the decimal _128, and 255 is indeed -1).

9.2-2 Unsigned Comparator #1

The VHDL code below is the counterpart of the code just presented (signed comparator).
Notice again the presence of the std_logic_arith package (line 4), which is necessary to operate with UNSIGNED (or SIGNED) data types (a and b were declared as UNSIGNED numbers in line 8).

```
1 ---- Unsigned Comparator #1: -----------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 USE ieee.std_logic_arith.all; -- necessary!
5 ---------------------------------------
6 ENTITY comparator IS
7 GENERIC (n: INTEGER := 7);
8 PORT (a, b: IN UNSIGNED (n DOWNTO 0);
9 x1, x2, x3: OUT STD_LOGIC);
10 END comparator;
11 ---------------------------------------
12 ARCHITECTURE unsigned OF comparator IS
13 BEGIN
14 x1 <= '1' WHEN a > b ELSE '0';
15 x2 <= '1' WHEN a = b ELSE '0';
16 x3 <= '1' WHEN a < b ELSE '0';
17 END unsigned;
18 ---------------------------------------
```

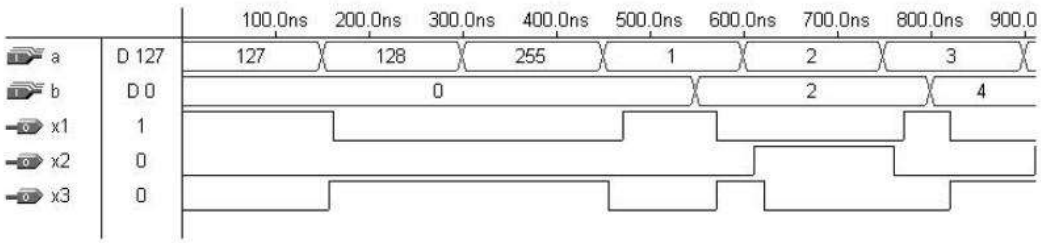


Figure 9.4
Simulation result of *signed* comparator of figure 9.3.

9.3Carry Ripple and Carry Look Ahead Adders

Carry ripple and carry look ahead are two classical approaches to the design of Adders . The former has the advantage of requiring less hardware, while the latter is Faster . Both approaches are discussed below.

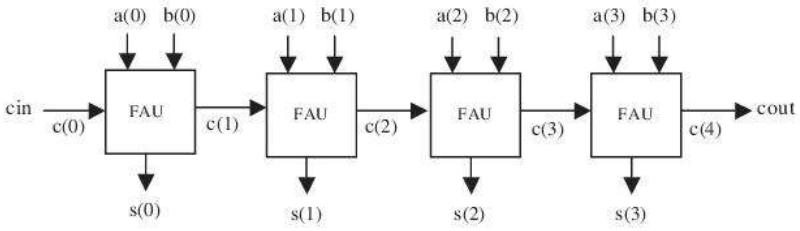
Carry Ripple Adder

Figure 9.6 shows a 4-bit unsigned carry ripple adder. For each bit, a **f**ull **a**dder **u**nit (**FAU**, section 1.4) is employed. The truth table of the FAU is also shown. In it, **a** and **b** represent the **input** bits, **cin** is the **carry-in** bit, **s** is the **sum** bit, and **cout** is the **carry-out** bit. **s** must be **high** whenever the number of **inputs** that are **high** is **odd** (**parity function**), while **cout** must be **high** when **two** or more **inputs** are **high** (**majority function**).

s = a XOR b XOR cin

cout = (a AND b) OR (a AND cin) OR (b AND cin)

Therefore, a VHDL implementation of the carry ripple adder is straightforward. The solution shown below works for any number (n) of input bits, defined by means of a GENERIC statement in line 5. Simulation results from the circuit synthesized with the code below are shown in figure 9.7.



a b cin	s cout
0 0 0	0 0
0 1 0	1 0
1 0 0	1 0
1 1 0	0 1
0 0 1	1 0
0 1 1	0 1
1 0 1	0 1
1 1 1	1 1

Figure 9.6
4-bit carry ripple adder and truth table of Full Adder Unit (FAU).

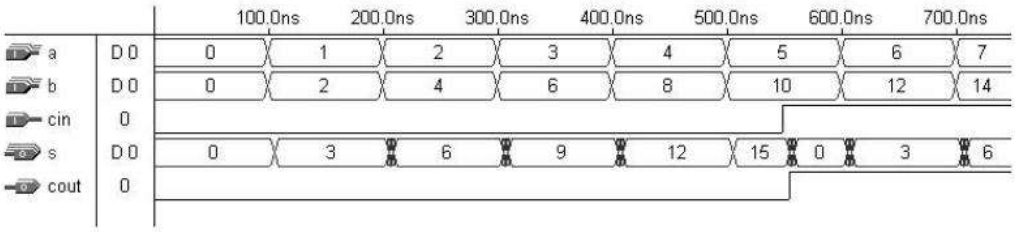


Figure 9.7
Simulation results from the carry ripple adder of figure 9.6.

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3 ---------------------------------------------
4 ENTITY adder_cripple IS
5 GENERIC (n: INTEGER := 4);
6 PORT ( a, b: IN STD_LOGIC_VECTOR (n-1 DOWNTO 0);
7 cin: IN STD_LOGIC;
8 s: OUT STD_LOGIC_VECTOR (n-1 DOWNTO 0);
9 cout: OUT STD_LOGIC);
10 END adder_cripple;
11 ---------------------------------------------
12 ARCHITECTURE adder OF adder_cripple IS
13 SIGNAL c: STD_LOGIC_VECTOR (n DOWNTO 0);
14 BEGIN
15 c(0) <= cin;
16 G1: FOR i IN 0 TO n-1 GENERATE
17 s(i) <= a(i) XOR b(i) XOR c(i);
18 c(i+1) <= (a(i) AND b(i)) OR
19 (a(i) AND c(i)) OR
20 (b(i) AND c(i));
21 END GENERATE;
22 cout <= c(n);
23 END adder;
24 ---------------------------------------------
```

Carry Look Ahead Adder

A diagram of a 4-bit carry look ahead adder is shown in figure 9.8. Its implementation is based on the generate and propagate concept, which gives the circuit higher speed than its carry ripple adder counterpart (at the expense of more silicon area).
Consider two input bits, a and b. The generate (g) and propagate (p) signals are defined as:

g =a AND b

p =a XOR b

Notice that such signals can be computed in advance, because neither depends on the carry bit.

If we consider now two input vectors, a =a(n -1) . . . a(1)a(0) and b =b(n -1) . . . b(1)b(0), then the corresponding generate and propagate vectors are g =g(n -1) . . . g(1)g(0) and p =p(n -1) . . . p(1)p(0), where
g(j) =a(j) AND b(j)
p(j) =a(j) XOR b(j)
Let us consider now the carry vector, c =c(n -1) . . . c(1)c(0). The carry bits can be computed from g and p:
c(0)=cin
c(1) =c(0)p(0) +g(0)
c(2) = c(0)p(0)p(1) +g(0)p(1) +g(1)
c(3) = c(0)p(0)p(1)p(2) +g(0)p(1)p(2) + g(1)p(2) +g(2), etc.

Independently; that is, none of the expressions above depends on preceding carry Computations, and that is the reason why this circuit is faster. On the other hand, the Hardware complexity grows very fast, limiting this approach to just a few bits (typically four). Larger carry look ahead adders can be implemented by associating such4-bit-or-so units.

The implementation of the adder of figure 9.8 is now straightforward. The **PGU** (**P**ropagate—**G**enerate **U**nit) computes p and g (four units are required), plus the Actual sum (s), while the **CLAU** (**C**arry **L**ook **A**head **U**nit) computes the carry bits. Note: In order to construct bigger carry look ahead adders, the CLAU block of figure 9.8 must posses Group Propagate (GP) and Group Generate (GG) outputs, Which were omitted in the figure because this implementation is intended for four bits Only.

```
1 ----------------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------------------
5 ENTITY CLA_Adder IS
6 PORT ( a, b: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
7 cin: IN STD_LOGIC;
8 s: OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
9 cout: OUT STD_LOGIC);
10 END CLA_Adder;
11 ----------------------------------------------
12 ARCHITECTURE CLA_Adder OF CLA_Adder IS
13 SIGNAL c: STD_LOGIC_VECTOR (4 DOWNTO 0);
14 SIGNAL p: STD_LOGIC_VECTOR (3 DOWNTO 0);
15 SIGNAL g: STD_LOGIC_VECTOR (3 DOWNTO 0);
16 BEGIN
17 ---- PGU: --------------------------------
18 G1: FOR i IN 0 TO 3 GENERATE
19 p(i) <= a(i) XOR b(i);
20 g(i) <= a(i) AND b(i);
21 s(i) <= p(i) XOR c(i);
22 END GENERATE;
```

```
23 ---- CLAU: --------------------------------
24 c(0) <= cin;
25 c(1) <= (cin AND p(0)) OR
26 g(0);
27 c(2) <= (cin AND p(0) AND p(1)) OR
28 (g(0) AND p(1)) OR
29 g(1);
30 c(3) <= (cin AND p(0) AND p(1) AND p(2)) OR
31 (g(0) AND p(1) AND p(2)) OR
32 (g(1) AND p(2)) OR
33 g(2);
34 c(4) <= (cin AND p(0) AND p(1) AND p(2) AND p(3)) OR
35 (g(0) AND p(1) AND p(2) AND p(3)) OR
36 (g(1) AND p(2) AND p(3)) OR
37 (g(2) AND p(3)) OR
38 g(3);
39 cout <= c(4);
40 END CLA_Adder;
41 ---------------------------------------------
```

Qualitatively, the simulation results obtained from the circuit synthesized with the code above are similar to those from the carry ripple adder presented in figure 9.7.

9.4 Fixed-Point Division

We saw in chapter 4 that the pre-defined ‘‘/’’ (division) operator accepts only power of two divisors, that is, it is indeed a ‘‘shift’’ operator. In this section, we will discuss the implementation of **generic division**, in which the dividend and divisor can be any integer. We start by describing the division algorithm, then we present two VHDL solutions followed by simulation results.

Division Algorithm

Say that we want to calculate y =a/b, where a, b, and y have the same number (n +1) of bits. The algorithm is illustrated in figure 9.9, for a =‘‘1011’’ (decimal 11) and b =‘‘0011’’ (decimal 3), from which we expect y =‘‘0011’’ (decimal 3) and remainder‘‘0010’’ (decimal 2). We first create a shifted version of b, whose length is 2n +1 bits (shown in the b-related column in figure 9.9). b_inp (i) is simply b shifted to the left by **i** positions (notice the underscored characters in the b-related column).

Index (i)	a-related input (a_inp)	Comparison	b-related input (b_inp)	y (quotient)	Operation on 1st column
3	1011	<	0011000	0	none
2	1011	<	0001100	0	none
1	1011	>	0000110	1	a_inp(i)-b_inp(i)
0	0101	>	0000011	1	a_inp(i)-b_inp(i)

0010(rem)

The computation of the quotient is performed as follows. Starting from the top of

the table, we compare(a-inp(i)) with (b-inp(i).) If the former is **bigger** than or **equal** to the latter, than **y(i) ='1'** and **b-inp(i)** is **subtracted** from **a-inp(i);** otherwise, **y(i) ='0'** and we simply proceed to the next line. After **n +1** iterations, the computation is completed and the value left in a-inp is the remainder.

Note: It is obvious that, to subtract b-inp from a-inp, the number of bits of a-inp cannot be **less** than that of b-inp, so the actual length of a-inp must be increased, which is attained by simply filling a-inp with n '0's on its left-hand side ('0's not shown in figure 9.9).

Another way of presenting the division algorithm is the following. We multiply b by **2**n**, where **n +1** is the number of bits. This, of course, corresponds to shifting **b n** positions to the left, but without throwing out any of its bits (so the new b-vector must be n bits longer than the original vector). If a is **bigger** than the new **b**, then **y(n) = '1'**, and **b** (the new value) must be **subtracted** from **a**; otherwise, **y(n) ='0'.** Now we move to the next iteration. We multiply b (the original value) by 2**(n -1), which is equivalent to shifting the original vector n -1 positions to the left, or shifting the value of b just used in the previous computation back one position to the right. Then we compare it to a, as we did before, to decide whether y(n -1) should be '1' or '0', and so on.

VHDL Dividers

Below are two solutions for the division problem. Both use **sequential code**: IF is used in the first, while LOOP plus IF are employed in the second. The first solution is a step-by-step code, so the division algorithm described above can be clearly observed. The second is more compact and is also generic (notice that n was defined

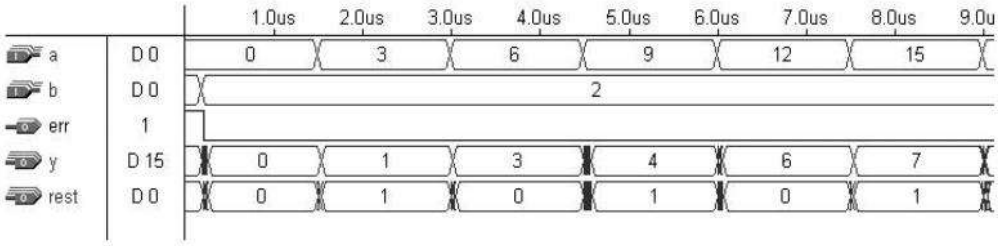


Figure 9.10
Simulation results of divider (for 4-bit operands).

by means of a GENERIC statement in line 6). The solutions include also a b =0 check routine. Simulation results are shown in figure 9.10.

```
1 ----- Solution 1: step-by-step----------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ------------------------------------------------
5 ENTITY divider IS
```

```
6 PORT ( a, b: IN INTEGER RANGE 0 TO 15;
7 y: OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
8 rest: OUT INTEGER RANGE 0 TO 15;
9 err : OUT STD_LOGIC);
10 END divider;
11 ------------------------------------------------
12 ARCHITECTURE rtl OF divider IS
13 BEGIN
14 PROCESS (a, b)
15 VARIABLE temp1: INTEGER RANGE 0 TO 15;
16 VARIABLE temp2: INTEGER RANGE 0 TO 15;
17 BEGIN
18 ----- Error and initialization: -------
19 temp1 := a;
20 temp2 := b;
21 IF (b=0) THEN err <= '1';
22 ELSE err <= '0';
23 END IF;
24 ----- y(3): ---------------------------
25 IF (temp1 >= temp2 * 8) THEN
26 y(3) <= '1';
27 temp1 := temp1 - temp2*8;
28 ELSE y(3) <= '0';
29 END IF;
30 ----- y(2): ---------------------------
31 IF (temp1 >= temp2 * 4) THEN
32 y(2) <= '1';
33 temp1 := temp1 - temp2 * 4;
34 ELSE y(2) <= '0';
35 END IF;
36 ----- y(1): ---------------------------
37 IF (temp1 >= temp2 * 2) THEN
38 y(1) <= '1';
39 temp1 := temp1 - temp2 * 2;
40 ELSE y(1) <= '0';
41 END IF;
42 ----- y(0): ---------------------------
43 IF (temp1 >= temp2) THEN
44 y(0) <= '1';
45 temp1 := temp1 - temp2;
46 ELSE y(0) <= '0';
47 END IF;
48 ----- Remainder: ----------------------
49 rest <= temp1;
50 END PROCESS;
51 END rtl;
52 ------------------------------------------------
```

```
1 ------ Solution 2: compact and generic-----------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ------------------------------------------------
5 ENTITY divider IS
6 GENERIC(n: INTEGER := 3);
7 PORT ( a, b: IN INTEGER RANGE 0 TO 15;
8 y: OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
9 rest: OUT INTEGER RANGE 0 TO 15;
10 err : OUT STD_LOGIC);
11 END divider;
12 ------------------------------------------------
13 ARCHITECTURE rtl OF divider IS
14 BEGIN
15 PROCESS (a, b)
16 VARIABLE temp1: INTEGER RANGE 0 TO 15;
17 VARIABLE temp2: INTEGER RANGE 0 TO 15;
18 BEGIN
19 ----- Error and initialization: -------
20 temp1 := a;
21 temp2 := b;
22 IF (b=0) THEN err <= '1';
23 ELSE err <= '0';
24 END IF;
25 ----- y: -----------------------------
26 FOR i IN n DOWNTO 0 LOOP
27 IF(temp1 >= temp2 * 2**i) THEN
28 y(i) <= '1';
29 temp1 := temp1 - temp2 * 2**I;
30 ELSE y(i) <= '0';
31 END IF;
32 END LOOP;
33 ----- Remainder: ----------------------
34 rest <= temp1;
35 END PROCESS;
36 END rtl;
37 ------------------------------------------------
```

9.5 Vending-Machine Controller

The inputs and outputs of the controller are shown in figure 9.11. The input signals nickel-in, dime-in, and quarter-in indicate that a corresponding coin has been deposited. Two additional inputs, clk (clock) and rst (reset), are also necessary. The controller responds with three outputs: candy-out, to dispense a candy bar, plus nickel-out and dime-out, asserted when change is due.
Figure 9.11 also shows the states of the corresponding FSM. The numbers inside the circles represent the total amount deposited by the customer (only nickels, dimes,

and quarters are accepted). State 0 is the idle state. From it, if a nickel is deposited, the machine moves to state 5; if a dime, to state 10; or if a quarter, to state 25. Similar situations are repeated for all states, up to state 20. If state 25 is reached, then a candy bar is dispensed, with no change. However, if state 40 is reached, for example, then a nickel is delivered, passing therefore the system to state 35, from which a dime is delivered and a candy bar dispensed. The three states marked with double circles are those from which a candy bar is delivered and the machine returns to state 0.
This problem will be divided into two parts: in the first, the fundamental aspects related to the design of the vending machine controller (figure 9.11) are treated; in the second, additional (and indispensable) features are added. The first part is studied in this section, while the second is proposed as a problem (problem 9.3). The introduction of such additional features is necessary for safety reasons; since we are dealing with money, we must assure that none of the parts (machine or customer) will be hurt in the transaction
A VHDL code, treating only the basic features of the problem depicted in figure 9.11, is presented below. We have assumed that the additional features proposed in problem 9.3 will indeed be implemented, in which case glitches are acceptable in the first part of the solution. Therefore, design style #1 (section 8.2) can be employed. The enumerated type state (line 12) contains a list of all states shown in the FSM diagram of figure 9.11. There are ten states, so four bits are necessary to encode them (so four flip-flops will be inferred). Recall that the compiler encodes such states in the order that they are listed, so st0 = ‘‘0000’’ (decimal 0), st5 = ‘‘0001’’ (decimal 1), . . . , st45 =‘‘1001’’ (decimal 9). Therefore, in the simulations, such numbers are shown instead of the state names.

```
1 ----------------------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------------------------
5 ENTITY vending_machine IS
6 PORT ( clk, rst: IN STD_LOGIC;
7 nickel_in, dime_in, quarter_in: IN BOOLEAN;
8 candy_out, nickel_out, dime_out: OUT STD_LOGIC);
9 END vending_machine;
10 ----------------------------------------------------
11 ARCHITECTURE fsm OF vending_machine IS
12 TYPE state IS (st0, st5, st10, st15, st20, st25,
13 st30, st35, st40, st45);
14 SIGNAL present_state, next_state: STATE;
15 BEGIN
16 ---- Lower section of the FSM (Sec. 8.2): ---------
17 PROCESS (rst, clk)
18 BEGIN
19 IF (rst='1') THEN
20 present_state <= st0;
21 ELSIF (clk'EVENT AND clk='1') THEN
22 present_state <= next_state;
23 END IF;
24 END PROCESS;
```

```
25 ---- Upper section of the FSM (Sec. 8.2): ---------
26 PROCESS (present_state, nickel_in, dime_in, quarter_in)
27 BEGIN
28 CASE present_state IS
29 WHEN st0 =>
30 candy_out <= '0';
31 nickel_out <= '0';
32 dime_out <= '0';
33 IF (nickel_in) THEN next_state <= st5;
34 ELSIF (dime_in) THEN next_state <= st10;
35 ELSIF (quarter_in) THEN next_state <= st25;
36 ELSE next_state <= st0;
37 END IF;
38 WHEN st5 =>
39 candy_out <= '0';
40 nickel_out <= '0';
41 dime_out <= '0';
42 IF (nickel_in) THEN next_state <= st10;
43 ELSIF (dime_in) THEN next_state <= st15;
44 ELSIF (quarter_in) THEN next_state <= st30;
45 ELSE next_state <= st5;
46 END IF;
47 WHEN st10 =>
48 candy_out <= '0';
49 nickel_out <= '0';
50 dime_out <= '0';
51 IF (nickel_in) THEN next_state <= st15;
52 ELSIF (dime_in) THEN next_state <= st20;
53 ELSIF (quarter_in) THEN next_state <= st35;
54 ELSE next_state <= st10;
55 END IF;
56 WHEN st15 =>
57 candy_out <= '0';
58 nickel_out <= '0';
59 dime_out <= '0';
60 IF (nickel_in) THEN next_state <= st20;
61 ELSIF (dime_in) THEN next_state <= st25;
62 ELSIF (quarter_in) THEN next_state <= st40;
63 ELSE next_state <= st15;
64 END IF;
65 WHEN st20 =>
66 candy_out <= '0';
67 nickel_out <= '0';
68 dime_out <= '0';
69 IF (nickel_in) THEN next_state <= st25;
70 ELSIF (dime_in) THEN next_state <= st30;
71 ELSIF (quarter_in) THEN next_state <= st45;
72 ELSE next_state <= st20;
```

```
73 END IF;
74 WHEN st25 =>
75 candy_out <= '1';
76 nickel_out <= '0';
77 dime_out <= '0';
78 next_state <= st0;
79 WHEN st30 =>
80 candy_out <= '1';
81 nickel_out <= '1';
82 dime_out <= '0';
83 next_state <= st0;
84 WHEN st35 =>
85 candy_out <= '1';
86 nickel_out <= '0';
87 dime_out <= '1';
88 next_state <= st0;
89 WHEN st40 =>
90 candy_out <= '0';
91 nickel_out <= '1';
92 dime_out <= '0';
93 next_state <= st35;
94 WHEN st45 =>
95 candy_out <= '0';
96 nickel_out <= '0';
97 dime_out <= '1';
98 next_state <= st35;
99 END CASE;
100 END PROCESS;
101
102 END fsm;
103 -------------------------------------------------------
```

Simulation results are presented in figure 9.12. As can be seen, three nickels and one quarter were deposited. Notice that, at the first positive clock edge after the first nickel was deposited, the FSM moves from state st0 (decimal 0) to st5 (decimal 1);

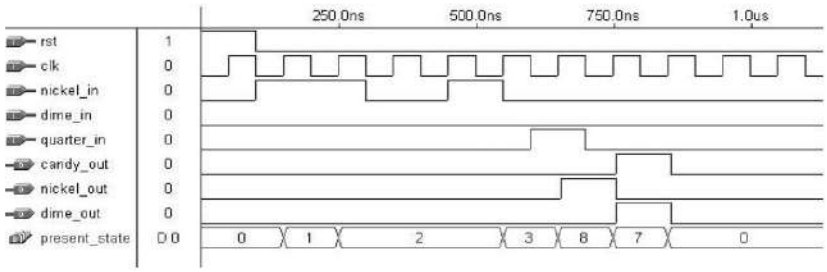


Figure 9.12
Simulation results from the vending-machine controller.

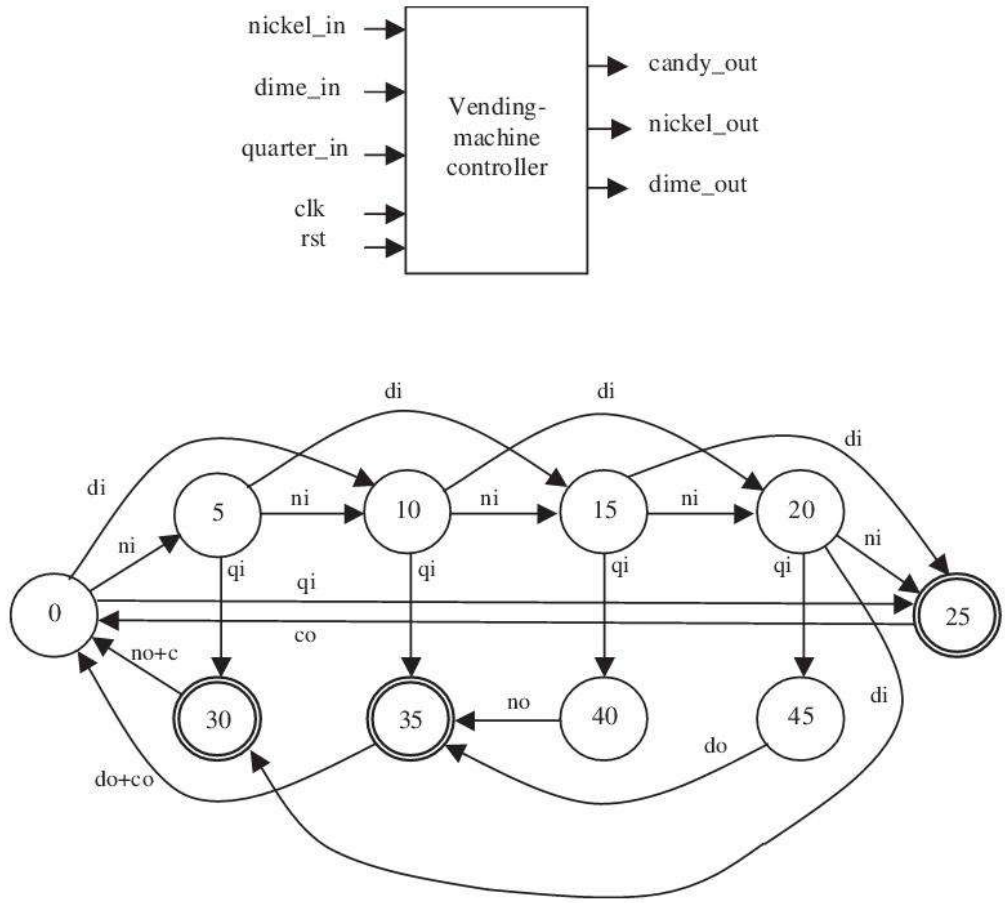


Figure 9.11
Vending-machine controller (top-level and states diagrams). The signals are. ni = nickel_in, di = dime_in, qi = quarter_in, no = nickel_out, do = dime_out, and co = candy_out.

after de second nickel, to state st10 (decimal 2); after de third, to state st15 (decimal 3); and, after de quarter has been deposited, to state st40 (decimal 8). After that, a nickel is returned to the customer (nickel-out ='1'), causing the FSM to move to state st35 (decimal 7), at which a dime is delivered (dime-out ='1') and a candy bar is dispensed (candy-out ='1'). The system returns then to its idle state (st0).
As mentioned above, additional features (like handshake) are necessary to increase the security of the transactions. Please refer to problem 9.3 for a continuation of this design.

9.6 Serial Data Receiver

The diagram of a serial data receiver is shown in figure 9.13. It contains a serial data input, din, and a parallel data output, data(6:0). A clock signal is also needed at the input. Two supervision signals are generated by the circuit: err (error) and data-valid. The input train consists of ten bits. The first bit is a start bit, which, when high, must cause the circuit to start receiving data. The next seven are the actual data bits. The ninth bit is a parity bit, whose status must be '0' if the number of ones in data is even, or '1' otherwise. Finally, the tenth is a stop bit, which must be high if the transmission is correct. An error is detected when either the parity does not check or the stop bit is not a '1'. When reception is concluded and if no error has been detected, then the data stored in the internal registers (reg) is transferred to data(6:0) and the data_valid output is asserted.
A VHDL code for this circuit is presented below. A few variables were used: count, to determine the number of bits received; reg, which stores the data; and temp, to compute the error. Notice in line 37 that reg(0) =din was used instead of reg(0) ='0', because we want the time slot immediately after the stop bit to be considered as possibly containing a start bit for the next input train.

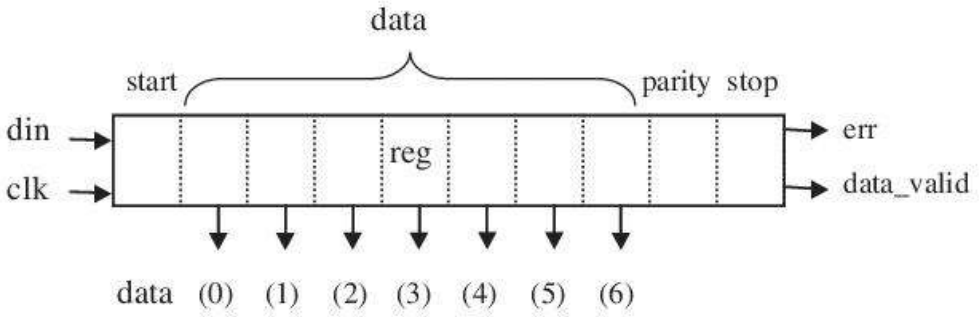


Figure 9.13
Serial data receiver.

```
1 ----------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------------
5 ENTITY receiver IS
6 PORT ( din, clk, rst: IN BIT;
7 data: OUT BIT_VECTOR (6 DOWNTO 0);
8 err, data_valid: OUT BIT);
9 END receiver;
10 ----------------------------------------
11 ARCHITECTURE rtl OF receiver IS
12 BEGIN
13 PROCESS (rst, clk)
14 VARIABLE count: INTEGER RANGE 0 TO 10;
15 VARIABLE reg: BIT_VECTOR (10 DOWNTO 0);
```

```
16 VARIABLE temp : BIT;
17 BEGIN
18 IF (rst='1') THEN
19 count:=0;
20 reg := (reg'RANGE => '0');
21 temp := '0';
22 err <= '0';
23 data_valid <= '0';
24 ELSIF (clk'EVENT AND clk='1') THEN
25 IF (reg(0)='0' AND din='1') THEN
26 reg(0) := '1';
27 ELSIF (reg(0)='1') THEN
28 count := count + 1;
29 IF (count < 10) THEN
30 reg(count) := din;
31 ELSIF (count = 10) THEN
32 temp := (reg(1) XOR reg(2) XOR reg(3) XOR
33 reg(4) XOR reg(5) XOR reg(6) XOR
34 reg(7) XOR reg(8)) OR NOT reg(9);
35 err <= temp;
36 count := 0;
37 reg(0) := din;
38 IF (temp = '0') THEN
39 data_valid <= '1';
40 data <= reg(7 DOWNTO 1);
41 END IF;
42 END IF;
43 END IF;
44 END IF;
45 END PROCESS;
46 END rtl;
47 ------------------------------------------------
```

Simulation results are presented in figure 9.14. The input sequence is din = {start =1, din =0111001, parity =0, stop =1}. As can be seen in the upper graph, no error was detected in this case, because the parity and stop bits are correct. Hence, after count reaches 9, the data is made available, that is, data =0111001, from data(0) to data(6), which corresponds to the decimal 78, and the data-valid bit is Figure

9.7 Parallel-to-Serial Converter

A parallel-to-serial converter is a typical application of shift registers. It consists of sending out a block of data serially. The need for such converters arises, for example, in ASIC chips when there are not enough pins available to output all data bits simultaneously.

A diagram of a parallel-to-serial converter is presented in figure 9.15. d(7:0) is the data vector to be sent out, while dout is the actual output. There are also two other inputs: clk and load. When load is asserted, d is synchronously stored in the shift register reg. While load stays high, the MSB, d(7), remains available at the output. Once load is returned to '0', the subsequent bits are presented at the output at each positive edge of clk. After all eight bits have been sent out, the output remains low until the next transmission.

```
1 ------------------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ------------------------------------------------
5 ENTITY serial_converter IS
6 PORT ( d: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7 clk, load: IN STD_LOGIC;
8 dout: OUT STD_LOGIC);
9 END serial_converter;
10 ------------------------------------------------
11 ARCHITECTURE serial_converter OF serial_converter IS
12 SIGNAL reg: STD_LOGIC_VECTOR (7 DOWNTO 0);
13 BEGIN
14 PROCESS (clk)
15 BEGIN
16 IF (clk'EVENT AND clk='1') THEN
17 IF (load='1') THEN reg <= d;
18 ELSE reg <= reg(6 DOWNTO 0) & '0';
19 END IF;
20 END IF;
21 END PROCESS;
22 dout <= reg(7);
23 END serial_converter;
24 ------------------------------------------------
```

Simulation results from the circuit synthesized with the code above are shown in figure 9.16. d ="11011011" (decimal 219) was chosen. As can be seen, d(7) ='1' is presented at the output at the first rising edge of clk after load has been asserted, staying there while load remains high (to illustrate this fact, load was kept high during two clock cycles). The other bits follow as soon as load returns to '0'. Notice that after all bits have been transmitted, the output stays low.

9.8 Playing with a Seven-Segment Display

We want to design a little game with an **SSD** (**s**even-**s**egment **d**isplay). The top-level diagram of the circuit is shown in figure 9.17. It contains two inputs, clk and stop, and one output, dout(6:0), which feeds the SSD. Assume that fclk =1 kHz.

Our circuit should cause a continuous clockwise movement of the SSD segments. Also, in order to make the circulatory movement more realistic, we want to momentarily

overlap neighboring segments. Consequently, the sequence should be a →ab →b →bc →c →cd →d →de →e →ef →f →fa →a, with the combined states (ab, bc, etc.) lasting only a few milliseconds. If stop is asserted, then the circuit should return to state a and remain so until stop is turned low again.

From chapter 8, it is clear that this is a circuit for which the FSM approach is appropriate. The states diagram is presented in figure 9.18. We want the system to remain in states a, b, c, etc. for time1 =80 ms, and in the combined states, ab, bc, etc., for time2 =30 ms. Therefore, a counter counting up to 80 (the clock period is 1 ms) or up to 30 can be employed to determine when to move to the next state.

A VHDL solution is shown below. Notice that it is a straight implementation of the FSM template seen in section 8.2. In lines 11–12, time1 and time2 were declared as two constants. Small values (4 and 2, respectively) were here used in order for the simulation results to fit well in one plot, but 80 and 30, respectively, were used in the actual physical implementation. A signal called flip was used to switch from time1 to time2, and vice-versa. Notice that the corresponding decimals are marked beside each value of dout, so they can be easily verified in the simulation results.

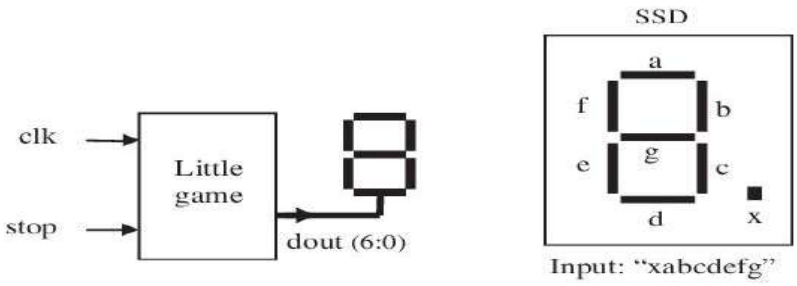


Figure 9.17
Playing with an SSD.

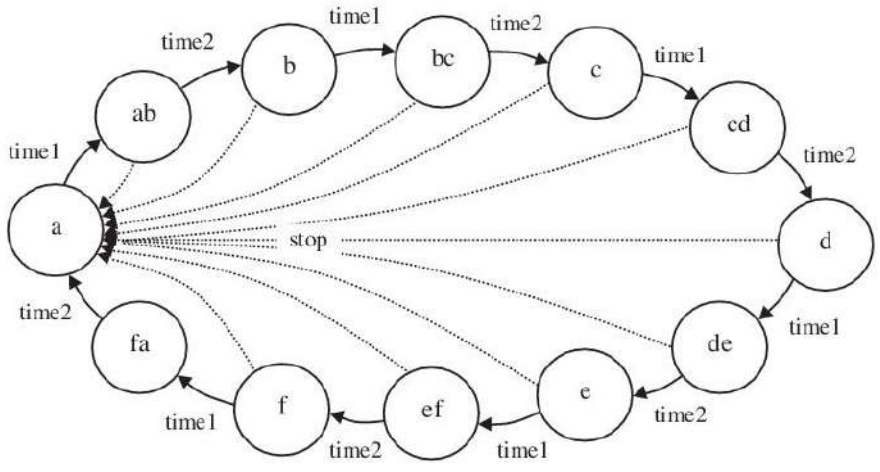


Figure 9.18
States diagram for the circuit of figure 9.17.

```
1 ----------------------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------------------------
5 ENTITY ssd_game2 IS
6 PORT ( clk, stop: IN BIT;
7 dout: OUT BIT_VECTOR (6 DOWNTO 0));
8 END ssd_game2;
9 ----------------------------------------------------
10 ARCHITECTURE fsm OF ssd_game2 IS
11 CONSTANT time1: INTEGER := 4; -- actual value is 80
12 CONSTANT time2: INTEGER := 2; -- actual value is 30
13 TYPE states IS (a, ab, b, bc, c, cd, d, de, e, ef, f, fa);
14 SIGNAL present_state, next_state: STATES;
15 SIGNAL count: INTEGER RANGE 0 TO 5;
16 SIGNAL flip: BIT;
17 BEGIN
18 ------- Lower section of FSM (Sec. 8.2): ------------
19 PROCESS (clk, stop)
20 BEGIN
21 IF (stop='1') THEN
22 present_state <= a;
23 ELSIF (clk'EVENT AND clk='1') THEN
24 IF ((flip='1' AND count=time1) OR
25 (flip='0' AND count=time2)) THEN
26 count <= 0;
27 present_state <= next_state;
28 ELSE count <= count + 1;
29 END IF;
30 END IF;
31 END PROCESS;
32 ------- Upper section of FSM (Sec. 8.2): ------------
33 PROCESS (present_state)
34 BEGIN
35 CASE present_state IS
36 WHEN a =>
37 dout <= "1000000"; -- Decimal 64
38 flip<='1';
39 next_state <= ab;
40 WHEN ab =>
41 dout <= "1100000"; -- Decimal 96
42 flip<='0';
43 next_state <= b;
44 WHEN b =>
45 dout <= "0100000"; -- Decimal 32
46 flip<='1';
47 next_state <= bc;
48 WHEN bc =>
```

```
49 dout <= "0110000"; -- Decimal 48
50 flip<='0';
51 next_state <= c;
52 WHEN c =>
53 dout <= "0010000"; -- Decimal 16
54 flip<='1';
55 next_state <= cd;
56 WHEN cd =>
57 dout <= "0011000"; -- Decimal 24
58 flip<='0';
59 next_state <= d;
60 WHEN d =>
61 dout <= "0001000"; -- Decimal 8
62 flip<='1';
63 next_state <= de;
64 WHEN de =>
65 dout <= "0001100"; -- Decimal 12
66 flip<='0';
67 next_state <= e;
68 WHEN e =>
69 dout <= "0000100"; -- Decimal 4
70 flip<='1';
71 next_state <= ef;
72 WHEN ef =>
73 dout <= "0000110"; -- Decimal 6
74 flip<='0';
75 next_state <= f;
76 WHEN f =>
77 dout <= "0000010"; -- Decimal 2
78 flip<='1';
79 next_state <= fa;
80 WHEN fa =>
81 dout <= "1000010"; -- Decimal 66
82 flip<='0';
83 next_state <= a;
84 END CASE;
85 END PROCESS;
86 END fsm;
87 ----------------------------------------------------------
```

Simulation results are presented in figure 9.19. As can be seen, the system stays in the single states, a, b, etc., for four clock cycles (time1 = 4 here) and in the combined states, ab, bc, etc., for two clock cycles (time2 = 2). Observe also that the decimals detected by the simulator match the decimals listed in the VHDL code.

9.9-Signal Generators

The signal of figure 9.20 can be modeled as an 8-state FSM. Using a counter from 0 to 7, we can establish that wave ¼'0' (1st pulse) when count ¼0, wave ¼'1' (2nd pulse) when count ¼1, and so on, thus creating the signal shown in the figure. This implementation requires a total of four flip-flops: three to store count (three bits), plus one to store wave (one bit). Recall from chapter 8, sections 8.2–8.3, that the output of a FSM will only be registered if design style #2 is employed, which is necessaryhere, because glitches are not acceptable in a signal generator.

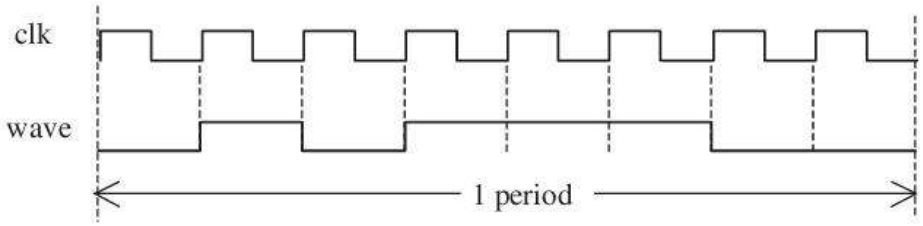


Figure 9.20
Signal generator problem.

The corresponding VHDL code, using design style #2 (section 8.3), is shown below. Simulation results appear in figure 9.21. Checking the report file created by the synthesis tool, we verify that a total of four flip-flops were indeed inferred from this code.

```
1 ----------------------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------------------------
5 ENTITY signal_gen IS
6 PORT (clk: IN STD_LOGIC;
7 wave: OUT STD_LOGIC);
8 END signal_gen;
9 ----------------------------------------------------
10 ARCHITECTURE fsm OF signal_gen IS
11 TYPE states IS (zero, one, two, three, four, five, six,
12 seven);
13 SIGNAL present_state, next_state: STATES;
14 SIGNAL temp: STD_LOGIC;
15 BEGIN
```

```
16
17 --- Lower section of FSM (Sec. 8.3): ---
18 PROCESS (clk)
19 BEGIN
20 IF (clk'EVENT AND clk='1') THEN
21 present_state <= next_state;
22 wave <= temp;
23 END IF;
24 END PROCESS;
25
26 --- Upper section of FSM (Sec. 8.3): ---
27 PROCESS (present_state)
28 BEGIN
29 CASE present_state IS
30 WHEN zero => temp<='0'; next_state <= one;
31 WHEN one => temp<='1'; next_state <= two;
32 WHEN two => temp<='0'; next_state <= three;
33 WHEN three => temp<='1'; next_state <= four;
34 WHEN four => temp<='1'; next_state <= five;
35 WHEN five => temp<='1'; next_state <= six;
36 WHEN six => temp<='0'; next_state <= seven;
37 WHEN seven => temp<='0'; next_state <= zero;
38 END CASE;
39 END PROCESS;
40 END fsm;
41 ------------------------------------------------
```

Conventional Approach

A conventional design, with the IF statement, is shown next. Notice that count and wave are both assigned at the transition of another signal (clk). Therefore, according to what you saw in section 7.5, both will be stored (that is, four flip-flops will be inferred,three for count and one for wave).
Simulation results are shown in figure 9.22.

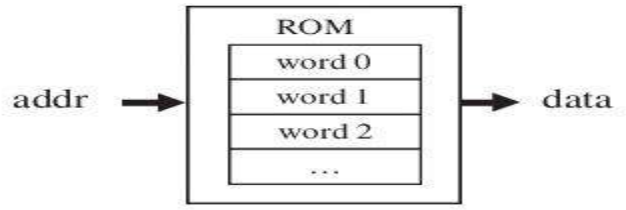


Figure 9.23
ROM diagram.

```
1 ---------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ---------------------------------------
5 ENTITY signal_gen1 IS
6 PORT (clk: IN BIT;
7 wave: OUT BIT);
8 END signal_gen1;
9 ---------------------------------------
10 ARCHITECTURE arch1 OF signal_gen1 IS
11 BEGIN
12 PROCESS
13 VARIABLE count: INTEGER RANGE 0 TO 7;
14 BEGIN
15 WAIT UNTIL (clk'EVENT AND clk='1');
16 CASE count IS
17 WHEN 0 => wave <= '0';
18 WHEN 1 => wave <= '1';
19 WHEN 2 => wave <= '0';
20 WHEN 3 => wave <= '1';
21 WHEN 4 => wave <= '1';
22 WHEN 5 => wave <= '1';
23 WHEN 6 => wave <= '0';
24 WHEN 7 => wave <= '0';
25 END CASE;
26 count := count + 1;
27 END PROCESS;
28 END arch1;
29 ---------------------------------------
```

9.10 Memory Design

In this section, the design of the following memory circuits is presented:
-ROM
-RAM with separate in/out data buses
- RAM with bidirectional in/out data bus

ROM (Read Only Memory)

Figure 9.23 shows the diagram of a ROM. Since it is a read-only memory, no clock signal or write-enable pin is necessary. As can be seen, the circuit contains a pile of pre-stored words, being the one selected by the address input (addr) presented at the output (data). In the code shown below, words (line 7) represents the number of words stored in the memory,
while bits (line 6) represents the size of each word. To create a ROM,
an array of CONSTANT values can be used (lines 15–22). First, a new TYPE, called

vector_array, was defined (lines 13–14), which was then used in the declaration of a CONSTANT named memory (line 15). An 8 *8 ROM is illustrated in this example, with the following (decimal) values stored in addresses 0 to 7: 0, 2, 4, 8, 16, 32, 64, and 128 (lines 15–22). Line 24 shows an example of call to the memory; the output (data) is equal to the word stored at address addr. When implementing a ROM, no data registers are inferred, because no signal assignment occurs at the transition of another signal. Logical gates forming an LUT (lookup table), are used instead.

```
1 ------------------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ------------------------------------------------
5 ENTITY rom IS
6 GENERIC ( bits: INTEGER := 8; -- # of bits per word
7 words: INTEGER := 8); -- # of words in the memory
8 PORT ( addr: IN INTEGER RANGE 0 TO words-1;
9 data: OUT STD_LOGIC_VECTOR (bits-1 DOWNTO 0));
10 END rom;
11 ------------------------------------------------
12 ARCHITECTURE rom OF rom IS
13 TYPE vector_array IS ARRAY (0 TO words-1) OF
14 STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
15 CONSTANT memory: vector_array := ( "00000000",
16 "00000010",
17 "00000100",
18 "00001000",
19 "00010000",
20 "00100000",
21 "01000000",
22 "10000000");
23 BEGIN
24 data <= memory(addr);
25 END rom;
26 ------------------------------------------------
```

Simulation results are shown in figure 9.24. As can be seen, the address changes from 0 to 7, then restarts from 0, with the outputs matching the values listed in the code above.

RAM with Separate Input and Output Data Buses

A RAM (Random Access Memory), with separate input and output data buses, is illustrated in figure 9.25.

Figure 9.25
RAM with separate in/out data buses.

As can be seen in figure 9.25(a), the circuit has a data input bus (data_in), a data output bus (data_out), an address bus (addr), plus clock (clk) and write enable (wr_ena) pins. When wr_enable is asserted, at the next rising edge of clk the vector present at data_in must be stored in the position specified by addr. data_out, on the other hand, must constantly display the data selected by addr.
From the register point-of-view, the circuit can be summarized as in figure 9.25(b). When wr_ena is low, q is connected to the input of the flip-flop, and terminal d is open, so no new data will be written into the memory. However, when wr_ena is turned high, d is connected to the input of the register, so at the next rising edge of clk d will be stored.
A VHDL code that implements the circuit of figure 9.25 is shown below. The chosen capacity was 16 words of length eight bits each. Notice that the code is totally generic. Simulation results are shown in figure 9.26.

```
1 --------------------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 --------------------------------------------------
5 ENTITY ram IS
6 GENERIC ( bits: INTEGER := 8; -- # of bits per word
7 words: INTEGER := 16); -- # of words in the
8 -- memory
9 PORT ( wr_ena, clk: IN STD_LOGIC;
10 addr: IN INTEGER RANGE 0 TO words-1;
11 data_in: IN STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
12 data_out: OUT STD_LOGIC_VECTOR (bits-1 DOWNTO 0));
13 END ram;
14 --------------------------------------------------
15 ARCHITECTURE ram OF ram IS
16 TYPE vector_array IS ARRAY (0 TO words-1) OF
17 STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
18 SIGNAL memory: vector_array;
19 BEGIN
20 PROCESS (clk, wr_ena)
21 BEGIN
22 IF (wr_ena='1') THEN
23 IF (clk'EVENT AND clk='1') THEN
24 memory(addr) <= data_in;
25 END IF;
26 END IF;
27 END PROCESS;
28 data_out <= memory(addr);
29 END ram;
30 --------------------------------------------------
```

RAM with Bidirectional In/Out Data Bus

A RAM with bidirectional in/out data bus is illustrated in figure 9.27. The overall structure is similar to that of figure 9.25, except for the fact that now the same bus (bidir) is used to write data into the memory as well to read data from it.
From the register point-of-view, the circuit can be summarized as in figure 9.27(b). When wr-ena is low, the output of the register is connected to its input, so no change on the store data will occur. On the other hand, when wr-ena is asserted, q is connected to d, allowing new data to be stored at the next rising edge of clk.
A VHDL code that implements the circuit of figure 9.27 is shown below. The chosen capacity was 16 words of length eight bits each. Notice that this code is also totally generic. Simulation results are shown in figure 9.28.

```
1 ------------------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ------------------------------------------------
5 ENTITY ram4 IS
6 GENERIC ( bits: INTEGER := 8; -- # of bits per word
7 words: INTEGER := 16); -- # of words in the
8 -- memory
9 PORT ( clk, wr_ena: IN STD_LOGIC;
10 addr: IN INTEGER RANGE 0 TO words-1;
11 bidir: INOUT STD_LOGIC_VECTOR (bits-1 DOWNTO 0));
12 END ram4;
13 ------------------------------------------------
14 ARCHITECTURE ram OF ram4 IS
15 TYPE vector_array IS ARRAY (0 TO words-1) OF
16 STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
17 SIGNAL memory: vector_array;
18 BEGIN
19 PROCESS (clk, wr_ena)
20 BEGIN
21 IF (wr_ena='0') THEN
22 bidir <= memory(addr);
23 ELSE
24 bidir <= (OTHERS => 'Z');
25 IF (clk'EVENT AND clk='1') THEN
26 memory(addr) <= bidir;
27 END IF;
28 END IF;
29 END PROCESS;
```

```
30 END ram;
31 ------------------------------------------------
```

10-Packages and Components

Packages and Components Introduction we will simply add new building blocks to the material already presented. These new building blocks are intended mainly for library allocation, being shown on the right-hand side of figure 10.1. They are: „h Packages „h Components „h Functions „h Procedures

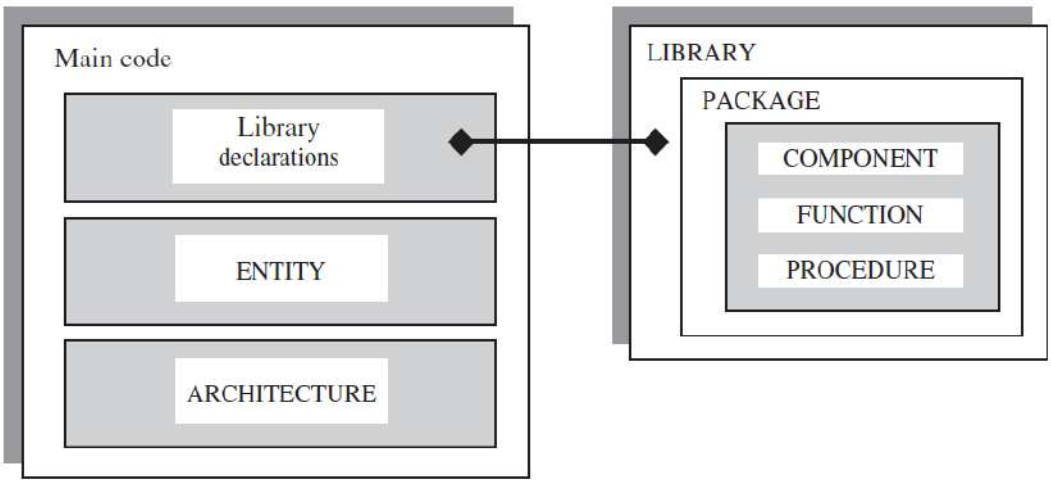


Figure 10.1: Fundamental units of VHDL code.

PACKAGE

frequently used pieces of VHDL code are usually written in the form of COMPONENTS, FUNCTIONS, or PROCEDURES. Such codes are then placed inside a PACKAGE and compiled into the destination LIBRARY. The importance of this technique is that it allows code partitioning, code sharing, and code reuse.

Packages syntax is presented below. the syntax is composed of two parts: PACKAGE and PACKAGE BODY. The first part is mandatory and contains all declarations, while the second part is necessary only when one or more subprograms (FUNCTION or PROCEDURE) are declared in the upper part, in which case it must contain the descriptions (bodies) of the subprograms. PACKAGE and PACKAGE BODY 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;]
```

The declarations list can contain the following: COMPONENT, FUNCTION, PROCEDURE, TYPE, CONSTANT, etc.

Example 10.1: Simple Package

The example below shows a PACKAGE called my_package. It contains only TYPE and CONSTANT declarations, so a PACKAGE BODY is not necessary

```
1  ------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------
5  PACKAGE my_package IS
6     TYPE state IS (st1, st2, st3, st4);
7     TYPE color IS (red, green, blue);
8     CONSTANT vec: STD_LOGIC_VECTOR(7 DOWNTO 0) := "11111111";
9  END my_package;
10 ------------------------------------------------
```

Example 10.2: Package with a Function

This example contains, besides TYPE and CONSTANT declarations, a FUNCTION. Therefore, a PACKAGE BODY is now needed. This function returns TRUE when a positive edge occurs on clk.

```
1  ------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------
5  PACKAGE my_package IS
6     TYPE state IS (st1, st2, st3, st4);
7     TYPE color IS (red, green, blue);
8     CONSTANT vec: STD_LOGIC_VECTOR(7 DOWNTO 0) := "11111111";
9     FUNCTION positive_edge(SIGNAL s: STD_LOGIC) RETURN BOOLEAN;
10 END my_package;
```

```
11 ---------------------------------------------
12 PACKAGE BODY my_package IS
13    FUNCTION positive_edge(SIGNAL s: STD_LOGIC) RETURN BOOLEAN IS
14    BEGIN
15       RETURN (s'EVENT AND s='1');
16    END positive_edge;
17 END my_package;
18 ---------------------------------------------
```

Any of the PACKAGES above (example 10.1 or example 10.2) can now be compiled, becoming then part of our work LIBRARY (or any other). To make use of it in a VHDL code, we have to add a new USE clause to the main code (USE work .my_ package .all), as shown below.

```
-----------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.my_package.all;
-----------------------------------
ENTITY...
...
ARCHITECTURE...
...
-----------------------------------
```

COMPONENT

A COMPONENT is simply a piece of conventional code (that is, LIBRARY

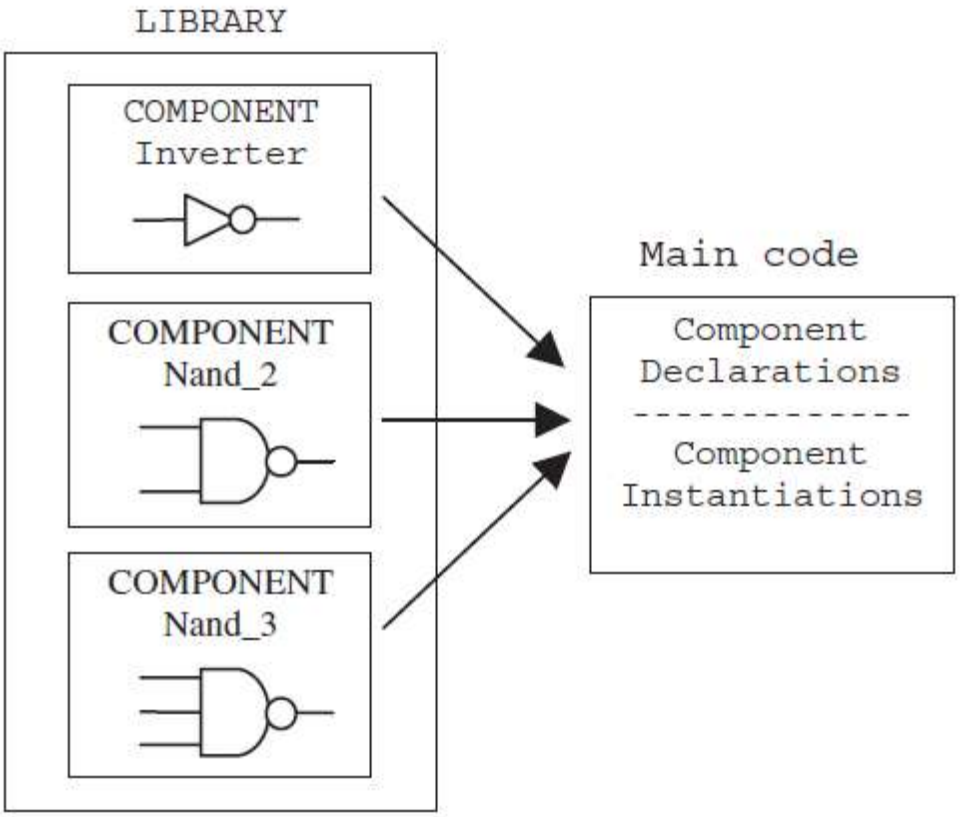
declarations + ENTITY + ARCHITECTURE). However, by declaring such code as being a COMPONENT, it can then be used within another circuit, thus allowing the construction of hierarchical designs A COMPONENT is also another way of partitioning a code and providing code sharing and code reuse. For example, commonly used circuits, like flip-flops, multiplexers ,adders, basic gates, etc., can be placed in a LIBRARY, so any project can make use of them without having to explicitly rewrite such codes. To use (instantiate) a COMPONENT, it must first be declared. The corresponding syntaxes are shown below COMPONENT declaration:

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

COMPONENT instantiation:

```
label: component_name PORT MAP (port_list);
```

Example 10.3: Components Declared in the Main Code We want to implement the circuit of figure 10.3 employing only COMPONENTS (inverter, nand_2, and nand_3), but without creating a specific PACKAGE to declare them thus as in figure 10.2(a).



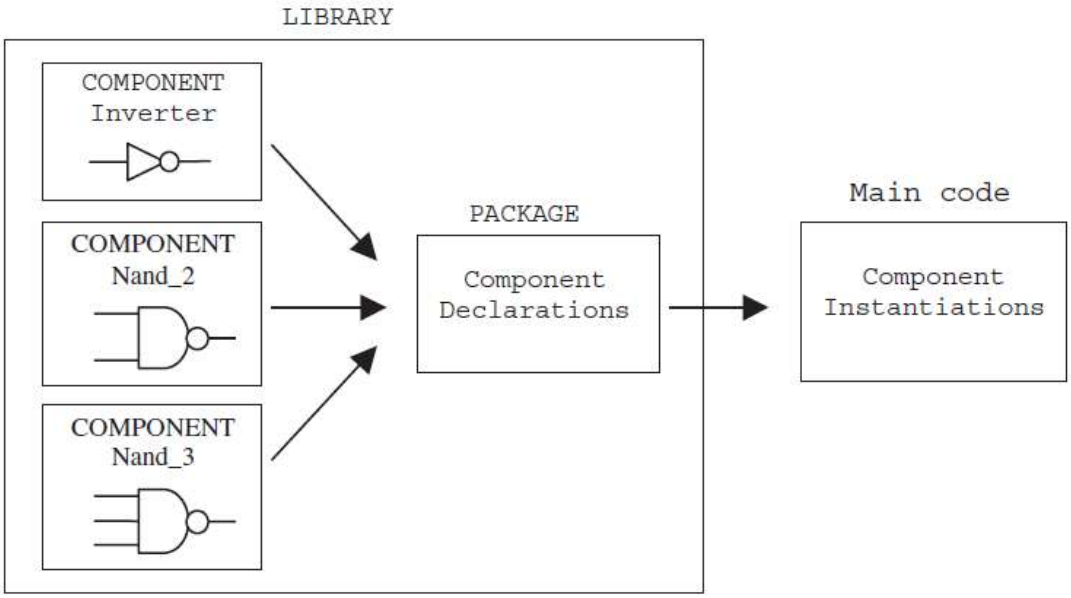


Figure 10.2: Basic ways of declaring COMPONENTS: (a) declarations in the main code itself, (b) declarations in a PACKAGE

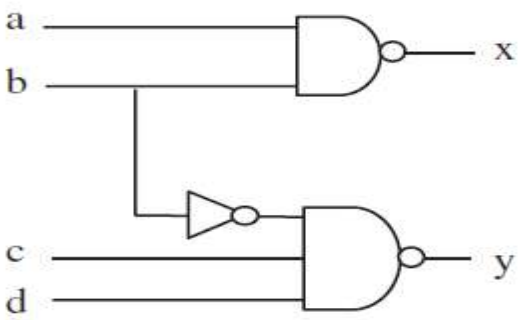


Figure 10.3: Circuit of example 10.3.

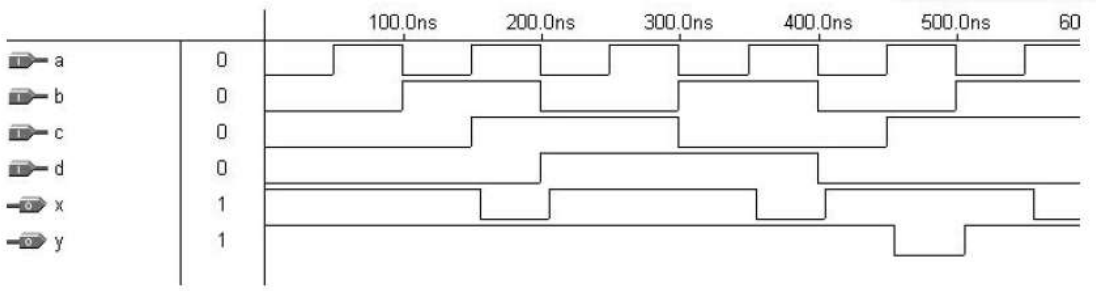
```
1  ------ File inverter.vhd: -------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------
5  ENTITY inverter IS
6     PORT (a: IN STD_LOGIC; b: OUT STD_LOGIC);
7  END inverter;
8  ------------------------------------
9  ARCHITECTURE inverter OF inverter IS
10 BEGIN
11    b <= NOT a;
12 END inverter;
13 ---------------------------------------------
```

```
1  ------ File nand_2.vhd: --------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------
5  ENTITY nand_2 IS
6     PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
7  END nand_2;
8  ------------------------------------
9  ARCHITECTURE nand_2 OF nand_2 IS

10 BEGIN
11    c <= NOT (a AND b);
12 END nand_2;
13 ---------------------------------------------
```

```
1  ----- File nand_3.vhd: ---------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------
5  ENTITY nand_3 IS
6     PORT (a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
7  END nand_3;
8  ------------------------------------
9  ARCHITECTURE nand_3 OF nand_3 IS
10 BEGIN
11    d <= NOT (a AND b AND c);
12 END nand_3;
13 ----------------------------------------------
```

```
1  ----- File project.vhd: --------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------
5  ENTITY project IS
6     PORT (a, b, c, d: IN STD_LOGIC;
7           x, y: OUT STD_LOGIC);
8  END project;
9  ------------------------------------
```

```
10 ARCHITECTURE structural OF project IS
11     -------------
12     COMPONENT inverter IS
13        PORT (a: IN STD_LOGIC; b: OUT STD_LOGIC);
14     END COMPONENT;
15     -------------
16     COMPONENT nand_2 IS
17        PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
18     END COMPONENT;
19     -------------
20     COMPONENT nand_3 IS
21        PORT (a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
22     END COMPONENT;
23     -------------

24      SIGNAL w: STD_LOGIC;
25  BEGIN
26      U1: inverter PORT MAP (b, w);
27      U2: nand_2 PORT MAP (a, b, x);
28      U3: nand_3 PORT MAP (w, c, d, y);
29  END structural;
30  ------------------------------------------
```



.Figure 10.4: Experimental results of example 10.3

Example 10.4: Components Declared in a Package We want to implement the same project of the previous example (figure 10.3) However, we will now create a PACKAGE where all the COMPONENTS (inverter ,nand_2, and nand_3) will be declared, like in figure 10.2(b).

```
5  ENTITY inverter IS
6     PORT (a: IN STD_LOGIC; b: OUT STD_LOGIC);
7  END inverter;
8  ------------------------------------
9  ARCHITECTURE inverter OF inverter IS
10 BEGIN
11    b <= NOT a;
12 END inverter;
13 ---------------------------------------------
```

```
1  ------ File nand_2.vhd: --------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------
5  ENTITY nand_2 IS
6     PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
7  END nand_2;
8  ------------------------------------
9  ARCHITECTURE nand_2 OF nand_2 IS
10 BEGIN
11    c <= NOT (a AND b);
12 END nand_2;
13 ---------------------------------------------
```

```
1  ----- File nand_3.vhd: ---------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------
```

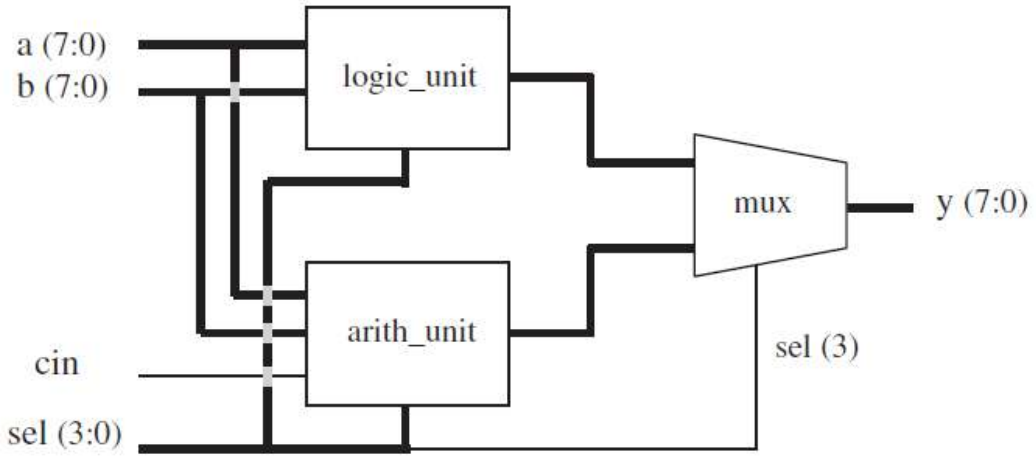
```
5  ENTITY nand_3 IS
6     PORT (a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
7  END nand_3;
8  ------------------------------------
9  ARCHITECTURE nand_3 OF nand_3 IS
10 BEGIN
11    d <= NOT (a AND b AND c);
12 END nand_3;
13 ------------------------------------------
```

```
1  ----- File my_components.vhd: ---------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -------+-----------------
5  PACKAGE my_components IS
6     ------ inverter: -------
7     COMPONENT inverter IS
8        PORT (a: IN STD_LOGIC; b: OUT STD_LOGIC);
9     END COMPONENT;
10    ------ 2-input nand: ---
11    COMPONENT nand_2 IS
12       PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
13    END COMPONENT;
14    ------ 3-input nand: ---
15    COMPONENT nand_3 IS
16       PORT (a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
17    END COMPONENT;
18    ------------------------
19 END my_components;
20 -------------------------------------------
```

```
1  ----- File project.vhd: --------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE work.my_components.all;
5  --------------------------------
6  ENTITY project IS
7     PORT ( a, b, c, d: IN STD_LOGIC;
8            x, y: OUT STD_LOGIC);

9  END project;
10 --------------------------------
11 ARCHITECTURE structural OF project IS
12    SIGNAL w: STD_LOGIC;
13 BEGIN
14    U1: inverter PORT MAP (b, w);
15    U2: nand_2 PORT MAP (a, b, x);
16    U3: nand_3 PORT MAP (w, c, d, y);
17 END structural;
18 ------------------------------------------
```

Example 10.5: ALU Made of COMPONENTS In the present example, however, we will assume that our library contains the three components (logic_unit, arith_unit, and mux) with which the ALU can be constructed

```
1  -------- COMPONENT arith_unit: --------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_unsigned.all;
5  ----------------------------------------
6  ENTITY arith_unit IS
7     PORT ( a, b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
8            sel: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
9            cin: IN STD_LOGIC;
10           x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
11 END arith_unit;
```



sel	Operation	Function	Unit
0000	y <= a	Transfer a	
0001	y <= a+1	Increment a	
0010	y <= a-1	Decrement a	
0011	y <= b	Transfer b	Arithmetic
0100	y <= b+1	Increment b	
0101	y <= b-1	Decrement b	
0110	y <= a+b	Add a and b	
0111	y <= a+b+cin	Add a and b with carry	
1000	y <= NOT a	Complement a	
1001	y <= NOT b	Complement b	
1010	y <= a AND b	AND	
1011	y <= a OR b	OR	Logic
1100	y <= a NAND b	NAND	
1101	y <= a NOR b	NOR	
1110	y <= a XOR b	XOR	
1111	y <= a XNOR b	XNOR	

١١

Figure 10.5: ALU constructed from three COMPONENTS.

```
12 ----------------------------------------
13 ARCHITECTURE arith_unit OF arith_unit IS
14    SIGNAL arith, logic: STD_LOGIC_VECTOR (7 DOWNTO 0);
15 BEGIN
16    WITH sel SELECT
17       x <=  a WHEN "000",
18             a+1 WHEN "001",
19             a-1 WHEN "010",
20             b WHEN "011",
21             b+1 WHEN "100",
22             b-1 WHEN "101",
23             a+b WHEN "110",
24             a+b+cin WHEN OTHERS;
25 END arith_unit;
26 ----------------------------------------------------
```

```
1  -------- COMPONENT logic_unit: --------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY logic_unit IS
6     PORT ( a, b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7            sel: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
8            x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
9  END logic_unit;
10 ----------------------------------------

11 ARCHITECTURE logic_unit OF logic_unit IS
12 BEGIN
13    WITH sel SELECT
14       x <=  NOT a WHEN "000",
15             NOT b WHEN "001",
16             a AND b WHEN "010",
17             a OR b WHEN "011",
18             a NAND b WHEN "100",
19             a NOR b WHEN "101",
20             a XOR b WHEN "110",
21             NOT (a XOR b) WHEN OTHERS;
22 END logic_unit;
23 ------------------------------------------------
```

```
1  -------- COMPONENT mux: --------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY mux IS
6     PORT ( a, b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7            sel: IN STD_LOGIC;
8            x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
9  END mux;
10 ----------------------------------------
11 ARCHITECTURE mux OF mux IS
12 BEGIN
13    WITH sel SELECT
14       x <=   a WHEN '0',
15              b WHEN OTHERS;
16 END mux;
17 ------------------------------------------------
```

```
1  -------- Project ALU (main code): ----------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY alu IS
6     PORT ( a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
7            cin: IN STD_LOGIC;
8            sel: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
9            y: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
10 END alu;
11 ----------------------------------------
```

```
11 ----------------------------------------
12 ARCHITECTURE alu OF alu IS
13 -----------------------
14    COMPONENT arith_unit IS
15    PORT ( a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
16           cin: IN STD_LOGIC;
17           sel: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
18           x: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
19    END COMPONENT;
20    -----------------------
21    COMPONENT logic_unit IS
22    PORT ( a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
23           sel: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
24           x: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
25    END COMPONENT;
26    -----------------------
27    COMPONENT mux IS
28    PORT ( a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
29           sel: IN STD_LOGIC;
30           x: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
31    END COMPONENT;
32    -----------------------
33    SIGNAL x1, x2: STD_LOGIC_VECTOR(7 DOWNTO 0);
34 -----------------------
35 BEGIN
36    U1: arith_unit PORT MAP (a, b, cin, sel(2 DOWNTO 0), x1);
37    U2: logic_unit PORT MAP (a, b, sel(2 DOWNTO 0), x2);
38    U3: mux PORT MAP (x1, x2, sel(3), y);
39 END alu;
```

Simulation results are shown in figure 10.6.

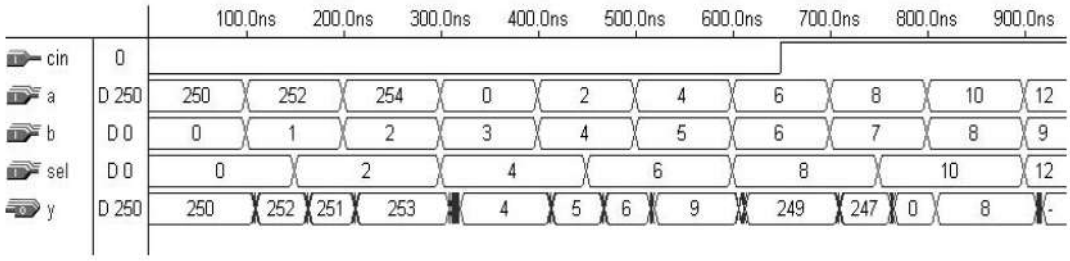


Figure 10.6: Simulation results of example 10.6.

Reference :Circuit Design with VHDL

Volnei A. Pedroni(Massachusetts Institute of Technology)