Introduction to VHDL programming

Juan Antonio Clemente
Translation to the English of the material written by: Marcos Sánchez-Élez
(Introducción a la programación en VHDL)

November 13, 2014
# Contents

1 Introduction 1

2 Basic Elements of VHDL 3
   2.1 Entity .................................................. 3
   2.2 Architecture ........................................... 4
   2.3 VHDL objects ........................................... 5
   2.4 VHDL types ............................................. 6
      2.4.1 Predefined types .................................... 6
      2.4.2 User-defined types .................................. 7
      2.4.3 Examples ........................................... 8
      2.4.4 Operators .......................................... 9

3 Basic Structure of a Source File in VHDL 11
   3.1 Concurrent statements ................................. 11
      3.1.1 WHEN-ELSE .......................................... 11
      3.1.2 WITH-SELECT-WHEN ................................ 12
   3.2 Conditional statements ............................... 13
      3.2.1 IF-THEN-ELSE ...................................... 13
      3.2.2 CASE-WHEN .......................................... 13
      3.2.3 FOR-LOOP .......................................... 14
      3.2.4 WHILE-LOOP ...................................... 14
   3.3 Process statement .................................... 15
   3.4 Structural description ............................... 19
   3.5 Examples .................................................. 22

4 Simulation of a VHDL Code 25
   4.1 Steps of simulation .................................... 25
CONTENTS

4.2 Simulation statements .................................................. 26
4.3 Simulation templates in VHDL ........................................... 27

5 Description of Sequential Logic ....................................... 31
  5.1 Sequential hardware .................................................. 31
  5.2 Counters .............................................................. 34
  5.3 Examples ............................................................. 37
    5.3.1 8-bit register ................................................... 37
    5.3.2 8-bit register built using 1-bit flip-flops ..................... 37

6 Design of a Finite State Machine (FSM) ................................. 39

7 Functions, Procedures and Packages ................................... 43
  7.1 Functions ........................................................... 43
  7.2 Procedures ......................................................... 44
  7.3 Examples .......................................................... 44

8 Design of a RAM Memory .................................................. 47

9 Appendixes ........................................................................ 49
  9.1 Discussion about using signals vs. variables ....................... 49
  9.2 Discussion about the effect of incorrectly coding the sensitivity list in a process ............................. 50
Chapter 1

Introduction

VHDL is a description language for digital electronic circuits that is used in different levels of abstraction. The VHDL acronym stands for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language. This means that VHDL can be used to accelerate the design process.

It is very important to point out that VHDL is NOT a programming language. Therefore, knowing its syntax does not necessarily mean being able to designing digital circuits with it. VHDL is an HDL (Hardware Description Language), which allows describing both asynchronous and synchronous circuits. For this purpose, we shall:

- Think in terms of gates and flip-flops, not in variables or functions.
- Avoid combinatorial loops and conditional clocks.
- Know which part of the circuit is combinatorial and which one is sequential.

Why to use an HDL?

- To discover problems and faults in the design before actually implementing it in hardware.
- The complexity of an electronic system grows exponentially. For this reason, it is very convenient to build a prototype of the circuit previously to its manufacturing process.
- It makes easy for a team of developers to work together.

In particular, VHDL allows not only describing the structure of the circuit (description from more simple subcircuits), but also the specification of the functionality of a circuit using directives, in a similar way as most standard programming languages do.

The most important aim of an HDL is to be able to simulate the logical behavior of a circuit by means of a description language that has many similarities with software description languages.

Digital circuits described in VHDL can be simulated using simulation tools that reproduce the operation of the involved circuit. For this purpose, developers use a set of rules standardized by the IEEE, which explain the syntax of the language, as well as how to simulate it. In addition, there are many tools that transform a VHDL code into a downloadable file that can be used to program a reconfigurable device. This process is named synthesis. The way a given tool carries out the synthesis process is very particular, and it greatly differs from what other synthesis tools do.
Chapter 1. Introduction

For Xilinx™ users: In this manual we will use the free synthesis tool provided by Xilinx® (Xilinx ISE Web Pack), which can be obtained through the following URL: http://www.xilinx.com/support/download/index.htm. All the examples in this manual that may include any coding that is specific from the Xilinx tool will be highlighted in a box like this one.

**TIP:** Throughout this manual, boxes like this one will be used to better highlight tips for an efficient programming in VHDL. These tips are a set of basic rules that make the simulation results independent of the programming style. Hence, these rules make the developed code synthesizable, so it can be easily implemented in any platform.

Webs and news related to VHDL programming and its simulation and synthesis tools:

- [www.edacafe.com](http://www.edacafe.com): Web page dedicated to spread news related to the world of circuit design. It has a forum of VHDL programming (troubleshooting, free tools ...).

- [www.eda.org/vasg/](http://www.eda.org/vasg/): “Welcome to the VHDL Analysis and Standardization Group (VASG). The purpose of this web site is to enhance the services and communications between members of the VASG and users of VHDL. We’ve provided a number of resources here to help you research the current and past activities of the VASG and report language bugs, LRM ambiguities, and suggest improvements to VHDL...”

- [www.cadence.com](http://www.cadence.com): “Cadence Design Systems is the world’s largest supplier of EDA technologies and engineering services. Cadence helps its customers break through their challenges by providing a new generation of electronic design solutions that speed advanced IC and system designs to volume...”

- [www.xilinx.com](http://www.xilinx.com): “In the world of digital electronic systems, there are three basic kinds of devices: memory, microprocessors, and logic. Memory devices store random information such as the contents of a spreadsheet or database. Microprocessors execute software instructions to perform a wide variety of tasks such as running a word processing program or video game. Logic devices provide specific functions, including device-to-device interfacing, data communication, signal processing, data display, timing and control operations, and almost every other function a system must perform.”
Chapter 2

Basic Elements of VHDL

A digital system is basically described by its inputs and its outputs, as well as how these outputs are obtained from the inputs.

The VHDL code of any circuit is divided into two separate parts: On the one hand, the entity specifies the input and output ports of the circuit. On the other hand, the architecture describes the behavior of that circuit. An architecture must be associated with an entity. It is also possible to associate several architectures to the same entity, so the programmer can select one of the available ones. This point is explained below in Chapter 2, Section 2.3.

For Xilinx™ users: The IEEE library and the following three packets (whose meaning is explained below) appear by default in any source VHDL code created with the Xilinx™ ISE tool.

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

2.1 Entity

An entity is an abstraction of a circuit, either from a complex electronic system or a single logic gate. An entity externally describes the I/O interface of the circuit.

The ports of an entity can be inputs (in), outputs (out), input-outputs (inout) or buffer. The input ports can only be read, and they cannot be modified inside the architecture. On the other hand, the output ports can only be written, but not read. In case an output port needs to be read (for instance, to make a decision about its value) or an input port needs to be written, they must be instantiated as an inout or a buffer port. However, in this course we will try to avoid these situations, so the utilization of inout and buffer ports are beyond the learning outcomes of this course.

The interface described by an entity may also include a set of generic values that are used to declare properties and constants of the circuits, independently of its architecture. Generics can have multiple uses: On the one hand, they can be used to define delays in signals and clock cycles (these definitions will not be taken into account at the synthesis level, as explained later throughout this manual). On the other hand, generics can also be used as constants that will be used inside the architecture. These
constants help to make the code more understandable, portable and maintainable. For instance, the length of a register (in number of bits) can be defined by means of a generic parameter. This means that another VHDL code can instantiate this register several times, even if this code instantiates registers with different number of bits. Generic parameters are not necessary. Hence, a circuit that does not need them, it simply does not instantiate any generic statement in the entity declaration.

The example below shows a description of the entity of a circuit. This circuit has two N-bit inputs (A and B) and a single output (Y). Thus, in this case the entity description includes a generic statement defining a parameter named N whose value is set to 8. This parameter is also used in the declaration of the circuit inputs.

```
entity F is
  generic (N: natural := 8);
  port (A, B: in bit_vector (N-1 downto 0); Y: out bit);
end F;
```

### 2.2 Architecture

The pairs `entity-architecture` are used in VHDL to completely describe the operation of a circuit. An architecture defines how the circuit operates, by including a set of inner signals, functions, procedures, functions... and its description can be either structural or behavioral (details about this will be given in Chapter 3.4).

The code below shows an example of an architecture. The association between this architecture and the entity it refers to is made in the first line (`architecture arch_name of entity_name is`). Next, the code must include the signals, customized types, and components (whose I/O is known) that will be used inside the architecture.

```
architecture arch_name of entity_name is
  -- architecture declarations:
  -- types
  -- signals
  -- components
  begin
  -- concurrent statements
  -- conditional statements
  -- components
  process (sensitivity list) begin
  -- code
  end process;
end arch_name;
```

The `begin` and the `end` reserved words mark the boundaries of the VHDL code that will actually describe the operation of the circuit. As shown in the example, this code may include: concurrent and
conditional statements, components and processes. Chapter 3 will get into deeper details about these statements.

It is also possible to define several architectures for the same entity. This is better explained in Chapter 9, Section ??.

2.3 VHDL objects

VHDL source codes can include objects. There are three types of objects:

- **Constant**: Objects that have an initial value that is assigned before the simulation. This value shall never be modified during the synthesis or the operation of the circuit. They can be declared before the begin of an architecture, and/or before the begin of a process. A constant declaration **MUST** assign a value to it.

  \[
  \text{constant identifier: type := value;} \\
  \]

- **Variable**: Objects that take a single value that can change during the simulation/execution by means of an assignment statement. Variables are usually used as indexes, mainly in loops, or to take values that allow to model other components. Variables **DO NOT** represent physical connections or memory elements. They can be declared before the begin of an architecture, and/or before the begin of a process. A variable declaration **MAY** or **MAY NOT** assign a value to it.

  \[
  \text{variable identifier: type [:= value];} \\
  \]

  The assignment of a value to a variable is done by means of the operator :=

  \[
  \text{name_variable := value;} \\
  \text{i := 10;} \\
  \]

- **Signal**: Objects that represent memory elements or connections between subcircuits. Contrarily to constants and variables, signals can be synthesized. In other words, a signal in a VHDL source code can be physically translated into a memory element (flip-flop, register...) in the final circuit. They must be declared before the begin of the architecture. The ports of an entity are implicitly declared as signals upon declaration, since they represent physical connections in the circuit.

  \[
  \text{signal identifier: type;} \\
  \]

  The assignment of a value to a signal is done by means of the operator <=

  \[
  \text{: name_signal <= value;} \\
  \text{A <= 10;} \\
  \]

**TIP**: If the developed VHDL code only uses constant and signal objects, it will not show any malign effect in the operation of the circuit (see Chapter 9, Sections 9.1 and 9.2). In addition, the obtained code will be easily portable to any other tool. For this reason, unless otherwise stated, all the objects referenced in this manual will be signals.
Chapter 2. Basic Elements of VHDL

2.4 VHDL types

In the previous definitions, as well as in the definition of the entity ports, it is necessary to define the type of the object. VHDL allows to use predefined types, as well as other user-defined ones.

2.4.1 Predefined types

The most commonly used predefined types are the following ones:

- **bit**: It only admits the values 0 and 1. In order to make an assignment between the object and its value, the latter must be written between single quotes (‘0’ or ‘1’).

- **bit_vector (range)**: The range, always written between brackets, indicates the number of bits of the bit_vector, which is an array of 0’s and 1’s. For an n-bit bit_vector, its range must be written in the format N-1 downto 0. The bit located to the far left is the most significant one (Most Significant Bit, or MSB), whereas the bit located to the far right is the least significant one (Least Significant Bit or LSB). In order to make an assignment between the object and its value, the latter must be written between quotation marks (i.e., “0011”).

- **boolean**: It only can take the values true or false.

- **character**: It can take any ASCII value.

- **string**: Any chain consisting of ASCII characters.

- **integer range**: Any integer number within the range, which in this case is not written between brackets. For instance, 0 to MAX. The range is optional.

- **natural range**: Any natural number within the range. The range is optional.

- **positive range**: Any positive number within the range. The range is optional.

- **real range**: Any real number within the range. The range is optional.

- **std_logic**: Type predefined in the IEEE 1164 standard. This type represents a multivalued logic comprising 9 different possible values. The most commonly used ones are: '0', '1', 'Z' (for high impedance), 'X' (for uninitialized) and 'U' (for undefined), among others. In order to make an assignment between the object and its value, the latter must be written between single quotes (‘0’, ‘1’, ‘X’, ‘...’).

- **std_logic_vector (range)**: It represents a vector of elements of type std_logic. Its assignment and definition rules are the same ones as the std_logic ones.

For Xilinx™ users: For Xilinx™ ISE, all the ports of the entity must be of type std_logic or std_logic_vector. The reason is that these two types allow simulating a circuit realistically. For instance, when a signal is instantiated but never initialized in the VHDL code, it will always take the ‘U’ (undefined) value. In addition, Xilinx™ ISE translates natural and integer signals into std_logic_vector with the number of bits needed for its complete representation.

In order to use the type std_logic, it is necessary to include the following library:
VHDL types 2.4

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

In order to use the pre-defined arithmetic and logic functions:

```
use ieee.std_logic_arith.all;
```

For vectors that are represented as unsigned binary:

```
use ieee.std_logic_unsigned.all;
```

For vectors that are represented as signed binary:

```
use ieee.std_logic_unsigned.all;
```

For vectors that are represented in 2’s complement:

```
use ieee.std_logic_signed.all;
```

**TIP:** It is strongly recommended to always use the std_logic_vector type independently of the operations that will be made on the involved objects. They can be used as integers or naturals thanks to the ieee.std_logic_arith.all and ieee.std_logic_unsigned.all libraries. Defining all the signals in the code as std_logic or std_logic_vector does not complicate the final VHDL code and helps a lot in its integration with Xilinx™ISE.

### 2.4.2 User-defined types

An enumerated type is a data type that comprises a number of user-defined values. Enumerated types are used mainly for the definition of finite state machines.

```
type name is (value1, value2, ...);
```

Assuming that `A` has been defined as an enumerated type, the assignment will be as follows: `A <= value1;` where `value1` must be one of the enumerated values in the type definition.

Enumerated types are sorted according to their values. Typically, synthesis tools automatically code the enumerated values in such a way that they can be further synthesized. For that purpose, they usually select an ascending sequence or a coding that minimizes the circuit or that maximizes its operating frequency. It may also be possible to directly type the coding by means of ad-hoc directives.

A composed type is a data type comprised by elements of other data types. Composed types can be either arrays and records.

- An **array** is a data object that comprises a set of elements of the same type.

```
type name is array (range) of type;
```
The assignment of a value on a position of the array is done by means of integer numbers (see examples at Subsection 2.4.3).

- A record is a data object that comprises a set of elements of different types.

```vhdl
type name is record
  element1: data_type1;
  element2: data_type2;
end record;
```

The assignment of a value on an element from a record is done by means of a dot (see examples at Chapter 4, Subsection 4.2.3).

Once defined the composed and/or enumerated data type, any signal in the design can be declared of belonging to this new type and this will be done by using the operator defined for signals <=.

### 2.4.3 Examples

This subsection presents some examples showing how to define and to assign values to signals and variables.

```vhdl
-- Two dashes are used to introduce comments in the VHDL code
-- Examples of definitions and assignments
constant DATA_WIDTH: integer := 8;
signal CTRL: bit_vector(7 downto 0);
variable SIG1, SIG2: integer range 0 to 15;
type color is (red, yellow, blue);
signal BMP: color;
  BMP <= red;
type word is array (0 to 15) of std_logic_vector (7 downto 0);
signal w: word;
-- w(integer/natural) <= vector of bits;
w(0) <= "00111110";
w(1) <= "00011010";
...  
w(15) <= "111111110";
type matrix is array (0 to 15)(7 downto 0) of std_logic;
signal m: matrix;
  m(2)(5) <= '1';
type set is record
  word: std_logic_vector (0 to 15);
  value: integer range -256 to 256;
end record;
signal data: set;
data.value <= 176;
```
2.4.4 Operators

Operators can be used to build a wide variety of expressions that allow to calculate data and/or to assign them to signals or variables.

- `+`, `-`: Sign change.
- `&`: Concatenation.
- `and`, `or`, `nand`, `nor`, `xor`: Logical operations.
- `:=`: Value assignment to constants and variables.
- `<=`: Value assignment to signals.

--- Assignment examples

```vhdl
y <= (x and z) or d(0);
y(1) <= x and not z;
y <= x1 & x2;  -- y = "x1x2"
c := 27 + r;
```

Chapter 3

Basic Structure of a Source File in VHDL

As previously pointed out, the VHDL code modeling a digital circuit is composed of two parts: an entity and one or several architectures. The latter contains the statements describing the behavior of the circuit.

```
architecture circuit of name is
  -- signals
  begin
    -- concurrent statements (assignment statements to signals)
    process (sensitivity list) begin
      -- conditional statements (assignment statements to variables)
      end process;
  end architecture circuit;
```

Inside the architecture, we can find:

- Types and intermediate signals needed to describe its behavior.
- Assignment statements to signals, as well as other concurrent statements.
- Processes, which may contain conditional and/or assignment statements to variables.

### 3.1 Concurrent statements

Concurrent statements are a kind of assignment statements to signals whose operation depends on a set of conditions. Two kinds of concurrent statements exist:

#### 3.1.1 WHEN-ELSE
This statement modifies the value of a given signal depending on a set of conditions, being the assigned values and the conditions independent among each other. The order in which the conditions are sorted determines their preference with respect to the others. In other words, in the previous definition, if condition_i is true, then value_i will be assigned to signal_to_modify, even if any other condition_j is also true (j>i).

### Example WHEN-ELSE

```vhdl
signal_to_modify <= value_1 when condition_1 else
value_2 when condition_2 else
... value_n when condition_n else
default_value;
```

### 3.1.2 WITH-SELECT-WHEN

This statement is less general than when-else one. It modifies the value of a signal, depending on the values that signal_condition may have.

```vhdl
with signal_condition select
signal_to_modify <= value_1 when value_1_signal_condition,
value_2 when value_2_signal_condition,
... value_n when value_n_signal_condition,
default_value when others;
```

### Example WITH-SELECT-WHEN

```vhdl
with input select
output <= "00" when "0001",
"01" when "0010",
"10" when "0100",
"11" when others;
```

From the point of view of the hardware, these two statements give as a result pure combinatorial hardware; in other words, logic gates, multiplexers, decoders...

**TIP:** A good VHDL programmer should be used to use these two kinds of sequential statements, since it will avoid having many problems associated to the if-then-else statements inside processes (explained in Section 3.3).
3.2 Conditional statements

These statements are assignment statements to variables that may or may not be based on a condition. As previously pointed out, they MUST be placed inside a process. The following conditional statements exist in VHDL:

3.2.1 IF-THEN-ELSE

```vhdl
process (sensitivity list)
begin
  if condition_1 then
    -- assignments
  elsif condition_2 then
    -- assignments
  else
    -- assignments
  end if;
end process;
```

Example IF-THEN-ELSE

```vhdl
process (control, A, B)
begin
  if control = "00" then
    output <= A + B;
  elsif control = "11" then
    output <= A - B;
  else
    output <= A;
  end if;
end process;
```

It is possible to chain as many if-then-else statements as desired, as in software description languages, such as Pascal, C, Java...

**TIP:** if-then-else statements should always have an else. In addition, as explained in Section 3.3, it is convenient to assign values to the same signals in each one of the branches of the statement, even if the value of some signals should be a don't care.

3.2.2 CASE-WHEN

```vhdl
process (sensitivity list)
begin
  case signal_condition is
    when value_condition_1 => -- assignments
    ..., when value_condition_n => -- assignments
    when others => -- assignments
  end case;
end process;
```
In this case, assignments may also be if-then-else statements. The when others clause must appear in the statement, but it is not necessary to write any assignment associated to it.

```
--- Example CASE-WHEN

process (control, A, B)
begin
  case control is
    when "00" => result <= A+B;
    when "11" => result <= A-B;
    when others => result <= A;
  end case;
end process;
```

As in other software programming languages, several types of loops are possible:

### 3.2.3 FOR-LOOP

```
--- Example FOR-LOOP

process (sensitivity list)
begin
  for var_loop in range loop
    --- assignments
  end loop;
end process;
```

The range can be defined as 0 to N or as N downto 0.

```
--- Example FOR-LOOP

process (A)
begin
  for i in 0 to 7 loop
    B(i+1) <= A(i);
  end loop;
end process;
```

### 3.2.4 WHILE-LOOP

```
--- Example WHILE-LOOP

process (sensitivity list)
begin
  while condition loop
    --- assignments
  end loop;
end process;
```

```
--- Example WHILE-LOOP

process (A)
```
3.3 Process statement

VHDL presents a particular structure named process that defines the limits of a code that will be simulated (or executed) if and only if any of the signals included in its sensitivity list has been modified in a previous simulation step.

A process features the following structure:

```
process (sensitivity_list)
  -- Assignments to variables
  -- This is optional and, in general, not recommended
begin
  -- Conditional statements
  -- Assignments to variables or to signals
end process;
```

Process statements are VERY used in VHDL programming, since, for software programmers, it is very easy to code the behavior of a hardware circuit as if it was a software program. However, this is an important drawback for beginners, since the software-like description of the behavior of the circuit may not actually synthesize into hardware. For this reason, a number of good coding practices exist, which are directly related with the properties of the process statement. One should be VERY aware of them in order to code a hardware circuit that simulates and synthesizes correctly.

Property I

Statements existing inside a process only run in the instant 0 of simulation OR if any of the signals of the sensitivity list changes.

Problem: The result of the simulation of the circuit may be unexpected due to the “malign effect” of the sensitivity list.

Solution: The sensitivity list MUST include all the signals that are read inside the process. (signal_written <= signal_read).

Let us explain this point by means of an example (Figures 3.1 and 3.2). In this example, no value is assigned to C until the instant 10 ns, although B changes at 5 ns. This happens because the code inside the process is not executed unless A changes (this happens at 10 ns). However, at the hardware level, one would expect C to take the value of A as soon as B changes to 1 (at 5 ns). Thus, following the solution proposed above, the correct code should be as follows:
Chapter 3. Basic Structure of a Source File in VHDL

--- Effect in the sensitivity list (1)

```vhdl
process (A)
begin
  if B = '1' then
    C <= A;
  end if;
end process;
```

(a) Example code

<table>
<thead>
<tr>
<th>t (ns)</th>
<th>0</th>
<th>5</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>U</td>
<td>U</td>
<td>1</td>
</tr>
</tbody>
</table>

(b) Table of transitions

Figure 3.1: Example for Property I of processes (1)

--- Effect in the sensitivity list (2)

```vhdl
process (A, B)
begin
  if B = '1' then
    C <= A;
  end if;
end process;
```

(a) Example code

<table>
<thead>
<tr>
<th>t (ns)</th>
<th>0</th>
<th>5</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>U</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

(b) Table of transitions

Figure 3.2: Example for Property I of processes (2)

Property II

Assignments to signals made inside a process have memory.

**Problem**: If, in a given simulation step, a process is executed and as a consequence, a signal S is modified; and if in a subsequent simulation step, the process is again executed but S is not modified inside the code of the process, then C will conserve the value assigned in the first process execution. This may lead to an expected behavior because of the “malign effect” of the process memory.

**Solution**: Conditional statements inside processes MUST assign a value to the same set of signals in any of the branches of the statements. In addition, unless it is strictly forbidden at the design level (see Chapter 5), all the conditions MUST have their corresponding else branch.

This is explained in the examples of Figures 3.3, 3.4 and 3.5. The first code includes an else branch, which means that there is a by-default value for C (in this example, at 5 ns). However, the second code does not include this else branch. Hence, at 5 ns, both for Case_1 and Case_2, the value of C differs in both codes ("00" vs. "10" and "00" vs. "11", respectively). The reason is that processes have memory and in this case, the input combination A = "11"; B = "10" does not match any branch in the second code. Hence, the value for C at 5 ns ("10" for Case_1 and "11" for Case_2) is the same one as in the previous simulation step (i.e., at 0 ns, see the table in Figure 3.4b). In addition, note that, for the code in Figure 3.3a, the output value for C is "00" in both Case_1 and Case_2 at 5 ns (in both cases, A = "11" and B = "10"). This is correct; however, this is not true for Case_2.

The code in Figure 3.5 is another example of an incomplete conditional statement. in this case, two different values at 5 ns are obtained for C and D in Case_3 and Case_4, even though the input values are
Process statement 3.3

--- Conditional complete

```plaintext
process (a, b)
begin
  if a = b then
    c := a or b;
  elsif a < b then
    c := b;
  else
    c := "00";
  end if;
end process;
```

(a) Example code

<table>
<thead>
<tr>
<th>Case_1 t (ns)</th>
<th>0</th>
<th>5</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>01</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>10</td>
<td>00</td>
<td>11</td>
</tr>
</tbody>
</table>

(b) Table of transitions

--- Conditional incomplete

```plaintext
process (a, b)
begin
  if a = b then
    c := a or b;
  elsif a < b then
    c := b;
  end if;
end process;
```

(a) Example code

<table>
<thead>
<tr>
<th>Case_3 t (ns)</th>
<th>0</th>
<th>5</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>01</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>10</td>
<td>00</td>
<td>11</td>
</tr>
</tbody>
</table>

(b) Table of transitions

---

Figure 3.3: Example for Property II of processes (1)

Figure 3.4: Example for Property II of processes (2)

exactly the same in both cases (A = "10"; B = "10"). This shows us that it is EXTREMELY important to make sure that not only the if-then-else statement has else branch, but also that the very same set of signals are assigned in all the branches. In this case, the problem arises because the if branch assigns a value to C, but not to D; whereas the elsif assigns a value to D, but not to C.

Property III

All the statements inside a process run in parallel, in a similar way as the statements outside a process do. However, if inside a process a signal is given a value at two different points, the final result will be the one of the last assignment, similarly as what happens in software programming languages. This may turn problematic and not synthesizable if not properly coded.

**Solution:** It is convenient to double-check that a signal is not assigned twice in the same process (this can be done in two different branches of an if-then-else statement).

In the following example, at 0 ns and at 10 ns, two values are assigned to C. When the process finishes, C takes the last assigned value, the one in the if and elsif branches, respectively. At 5 ns, C is assigned only one value "00", which is its final value.
Chapter 3. Basic Structure of a Source File in VHDL

---

**Example Code**

```
process (a, b)
begin
  if a = b then
    c := a or b;
  elsif a < b then
    d := b;
  else
    c := "00";
    d := "11";
  end if;
end process;
```

(a) Example code

---

**Table of Transitions**

**Case 3**

<table>
<thead>
<tr>
<th>t (ns)</th>
<th>0</th>
<th>5</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>UU</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>D</td>
<td>10</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

**Case 4**

<table>
<thead>
<tr>
<th>t (ns)</th>
<th>0</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>B</td>
<td>00</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>00</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>D</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

Figure 3.5: Example for Property II of processes (3)

---

**Example Code**

```
process (a, b)
begin
  c := "00";
  if a = b then
    c := a or b;
  elsif a < b then
    c := b;
  end if;
end process;
```

(a) Example code

---

**Table of Transitions**

**Case 3**

<table>
<thead>
<tr>
<th>t (ns)</th>
<th>0</th>
<th>5</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>01</td>
<td>11</td>
<td>01</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>10</td>
<td>00</td>
<td>11</td>
</tr>
</tbody>
</table>

Figure 3.6: Example for Property III of processes

---

**Property IV**

All process statements run in parallel.

**Problem:** In two processes P1 and P2 modify the same signal, then it is impossible to know its actual value. (The one assigned by P1 or P2?).

**Solution:** One should always double-check that a signal is not modified in two or more different processes. In that case, a possible solution is to merge the involved processes.

---

**Property V**

The values of all the signals that are modified inside a process are not updated until the whole process finishes.

**Problem:** If the sensitivity list is not correctly coded, the update of a signal may be postponed one
or several simulation events. This can be observed in the example of Figure 3.7, where only \( A \) is included in the sensitivity list. The example in Figure 3.8 does not present this problem any more.

**Solution:** As in Property IV, always double-check that a signal is not modified in two or more different processes.

![Wrong sensitivity list](image1)

\[
\begin{align*}
\text{process} & \quad (A) \\
\text{begin} & \\
B & \leftarrow A; \\
C & \leftarrow B; \\
\text{end process;}
\end{align*}
\]

(a) Example code

\[
\begin{array}{|c|c|c|c|}
\hline
t (ns) & 0 & 5 & 10 \\
\hline
A & 0 & 1 & 0 \\
B & 0 & 1 & 0 \\
C & U & 0 & 1 \\
\hline
\end{array}
\]

(b) Table of transitions

Figure 3.7: Example for Property V of processes (1)

![Sensitivity list OK](image2)

\[
\begin{align*}
\text{process} & \quad (A, C) \\
\text{begin} & \\
C & \leftarrow A; \\
B & \leftarrow C; \\
\text{end process;}
\end{align*}
\]

(a) Example code

\[
\begin{array}{|c|c|c|c|}
\hline
t (ns) & 0 & 5 & 10 \\
\hline
A & 0 & 1 & 0 \\
B & 0 & 1 & 0 \\
C & 0 & 1 & 0 \\
\hline
\end{array}
\]

(b) Table of transitions

Figure 3.8: Example for Property V of processes (2)

### 3.4 Structural description

This description is used to create an architecture that instantiates other entities that have already been defined elsewhere. This makes possible to build hierarchical descriptions of circuits, which improves their reusability and scalability.

In order to do this, such an architecture must declare the entities that will be instantiated as components, and add as many instances of these components as needed in the body of the architecture, as the following code illustrates. Structural descriptions are very useful in bottom-up hierarchical designs.

\[
\begin{align*}
\text{architecture} & \quad \text{circuit of name is} \\
\text{component} & \quad \text{subcircuit} \\
\text{port} & \quad (...) \\
\text{end component;}
\end{align*}
\]

---

...  

---  

"chip\_i" is the name of the instance declared in this code  

"subcircuit" is the name of the component that is used  

chip\_i: subcircuit port map (...) ;
The architecture may add as many instances of the same component as needed. The only restriction that VHDL imposes is that each one of the component instances must be given a different name in the body of the architecture.

**Figure 3.9: Example of a structural description of an entity**

The code below is an example of a structural description of the circuit depicted in Figure 3.9. Note that, in order to make the interconnections needed between the output of a component and the input of another one, intermediate signals are needed.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity F is
    port (A, B: in std_logic; Y: out std_logic);
end F;

architecture structural of F is
    component G
        port (Ag, Bg: in std_logic; Yg: out std_logic);
    end component;
    component H
        port (Ah, Bh: in std_logic; Yh: out std_logic);
    end component;
    component I
        port (Ag, Bg: in std_logic; Yg: out std_logic);
    end component;
    signal YA, YB, Yout: std_logic;
begin
    mod_G: G port map (A, B, YA);
    mod_H: H port map (A, B, YB);
    mod_I: I port map (YA, YB, Yi);
    Y <= Yout;
end structural;
```
**IMPORTANT:** In this example, note that the Y intermediate signals are needed, whereas no intermediate signal is needed in order to connect the inputs of the entity F (A and B) and the inputs of components mod_G and mod_H.

Structural descriptions of circuits can also be made by means of `generate` statements. These statements are used to automatically create an array of instances of the same component and/or other concurrent statements. The syntax of the `generate` statement is as follows:

```vhdl
for index in range generate
  -- range can be 0 to N or N downto 0; N being a constant
  -- concurrent statements
  -- component instances
end generate;
```

The two following examples show how `generate` instances are instantiated and used in a VHDL code:

---

```vhdl
-- Example GENERATE 1

signal a, b: std_logic_vector(0 to 7)
...
gen1: for i in 0 to 7 generate
  a(i) <= not b(i);
end generate gen1;
```

---

```vhdl
-- Example GENERATE 2

component subcircuit
  port( x: in std_logic; y: out std_logic);
end component comp
...
signal a, b: std_logic_vector(0 to 7);
...
gen2: for i in 0 to 7 generate
  u: subcircuit port map(a(i), b(i));
end generate gen2;
```

---

Component instantiations in `generate` statements can also include conditions, as long as they are referred to the index of the `for` in the `generate` statement. The following code shows an example of this:

---

```vhdl
-- Example GENERATE 3

signal a, b: std_logic_vector(0 to 7)
...
loop_1: for i in 0 to 7 generate
  condition: if i > 0 generate
    a(i) <= b(i-1);
  end generate condition;
end generate loop_1;
```
This example assigns the values of the vector \( b \) to the vector \( a \), by left-shifting them 1 position. However, it does not assign any value to \( a(0) \), since in that case, the condition in the \texttt{generate} statement is not met.

However, the following code (which is NOT correct), the generated hardware depends on the value of \( b(i) \), which is not known at design time. Since the actual value of the \( b \) vector depends on the execution of the circuit at any point of time, it is not possible to generate any hardware with this \texttt{generate} statement.

```
--- Example GENERATE 4

signal a, b: std_logic_vector(0 to 7)
loop_1: for i in 0 to 7 generate
  condition: if b(i) = '0' generate
    a(i) <= b(i-1);
  end generate condition;
end generate bucle;
```

### 3.5 Examples

#### 2:1 multiplexer

A possible \texttt{entity} description for this module would be as follows:

```
--- Entity declaration for a 2:1 MUX

entity mux2 is
  port (D0, D1, S0: in std_logic; O out std_logic);
end mux2;
```

**Table 3.1: Truth table of a 2:1 multiplexer**

<table>
<thead>
<tr>
<th>S0</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>D0</td>
</tr>
<tr>
<td>1</td>
<td>D1</td>
</tr>
</tbody>
</table>

Table 3.1 summarizes the operation of a 2:1 multiplexer, which can be coded in VHDL as follows:

```
--- Behavioral VHDL code for a 2:1 MUX (1)

architecture behavioral_1 of mux2 is begin
  O <= D1 when (S0 = '1') else D0;
end behavioral_1;
```

or as follows:
However, we also know that the operation of this circuit is equivalent to the truth table of the circuit depicted in Figure 3.10.

![Figure 3.10: Structural description of a multiplexer](image)

Which is equivalent to the following structural code:

```vhdl
-- Structural VHDL code for a 2:1 MUX
architecture structural of mux2 is

-- component declarations
component AND2
  port (I0, I1: in std_logic; O: out std_logic);
end component;

component OR2
  port (I0, I1: in std_logic; O: out std_logic);
end component;

component INV
  port (I0, I1: in std_logic; O: out std_logic);
end component;

-- signal declarations
signal S1, S2, S3: std_logic;

begin
  U1: INV port map (S0, S1);
  U2: AND2 port map (D0, S1, S2);
  U3: AND2 port map (S0, D1, S3);
  U4: OR2 port map (S2, S3, O);
end structural;
```
Or the following behavioral code:

```vhdl
-- Hybrid structural/behavioral VHDL code for a 2:1 MUX

architecture mixed of mux2 is
  signal S1, S2: std_logic;
begnin
  S1 <= D0 and not S0;
  S2 <= D1 and S0;
  O <= S1 or S2;
end mixed;
```
Chapter 4

Simulation of a VHDL Code

Typically, simulation tools used for VHDL programming follow a discrete event time model for simulating circuits described in this language. This means that these simulators model the operation of a system as a discrete sequence of events in time. Events occur each time any signal changes its value. This marks a potential change of state in the circuit. Between two consecutive events, no change in the system is assumed to occur. Thus, the simulation can directly jump in time from one event to the next one, independently of the time elapsed between them (i.e., just a few picoseconds or several seconds).

4.1 Steps of simulation

VHDL simulations comprise three steps:

- **Step 0**: All the signals are initialized and the time count is set to 0.
- **Step 1**: All the transitions scheduled for that time are carried out.
- **Step 2**: All the signals that are modified as a consequence of transitions occurring at instant $= t$ are written down in the list of events and scheduled for instant $= t + \delta$, where $\delta$ is infinitesimal.

Steps 1 and 2 are repeated as many times as necessary until no more transitions exist. As previously stated, the values assigned to signals remain constant from one event to the following one.

The examples in Figures 4.1, 4.2 and 4.3 illustrate these three simulation steps. On the one hand, Examples 1 and 2 simulate two concurrent assignments that are placed outside a process, where $A$ takes value ‘0’ at 0 ns, and ‘1’ at 5 ns.

![Example code](image1)

<table>
<thead>
<tr>
<th>$t$ (ns)</th>
<th>0</th>
<th>0+$\delta$</th>
<th>No more changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>U</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>U</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

![Table of transitions](image2)

Figure 4.1: Example 1: Simulation steps in VHDL.
Chapter 4. Simulation of a VHDL Code

\[
\begin{array}{c|cc|cc|cc|c}
 t (\text{ns}) & 0 & 0+\delta & 0+2\delta & 5 & 5+\delta & 5+2\delta \\
\hline
 A & 0 & 0 & 0 & 1 & 1 & 1 \\
 B & U & 0 & 0 & 0 & 1 & 1 \\
 C & U & U & 0 & 0 & 0 & 1 \\
\end{array}
\]

(a) Example code

(b) Table of transitions

Figure 4.2: Example 2: Simulation steps in VHDL.

On the other hand, in the following example, the two assignments are made inside a \texttt{process} with a sensitivity list:

\begin{verbatim}
process (A)
begin
B <= A;
C <= B;
end process;
\end{verbatim}

(a) Example code

(b) Table of transitions

Figure 4.3: Example 3: Simulation steps in VHDL.

In this case, we can observe that the output value at the end of the simulation in Figure 4.3b differs from what was obtained in Figures 4.1b and 4.2b (B='1' and C='0'). Let us analyze in detail what is happening in this example: At 0 ns, the \texttt{process} is executed, B is assigned the value of A, and C is assigned the value that B had before the \texttt{process} execution (which is 'U' or "undefined"). At 0 + \delta, A does not change, hence the \texttt{process} is not executed again and all the signals keep their values. The simulation is resumed at 5 ns, when A changes (from '0' to '1'). Hence, the \texttt{process} is executed again and as a consequence, B is assigned the value of A, and C is assigned the value that B had before this new \texttt{process} execution (which is '0'). In this new simulation step (5 + \delta), A does not change, hence the \texttt{process} is not executed again and the signal values do not change. Figures 4.4 and 4.5 show what could be seen in any VHDL simulator. Note that \delta is infinitesimal and therefore, not visible in the simulation.

\begin{verbatim}
A B C
\end{verbatim}

\begin{verbatim}
0 1 2 3 4 5 6
\end{verbatim}

Figure 4.4: Simulation results for Examples 1 and 2

Obviously, the simulation result obtained in the table of Figure 4.5 is incorrect. This can be easily solved by adding B to the sensitivity list of the \texttt{process}.

4.2 Simulation statements

VHDL features the \texttt{wait} statement, which stops the simulation of the code until a condition is met. A \texttt{process} must include a \texttt{wait} statement if it does not have any sensitivity list. In addition, it is also
possible to generate sequential hardware by using wait statements. Chapter 5 makes a deeper discussion about this. There are three types of wait statements:

- **wait on list_of_signals;** The simulation stops until any signal in the list_of_signals is modified.
- **wait for time;** The simulation stops for the time specified in the time variable.
- **wait until condition;** The simulation stops until the condition is met.

The following code illustrates the operation of the wait statement. In this example, C cannot be updated unless A is ‘1’, which occurs at 5 ns.

```
process
begin
  B <= A;
  wait until A = '1';
  C <= B;
end process
```

<table>
<thead>
<tr>
<th>t (ns)</th>
<th>0</th>
<th>0+δ</th>
<th>5</th>
<th>5+δ</th>
<th>5+2δ</th>
<th>No more change</th>
<th>No more change</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>No more change</td>
<td>No more change</td>
</tr>
<tr>
<td>B</td>
<td>U</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>No more change</td>
<td>No more change</td>
</tr>
<tr>
<td>C</td>
<td>U</td>
<td>U</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>No more change</td>
<td>No more change</td>
</tr>
</tbody>
</table>

(a) Example code  (b) Table of transitions

4.3 Simulation templates in VHDL

Many simulation and synthesis tools include a graphical user interface (GUI) to help set the stimuli to the circuit inputs in order to check if the design works correctly. However, for large circuits and/or large test benches, it is much more practical to create a testbench directly using VHDL.

![Simulation Entity](image)

Figure 4.7: RTL description of a simulation template for testbenches in VHDL

Either if the testbench has been created by using the GUI or it has directly typed, the final result will be a VHDL file containing an entity without any inputs or outputs, and that instantiates two process and a component, as indicated in Figure 4.7. The latter actually instantiates the circuit under test.
It is important to know what a VHDL testbench file looks like. First of all, it must include the following libraries:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_TEXTIO.ALL;
use STD.TEXTIO.ALL;
```

Next, an `entity` without any inputs or outputs must be created:

```vhdl
entity simulation is
end simulation;
```

Next, the `architecture` is described, which includes the processes and the components of the circuit under test (if any), as described above. A possible template to instantiate the circuit under test and the process to set stimuli to its input signals could be as follows:

```vhdl
architecture testbench_arch of simulation is
begin
UUT : circuit port map (input, ..., output);
process
begin
wait for 200 ns;
input <= '1';
...
wait for 100 ns; -- Total: 300 ns
input <= '0';
...
wait for T ns; -- Total: 300 + T ns
input <= '1';
...
wait for ...
...
```
The first `wait` of the code (`wait for 100 ns;`) keeps the input signal to its initial value (`'0'`) for the first 100 ns. Then, the following statements are executed, among which the assignment `input <= '1'`. The second `wait` (`wait for 100 ns;`) keeps this new value for another 100 ns.

The following code is another example, which simulation result is depicted in Figure 4.8.

```
--- Example for the input stimuli process in a VHDL testbench
process
begin
    wait for 5 ns;
    A <= '1';

    wait for 5 ns;  -- Total: 10 ns
    A <= '0';

    wait for 10 ns;  -- Total: 20 ns
    A <= '1';

    wait for 5 ns;  -- Total: 25 ns
    A <= '0';

    wait for 10 ns;
end process;
```

![Figure 4.8: Example of a signal simulation, whose values are set manually](image)

The second `process` included in the VHDL template defines the clock signal very easily:

```
--- Template for the process that defines the clock
process
begin
    wait for 10 ns;
    CLOCK_LOOP : loop
        clk <= '0';
        wait for time_low ns;
        clk <= '1';
        wait for time_high ns;
    end loop CLOCK_LOOP;
end process;
```

Through the `loop` statement, this process generates a signal (`clk`) whose value is set to `'0'` and to `'1'` alternatively, according to the times specified in the `time_low` and `time_high` constants. The first
wait statement keeps the initial value of clk for the first 10 ns. Then, the second wait indicates the time that clk is set to its low value ('0'). Finally, the last wait indicates the time that clk is set to its high value ('1'). This sequence is repeated forever. Note that this is possible thanks to the loop statement. Actually, this is a slight variation with respect to the for-loop and while-loop ones statements, already described in Chapter 3, Section 3.2. In this case, since the loop statement does not have any condition, it never stops iterating. In other words, it will iterate until the final execution of the testbench.
Chapter 5

Description of Sequential Logic

As previously explained in Chapter 3, Section 3.3, one of the most important properties of processes is their ability to keep the values assigned to signals inside them, as long as the process is not executed again or a subsequent execution of that process does not assign any other value to that signal (see Property II in Chapter 3, Section 3.3). For this reason, processes can be used to describe sequential logic.

**IMPORTANT:** Processes being used to described sequential logic DOES NOT mean that the statements comprised in that process run sequentially.

5.1 Sequential hardware

Processes can be used in order to describe flip-flops and registers. To this end, the 'event attribute can be used on the clock signal as follows:

```
if (clk 'event and clk = '1') then ...
```

The 'event attribute on a signal returns true if that signal has just been modified, and it returns false otherwise. The previous if-then statement checks if there has been any modification on the clk signal, and if its new value is '1'. Thus, it recognizes a rising edge in the clock signal. Using this concept, a D flip-flop can be described as follows:

```
--- D Flip-Flop

entity D_FF is
  port (d, clk: in bit; q: out bit);
end D_FF;
architecture ARCH of D_FF is
begin
  process (clk, d)
  begin
    if (clk 'event and clk = '1') then q <= d; end if;
  end process;
end ARCH;
```
However, typically a D flip-flop has also a reset signal. In case this signal was asynchronous, it could be implemented as follows:

```vhdl
-- D Flip-Flop with asynchronous reset
entity D_FF_SReset is
  port(d, clk, reset: in bit; q: out bit);
end D_FF_SReset;
architecture ARCHASYN of D_FF_SReset is
begin
  process (clk, reset, d)
  begin
    if (reset = '1') then q <= '0';
    elsif clk = '1' and clk'event then q <= d;
  end if;
end process;
end ARCHASYN;
```

As this code shows, the process runs if there is any change in the clk, reset or d signals, as indicated in its sensitivity list. Then, the if-then statement checks if the reset signal has been set to '1'. In that case, the output value of the FF is set to '0'. Otherwise, the clk signal is checked in a similar way as in the D flip-flop example.

Finally, a possible code for a D flip-flop with synchronous reset could be as follows:

```vhdl
-- D Flip-Flop with synchronous reset
architecture ARCH_SYN of D_FF_AReset is
begin
  process (clk, reset, d)
  begin
    if clk = '1' and clk'event then
      q <= d;
    elsif (reset = '1') then q <= '0';
    end if;
  end process;
end ARCH_SYN;
```
For Xilinx\textsuperscript{TM} users: In general, in order to create sequential hardware with the expected behavior, the following rules must be fulfilled:

- An if-then statement used to detect a clock edge cannot have an else branch. Otherwise, the else branch would always run, except the precise moments when the clock signal changes.
- In if-then-elsif statements, the edge in a clock signal can only be detected in the last branch of the if-then-elsif sentence (which MUST NOT have an else branch).
- An if-then statement used to detect a clock edge can have as many chained if-else statements as necessary.
- A process can only have one edge detection. Otherwise, it would mean that the specified hardware would be sensible to several clock signals. This is far beyond the objectives of this course.

These ideas are illustrated by means of the examples depicted in Figures 5.1, 5.2 and 5.3. The difference between the codes in Figures 5.1a and 5.2a is the order in the assignment of values to a, b and c. In Figure 5.1a, this assignment would be done as in any other software description language. However, in Figure 5.2a, one could think that the assignments are incorrect. In order to understand the results, one should remember Properties IV and V, already described in Chapter 3, Section 3.3:

Property IV: All process statements run in parallel. Hence, the order in which the assignments appear in the code is not relevant to the final result. This explains why, for Figures 5.1b and 5.2b, the values of the outputs b and c are the same (‘1’ and ‘0’ at 5 ns; ‘1’ and ‘1’ at 10 ns) in both cases.

Property V: The values of all the signals that are modified inside a process are not updated until the whole process finishes. This explains why the value of c is not updated at 5 ns, but at 10 ns.

Finally, in the code of Figure 5.3a, b and c are directly assigned the value of a. As a consequence, both signals are updated simultaneously.
Chapter 5. Description of Sequential Logic

5.2 Counters

One of the most common components of digital circuits are counters. A “possible” way of describing a counter would be by means of the following code. However, it does not work as a counter. WHY?
The reason is simple: The code above does not manage the update of the output signal in a controlled way (i.e., following a clock signal). This means that the inferred circuit from that code would look like as in Figure 5.4a. This is not a sequential circuit, but a combinatorial one, which implementation makes no sense whatsoever. A behavioral simulation of this circuit would not make any sense either. This is illustrated in Figure 5.4b, where the counter never stops iterating after its input has been set to '1' (in this example, this is assumed to happen at 5 ns).

As we already know, a counter is a sequential circuit that is controlled by means of a clock signal. This makes possible to generate an ascending or descending sequence whose values are generated each new clock cycle. Therefore, a VHDL code for a counter must include clock edge recognition (for instance, with the 'event attribute).

```vhdl
-- Example of a mod-8 counter

entity counter is
  port (reset, clk: in std_logic; n: out std_logic_vector(3 downto 0));
end counter;

architecture arch of counter is
  signal intermediate: std_logic_vector(3 downto 0);
begin
  process (reset, clk, intermediate)
  begin
    if (reset = '1') then
      intermediate <= "000";
    else
      intermediate <= intermediate + 1;
    end if;
  end process;
  n <= intermediate;
end arch;
```

Figure 5.4: Example of a badly coded counter
Chapter 5. Description of Sequential Logic

The signal \texttt{intermediate} is defined as a \texttt{std_logic_vector(3 downto 0)}. By applying the rules of the IEEE standard, we have "1111' + '1' = "0000". Thus, the code above implements an ascending count that is follows the sequence [0...15] and then, it starts over again.

\textbf{For Xilinx\textsuperscript{TM} users:} Xilinx\textsuperscript{TM} tools need VHDL descriptions of circuits to include a reset signal. It can be either asynchronous (as in the previous example) or synchronous. This makes possible to design counters that are updated until they reach a given maximum value and then they are initialized to 0; or counters that stay in that maximum value; or counters that start from a given value that can be previously loaded using a load signal. All these functionalities can be described by using \texttt{if-then-else} statements inside the \texttt{elsif clk'event and clk = '1' then}... branch.

Thus, the following example describes a generic counter that, once it reaches a maximum value, is re-initialized and starts over again.

\begin{verbatim}
  entity counter is
    generic (maximum : natural := max; N : natural := 8);
    port (reset, clk : in std_logic; n : out std_logic_vector(N-1 downto 0));
  end counter;

  architecture arch of counter is
    signal intermediate : std_logic_vector(N-1 downto 0);
  begin

    process (reset, clk, intermediate)
    begin
      if (reset = '1')
        intermediate <= "000";
      elsif clk'event and clk = '1' then
        if intermediate < max
          intermediate <= intermediate + 1;
        else
          -- This statement sets all the bits in "intermediate" to 0
          intermediate <= (others=>'0');
        end if;
      end if;
    end process;

    n <= intermediate;
  end arch;
\end{verbatim}
5.3 Examples

5.3.1 8-bit register

```vhdl
-- 8-bit register: behavioral description

entity register_8 is
  port (clk, reset: in bit;
        A: in bit_vector(7 downto 0);
        B: out bit_vector(7 downto 0));
end register_8;

architecture behavioral of register_8 is
begin
  process (clk, reset)
  begin
    if reset = '1' then B <= "00000000";
    elsif (clk'event and clk='1') then B <= A;
    end if;
  end process;
end behavioral;
```

5.3.2 8-bit register built using 1-bit flip-flops

```vhdl
-- 8-bit register: structural description

entity FF is
  port (clk, reset, C: in bit; D: out bit);
end FF;

architecture arch of FF is
begin
  process (clk, reset)
  begin
    if reset = '1' then D <= '0';
    elsif (clk'event and clk='1') then D <= C;
    end if;
  end process;
end arch;

entity register_8 is
  port (clk, reset: in bit;
        A: in bit_vector(7 downto 0);
        B: out bit_vector(7 downto 0));
end register_8;

architecture structural of register_8 is
component FF
  port(clk, reset, c: in bit; d: out bit);
end component FF;
signal F: bit_vector(7 downto 0);
```
begin
  gen: for i in 0 to 7 generate
  u: FF port map(clk, reset, A(i), F(i));
  end generate gen;
  B <= F;
end structural;
Chapter 6

Design of a Finite State Machine (FSM)

VHDL allows to describe finite state machines (FSMs) at the algorithmic level. This makes possible to easily coding the operation of any FSM without having to actually write the state transition and the output functions.

The register transfer level description of a FSM looks like as indicated in Figure 6.1. There are many ways of describing FSMs in VHDL. The one proposed in this chapter is valid to any synthesis tool that works with this programming language.

First of all, one must define an enumerated type including all the identifiers of the states. It is practical to select representative names for the states. The synthesis tool will be able to assign a binary code to each one of them.

```vhdl
type STATES is (up, down, stop, ...);
```

Next, the body of the architecture must define the state transition function (F) and the output function (G); as well as the ability to change from one state to the following one. For this purpose, two processes are defined:

- The first one codes F and G functions. In other words, depending on the current state, it specifies the new values of the state and the output(s).
- The second one is a sequential process that models the flip-flops for the state. Hence, its only objective is to update the current state of the FSM.
Let us illustrate this in greater detail by means of an example. Let a FSM be a sequence recognizer that detects the sequence "001" that comes through a serial input E. It is assumed that E is synchronized with a clock signal. This sequence recognizer can be implemented as the Moore machine depicted in Figure 6.2 with the following four states:

- S1: Wait for the first '0' in the sequence.
- S2: Wait for the second '0' in the sequence.
- S3: Wait for the '1' in the sequence.
- S4: Activate the output signal.

Thus, for the VHDL implementation, the first step is to define an enumerated type that comprises the 4 states involved:

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity FSM is
port(reset, E, clk: in bit; O: out bit);
end FSM;

architecture ARCH of FSM is
type STATES is (S1, S2, S3, S4);
signal STATE, NEXT_STATE: STATES;
begin

SYNCHRONOUS: process(clk, reset)
begin
if reset = '1' then
STATE <= S1;
end if;
end process;
```

This completes the VHDL implementation of the FSM sequence recognizer.
elsif clk 'event and clk = '1'
then
  STATE <= NEXT_STATE;
end if;
end process SYNCHRONOUS;

COMBINATORIAL: process (STATE, E)
begin
  case STATE is
  when S1 =>
    O <= '0';
    if (E = '0') then
      NEXT_STATE <= S2;
    else
      NEXT_STATE <= S1;
    end if;
  when S2 =>
    O <= '0';
    if (E = '0') then
      NEXT_STATE <= S3;
    else
      NEXT_STATE <= S1;
    end if;
  when S3 =>
    O <= '0';
    if (E = '0') then
      NEXT_STATE <= S3;
    else
      NEXT_STATE <= S4;
    end if;
  when S4 =>
    O <= '1';
    if (E = '0') then
      NEXT_STATE <= S2;
    else
      NEXT_STATE <= S1;
    end if;
end case;
end process COMBINATORIAL;
end ARCH;

For Xilinx™ users: Xilinx™ might not be able to recognize a FSM. In this case, it may delete many intermediate signals and group conditions. If we want Xilinx™ to recognize the FSM, the following two rules must be fulfilled:

- The state machine must include a reset so it can be initialized.
- The combinatorial process must ALWAYS assign a value to NEXT_STATE (even if this may seem redundant).
Chapter 7

Functions, Procedures and Packages

VHDL supports two kinds of subprograms (functions and procedures) that greatly help to improve the description, scalability and reusability of the code. A number of these subprograms can be gathered under a common structure named package, as shown in the code below:

```vhdl
package p is
  function fname (input_signals) return type;
  procedure pname (input_signals; output_signals);
end p;

package body p is
  ...
end p;
```

### 7.1 Functions

They are used to carry out punctual calculations and they return a value instantly.

- They cannot modify their input parameters.
- They cannot modify signals or variables externally declared to the function.
- They always return a value whose type has been specified in the function declaration.
- Their execution time is 0. Hence, they cannot contain any `wait` statement.

The syntax of functions is as follows:

```vhdl
function identifier(...) return type
  begin
    -- Signals, variables
    begin
    -- Function body
    -- It can use any VHDL statement
    return value;
  end function identifier;
```

43
7.2 Procedures

Procedures constitute another way to describe small circuits.

- They can exchange data bidirectionally with the outside world.
- They can contain wait statements.
- They can assign values to signals.
- They are defined in the declarations zone of the architecture.

```
procedure name(parameters)
  -- signals, variables
begin
  -- body of the procedure
end procedure name;
```

7.3 Examples

The package pf the code below includes various functions for converting data from vector to natural and vice versa, as well as a procedure to add vectors.

```
library IEEE;
use IEEE.std_logic_1164.all;

package arith_operations is
  function vector_to_natural (v: in std_logic_vector) return natural;
  function natural_to_vector (nat: in natural; length: in natural) return std_logic_vector;
  procedure vector_add (v1, v2: in std_logic_vector; vo: out std_logic_vector);
end arith_operations;

package body arith_operations is
  function vector_to_natural (v: in std_logic_vector) return natural is
    variable aux : natural := 0;
  begin
    for i in v'range loop
      if v(i)='1' then
        aux := aux + (2**i);
      end if;
    end loop;
    return aux;
  end vector_to_natural;

  function natural_to_vector (nat: in natural; length: in natural) return std_logic_vector is
    variable v: std_logic_vector(length-1 downto 0);
  variable quotient, aux, i, remainder: natural;
  begin
    aux := nat;
    i := 0;
    while (aux/=0) and (i<length) loop
      quotient := aux/2;
      aux := aux mod 2**i;
      i := i + 1;
    end loop;
```

Examples 7.3

remainder := aux mod 2;
if remainder=0 then v(i):='0'; else v(i):='1';
end if;
i := i+1;
aux := quotient;
end loop;
for j in i to length-1 loop
v(j):='0';
end loop;
return v;
end natural_to_vector;

procedure vector_add (v1, v2 : in std_logic_vector; vo : out std_logic_vector) is
variable sum, long: natural;
begin
long := v1'length;
sum := vector_to_natural(v1) + vector_to_natural(v2);
result := natural_to_vector(sum, long);
end vector_add;
end arith_operations;

library IEEE;
use IEEE.std_logic_1164.all;
use work.arith_operations_all; -- The packet must be included in order to use it

entity sum is
port (v1, v2: in std_logic_vector; v_result : out std_logic_vector);
end sum;

architecture beh of sum is begin
pl: process(v1, v2)
variable sum_var: natural;
begin
vector_add(v1, v2, sum);
result <= sum_var;
end process pl;
end beh;
Chapter 8

Design of a RAM Memory

By reusing all the concepts that have been described throughout this document, we can now design a kind of memory that is very common in digital circuits design and computer architecture: a RAM memory with synchronous write and asynchronous read:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-- RAM memory with 32 8-bit words
entity ram is
port (addr: in std_logic_vector (4 downto 0);
we, clk: in std_logic;
data_i: in std_logic_vector(7 downto 0);
data_o: out std_logic_vector(7 downto 0));
end ram;

architecture archxi of ram is
type ram_table is array (0 to 31) of std_logic_vector(7 downto 0);
signal rammemory: ram_table;
begin
process (we, clk, addr)
begin
if clk'event and clk='1' then
if we = '1' then
rammemory(conv_integer(addr)) <= data_i;
end if;
end if;
data_o <= rammemory(conv_integer(addr));
end process;
end archxi;
```

For Xilinx™ users: Xilinx™ recognizes the previous code as a memory, but it does not synthesize that code using the memory blocks that typically exist in FPGAs for that purpose (Block RAMs or BRAMs). In order to achieve this, one can either directly instantiate BRAMs by means of specific primitives provided by Xilinx™, or to code a generic synchronous memory with read and write ports, as well as enable, write and read signals, as in the code below.
The `conv_integer` function is defined in the package `ieee.std_logic_unsigned.all`. It converts a binary vector to an integer value. Note that, in VHDL, the access index to the vectors are integer values.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity SRAM is
genéric (w: integer := 4; — word width
d: integer := 4; — number of words
a: integer := 2); — address width
port (Clock: in std_logic;
  Enable: in std_logic;
  Read: in std_logic;
  Write: in std_logic;
  Read_Addr: in std_logic_vector(a-1 downto 0);
  Write_Addr: in std_logic_vector(a-1 downto 0);
  Data_in: in std_logic_vector(w-1 downto 0);
  Data_out: out std_logic_vector(w-1 downto 0)
); end SRAM;

architecture behavioral of SRAM is
  — We use an array to store the memory values
  type ram_type is array (0 to d-1) of std_logic_vector(w-1 downto 0);
  signal tmp_ram: ram_type;
begin
  — Read
  process (Clock, Read)
  begin
    if (Clock’event and Clock = '1') then
      if Enable = '1' then
        if Read = '1' then
          Data_out <= tmp_ram(conv_integer(Read_Addr));
        else
          Data_out <= (Data_out'range => 'Z'); — All bits of Data_out are set to 'Z'
        end if;
      end if;
    end if;
  end process;

  — Write
  process (Clock, Write)
  begin
    if (Clock’event and Clock = '1') then
      if Enable = '1' then
        if Write = '1' then
          tmp_ram(conv_integer(Write_Addr)) <= Data_in;
        end if;
      end if;
    end if;
  end process;
end behavioral;
```
Chapter 9

Appendixes

9.1 Discussion about using signals vs. variables

Signals are used to connect different components in a circuit, whereas variables are used inside process to compute certain values. The following example illustrates this point:

```vhdl
entity sig_var is
  port (d1, d2, d3 : in std_logic; res1, res2 : out std_logic);
end sig_var;

architecture behv of sig_var is
  signal sig_s1 : std_logic;
begin
  proc1 : process(d1, d2, d3)
  variable var_s1 : std_logic;
  begin
    var_s1 := d1 and d2;
    res1 <= var_s1 xor d3;
  end process;

  proc2 : process(d1, d2, d3)
  begin
    sig_s1 <= d1 and d2;
    res2 <= sig_s1 xor d3;
  end process;
end behv;
```

One could think that both process should return exactly the same result, since the operations that are carried out are exactly the same.

However, this is not true. Let us take a look at the simulation in Figure 9.1.

Why is res2 set to '1' later than res1? The reason is that the variable var_s1 and the signal res1 are updated in the same simulation step in proc1, whereas the signals sig_s1 and res2 need several simulation steps in proc2 for the update of these two signals, respectively.

When d1=1, d2=1 and d3=0, the process proc1 updates var_s1 to its new value (which is '0'), and this new value is taken in the same simulation step to update res1. Therefore, res1 is automatically set to 1. However, in proc2, at this same instant of time, sig_s1 is set to 1, but nothing
Chapter 9. Appendixes

Figure 9.1: Simulation example showing the difference between the update of signals and variables

happens to res2 at that simulation step. Since a change in $\text{sig}_s1$ does not trigger the execution of the process proc2 (note that $\text{sig}_s1$ is not in the sensitivity list of this process), res2 remains unchanged until a new modification of $d_1$, $d_2$ or $d_3$. The next simulation step (occurring when $d_1=0$, $d_2=0$ and $d_3=1$) does not trigger the change of res2 either. The following one (when $d_1=0$, $d_2=0$ and $d_3=1$) does trigger the modification of res2 from 0 to 1. The problem in this case is that this modification comes too late.

As previously hinted, looking at the code of both proc1 and proc2, one would expect the behavior obtained for proc1 in both cases. Hence, proc2 returns a wrong simulation result. Does this mean that we should better using variables inside processes instead of signals, in order to guarantee a proper and immediate update? NOT AT ALL. What we should do is to double-check the sensitivity list of proc2 and to realize that the signal $\text{sig}_s1$, which is updated in that process, is missing in that sensitivity list. If added, the simulation result of proc2 will be exactly the same as that of proc1. This is closely related with what is explained in Section 9.2.

9.2 Discussion about the effect of incorrectly coding the sensitivity list in a process

For Xilinx\textsuperscript{TM} users: When implementing a circuit, Xilinx\textsuperscript{TM} does not take into account the sensitivity list of a process whatsoever. This means that, if the VHDL code is not properly written, the simulation and the final implementation results will differ. Thus, it is very important to remember that the sensitivity list of a process MUST include all the signals that are read inside it.

The code below and the simulation in Figure 9.2 illustrate this point:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity das is
port(din, sel, clk : in std_logic; dout : out std_logic);
end das;

architecture Behavioral of das is
signal A, B, C : std_logic;
begin
Type_C : process(clk)
```
Discussion about the effect of incorrectly coding the sensitivity list in a process.

Figure 9.2: Simulation example that illustrates the effect of an incomplete sensitivity list.