**Programming elements in VHDL**

1. **Introduction**

VHDL (VHSIC HDL) is a **H**ardware **D**escription **L**anguage used in electronic design automation to describe digital and mixed-signal systems such as **F**ield-**P**rogrammable **G**ate **A**rrays (FPGAs) and **I**ntegrated **C**ircuits. VHSIC stands for **V**ery **H**igh **S**peed **I**ntegrated **C**ircuit. It describes the behavior and structure of electronic systems, but is particularly suited as a language to describe the structure and behavior of digital electronic hardware designs (ASICs, FPGAs, conventional digital circuits, etc.). VHDL is an international standard, regulated by the IEEE, but the definition of the language is non-proprietary.

Basically, VHDL was developed as an alternative to huge, complex manuals which were subject to implementation-specific details. Also, the idea of being capable to simulate this documentation was very attractive and obvious. Thus, logic simulators were developed that could read the VHDL files. Next was the development of logic synthesis tools that read the VHDL, and output a definition of the physical implementation of the circuit.

The initial version of VHDL was designed to IEEE standard 1076-1987, in order to document the behavior of the ASICs that supplier companies were including in equipment. This first version included a wide range of data types: numerical (integer and real), logical (bit and boolean), character, time, and arrays of bits (bit\_vector and strings). Future version updates included multi-valued logic (std\_logic and std\_logic\_vector), syntax became more consistent and allowed more flexibility in naming.

In normal utilization, VHDL is used to write text models that describe a logic circuit, which is afterwards processed by a synthesis program. A simulation program is used to test the logic design using simulation models to represent the logic circuits that interface to the design, called *testbench*.

VHDL has constructs to handle the parallelism inherent in hardware design, but these constructs (called *processes*) differ in syntax from the normal code (e.g. tasks in ADA). Note that each VHDL model is translated into gates and wires that are mapped onto programmable logic devices (CPLDs, FPGAs, etc.). Thus, the actual hardware is configured, rather than VHDL code being executed as if on some form of a processor chip.

Advantages of VHDL:

* Allows the behavior of the required system to be described/modeled and verified/simulated before synthesis tools translate the designs into real hardware
* Allows the description of concurrent systems; VHDL is a dataflow language, unlike procedural computing languages such as C, assembly, etc., which all run sequentially, one instruction at a time
* Any VHDL project is portable and it can be ported to another element base (e.g. VLSI)

This laboratory work contains how to write code in VHDL (concurrent, sequential, etc.) and how to simulate any design. There is also an annex with the basic language elements, data types, operators, etc. (called L3a - Annex).

1. **Programming in VHDL**
   1. **Behavioral model - Sequential statements**

A VHDL code can be written in a *sequential* way or a *concurrent* way. A *sequential* code is represented by a process or subprogram that contains sequential statements and the statements are executed in the order they appear within the process or subprogram (as in normal programming languages). A *concurrent* code is represented by an architecture that contains processes, concurrent procedure calls, concurrent signal assignments and component instantiations.

In the following paragraphs, the format and use of sequential statements are described.

**Processes**

A process is a sequence of statements that are executed in the specific order. The declaration of a process may appear anywhere in the architecture body (after the keyword *begin*). The syntax of a process declaration is the following:

[name:] process [(sensitivity\_list)]

[type\_declarations]

[constant\_declarations]

[variable\_declarations]

[subprogram\_declarations]

begin

sequential\_statements

end process [name];

The declaration of a process is contained between the keywords process and end process. An optional name may be assigned for simpler identification. Note that any event on any of the signals from the sensitivity list causes the sequential instructions in the process to be executed. Also, the process will be executed in an infinite loop.

A declaration of a simple process is visible in Example 1 below.

**Example 1**

proc1: process (a, b, c)

begin

x <= a and b and c;

end process proc1;

**Wait statement**

Instead of a sensitivity list, a process may contain a *wait* statement. The use of a *wait* statement has two reasons:

* To suspend the execution of a process;
* To specify a condition that will determine the activation of the suspended process.

A process containing a *wait* statement cannot have a sensitivity list. The VHDL language allows several *wait* statements in a process. The process from example 1 can be rewritten using a *wait* statement (Example 2).

**Example 2**

proc2: process

begin

x <= a and b and c;

wait on a, b, c;

end process proc2;

There are 3 forms for the *wait* statement:

* wait on *sensitivity\_list*;
* wait until *conditional\_expresion*;
* wait for *time\_expression*;

The *wait until* statement suspends a process until the specified condition becomes true, due to a change of any of the signals listed in the conditional expression. The *wait for* statement allows suspending the execution of a process for a specified time (cannot be used for synthesis).

**Variables**

Because signals can only hold the last value assigned to them, they cannot be used to store intermediary results within a process. Also, the new values are not assigned to signals when the assignment statement executes, but only after the process execution suspends.

Variables can be declared inside processes and used within the process (local to that process). Some examples are below.

**Example 3**

variable a, b, c: bit;

variable x, y: integer;

variable index integer range 1 to 10 := 1;

variable cycle\_t: time range 10 ns to 50 ns := 10 ns;

variable mem: bit\_vector (0 to 15);

**If statement**

The *if* statement selects one or more statement sequences for execution, based on the condition corresponding to that sequence. The syntax is below:

if condition then statement\_sequence

[elsif condition then statement\_sequence...]

[else statement\_sequence]

end if;

An example of usage for the *if* statement is below, in Example 4.

**Example 4**

process (a, b)

begin

if a = b then

result <= 0;

elsif a < b then

result <= -1;

else

result <= 1;

end if;

end process;

**Case statement**

Similar to the *if* statement, the *case* statement selects for execution one of several alternative statement sequences, based on the value of an expression. Unlike the *if* statement, the expression does not need to be Boolean, but it may be represented by a signal, variable or expression of any discrete type or a character array type. The syntax is the following:

case expression is

when options\_1 =>

statement\_sequence

...

when options\_n =>

statement\_sequence

[when others =>

statement\_sequence]

end case;

Example 5 presents a process for sequencing through the values of an enumeration type representing the states of a traffic light (using a *case* statement).

**Example 5**

type type\_color is (red, yellow, green);

signal color, next\_color: type\_color;

process (color)

case color is

when red =>

next\_color <= green;

when yellow =>

next\_color <= red;

when green =>

next\_color <= yellow;

end case;

end process;

**Loop Statements**

*Loop* statements allow the repeated execution of a statement sequence (e.g. processing each element of an array). There are 3 types of *loop* statements in VHDL:

* simple *loop*
* *while loop*
* *for loop*

The simple *loop* statement specifies an indefinite repetition of some statements.

[label:] loop

statement\_sequence

end loop [label];

The *while loop* statement allows the loop body to be repeated until a condition specified becomes false.

**[label:]** **while** condition **loop**

statement\_sequence

**end** **loop** **[label];**

The code in Example 6 counts the rising edges of the clock signal (*clk*) while the *level* signal is ‘1’.

**Example 6**

process

variable count: integer := 0;

begin

wait until clk = '1';

while level = '1' loop

count := count + 1;

wait until clk = '0';

end loop;

end process;

The *for loop* statement allows the loop body to be repeated a specified number of times.

[label:] for counter in range loop

statement\_sequence

end loop [label];

Iterations in a loop can be skipped by using the *next* statement.

next [label] [when condition];

A loop can be stopped completely (forced) by using the *exit* statement.

exit [label] [when condition];

**Examples of sequential circuits**

Sequential circuits are a category of circuits that include storage elements. These circuits contain feedback loops from the output to the input. The signals generated at the outputs of a sequential circuit depend on both the input signals and on the state of the circuit. Synchronous circuits have a clock signal that controls any change of the states, thus, more reliable. An asynchronous circuit is less secure because the state evolution is also influenced by the delays of the circuit’s components.

In order to design sequential circuits, there are 2 well-known techniques: Mealy and Moore. Mealy sequential circuits have the output signals depend on both the current state and the present inputs. Moore sequential circuits have the outputs depend only on the current state and they do not depend directly on the inputs.

**Flip-Flops**

D-type flip-flops are basic storage elements. A simple design of such a flip-flop is described below.



Figure 1. D-type flip-flop

**Example 7**

library ieee;

use ieee.std\_logic\_1164.all;

entity dff is

port (clk: in std\_logic;

d: in std\_logic;

q: out std\_logic);

end dff;

architecture example of dff is

begin

process (clk)

begin

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

q <= d;

end if;

end process;

end example;

**Registers**

**Example 8**

library ieee;

use ieee.std\_logic\_1164.all;

entity reg8 is

port (clk: in std\_logic;

ce: in std\_logic;

d: in std\_logic\_vector (7 downto 0);

q: out std\_logic\_vector (7 downto 0));

end reg8;

architecture ex\_reg of reg8 is

begin

process (clk)

begin

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

if (ce = '1') then

q <= d;

end if;

end if;

end process;

end ex\_reg;

**Shift Registers**

**Example 9**

library ieee;

use ieee.std\_logic\_1164.all;

entity shift\_reg8 is

port (clk: in std\_logic;

ce: in std\_logic;

si: in std\_logic;

so: out std\_logic);

end shift\_reg8;

architecture shift\_reg of shift\_reg8 is

signal tmp: std\_logic\_vector (7 downto 0);

begin

process (clk)

begin

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

if (ce = '1') then

for i in 0 to 6 loop

tmp(i+1) <= tmp(i);

end loop;

tmp(0) <= si;

end if;

end if;

end process;

so <= tmp(7);

end shift\_reg;

**Counters**

**Example 10**

library ieee;

use ieee.std\_logic\_1164.all;

entity count3 is

port (clk: in std\_logic;

count: out integer range 0 to 7);

end count3;

architecture count3\_integer of count3 is

signal tmp: integer range 0 to 7;

begin

cnt: process (clk)

begin

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

tmp <= tmp + 1;

end if;

end process cnt;

count <= tmp;

end count3\_integer;

* 1. **Dataflow model - Concurrent statements**

Concurrent operations are used in real systems. Real models in VHDL are subsystems that operate concurrently and each of these subsystems may be specified as a separate process (communication is done via signals).

In the following sections, the structure and architecture of concurrent statements are described.

An architecture definition has 2 parts:

* Declarative part: definition of internal objects
* Statement part: concurrent statements which define the processes that describe the operations

While the processes in an architecture are executed concurrently with each other, the statements within a process are executed sequentially. Any suspended process can be activated again if one of the signals from its sensitivity list changes its value. Note that if a signal change its value and it is specified in the sensitivity list of multiple processes, then all processes are activated.

Consider the logic diagram of a full adder in Figure 1. The corresponding code is described in Example 7. Note that each gate is described by a separate process and all processes are executed concurrently.



Figure 2. Full adder logic diagram

**Example 11**

library ieee;

use ieee.std\_logic\_1164.all;

entity add\_1 is

port (a, b, cin: in std\_logic;

s, cout: out std\_logic);

end add\_1;

architecture processes of add\_1 is

signal s1, s2, s3, s4: std\_logic;

begin

p1: process (b, cin)

begin

s1 <= b xor cin;

end process p1;

p2: process (a, b)

begin

s2 <= a and b;

end process p2;

p3: process (a, cin)

begin

s3 <= a and cin;

end process p3;

p4: process (b, cin)

begin

s4 <= b and cin;

end process p4;

p5: process (a, s1)

begin

s <= a xor s1;

end process p5;

p6: process (s2, s3, s4)

begin

cout <= s2 or s3 or s4;

end process p6;

end processes;

Signals are used to communicate between processes, while variables are local objects and are used only within the process where they are declared. The signals can also be used to activate and synchronize processes. Thus, a signal declared in the declarative part of an architecture is visible for all processes within the architecture.

As explained before, every process consists of sequential statements, but process declarations are concurrent statements. The main features of a process are the following:

* Executed in parallel with other processes
* Cannot contain concurrent statements
* Defines a region of the architecture where statements are executed sequentially
* Must contain a sensitivity list or a **wait** statement
* Allows functional descriptions, allows access to signals defined in the architecture

The description of the full adder from Example 1 can be simplified by using concurrent assignment statements, directly in the architecture (Example 8).

**Example 12**

library ieee;

use ieee.std\_logic\_1164.all;

entity add\_1 is

port (a, b, cin: in std\_logic;

s, cout: out std\_logic);

end add\_1;

architecture concurrent of add\_1 is

signal s1, s2, s3, s4: std\_logic;

begin

s1 <= b xor cin;

s2 <= a and b;

s3 <= a and cin;

s4 <= b and cin;

s <= a xor s1;

cout <= s2 or s3 or s4;

end concurrent;

Conditional assignment statements are functionally equivalent to the *if* statement, with few minor differences.

signal <= [expression when condition else ...]

expression;

While the conditional assignment statement is a concurrent statement, and thus, can be used in an architecture, the *if* statement is a sequential statement (can be used only inside a process). Also, the conditional assignment statement can only be used to assign values to signals, while the *if* statement can be used to execute any sequential statement.

The example below defines an entity and 2 architectures for a 2-input XOR gate.

**Example 13**

library ieee;

use ieee.std\_logic\_1164.all;

entity xor2 is

port (a, b: in std\_logic;

x: out std\_logic);

end xor2;

architecture arch1\_xor2 of xor2 is

begin

x <= '0' when a = b else

'1';

end arch1\_xor2;

architecture arch2\_xor2 of xor2 is

begin

process (a, b)

begin

if a = b then x <= '0';

else x <= '1';

end if;

end process;

end arch2\_xor2;

Like the conditional signal assignment statement, selected signal assignment statement allows to select a source expression based on a condition.

with selection\_expression select

signal <= expression\_1 when options\_1,

...

expression\_n when options\_n,

[expression when others];

This statement is similar to the *case* sequential statement, but has some constraints, for example: if the *others* option is missing, all the possible values of the selection expression must be covered by the set of options.

Example 10 describes the functionality of a 2-input XOR gate by using a selected signal assignment statement.

**Example 14**

library ieee;

use ieee.std\_logic\_1164.all;

entity xor2 is

port (a, b: in std\_logic;

x: out std\_logic);

end xor2;

architecture arch\_xor2 of xor2 is

signal tmp: std\_logic\_vector (1 downto 0);

begin

tmp <= a & b;

with tmp select

x <= '0' when "00",

'1' when "01",

'1' when "10",

'0' when "11";

end arch\_xor2;

**Examples of combinational circuits**

**Multiplexers**

Multiplexers are circuits that output an input signal, based on a select signal. The figure and example below illustrates a 4-to-1 multiplexer (selected signal assignment), with inputs of 4 bits.

****

Figure 3. Block diagram of a 4:1 multiplexer

**Example 15**

library ieee;

use ieee.std\_logic\_1164.all;

entity mux is

port (a, b, c, d: in std\_logic\_vector (3 downto 0);

s: in std\_logic\_vector (1 downto 0);

x: out std\_logic\_vector (3 downto 0));

end mux;

architecture arch\_mux of mux is

begin

with s select

x <= a when "00",

b when "01",

c when "10",

d when "11",

d when others;

end arch\_mux;

**Decoders**

A decoder is a combinational circuit that identifies an input code by asserting a single output line, corresponding to the input code. A decoder with *n* input lines has 2n output lines. The example below describes a 1-to-8 decoder with active-high outputs.

**Example 16**

library ieee;

use ieee.std\_logic\_1164.all;

entity decoder\_1\_8 is

port (a: in std\_logic\_vector (2 downto 0);

y: out std\_logic\_vector (7 downto 0));

end decoder\_1\_8;

architecture decod of decoder\_1\_8 is

begin

y <= "00000001" when a = "000" else

"00000010" when a = "001" else

"00000100" when a = "010" else

"00001000" when a = "011" else

"00010000" when a = "100" else

"00100000" when a = "101" else

"01000000" when a = "110" else

"10000000";

end decod;

**Combinational shifters**

A combinational shifter performs a logical or arithmetic shift operation on the input data. The shifter also has a selector input whose binary value specifies the shift distance. The example below describes a combinational shifter for 8-bit vectors that can be shifted left with 1, 2 or 3 positions.

**Example 17**

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity shift\_left is

port (din: in unsigned (7 downto 0);

sel: in unsigned (1 downto 0);

dout: out unsigned (7 downto 0));

end shift\_left;

architecture arch\_shift of shift\_left is

begin

with sel select

dout <= din when "00",

din sll 1 when "01",

din sll 2 when "10",

din sll 3 when others;

end arch\_shift;

* 1. **Structural model**

Structural descriptions specify a system as a set of interconnected components. These allow creating multiple *hierarchical levels,* in which a design is divided into smaller design units. Structural design can be achieved by using components. Thus, complex systems can be built in several steps from lower-level components.

The main advantages of structural design are:

* Each component may be design and tested individually before being integrated into higher levels of the design. This intermediate level testing is simpler than system testing and more thorough.
* Useful component can be collected into libraries, so they can be reused later. One of the advantages of logic synthesis is that such components or modules are technology independent.

A structural description consists of components interconnected by signals. Every component that will be used in a structural architecture description must be defined by a *component* declaration. Afterwards, in order to use a component, it must be *instantiated* within the structural description. In any component instantiation, the *port mapping* is specified, which indicates the signals connected to the component’s ports.

The syntax for a component declaration is the following:

component component\_name [is]

generic (generic\_list);

port (port\_list);

end component [component\_name];

The *generic* clause specifies the generics of the component, while the *port* clause specifies its ports. In practice, the name of the component, the name of its generics and ports, as well as their order, are identical to the elements that appear in the entity declaration corresponding to the component. Note that a component can be declared in an architecture, a block, an entity or in a package, and every component instantiation is a concurrent statement.

A component instantiation associates signals or values with the ports of a component. The syntax is below.

label: [component] component\_name

[generic map (generic\_association\_list)]

port map (port\_association\_list);

Direct entity instantiation is also possible, so it’s not always necessary to define a component in order to instantiate it.

label: entity library\_name.entity\_name

[(architecture\_name)]

[generic map (generic\_association\_list)]

port map (port\_association\_list);

Consider the structural description example of 2 D-type flip-flops connected in series as a pipeline. The figure below illustrates the circuit structure.



Figure 4. Structural description example of 2 D-type flip-flops

We assume the D-type flip-flop is already defined, as presented below in Example 11.

**Example 18**

library ieee;

use ieee.std\_logic\_1164.all;

entity dff is

port (d, clk: in std\_logic;

q, qn: out std\_logic);

end dff;

architecture arch\_dff of dff is

signal tmp: std\_logic;

begin

process (clk)

begin

if rising\_edge (clk) then

tmp <= d;

end if;

end process;

q <= tmp;

qn <= not tmp;

end arch\_dff;

Example 12 below is a possible description of the circuit using components.

**Example 19**

library ieee;

use ieee.std\_logic\_1164.all;

entity delay2 is

port (din, clock: in std\_logic;

qout: out std\_logic);

end delay2;

architecture structural of delay2 is

signal intern: std\_logic;

-- Component declaration

component dff is

port (d, clk: in std\_logic;

q, qn: out std\_logic);

end component dff;

-- Configuration specification

for all: dff use entity work.dff (arch\_dff);

begin

-- Component instantiation

d1: dff port map

(d => din, clk => clock, q => intern, qn => open);

d2: dff port map

(d => intern, clk => clock, q => qout, qn => open);

end structural;

1. **Applications**
   1. Explain the following differences:

* Signal assignment vs. variable assignment
* If statements with signals vs. if statements with variables
* If statements vs. case statements
  1. Write the VHDL code for the circuits listed below using the Xilinx ISE development system and compile each code. Identify and correct any errors.
* XOR gate
* D-type Flip-Flop
* Register
* Shift register
* Counter
* Multiplexer
* Decoder
  1. Design an 8-bit 4:1 demultiplexer using a *case* statement.
  2. Design a BCD decoder for a 7-segment display.