Programming elements in VHDL

1 Purpose

This laboratory work introduces basic programming elements in VHDL. The most important syntactic structures are presented and exemplified. The focus is on how VHDL can be used to develop various types of digital systems.

2 Introduction to VHDL

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 (**FPGA**s) 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 shows 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 `SCS_L03_VHDL_Annex`).

3 Programming in VHDL

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. These programming models are discussed as follows.

3.1 Behavioral model. Sequential statements

The behavioral model aims to describe the way a system is meant to be used and is based on processes and sequential statements.

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

3.1.1 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;
```

3.1.2 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 expression>`
- `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).

3.1.3 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);
```

3.1.4 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;
```

3.1.5 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;
```

3.1.6 Loop statement

`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];
```

3.1.7 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.

1. **Flip-Flops**

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

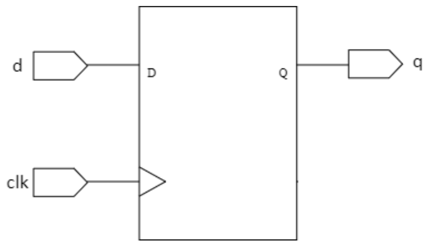

Figure 1: D-type Flip-Flop

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;
```

2. **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;
```

3. 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;
```

4. Counters

Example 10.1 3-bit counter

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

entity count3 is
    port (clk: in std_logic;
        count: out std_logic_vector (2 downto 0));
end count3;
```

```
architecture Behavioral of count3 is
    signal tmp : std_logic_vector (2 downto 0) := (others => '0');
begin
    cnt: process (clk)
    begin
        if (clk'event and clk = '1') then
            tmp <= tmp + 1;
        end if;
    end process cnt;

    count <= tmp;
end Behavioral;
```

Example 10.2 Testbench for the 3-bit counter

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity tb_counter3 is
end tb_counter3;

architecture Tb of tb_counter3 is

component count3 is
    port (clk: in std_logic;
        count: out std_logic_vector (2 downto 0));
end component;

constant T : time := 10 ns;
signal clk : std_logic := '0';
signal count : std_logic_vector (2 downto 0) := (others => '0');

begin
    clk <= not clk  after T / 2;

    cnt3 : count3 port map (
        clk => clk,
        count => count
    );
end Tb;
```

3.2 Dataflow model. Concurrent statements

3.2.1 Basic concepts of the concurrent programming models

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 subsections, 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 2. The corresponding code is described in Example 11. Note that each gate is described by a separate process and all processes are executed concurrently.

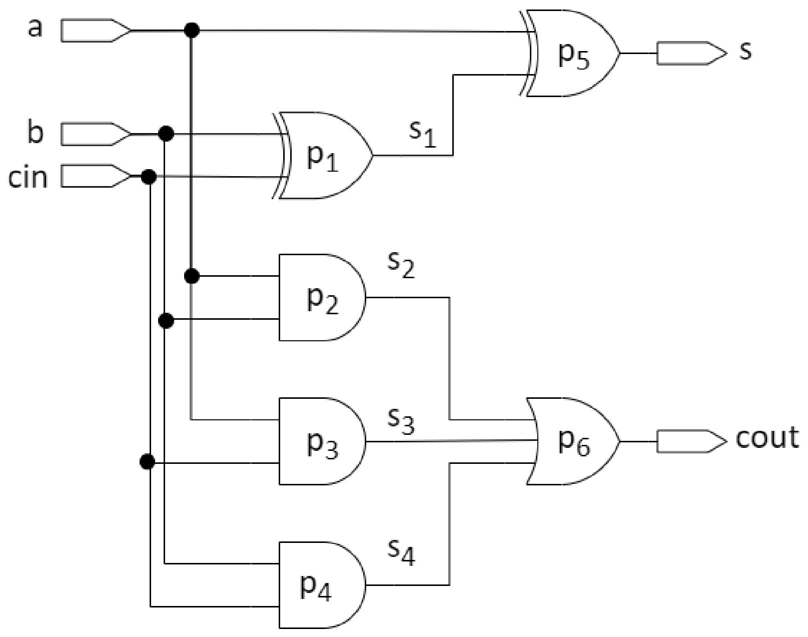

Figure 2: Full-adder logic diagram

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:

- Is 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 11 can be simplified by using concurrent assignment statements, directly in the architecture (Example 12).

Example 12.1 1-bit full-adder

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

    entity full_adder is
    port (a, b, cin: in std_logic;
            s, cout: out std_logic);
    end full_adder;

    architecture Dataflow of full_adder 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 Dataflow;
```

Example 12.2 Testbench for the 1-bit full-adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity tb_full_adder is
end tb_full_adder;

architecture Tb of tb_full_adder is

component full_adder is
port (a, b, cin: in std_logic;
        s, cout: out std_logic);
end component;

constant T : time := 10 ns;
signal a, b, cin, s, cout : std_logic := '0';

begin
    cin <= not cin after T;
    a <= not a after 2 * T;
    b <= not b after 4 * T;

    adder : full_adder port map (
        a => a,
        b => b,
        cin => cin,
        s => s,
        cout => cout
    );
end Tb;
```

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 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 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 14 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;
```

3.2.2 Examples of combinatorial circuit

1. **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 4-bit inputs.

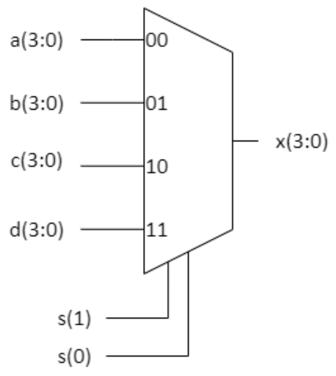

Figure 3: Block diagram of a 4:1 multiplexer

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;
```

2. **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;
```

3. Combinatorial shifters

A combinatorial 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 combinatorial 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;
```

3.3 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 involves 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 ports of the component.

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 given 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, as shown below.

```
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 shown in Figure 4. We assume the D-type flip-flop is already defined, as presented below in Example 18.

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;
```

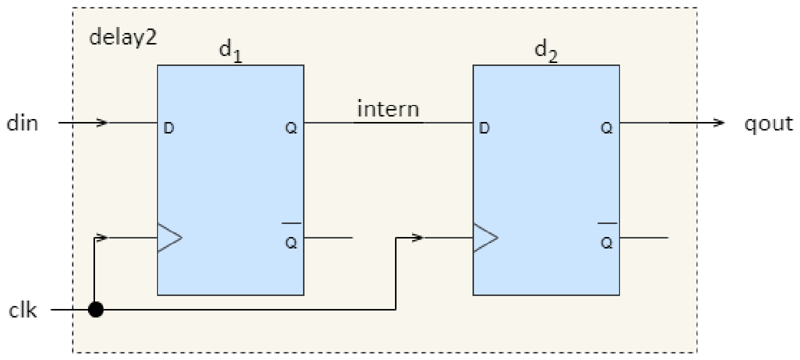

Figure 4: Structural description example of 2 D-type flip-flops

Example 19 below is a possible structural description of the circuit in Figure 4 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;
```

4 Exercises

1. **Explain the differences between**

   a. Signal assignment vs. variable assignment

   b. If statements with signals vs. if statements with variables

   c. If statements vs. case statements

2. **Create a new Vivado RTL project targeting the Basys 3 development board**. For this, open Vivado, click on *File* → *Project* → *New...* and in the opened window click *Next*. Provide the path or browse to the directory where you want to create the project and click *Next*. In the *Project Type* window select *RTL Project* and tick the *Do not specify sources at this time* option. The device properties that you need to choose are:

   - **Product category:** All
   - **Family:** Artix-7
   - **Package:** cpg236
   - **Speed grade:** -1

   Then select from the table the **xc7a35tcpg236-1** board and click *Next*. Click *Finish*.

3. **Write the VHDL code for the circuits listed below within the project created at exercise 2 and compile each code. Identify and correct any errors. Create an individual testbench for each of the designs and simulate the behavior of the implemented designs.**

   a. XOR gate (dataflow model)

   b. D-type Flip-Flop

   c. Register

   d. Shift register

   e. Counter

   f. Multiplexer

   g. Decoder

4. **Design an 8-bit 4:1 demultiplexer using a `case` statement.**

5. **Design a BCD decoder for a 7-segment display.**

References

[1] VHDL and FPGA terminology [accessed Sept. 2024], https://vhdlwhiz.com/terminology/

[2] VHDL snippet library [accessed Sept. 2024], https://vhdlwhiz.com/snippets/

[3] How to use a For loop in VHDL [accessed Sept. 2024], https://vhdlwhiz.com/for-loop/