Digital Electronics and VHDL

Practical 1 - combinational logic using a dataflow style

# Introduction (please read)

This practical is intended to provide you with an introduction to the ModelSim ~~Quartus II~~ tools and to introduce you to writing VHDL. Everything we will do (for now) will be focused on creating **combinational logic**. Sequential logic fill follow in later weeks.

There is one topic per week on VHDL. **These are intended to work as remote self-study tasks.** There is a free version of Quartus and ModelSim available from the Intel / Altera website. The Lite version is perfectly adequate for home study. The software is also packaged and deployed to open access areas across the university, including Babbage, Smeaton and the Library. The majority of VHDL development uses a simulator. Therefore, the majority of the tasks this semester are simulation based. This makes it simpler to work at home (on even on a bus or train!).

## **IMPORTANT UPDATES – READ THIS BEFORE YOU START**

With each year, there may be some minor updates to the tools. These are documented here. Please read this section before you begin.

### Target FPGA?

In semester 1, we focus on ModelSim where we are not concerned with physical devices. In semester 2, will be using a **Cyclone® IV EP4CE22F17C6.** Sometimes, we might use Quartus to generate VHDL for us. In this case, use this device when setting up your project.

### Quartus II and the Vector Waveform Editor

In earlier versions of this course, we use the built-in vector waveform editor to create test signals in Quartus. We are no longer using or supporting these at level 5. You will be using ModelSim for all your testing.

### Sample and Starter Code

All starter projects can be [found on GitHub here](https://github.com/UniversityOfPlymouth-Electronics/ELEC240) . You are encouraged to install [github desktop](https://desktop.github.com/)

### VIDEOS

The practical materials will often refer to videos. These are now all held on the module site (that’s our private equivalent to YouTube 😊). [Use this link to see all videos](https://plymouth.cloud.panopto.eu/Panopto/Pages/Sessions/List.aspx#folderID=%22d90459fa-4e14-4f5a-8254-ac7c00c9b478%22)

# 01 - Creating a project in MODELSIM

In this section, you are going use ModelSim to simulate some existing VHDL. First, you should obtain the same code for the lab(s).

## TASK 01-1.

1. Watch the video "[**01-01-Obtaining the sample and starter code**](https://plymouth.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=337e95ca-f711-494f-be02-ac7c00eb3698)"
2. Watch the video **"**[**01-01-Compile and Simulate in ModelSim**](https://plymouth.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=34b9b8d4-f01e-4179-8862-ac7c00ed185b)" and try to replicate this yourself
3. Replicate what you saw in this video and run the simulation
4. Now create another component called myor2, which outputs the logical OR of inputs A and B.
   1. Add an additional file in **ModelSim**.
   2. Make sure the filename matches the entity name (myor2.vhdl)
   3. Copy and paste the code from myand2, and modify it (careful!).
   4. The entity name should be myor2.

The VHDL for part 3 is shown below.

|  |
| --- |
| myand.vhdl |
| **entity** myand2 is  port  (  -- Input ports  A : in bit;  B : in bit;  -- Output ports  Y : out bit  );  end myand2;  **architecture** myand2\_v1 of myand2 is  **begin**  Y <= A and B;  **end** myand2\_v1; |

See **Appendix A** for more information on entities and architectures.

**Notes:**

The entity is called "myand2" and the vhdl file shares the same name.

The inputs are labeled A and B and their **type** is 'bit'

In the architecture, there is one line that specifies any logic:

Y <= A **and** B;

The **and** keyword is a reserved word, known as an **operator**

See Appendix B for a list of different data types and operators.

## TASK 01-2

Using the style of VHDL above, **create and exhaustively test\*\*** a VHDL component that replicates the following truth tables:

1. 2-Input component “task1p2a.vhdl

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **Y** |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

1. 3-Input component “task1p2b.vhdl

|  |  |  |  |
| --- | --- | --- | --- |
| **A** | **B** | **C** | **Y** |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

**Hint** - look at appendix B for clues about logic operators.

Show the tutor your simulation results when done.

# 02 – DATA FLOW CONCURRENT statements

Designing concurrent (parallel) combinational logic with Boolean primitive functions, such as **and**, **or**, **not**, is not the only option. There are other ways, which may be simpler to write and read.

## Vectors

So far, we've only worked with single logic lines of type '**bit**'. Many digital systems use an N-bit bus, such as a data bus or address bus, to represent integer values.

The pre-defined data type for an N-bit value is **bit\_vector**.

When you declare a signal of type bit vector, you must tell it how many bits wide it actually is. For example:

|  |
| --- |
| myand.vhdl |
| **entity** vector\_bits **is**  **port**  (  X : **in** **bit\_vector**(7 downto 0); -- 8-bit value  Y : **out** **bit\_vector**(1 downto 0) -- 2 bit output  );  **end** vector\_bits;  **architecture** circuit1 **of** vector\_bits **is**  **begin**  Y(0) <= X(0) **and** X(1) **and** X(2) **and** X(3); -- AND the 4 lsb’s  Y(1) <= X(4) **and** X(5) **and** X(6) **and** X(7); -- AND the 4 msb’s  **end** circuit1; |

Here we see two **bit\_vectors**. X is an 8-bit vector and Y is a 2-bit vector.

**You access individual 'bits' or sub-vectors using round braces ( )**.

## Task 02-01

Sketch on paper a schematic for myand.vhdl.

Consider what inputs you might apply to test this circuit.

Now watch the video "[**02-01-Vectorised Signals**](https://plymouth.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=eaabc2e4-87b3-49be-87db-ac7c00f2a202)" to see how to compile and test this VHDL in ModelSim.

Now build it yourself and test with the simulator.

## vector manipulation

Consider an 8-bit output signal of type bit\_vector called **databus** such that

databus : out **bit\_vector(7 downto 0)**

and another one-bit signal called **abit** of type bit such that

abit : out bit

|  |  |
| --- | --- |
| Setting an individual bit | databus(0) <= '1';  databus(0) <= '0';  databus(0) <= abit; |
| Setting a range of bits | databus <= "10101010"; --note that double-quotes are needed  databus <= "0000000" & '1' --concatenation using &  databus <= "0000000" & '1' --concatenation  databus<= abit & "000000" & not abit --concatenation  databus <= ('0', others => '1'); --concatenation using others  databus <= databus(6 downto 0) & '0'; --shift left |

## TASK 02-01

Create and test a logic device with an 8-bit input X and an 8-bit output Y (both bit\_vectors)

* Y should be the bit-reversal of X
* For example, if X="11010001" then Y="10001011"

(Note, using what you know, there is no concise way to do the above)

## TASK 02-02

If X is an 8-bit bit\_vector, and X = 11010001, write down the results of the following VHDL statements:

(you can use Quartus to help you)

|  |  |
| --- | --- |
| VHDL | Result |
| X & '1' | (9-bit) |
| '1' & X & '0' | (10-bit) |
| X(0) & X(1) & X(2) & X(3) | (4-bit) |
| X(4 downto 2) | (3-bit) |

**Write/modify some VHDL to verify your results**

**Notes:**

When we extract a sub-array, this is known as a **slice**.

"The direction of the parent array and its slice must match (i.e. in both cases either to or downto keyword must be used)" (<http://www.vhdl.renerta.com/source/vhd00065.htm> , accessed 06/10/2010)

where Y is declared as bit\_vector(3 downto 0)

Y <= X(3 downto 0) is perfectly legal, whereas

Y<= X(0 to 3) is not.

If you want to perform bit-reversal, you need to use a different method (later!).

## WHEN-ELSE

A useful conditional statement is the when-else combination. Consider the following VHDL entity:

|  |
| --- |
| vector\_scrunge.vhdl |
| **entity** vector\_scrunge is  port  (  -- Input ports  X : in bit\_vector(7 downto 0);  A : in bit;  -- Output ports  Y : out bit\_vector(7 downto 0)  );  **end** vector\_scrunge;  **architecture** circuit1 of vector\_scrunge is  begin  Y <= X when A = '0' else  not X when A = '1';    **end** circuit1; |

We will see more examples of this later. **See Appendix C for more information.**

## TASK 02-03

What does this code do?

Use ModelSim to build this code and test it.

**Note** how the operator **not** (as well are and, or etc..) can operate on vectors as well as individual bits.

## Decoder

|  |
| --- |
| vector\_decoder.vhdl |
| **entity** vector\_decoder is  port  (  -- Input ports  S : in bit\_vector(2 downto 0); --3-bit select  -- Output ports  Y : out bit\_vector(7 downto 0) --output  );  **end** vector\_decoder;  **architecture** circuit1 of vector\_decoder is  begin  Y <= "00000001" when S = "000" else  "00000010" when S = "001" else  "00000100" when S = "010" else  "00001000" when S = "011" else  "00010000" when S = "100" else  "00100000" when S = "101" else  "01000000" when S = "110" else  "10000000";    **end** circuit1; |

The circuit about is a 3-to-8 decoder. A binary input S specified which of the 8 output bits should become HIGH.

## TASK 02-04

Modify the VHDL above to that the output Y is active LOW (that is, all output bits are HIGH except the selected output). Show your test results using ModelSim.

## SELECT STATEMENTS

Another somewhat simpler way to implement a 3-to-8 decoder is to use a **with-select** statement. This is rather similar to a C-programming **switch-case** statement. This is a suitable technique if the Boolean expression being tested in always on the same signal (in this case S).

|  |
| --- |
| Select statement example |
| **with** (S) **select**    Y <= "00000001" when "000",  "00000010" when "001",  "00000100" when "010",  "00001000" when "011",  "00010000" when "100",  "00100000" when "101",  "01000000" when "110",  "10000000" when **others**; |

## TASK 02-05

Build, modify and verify that an active LOW 3-to-8 decoder can be implemented this way.

## TASK 02-06

**Create** and **test** a logic device which does the following:

X is a 7-bit input (0..127 ASCII character)

Y is a 10-bit output frame

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| START | X(6) | X(5) | X(4) | X(3) | X(2) | X(1) | X(0) | PARITY | STOP |

* The most significant bit is a '1' (start bit)
* The next most significant bits are X
* The next is the parity bit (bit 1)
* The last bit (least significant bit) should be a '1' (stop bit)

### Multiple Choices

Sometimes you want to same output for multiple inputs. In this case, you can use the following pattern:

|  |
| --- |
| Select statement multiple choices (example) |
| **with** (S) **select**    Y <= "001" when "000" | "001" | "010",  "010" when "011" | "100",  "100" when **others**; |

The above could be the basis of an address decoder for example.

# 03 - Signals

So far, the only signals you have seen are inputs and outputs. However, from the definition of the architecture block in Appendix A, you will see a reference to 'local' **signals**.

Signals are typically wires or nodes. They are not to be confused with 'variables' (covered later) which can imply storage.

Consider the following example. The architecture now has some code in the declarations area. These signals are intermediate wires. This helps break a problem down into multiple lines.

|  |
| --- |
| **Example** - Bit scrambler |
| **entity** vector\_bits **is**  port  (  X : in bit\_vector(7 downto 0);  Y : out bit\_vector(7 downto 0)  );  **end** vector\_bits;  **architecture** circuit1 **of** vector\_bits **is**  -- Declarations (optional)  **signal** Y1 : bit\_vector(7 downto 0);  **constant** Y2 : bit\_vector(7 downto 0) := "10110010"; -- private key  **signal** Y3 : bit\_vector(7 downto 0);  **begin**  -- scramble all the bits with xor, this is reversible  Y1 <= X xor Y2;  -- scramble the positions of the bits  Y3 <= Y1(4 downto 2) & Y1(1 downto 0) & Y1(7 downto 5);  Y <= Y3;  **end** circuit1; |

The VHDL above is designed to scramble an 8-bit signal. There are two steps:

* xor the input value with a constant bit-pattern
* scramble the bit order

Although weak, this is a form of encryption. To reverse this (to decrypt), you would do the following:

* re-order the bits
* xor the date with the same bit-pattern

## TASK

Build another entity that decrypts the data as described above.

Cascade the encryption and decryption blocks and demonstrate the original data can be recovered.

# Appendix A – ENTITIES AND ARCHITECTURES

## Entity

**entity** entity-name **is**

**port** (signal-names : mode signal-type [ := initial value ];

**port** (signal-names : mode signal-type [ := initial value ];

**port** (signal-names : mode signal-type [ := initial value ]

**end** entity-name;

**NOTE** – no semi-colon here!

|  |  |
| --- | --- |
| **Item** | **DESCRIPTION** |
| Entity-name | A name you choose, that matches the filename |
| Signal-names | A comma separated list of one or more input or output signals |
| Mode | This can be:  in – input  out – output  buffer – an output that can be read from within the architecture  inout – input or output, normally associated with tri-state outputs on PLD’s |
| Signal-type | The signal type. See Appendix B for pre-defined types. You can also create your own. |

## Architecture

**architecture** architecture-name if entity-name **is**

-- local variables, types etc…

type declarations

signal declarations

constant declarations

function definitions

procedure definitions

component declarations

**begin**

concurrent statement 1

concurrent statement 2

**end** architecture-name;

# APPENDIX B – PREDEFINED TYPES AND OPERATORS

## VHDL PREDEFINED TypeS

|  |  |
| --- | --- |
| **TYPE** | **DESCRIPTION** |
| bit | Single bit that takes values '0', '1' |
| bit\_vector | Vector (array) of bits |
| boolean | *true* or *false* |
| character | ISO 8-bit character |
| integer | Whole number between |
| real | Fractional numbers |
| severity\_level |  |
| string |  |
| time |  |

## VHDL INTEGER Operators

|  |  |
| --- | --- |
| **OPERATOR** | **DESCRIPTION** |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division |
| Mod | Modulo division |
| Rem | Modulo remainder |
| Abs | Absolute value |
| \*\* | Exponentiation |

## VHDL BINARY OPERATORS

|  |  |
| --- | --- |
| **OPERATOR** | **DESCRIPTION** |
| and | AND |
| or | OR |
| nand | NAND |
| nor | NOR |
| xor | Exclusive OR |
| xnor | Exclusive NOR |
| not | Compliment (Inverter) |

# Appendix C - CONCURRENT STATEMENTS

## When-Else

*signal-name* <= *expression* **when** *boolean-expression* **else**

*expression* **when** *boolean-expression* **else**

...

...

*expression* **when** *boolean-expression* **else**

*expression*;

## SELECT

**with** *expression* **select**

*signal-name* <= *signal-value* **when** *choices*,

*signal-value* **when** *choices*,

...

..

*signal-value* **when** *choices,*

*signal-value* **when****others**;