Digital Electronics and VHDL

Practical 4 - combinational logic Part 3 - BEHAVIOURAL style VHDL

Table of Contents

[**IMPORTANT UPDATES – READ THIS BEFORE YOU START** 2](#_Toc530575475)

[Target FPGA 2](#_Toc530575476)

[Quartus II and the Vector Waveform Editor 2](#_Toc530575477)

[Quartus User Interface 2](#_Toc530575478)

[Introduction 4](#_Toc530575479)

[01 - concurrent statements and process blocks 4](#_Toc530575480)

[Process blocks 5](#_Toc530575481)

[Sensitivity List 5](#_Toc530575482)

[Local Declarations 5](#_Toc530575483)

[Sequential statements 5](#_Toc530575484)

[example 1-1 process block 6](#_Toc530575485)

[TASK 01-01 Simple combinational logic 7](#_Toc530575486)

[02 - Conditional Statements 7](#_Toc530575487)

[Task 02-01 if statement 7](#_Toc530575488)

[TASK 01-03 IF STATEMENT 8](#_Toc530575489)

[Task 02-03 Latching behaviour 9](#_Toc530575490)

[TASK 02-04 CASE STATEMENT 10](#_Toc530575491)

[03 - Looping 11](#_Toc530575492)

[Task 03-01 for loop 11](#_Toc530575493)

[An important note on variables and signals 12](#_Toc530575494)

[TASK 03-02 - type conversion 12](#_Toc530575495)

[Appendix A – entities and architectures 13](#_Toc530575496)

[Entity 13](#_Toc530575497)

[Architecture 13](#_Toc530575498)

[APPENDIX B – PREDEFINED TYPES AND OPERATORS 14](#_Toc530575499)

[VHDL PREDEFINED TypeS 14](#_Toc530575500)

[VHDL INTEGER Operators 14](#_Toc530575501)

[VHDL BINARY OPERATORS 14](#_Toc530575502)

[Appendix C - Concurrent statements 15](#_Toc530575503)

[When-Else 15](#_Toc530575504)

[SELECT 15](#_Toc530575505)

[Appendix D - TYPE and subtype DEFINTIONS 16](#_Toc530575506)

[Appendix E - Arrays 16](#_Toc530575507)

[APENDIX F - IEEE STD\_ULOGIC and STD\_LOGIC 16](#_Toc530575508)

[Appendix G - Structural statements 17](#_Toc530575509)

[component Declaration 17](#_Toc530575510)

[Instantiation 17](#_Toc530575511)

[Generate 17](#_Toc530575512)

[Generic Declarations 17](#_Toc530575513)

[Appendix H - Behavioural Statements 18](#_Toc530575514)

[process statement 18](#_Toc530575515)

[if statement 19](#_Toc530575516)

[If-ELSe 19](#_Toc530575517)

[if-elsif 19](#_Toc530575518)

[if-elsif-ELSE 19](#_Toc530575519)

[CASE 20](#_Toc530575520)

[LOOP 20](#_Toc530575521)

[FOR LOOP 20](#_Toc530575522)

[while loop 21](#_Toc530575523)

## **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

The Cyclone II device sometimes featured in the videos is no longer supported in Quartus. We are currently using a **Cyclone® IV EP4CE22F17C6**

### Quartus II and the Vector Waveform Editor

Since version 13, Altera reinstated the vector waveform editor directly into Quartus II. This has not changed since v13. Some of the videos may make reference to an external tool. The tool is fundamentally the same, but now you can add a “vector waveform file” directly into your project from Quartus (File->New->… VWF)

To see how this has changed, please refer to the video “Using Quartus 13+ with VWF HD.mp4”

### Quartus User Interface

With each version that is released, the user interface can sometimes change in appearance. With version 16, the user interface has been noticeably re-skinned. However, the same basic functionality is still available from the menus as tool bar (the icons are now more colourful). To find the new icon, if necessary you can hover your mouse over the toolbar buttons, and you will see a text prompt.

# Introduction

This practical considers the 'behavioural style' of VHDL for simulating and synthesising combination logic. It also introduces some more key elements of the VHDL language, including:

* **variables**
* **process blocks**
* **sensitivity lists**
* **if statements**
* **for loops**
* **latching behaviour**

# 01 - concurrent statements and process blocks

Everything you have done in the VHDL until now has been combination logic, that is, no clock signals have been required. This is the last session exclusively on combinational logic.

All the VHDL statements you have used so far have been 'concurrent' statements. The order in which they were specified was not important. There is one other type of statement, a "process block", which also runs concurrently with all other statements. However, **within** the process block, you can write sequential statements like a conventional programming language. These statements still only **describe hardware** and are said to execute in zero time, thus are **simply a sequential way to describe hardware**, and not (directly) a way in which to synthesise sequential logic. Confused yet? Don't feel bad if you are. This **is** hard to understand and you will most likely need to re-read this paragraph having worked through all the examples!

**ENTITY**

**ARCHITECTURE**

Concurrent Statement

Concurrent Statement

**(process block)**

sequential statement

sequential statement

sequential statement

Concurrent Statement

## Process blocks

As stated above, process blocks are themselves concurrent statements. Within these process blocks you can write sequential statements to *describe* the logic you wish to synthesise or simulate. Although you write process blocks in a sequential manner, what is produced his concurrent hardware.

The syntax of a process statement is shown in appendix H, and repeated below.

**process** (signal-name, signal-name, ..., signal-name)

type declarations

variable declarations

constant declarations

function definitions

procedure definitions

Sensitivity list

**begin**

sequential statement

sequential statement

Local declarations

...

sequential statement

**end process;**

Sequential statements

Let's look at the different sections of this before we look at some examples.

### Sensitivity List

**process** (signal-name, signal-name, ..., signal-name)

We start the process statement itself which takes a number of optional arguments. These arguments will be signal-names, either internal signals or inputs /outputs from the entity declaration. This list of signals is also known as the **sensitivity list**. Statements within the process block that follow are only invoked when one of the signals in the sensitivity list changes.

### Local Declarations

Next come local declarations. A process block can see the signals in the architecture in which it is enclosed. However it can also have its own private declarations. This can include variable types, 'variables', constants, functions and procedures. These are new concepts, which I will gradually introduce over the next few weeks.

type declarations

variable declarations

constant declarations

function definitions

procedure definitions

What is important to note is that there are no signal declarations. Signal declarations are not allowed inside process blocks as they have no meaning. Instead you use variables much as you would in a programming language.

### Sequential statements

Finally there is the code block itself. This will be VHDL that can include conditional and looping instructions such as for loops and if statements. Again, it's important to emphasize that this will be a sequential way of specifying hardware that may ultimately be synthesized into sequential logic and/or combination logic. Effectively what happens in simulation is that all statements are executed and repeated until the signals in the sensitivity lest cease to change. It is therefore important to ensure that the signals reach a steady state. Failure to do so will result in an error which (hopefully) the VHDL compiler will detect. The cleverest part is that this can also be synthesized in hardware! You'll probably want to come back to this once you've seen a few examples, so once again, don't feel bad if you feel confused at this point.

## example 1-1 process block

* Open the project in the folder Task 01 - 01
* Inspect the VHDL code
* Build and run a timing-simulation for the circuit (there is a vector waveform file included)
* How does the output of the VHDL compared to the standard library function?

The entity declaration is shown below. It is simply the declaration for an and gate.

**entity** and2\_behavioural **is**

**port**

(

-- Input ports

A : **in** std\_logic;

B : **in** std\_logic;

-- Output ports

Y : **out** std\_logic

);

**end** and2\_behavioural;

The architecture definition is shown below. Note that one of the statements in this architecture is a process block. The sensitivity list, A and B, happened to be the entity's inputs. When the compiler sees either A or B changing, then the code with in the process block will run in zero time. Therefore the **whole** process block is seen as one single concurrent statement.

**architecture** and2\_behavioural\_v1 **of** and2\_behavioural **is**

**begin**

-- Process Statement (optional)

plist1: --optional label

**process**(A,B) is

-- Declaration(s)

**variable** P : std\_logic;

**begin**

-- Sequential Statement(s)

P := A and B;

Y <= P;

**end process**;

**end** and2\_behavioural\_v1;

Note that the process block has a local variable P. The assignment operator for a variable is different to that of the signal. For variables, you use the := symbol to help differentiate between variables and signals. However it is quite legal to assign a variable to a signal as is done in the following line, Y <= P. The difference is subtle but important (as discussed in the lecture). See the section on looping for more information.

## TASK 01-01 Simple combinational logic

* Modifying the architecture block in the preceding example to implement the following truth table

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

With such a simple example, you might be wondering what possible benefits behavioral VHDL brings us. In the next example we will see the use conditional statements which would otherwise be illegal outside the process block.

# 02 - Conditional Statements

There are a number of conditional statements you can use in behavioral VHDL, including:

if-elsif-else

case statements

## Task 02-01 if statement

Take a look at APENDIX H, specifically the syntax for the variations on the **if** statement. We're now going to use an **if** statement to create some combinational logic. In this task we build an entity called "clevergate\_behavioural".

entity clevergate\_behavioural is

port

(

-- Input ports

A : in std\_logic;

B : in std\_logic;

mode : in std\_logic;

-- Output ports

Y : out std\_logic

);

end clevergate\_behavioural;

* Open the project "Task01-02 - if statements"
* Build and simulate using the provided vector waveform file.
* In you are unsure, confirm with he tutor what the entity does (logically speaking)
  + What is the purpose of the mode input?
* If you were to build this with dataflow and structural VHDL, which of the three styles would be easiest to understand when reading the VHDL code?

## TASK 01-03 IF STATEMENT

* Now open the project in the folder task 01-03.
* What is the purpose of the 'mode' input?
* What is the purpose of the ‘EN’ input?
* Run a functional simulation using the VWF
  + What do the 'Z' outputs mean?

In this example, we saw the use of vectors being used within process blocks, but probably more significantly, the use of the 'Z' state (high-impedance) in the std\_logic type.

For this module, we will be using a FPGA device that supports tri-state logic such as this, so it is likely this code will synthesise. Note however that not all programmable logic devices do support tri-state logic.

**[NOTE –project tasks 02-01 and 02-02 are now deprecated.**

**The remaining tasks may cover theory not yet covered in the lectures.**

(This emphasizes the point that practical’s also cover theoretical and examinable material)

## Task 02-03 Latching behaviour

For this exercise, you need to be familiar with a D-Type Latch. If not, you are advised to review this before proceeding.

In the previous examples, we used the if statement and covered all possible input combinations. We say these if-statements provided **complete coverage**. So what happens with an in-complete if statement?

* open task 02-03
* examine the process block - note than EN and D are in the sensitivity list
* in the case of EN='1', what does this architecture do? \*\*
* in the case of EN='0', what does this architecture do? \*\*
* Build the VHDL and check the warnings – does it mention ‘latching behavior’? \*\*
* Now run the simulation using the vector waveform provided and confirm any 'latching' behavior

(\*\*If unsure, ask the tutor)

**An incomplete conditional statement, such as an if statement, implies a latch.**

The this example, nothing was specified for the condition where EN='0', so the output Q was left unchanged. By default, it will be latched. We often say it is “implicitly latched”

**Challenge:** (You may want to complete the lab first, then come back to this)

* Extend this example to work with a 16-bit std\_logic\_vector input
* Simulate and show this is working to the tutor
* Now use **generics** (see previous session) to allow for a variable number N of input (and output) bits

## TASK 02-04 CASE STATEMENT

For this task, you need to be familiar with the function of a **multiplexer**. If you are unsure, please review before proceeding with this task.

In previous examples, you might notice that the **if-elsif-else** statements can become rather verbose, and unnecessarily verbose in some cases. In the case where you are repeatedly testing the same variable, then a case statement might be more appropriate.

The syntax for the **case** statement is in Appendix H.

* Open the project 02-04
* Build and simulate - check you understand the output and the VHDL (ask the tutor for help if not)
* Examine the process block

You may notice the repetitive **if-elsif-else** statements which become quite hard to read.

* Using the syntax in the appendix, replace the if-statements with a **case** statement
* Re-simulate to test it is working

**Challenge:** (You may want to complete the lab first, then come back to this)

* Use the EN (active low) input such than when EN='1', the output is high-impedance 'Z'
* Test and show the tutor if you are unsure.

# 03 - Looping

So far, you have only seen process blocks being used (i) for combinational logic and (ii) that run once. One of the major benefits of the process block is with **iterative statements** (looping).

These include:

* for loop
* loop
* while loop

Let's look at examples of these.

## Task 03-01 for loop

In this next example we will look at a for-loop. This is the same example as was used in the lecture.

* Open the project 03-01
* Inspect the VHDL code
* Build and simulate
* What does this code do? (If unsure, ask the tutor)

Let's now examine the process block.

**process**(A) **is**

-- Declaration(s)

**variable** P : integer;

**variable** Q : std\_logic\_vector((DATA\_WIDTH-1)downto 0);

**begin**

-- Sequential Statement(s)

P := 0;

**for** n **in** (DATA\_WIDTH-1) **downto** 0 loop

Q(n) := A(P);

P := P + 1; -- works with integer types

**end** **loop**;

Y <= Q;

**end process**;

We now introduce a variable P of type **integer**. This was done because it is simple to perform scalar arithmetic on integers.

We also have another variable, Q. This is a temporary storage placeholder that is used to construct the final result (reversed bit pattern).

Note again, the whole process block (if synthesized), would probably result in simple combinational logic. The sequential statements simply describe what it does. They do NOT describe the structure, and there is no CPU. The whole process block can therefore be seen as a single concurrent statement.

**Task:**

* Consider what this actually does (in a logical sense).
* How would you build it yourself?
* Look at the compilation report – how many gates does it use?
  + Surprised?

### An important note on variables and signals

At the beginning of this tutorial, it was mentioned that you cannot declare local signals inside a process block. Instead you can declare local **variables**.

Inside a process block, as the statements are interpreted, variables are changed immediately - you can sequentially modify variables until the process block is complete. Conversely, **assigning values to signals has no effect until the process block completes**. Any changes make to a signal inside a loop would be ignored until the complete process is complete.

* Sequentially changing a signal, only the last value assigned will be used.

### TASK 03-02 - type conversion

VHDL is a very type-sensitive (type-safe) language. This is illustrated in the next example.

* Open the project 03-02
* Build and simulate using the vector waveform file provided
* Inspect the VHDL
* Do you understand what this does?

Note the following two lines

use ieee.std\_logic\_arith.all;

Y <= CONV\_STD\_LOGIC\_VECTOR(res, DATA\_WIDTH);

This demonstrates a conversion function CONV\_STD\_LOGIC\_VECTOR which converts an integer to a std\_logic\_vector. This is the older way of performing such a conversion. Alternatively, if you use ieee.numeric\_std (recommended for new designs), you would write the following:

Y <= std\_logic\_vector(to\_unsigned(res, DATA\_WIDTH));

For new designs, you are advised to use ieee.numeric\_std instead of std\_logic\_arith. However, you might encounter it, which is why it was included here.

**Challenge**

* Create an entity that determines the **parity** of a **std\_logic\_vector** input.
* ONLY use numeric\_std
* Simulate and show the tutor
* Hint. This should have a single std\_logic output

**Next week -** we will look at loop and while-loop structures, and begin to look at sequential logic that uses clock signals.

# 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**;

# Appendix D - TYPE and subtype DEFINTIONS

**type** *type-name* **is** (*value list*);

**subtype** *subtype-name* **is** *type-name* **range** *start* **to** *end*;

**subtype** *subtype-name* **is** *type-name* **range** *start* **downto** *end*;

**constant** *constant-name*: *type-name* := *value*;

# Appendix E - Arrays

**type** *type-name* **is** **array** (*start* **to** *end*) **of** *element-type*;

**type** *type-name* **is** **array** (*start* **downto** *end*) **of** *element-type*;

**type** type-name **is** **array** (*range-type*) **of** *element-type*;

**type** type-name **is** **array** (*range-type* **range** *start* **to** *end*) **of** *element-type*;

**type** type-name **is** **array** (*range-type* **range** *start* **downto** *end*) **of** *element-type*;

**type** *type\_name* **is array** (*type* **range <>) of** *element\_type*; -- unconstrained array

# APENDIX F - IEEE STD\_ULOGIC and STD\_LOGIC

**type** STD\_ULOGIC **is** ( 'U', -- uninitialized

'X', -- forcing unknown

'0', -- forcing 0

'1', -- forcing 1

'Z', -- High Impedance

'W', -- Weak unknown

'L', -- Weak 0

'H', -- Weak 1

'-', -- Don't care

);

**subtype** STD\_LOGIC **is resolved** STD\_ULOGIC;

-- and the vectors

**type** STD\_ULOGIC\_VECTOR **is array** (natural **range** <>) **of** STD\_ULOGIC;

**type** STD\_LOGIC\_VECTOR **is array** (natural **range** <>) **of** STD\_LOGIC;

# Appendix G - Structural statements

## component Declaration

**component** *component-name*

**port** ( *signal-names* : *mode* *signal-type*;

*signal-names* : *mode* *signal-type*;

...

*signal-names* : *mode* *signal-type )*;

**end component**;

## Instantiation

*label: component-name* **port map** (*signal1, signal2, ..., signaln*);

*label: component-name* **port map** (*port1=>signal1, port2=>signal2, ..., portn=>signaln*);

## Generate

*label*: **for** *identifier* **in** *range* **generate**

*concurrent-statement*

**end generate;**

## Generic Declarations

**generic** ( *constant-names* : *constant-type*;

*constant-names* : *constant-type*;

...

*constant-names* : *constant-type*);

# Appendix H - Behavioural Statements

## process statement

**process** (*signal-name, signal-name, ..., signal-name*)

*type declarations*

*variable declarations*

*constant declarations*

*function definitions*

*procedure definitions*

**begin**

*sequential statement*

*sequential statement*

*...*

*sequential statement*

**end process;**

## if statement

**if** *boolean-expression* **then** *sequential-statements*

end if;

## If-ELSe

**if** *boolean-expression* **then** *sequential-statements*

**else** *sequential-statements*

**end if;**

## if-elsif

**if** *boolean-expression* **then** *sequential-statements*

**elsif** *boolean-expression* **then** *sequential-statements*

**elsif** *boolean-expression* **then** *sequential-statements*

...

**elsif** *boolean-expression* **then** *sequential-statements*

**end if;**

## if-elsif-ELSE

**if** *boolean-expression* **then** *sequential-statements*

**elsif** *boolean-expression* **then** *sequential-statements*

**elsif** *boolean-expression* **then** *sequential-statements*

...

**elsif** *boolean-expression* **then** *sequential-statements*

**else** *sequential-statements*

**end if;**

## CASE

**case** expression **is**

**when** choices => sequential-statements

**when** choices => sequential-statements

...

**when** choices => sequential-statements

**end case;**

## LOOP

**loop**

*sequential-statement*

*sequential-statement*

*...*

*sequential-statement*

**end loop**;

## FOR LOOP

**for** identifier **in** range **loop**

*sequential-statement*

*sequential-statement*

...

*sequential-statement*

**end loop**;

## while loop

**while** boolean-expression **loop**

*sequential-statement*

*sequential-statement*

...

*sequential-statement*

**end loop**;