Digital Electronics and VHDL

Practical - combinational logic Part 3 - BEHAVIOURAL style VHDL

Table of Contents

[Introduction 3](#_Toc60932308)

[**IMPORTANT UPDATES – READ THIS BEFORE YOU START** 3](#_Toc60932309)

[Target FPGA 3](#_Toc60932310)

[Quartus II and the Vector Waveform Editor 3](#_Toc60932311)

[Quartus User Interface 3](#_Toc60932312)

[StaRter Code 3](#_Toc60932313)

[01 - concurrent statements and process blocks 4](#_Toc60932314)

[Process blocks 5](#_Toc60932315)

[Sensitivity List 5](#_Toc60932316)

[Local Declarations 5](#_Toc60932317)

[Sequential statements 5](#_Toc60932318)

[example 1-1 process block 6](#_Toc60932319)

[TASK 01-01 Simple combinational logic 7](#_Toc60932320)

[Advanced – automatic simulation 7](#_Toc60932321)

[02 - Conditional Statements 8](#_Toc60932322)

[Task 02-01 if statement 8](#_Toc60932323)

[TASK 01-03 IF STATEMENT 9](#_Toc60932324)

[Task01-04 – sensitivity lists 9](#_Toc60932325)

[TASK02 – Implicit Latching 10](#_Toc60932326)

[Task 02-03 Latching behaviour 10](#_Toc60932327)

[TASK 02-04 CASE STATEMENT 11](#_Toc60932328)

[03 - Looping 12](#_Toc60932329)

[Task 03-01 for loop 12](#_Toc60932330)

[An important note on variables and signals 13](#_Toc60932331)

[TASK 03-02 - type conversion 14](#_Toc60932332)

[Appendix A – entities and architectures 15](#_Toc60932333)

[Entity 15](#_Toc60932334)

[Architecture 15](#_Toc60932335)

[APPENDIX B – PREDEFINED TYPES AND OPERATORS 16](#_Toc60932336)

[VHDL PREDEFINED TypeS 16](#_Toc60932337)

[VHDL INTEGER Operators 16](#_Toc60932338)

[VHDL BINARY OPERATORS 16](#_Toc60932339)

[Appendix C - Concurrent statements 17](#_Toc60932340)

[When-Else 17](#_Toc60932341)

[SELECT 17](#_Toc60932342)

[Appendix D - TYPE and subtype DEFINTIONS 18](#_Toc60932343)

[Appendix E - Arrays 18](#_Toc60932344)

[APENDIX F - IEEE STD\_ULOGIC and STD\_LOGIC 18](#_Toc60932345)

[Appendix G - Structural statements 19](#_Toc60932346)

[component Declaration 19](#_Toc60932347)

[Instantiation 19](#_Toc60932348)

[Generate 19](#_Toc60932349)

[Generic Declarations 19](#_Toc60932350)

[Appendix H - Behavioural Statements 20](#_Toc60932351)

[process statement 20](#_Toc60932352)

[if statement 21](#_Toc60932353)

[If-ELSe 21](#_Toc60932354)

[if-elsif 21](#_Toc60932355)

[if-elsif-ELSE 21](#_Toc60932356)

[CASE 22](#_Toc60932357)

[LOOP 22](#_Toc60932358)

[FOR LOOP 22](#_Toc60932359)

[while loop 23](#_Toc60932360)

# 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**

## **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 device we are currently using a **Cyclone® IV EP4CE22F17C6**

### Quartus II and the Vector Waveform Editor

We are no longer supporting the vector waveform editor in year 2.

### Quartus User Interface

With each version that is released, the user interface can sometimes change in appearance. Since 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.

### StaRter Code

Please use the examples in the GitHub repository:

<https://github.com/UniversityOfPlymouth-Electronics/ELEC240>

# 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 several 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 **at least** **one** of the signals in the sensitivity list changes. Forgetting to add a signal to the list can have very strange effects!

### 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 sessions.

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**. Instead, you declare 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 is important to emphasize that this will be a sequential way of specifying hardware logic 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 will 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
* Watch [this video](https://plymouth.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=5e3d626d-1e36-4ab3-9b76-aca900f497fd) on gate level simulation
* Build and run a gate-level-simulation as shown in the video
* 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 within 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 of conditional statements which would otherwise be illegal outside the process block.

## Advanced – automatic simulation

Watch [this video](https://plymouth.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=ef2c92dd-ce4c-4fb6-a32c-aca900fa0417) on automation of gate-level simulation in Quartus. Apply this to your project. Note that for timing simulations, you need to use Quartus to generate the .sof files.

# 02 - Conditional Statements

There are several conditional statements you can use in behavioral VHDL, including:

if-elsif-else

case

## Task 02-01 if statement

Look at APENDIX H, specifically the syntax for the variations on the **if** statement. We are 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"
* Watch [this video](https://plymouth.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=7ed9a661-4da1-4d8a-a297-aca90100ebd8) on using the debugger in ModelSim
* Build and simulate clevergate\_behavioural.vhd (you can use Quartus or just ModelSim)
* In you are unsure, confirm with the tutor what the entity does (logically speaking)
  + What is the purpose of the mode input?

## 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 in ModelSim or Quartus
  + Use the debugger to see the effect of changing the mode signal
  + 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 (at least for output pins) such as this, so it is likely this code **will** synthesise. Note however that not all programmable logic devices do support tri-state logic. It is not recommended you use Tri-State logic for internal nodes (these will be converted to multiplexed outputs).

## Task01-04 – sensitivity lists

This project task highlights on of the easiest errors to make, and that is missing a signal from a sensitivity list.

* Open task 01-04 in Quartus or ModelSim
* Watch [the following video](https://plymouth.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=01516a44-06bc-4d74-987f-aca9010b5923) on do-scripts
* Repeat the steps
* Modify the script to perform exhaustive testing (all input combinations)

Note how the sensitivity list error is difficult to spot.

**Question** – If you put the bug back in, and rerun your script, how easy is it to spot the error? Would automated checking be a good idea?

# TASK02 – Implicit Latching

**Tasks 02-01 and 02-02 have been removed. Please go to the next section**

## 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 and compile dlatch\_behavioural.vhd in task 02-03
* Examine the process block - note that signals 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? \*\*
* Start the simulation
* Type do test.do and observe the waveform output
  + Why is the output red for the first 500ns?
  + Can you see evidence of any 'latching' behaviour? Is it easy to spot?

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

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

In this example, nothing was specified for the condition where EN='0', so the output Q was latched (left unchanged). As this condition is not covered, we say by default, it will be *implicitly* latched. Sometimes you see warnings (especially in Quartus) about this, but it is a language feature.

**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
* 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 mux\_if.vhd - 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'

# 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 us 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 for\_behavioural.vhd in the project 03-01
* Inspect the VHDL code
* Build and simulate – use the single-stepping to help you understand it
* 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**. Type integer is useful as 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 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 does (in a logical sense).
* How would you build it yourself?
* Build the Quartus project, and look at the compilation report – how many gates (logic elements) does it use?
  + Are you 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, note the following:

* Variables are changed with immediate effect - 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.
* If you change a signal more than once, only the last value assigned will be used.

Also note that no logic gates were used in this component – it is purely wiring (signals). The variables never feature in the actual synthesised design!

* + [Watch this video](https://plymouth.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=a0d49177-78d2-43a0-a012-aca901178ddc) for a reflective account of this lab

### TASK 03-02 - type conversion

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

* Open bit\_counter.vhd in the project 03-02
* Simulate using ModelSim
* Run the test.do script and examine the output
* Do you understand what this does?
* Add some more tests to confirm
* Inspect the VHDL

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 to confirm it works
* Hint. This should have a single std\_logic output

**Next -** 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**;