Digital Electronics and VHDL

Practical 2 - combinational logic Part 2 - types, arrays and STRUCTURAL style VHDL

Contents

[Introduction 1](#_Toc530575813)

[**IMPORTANT UPDATES – READ THIS BEFORE YOU START** 1](#_Toc530575814)

[Target FPGA 1](#_Toc530575815)

[Quartus II and the Vector Waveform Editor 1](#_Toc530575816)

[Quartus User Interface 1](#_Toc530575817)

[01 - SUB-TYPES and the IEEE Standard Packages 1](#_Toc530575818)

[Types and subtypes 2](#_Toc530575819)

[Arrays 2](#_Toc530575820)

[TASK 01-01 4](#_Toc530575821)

[ShortCuts 5](#_Toc530575822)

[Unconstrained arrays 5](#_Toc530575823)

[2D Arrays 6](#_Toc530575824)

[IEEE Standard LOGIC TYPES 7](#_Toc530575825)

[02 - Structural Design Elements 8](#_Toc530575826)

[Task 02-01 8](#_Toc530575827)

[**TASKS**: 11](#_Toc530575828)

[Task 02-02 12](#_Toc530575829)

[Task 02-03 Using custom components 13](#_Toc530575830)

[Generate statements 14](#_Toc530575831)

[TASK 02-04 14](#_Toc530575832)

[Generics 16](#_Toc530575833)

[TASK 02-05 16](#_Toc530575834)

[Appendix A – entities and architectures 18](#_Toc530575835)

[Entity 18](#_Toc530575836)

[Architecture 19](#_Toc530575837)

[APPENDIX B – PREDEFINED TYPES AND OPERATORS 20](#_Toc530575838)

[VHDL PREDEFINED TypeS 20](#_Toc530575839)

[VHDL INTEGER Operators 20](#_Toc530575840)

[VHDL BINARY OPERATORS 21](#_Toc530575841)

[Appendix C - Concurrent statements 22](#_Toc530575842)

[When-Else 22](#_Toc530575843)

[SELECT 22](#_Toc530575844)

[Appendix D - TYPE and subtype DEFINTIONS 23](#_Toc530575845)

[Appendix E - Arrays 23](#_Toc530575846)

[APENDIX F - IEEE STD\_ULOGIC and STD\_LOGIC 23](#_Toc530575847)

[Appendix G - Structural statements 24](#_Toc530575848)

[component Declaration 24](#_Toc530575849)

[Instantiation 24](#_Toc530575850)

[Generate 24](#_Toc530575851)

[Generic Declarations 24](#_Toc530575852)

# Introduction

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

* **user defined types and signal resolution** - creating and using user-defined data types
* **IEEE standard logic packages** - sub-types which allow more detailed error detection and tri-state logic
* **generics** - a way to allow designers a simple way to change a constant at compile time
* **component declarations** - allowing other VHDL / Library functions to be used in your VHDL architecture
* **the generate statement** - used to replicate components
* **the portmap statement** - used to physically connect components together in a way that is analogous to schematic design

## **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 - SUB-TYPES and the IEEE Standard Packages

Until now, we have only used two data types:

bit

bit\_vector

In practice, these data types are seldom used. These data types present several limitations, including the problem that they are only 2-state. **Tri-state logic**(1,0,HiZ) cannot be implemented using bit or bit\_vector.

Connecting the outputs of two gates together is sometimes an error condition, and sometimes (in the case of Tri-State Logic) it is by design.

We want the compiler / simulator to help us detect faults as early as possible. We may want to discriminate between a legal condition where all outputs except one are in a high-impedance state and the illegal condition where more than one device is trying to assert a common line (an error condition)[[1]](#footnote-2). This cannot be done with bit and bit\_vector types. When these situations arise in simulation, they need to somehow be **resolved**. This is done (in part) using standardised, user defined types.

## Types and subtypes

If we wish, we can define our own data types (see Appendix D for the syntax):

|  |
| --- |
| Type definition |
| **type** traffic\_light\_state **is** (red, amber, green, flashing\_amber);  you can also create subtypes of pre-existing data types as well  **subtype** crossing **is** traffic\_light\_state **range** red **to** green;  **subtype** age **is** integer **range** 0 **to** 120;  The VHDL language allows you to use a subtype as a direct replacement (you would be allowed to sum an integer with age for example). |

Without going into too much detail at this stage, you cannot use these data types with a pre-defined operator such as **and**. The expression Y <= red **and** green; would result in a compiler error as the compiler does not know how to logically and red and green (and neither do I!).

However, it is possible to ‘overload’ the **and** operator (or any other operator) for a given datatype and define its behavior. It is also possible to both define and resolve what happens when two outputs with different values are connected (such as '1' and '0').

This is rather beyond the scope of this course, but it’s important to be aware of it as this is what the widely used IEEE **std\_ulogic** and **std\_logic** data types do (see below).

## Arrays

Referring to Appendix E for the syntax, like many programming languages, it is possible to define arrays (bit\_vector is a type of array).

|  |
| --- |
| Array type definitions |
| -- An array of 31 integers  **type** day\_in\_month **is** **array** (1 **to** 31) **of** integer;  -- An array of bits, using the traffic\_light\_state range as an index  **type** single\_output\_for\_state **is array** ( traffic\_light\_state ) **of** bit;  **type** output\_pattern **is array** ( 4 **downto** 1 ) **of** bit**;**  -- An array of arrays  **type** output\_patterns **is array** (10 **downto** 1) **of** output\_pattern**;** |

To access each element of an array, use the round parenthesis.

**type** byte **is** array (7 **downto** 0) **of** bit;

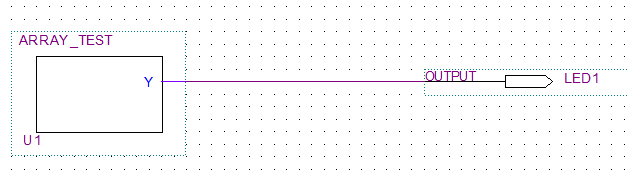
**signal** X : byte**;**

-- Array literal

X <= (‘1’, ‘0’, ‘0’, ‘1’, ‘1’, ‘0’, ‘0’, ‘0’);

Note that the array contents can be defined using an ‘**array literal**’. If you now refer to byte(7), it will return the most significant bit ‘1’ (type bit).

## TASK 01-01

You will need Quartus II to complete this task. 

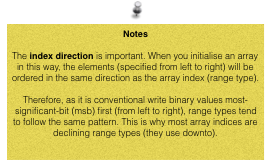
* Watch the Video [“Using Quartus with ModelSim”](https://plymouth.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=7e9f0b9e-bc68-4ae1-bb2e-ac8300b352a6)
* Open the project Task02-01. Inspect the schematics and look inside the component U1.
* Compile and program and launch ModelSim. Simulate array\_test and run for 10ps. What is the value of Y?
* Now change the line

**type** byte **is** array (7 **downto** 0) **of** bit;

to

**type** byte **is** array (0 **to** 7) **of** bit;

* Repeat -what value does Y have now? **Explain.**

****

### ShortCuts

There is another notation that is seen frequently.

**type** byte **is** array (7 **downto** 0) **of** bit;

**signal** X : byte**;**

-- Previously X <= (‘1’, ‘0’, ‘0’, ‘1’, ‘1’, ‘0’, ‘0’, ‘0’);

-- Shorthand (specify values by index)

X <= (7>=’1’, 3>=’1’, 4>=’1’, **others**=>’0’);

With this style, you specifically index individual bit positions. Note that ‘**others’** represents all the bit positions not explicitly indexed. Another way you've already seen used for bits is the use of double quotes:

**type** byte **is** **array** (7 **downto** 0) **of** bit;

**signal** P : byte;

P <= “11111100";

### Unconstrained arrays

Sometimes, you don't want to pre-define in the type-definition how many elements an array type will have. To leave it open, use the following syntax:

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

An example that you have already seen is the pre-defined type bit\_vector.

**type** bit\_vector **is** **array** (**Natural** **range** **<>**) **of** Bit;

Natural is a subtype of integer, covering of all positive integers including zero[[2]](#footnote-3). This type definition essentially says bit\_vector is an array of elements of type bit with (as-yet) an unspecified range (that are natural numbers). You specify the range later when you declare a signal or variable of the array type. So, in this case, an example might be:

**signal** : bit\_vector(7 **downto** 0);

where the range is made up of natural numbers.

### 2D Arrays

Arrays are not limited to 1D. It is not uncommon to create a 2D matrix as a look up table.

**type** bitmatrix **is array** (2 **downto** 0, 2 **downto** 0) **of** bit;

**constant** X1 : bitmatrix := ( ( '1', '0', '0') ,

( '0', '1', '0') ,

( '0', '0', '1') );

**signal** Y : bit;

Y <= X1(1,0);

**Question:**

What **type** is X1?

What **type** is X1(2,0) ?

## IEEE Standard LOGIC TYPES

All this time, we've been using type bit and bit\_vector. However, you might be surprised to learn these are seldom used. These types are limited to two states, '0' and '1'. You might be wondering about tri-state logic (which many FPGA devices can synthesise). VHDL goes much further than tri-state logic. Using the IEEE 1164 standard package, two other standard, custom-defined logic types are available:

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

'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;

**Notes**

Signals and variables of these types have the following properties:

* Except for '0','1' and 'Z', all other values are typically used for simulation only and cannot always be synthesized.
* ‘L’ and ‘H’ may be supported for open-drain outputs.
* Unlike STD\_ULOGIC, STD\_LOGIC resolves the condition where tri-state logic is used to create a bus.

Therefore, most designs tend to use STD\_LOGIC and STD\_LOGIC\_VECTOR instead of BIT and BIT\_VECTOR. To use these, you must include the right headers

**library** ieee;

**use** ieee.std\_logic\_1164.all;

**use** ieee.numeric\_std.all;

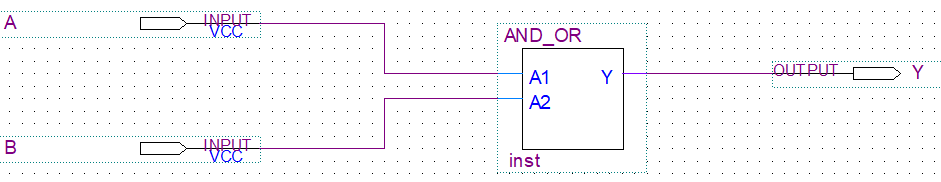
# 02 - Structural Design Elements

Up until now, we have only looked at the "data-flow" style of VHDL. Dataflow is what one might call an "abstracted" form of VHDL, in that it is abstracted from the actual hardware synthesis. The build tools have the job of converting your VHDL into a physical structure. Something a lot closer to the actual hardware synthesis is the "structural design" style. As you will see, this rather resembles a set of placement and wiring instructions for digital logic.

We begin with an example to illustrate.

## Task 02-01

* Open the project within the folder Task 02-01-Structural With Primitives



* Double click the AND\_OR entity to open the VHDL inside.
* Check that the entity declaration matches the component schematic for ANY\_OR
* Now examine the architecture block before the begin statement

**architecture** myLogic **of** AND\_OR **is**

-- Component declarations

**component** ANDGATE2

(A1 **port**, A2: **in** std\_logic; Y: **out** std\_logic);

**end** **component**;

**component** ORGATE2

port (A1, A2: **in** std\_logic; Y: **out** std\_logic);

**end** **component**;

**component** NOTGATE

**port** (A1: **in** std\_logic; Y: **out** std\_logic);

**end** **component**;

-- Signals (nodes in the circuit)

**signal** I1, I2, S1, S2 : std\_logic;

* Open the files andgate2.vhdl, orgate2.vhdl and notgate.vhdl to see their contents
  + Note the input and output labels
  + How does this relate to the component declarations above[[3]](#footnote-4)?

So far, we have declared what components we are going to use. We also declare all the internal signals. Next, we wire them up!

U1: NOTGATE **port** **map** (A1, I1);

U2: NOTGATE **port** **map** (A2, I2);

U3: ANDGATE2 **port** **map** (A1, I2, S1);

U4: ANDGATE2 **port** **map** (I1, A2, S2);

U5: ORGATE2 **port** **map** (S1, S2, Y);

Consider the first line:

U1: NOTGATE **port** **map** (A1, I1);

This can be interpreted as:

“Place a NOTGATE component with a label U1 (I chose the same notation used in regular schematics)

Connect signal A1 to the input and I1 to the output of the NOT gate.”

This is somewhat analogous to placing and wiring components. This is closer to the hardware than the Boolean style we looked at last week.

## **TASKS**:

* Draw the complete schematic and show the tutor.
* What type of circuit it is?
* Run ModelSim and test all input combinations - do the results agree?
* Create a second component block to perform the same logic function, only this time using the dataflow style[[4]](#footnote-5). This must be written as a separate entity in a different VHDL file.
* Again, use ModelSim to check that the results are the same

## Task 02-02

Using the structural VHDL, build and test a circuit with the following truth table. Use std\_logic and instead of bit as input and output data types (This was the truth table used last week in task 01-02)

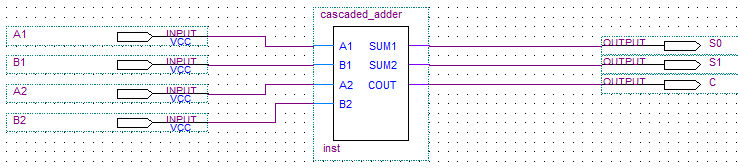
|  |  |  |  |
| --- | --- | --- | --- |
| **A** | **B** | **C** | **Y** |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

Use ModelSim to simulate to test your solution.

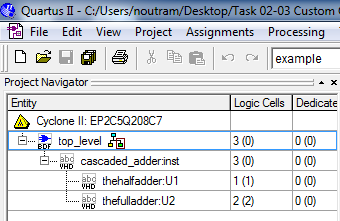
**Tip.** Create your test input signals using clock signals to save you time. [Watch this video if you are unsure how to do this](https://plymouth.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=e5c80b56-0eac-48c7-8234-ac8300fbb1a3).

## Task 02-03 Using custom components

Open the project "Task 02-03 Custom Components" and build. The top-level schematic for this is shown below. This is a two-bit adder S = A + B, where A and B are two bit binary numbers. C (the output carry) and S make up the sum.



* Double click the cascaded\_adder entity to see the VHDL. You will see two component declarations in the architecture block - a full adder and a half-adder.
* Draw and label a schematic of the circuit
* Write a truth table for the top-level circuit (run a simulation in ModelSim to verify your results)  
  *It is important you full understand what this circuit does before proceeding.*
* The full and half-adder are also written in VHDL. Expand the Project Navigator and you will see the relationship between entities.



* Inspect "thehalfadder" and "thefulladder". What style are they written with?
* **Challenge**: Create two additional components to implement both " thehalfadder" and "thefulladder" using **structural** VHDL
* Build, test and compare - are the results identical?

## Generate statements

One of the impressive aspects of VHDL you first encounter is in its ability to replicate components. A way to do this is with the **generate** statement and special type of **for-loop**, known as a **for-generate loop**:

*loop\_label:* **for** *identifier* **in** *range* **generate**

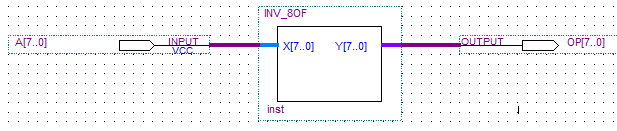
*concurrent-statement*

**end generate**

Rather than try and explain this, let's look at a simple example where we build a block of inverters to invert an 8-bit byte of data.

## TASK 02-04

* Open the project 02-04-generate.   
    
  Here is the top-level schematic.



* Double click the entity to see its VHDL.

This entity simply takes an 8-bit input X, inverts all the bits, and writes this to the output Y.

**library** ieee;

**use** ieee.std\_logic\_1164.all;

**use** ieee.numeric\_std.all;

**Entity** INV\_8OF **is**

**port**( X: **in** std\_logic\_vector(7 **downto** 0);

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

**End entity**;

* Which data type(s) is/are used for the input and output?

Note the **library** and **use** statements. These are needed to use **std\_logic\_vector** data type. It is suggested that ALL your VHDL in this module includes these lines.

Now let us look at the architecture.

**architecture** myInvArray **of** INV\_8OF **is**

-- Component declarations

**component** notgate

**port** (A1: **in** std\_logic; Y: **out** std\_logic);

**end component**;

Here we see a single component declaration (we are only using inverters in this circuit). Again, notice these primitives use **std\_logic** and not **bit** for inputs and outputs.

Now the *"business end"..... the architecture*

**begin** **generate**

g1: **for** b **in** 7 **downto** 0 **generate**

u1: notgate **port** **map** (X(b), Y(b));

**end**;

**end** **architecture** myInvArray;

We have one generate block (labeled g1:).

There is a for-loop which generates 8 NOT gates - one for each value of *b* (the *identifier*).

The bth input X(b) is connected to the bth NOT gate input

The bth output Y(b) is connected to the bth NOT gate output

* What data type is X(b) ?
* What data type is Y(b) ?

*hint - refer back to the discussion about arrays*

* Now run the simulation to confirm the VHDL works.
* Modify the simulation and VHDL to perform a 16-bit inversion

## Generics

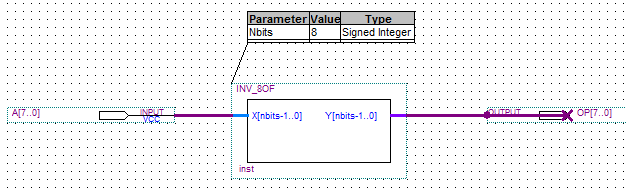
The last step of task 02-04 might have raised a question. What if I want an 8-bit and a 16-bit inverter block in a design? Do I have to create two entities?

As you might expect, the answer is no. There is a very useful way minimise replication, and that is to use **generics**.

When VHDL is compiled, all constants need to be supplied. However, you can delay specifying some constants until you instantiate it. Again, an example will help illustrate.

## TASK 02-05

* Open the project Task 02-05-generics and view the top-level schematic



Notice the top-level schematic has a parameter Nbits which can be specified before the circuit is compiled.

You could easily place another of these components on the schematic and choose a different value of Nbits.

* Double click the entity to see the VHDL.
* Examine the entity block

**Entity** INV\_8OF **is**

***generic*** *( Nbits : positive := 8 );*

**port**( X: **in** std\_logic\_vector(Nbits-1 **downto** 0);

Y: **out** std\_logic\_vector(Nbits-1 **downto** 0) );

**End** **entity**;

We have a new statement - **generic**. It has a label Nbits, a type positive and a default value of 8. This is in-effect, a constant which can be used anywhere in the entity or architecture.

* Examine the architecture block

**architecture** myInvArray **of** INV\_8OF **is**

-- Component declarations

**component** notgate

**port** (A1: **in** std\_logic; Y: **out** std\_logic);

**end** **component**;

**begin**

-- A bunch of parallel inverters

g1: **for** b **in** Nbits-1 **downto** 0 generate

u1: notgate **port** **map** (X(b), Y(b));

**end** **generate**;

**end architecture** myInvArray;

* Note. All that has changed is constant '7' has been replaced with 'Nbits'.
* Now add an additional inverter block to invert a 16-bit integer
* Test in ModelSim

#### ChALLENGE

If you have time, maybe design a generic ripple adder. Instantiate a half-adder and use a for-loop and the generate statement to instantiate a series of adders + connections.

Simulate your design.

# 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*);

1. This is an example of where VHDL as a simulator has diagnosed problems that would otherwise be missed using real hardware. [↑](#footnote-ref-2)
2. Mathematicians define natural numbers to start at 1 [↑](#footnote-ref-3)
3. The NOT gate is somewhat different in Quartus as NOT is also a reserved word. Try typing the word **not** into Quartus and you will see it come up in blue. This is why the primitive component is called \NOT\ [↑](#footnote-ref-4)
4. Unless told otherwise, the last architecture block in a VHDL file will be used. [↑](#footnote-ref-5)