Digital Electronics and VHDL

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

# 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 simply 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

# 01 - SUB-TYPES and the IEEE Standard Packages

Until now, we have only used two data types:

bit

bit\_vector

In practise, these data types are seldom used. These data types present a number of 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) through the use of 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 behaviour. It is also possible to both define and resolve what happens when two outputs with different values are connected together (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;

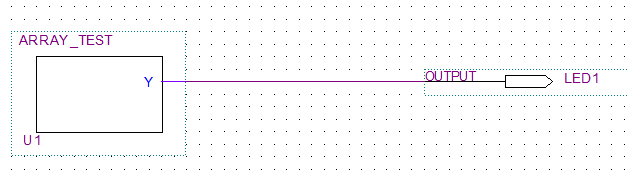
**signal** X : byte**;**

-- 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 a DE-0 NANO BOARD to complete this task. Please check one out from the stores and connect it using the USB cable provided.

* Open the project Task01-01. Inspect the schematics and look inside the component U1.
* Compile and program the target board. The LED should light (see video for this task)
* Now change the line

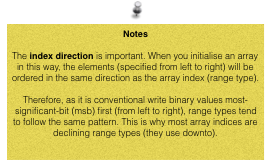
**type** byte **is** array (7 **downto** 0) **of** bit;

to

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

* **What value does Y have now?**
* **Explain?**

[**CLICK HERE FOR THE VIDEO**](https://plymouth.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=1c6ec698-fc6c-414d-8028-a99e010a80fb)

**(Video -** Task 01-01 Programming the FPGA Board****)

### 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've 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 synthesised.
* ‘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;

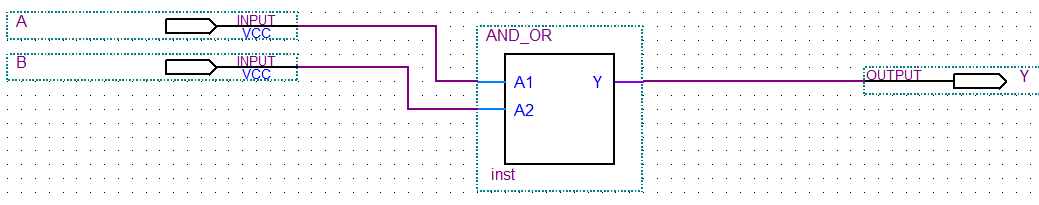
# 02 - Structural Design Elements

Up until now, we have really 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 schematic
* Now examine the architecture block before the begin statement

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

-- Component declarations

**component**

\NOT\ **port** (IN1: **in** std\_logic; \OUT\: **out** std\_logic);

**end component**;

**component AND2**

**port** (IN1, IN2: **in** std\_logic; \OUT\ : **out** std\_logic);

**end component**;

**component** OR2

**port** (IN1, IN2: **in** std\_logic; \OUT\: **out** std\_logic);

**end component**;

-- Signals (nodes in the circuit)

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

* Go to the online help and search for the term "[Primitives](http://quartushelp.altera.com/14.0/mergedProjects/hdl/prim/prim_list.htm)"

scroll down the list until you find the word Primitives again

* Examine the NOT, AND and OR primitive gates (which include AND2 and OR2).

What are the inputs and outputs labelled as?

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!

**begin**

U1: \NOT\ **port** **map** (A1, I1);

U2: \NOT\ **port** **map** (A2, I2);

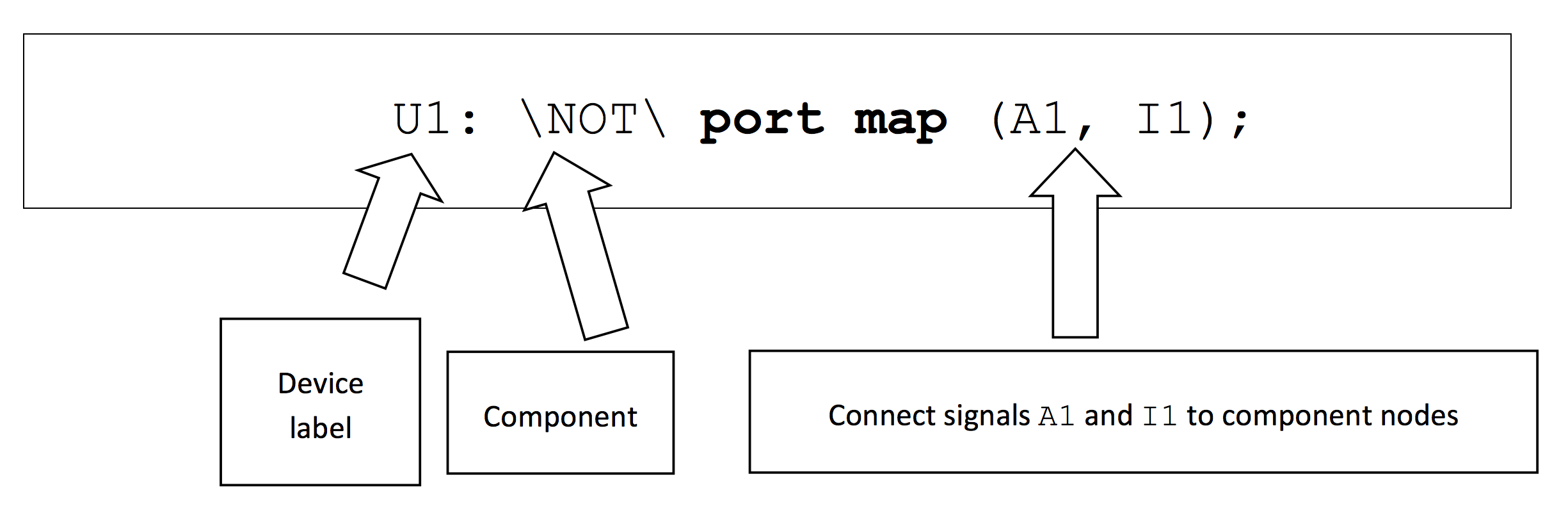
U3: AND2 **port** **map** (A1, I2, S1);

U4: AND2 **port** **map** (I1, A2, S2);

U5: OR2 **port** **map** (S1, S2, Y);

**end** **architecture** myLogic;

Consider the first line:



This can be interpreted as:

“Place a NOT 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 the vector waveform simulation - do the results agree?
* Deploy to the FPGA - using the push switches, do the results agree?
* Create a second architecture block to perform the same logic function, only this time using the dataflow style[[4]](#footnote-5). This must be written under the first (by default, the last one compiled is the one that is used for schematic capture)
* Again, check that the results are the same

Note - When you instantiate components in VHDL, you can specify which architecture to use. I’ve yet to find a way to do this with schematic capture (does not mean I am correct of course).

## 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 |

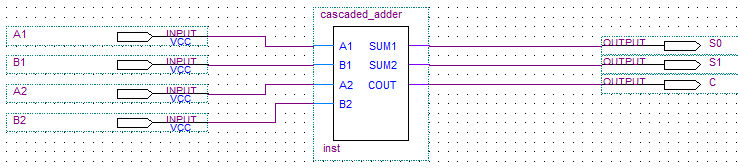
Create a vector waveform file and simulate to test your solution.

**Tip.**

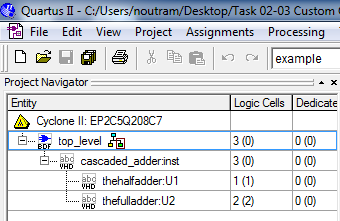
Create your stimulus signals using clock signals to save you time.

## Task 02-03 Using custom components

Open the project "Task 02-03 Custom Components". 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 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?
* Now create additional architecture blocks in each and implement both " thehalfadder" and "thefulladder" using **structural** VHDL
* Build and test - 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**

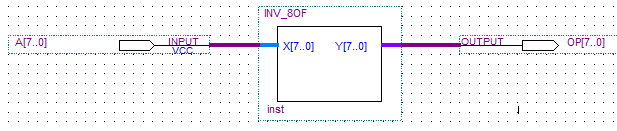
*concurrent-statement*

**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 in order to use std\_logic\_vector data type. It is suggested that ALL your VHDL in this module includes these lines.

Now let's look at the architecture.

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

-- Component declarations

**component** \NOT\

**port** (IN1: **in** std\_logic; \OUT\: **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**

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

u1: \NOT\ **port** **map** (X(b), Y(b));

**end** **generate**;

**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

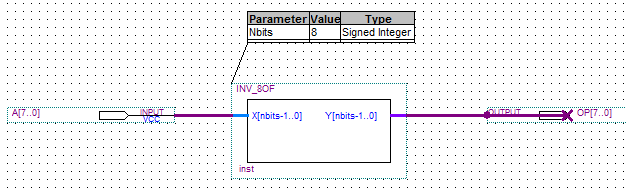
The last step of task 02-04 might have raised a question. What if I want an 8-bit and a 16-bit inverter block in a design. Do I have to create two entities?

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 actually specifying some constants until you actually 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** \NOT\

port (IN1: in std\_logic; \OUT\: out std\_logic);

end **component**;

**begin**

-- A bunch of parallel inverters

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

u1: \NOT\ 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
* Modify the vector waveform file and test this

#### 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)