Digital Electronics and VHDL

Practical 7 - SEQUENTIAL MACHINES

Table of Contents

[Introduction 3](#_Toc278529055)

[01 - SHIFT REGISTERS 3](#_Toc278529056)

[Parallel to Serial Conversion 4](#_Toc278529057)

[Serial to Parallel shift register 6](#_Toc278529058)

[02 - Finite State Machines 8](#_Toc278529059)

[TASK 02-01 Tracking an Input Sequence 8](#_Toc278529060)

[TASK02-02 - Generating a Sequence 9](#_Toc278529061)

[Appendix A – entities and architectures 10](#_Toc278529062)

[Entity 10](#_Toc278529063)

[Architecture 10](#_Toc278529064)

[APPENDIX B – PREDEFINED TYPES AND OPERATORS 11](#_Toc278529065)

[VHDL PREDEFINED TypeS 11](#_Toc278529066)

[VHDL Comparison operators 11](#_Toc278529067)

[VHDL INTEGER Operators 11](#_Toc278529068)

[VHDL Shifting Functions 11](#_Toc278529069)

[VHDL BINARY OPERATORS 12](#_Toc278529070)

[Appendix C - Concurrent statements 13](#_Toc278529071)

[When-Else 13](#_Toc278529072)

[SELECT 13](#_Toc278529073)

[Appendix D - TYPE and subtype DEFINTIONS 14](#_Toc278529074)

[Appendix E - Arrays 15](#_Toc278529075)

[APENDIX F - IEEE STD\_ULOGIC and STD\_LOGIC 16](#_Toc278529076)

[Appendix G - Structural statements 17](#_Toc278529077)

[component Declaration 17](#_Toc278529078)

[Instantiation 17](#_Toc278529079)

[Generate 17](#_Toc278529080)

[Generic Declarations 17](#_Toc278529081)

[Appendix H - Behavioural Statements 18](#_Toc278529082)

[process statement 18](#_Toc278529083)

[if statement 19](#_Toc278529084)

[If-ELSe 19](#_Toc278529085)

[if-elsif 19](#_Toc278529086)

[if-elsif-ELSE 19](#_Toc278529087)

[CASE 20](#_Toc278529088)

[LOOP 20](#_Toc278529089)

[FOR LOOP 20](#_Toc278529090)

[while loop 21](#_Toc278529091)

[NEXT and EXIT 21](#_Toc278529092)

[Appendix I - STANDARD ATTRIBUTES 22](#_Toc278529093)

[Type Attributes 22](#_Toc278529094)

[Array Attributes 22](#_Toc278529095)

[SIGNAL ATTRIBUTES 23](#_Toc278529096)

## **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. Given we only use a small subset of the buttons, it was not deemed necessary on this occasion to re-record the tutorial videos.

# Introduction

We will now begin to look at some sequential applications in VHDL. In particular, we will look at how we begin to build finite state machines (for digital control) and serial interfaces. Topics will include:

* Shift Registers
* Finite State Machines
* Multiple process blocks

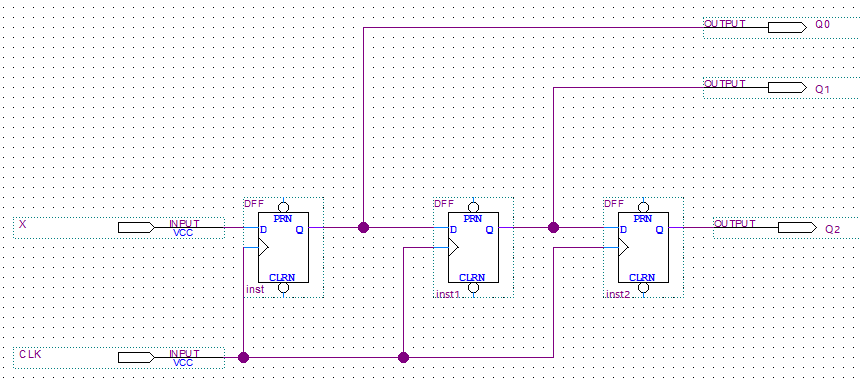
# 01 - SHIFT REGISTERS

The four basic categories of shift register are shown below (control signals not shown). There a number of different variants and applications of shift registers, and one important example is serial interfacing. Before we look at serial interfaces, let's first look at two important categories of shift register which make up the basis of many serial interface devices: parallel-to-serial and serial-to-parallel.

|  |  |  |
| --- | --- | --- |
|  | Serial In | Parallel In |
| Serial  Out |  |  |
| Parallel  Out |  |  |

## Fundamentals

Central to shift registers is the idea of cascading flip-flop devices. Figure 1 shows three D-type Flip-Flops (DFF) cascaded in sequence. The input signal is presented on X and the outputs are Q0, Q1 and Q2



Note how each DFF shares the same clock. This means **their outputs all change at the same time**. We say they are **synchronous**. The functional timing diagram for this circuit is shown in Figure 2. Here we see the input signal X cascading through each stage of the circuit.

X is the input of the first DFF. We see it’s output Q0 update on the rising edge of the clock. Q0 acts as the input for the second DFF and Q1 as the input for the third.

Figure . Cascaded D-Type Flip-Flops. Now how each device shares the same clock signal

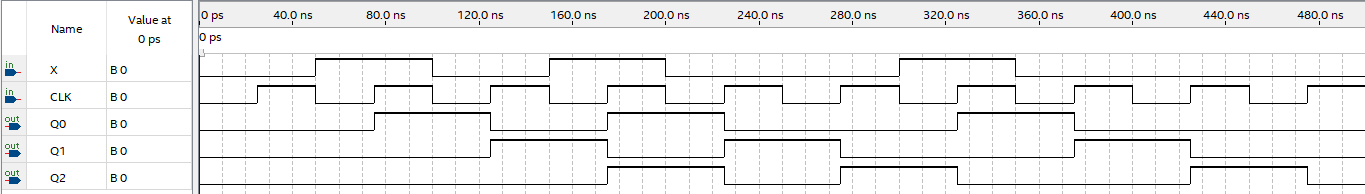


Figure . Functional simulation of the cascaded flip-flops

Consider the second DFF, where Q0 is the input. Consider the timing of this signal with respect to the rising edge of the clock. Does this seem concerning to you?

Remember that the input of a DFF must be stable Tsetup before the clock and Thold after the clock edge. It would appear that Q0 is changing at the same time as the clock CLK. Surely for the second DFF, this is a violation?

For functional simulation, there is no simulation of propagation delay, setup or hold time. So, will this work in reality? What is very important to understand is that functional simulation is misleading in this case.

To understand this, let us now look at the output for a gate-level simulation, as shown in Figure 3

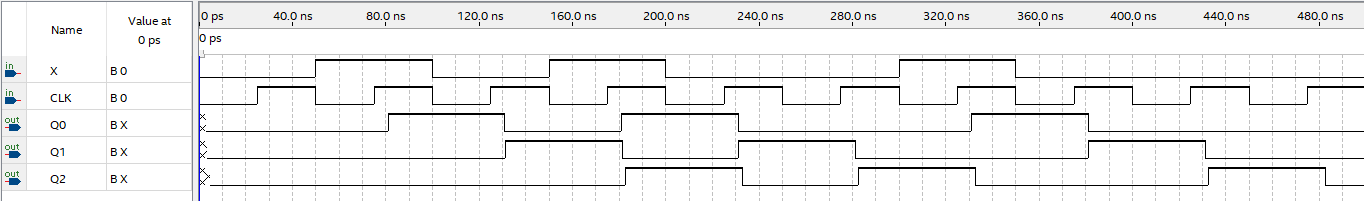


Figure . Gate-level simulation of the cascaded flip-flops. Note the propogation delay between the rising clock edge and each flip-flop output

Look carefully again at this waveform, and in particular the timing of Q0 with respect to the clock edge.

1. We note that Q0 is delayed by the propagation delay Tpd of the DFF. This means that the second DFF only observes the previous output of Q0. Therefore each stage is a delayed version of the previous. It also means the input signals are stable before the clock edge, so not violating the setup time.
2. Secondly, we note that Q0 changes shortly after the clock edge. For the second DFF, is this a violation of hold time? As long as Tpd >Thold then this does not violate the hold time. The good news is that the DFF is engineered to have zero hold-time (at the expense of set-up time), so this condition is always met.

It turns out that the propagation delay Tpd is essential to the correction operation of a shift register! With this in mind, let us now look at some behavioral models of shift registers.

## Parallel to Serial Conversion

You may be familiar with the circuit below. This is a 4-bit **synchronous** parallel-to-serial converter.



Figure . Parallel to serial shift register. Pull the signal LOW to pre-load each DFF with a known value on the next clock edge (D3 will automatically appear on the output), then pull it HIGH to shift out each bit in turn.

|  |
| --- |
| Synchronous parallel to serial converter |
| Why is this considered to be **synchronous** |
|  |
| What is the function of the input? |
|  |
| If we wanted to scale this circuit up to 32 bits, are there any problems with scalability? |
|  |

Let's now look at how we might build this in VHDL.

**entity** uop\_shiftreg\_p2s **is**

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

**port**

(

--Inputs

CLK : **in** std\_logic;

DATA\_IN : **in** std\_logic\_vector( (N-1) **downto** 0 );

LOAD : **in** std\_logic;

RESET : **in** std\_logic;

--Outputs

DATA\_OUT : **out** std\_logic

);

**end** **entity**;

**architecture** p2s **of** uop\_shiftreg\_p2s **is**

**begin**

**process** (CLK) **is**

**variable** idx : integer := 0;

**variable** Nreg : std\_logic\_vector( (N-1) **downto** 0 );

**begin**

**if** (RESET='0') **then**

idx := 0;

Nreg := (others => '0');

**elsif** (CLK'event and CLK = '1') **then**

**if** (LOAD = '0') **then**

Nreg := DATA\_IN;

idx := 0;

DATA\_OUT <= Nreg(idx) **after 2 ps**;

**elsif** (idx < (N-1)) **then**

idx := idx + 1;

DATA\_OUT <= Nreg(idx) **after** 2 ps;

**end if;**

**end if;**

**end process;**

**end p2s;**

|  |
| --- |
| Synchronous parallel to serial converter |
| Open Task 01-01 |
| Build and simulate using the testbench uop\_shiftreg\_p2s.vht (remember to compile it first!) |
| Read through the VHDL (and the comments) |
| **Question:**  There is an implicit latch in this architecture. Can you identify where it is?  What is causing the latching behavior?  Is the data sent most or least significant bit first?  When the parallel data is loaded into the internal register, how many clock cycles does it take for the first bit appear on the output?  Note the timing of the LOAD input in relation to the clock edge. Why is the LOAD signal timed the way it is? |

You might have noticed the following line:

DATA\_OUT <= Nreg(idx) **after** 2 ps;

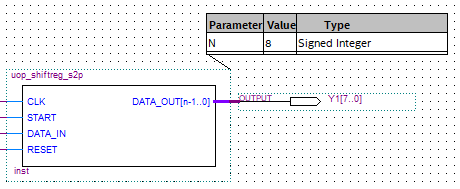
This is to give the simulation has *some* propagation delay in the output. This helps to model a real system more realistically. If we were to perform “gate level simulation”, these delays would be inserted in multiple places by Quartus.

Creating a shift register such as this is not overly complex, but a shift register alone is not generally sufficient when designing a serial interface. Generating the correct timing signals requires more thought and careful control!

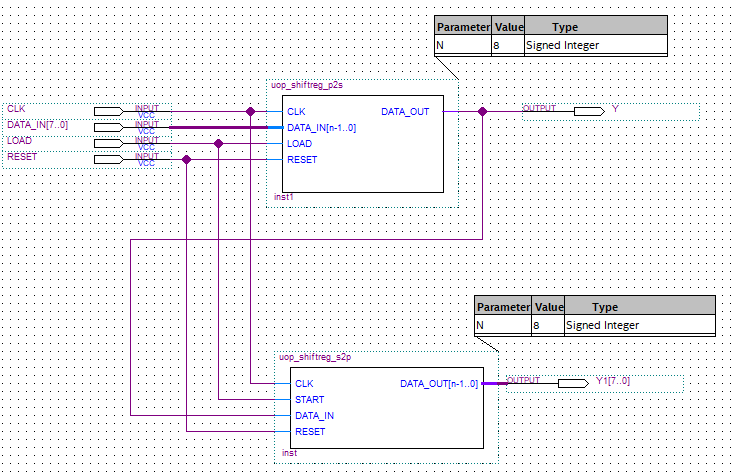
Consider designing another circuit (e.g., serial to parallel) that receives the serial bits and reconstructs them into a (latched) parallel word. Whatever circuit is receiving the serial stream, whether synchronous or asynchronous, it will need to know **when to start reading the bits and when to stop**. Some form of "**protocol**" is often required to ensure timing is perform correctly and "kept in sync". This is usually orchestrated by a state-machine. To keep things simple, we are going to use a single control line to synchronise the two devices.

## Serial to Parallel shift register

Let's now look at the converse operation, converting a series of serial bits to a parallel word. First, let's look at the entity.



For inputs, there is a clock input (CLK), a serial data in (DATA\_IN), a control line (START) and a reset (RESET). For outputs, there is a parallel N-bit wide output DATA\_OUT. Now let us look at the context in which it is used:



Note two devices use a shared clock, control line and reset. The LOAD signal and the START signal are connected[[1]](#footnote-1). There are some subtle timing issues with designs like this, but this will be discussed later. For now, let us look at the serial to parallel converter.

The VHDL is given below. Note again the use of generics to make this entity more general purpose.

**entity** uop\_shiftreg\_s2p **is**

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

**port**

(

--Inputs

CLK : **in** std\_logic;

START : **in** std\_logic;

DATA\_IN : **in** std\_logic;

RESET : **in** std\_logic;

--Outputs

DATA\_OUT : **out** std\_logic\_vector( (N-1) **downto** 0 )

);

**end entity;**

The architecture for this entity is given below:

**architecture** s2p **of** uop\_shiftreg\_s2p **is**

**type** state\_type **is** (WAITING, SHIFTING);

-- Register to hold the current state

**signal** state : state\_type;

**begin**

**process** (CLK,RESET) **is**

**variable** Nreg : std\_logic\_vector( (N-1) **downto** 0 );

**variable** idx : integer := 0;

**begin**

**if** (RESET = '0') **then**

Nreg := (**others**=>'0');

state <= WAITING;

DATA\_OUT <= (**others**=>'0');

**else**

**if** rising\_edge(CLK) then

**case** state **is**

**when** WAITING=>

**if** START = '0' **then**

idx := 0;

state <= SHIFTING;

**end if;**

**when** SHIFTING=>

Nreg(idx) := DATA\_IN;

idx := idx + 1;

**if** (idx = N) **then**

state <= WAITING;

DATA\_OUT <=Nreg;

**end if;**

**end case;**

**end if;**

**end if;**

**end process;**

**end s2p;**

This is the receiver which converts serial data to parallel. Remember – the transmitter (parallel to serial) is writing its data on the RISING EDGE of the clock. Interestingly, we are also reading the serial data on the rising edge of the clock, which sounds like a scenario that could invalidate set-up or hold time. Keep this is mind!

|  |
| --- |
| Synchronous serial to parallel converter |
| **Question -** Is this device output triggered on the rising or falling edge of the clock? |
| Open task 01-02 |
| Build the project then launch ModelSim. Now simulate using the testbench uop\_shiftreg\_s2p.vht (remember to compile it first) |
| Examine the waveform output and the testbench code. Check the output agrees with expectation |
| Next, we can see if the whole system is working. The top-level schematic has already been converted to VHDL for you (top\_level.vhd). Compile top top\_level.vhd and then compile the test bench top\_level.vht |
| Simulate top\_level.vht to see the parallel data concert to serial and back to parallel again. You should see the input data appearing in the output, albeit delayed. This demonstrates a successful transmission of data over a serial wire. |
| Read through the test bench to see how it works. Key to this is structural VHDL. |
| **Questions**  The parallel to serial converter writes the serial data *on the rising edge of the clock*.   * Once the (active low) START signal goes LOW, how many clock cycles does the serial-to-parallel device wait until reading the input and why is this necessary? * If the output device is writing serial data on the rising edge of the clock, and the input is also reading on the same rising clock edge, how is it that we not invalidating the setup and hold times?  We are assuming the internal logic to have a finite setup time and 0ps hold time.   *hint:* this works for the same reason you can cascade D-Type flipflops. *This is not an easy question to answer but is an important concept.* Also remember that we purposely modeled some propagation delay on the output of the transmitter (parallel to serial). |

**Comments and Answers:**

Sequential logic is much more complex than combinational. One of the biggest challenges is avoiding timing errors, and these can be hard to spot. In the previous example, a few issues were highlighted.

Pure functional simulations can be misleading. In real circuits, propagation delay is unavoidable, and in some cases, is a requirement for circuits to work as expected (as explained at the start of this lab). A good example of this is with shift registers.

Consider what happens **on each rising edge of the clock**:

* For the transmitter:
  + This is when the serial data is written to the serial line.
  + There is always some propagation delay, so the signal changes Tpd **after** the clock edge.
* For the receiver:
  + Data is read on the clock edge.
  + If the signal on the input is stable Tsetup before the clock edge, there is no metastability.
  + If the signal on the input is stable Thold after the clock edge, there is no metastability.
    - For this to work, we therefore require Thold < Tpd
    - Remember that for D-Type flip-flops, Thold is zero so this is not an issue.
  + In effect, *the receiver reads the (stable) output from the previous clock cycle*.

**The behavioral VHDL style rather masks this as we do not see the flip-flops.**

# 02 - Finite State Machines

One of the most common general sequential designs is a finite state machine. In this example, we are going to consider a "Moore Machine" - one where the outputs are purely a function of the **state**.

Recall that the **state** is simply a number which records where we are in some sequence. For each state, on every clock edge, we define the outputs and the next state.



In VHDL, a state machine can easily be implemented. An example is given in task 02-01

## TASK 02-01 Tracking an Input Sequence

In this task, you are going to use a finite state machine to detect a pre-defined sequence of input bits. Finite state machines fit well with VHDL and are almost self-documenting.

|  |
| --- |
| Finite State Machine - Detecting an input sequence |
| Open Task 02-01 |
| Examine the top level schematic and the entity "moore\_state\_machine.vhdl" |
| Can you draw the state diagram for this VHDL code? |
| Build the project. Then run ModelSim, compile and run the testbench moore\_state\_machine\_vhd\_tst.vht. |
| Does the output agree with what you expect? |

This is such a common pattern that there is a template in Quartus to help you get started.

Did you notice that there were **two process blocks** in this design? One sets the next state and the other sets the outputs. Each process block is a piece of parallel hardware (within each process block, we describe the expected behavior sequentially of course).

Discuss with the tutor if you do not understand this.

## TASK02-02 - Generating a Sequence (or….now it is your turn!)

Sometimes you can use a state machine to generate one or more sequences with each clock edge.

|  |
| --- |
| Finite State Machine - Sequence Generator |
|  |
| Copy Task 02-01 and rename to Task 02-02 |
| Create a new VHDL file |
| Right Click and click on "Insert Template" |
| Select VHDL->Full Designs->State Machines->Four-State-Moore-State-Machine |
| Click Insert then Close |
| Save the VHDL file as uop\_pattern\_gen.vhdl |
| Modify the VHDL to implement the state machine described below |
| Simulate and test using ModelSim. You will need to create your own testbench  In Quartus:   * Make the VHDL state machine (the unit under test or UUT) the top level entity * Rebuild the project * Click Processing→Start→Start Testbench Template Writer * Note the file that is created. * Start ModelSim and modify this file to test your state machine   Remember that components only appear in the work library once they are compiled. |
| (a solution is also provided) |

Figure 1 is a finite state machine design for a pattern generator. While the RESET is held low, the output Y is held LOW. When RESET=1, a 4-bit output sequence is generated. The input "Pattern" selects the sequence pattern to generate. Note that RESET is an input and it synchronous.

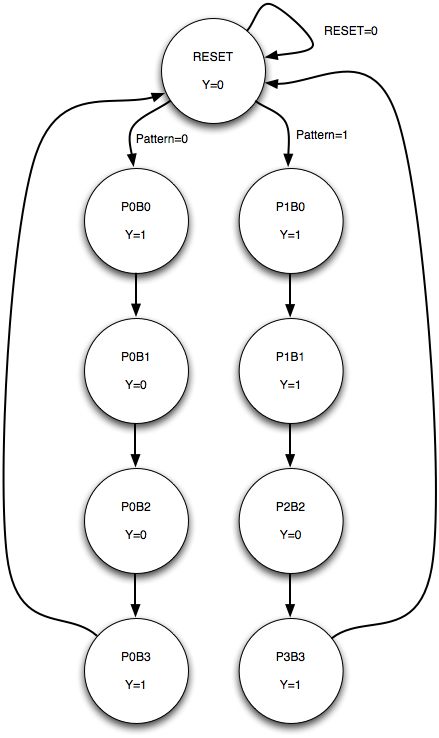


Figure . Finite State Diagram. Two synchronous inputs RESET and Pattern; One output bit Y

# 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 Comparison operators

|  |  |
| --- | --- |
| **OPERATOR** | **DESCRIPTION** |
| = | Equals |
| /= | Not equals |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |

## VHDL INTEGER Operators

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

## VHDL Shifting Functions

|  |  |
| --- | --- |
| **OPERATOR** | **DESCRIPTION** |
| sll | Shift left logical |
| srl | Shift right logical |
| sla | Shift left arithmetic (preserve sign bit) |
| sra | Shift right arithmetic (preserve and copy sign bit) |
| rol | Rotate left |
| ror | Rotate right |

## 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*

*[****next*** *[label] [****when*** *condition];*

*[****exit*** *[label] [****when*** *condition];*

**end loop**;

## FOR LOOP

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

*sequential-statement*

*sequential-statement*

...

*sequential-statement*

*[****next*** *[label] [****when*** *condition];*

*[****exit*** *[label] [****when*** *condition];*

**end loop**;

## while loop

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

*sequential-statement*

*sequential-statement*

...

*sequential-statement*

*[****next*** *[label] [****when*** *condition];*

*[****exit*** *[label] [****when*** *condition];*

**end loop**;

## NEXT and EXIT

The **next** statement skips to the next iteration of the loop.

The **exit** statement skips the reset of the statements in the loop body and breaks out of the loop.

# Appendix I - STANDARD ATTRIBUTES

*(from http://www.cs.umbc.edu/portal/help/VHDL/attribute.html, accessed 4/11/2010)*

**T** represents any type

**A** represents any array or constrained array type

**S** represents any signal

**E** represents a named entity

## Type Attributes

**T'BASE** *is the base type of the type T*

**T'LEFT** *is the leftmost value of type T (Largest if using downto)*

**T'RIGHT** *is the rightmost value of type T. (Smallest if using downto)*

**T'HIGH** *is the highest value of type T*

**T'LOW** *is the lowest value of type T*

**T'ASCENDING** *is Boolean true if range and T defined with 'to'*.

**T'IMAGE(X)**  *is a string representation of X that is of type T*

**T'VALUE(X)**  *is a value of type T converted from the string X*

**T'POS(X)** *is the integer position of X in the discrete type T*

**T'VAL(X)**  *is the value of discrete type T at integer position X*

**T'SUCC(X)** *is the value of discrete type T that is the successor of X*

**T'PRED(X)** *is the value of discrete type T that is the predecessor of X*

**T'LEFTOF(X)** *is the value of discrete type T that is left of X*

**T'RIGHTOF(X)** *is the value of discrete type T that is right of X*

## Array Attributes

**A'LEFT** *is the leftmost subscript of array A or constrained array type*

**A'LEFT(N)** *is the leftmost subscript of dimension N of array A*

**A'RIGHT** *is the rightmost subscript of array A or constrained array type*

**A'RIGHT(N)** *is the rightmost subscript of dimension N of array A*

**A'HIGH** *is the highest subscript of array A or constrained array type*

**A'HIGH(N)** *is the highest subscript of dimension N of array A*

**A'LOW**  *is the lowest subscript of array A or constrained array type*

**A'LOW(N)** *is the lowest subscript of dimension N of array A*

**A'RANGE** *is the range A'LEFT to A'RIGHT or A'LEFT downto A'RIGHT*

**A'RANGE(N)** *is the range of dimension N of A*

**A'REVERSE\_RANGE** *is the range of A with to and downto reversed*

**A'REVERSE\_RANGE(N)** *is the REVERSE\_RANGE of dimension N of array A*

**A'LENGTH** *is the integer value of the number of elements in array A*

**A'LENGTH(N)** *is the number of elements of dimension N of array A*

**A'ASCENDING** *is boolean true if range of A defined with to*

**A'ASCENDING(N)** *is boolean true if dimension N of array A defined with to*

## SIGNAL ATTRIBUTES

**S'DELAYED(t)** *is the signal value of S at time now - t*

**S'STABLE** *is true if no event is occurring on signal S*

**S'STABLE(t)** *is true if no event has occurred on signal S for t units of time*

**S'QUIET** *is true if signal S is quiet (no event this simulation cycle)*

**S'QUIET(t)** *is true if signal S has been quiet for t units of time*

**S'TRANSACTION** *is a bit signal, the inverse of previous value each cycle S is active*

**S'EVENT** *is true if signal S has had an event this simulation cycle*

**S'ACTIVE** *is true if signal S is active during current simulation cycle*

**S'LAST\_EVENT** is the time since the last event on signal S

**S'LAST\_ACTIVE** *is the time since signal S was last active*

**S'LAST\_VALUE** *is the previous value of signal S*

**S'DRIVING** *is false only if the current driver of S is a null transaction*

**S'DRIVING\_VALUE** *is the current driving value of signal S*

**E'SIMPLE\_NAME** *is a string containing the name of entity E*

**E'INSTANCE\_NAME** *is a string containing the design hierarchy including E*

**E'PATH\_NAME** *is a string containing the design hierarchy of E to design root*

1. Real world serial interfaces typically use a different mechanism to coordinate when to start reading and writing [↑](#footnote-ref-1)