Digital Electronics and VHDL

Practical 6 - COMMON SEQUENTIAL LOGICAL COMPONENTS

Table of Contents

[Introduction 3](#_Toc277318814)

[01 - The LATCH 3](#_Toc277318815)

[01-01 D-Latch 3](#_Toc277318816)

[SR-Latch 4](#_Toc277318817)

[02 - Flip-Flops 5](#_Toc277318818)

[02-01 D-Type Flip Flop 5](#_Toc277318819)

[02-02 JK Flip Flop 6](#_Toc277318820)

[02-03 Registers 6](#_Toc277318821)

[03 - Counters 8](#_Toc277318822)

[N-Bit Counters 8](#_Toc277318823)

[03-02 - CASE STUDY - HARDWARE TIMER 9](#_Toc277318824)

[Appendix A – entities and architectures 11](#_Toc277318825)

[Entity 11](#_Toc277318826)

[Architecture 11](#_Toc277318827)

[APPENDIX B – PREDEFINED TYPES AND OPERATORS 12](#_Toc277318828)

[VHDL PREDEFINED TypeS 12](#_Toc277318829)

[VHDL Comparison operators 12](#_Toc277318830)

[VHDL INTEGER Operators 12](#_Toc277318831)

[VHDL Shifting Functions 12](#_Toc277318832)

[VHDL BINARY OPERATORS 13](#_Toc277318833)

[Appendix C - Concurrent statements 14](#_Toc277318834)

[When-Else 14](#_Toc277318835)

[SELECT 14](#_Toc277318836)

[Appendix D - TYPE and subtype DEFINTIONS 15](#_Toc277318837)

[Appendix E - Arrays 16](#_Toc277318838)

[APENDIX F - IEEE STD\_ULOGIC and STD\_LOGIC 17](#_Toc277318839)

[Appendix G - Structural statements 18](#_Toc277318840)

[component Declaration 18](#_Toc277318841)

[Instantiation 18](#_Toc277318842)

[Generate 18](#_Toc277318843)

[Generic Declarations 18](#_Toc277318844)

[Appendix H - Behavioural Statements 19](#_Toc277318845)

[process statement 19](#_Toc277318846)

[if statement 20](#_Toc277318847)

[If-ELSe 20](#_Toc277318848)

[if-elsif 20](#_Toc277318849)

[if-elsif-ELSE 20](#_Toc277318850)

[CASE 21](#_Toc277318851)

[LOOP 21](#_Toc277318852)

[FOR LOOP 21](#_Toc277318853)

[while loop 22](#_Toc277318854)

[NEXT and EXIT 22](#_Toc277318855)

[Appendix I - STANDARD ATTRIBUTES 23](#_Toc277318856)

[Type Attributes 23](#_Toc277318857)

[Array Attributes 23](#_Toc277318858)

[SIGNAL ATTRIBUTES 24](#_Toc277318859)

## **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 applications on VHDL. In particular, we will look at how to build some of the most common sequential logic components, including:

* D-Latch
* SR-Latch
* D-Type Flip Flop
* J-K Flip Flop
* Counter (simple state machine)

# 01 - The LATCH

You may recall that VHDL has a language feature that allows you to 'implicitly' latch a signal. Let's look at an example to remind us of this.

## 01-01 D-Latch

**entity** uop\_dlatch **is**

**port**

(

D : **in** std\_logic;

EN : **in** std\_logic;

Q : **out** std\_logic

);

**end entity**;

**architecture** dlatch\_v1 **of** uop\_dlatch **is**

**begin**

**process** (D,EN) **is**

--Local declarations

**begin**

**if** (EN = '1') **then**

Q <= D;

**end if**;

**end process**;

**end** dlatch\_v1;

|  |
| --- |
| TASK 01-01 - D-Latch |
| * Open task 01-01 * Build and simulate * Make sure you understand the simulator output (and latching behavior) |
| * Is this device edge or level triggered? |
| * What part of the VHDL specifies that the output Q shall be latched? |
| * What is the sensitivity list in the process block? |

*Solutions over the page......*

**Solutions**

This is level triggered - there is no test for a rising for falling edge.

The if statement only covers the case where EN=1. It does not specify what to do when EN='0', therefore, the compiler assumes the output to be latched.

Both input signals D and EN are in the sensitivity list.

*If you do not understand this, or want more explanation, discuss with the tutor now.*

**Note the following!**

Look at the simulation output between 0 and 35ns. Because we are using std\_logic, we can see it's state to be uninitialized (undetermined). This could be potentially hazardous in some circumstances, so let's now fix this by adding a reset input R.

Here is the entity:

entity uop\_dlatch is

port

(

D : in std\_logic; -- D input

EN : in std\_logic; -- ENable

R : in std\_logic; -- Active LOW Reset

Q : out std\_logic -- Q output

);

end entity;

|  |
| --- |
| TASK 01-02 |
| * Copy task 01-01 to a folder called task 01-02 * Create a new variant on the D-Latch using the entity above * Now change the architecture so that the device output Q is reset to zero when ever R=0 (irrespective of D or EN) |
| * Build and simulate and show the tutor when you have achieve this (or if you are stuck) |
| * Now compare with the solution provided |

## SR-Latch

You may recall from stage-1 Digital Electronics the S-R Latch.

|  |
| --- |
| TASK 01-03 - SR Latch |
| * Sketch the timing diagram of a S-R latch and show the tutor to make sure you understand what it does. |
| * What is the illegal input condition for a SR Latch? |
| * Open 01-03, build and simulate |
| * How has the illegal condition been handled in this solution? hint - it's not a pure SR Latch |

Given that Quartus will always try to synthesize the VHDL you write, the outputs are limited to '0', '1' and 'Z'. If we were to use another tool, such as **modelsim** (simulation only), we could have set the output to 'X' (unassigned) in order to detect the error condition.

# 02 - Flip-Flops

So far in this task, we've only looked at level triggered devices (latches). The latching behaviour was achieved by conditional statements that only define outcomes for a subset of all possible input combinations.

Of equal importance are the class of device that are edge triggered, usually from the edge of a clock. These include:

* D-Type Flip Flops
* JK-Flip Flops

You may remember that flip-flop devices only respond to inputs on the **edge** of a clock signal, as opposed to a latch which responds as long as the **level** of the ENable pin is high.

Remember that a process statement is only said to execute if one of the variables in the sensitivity list changes. However, a sensitivity list can have more than one variable of course. In the case of the latch, we did not need to check which variable has changed. **For a flip-flop, we need to know if it is the clock that has changed**,and to react accordingly**.**

You may recall from the previous session that we tested whether a variable has changed by using the *event* attribute on a signal. Let's remind ourselves of how we use this attribute to build a D-Type Flip-Flop.

## 02-01 D-Type Flip Flop

In this section, we will observe the following:

* Difference between a D-Latch and a D-Flip-Flop
* How to detect a rising clock edge
* How to detect a falling clock edge
* How to implement an asynchronous reset

First, let us look at the entity for a D-Type FlipFlop.

**entity** uop\_dflipfop **is**

**port**

(

D : **in** std\_logic; -- D input

CLK : **in** std\_logic; -- CLK

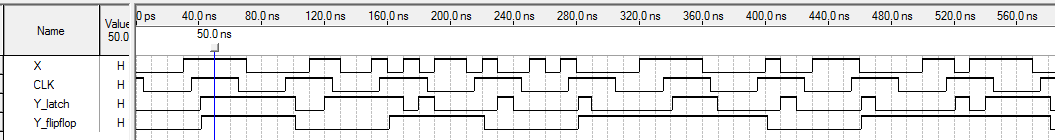
Q : **out** std\_logic -- Q output

);

**end entity**;

There is a single data input D, and a clock input CLK and a single output Q.

|  |
| --- |
| TASK 02-01-1 - D-Type Flip Flop |
| * Sketch the timing diagram of a D-Type Flip Flop |
| * Open the project 02-01, build and simulate |
| * Examine the top level schematic |
| * Examine the simulation results and contrast the Latch with the FlipFlop |
| * What is the major difference? |



The figure above shows the output from the simulation. Now how the latch output simply follows the input signal X when the clock level is HIGH. For example, look at the interval 160ns-240ns. The output of the flip-flop however is only being latched on the rising edge of the clock input.

|  |
| --- |
| TASK 02-02-2 - D-Type Flip Flop with Asynchronous Reset |
| * Copy the folder 02-01-1 to 02-01-2 |
| * Open the project and modify the flip-flop such that (i) it is triggered on a falling clock edge; and (ii) it has an asynchronous\* reset (active low). * Build, simulate and show the tutor |
| A solution is also provided |

\*By asynchronous, we mean it is not clock driven - pulling the reset line low will reset the output immediately and does not depend on the clock (useful for determining power on states).

## 02-02 JK Flip Flop

The JK flip flop has two data inputs J and K, and a clock input CLK. The behavior is described by the following table:

|  |  |  |
| --- | --- | --- |
| **J** | **K** | **Q** |
| 0 | 0 | Latch |
| 1 | 0 | SET (1) |
| 0 | 1 | RESET (0) |
| 1 | 1 | TOGGLE (0->1->0) |

When J and K are low, the output is simply latched. When J is exclusively HIGH, Q is set HIGH and when K is exclusively HIGH, then Q is set LOW. If BOTH J and K are HIGH, then Q toggles. Note that J and K are only sampled on a clock edge.

|  |
| --- |
| TASK 02-02-1 -JK Flip Flop |
| * Open the project 02-02 (task) |
| * Complete the architecture to build a JK flip-flop that is negative edge triggered * Simulate and demonstrate this is working to the tutor   *The solution is also provided in 02-02 (solution)* |
| * Now modify your schematic to demonstrate how your JK FlipFlop can be used as a clock divider |

## 02-03 Registers

The figure below is a 16-bit register, essentially a parallel set of 16-bit D-Type flip-flops. This can also be viewed as a 16-bit wide static memory[[1]](#footnote-1). Microprocessor registers (status registers, general purpose registers) can be modeled or in this way.



|  |
| --- |
| TASK 02-031 -N Register |
| * Open the project 02-03 |
| * Build the project and simulate - verify you understand the behavior of this device |
| * What is the LOAD input for? |
| * Inspect the VHDL for the uop\_dregister. * Is the LOAD input synchronous or asynchronous? * Is this a positive or negative edge (clock) triggered device? |
| * What do you notice about the relative timing between the input data, the LOAD input and the CLOCK edges? |

Discuss with the tutor when you have completed this task.

# 03 - Counters

The last category of sequential logic we are going to consider in this session is the counter device. Counters have a large number of applications, including driving state machines, simple counting / hardware timers and clock division. They are of course a special type of state machine ( a topic we will visit in the next session ).

## N-Bit Counters

In this task, you are going to look at a simple N-bit counter.

|  |
| --- |
| TASK 03-01 -N Bit Counter |
| * Open the project 03-01 |
| * Build the project and simulate - verify you understand the behavior of this device |
| * Note this time that we are using the use IEEE.std\_logic\_unsigned.all package * What **type** of variable is x? * Why was x declared to be of this type? |

Note that this time, we used a different package to the previous session. Although both styles work and both are used in books / web pages, you are recommended to use the package declared in IEEE.std\_logic\_unsigned as opposed to ieee.std\_logic\_arith

Next, we develop this counter into a useful device - a hardware timer.

## 03-02 - CASE STUDY - HARDWARE TIMER

In this section, you are set a challenge.

|  |
| --- |
| **CHALLENGE - Building a Hardware Timer (part of a Microcontroller circuit)** |
| Build, simulate and test a N-bit counter that recycles at a defined value 'PR' (which can also be set)  (the timing diagram is given over the page, and entity is given below)  A solution is provided. |

**entity** my\_counter **is**

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

**port**

(

--Clock

CLK : **in** std\_logic;

**-- Input signals**

**-- -------------**

--set COUNT to HIGH to count up; LOW to stop

COUNT : **in** std\_logic;

--set RESET to LOW to asynchronously reset

RESET : **in** std\_logic;

--Period register PR. When count = PR, recycle

PR : **in** std\_logic\_vector((N-1) **downto** 0) := ('0', **others**=>'0');

--Load the PR register - set HIGH for Period register = PR

LOAD : **in** std\_logic;

**-- Output signals**

**-- -------------**

--REC Goes momentarily high when the counter re-cycles

REC : **out** std\_logic;

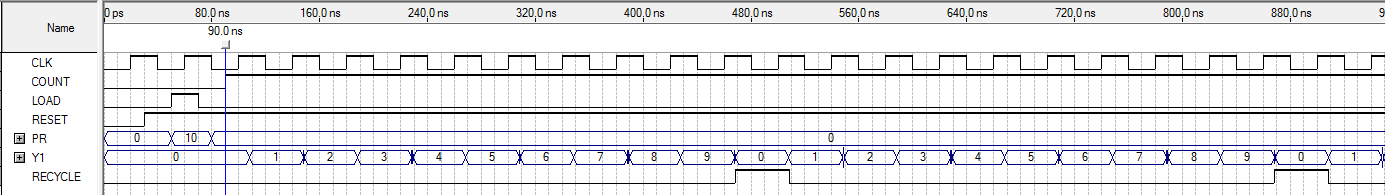
--Counter output Y

Y : **out** std\_logic\_vector((N-1) **downto** 0) := ('0', **others**=>'0')

);

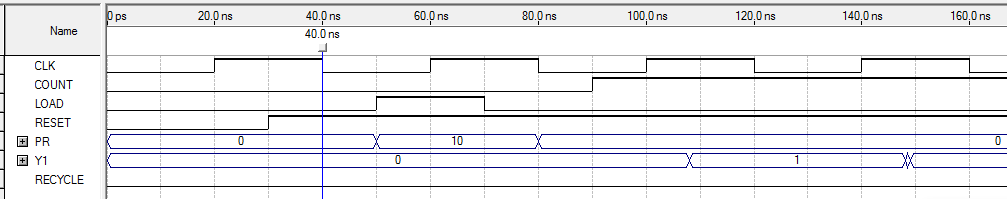
**end entity**;

Example timing diagram with the Period Register set to 10



Note the timing of the LOAD and COUNT inputs. LOAD is used to set the Period Register to 10, and COUNT is used to start the timer. The period register PR is set with the LOAD and PR inputs. The period register defines the number of states. Ever period, the output RECYCLE goes high for one clock cycle. This is useful for (i) designing clock dividers and (ii) flagging an interrupt (if part of a microcontroller design).

Taking a closer look at the LOAD timing. Note that everything is driven of the rising edge of the clock EXCEPT the reset (which is asynchronous).



# 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. If you were to scale up to many thousands / millions, you would not use behavioural VHDL to do it. It would be far to inefficient. [↑](#footnote-ref-1)