VHDL Coding Guidelines

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Author** | **Description** |
| 10/06/2022 | 0.1 | feikema1 | First draft. |
|  |  |  |  |
|  |  |  |  |

Contents

[1. Purpose 5](#_Toc124516598)

[2. Scope 5](#_Toc124516599)

[3. overview 5](#_Toc124516600)

[4. Coding Guidelines 6](#_Toc124516601)

[4.1 Naming Conventions 6](#_Toc124516602)

[4.1.1 All VHDL file names must use the .vhd extension. 6](#_Toc124516603)

[4.1.2 All file names must use lowercase letters. 6](#_Toc124516604)

[4.1.3 All module (entity) names must use lowercase letters. 6](#_Toc124516605)

[4.1.4 All ports and parameter (generic) names must use lowercase letters. 6](#_Toc124516606)

[4.1.5 Each module (entity) name must match the file name. 6](#_Toc124516607)

[4.1.6 Each module (entity) must have a unified header. 6](#_Toc124516608)

[4.1.7 Each module (entity) must have a revision history. 7](#_Toc124516609)

[4.1.8 All user-defined constants must use upper case letters. 7](#_Toc124516610)

[4.1.9 Only use legal VHDL names and avoid using VHDL keywords listed in Appendix A. 7](#_Toc124516611)

[4.1.10 Avoid using reserved words of other popular languages. 7](#_Toc124516612)

[4.1.11 All signal and variable names must use lowercase letters. 7](#_Toc124516613)

[4.1.12 Use “\_n” to indicate active low signals. 7](#_Toc124516614)

[4.1.13 Use “\_z” to indicate tri-state signals. 7](#_Toc124516615)

[4.1.14 If possible, buses should be defined using MSB-first (downto) format. 8](#_Toc124516616)

[4.1.15 If possible, use only one clock in the module. If there are multiple clocks, the names must have meaningful descriptions. 8](#_Toc124516617)

[4.1.16 Do not use mixed case for user-defined identifiers. 8](#_Toc124516618)

[4.1.17 Name objects according to function or purpose. 8](#_Toc124516619)

[4.1.18 Do not use identifiers that are overloaded or hidden by identical declarations in a different scope. 8](#_Toc124516620)

[4.1.19 Use underscore “\_” to separate signal names composed of several words. 9](#_Toc124516621)

[4.1.20 Label all processes, concurrent statements, loops, etc. 9](#_Toc124516622)

[4.2 Coding Style 10](#_Toc124516623)

[4.2.1 Use one top level design file. 10](#_Toc124516624)

[4.2.2 Assign signals types according to function. 10](#_Toc124516625)

[4.2.3 Each line should have one definition or statement. 10](#_Toc124516626)

[4.2.4 Each line should have a maximum length of 80 characters. 10](#_Toc124516627)

[4.2.5 Unused code must be removed instead of commented out. 10](#_Toc124516628)

[4.2.6 Maintain a tabular code format by applying a consistent indentation file. 10](#_Toc124516629)

[4.2.7 When implementing hierarchical designs, always use named port mapping instead of position port mapping. 11](#_Toc124516630)

[4.2.8 Put common constants, constructs, etc. in a separate package for use across multiple designs. 11](#_Toc124516631)

[4.2.9 Do not hard-code literal values. Define constants instead. 11](#_Toc124516632)

[4.2.10 Initialize bus signals with “others” to make sure no signal is left uninitialized for simulation. 11](#_Toc124516633)

[4.2.11 Do not gate clocks. 12](#_Toc124516634)

[4.2.12 Put a range on integers, etc. if known. 12](#_Toc124516635)

[4.2.13 Synchronize all resets to the clock(s) in the necessary scope. 12](#_Toc124516636)

[4.2.14 Avoid using a specific bit or bit range when referring to a vector in case the vector is re-sized 12](#_Toc124516637)

[4.2.15 Encapsulate repeatedly used operations or statements in subprograms. 12](#_Toc124516638)

[4.2.16 If possible, use a maximum of three nesting levels of control-flow statements. 12](#_Toc124516639)

[4.2.17 Assign signals from a single *always* block. 12](#_Toc124516640)

[4.2.18 Do not use bitwise operations in a boolean context. 13](#_Toc124516641)

[4.2.19 Treat cross-clock domains with extreme caution. 13](#_Toc124516642)

[4.2.20 Use an identical directory structure for every design. 15](#_Toc124516643)

[4.2.21 Include a mechanism to automatically exclude debug code from synthesis. 15](#_Toc124516644)

[4.2.22 Preface each major section with a comment describing what is does, why it exists, how it words, and/or assumptions. 15](#_Toc124516645)

[4.2.23 Use comments to describe the intent of functional of the code, not its behavior. 15](#_Toc124516646)

[4.2.24 Use relative pathnames in any scripts, etc. written to manipulate the code. 16](#_Toc124516647)

[4.3 Synthesizable RTL Design Guidelines 17](#_Toc124516648)

[4.3.1 Only use synthesizable VHDL constructs. 17](#_Toc124516649)

[4.3.2 All required signals must be listed in the sensitivity list 17](#_Toc124516650)

[4.3.3 No hard-coded delays should be used in the synthesizable RTL code. 18](#_Toc124516651)

[4.3.4 Do not omit “else”, “default”, or “when others” in “if” or “case” statements. 19](#_Toc124516652)

[4.3.5 All data types should be assigned before being read. 20](#_Toc124516653)

[4.2.25 Do not use variables unless absolutely necessary. 21](#_Toc124516654)

[4.3.6 ‘X’ should not be assigned to variables or signals. 21](#_Toc124516655)

[4.3.7 Basic guidelines for finite state machines. 21](#_Toc124516656)

[4.4 Simulation Model Design Guidelines 22](#_Toc124516657)

[4.4.1 Avoid using circular references in the library reference. 22](#_Toc124516658)

[4.4.2 All input ports should either have a default value or be connected. 22](#_Toc124516659)

[4.4.3 Bi-directional ports should only be used for true bi-directional signals. 22](#_Toc124516660)

[4.4.4 “for” loops should not be used to pass a specific amount of simulation time. 22](#_Toc124516661)

[4.4.5 Limit calculations in the necessary scope. 23](#_Toc124516662)

[4.4.6 Model clock edges precisely. 23](#_Toc124516663)

[4.5 Recommendations 24](#_Toc124516664)

[4.5.1 Always use descriptive and meaning names 24](#_Toc124516665)

[4.5.2 Avoid using single letter names except for loop iteration variables. 24](#_Toc124516666)

[4.5.3 Signal names should not be too long. 24](#_Toc124516667)

[4.5.4 Use consistent port names in all hierarchies. 24](#_Toc124516668)

[4.5.5 Use prefixes “MIN” and “MAX” to define constants of minimum and maximum values. 24](#_Toc124516669)

[4.5.6 Comment all port and generic declarations. 24](#_Toc124516670)

[4.5.7 For simulation models, signals should not be assigned ‘X’ values unless the hardware actually behaves that way. 25](#_Toc124516671)

[Appendix 1 – VHDL keywords 26](#_Toc124516672)

[Appendix 2 – VHDL constructs not supported in synthesis 32](#_Toc124516673)

Figures

[Figure 1. 100 MHz pulse is not stretched. 25 MHz pulse never triggers. 13](#_Toc116015975)

[Figure 2. 100 MHz pulse is stretched. 25 MHz pulse will now trigger. 14](#_Toc116015976)

[Figure 3. Toggle exchange protocol for data transfer 14](#_Toc116015977)

[Figure 4. Synthesized circuit 17](#_Toc116015978)

[Figure 5. Simulation for incomplete sensitivity list design 18](#_Toc116015979)

[Figure 6. Simulation for complete sensitivity list design 18](#_Toc116015980)

[Figure 7. Unnecessary latch inserted for uncovered cases 19](#_Toc116015981)

[Figure 8. Unnecessary latch removed for all covered cases 19](#_Toc116015982)

[Figure 9. Unnecessary latch inserted for uninitialized variable 20](#_Toc116015983)

[Figure 10. Unnecessary latch removed for initialized variable 21](#_Toc116015984)

# Purpose

The purpose of this document is to provide general guidelines for VHDL development in different scopes.

VHDL (VHSIC Hardware Description Language) has played a very important role in the Digital Design world. In order to deliver high quality VHDL code to both external customers and internal cross-functional teams, it is critical that certain standards are followed. Here are some benefits of adopting these standards:

* Improve overall software quality, including:
* LRM (Language Reference Manual) compliance (VHDL 2000)
* Robustness
* Readability
* Portability
* Reusability
* Avoid common mistakes by continuously updating the standard
* Promote common practice within a team
* Reduce overall maintenance effort

# Scope

The scope of this document will provide the reader with general coding guidance as well as more specific guidance regarding different aspects of VHDL development. Specific rules will be demonstrated with justifications and examples. In addition to rules that should be followed, recommendations are also made. Language references will be provided as Appendices to create a more self-contained document.

# overview

It is important to develop reliable digital systems, thus verification is very important.

The guidelines listed in this document are applicable for the following scopes:

* RTL code for hardware systems
* Verification testbenches for hardware systems

# Coding Guidelines

## Naming Conventions

Good naming conventions will make code more readable and consistent. Here are some guidelines to be followed when naming at different levels.

### All VHDL file names must use the .vhd extension.

Although some VHDL synthesis and simulation tools allow an arbitrary file extension, **.vhd** is the standard file extension for all VHDL files.

### All file names must use lowercase letters.

The main goal for this guideline is to ensure consistent representation of file names.

### All module (entity) names must use lowercase letters.

The main goal for this guideline is to ensure consistent representation of module names.

### All ports and parameter (generic) names must use lowercase letters.

The main goal for this guideline is to ensure consistent representation of port and parameter names.

### Each module (entity) name must match the file name.

The main goal for this guideline is to ensure all entities/modules for a particular project are named consistently.

### Each module (entity) must have a unified header.

Example:

---------------------------------------------------------

--

-- File: project1.vhd

-- Author: My\_name or OUN

-- Description: Top level firmware for Project1

-- Limitation: FLASH memory is limited to 4GB.

-- Copyright ©: Lawrence Livermore National Laboratory

--

---------------------------------------------------------

### Each module (entity) must have a revision history.

Every revision of a module (entity) that is checked into source control must have an entry in the revision table in the file describing the name of the person who made the changes, the date(s), a description of the changes, and why the changes were made (such as fixed bug #123).

Example:

---------------------------------------------------------

-- (unified header goes here)

---------------------------------------------------------

---------------------------------------------------------

--

-- REVISION HISTORY

--

-- Date: 8/6/2022

-- Author: My\_name or OUN

-- Description: Original

-- Purpose:

--

-- Date: 9/31/2022

-- Author: My\_name or OUN

-- Description: Changed signal ram\_address from 8 bits to 16

-- bits.

-- Purpose: RAM size on PCB increased to store more data.

--

---------------------------------------------------------

### All user-defined constants must use upper case letters.

Example:

constant ADDR\_WIDTH : integer := 12; -- width of RAM address

### Only use legal VHDL names and avoid using VHDL keywords listed in Appendix A.

### Avoid using reserved words of other popular languages.

As many designs contain mixed languages, avoid reserved words of languages such a C, C++, Verilog, SystemVerilog, PERL, TCL, etc.

### All signal and variable names must use lowercase letters.

### Use “\_n” to indicate active low signals.

This guideline enforces the visibility of active levels for signals.

### Use “\_z” to indicate tri-state signals.

This guideline enforces the visibility of tri-state signals.

### If possible, buses should be defined using MSB-first (downto) format.

This guideline enforces consistency when defining buses.

Example:

data\_bus : in std\_logic\_vector(MAX\_WIDTH-1 downto 0);

### If possible, use only one clock in the module. If there are multiple clocks, the names must have meaningful descriptions.

Include the frequency in the clock name only if there is an absolute need. This ensures that only a few files must be updated if the clock frequency changes instead of re-naming the clock signal in many levels of the hierarchy.

Example (single clock in the design):

clk : in std\_logic;

Example (mutliple clocks in the design):

clk\_ram\_rd : in std\_logic; -- clock to read RAM

clk\_ram\_wr : in std\_logic; -- clock to write RAM

Example (clock frequency if needed):

Clk\_50mhz : in std\_logic;

### Do not use mixed case for user-defined identifiers.

Some tools may be case-insensitive.

### Name objects according to function or purpose.

Avoid naming objects according to type or implementation

Bad example:

signal count8 : unsigned(7 downto 0);

Good example:

signal addr\_count : unsigned(7 downto 0);

### Do not use identifiers that are overloaded or hidden by identical declarations in a different scope.

Bad example:

signal address : unsigned(7 downto 0);

Good example:

signal decoder\_address : unsigned(7 downto 0);

### Use underscore “\_” to separate signal names composed of several words.

Bad example:

readIndexInTable <= 0;

Good example:

read\_index\_in\_table <= 0;

### Label all processes, concurrent statements, loops, etc.

This makes tracking signals in simulation much easier.

Bad example:

process (i\_fast\_clk) is

begin

...

end process;

process (i\_fast\_clk) is

begin

...

end process;

Good example:

p\_ram\_write : process (i\_fast\_clk) is

begin

...

end process p\_ram\_write;

p\_ram\_read : process (i\_fast\_clk) is

begin

...

end process p\_ram\_read;

## Coding Style

Good coding style produces consistent VHDL code that is readable, maintainable, and reusable. The following guidelines are provided for coding styles.

### Use one top level design file.

Instantiate this single file in the test bench.

### Assign signals types according to function.

Input/outputs signals are usually std\_logic or std\_logic\_vector.

Internal signals are usually signed/unsigned if possible.

### Each line should have one definition or statement.

This guideline applies to all code including port definitions, generic definitions, port mapping, generic mapping, and any executable statements.

### Each line should have a maximum length of 80 characters.

This guideline makes code more readable by avoiding inadvertent line wraps.

### Unused code must be removed instead of commented out.

It is a good practice to remove unused code from any file. The revision control system provides the capability of rolling back to a previous code base.

### Maintain a tabular code format by applying a consistent indentation file.

This guideline ensures the code is in a tabular format where all the code blocks of the same kind are aligned.

* Never use tabs – they are defined differently by different editors.
* Always indent with 3 or 4 spaces.

Bad example:

signal counta, countb : integer;

signal c : real := 0.0;

signal sum : signed(0 to 31);

signal z : unsigned(6 downto 0);

Good example:

signal counta : integer;

signal countb : integer;

signal c : real := 0.0;

signal sum : signed (0 to 31);

signal z : unsigned(6 downto 0);

### When implementing hierarchical designs, always use named port mapping instead of position port mapping.

This is a very important rule to follow. It is very common to add/remove ports during the development phase. If positional mapping is used, the risk of introducing mapping errors is very high.

Bad example:

fifo in\_buffer(voice\_sample\_retimed,

valid\_voice\_sample, overflow, ,

voice\_sample, 1’b1, clk\_8kHz,

clk\_20MHz);

Good example:

fifo in\_buffer(.data\_in => voice\_sample,

.valid\_in => 1’b1,

.clk\_in => clk\_8kHz,

.data\_out => voice\_sample\_retimed,

.valid\_out => valid\_voice\_sample,

.clk\_out => clk\_20MHz,

.full => overflow,

.empty => oepn);

### Put common constants, constructs, etc. in a separate package for use across multiple designs.

If a common constants, construct, etc. is changed, there is only the need to change one common file, not many across the orihect,

### Do not hard-code literal values. Define constants instead.

Whenever a hard-coded literal value is needed, define a constant for it. This rule ensures better readability and ease of code maintenance.

Bad example:

if (start\_addr = “FE”) then ...

Good example:

constant MEM\_EMPTY : std\_logic\_vector(7 downto 0) : x”00”;

constant MEM\_AFULL : std\_logic\_vector(7 downto 0) : x”FE”;

constant MEM\_FULL : std\_logic\_vector(7 downto 0) : x”FF”;

if (start\_addr = MEM\_AFULL) then ...

### Initialize bus signals with “others” to make sure no signal is left uninitialized for simulation.

This is especially useful when generating test benches since uninitialized signals can cause unexpected behavior during simulation.

Bad example:

my\_bus : std\_logic\_vector(7 downto 0) := x”00”;

Good example:

my\_bus : std\_logic\_vector(7 downto 0) := (others => ‘0’);

### Do not gate clocks.

Use an enable signal.

### Put a range on integers, etc. if known.

This speeds simulation time and improve synthesizer efficiency.

### Synchronize all resets to the clock(s) in the necessary scope.

Manage input reset signals carefully. Asynchronous input reset signals can cause unexpected behavior to synchronous logic.

### Avoid using a specific bit or bit range when referring to a vector in case the vector is re-sized

Bad example:

signal count : unsigned(15 downto 0);

...

count(15 downto 0) <= count(15 downto 0) + 1;

carry <= count(15);

Good example:

signal count : unsigned(15 downto 0);

...

count <= count + 1;

carry <= count(count’left);

### Encapsulate repeatedly used operations or statements in subprograms.

Place subprograms in a common package if used in multiple designs across the project.

### If possible, use a maximum of three nesting levels of control-flow statements.

### Assign signals from a single *always* block.

Bad example:

always @ (s)

begin

if (s = ‘1’) then

q <= ‘1’;

end

always @ (r)

begin

if (r = ‘1’) then

q <= ‘0’;

end

Good example:

always @ (s or r)

begin

if (s = ‘1’) then

q <= ‘1’;

elsif (r = ‘1’) then

q <= ‘0’;

end if;

end

### Do not use bitwise operations in a boolean context.

Bad example:

signal byte : std\_logic\_vector(3 downto 0);

signal valid : std\_logic;

if (byte AND valid) begin

...

end if;

Good example:

signal byte : std\_logic\_vector(3 downto 0);

signal valid : std\_logic;

if (byte! = “0000” and valid = ‘1’) begin

...

end if;

### Treat cross-clock domains with extreme caution.

Crossing clock domains is the cause of many timing errors. Commonly used ways to cross clock domains are:

* Metastability registers: used for crossing from a slower clock domain to a faster clock domain
* Pulse-stretching: used for crossing from a faster clock domain to a slower clock domain

Diagram

Description automatically generated

Figure 1. 100 MHz pulse is not stretched. 25 MHz pulse never triggers.

Diagram

Description automatically generated

Figure 2. 100 MHz pulse is stretched. 25 MHz pulse will now trigger.

* Toggle exchange protocol

Use a toggle or trigger to signal to the other block that data is ready. The signal must be clocked for metastability in the new clock domain.

Diagram

Description automatically generated

Diagram

Description automatically generated

Figure 3. Toggle exchange protocol for data transfer

* Look for edges

Example:

p\_example : process (i\_fast\_clk) is

begin

if rising\_edge(i\_fast\_clk) then

-- r1\_data is METASTABLE, r2\_data and r3\_data are STABLE

r1\_data <= i\_slow\_data;

r2\_data <= r1\_data;

r3\_data <= r2\_data;

if (r3\_data = ‘0’ and r2\_data = ‘1’) then

-- positive edge condition

end if;

end if;

end process p\_example;

* Use a FIFO for large amount of data

Always monitor the empty and full flags of the FIFO. Never:

* Read from an empty FIFO
* Write to a full FIFO

### Use an identical directory structure for every design.

This ensures consistency in the file hierarchy and all designers know where specific code is located across the project.

### Include a mechanism to automatically exclude debug code from synthesis.

Example:

generate\_label:

if (expression) generate

< conditionally-compiled or debug code here >

end generate;

### Preface each major section with a comment describing what is does, why it exists, how it words, and/or assumptions.

A recommended practice is to insert break lines or other visual between major sections for readability. It is very difficult to read a many lines to code with no comments and/or break lines.

Good example:

--------------

/\* comment for code here \*/

< code >

--------------

/\* comment for code here \*/

< code >

--------------

/\* comment for code here \*/

< code >

### Use comments to describe the intent of functional of the code, not its behavior.

Bad example:

/\* increment \*/

i <= i + 1;

Good example:

/\* move the read pointer to the next element \*/

i <= i + 1;

### Use relative pathnames in any scripts, etc. written to manipulate the code.

This makes scripts easily portable across multiple design, especially if the same design hierarchy is used for all design (refer to section 4.2.20).

Bad example:

re -f vhdl design.vhd

Good example:

read -format vhdl design.vhd

## Synthesizable RTL Design Guidelines

This section lists the guidelines that are applicable for synthesizable RTL design code. The goals of these guidelines are:

* Ensure the post-synthesis netlist does not have unnecessary logic such as latches and combinatorial loop. The unnecessary logic will occupy silicon resource and affect performance of the entire design.
* Ensure the post-synthesis netlist is functionally equivalent to the original design. The pre- and post-synthesis should simulate consistently.
* Ensure the post-synthesis circuit infers the intended logic component. Designs such as State Machines require certain coding styles to be implemented correctly.

### Only use synthesizable VHDL constructs.

The constructs listed in Appendix 2 are not (or partially not) supported in synthesis and should be avoided in RTL design. If any of these are used, synthesis tools will ignore them and incorrect results will yield.

### All required signals must be listed in the sensitivity list

Signals that are used within the process should be included in the sensitivity list. Missing signals on the sensitivity list will results in inconsistent pre- and post-synthesis simulation. The reason behind this is that synthesis always implements the logic as if the sensitivity list is complete. But pre-synthesis simulation only simulates the process according to the sensitivity list. The two implementations listed below yield the same synthesized circuit as shown in Figure 1.

Use “all” sparingly or not at all.

Bad example:

p\_example : process (ina)

begin

result <= ina AND inb;

end process p\_example;

Good example:

p\_example : process (ina, inb)

begin

result <= ina AND inb;

end process p\_example;

Chart, line chart

Description automatically generated

Figure 4. Synthesized circuit

Graphical user interface, application, Excel

Description automatically generated

Figure . Simulation for incomplete sensitivity list design

Graphical user interface, application, Excel

Description automatically generated

Figure . Simulation for complete sensitivity list design

### No hard-coded delays should be used in the synthesizable RTL code.

Hard-coded delays will cause mismatches between pre-synthesis simulation results and post-synthesis results. All delays are ignored by synthesis.

Bad example (“after” statement is ignored):

results <= ‘0’ after 10 ns;

### Do not omit “else”, “default”, or “when others” in “if” or “case” statements.

Uncovered cases will result in latches in the synthesized circuit. By covering all possible cases,

it prevents unintentional latch generation, producing a circuit that uses hardware resource more efficiently.

Bad example:

p\_example : process (ina, inb)

begin

if (ina = ‘1’) then

if (inb = ‘1’) then

result <= ‘1’;

end if;

else

result <= ‘0’;

end if;

end process p\_example;

Good example:

p\_example : process (ina, inb)

begin

if (ina = ‘1’) then

if (inb = ‘1’) then

result <= ‘1’;

**else**

**result <= ‘0’;**

end if;

else

result <= ‘0’;

end if;

end process p\_example;

Diagram, schematic

Description automatically generated

Figure 7. Unnecessary latch inserted for uncovered cases

Chart, diagram

Description automatically generated

Figure 8. Unnecessary latch removed for all covered cases

### All data types should be assigned before being read.

If any data object is read before being assigned will result in unnecessary latches. To avoid latches generated unintentionally, all data object should be initialized before being read. For the given example, the synthesized circuit is shown in Figure 6 and 7, with and without unnecessary latches respectively.

Bad example:

p\_example : process (ina, inb)

variable vint : integer range 0 to 1;

begin

if (ina = ‘1’) then

vint := vint;

else

vint := vint + 1;

end if;

case (vint) is

when 0 => result <= ina;

when others => result <= inb;

end case;

end process p\_example;

Good example:

p\_example : process (ina, inb)

variable vint : integer range 0 to 1;

begin

**vint := 0;**

if (ina = ‘1’) then

vint := vint;

else

vint := vint + 1;

end if;

case (vint) is

when 0 => result <= ina;

when others => result <= inb;

end case;

end process p\_example;

Diagram, schematic

Description automatically generated

Figure 9. Unnecessary latch inserted for uninitialized variable

Chart

Description automatically generated

Figure 10. Unnecessary latch removed for initialized variable

### Do not use variables unless absolutely necessary.

It is very easy to generate latches when using variables. They are not needed in most code.

### ‘X’ should not be assigned to variables or signals.

In VHDL, ‘X’ assignments are ignored by synthesis tools. For simulation, they are treated as unknown values. Using the ‘X’ assignment will cause inconsistent pre-synthesis simulation results and post-synthesis results.

### Basic guidelines for finite state machines.

The following guidelines are helpful in modeling finite state machines.

* Clearly define state variables according to naming convention guidelines.
* Always use “case” statements to check for current state.
* All state machine outputs should be registers.
* All state machine inputs, including resets, should be synchronous.
* Do not have dead states.

## Simulation Model Design Guidelines

This section lists the guidelines that are applicable for VHDL simulation model code. The goals of these guidelines are:

* Ensure accurate modeling of the hardware blocks.
* Ensure generic simulation models are usable for all simulators.
* Ensure efficient execution of simulation in simulators.

### Avoid using circular references in the library reference.

Bad example (requires my\_pkg to be compiled with “-work mylib” option):

library IEEE, **mylib**;

use IEEE.std\_logic\_1164.all;

use work.my\_pkg.all;

Good example:

library IEEE;

use IEEE.std\_logic\_1164.all;

use mylib.my\_pkg.all;

### All input ports should either have a default value or be connected.

This ensures the full connectivity for the netlist. The candidate default values are:

* ‘0’
* ‘1’
* ‘X’
* ‘Z’

### Bi-directional ports should only be used for true bi-directional signals.

The bi-directional port is a super set of input and output ports. Input/output only ports should not be declared as INOUT.

### “for” loops should not be used to pass a specific amount of simulation time.

“for” loops are expensive in terms of evaluations when used to control elapsed time. Such usage will significantly slow down simulation.

Bad example (requires my\_pkg to be compiled with “-work mylib” option):

for i in 1 to 50 loop

wait until clk’event and clk = ‘1’;

end loop;

Good examples:

wait for (clk\_period\*50-1 ns);

wait until clk’event and clk = ‘1’;

### Limit calculations in the necessary scope.

All calculations adds extra simulation time, thus should be limited to the necessary scope only.

Bad example (requires my\_pkg to be compiled with “-work mylib” option):

vint := to\_integer(vec\_value);

if (ina = inb) then

result <= vint;

else

result <- 0;

end if;

Good example:

if (ina = inb) then

vint := to\_integer(vec\_value);

result <= vint;

else

result <- 0;

end if;

### Model clock edges precisely.

In order to model clock edges precisely, it is important to make sure only valid clock edges can trigger activities. Unexpected behavior can occur in each of the following scenarios:

* When clock signals get corrupted and then recovered.
* At power-up stage.

Bad example (‘0’ 🡪 ‘1’ is treated as a rising clock edge; **‘X’ 🡪 ‘1’ is treated as a rising clock edge**):

if (clk’event and clk=’1’) then ...

Good example (‘0’ 🡪 ‘1’ is treated as a rising clock edge; ‘X’ 🡪 ‘1’ is not treated as a rising clock edge):

if rising\_edge(clk) then ...

## Recommendations

In addition to the guidelines listed above, the following recommendations are provided to further improve code quality and reliability.

### Always use descriptive and meaning names

Applicable name spaces are:

* Modules
* Ports
* Generics
* Signals
* Variables (if used)
* Constants

### Avoid using single letter names except for loop iteration variables.

A minimum of 5 characters is suggested.

Bad example:

if (e = ‘1’) then

c <= c + 1;

end if;

Good example:

if (enable = ‘1’) then

count16 <= count16 + 1;

end if;

### Signal names should not be too long.

Names that are extremely long in HDL will make the code hard to read and maintain. Some tools have length restrictions on names. A maximum of 25 characters is suggested.

### Use consistent port names in all hierarchies.

This ensures unique names are being used for the same signal(s) throughout the design, in all levels or the hierarchy.

### Use prefixes “MIN” and “MAX” to define constants of minimum and maximum values.

### Comment all port and generic declarations.

The comment is preferred to be on the same line as long as it does not break the length guidelines of 80 characters.

Bad example:

entity test\_port is

port (

port\_a : in std\_logic;

port\_b : in std\_logic;

port\_c : out std\_logic);

Good example:

entity test\_port is

port (

port\_a : in std\_logic; -- comment for port\_a

port\_b : in std\_logic; -- comment for port\_b

port\_c : out std\_logic); -- comment for port\_c

### For simulation models, signals should not be assigned ‘X’ values unless the hardware actually behaves that way.

# Appendix 1 – VHDL keywords

| **Keywords** | **Oper** | **Purpose** |
| --- | --- | --- |
| abs | √ | Unary operator for absolute value, predefined for any numeric type. |
| access |  | A variety of data type whose values are pointers to (or links to, or addresses of) dynamically-allocated objects of some other type. |
| after |  | Specifies delay information in a signal assignment. If there is no after clause, default delay of one simulation delta is assumed. |
| alias |  | Creates an alternate name for all or part of an existing identifier. |
| all |  | Suffix for identifying all declarations that are contained within the package or library denoted by the prefix. |
| and | √ | Logical “and” operator for types bit and Boolean, as well as for one-dimensional arrays of these types. |
| architecture |  | A secondary design unit containing the description of the design. |
| array |  | Used to define an array, vector or matrix. |
| assert |  | Statement that implements a self-checking. Often used in conjunction with reporting of error messages. |
| attribute |  | Describes a characteristic of a given object. An attribute declaration declares an attribute name and its type. An attribute specification assigns a value to the attribute. |
| begin |  | Marks the beginning of the statement portion of a process statement or architecture body. |
| block |  | Indicates the start of a block structure. |
| body |  | Designates a procedure body. |
| buffer |  | A mode of a signal that enables a port to be read and updated within the entity model. A buffer port can have only one source, and can be connected only to another buffer port or to a signal with no more than one source. |
| bus |  | A kind of signal that represents a hardware bus that can have multiple drivers. |
| case |  | Part of a “case” statement. |
| component |  | Starts the definition of a component. |
| configuration |  | A declaration used to create a configuration for an entity. It associates particular component instances with specific design entities, and associates entity declarations with specific architectures. |
| constant |  | Declares an identifier for a constant value of the specified type. |
| disconnect |  | Specifies the disconnect time for a guarded signal. |
| downto |  | Specifies descending range. |
| else |  | Optional identifier for the final alternative in an “if” or “when” statement. |
| elsif |  | Identifier for an interim alternative in an “if” statement. |
| end |  | Marks the end of a statement, subprogram, or declaration of a library unit. |
| entity |  | Specifies input and output definitions of the design. |
| exit |  | Sequential statement to causes execution to jump out of the innermost loop or the loop whose label is specified. |
| file |  | Declares a file. |
| for |  | Used to iterate a predetermined number of replications in replicated logic, such as generate and loop statements. Also used in specifying blocks, components, and configurations, and in specifying time expression in a timeout clause. |
| function |  | Defines a group of sequential statements that returns a single value. |
| generate |  | Replicates one or more concurrent statement. Can be in for or if format. |
| generic |  | Defines constants whose values may be controlled by the environment. Generics are of the object class constant. The declaration of a generic may also include a default value, which will be used if an actual value is missing in the generic map. |
| guarded |  | Option for a concurrent signal assignment. The guarded option specifies that the signal assignment statement will execute only when the guard condition of the block statement that contains the assignment is true. |
| if |  | Sequential statement that describes conditional logic. |
| impure |  | Declares a function that may return a different value given the same parameters due to side effects. |
| in |  | Port mode that allows the port to be read only. If no mode is specified, in is assumed. |
| inertial |  | An option for specifying delay information in a signal assignment statement. Inertial delay is characteristic of switching circuits: a pulse whose duration is shorter than the switching time of the circuit will not be transmitted or in the case that a pulse rejection limit is specified, a pulse whose duration is shorter than that limit will not be transmitted. |
| inout |  | Port mode that allows a bidirectional port to be read and updated within the entity model. |
| is |  | Equates the identity portion to the definition portion of a declaration. |
| label |  | Specifies a label name in an attribute statement. |
| library |  | Specifies the logical names of design libraries that can be referenced within a design unit. The following library clause is implied for every design unit:  library std, work |
| linkage |  | A port mode similar to inout used to connect VHDL ports to non-VHDL ports. |
| literal |  | Used in attribute statement as entity specification. |
| loop |  | Statement used to execute through a set of sequential statements multiple times. |
| map |  | Map actual port and parameters to the declared ones. In either case, the corresponding port declaration must include a default value. |
| mod | √ | Arithmetic operator for modulus of any integer type. The operands and the result are of the same type. The result of a mod operator has the sign of the second operand and is defined (for some integer n) as:  a mod b = a-b\*n |
| nand | √ | Logical “nand” operator for types bit and Boolean, as well as for one-dimensional arrays of these types. |
| new |  | Allocates memory and returns access pointer. |
| next |  | Statement that causes the current iteration of the specified loop to be prematurely terminated, resuming execution with the next iteration of the loop. |
| nor | √ | Logical “nor” operator for types bit and Boolean, as well as for one-dimensional arrays of these types. |
| not | √ | Unary logical operator for types bit and Boolean. |
| null |  | Sequential statement that performs no action. |
| of |  | Keyword used to link an identifier to its entity name, and used when specifying type mark in a file type definition. |
| on |  | Used as a connective in the sensitivity clause of a wait statement. |
| open |  | An entity aspect, used as a binding indication to indicate that binding is not yet specified and that it is to be deferred. |
| or | √ | Logical “or” operator for types bit and Boolean, as well as for one-dimensional arrays of these types. |
| others |  | Specifies all remaining elements in case statement, element association, attribute specification, or variable assignment for array types. |
| out |  | Port mode that enables the port to be updated only. It cannot be read. |
| package |  | Specifies a set of declarations including aliases, attributes, components, constants, files, functions, types, and subtypes. It can also include attribute specifications, disconnection specifications, and use clauses. |
| port |  | Signals through which an entity communicates with the other models in its external environment. |
| postponed |  | Option for a concurrent signal assignment or process statement. |
| procedure |  | A group of sequential statements. |
| process |  | Defines a sequential process intended to model all or part of a design entity. The sensitivity list identifies signals to which the process is sensitive. Whenever an event occurs on any item on the sensitivity list, the sequential code in the process will be executed.  In general, processes may or may not be synthesizable, depending on the details of how they are written. |
| pure |  | Option for a function in a subprogram specification. A pure function does not have side effects. |
| range |  | Defines a range constraint for a scalar type. |
| record |  | A composite data type in which the collection of values may belong to the same or different types. |
| register |  | A kind of signal which models a latch. If all drivers to such a signal are disconnected, the signal retains its old value. |
| reject |  | An option for specifying delay information in a signal assignment statement. It specifies the minimum pulse width to prorogate as a result of an “after” clause. |
| rem | √ | Arithmetic operator for remainder for any integer type; the operands and the result are of the same type. The result of a rem operator has the sign of the first operand and is defined as:  a rem b = a-(a/b)\*b |
| report |  | Statement for generating report messages. Not supported for synthesis. |
| return |  | Sequential statement used at the end of a subprogram to terminate, returning control back to the calling object. All functions must have a return statement, and the value of the expression in the return statement is returned to the calling program. |
| rol | √ | Shift operator: rotate left. Each bit in the left operand is shifted by the number of bits specified in the right operand. Bits in the left-most positions of the operand are shifted to the right-most bits of the operand. |
| ror | √ | Shift operator: rotate right. Each bit in the left operand is shifted by the number of bits specified in the right operand. Bits in the right-most positions of the operand are shifted to the left-most bits of the operand. |
| select |  | Concurrent signal assignment that selects a value for a target signal from a list of alternatives. |
| severity |  | A predefined type in the language with values note, warning, error, and failure for assertion and reporting purposes. |
| shared |  | Declares shared objects. |
| signal |  | Declares a wire or a placeholder for a value. Signals are assigned in signal assignment statements, and declared in signal declarations. Signal assignments always occur with some amount of delay. When delay\_mechanism is not specified, signal assignments will occur one delta delay after the signal assignment statement is executed. |
| sla | √ | Shift left arithmetic operator defined for any one-dimensional array type whose element type is either bit or Boolean. The arguments of sla are the array that will be shifted and the amount by which it will be shifted. This shift operator will fill with the leftmost bit. |
| sll | √ | Shift left logical operator defined for any one-dimensional array type whose element type is either bit or Boolean. The arguments of sll are the array that will be shifted and the amount by which it will be shifted. This shift operator will fill with zeros. |
| sra | √ | Shift right arithmetic operator defined for any one-dimensional array type whose element type is either bit or Boolean. The arguments of sra are the array that will be shifted and the amount by which it will be shifted. This shift operator will fill with the rightmost bit. |
| srl | √ | Shift right logical operator defined for any one-dimensional array type whose element type is either bit or Boolean. The arguments of srl are the array that will be shifted and the amount by which it will be shifted. This shift operator will fill with zeros. |
| subtype |  | Declares a subtype, which is a type with a constraint. The constraint is based on an existing parent type. |
| then |  | Part of syntax of “if” statement. |
| to |  | Specifies direction in an ascending range. |
| transport |  | An option for specifying delay mechanism in a signal assignment statement. Transport delay is characteristic of hardware devices, such as transmission lines, that exhibit nearly infinite frequency response: any pulse is transmitted, no matter how short its duration. Not supported for synthesis. |
| type |  | Declares a type. A type declaration includes the keyword “type”, the type identifier, the keyword “is”, and the type definition. Each data type has a set of values and a set of operations associated with it. A type definition can be: scalar, composite, access, and file. In addition, there are non-predefined types established by the IEEE standard 1164. Each of these types is listed below.   * Scalar Types * Enumerated * Character * Bit Boolean * Severity\_level * Numeric * Integer * Physical * Floating\_point * Composite Types * Array * String * Bit\_vector * Record * Access Types * File Types * Non-predefined Types: IEEE standard 1164 has the following types added for modeling digital logic: * std\_ulogic (an enumerated type with the values ’U’, ’X’, ’0’, ’1’, ’Z’, ’W’, ’L’, ’H’, ’-’) * std\_logic (same as std\_ulogic except that this is a resolved type) * std\_ulogic\_vector (an array of std\_ulogic) * std\_logic\_vector (an array of std\_logic) * Predefined-Types: Types defined by the IEEE standard 1076.3 are: * Unsigned (an array of std\_logic) * Signed (an array of std\_logic) * Overloaded arithmetic and conversion operators for types unsigned and signed are defined in the package numeric\_std.   IEEE VHDL Language Reference Manual (LRM) has a Design Considerations section for more information. |
| unaffected |  | Indicates in a conditional or selected signal assignment when the signal is not to be given a new value. |
| units |  | An entity class, to be stated during attributes specification of user-defined attributes. Also declares physical types. |
| until |  | Part of the condition clause of a wait statement. |
| use |  | Makes a package available to this design unit. |
| variable |  | Declares an object as a variable. Variables are created at the time of elaboration and retain their values throughout the entire simulation run. Variable assignments occur without delay (unlike signal assignments). This has major implications when variable assignments are used as part of a block of sequential statements within a process. |
| wait |  | Temporarily suspends a process until:   * wait on sensitivity\_list; -- an event occurs that affects one or more signals on the list * wait until boolean\_expression; – a specific condition is met * wait for time\_value; -- a specific time has passed   These forms can be combined:  wait on sensitivity\_list until boolean\_expression for time\_value; |
| when |  | Specifies a condition during which an “exit” or “next” statement will be executed. |
| while |  | Specifies a condition during which a loop will be executed. |
| with |  | Used in the syntax of a selected signal assignment. |
| xnor | √ | Logical “exclusive nor” operator for types bit and Boolean and for one-dimensional arrays of these types. |
| xor | √ | Logical “exclusive or” operator for types bit and Boolean and for one-dimensional arrays of these types. |

# Appendix 2 – VHDL constructs not supported in synthesis

|  |  |
| --- | --- |
| **VHDL Construct** | **Notes** |
| array | Multidimensional arrays are not supported by some synthesis tools. |
| assertion | Assertion statements are ignored. |
| attribute | * “all” is supported * “others” is supported * User-defined attributes are not supported |
| configuration | Configuration specification is not supported. |
| Disconnection | Disconnection specification is not supported. |
| entity | * Generics are supported * Port default values are ignored |
| file | File type is not supported/ |
| floating | Floating-point type declarations are ignored. |
| function calls | Function conversions on input ports are not supported. |
| null literals | Null slices, null ranges, and null arrays are not supported. |
| now | “now” function is not supported. |
| physical | Physical type declarations are ignored. |
| severity\_level | “severity\_level” is not supported. |
| TEXTIO | TEXTIO package is not supported. |
| time | “time” type is not supported. |