### COLLABORATORS

<table>
<thead>
<tr>
<th>ACTION</th>
<th>NAME</th>
<th>DATE</th>
<th>SIGNATURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITTEN BY</td>
<td>Tristan Gingold</td>
<td>2018-01-15</td>
<td></td>
</tr>
</tbody>
</table>
# Contents

1 Introduction 1

2 Style Checker 1

3 Exemptions 1

4 Rules Description 1

5 Rules 2

5.1 File rules 2

5.1.1 [FileName] [M] Name of VHDL file 2

5.1.2 [FileContent] [R] Content of a VHDL file 2

5.1.3 [FileHeader] [M] Header comment of a VHDL file 2

5.1.4 [LineLength] [M] Source line length 3

5.1.5 [EndOfLine] [M] End of line 3

5.1.6 [Language] [M] Language for comments and identifiers (I) 3

5.1.7 [CharSet] [M] Character set 3

5.1.8 [NoTAB] [M] No tabulation 3

5.1.9 [LastLine] [M] Last line in a file 3

5.1.10 [TrailingSpaces] [M] Trailing spaces 3

5.2 Format rules 4

5.2.1 [Comments] [M] Comment style 4

5.2.2 [Indentation] [M] Indentation 4

5.2.3 [WhiteSpaces] [M] Spaces 4

5.2.4 [Context] [M] Context clauses 5

5.2.5 [UseClause] [M] Place of use clause 5

5.2.6 [EntityLayout] [M] Layout of entity declaration 5

5.2.7 [ComplexStmtLayout] [M] Layout of complex statements 6

5.2.8 [BeginEndLayout] [M] Layout of begin/end keywords 7

5.2.9 [EndLabel] [M] Presence of the label after end 7

5.2.10 [Instantiation] [M] Layout of instantiation 8

5.2.11 [ProcessLabel] [M] Label of processes 8

5.2.12 [Parenthesis] [M] Use of parenthesis in expressions 8

5.3 Identifiers 8

5.3.1 [Keywords] [M] Keywords case 8

5.3.2 [Identifiers] [M] Identifiers case (I) 8

5.3.3 [Underscores] [M] Use of underscore in identifiers (I) 8

5.3.4 [ReferenceName] [M] Reference 9
1 Introduction

Coding style (also named programming style) is a set of rules used when writing source code. These rules specify aspects of free-form languages (like indentation, use of white-spaces...) and restrict the language. This document describes a VHDL coding style. It was written to be universal and is currently being used for many ohwr.org projects.

This coding style is written assuming some general ideas:

- Team spirit is important and coding style is one way to promote it. So code should be as uniform as possible, so that anyone feels at home everywhere and can improve the code easily. Of course, some designers know some parts much better than other members of the team, but this could change by time and this shouldn’t prevent anyone to improve code anywhere.

- A coding style would result in better code quality through uniformity. Your eye will be able to easily recognize patterns when they have the same structure. A coding style makes the code easier to read.

- The coding style document is the place to write rules for portability over all the tools used. Work-around for tool bugs should be described (as rules) in this document.

- This coding style has to stay alive. A rule can be written with a very good intent, but practice may tell later that it isn’t.

- Describing and checking coding style is tedious, so the rules should be checked by a tool. Rules should be written in such a way they could be mechanically checked; rules that cannot must be the exception. It is advised for users to read this document once; but they will learn it by applying the tool on their code and by fixing the diagnostics.

2 Style Checker

We strongly believe that a tool to detect coding style violation is as important as the coding style document. Sometimes this is also called a linter.

It is indeed boring to manually check coding style rule, and very easy to miss violations. As many rules are about taste, people tend to accept diagnostics from a tool more easily than from a colleague.

Some rules (like using meaningful names) cannot be enforced by a tool. The number of such rules should be as limited as possible.

3 Exemptions

It must be possible to have exemption from a rule at any place in a file. The designer knows the context better than a tool, and in some cases a rule is counter-productive. The amount of exemptions must be as limited as possible and if an exemption is used many times this might be a hint to change a rule.

Exemptions should be justified by a comment. A possible (to be implemented) way to declare an exemption for the automatic tool is through a pragma:

```--pragma codestyle No-Identifier comment```

4 Rules Description

A rule is defined by a sub-section in this document. The title of the sub-section has a fixed format:

```
[Identifier] [S] Short description
```

The title starts with a unique identifier, so that a tool can refer to a rule. A number could have been used instead, but they may be subject to renumbering.

The status [S] describes how important the rule is:
• [M] for mandatory. It is highly unusual to not follow the rule.
• [R] for recommended. It is not unusual to have an exemption.

The short description can be followed by (I) when it is normally not possible to automatically check the rule and need manual inspection.

The rule should have a reason to explain its origin.

The rules apply only to human written files. That is automatically generated code may not follow the rules because usually you cannot control the style of generated code.

5 Rules

5.1 File rules

5.1.1 [FileName] [M] Name of VHDL file

The name of the file is the name of the first design unit (using exactly the same casing) followed by the .vhd extension.

Reason: It makes search of a unit easier. The extension is the current practice (from the DOS 3 letters for extension area). It should be noted that vhd is often confused with the extension for virtual hard disk.

5.1.2 [FileContent] [R] Content of a VHDL file

A VHDL file may contains either:

• an entity and its architecture. In that case, the entity must not have another architecture.
• a configuration (unusual).
• a package declaration (if there is no body)
• a package declaration and its body.

Reason: Simplify the search of a unit. It should be noted that synthesis tools do not support well multiple architectures and configurations.

5.1.3 [FileHeader] [M] Header comment of a VHDL file

Each source file should start with a header comment to describe the content of the file and the license. The template of this comment is:

```
-- COMPANY
-- Name of the project
-- URL of the project
------------------------------------------------------------------------------
-- unit name:    project_top
--
-- description:
--
-- This unit implements the interface of the project.
--
------------------------------------------------------------------------------
-- Copyright (c) YEARS COMPANY
------------------------------------------------------------------------------
```
Reason: Trade-off. Name of the author does not appear because it does not bring anything, because it is not clear when a contributor becomes an author, and because the versioning control system tracks the authors of the changes.

5.1.4 [LineLength] [M] Source line length

The recommended line length is 100 characters (without the end of line), and the maximum is 132 characters.

Reason: People often have a fixed setup of editor windows.

5.1.5 [EndOfLine] [M] End of line

The end of line is the single LF character (aka \n) to follow the Unix convention.

Reason: Most users develop on Linux.

5.1.6 [Language] [M] Language for comments and identifiers (I)

Always use English for comments and identifiers.

Reason: Maximalize usability

5.1.7 [CharSet] [M] Character set

Restrict to plain 7-bit ASCII codeset. The only control character allowed is the end of line marker (LF). If you need to write a non-English word, use its transliteration and remove accents.

Reason: ASCII is universal, other code sets need an encoding.

5.1.8 [NoTAB] [M] No tabulation

As a consequence of the previous rule, horizontal tabulations (HT) are not allowed.

Reason: The rendering of HT is not completely fixed.

5.1.9 [LastLine] [M] Last line in a file

The last line must finish with an end of line, and must not be empty. So blank lines at the end are not allowed.

Reason: Blank lines at the end are useless and removed by many editors. Git complains if the file ends with two end of line characters.

5.1.10 [TrailingSpaces] [M] Trailing spaces

The last character (if any) before the end of line must not be a space. So trailing spaces are not allowed.

Reason: Trailing spaces may not be visible, are often removed by editors.
5.2 Format rules

5.2.1 [Comments] [M] Comment style

It is clearer to have dedicated lines for comments; but it is possible to have a very short comment at the end of a line (after VHDL code) to give a hint.

```vhdl
-- This comment is for the variable
variable v : natural

v_next := v; -- Keep current value
```

Comments start with `--`. For inline comments, there must be at least one space before the double dash.

There must be a space after the `--`, except for a line comment (a line that consists of only `-` or `=`), in the case the comment must start on the first column.

```vhdl
-- This is a box comment
--
-- constant c : natural := 25; -- This is an inline comment
```

Reason: Allow inline comments and boxes.

5.2.2 [Indentation] [M] Indentation

The indentation is two spaces. Indentation is used:

- for declarative item
- for nested concurrent or sequential statements
- for the second and later lines of a multi-line declaration or statement.

Reason: Indentation makes code structure clearer; 2 spaces is an historical choice.

5.2.3 [WhiteSpaces] [M] Spaces

There is at least one white space:

- before and after `:` in declarations
- before and after `:=` and `<=` in assignments (or for default values)
- before and after comparaison operators
- before and after `⇒` in named associations

The usual number of white spaces is one, but for alignment purpose there can be more than one white space.

There is no white space:

- before `,` or `;`
- before `{` when used for conversion, function call, index or slice name.
- between `process` and `{`
Examples:

```vhdl
constant c_address : t_address := x"00000C00";
g_simulation /= 0
  rst_aux_n_o <= rst_net_n;
  clk_i  => clk_ref_i,
  std_logic_vector(to_unsigned(i+1, 4))
```

Reason: Try to follow common punctuation rules.

### 5.2.4 [Context] [M] Context clauses

Context clauses are organised by groups of a library clause followed by use clauses and a blank line. The library clause must be omitted for `std` and `work` (as they are implicit).

There must be only one library name per library clause, and one selected name per use clause. A use clause must only be used to make a whole package visible, and therefore be written as the name of the library, followed by the name of the package followed by `all`. A use clause must follow the library clause for the related library, the only exception is for `std.textio` which must be the last use clause of the `ieee` group (if present).

The first group is the one for the `ieee` library (if used), followed by the ones for vendor libraries, then project libraries and finally use clauses for the `work` library.

Example:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
library unisims;
use unisims.VPKG.all;
library proj_pci;
use proj_pci.pci_defs.all;
use work.my_pkg.all;
```

Reason: Makes search of dependencies easier.

### 5.2.5 [UseClause] [M] Place of use clause

Use clause are not allowed outside of context clauses.

Reason: Makes search of dependencies easier.

### 5.2.6 [EntityLayout] [M] Layout of entity declaration

An entity declaration must be: `entity`, space, `entity name`, space, `is`, newline. If there are generic interfaces, they must be declared one per line. In a block of generics (sequence of comment and generic declarations without any empty line), the name, the colon, the type, and the default value (if present) must be aligned. Likewise for ports.

There must be a new line after the first `'( but not before the last `')'.

If there is only one generic (or one port), it is allowed to have the declaration on the same line as `generic` or `port`.
For ports, the mode \textit{(in, out, inout)} cannot be omitted.

Example:

```vhdl
generic (  
  -- If set to 1, then use small calibration counter to speed up simulation  
g_simulation : integer := 0;  
g_with_external_clock_input : boolean := true;  
--  
g_board_name : string := "NA ");
```

Reason: Alignment makes code easier to read.

5.2.7 \textbf{[ComplexStmtLayout]} [M] Layout of complex statements

For the \textit{if} statement, the \textit{then} must be on the same line as the \textit{if} or the \textit{elsif}, unless the condition is too long. In the latter case, the \textit{then} must be on the same column as the related \textit{if} or \textit{elsif}. The \textit{if}, \textit{elif} and \textit{end if} must be on the same column.

Example:

```vhdl
if condition1 then  
  ...
elsif condition2 then  
  ...
elsif (this_is_a_very_long_condition  
  and with_another_very_long_condition)  
  then  
    ...
end if;
```

Likewise for \textit{for} loop and \textit{while} loops: the \textit{loop} keyword must be either on the same line when it fits or on the same column as the \textit{for} or \textit{while}. The \textit{end loop} must be on the same column as the \textit{for}, the \textit{while} or the \textit{loop} keyword (for simple loop).

Example:

```vhdl
for i in arr'range loop  
  ...
end loop;
```

For \textit{case} statement, the \textit{is} must be on the same line or the same column as the \textit{case}. Alternative must be indented. Statements in alternative must also be indented except when there is only one simple statement. In the latter case the simple statement can directly follow the \textit{=>}.

Example

```vhdl
case state is  
  when S_INIT =>  
    ...
  when S_S1 | S_S2 =>  
    ...
end case;
```

```vhdl
case addr(2 downto 0) is  
  when "000" => s <= "010";  
  when "001" => s <= "001";  
  when others => s <= "000";
end case;
```

Reason: Makes easier to identify nested statements.
5.2.8 [BeginEndLayout] [M] Layout of begin/end keywords

In many VHDL constructs, the \textbf{is} introduces declarations, the \textbf{begin} statements and the \textbf{end} terminates the statements.

The \textbf{begin} and the \textbf{end} must always be on the same column. If there are declarations, the \textbf{is} must be on a new line otherwise it must be on the same line.

Examples:

\begin{verbatim}
-- Without declarations:
procedure pack
    (signal din : t_data_type) is
begin
    ...
end pack;

-- With declaration of c_CST:
procedure pack2
    (signal din : t_data_type)
is
    constant c_CST : natural := 5;
begin
    ...
end pack;
\end{verbatim}

Reason: Usual indentation rule.

5.2.9 [EndLabel] [M] Presence of the label after end

In following constructs, the \textbf{end} must be followed by the name of the construct:

- entity declarations
- package declarations and bodies
- architecture bodies
- configuration declarations
- subprogram bodies
- physical types
- record types
- protected types
- loop, case and if statements (when they are labelled)
- block statements
- process statements
- generate statements

Example:

\begin{verbatim}
function atoi (str : string) return natural is
begin
    ...
end atoi;
\end{verbatim}

Reason: Makes navigation easier.
5.2.10  [Instantiation] [M] Layout of instantiation

For component or entity instantiation, generics and ports must be associated by name, following the order of the declaration, one per line, and the arrows must be aligned. Label and instantiated unit must appear on the first line. If the instance has generics, generic map ( must be on the second line, followed by the association. port map ( must appear on a separate line.

Example:

```vhdl
sync_gating_pulse : gc_sync_ffs
    generic map (g_sync_edge => "positive")
    port map (clk_i => clk_fbck_i, rst_n_i => rst_fbck_n_i, data_i => gate_sreg(0), ppulse_o => gate_p);
```

Reason: Makes code more readable.

5.2.11  [ProcessLabel] [M] Label of processes

Each process statement must either have a label (which clearly indicates its purpose) or a comment just before the statement. Reason: A short label might be as describing as a long sentence.

5.2.12  [Parenthesis] [M] Use of parenthesis in expressions

Parenthesis in expressions are used to make evaluation order explicit. You do not need explicit parenthesis when the normal order of arithmetic operations is used (*, /, +, -).

Parenthesis around conditions in if and while statements must not be used, unless the condition spans on multiple lines. Reason: Avoid to make expression larger than needed.

5.3  Identifiers

5.3.1  [Keywords] [M] Keywords case

Keywords (VHDL reserved identifiers) must be written in lower case. Reason: Lower cases is common.

5.3.2  [Identifiers] [M] Identifiers case (I)

In general, identifiers should be written in lower case with the exception of acronyms. Reason: Common practice even for normal texts.

5.3.3  [Underscores] [M] Use of underscore in identifiers (I)

If an identifier is composed of words, they should be separated by an underscore. Do not use CamelCase (compound words such as each word begins with a capital letter without any space or underscore between words).

Example:

```vhdl
counter_gate
```

Reason: Use of underscore is more readable than CamelCase.
5.3.4 [ReferenceName] [M] Reference

When an identifier references a named entity, it must have exactly the same casing (even if VHDL is case insensitive).
Reason: Coherence.

5.3.5 [ArchNames] [M] Architectures name

The name of architecture must be arch. In the particular case of multiple architectures (like per vendor architectures), other names are allowed.
Reason: Neutral name that supports any implementation (behavioural, structural, rtl, …) style.

5.3.6 [Constants] [M] Constants name

Constant declaration identifiers should be in UPPER case, with a c_ prefix.
Reason: Follows the C convention, and clearly make the difference between constants and generics.

5.3.7 [GenericsName] [M] Generics name

Generic identifiers should also be in UPPER case, with the g_ prefix.
Reason: Same as the Constants rule.

5.3.8 [PortsName] [M] Ports name

Ports name must be in lower case (as ruled by Identifiers), but must also have a suffix:
• _i for normal input.
• _o for normal output.
• _b for bidirectional port.

The suffix must be the last one.
Reason: Helps to specify the purpose of a port, makes the dataflow more obvious.

5.3.9 [SignalsName] [M] Signals name

Clock signals (and ports) must have the clk_ prefix. In case of multiple clocks, the clk_ prefix is followed by the clock domain name and the frequency if known.
Reset signals (and ports) must begin with the rst_ prefix. In case of multiple clocks, there should be one reset per clock, and the rst_ prefix is followed by the clock domain name. Each reset signal is synchronized with its clock.

Active-low signals (and ports) must have the _n suffix (first suffix). Asynchronous signals (and ports) must have the _a (before _n).

If a pulse signal is derived from another signal, it should have a _p suffix.

Delayed signals must have the _d suffix. If you have more than one cycle, then you can either put an index (_d1, _d2, …) or create a vector starting from index 1.
Reason: Purpose of clock and reset signals are therefore made obvious, semantic of active-low signals is also made obvious. Avoid to create funny names for pulses and delayed signals.
5.3.10  [TypesName] [M] Types name
User declared types and subtypes must have the t_ prefix.
Reason: Makes purpose of the type more obvious.

5.3.11  [PackageName] [M] Packages name
The name of packages must have the _pkg suffix. No entity can have that suffix.
As a consequence, the filename for a package also finishes with _pkg.
Reason: Makes the identification easier.

5.4  Language subset

5.4.1  [VHDLVersion] [M] VHDL standard version
Synthesizable units must follow the VHDL-93 (IEEE 1076 1993) standard.
Reason: The latest revision (2008) is not fully supported by all tools.

5.4.2  [IEEEPkg] [M] Use of IEEE packages
The only IEEE packages allowed are:

- std_logic_1164
- numeric_std
- numeric_bit
- math_real
- math_complex
- std_logic_misc
- std_logic_textio

(Note that the last two ones are not standard when VHDL-93 was defined).
From std_logic_misc, only the reduce functions can be used.
In particular, std_logic_unsigned, std_logic_signed and std_logic_arith are not allowed. Use numeric_std instead.
Reason: Makes the code more portable.

5.4.3  [NoUserAttributes] [M] Attribute declarations not allowed
It is not allowed to declare attributes, except for tool specific attributes.
List of allowed attributes: keep, shreg_extract, opt_mode, resource_sharing, altera_attribute.
Reason: The initial purpose of the attribute is to convey informations to a tool.

5.4.4  [NoUserAttrName] [M] Attribute names
The only attribute names allowed are those specified by the language. User attribute names are not allowed.
Reason: Consequence of the NoUserAttributes rule (but extended to tool attribute).
5.4.5 [EntityItems] [M] Entity declarative items

The only declarative items allowed in entity declarations are:

• attribute declarations
• attribute declarations

The only concurrent statement allowed in entity declarations is:

• assertions

Reason: Obscure features, but assertions may be used to document the interface, and the only place to specify attributes for ports and generics are within the entity declarations.

5.4.6 [NoCharEnumLit] [M] Character as enumeration literal

All user defined enumeration must use names for literals. Characters are not allowed.

Reason: Use of characters is always special and reserved for bit and strings.

5.4.7 [GuardedSignals] [M] Guarded signals

Guarded signals (bus or register signal kind) are not allowed.

Reason: Usually not supported by synthesis tools.

5.4.8 [Disconnection] [M] Disconnection Specification

Disconnection specifications are not allowed.

Reason: Useless and guarded signals are not allowed.

5.4.9 [BlockStatement] [M] Block statements

Block statements can be used to group concurrent statements or to create a scope for declarations. Ports, generics and implicit GUARD signals are not allowed in block statements (therefore port maps and generic maps are also not allowed).

Reason: Mostly useless features.

5.4.10 [GroupDeclaration] [M] Group and group template

Group template declarations and group declarations are not allowed.

Reason: Useless feature

5.4.11 [PortMode] [M] Buffer and linkage mode

The modes linkage and buffer are not allowed. If you need to read from an output port, use an intermediate signal.

Reason: Obscure features.

5.4.12 [ConfigSpec] [M] Configuration specification

Configuration specifications are not allowed.

Reason: Obscure and mostly useless feature.
5.4.13 [RemovedSynth] [M] Language features not allowed for synthesis

These following features must not be used for synthesizable units:

- Configuration declarations for synthesis.
- Function declarations with an operator symbol.
- Resolution function other than resolved defined in ieee.std_logic_1164.
- New integer and physical type declaration.
- Wait statement.

Reason: They are often not supported by synthesis tools.

5.5 Synthesis rules

5.5.1 [PortsType] [M] Type of top-level ports

A top-level entity is the main entity of a core.

The type of the ports in a top-level entity must be either std_logic or std_logic_vector, or a user-defined bounded composite type (array or record) composed of these types. This excludes signed, unsigned or bit types.


5.5.2 [GenericType] [M] Type of top-levels generics

The type of a top-level generic must be either string, integer, boolean, std_logic, std_logic_vector, an enumerated types, or a user-defined bounded composite type of these types.

Reason: Same as PortsType

5.5.3 [WrapperUnit] [R] Wrapper of top-level units

It is convenient to group bus signals in records as this reduces the number of connections. But other HDL languages (in particular Verilog) do not have any equivalent feature for records.

So there can be two versions of the top-level unit: a wrapped one and a non-wrapped one. The name of the wrapped unit is the name of the normal one but with the x prefix.

Each bus should have two records: one for the input signals and one for the output signals. The records should be declared in a package.

The default top-level entity should be the wrapped version, the un-wrapped version must only unwrap the signals.

Reason: As Verilog is deprecated in favour to SystemVerilog, the wrapped unit should be the default.

5.5.4 [RegisterTemplate] [R] Process for a register.

Use only registers triggered on the positive edge of the clock and with a synchronous reset.

Write it using this template:
process(clk)
begin
  if rising_edge(clk) then
    if rst_n_i = '0' then
      q <= '0';
    else
      q <= d;
    end if;
  end if;
end process;

The sensitivity list is composed of only the clock. The process has one if statement, whose condition is rising_edge of the clock. The if statement has one if statement to reset the registers and do the computation.

Reason: This is the simplest way to create a register. Having one way makes register identification easier.

5.5.5 [AsyncReset] [M] Asynchronous reset

If the reset is asynchronous, it must be synchronously deasserted.

Reason: This simplifies timing analysis, as the asynchronous reset could then be considered as a normal signal.

5.5.6 [RegisterReset] [M] Register reset

All registers must be initialized during reset

Reason: In an FPGA, it takes no additional resources and gives a known initial value.

5.5.7 [SignalAttribute] [M] Signal attributes

Do not use signal attributes (Event, Active, Delayed...) for synthesis. Use function rising_edge (and maybe falling_edge).

Reason: Makes code shorter and more uniform.

5.5.8 [VectorDirection] [M] Direction of indexes

When declaring subtypes of std_logic_vector, use the downto direction. You can use the to direction to declare RAMs (arrays of std_logic_vector) or other arrays.

Reason: Avoid null ranges.

5.5.9 [EmptyArray] [M] Minimal length of arrays

Do not declare arrays of length 0 or 1 (unless the bounds are computed).

Reason: Confusing and useless in general.

5.5.10 [ClockResetPorts] [M] Clock and reset ports

When clock and reset signals are in a ports list, the main clock must be the first port, the main reset must be the second port.

Reason: Makes identification easier.
5.5.11  [ClocksUse] [M] Usage of clocks

Clocks must only be used in component associations or within rising_edge or falling_edge. Logics on clocks are not allowed, except in special and commented cases.
Reason: Synchronous design good practice.

5.5.12  [FSMCoding] [R] FSM code style

There is no mandatory rules for FSMs, we recommend the simplest implementation:

- If all outputs are a function of the current state, the FSM can be written using only one process (Moore machine).
- If outputs are a function of the current state and of the current inputs, the FSM can be written using two processes (Mealy machine). The first process can handle the register (assign current state from next state on clock edge), and the second process can compute the next state and the outputs.

Reason: Do not make code more complex than needed.

6  Appendix: Example

TBD

7  Appendix: Future rules

Possible future rules:

- Require counter-generate for undriven out ports
- Do not allow unused declarations

8  Appendix: Document format

The asciidoc file format was selected because:

- the file can be read in raw form, annotations are very light
- it could be rendered in html or pdf
- vhdl syntax highlighting is supported (at least with the hightlight tool)
- it is recognized by platforms like github

We have also considered other formats; markdown rendering is not very good for books.
Rules are described so that the list of the rules and the list of rules to be manually checked can be easily extracted from this document.
See the Makefile for creating a pdf or an html page.

9  Appendix: Changelog