Guidelines for Writing VHDL Models in a Team Environment

Janick Bergeron
Bell-Northern Research Ltd
P.O. Box 3511, Station C
Ottawa, Ontario, Canada K1Y 4H7
janick@bnr.ca

Abstract

When a large VHDL model is being developped by several individuals, a set of guidelines is necessary to ensure maintainability, consistency and readability throughout. This paper presents a set of guidelines covering issues like file naming, capitalization and code layout, successfully used within BNR.

Section 1. File System Structure

A file shall contain one and only one library unit. This minimizes the amount of recompilation required when a library unit, on which other library units depend, is modified. It also helps in structuring the model in the host computer's file system as described in this section.

Each file shall be named according to the unit it contains as shown in Table 1. This makes identifying the file that contains a particular library unit easier and, when alphabetically sorted, each entities are grouped with its corresponding architecture and configuration, and package declarations are grouped with their corresponding body. Since certain file systems are case-insensitive, the case mix of the filename need not follow the case mix of the various library unit names it is made of. If the file system imposes a limit on the length of a filename, a different convention that yields shorter names should be used (but a convention nonetheless) in favor of keeping longer, more meaningful names for the library units. Note that a dash (-), not an underscore, separates the components of the filename since the underscore is used as a word separator in user-defined identifiers (see section 2).

Table 1: File Naming Convention

Entity	*entityName*.**vhd**
Architecture	*entityName-archName*.**vhd**
Configuration	*entityName-confName*.**vhd**
Package	*packageName*.**vhd**
Package Body	*packageName*-**body.vhd**

The structure of the file system hierarchy shall mirror the logical structure of the system being modelled. This makes it easier to locate the model of a particular sub-module of the system. It does not imply that each component in a structural description have to be in individual subdirectories: a structural description of a "logical" structural component of the overall system should be in a single directory. Soft-links may be used to "instantiate" a directory containing a re-used subcomponent into the various directories where it is used.

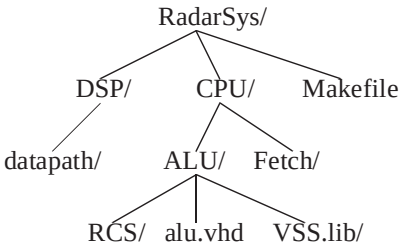

Figure 1: File System Hierarchy

A directory shall correspond to one and only one library. This is a requirement of certain toolsets. It also reduces clutter of library units in a single library.

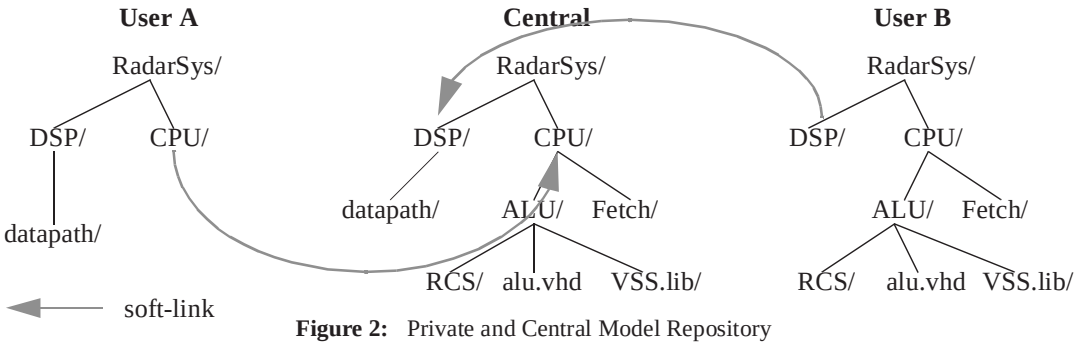

Figure 2: Private and Central Model Repository

A directory shall have the same name as the top-most unit it contains. The top-most unit should be the only unit in the library usable outside of this library and all others considered private. This facilitates the identification of the top-level entities in a "logical" structural model. This guideline does not apply when a directory (and library) contains several shared packages.

*The name of a VHDL library shall be "directoryName**_LIB**".* It facilitates the identification of the library that contains the units located in a given directory and vice-versa. It implies that the various directories in the hierarchy containing the model must have unique names but that is also almost garanteed by the previous guideline.

*If the VHDL system requires a directory or a file for its own data or object files, it shall be named "toolName**.lib**".* This allows the use of several VHDL systems on a single source hierarchy. If several version of the same tool are used, the version number should also be included.

```
Vantage-3.3.lib
Vantage-4.1.lib
VSS.lib
```

A complete copy of the entire hierarchy of the model shall be kept in a shared central location. This copy is a snapshot of the latest working version of the model others can refer to. Developpers copy their portion of the model into this central repository at appropriate points throughout the development

Each member of the development team shall have a copy of only the portion of the hierarchy for which (s)he is responsible for. Each member creates a complete picture of the model by having soft-links to the missing portions in the central repository (see figure 2).

Each directory shall contain a makefile. A rule named "**all**" should compile all units contained in the directory and invoke *make* for all subdirectories. If several toolsets are used, the makefiles should be named "**Makefile**.*toolName*" and a soft-link named "**Makefile**" should point to the makefile for the tool currently used. BNR has released in the public domain a tool to automatically create a makefile for various VHDL toolsets from a set of VHDL source files[1].

```
all:
    for dir in $(SUBDIRS); do \
        cd $$dir; \
        make all; \
        cd ..; \
    done
    make all
```

All VHDL source and testcase input files shall be maintained using a source control system. Systems like (but not limited to) SCCS or RCS should be used. BNR has a PERL script which provides an interface identical to Digital's CMS to both systems. The source control system files should be kept in a subdirectory named "RCS" or "SCCS" in each directory containing VHDL source files.

Section 2. VHDL Syntax

All VHDL reserved words shall be in lowercase and all other keywords shall be in uppercase. User-defined keywords should not depend on capitalization to be readable or meaningful since a tool or filter may recapitalize the VHDL source. It also makes user identifiers stand out from the VHDL keywords.

```
entity MEMORY is
   port(WHERE: in ADDRESS_TYP;
        WHAT: inout DATA_TYP;
        WRITE_IT: in BOOLEAN);
end entity MEMORY;
```

User-defined identifiers shall be meaningful and have words separated by underscores. Abbreviations and acronyms should be avoided at all cost because they reduce readability and maintainability. Identifiers should have at least 8 characters.

```
PROGRAM_COUNTER
LEFT_DATA_BUS
```

*User-defined type and subtype identifiers shall end with "**_TYP**".* Coming up with significant but different names for signals, variables and type marks is often difficult and make the type mark more meaningful.

```
type DATA_TYP is ...;
subtype ADDRESS_TYP is ...;
```

*User-defined package identifiers shall end with "**_PKG**".* This makes it possible to have similar names for the types and the packages that contains them. It also makes the package name more meaningful.

```
package INT_64_BIT_PKG is
    type INT_64_BIT_TYP is ...
end package INT_64_BIT_PKG;
```

All predefined attributes shall be in lowercase. Although they are not reserved words, it facilitates differentiating them from user-defined attributes.

```
if CLOCK'event and CLOCK = '1'
```

The identifiers in predefined packages shall be used in uppercase. Since the identifiers in packages like STD.STANDARD or STD_1164_STANDARD can be overloaded as any other user-defined identifier, they should be capitalized the same way. It also helps identifying them as separate from the language itself.

Underscore shall not be used in literals. This lexical convenience mechanism is questionable as some downstream tool may not accept it and complicates the automatic translation of VHDL into other languages that do not support it.

Only literals in base 2, 8, 10 or 16 shall be used. These bases are the only ones most computer scientists and hardware designers are familiar with.

Extended digits in base-16 literals and base specifiers shall be in uppercase. This makes for more consistent and readable literals.

```
ADDRESS_BUS := 16#45E7FF0A#;
DATA_VALUE := 0"0377";
```

Real literals shall be in decimal only. Non-integer mantissas and exponents in based literals are very confusing.

Allowable replacement characters shall not be used. They are a provision for limited character sets and should be avoided at all cost. This section of the LRM (13.10) makes for an excellent "Trivial Pursuit" question amongst collegues.

Section 3. VHDL Source Layout

Declarative regions and blocks of statements shall be indented by 4 spaces. A block of statement can be the concurrent statement part of an architecture, the "else" clause of an **if** statement, the body of a subprogram, etc...

```
process
    variable VARIABLE_NAME ...
begin
    if CONDITION then
        STATEMENT;
        STATEMENT;
    else
        STATEMENT;
    end if;
end process;
```

Indentation levels in sequential statements shall not exceed 4. A large number of indentation levels is often an indication of bad programming style. Use subprograms to break the code into manageable parts.

```
begin
    if CONDITION then
        loop
            if CONDITION then
                loop
                    LAST_LEVEL;
                end loop;
            end if;
        end loop;
    end if;
end;
```

Indented regions in sequential statements shall not have more than 60 lines. This keeps indented regions from spanning more than two pages when printed which makes inspecting the code easier when a complete control region is visible. Long indented regions are often an indication of bad programming style. Use subprograms to break the code into manageable parts.

The TAB character shall not be used to indent. Only used the SPC character. A mix of TAB and spaces may result in improper indentation when viewed in an environment with different tabstop settings. The TAB key can still be used to indent code in EMACS in an appropriate VHDL mode but should only insert space characters. Use the UNIX command expand(1) to replace TABs with spaces.

Lines shall not exceed 80 characters in length. It avoids confusing wrap-arounds when viewing the source on a regular text terminal, standard-sized window or when printed.

Long lines shall be broken where there is white spaces. Breaking a line between adjacent tokens is confusing.

Line continutations shall be indented to line-up with the first token at the same nesting level or by 4 spaces. It makes it possible to quickly distinguish the continuation of a line from a new statement.

```
SINK = FUNCTION_NAME(PARAM1,
                     PARAM2,
                     PARAM3);
variable LONG_NAME =
    VERY_LONG_EXPRESSION;
```

Comments shall be on a line of their own. Trailing comments are cumbersome when the line is lenghtened or shortened and often need to be reformatted. They are also difficult to see unless they are located close to the right margin.

Multi-line comments shall start and end with with an empty comment line. It makes the comment paragraph stand out from the surrounding code. Single-line comments should be reserved for strategic comments in hard-to-understand code.

```
--
-- This is a multi-line
-- comment followed by the
-- code it describes
--
STATEMENT;
```

Comments shall be immediately followed by the code they describe. One should be able to read a comment to help understand the code one is about to read, not the other way around.

Each file shall have a descriptive comment of its content a the top. This can include the name of the authors and subsequent contributors, copyright notices and a description of the content of the file.

```
--
--  Interface of RS-232 modem
--
--  Janick Bergeron
--  Bell-Northern Research Ltd
--  janick@bnr.ca
--
--  (c) Copyright
--      Northern Telecom Ltd.
--      All rights reserved.
--

entity RS_232_MODEM is
    ...
end RS_232_MODEM;
```

Concurrent statements (and their descriptive comments) shall be separated by 2 blank lines. This clearly separates the section of codes which execute in parallel.

```
-- Clock generator
CLOCK <= not CLOCK after 5 ns;


-- State Transition
process(CLOCK)
begin
    STATE <= NEW_STATE;
end process;
```

Groups of logically related statements and declarations shall be separated by 1 blank line. A group is a sequence of statements or declarations performing a given task at the same indentation level as other goups and includes its descriptive comment. It visually separates sections of codes that perform different tasks.

```
-- Decode the instruction
STATEMENT;
STATEMENT;

-- Execute the instruction
STATEMENT;
```

*Choices in a **case** statement shall be separated by 1 blank line and not be indented.*. The sequence of statements should immediately follow the case alternative specification and be indented normally by 4 spaces. This visually separates the various alternatives to choose from.

```
case EXPRESSION is
when CHOICE1 =>
    STATEMENT;

when CHOICE2 =>
    STATEMENT;

when others =>
    STATEMENT;
end case;
```

Unless otherwise specified, tokens shall be separated by 1 space. This aerates the code and makes it easier to read.

No space shall preceed a close parenthesis, comma, colon or semi-colon nor follow an open parenthesis and no space shall surround a single quote or dot. Furthermore, an open parenthesis that encloses function arguments or array indices should abut its subject array or function name. This makes the code look more like written natural language.

```
VAR := FCT(A, B.C) + D'right;
EXPRESSION := A * (B + C);
```

Each statement shall start on a new line. More than one statement on a single line makes the code difficult to read or scan quickly. This includes statements within statements.

```
STATEMENT;
if CONDITION then
    STATEMENT;
else
    STATEMENT;
end if;
```

Each declaration shall start on a new line. It makes it easier to identify individual ports, generics, signals and variables, change their order or modify the type of one of them.

```
signal NAME1: SIGNAL_TYP;
signal NAME2: SIGNAL_TYP;
```

Elements in interface declarations shall be vertically aligned. This allows quick identification of the various kind of interface declaration, their name, direction and type.

```
procedure FOO(
    signal   FORM1:  in    A_TYP
    variable FORM2:  inout BIT;
    constant FACTOR: in    REAL;
             RESULT: out   BIT);
```

Elements in named associations than span more than one line shall be vertically aligned. It makes identifying the formals and actuals easier.

```
CALL(FRM1 => ACT1, FRM2   => VAL,
     FRM3 => PI,   RESULT => Z);
```

Concurrent statements shall be labeled. It facilitates cross-referencing when a statement is replaced by a component instantatiation in an alternative architecture and makes for better documentation.

```
-- Clock generator
CLOCK_GENRATOR:
CLOCK <= not CLOCK after 5 ns;
```

Loop statements shall be labeled. It enables better loop control with the **next** and **exit** statements.

```
INFINITE_LOOP: loop
    STATEMENT;
end loop INFINITE_LOOP;
```

***Next** and **exit** statements shall specify the loop label they control*. It makes for more readable code in nested loops and will prevent errors if a portion of the body of the loop containing a **next** or **exit** statement is included in an inner loop later.

```
CONTROL_LOOP: loop
    exit CONTROL_LOOP;
end loop CONTROL_LOOP;
```

*Whenever possible **End** keywords shall be qualified.* This makes identifying the corresponding **end** of a statement with a body much easier.

```
CLOCK_GENERATOR:
process
begin
    CLOCK <= not CLOCK;
    wait for 5 ns;
end process CLOCK_GENERATOR;
```

Section 4. VHDL Constructs

Named association shall be used preferably to positional association. It reduces the risk of errors when adjacent parameters are of the same type and when parameters are added or removed. It also makes using default values easier and conversion functions on **out** and **inout** formals cannot be used if positional association is used. It also applies in literals for arrays and records.

```
FCT(FORMAL => ACTUAL);
MEMORY := (0 => -1; others => 0);
INT64 := (MSB32 => 0; LSB32 = 1);
```

***Buffer** ports shall not be used*. Although handy when an **out** port needs to be read, they do not have any correspondence in actual hardware and they impose restrictions on what can be connected to them. The name “buffer” suggests the presence of a logical buffer which isolate the internal value from the external drivers, a buffer which is not present when a netlist is automatically generated from the structural description. If internal feed-back in required, use an **out** port with an internal signal and a concurrent signal assignment.

```
signal OUT_VALUE: VALUE_TYP;

-- Emulate buffer port
BUFFERED_PORT:
OUT_PORT <= OUT_VALUE;
```

***Linkage** ports shall not be used.*We are still not sure what **linkage** ports are. Coolant fluid ?

Blocks shall not be used. The GUARD signal in a guarded block can be explicitely emulated and such an emulation is easier to understand as the semantics are obvious in the model rather than hidden in the simulation engine. Guarded blocks also make it difficult to refine a model toward a synthesizeable description. Blocks with ports and generics are an attempt at structural decomposition without using the structural modelling constructs; however, the declarations in the enclosing scopes are still visible which does not make a block a true structural decomposition construct thus a proper instantiation of an entity/architecture should be used. Blocks can also be used to create a declarative region: they are useful when declarations are only needed by a subset of the concurrent statements in an architecture but cannot be used if the subsets intersect. Furthermore, block-level declarations can overload a declaration in the enclosing scopes which may lead to confusion or errors if a process is moved outside the block. This guideline does not apply when a block is required to create a declarative region needed by a **generate** statement

Guarded signals shall not be used. They are the feature most often not (or badly) implemented in VHDL tools. Their semantics are confusing and they are difficult to debug. They cannot be initialized to the disconnected state and are difficult to refine into a synthesizeable form. It is preferable to model the behavior of the guarded signal within the model itself by having a “disconnect” value properly handled by the resolution function. For example, using a record with a “disconnect” flag member has the advantage that a signal can be easily disconnected then reconnected with the same value, which is not possible with a guarded signal.

Conditional signal assignment shall not be used. Most of the time, a signal changes under a given set of conditions and otherwise remains unchanged. Since the conditional signal assignment requires an **else** clause, this clause is often used to simply assign the current value of the singal to itself again. This makes the statement sensitive to itself and will cause its evaluation twice as often as necessary. Furthermore, the order in which the value and conditions are specified is counter-intuitive. Use an equivalent process statement instead.

Operators shall not be overloaded lightly. Only overload an operator to perform an operation logically equivalent to its original intent and to yield

more readable expressions. Operator overloading is a powerful mechanism for writing concise and easy to understand descriptions but can be a source of confusion if they do not do what the reader expects. Keep also in mind that the priority of the operators remains the same, reguardless of its overloaded function.

Section 5. VHDL Coding Style

Variables shall be used in preference to signals. Signals carry more overhead than variables do. Unless something needs to be seen in another process, use a variable.

Attributes ***'range*** *and* ***'reverse_range*** *shall be used when scanning arrays.* Assuming specific bounds makes resizing or changing the direction of the indexes more difficult and may not be caught at compile time. If a zero-based (or n-based) index is required, create it in combination with **'left** or **'right**.

```
SCAN: for I in A'range loop
    A(I) := VALUE;
    ZERO_BASED := I - A'left;
end loop SCAN;
```

Variable-width ports shall be constrained using generics. It makes for better documentation of the relationship between the size of different ports, provides better control on the numbering and direction of the index and lets the VHDL environment perform a complete width consistency check automatically. Also, downstream tools, such as synthesis, that work on the uninstantiated version of the entity will require size information for the variable-width ports. This guideline does not apply to formal parameters of subprograms since they don't have generics.

```
entity ADDER is
    generic(N: NATURAL);
    port(A: in VECT(1 to N);
         B: in VECT(1 to N);
         S: out VECT(1 to N+1));
end ADDER;
```

Enumerals shall be used to represent non-arithmetic discrete values. Unless arithmetic operations have to be performed on discrete values, enumerals provide better documentation of the individual values. Most synthesis tools will have a mechanism to specify numeric values for specific enumerals at the implementation-level.

```
type OPCODES_TYP is (
    ADD, SUB, MAD, CLR, LOD);
```

Constants shall be used to represent limits and parameters. It makes it easier to globally change these limits and parameters and makes for better documentation.

```
constant SIZE: NATURAL := 32;
variable MEM: array(1 to SIZE)
    of INTEGER;
```

Processes with a sensitivity list shall not be used. They require special treatment when they are run during initialization since none of the signals in the sentivity list has experienced an event and requires a condition which is evaluated every time the process is run, condition which must be appropriately modified whenever the sensitivity list is changed. A process with a loop and an explicit wait statement better separates the initialization and run-time portions of the process. This guideline does not apply to synthesizeable descriptions.

```
BETTER: process
begin
    -- Initialization stuff
    INITIALIZATION_CODE;

    INFINITE_LOOP: loop
        wait on SENSITIVITY;

        -- Actual model code
        STATEMENT;
    end loop INFINITE_LOOP;
end process BETTER;
```

The ***'event*** *attribute shall be used explicitely when testing for a change to a particular level.* Relying on the fact that the process as been awakened to assume that a signal has changed is not robust. If more signals are added to the sensitivity list, this assumption is no longer valid.

```
wait on CLOCK;
if CLOCK'event and CLOCK = '1'
```

There shall be no more than 2 process statements in an architecture. Processes cannot be tested in isolation and having a lot of parallelism makes understanding and debugging an architecture difficult. A structural decomposition is more appropriate. When there are several processes in a

synthesizeable description, it may be difficult to identify the portion of the description which creates innefficiencies in the synthesized hardware. There can be additional concurent signal assignements and component instantiation statements.

Section 6. Conclusion

These guidelines are not intended to be used as a strict set of laws to abide by under all circumstances, but as a starting point to make the VHDL source code in a large model have a consistent "look-and-feel", reguardless of who actually wrote it. They also aim at making the code more portable and maintanable. Use them wisely and adapt (or add to) them to fit your particular situation. Remember one thing: **Be consistent!**.

Section 7. Aknowledgements

This set of guidelines is the product of the contributions, experience and comments of several individuals. I would like to thank (in no particular order) Himanshu Thaker, Bernard Doray, Silvana Romagnino, Barry Willms and Parviz Yousefpour from Bell-Northern Research, Duncan Kitchin and Dan Clarke from BNR Europe and Neal Ziring from the US Department of Defense.

Section 8. References

[1] **vmkr**, written by Himanshu Thaker (hemi@bnr.ca), is available via anonymous ftp on *thor.ece.uc.edu* in */pub/Vdhl/tools/vmkr.2.7.tar.Z.* and is distributed under the terms of the *GNU copyleft.*