Guidelines and conventions for VHDL representation of digital hardware

Lars Svensson
larssv@chalmers.se

Version 1.2, August 19, 2015

1 Introduction

This is a collection of rules and guidelines intended to help you develop working designs in VHDL. It is intended as a companion for a VHDL textbook or language reference, *not* as a replacement for either. It builds heavily on several sets of rules and guidelines listed in Section 6. It is intended to be brief; therefore it is assumed that the reader knows basic VHDL syntax and semantics, and has some understanding of how code is turned into hardware.

Hardware description languages are superficially similar to programming languages such as C or Java, but there are also important differences. You may recognize some of the rules from similar rule sheets for software development. Other rules are peculiar to hardware design and to VHDL in particular.

In special circumstances, there may be good reasons to ignore some of these rules. When considering such a departure, be sure to *document your decisions,* and weigh the extra time you are likely to spend in verification and debugging your design.

2 Lexical and syntactical issues

Syntax-sensitive editors (such as Emacs, or those built into many integrated design environments) will help the developer observe rules such as these.

2.1 Code readability

Code readability benefits from many of the same rules as in software programmes.

- Use a separate line for each statement or declaration (this practice also makes it easier to comment out selected statements during development).

- Use spaces liberally to separate lexical elements.
- Use parentheses to control precedence in expressions.
- Avoid very long source code lines; 72 characters is a traditional limit. Split lines at "logical" points.
- Use indentation to highlight code structure. 2 spaces is enough to serve the purpose and will not use up too much of the line width. Do not use tab characters; text editors may attempt to convert them to spaces, often with confusing results.

2.2 Comments

Guidelines for comments are similar to those for software programming languages.

- Use comments to explain and elucidate your code. Experience strongly suggests that no code of more than trivial size is self-explanatory, however clear and lucid it may seem at the time of writing.
- Strive for conciseness in comments, and place them close to the code they describe, but not interfering with it.
- Strive to comment logical sections of code rather than single lines.
- Include a standard-format header in each code file, with author name, creation date, modification history, an overall description of the module, and any legal matter which your organization may require (such as copyright notices).

2.3 Names

In any project of more than trivial size, a *naming convention* should be developed, *documented*, and *used*. It helps tremendously when working in a development team or when just coming back to your own code after a few weeks. If possible, before you decide on conventions for a project, verify compatibility with all tools you plan to use.

Be aware that many real-life projects will require you to interface to modules written in other languages (such as VerilogHDL or SystemC). Sticking with a naming convention that works also in other languages, at least for names that are visible outside your module, is likely to minimize your problems. Case in point: VHDL basic identifier names are case-insensitive, whereas VerilogHDL and SystemC identifier names are case-sensitive.

Here is a list of rules that may serve *as a starting point* for a naming convention:

- Use a consistent case for each identifier.

- Use lower-case letters for signal names, variable names, and port names.
- Use upper-case letters for constants and user-defined types.
- Use underscore to make identifiers more readable: `high_byte` is better than `highbyte`.
- Use meaningful and descriptive names whenever possible (so `addr` rather than `a`, etc); but see next point.
- Use short names rather than long ones; some tools will use a concatenation of module and parameter names to refer to a certain cell or signal instance.
- Do not use VHDL "extended identifiers" in your source files.
- Use a consistent name for the clock signal, such as `clk`; in case you use several clock signals, use a common prefix when naming them.
- Use a consistent name for reset signals, such as `rst`.
- Use `_s` as a suffix for signal names and `_v` for variable names.
- Use common suffixes for names of signals or variables that deviate from "default" properties, such as `_n` for active-low signals, `_a` for asynchronous signals, and `_z` for tri-state signals (the last category should be quite uncommon).
- Avoid using VerilogHDL or SystemC (etc) reserved words as identifiers.
- Use a consistent bit ordering in all multibit buses; `(x downto 0)` is suggested.

The *state machine* is a very common design pattern in digital hardware design. We primarily recommend the *Gaisler two-process model*, revisited below in Section 4.6, for state machine representation; if you decide *not* to use this model, the following rules may be useful:

- Use the names `current_state` and `next_state` in state machine descriptions. In case you describe several of these in the same piece of code (a dubious practice), use suffixes such as `_cs` for current state and `_ns` for next state.

2.4 Ports and generics

Ports provide interfaces for modules. A complex module may have many ports, and mixing them up is likely to cause bewildering malfunction. The same goes for generics, which provide module parameters for things like word lengths.

- Decide on a principle for ordering the list of ports, and use it consistently; the following order is suggested:

  - Inputs, bidirectionals (if any), and outputs.
  - Within each category, list clocks, resets, enables, and other control signals, and finish with data and address lines.

- Use a separate source code line for each port; describe individual ports and port groups in comments.
- Use explicit mapping via port and generic names rather than relying on positional association.

3 Code organization

Decide on and document a code organization convention, so as to make it easy to find a given piece of code. Your design tool environment might prefer a certain file organization, or even liberate you from thinking about the files; but beware that you may want to re-use your code in another environment some day.

- Describe only one `entity` and one `architecture` in each file. Use the entity name as the file name.

Digital hardware is often highly regular, so a one-for-one description would be repetitive, making it easy for a typo to hide in the code. Also, when a change is needed, that change must be repeated in each code instance; missing one instance will typically introduce a bug. Use available abstractions to avoid this source of mistakes, to reduce code size, and to speed up simulation and analysis.

- Use functions rather than repeating complex expressions.
- Use loops to improve readability of repetitive code.
- Use arrays and vector operations rather than individual operations per bit.

VHDL does not provide true abstract data types, but the `package` construct can be used to group constants, types, and functions that together describe a reusable part of a design. The `library` construct encapsulates packages. A composite type is best represented by a `record`, which corresponds to a `struct` in C or C++ or a java `class` with fields but no methods.

- Use packages to group related constructs under a single name, and libraries for encapsulation.
- Use records to collect related signals for passing to other entities.

Many language constructs (in particular loops, processes, and components/instances) may be labeled with identifiers. Such labels are very useful in debugging.

- Use labels for all loops, processes, and components.
- Use a naming convention to set labels apart from other identifiers; *xxx*`_loop`, *xxx*`_proc`, and *xxx*`_comp` are suggested.

3.1 Constants

Design constants should never be entered literally in the code. Then, if you need to change the value, you must recall to change it everywhere.

- Use `constant` declarations for design parameters such as word length, and derive related values with arithmetic expressions.
- If you might need to use the same block more than once in the same design, but with different parameter values, use `generic` rather than constant declarations. (This is actually the preferred alternative in most cases.)

3.2 Types

Certain types (such as `integer`) are built into VHDL, and some others are provided in standard libraries. For hardware design, especially `std_logic`, `std_ulogic`, and `bit` (and their associated `_vector` versions) are useful. Functionally, these types overlap to a large extent. Using one of them consistently lets you avoid explicit type conversions.

- Use `std_logic` rather than `std_ulogic` or `bit` unless you have good reasons not to (and document those reasons in that case).
- Use the `numeric_std` package to provide the types `signed` and `unsigned` for two's-complement arithmetic. Avoid `std_logic_signed` and `std_logic_unsigned`, as interactions with the `numeric_std` types are easy to mishandle.

VHDL allows creation of data types by enumeration and by subtyping from existing types. Each of these mechanisms must be handled with care. Enumerated types have no obvious hardware representation; and many disjunct subtypes may necessitate explicit type conversions in expressions and thus make the code difficult to understand. (The exception is subtypes of `integer`, which may indeed make arithmetic more readable without introducing arbitrary width restrictions.)

- Avoid enumerated types in code that is to be synthesized. (Exception: states in state machines fit nicely with enumerated types; cf. Section 4.6).

- Be restrictive in the number of subtypes you declare.

VHDL allows operator overloading, such that a newly defined type may be operated upon with arithmetic operators such as "+". This mechanism may confuse even experienced designers.

- Only use operator overloading if you design a full package of operations on a new type.

4 Hardware corresponding to certain constructs

It is always good to keep hardware in mind while writing VHDL. Designers have over the years arrived at certain design practices which yield good synthesis results.

4.1 Register inference

Strive to use registers rather than latches to keep state. Registers will be synthesized properly if a process block uses edge triggering as in these examples.

```
-- process with asynchronous reset
asynch_reset_proc: process (clk, rst_a)
begin
  if rst_a = '1' then
    ...
  elsif rising_edge(clk) then
    ...
  end if;
end process asynch_reset_proc;
```

```
-- process with synchronous reset
synch_reset_proc: process (clk)
begin
  if rising_edge(clk) then
    if rst = '1' then
      ...
    else
      ...
    end if;
  end if;
end process synch_reset_proc;
```

- Use edge triggering in sequential processes. Follow a process template such as one of those above.
- Use the standard-logic function `rising_edge(clk)` rather than `(clk'event and clk = '1')`. The former expression is shorter and clearer. The latter expression will return `TRUE` even if `clk` goes from `Z` or `U` to `1`, which is probably not what you want.
- Use a reset signal to initialize registered signals, rather than assigning initial values in the signal declarations. (Initial values are commonly ignored by synthesis tools, so hardware might behave differently from simulation.)
- At reset time, all bi-directional ports should be in input state, so as not to inject `X` values into the surrounding logic.

4.2 Latch avoidance

Latches should generally be avoided, as their timing requirements are very different from those for flip-flops, which means that they complicate clock generation and distribution and also static timing analysis. Also, certain common VHDL coding mistakes typically cause latches to be generated; by not ever using latches on purpose, you can use their appearance as a warning sign.

The following rules will ensure that all signals are assigned values for all input combinations; otherwise, a latch will typically be generated to hold the previous value.

- Assign default values for all signals at the beginning of a *combinational* process block. (See Section 4.1 for sequential blocks.)
- Assign output values for *all* combinations of input conditions.
- Use an `else` clause rather than an `elsif` clause to end an `if` construct in a combinational process.

4.3 Combinational feedback

In addition to flip-flops and latches, it is possible to keep state in a feedback loop built from combinational circuits. Such state variables will be unreachable for scan-chain insertion and the circuits will be very difficult to analyze for timing.

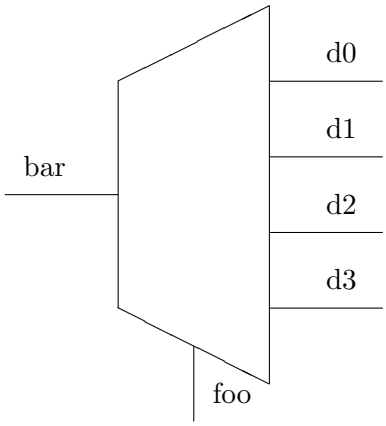
- Do not use combinational feedback loops unbroken by clocked elements (such as flip-flops).

4.4 If-then-else vs. case statements

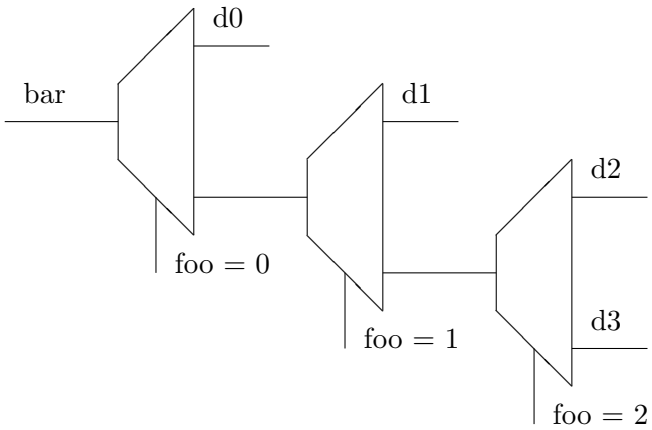
In many software languages, a `case` statement is equivalent to a set of nested `if-then-else` statements which embody the same logic. In VHDL, the two alternatives will typically[1] result in different hardware implementations: the `case` statement produces a single multi-input multiplexer, whereas the `if-then-else` version will produce a chain of two-input multiplexers corresponding to the nested `if` statements. In the latter case, one condition will dominate the others, which may sometimes be useful. Also, the inputs will have different timing properties, and thus it is possible to assign a shorter input-output delay to a late-arriving input signal.

[1] Some design toolchains may optimize these structures such that no difference is seen in the result. You should not trust such optimizations to be available. In any case, many designers would prefer to turn off such optimizations in order to retain more control over the results.

```
case foo is
when 0 =>
  bar <= d0 ;
when 1 =>
  bar <= d1 ;
when 2 =>
  bar <= d2 ;
when others =>
  bar <= d3 ;
end case ;
```




```
if foo = 0 then
  bar <= d0 ;
elsif foo = 1 then
  bar <= d1 ;
elsif foo = 2 then
  bar <= d2 ;
else
  bar <= d3 ;
end if ;
```



4.5 Sensitivity lists

The sensitivity list for a process indicates when the process code should be evaluated. An incorrect list could cause synthesized logic to behave differently than simulations predict.

- Include *all* input signals of a combinational block in its sensitivity list.
- Include the clock signal (and any asynchronous signals, such as an asynchronous reset signal) in the sensitivity list for a sequential block; cf. code examples in Section 4.1.

4.6 Two-process model

VHDL does not enforce a certain style of hardware organization. In particular, it is possible to mix sequential and combinational parts freely. For state machines, a more restricted design style, the *two-process model*, has been observed to improve readability and simplify debugging, especially for larger designs.

In the two-process model, one process each is used for combinational logic and sequential flip-flops. Data input and output from the sequential portion is collected in a record type used only within the module, which makes it very simple to add a signal if necessary. A full description of the two-process model is available from Gaisler Research (see Section 6), and is highly recommended reading.

A more elaborate three-process version of the two-process model separates the combinational logic into two parts: one generating the input for the sequential logic, and one for generating the output signals. It may give clearer code in some circumstances, at the cost of extra verbosity.

- Select the two-process model or the three-process model for sequential logic, and use the selected model consistently.

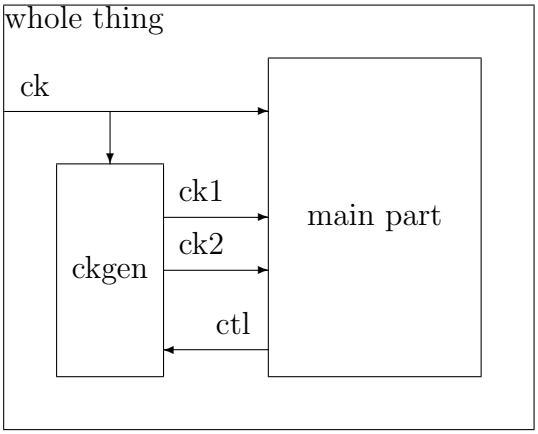
5 Clocking and synchronization

Strive to use simple constructs for clocks and reset signals. In general, simple clocking structures are easier to understand, analyze, and maintain.

5.1 Clock gating

Clock gating is a very useful technique to reduce power dissipation in digital ASIC systems; as most of the dissipation is caused by signal transitions, eliminating unnecessary transitions will typically improve power. Present-day synthesis tools can recognize many cases when the clock can be turned off locally without changing the behavior. At a larger scale, clocks for a module may be turned on and off explicitly with VHDL logic; but functionality of such a clock-gated system may be difficult to verify.

- Do not use gated clocks to define the functionality of a module. The module should behave identically with all clocks active as with a correctly gated clock.
- When explicitly defining gated clocks in your VHDL code, isolate the clock generation circuits in a module of its own, to be able to verify it and the clock-gated module separately.



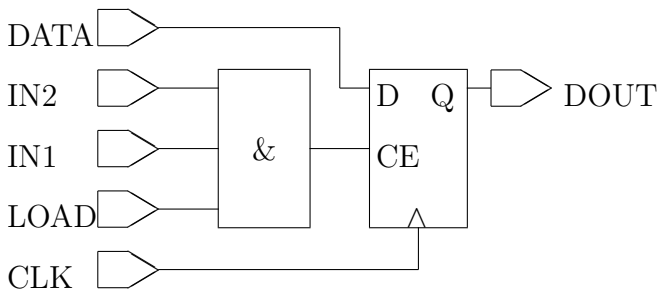
Reset signals should be subject to similar restrictions as the clocks.

- Use a single reset signal for all registers in the module.
- Use a single reset level (either high or low; document!) for all registers.
- Use the reset signal *only* for clearing all flip-flops.
- If you must use a conditional reset, derive the signal in an external module, as for the gated clocks.

5.2 Clocks in FPGAs

While it is in general possible to map the same VHDL code both to ASIC and FPGA platforms, certain constructs are more or less suitable for the different target technologies. FPGAs typically offer clock-enable (CE) inputs on each memory element, but on the other hand the number of clock buffers and clock nets available is limited. As a result, fine-grain clock gating is often impractical, and the corresponding power savings cannot be fully realized. It is still possible to inhibit unnecessary output toggling by using conditions in the CE signal instead:

```
ENABLE <= IN1 and IN2 and LOAD;
process (CLK)
begin
  if (rising_edge(CLK)) then
    if (ENABLE = '1') then
      DOUT <= DATA;
    end if;
  end if;
end process;
```



5.3 Dual-edge clocks

It may occasionally be useful to use both the positive and the negative edges of a clock signal in different parts of a design. If you do, correctness will in general depend not only on the clock frequency but also on the duty cycle, which therefore must be *controlled* and *verified*. Also, be aware that flip-flop scan chains will require all flip-flops to use the same clock edge.

- Avoid using dual-edge clocking as far as possible. Preferrably, use only positive-edge clocking.
- If you absolutely must use dual-edge clocking, document the duty-cycle assumptions you rely on.
- Even when you do use dual-edge clocking, don't let one process trigger on both edges of a clock signal.

5.4 Multiple clock domains

There are cases, especially in large designs, when it is impossible or impractical to use a single clock to synchronize all activity. The design must then be split into several clock domains, each controlled by a separate clock signal. Signals emanating in one clock domain must then be re-synchronized to the clock in the receiving clock domain. Such synchronization of signals across clock-domain borders brings a danger of metastability and malfunction. The exact metastability parameters will often be unknown at module design time, so synchronizer design may have to be revisited at a later stage.

- Use the minimum number of clock domains in order to reduce hardware and design-time overhead.
- Use a single synchronizer design for *all* cross-clock-domain signal transfer.
- Label the synchronizer components distinctively to make special treatment possible, as may be required when modules are assembled into a system.

6 To probe further

Keating, Bricaud. Re-use Methodology Manual for System-on-Chip design, 3rd ed. Springer, 2002.

Jiri Gaisler. A structured VHDL design method. `http://www.gaisler.com/doc/vhdl2proc.pdf`. Accessed on August 19, 2015.

Jan Decaluwe. These ints are made for countin'. `http://www.jandecaluwe.com/hdldesign/counting.html`. Accessed on August 19, 2015.

OpenCores HDL modeling guidelines. http://cdn.opencores.org/downloads/opencores_coding_guidelines.pdf. Accessed on August 19, 2015.

Philippe Garrault, Brian Philofsky. HDL Coding Practices to Accelerate Design Performance. Xilinx White Paper, http://www.xilinx.com/support/documentation/white_papers/wp231.pdf. Accessed on August 19, 2015.