Design and Coding Style Guidelines for Synthesizable VHDL-FPGA Code

Cristian Sisterna

Introduction

These guidelines are the results of several years of writing VHDL code for synthesis and verification in several companies and doing the most different kind of designs. Even though it is a long list, the most important points are described and enumerated. It has to be said that following these guidelines does not mean that your hardware description is correct or it's going to work at first shot, but it will help you to avoid the most common mistakes, to order your code and to facilitate the understanding of the described design.

1. Project Organization

G1.1 First of all read through all the specification of the project to get a comprehensive understanding of the requirements to be met.

G1.2 A project should consist of one top-level module and several sub-level modules. The top-level module should only contain the instantiations of the sub-level modules.

G1.3 The name of a *.vhd* file, that is a VHDL component, must match the entity name. It has to be a short and meaningful name describing the functionality of the code. For test benches files add the suffix *_tb* to both the entity name and the file name. Likewise for packages by using the suffix *_pkg*.

G1.4 Each VHDL component should have only one entity and only one architecture.

G1.5 Each VHDL component should have just one functionality or related functionalities.

G1.6 Per each VHDL component use a header that should include at least the following items:

- Project name
- File name
- Author
- Creation date
- Version
- Detailed description about the functionality of the written code

All these items need to be maintained up to date.

G1.7 A project should have at least the following directories and the respective sub-directories:

- RTL Sources: for the .vhd source files (components) related to the project.
  - RTL Components: .vhd source files of the different components that are part of the RTL Sources; i.e. IPs VHDL files.
  - RTL FPGA vendor specific components: .vhd source files generated from the FPGA vendor software, i.e. memory blocks, clock managers, high speed serial interfaces, communication blocks, etc.

- Test Bench:
  - Test bench auxiliary files: like stimulus vectors, memory contents, behavioral models, etc.
  - Test bench package: package that holds functions, procedures, declarations, etc., only related to the test benches.
  - Component level test benches: underneath this directory there can be different sub-directories, one per each component of the system. At least one test bench per component should be written.
  - System level test bench: test bench files used to test the design at the system level. To assure a better test of the system more than one system level test bench should be written.
- Package: package declaration and package body related to the project.
- Synthesis: scripts and synthesis related files, such as synthesis constraint files.
- PAR: place and route related script and the post place and route results.
  - Netlist: hard-coded IPs files.
- Constraints: FPGA implementation constraint file.
- Documents: the entire project's related documents.
- Simulation:
  - Functional Simulation: scripts and functional simulation files, such as ModelSim's .do files
  - Gate Level Simulation: scripts and gate level simulation files (post place and route simulation)
- Bitstream: FPGA configuration file.

G1.8 Definitely adopt VHDL'93 or VHDL 2001. VHDL'87 is already obsolete.

G1.9 Every step involved from the beginning until getting the bitstream must be clearly documented, (including scritps, constraint file(s), etc.) versioned and archived. Identify software versions and tools used per each step accomplished.

G1.10 Do maintain the availability of all the tools used to implement the released design. Assuring the reproducibility of it.

G1.11 Use version control software, such as CVS or SVN.

Project Partitioning

G1.12 For project using multiple clocks, partition the project in components that have a unique clock. Thus, each clock domain operates independent from the others. Exception: synchronizers and clock crossing components.

G1.13 Partition the project having in mind design reuse. Utilize standardized interfaces and write parameterized VHDL code.

G1.14 If there is any logic related to the generation of the reset signal, keep it in a specific reset VHDL component.

G1.15 Keep related combinational logic together, facilitating the logic optimization. Place it along with the input and output registers in the respective VHDL component.

G1.16 Partition the logic with different design goals into separated components. Isolate the critical speed logic from the area critical logic.

G1.17 Keep sharable resources together in a component. Synthesis tool might be able to share large resources as long as they belong to the same VHDL process.

G1.18 If the design is well partitioned, incremental compilation could be used to optimize the design, reducing the compilation time to meet the system performance requirements.

2. Code Writing

G2.1 Do not start writing any VHDL code until the specifications have been completely understood.

G2.2 Use only IEEE libraries: *std_logic_1164* and *numeric_std*. Do not use the commonly used Synposys' libraries std_logic_arith, std_logic_(un)signed, numeric_bit or any other no IEEE library.

G2.3 Write the code in a parameterized style. Use generics.

G2.4 Indent the code, facilitating the read and the comprehension of the code. Be careful in the use of 'tab'. Differences in text editors make the positioning of the tabs unpredictable and can corrupt indentation. Thus, preferably use spaces.

G2.5 When using `case` or `if` statements with multiple branches, use as many indents as necessary, especially when they are nested.

G2.6 When the number of lines of code in a component is very large, organize the code following a logic sequence to facilitate the functionality search. Preferably create sections and sub-sections in the code using different characters, such as ****, &&&&, to graphically represent the different section/sub-section divisions.

G2.7 Comment the general functionality of each section/sub-section.

G2.8 The comments written in the code must include valuable and significant information. Avoid superfluous or trivial comments.

G2.9 Every process and every concurrent statement should be preceded by a comment about the functionality of the code.

G2.10 Do not leave code lines commented. Any inactive or wrong code has to be deleted. If needed use older versions (for instance from CVS, SVN) for comparison.

G2.11 Keep the signal name's consistency through all the hierarchy of the design.

G2.12 Write just one statement per line of code, even when doing component instantiation.

G2.13 Preferably use lowercase letters for all the code.

G2.14 Regarding the vectors:

- Use descending (`downto`) if the vector represents a bus.
- Use ascending (`to`) for the first dimension of a memory array. For example, `array(0 to 5) of std_logic_vector(7 downto 0);`
- Use ascending (`to`) for items collections, such LEDs array.

G2.15 Use *signal attributes* instead of fixed numeric value. For instance:

~~`out_data <= (data(15) and in_data) & port_data(6 downto 0);`~~
`out_data <= (data'high and in_data) & port_data'range);`

G2.16 A line of code must not contain more than 80 characters, including valid code and comments. Use <enter> to split long lines and indent the next line to show continuity from the previous line.

G2.17 Do not use global signals and shared variables under any circumstances.

G2.18 Be careful using synthesis tool proprietary *pragmas.* If they are really needed, write a comment explaining the reason of using it.

G2.19 For synthesizable code do not use 'after' to symbolize delay: `sum<=a+b after 5 ns;` The 'after' must only be used in test bench o for modeling purpose.

G2.20 Use alias only when it clearly enhance readability without adding complex redirections.

G2.21 Check the code for latch generation. For long `if-then-else` or case statements assign values to signals by default to avoid missing any assignment.

G2.22 Do not use combinational feedback under any circumstances. Avoid something like: `a <= a and b;`

G2.23 Use an `if-generate` statement controlled by a generic value to enable or disable either debug or simulation only lines of code.

G2.24 To assure portability, avoid FPGA vendor primitive instantiations.

G2.25 FPGA vendor-specific components, such as DLLs, PLLs, etc. should be in its own VHDL component file.

G2.26 Remember that assert statements can be left in synthesizable VHDL code. The synthesis tool will completely ignore them.

G2.27 `for-loops` statements, like `if-then-else` statements, generates a priority encoded logic in the majority of the cases. Be careful when trying to use `for-loops` statements to generate parallel logic.

G2.28 The VHDL code through all the VHDL components should have a common look in order to enhance code familiarity between different components. Even in projects with different designers.

G2.29 Use only active high signals. Avoid active low signals.

G2.30 When possible write the RTL code to infer components such as memory blocks (RAM, ROM), multiplication blocks (DSP blocks), etc. Review the synthesizer manual to write

the code to properly infer the desired component.

G2.31 Tri-state or bi-direccional I/O should only be coded in the top-level entity. Avoid internal tri-states.

G2.32 Do not utilize glue-logic in the top-level entity. The only two kinds of discreet logic allowed in the top-level entity are the tristate and bidirectional related logic.

G2.33 Do register all the inputs and outputs of each VHDL component. Avoid combinational outputs. In case of needing a combinational output, explicitly detail its use in the code.

G2.34 Preferably do not declare *user-defined types*, except for the type that defines the states of an FSM. If necessary, declare *subtypes.* Subtypes keep the compatibility with the original *type* inheriting all the functions of the parent type.

G2.35 For synthesizable code do not initialize a signal when declaring it.

G2.36 For synthesizable code do not initialize a variable when declaring it in a process.

G2.37 Avoid using specific values in the RTL code. For instance do not use `if data="1010"`; instead define a constant with the value "1010".

G2.38 Use parenthesis in an expression with several logic operators.

G2.39 Keep the mathematical related operations in the same component, making easy sharing the resources.

Naming Convention

G2.40 Use the suffix *_i* to name a local signal that is an internal representation of an output port. Thus, an internal signal that will eventually be used as an output port will be easily identified. For instance: declare *data_2adc_i* for the output *data_2adc*.

G2.41 Use the suffix *_async* for asynchronous signals: *<signal_name>_async;*

G2.42 Use the suffix *_sync* to name a signal that has been synchronized. This is optional, as all the asynchronous signals in the design must be synchronized.

G2.43 Use the suffix *_v* to name an object of class variable: *<signal_name>_v;*

G2.44 Use the suffix *_x* to name a signal that is a vector: *<signal_name>_x;*

G2.45 Use the suffix *_pp0, _pp1,* and so forth, to name the pipelined outputs of the signal *<signal_name>* through the different pipeline stages: *<signal_name>_pp0, <signal_name>_pp1;*

G2.46 Use the suffix _q0, _q1, and so forth, to name the input to the pipeline of the signal *<signal_name>*: *< signal_name >_q0, < signal_name >_q1;*

G2.47 Use the sufix *_ce<n>* to name signals controlled by a clock enable signal and will be part of a clock enabled path (i.e., multi-cycle path). *<n>* is the number of disabled clock. Thus, for a signal controlled by a clock enable that enable the update of the signal every 4 clock cycles; the output signal should use the suffix *_ce4*.

G3.48 Do not use either the suffix *_in* or *_out* (*<signal_name>_in* or *<signal_name>_out*) in the signals defining the I/O ports or internal signals of a component. Its use is confused in assignment statements.

G2.49 Use the sufix *_cs* to name an FSM signal associated to the FSM current state.

G2.50 Use the sufix *_ns* to name an FSM signal associated to the FSM next state.

G2.51 Use a meaningful name when declaring signals interconnecting different modules. A standard naming structure is: *<origin>_<destination>_<signalport_name>_<suffix>.* For instance use *fsm_lcd_data_x* for a data bus from the FSM module to the LCD module. Other example use *ctrl_we*, for a write enable signal originated in the control module that goes to multiple units; thus, there is not a specific destination detailed.

G2.52 Select a unique notation for active low signals to easily identify them. For instance use rst_n or RSTn, load_l or LOADl. But, don't mix styles.

G2.53 Use lower case and undescore (_) when defining the signal and variable names for improving readability. Example: read_enable. Uppercases also facilitates the meaning of the signal's name. For instance: ReadEnable. However, avoid mixing uppercase/lowercase-underscore.

G2.54 Identify the system clock with a signal name that helps to recognize its source. For instance *sys_clk_<freq>.*

G2.55 Identify other clocks different from the system clock with a name like *clk_<freq>_<suffix>*, where suffix is optional. For instance, in case an FPGA internal clock uses a dedicated global distribution resource, use the suffix *_g*.

G2.56 Use meaningful and conventional names for the architecture. In the appropriated context use "rtl", "behavioral", "structural", "test_bench".

G2.57 Identify the different type of resets by naming them something like *reset* for synchronous reset and *areset* for asynchronous reset.

G2.58 Avoid using 1, l (character 'l'), L, I ('i' uppercase), O ('o') y 0 (zero) in names or situations that could create ambiguity.

G2.59 Avoid too short names for signals and variables. Do not use something like 'p', 'n', 'in', etc. Except for the *for-loop* and *for-generate* statements.

3. Entity

G3.1 Write an I/O port declaration per line of code, along with a comment about the port.

G3.2 In the entity declaration group the I/O ports by function not by direction. In each functional group place first the inputs, then the outputs and finally bidirectional. Keep the same grouping when instantiating the component.

G3.3 Do not use *buffer* mode in any of the entities. Use a 'dummy' internal signal if there is a need to read an output.

G3.4 For the I/O ports of an entity used exclusively types such as *std_logic, std_logic_vector or std_ulogic, std_ulogic_vector.*

G3.5 Use *inout* mode ports only in the higher hierarchy entity (i.e., top-level).

G3.6 Write a generic declaration per line of code, followed by a comment about the reason of the generic and its assigned value.

G3.7 Unconstrained arrays in the entity are an elegant code style, but they can cause problems in unitary synthesis. So, avoid it for synthesis proposes.

G3.8 When using synthesis attributes detail the reason of using them and why they should be in the entity part.

4. Architecture

G4.1 When declaring internal signals in the declarative part of the architecture group them according to their functionality. Write comments detailing the signal(s) o signal group(s) function(s).

G4.1 When declaring constants in the declarative part of the architecture details the reason of the particular constant value.

G4.2 Use the naming convention detailed above when declaring signals and variables.

G4.3 When declaring an object, signal or variable class, as integer type; be sure to restrict the range of values to those values that will be used.

Component Instantiation

G4.4 In a component instantiation statement use as instance name a name related to the instantiated component: *u<n>_<component_name> <component_name>.* For instance: *u<n>_count_16 count_16* (where count_16 is the instantiated component). The letter '*n*', represents a number and is optional. However, it must be used when the same component is instantiated more than once.

G4.5 Use '*named association port map*'. Do not use '*position association*'. The same rule applies for '*generic map*'.

G4.6 Use separated lines per each port map association.

G4.7 Assign 'open' to unconnected output signals.

G4.8 Connect the unconnected input signals to the right value to get the desired functionality.

G4.9 For multiple instantiations of the same component do use `for-generate` statement.

Process

G4.10 Use a descriptive name to name each process in a component. Use the suffix *_proc* at the end of the process' name: <process_name>_proc. Use the same name to end the process.

G4.11 Each process must have at least one comment, like a header comment, describing its functionality. But, add as many comments as needed through the process' code lines.

G4.12 Preferably write combinational code in a sequential process.

G4.13 In a combinational process, all the signals that are read within the process have to be in the sensitivity list. In case of using VHDL 2008, it is possible to write: *process(all).*

G4.14 In a combinational process, never assign to a signal and read from the same signal in the same process. This will avoid infinite loops.

G4.15 In a sequential process only the reset and clock signals have to be in the sensitivity list. Being reset an asynchronous one, otherwise just the clock signal.

G4.16 In a sequential process use the reset signal to initialize the respective signals and variables (regardless if the reset is asynchronous or synchronous). Only constant values can be assigned during reset. Avoid asynchronous load.

G4.17 For synthesizable code do not use *wait* statements in a process.

G4.18 Do do not read a variable before assign a value to it.

G4.19 Use *rising_edge* or *falling_edge* for clock edges. Get ride off *(clk'event and clk='1')*.

G4.20 Preferably use `case` statements if priority-encoding structure is not required.

G4.21 Use `if-then-else` statement if priority-encoding structure is required.

G4.22 When using *integer* or *bolean types* be careful on providing a correct and safe initialization (reset) of them. By default, they do have a value other then 'X' or 'U' when they are declared.

G5.25 Do not use the VHDL operators rotate and shift. Instead, use concatenates and slices.

G4.26 Avoid slicing signals. Better, when possible slice variables.

G4.27 Preferably use `for-loop` statement to avoid repetitive code.

G4.28 When solving a problem with an elaborated solution, use a detailed comment before the respective statements, to describe the solution.

Functions and Procedures

G4.29 Do initialize variables declared in a *function* or in a *procedure,* even if the code is for synthesis.

G4.30 Preferably use *functions* in RTL code. Use *procedures* in test bench.

G4.31 When writing *functions* or *procedures* use unconstrained vectors. Avoid specific I/O sizes.

G4.32 Preferably use packages for the *functions* o *procedures* declarations and body; unless just one *function*/*procedure* in the whole project is used, in such a case declare it locally in the declarative part of the architecture.

Clocks and Timing

G4.33 Make sure the high frequency logic is in the same VHDL component or different components, but clocked with the same high frequency clock.

G4.34 Write a specific VHDL component for the logic related to the clock signal generation. In case of having different clock signals, write different components for each clock; unless they are related, multiple or sub-multiple, in such a case use just one component.

G4.35 Keep the critical path logic in separate module from the non-critical path logic. Thus, the synthesis tool can optimize the critical path logic for speed, whereas optimizing the non-critical path logic for area.

G4.36 When writing the code lines insert specific comments on cases such as *false_paths, input setup time or clock to output time.* Hence, facilitating the writing of the respective timing constraint file.

G4.37 Use only one clock domain per VHDL component. Exceptions: the top level entity and when it is necessary to synchronize signals or in clock crossing component(s).

G4.38 Do not generate a clock signal using gated-clock, counter output or something similar. Instead do use clock enable.

G4.39 Only one clock edge has to clock all the sequential logic in the whole design. Either rising edge or falling edge, but just one of them. Exception: when working with DDR.

G4.40 When, for some reason, the design does have to deal with both edges of the clock, keep the positive edge logic and the negative edge logic in separated VHDL components.

G4.41 For the case that the setup time of an input signal be critical or very tight, use the flip-flop in the I/O block the capture the input signal.

G4.42 For the case that the clock-to-output time of an output signal be critical or very tight, use the flip-flop in the I/O block to register the output signal.

G4.43 When possible draw a timing diagram as comment of the related coded logic. It facilitates the understanding of the RTL code.

G4.44 No combinatorial path might pass through more than one VHDL component. This path must not be defined as “multi-cycle path” or “false path” by either the synthesis tool or the designer.

G4.45 For high frequency designs generate an asynchronous reset assertion and a synchronous reset des-assertion.

Synchronization

G4.46 Every asynchronous input has to be synchronized. Do not insert any kind of logic between the input pad and the synchronization flip-flop. Preferably, use two flip-flop for synchronization purpose.

G4.47 To synchronize high frequency bus signals, do not use double flip-flop, not all the bits of the bus arrives exactly at the same time. Implement a single bit synchronized enable signal, handshaking mechanism or a FIFO to synchronize the bus.

G4.48 All the signals generated in a clock domain different to the one to be used, has to be consider asynchronous. Therefore, has to be synchronized before using it in the new clock domain.

G4.49 In multi-clock designs, ensure that signals crossing clock domains are properly synchronized. Typically, this involves using a synchronizer, a handshake mechanism, or a FIFO.

FSM

G4.50 Before coding the FSM, read through full the specifications; then draw either the State Diagram or the Algorithmic State Machine (ASM).

G4.51 Declare an enumerated type that holds each state of the FSM. Use meaningful name for each state.

G4.52 Declare the signals for the next state and current state of the FSM as signal of the enumerated type defined previously.

G4.53 The FSM code must have a synchronous or asynchronous reset, to be correctly inferred.

G4.54 When describing a Finite State Machine (FSM) use two processes: one combinational process for the next state logic and the other, a sequential process to describe the current state logic.

G4.55 Per each FSM output use a sequential process in which the output will be activated in the respective current state(s) on the active edge of the clock.

G4.56 Use `case` statement(s) in the next state combinational process. In case of being necessary `if-then-else` statement(s) can be nested within the *case* statement.

G4.57 When synthesizing FSMs, check the synthesis report file to verify the code assignment done by the synthesis tool to each state of the FSM.

G4.58 In case of using a synthesis attributes to override the default state encoding of the synthesis tool, read the log report to check whether the synthesizer has really used the suggested coding style.

G4.59 Every signal to be used in an FSM must be generated in the same clock domain. Hence, any signal coming from a different clock domain is considered asynchronous signals and needs to be synchronized before using it.

5. Synthesis

G5.1 Always read through all the warnings generated by the synthesis tool. Get ride off the ones that can be solved.

G5.2 In a system with several components, each component has to be synthesized separately.

G5.4 When possible avoid using synthesis attributes. When needed, describe in a comment the reason for using it.

G5.5 Utilize the RTL viewer to have a glace of the synthesized logic.

G5.6 When possible separate the logic with different optimization criteria in different components.

G5.7 In large designs verify the fan-out set by default for the synthesis tool. Remember to use the register duplication in cases when the fan-out value is too large.

G5.8 Review the optimization criteria that by default use the synthesizer. Check whether it matches with the project's need.

G5.9 For designs with aggressive timing goals, set the synthesizer's clock period to the target frequency to have a broad idea of the logic delay; before spending (wasting) long runs with the place and route tool. Then, check the timing results in the respective log file.

6. Test Bench

G6.1 Each component of the project should have at least one associated test bench.

G6.3 Per each VHDL component do a functional simulation before synthesizing it.

G6.2 Each project has to have different test bench testing thoroughly the system level functionality.

G6.4 Always run post P&R simulations. Even though they are very time consuming task, it will save future headaches.

G6.5 Use *procedures* to simulate the functionality of buses like AMBA Bus or Wishbone Bus.

G6.6 For large designs define a package to hold the functions, procedures, constants, types and sub-types related only to the test benches.

G6.7 Verify that the design behave as expected when the input stimulus behave as not expected.

G6.8 Use all the power of the simulation tool to facilitate the view of the waveforms. Use dividers, different color waveforms, etc.

G6.9 Use functions, procedures o simply asserts to display in the simulator transcript window the execution and outcome of the test bench.

G6.10 When testing an FSMs, verify the behave when it is forced to reach no specified states.

G6.11 Do utilize as many variables as possible. They reduce the simulation time.

G6.12 Define the simulation time and stop the simulator when that time is reached (use *assert*).

G6.13 Do use as many asserts as possible to make the test bench auto-checkeable.

G6.14 Do use `wait` statements in processes. Thus, not sensitivity list at all.

G6.15 Preferably avoid using non-portable run-time simulator commands, such as ‘force’, ‘unforce’.

G6.16 Do not use clock signals as data signals and vice versa.

G6.17 Use separate processes for: data generation, clock generation and reset generation.

G6.18 Preferably use an array of data stimulus along with a `for-loop` to go through all the values to test combinatorial logic.

G6.19 Preferably the test bench should be written by a different designer than the one who created the module under test.