Adiuvo FPGA Coding Standards

Version: 1.0

Author: A P Taylor

Adiuvo Engineering

1. Contents

1. Contents 1
2. Change Log 2
1. General Requirements 3
2. Naming Conventions 3
3. Architecture Naming 4
4. Configuration Naming 4
5. Package Naming 4
6. Component Naming 4
7. Instance Naming 5
8. Signal Naming 5
9. Port and Generic Maps 5
10. Formatting Conventions 5
11. Vector Conventions 6
12. Item numbering 6
13. Comments 6
14. Model Interfaces 7
15. TYPES 7
16. Assertion Reports 8
17. File Input and Output 8
18. Prohibited Language Constructs 8
19. Communication between models, or parts of a model, using file I/O 8
20. Non-commutative resolution functions 9
21. Non-associative resolution functions 9
22. Shared variables 9
23. Coding Guidelines 10

2. Change Log

Version	Notes
1.0	Extracted from Confluence on 31/3/2023. For reference only not to be used for development work.

1. General Requirements

- All synthesisable models shall be written in accordance with IEEE-1076-2008 Standard VHDL.
- Test structures may leverage IEEE-1076-2008 Standard VHDL
- All identifiers, comments, messages, filenames etc. shall be based on the English language.
- VHDL reserved words shall appear in all uppercase characters. This includes identifiers defined in package STD.STANDARD
- All other identifiers whether user defined or derived from packages (e.g. IEEE.std_logic_1164) shall be in all lowercase.
- For RTL there shall be a maximum of one statement per line; this facilitates code coverage analysis. It is strongly recommended that this also applies to non RTL code.
- Lines shall not exceed 120 characters; it is recommended that lines are restricted to 80 characters.
- Indenting of code shall be used to clarify the code structure.
- Indents shall consist of two spaces, TABS shall not be used.
- Line spacing shall be used to clarify the code, e.g. grouping related statements together.
- All instances and processes shall be labelled with a short descriptive name. Note that process naming simplifies the display of variables during simulation. Instance names should be short as their hierarchical concatenation is used in netlist signal naming.
- It is recommended that loops, concurrent signal assignments, IF and CASE statements are labelled where this improves readability.
- Each component declaration shall be a copy of the component's equivalent entity declaration, i.e. it shall have the same port names, types, modes and ordering as the entity.
- Test Benches shall be self-checking.

2. Naming Conventions

The basic entity name can be any meaningful name up to 28 characters.

For the special case of a test bench entity, the meaningful name shall have _tb appended

The filename for this entity shall be the basic entity name followed by _entity, and appended with .vhd

3. Architecture Naming

The architecture name shall contain an architecture type:

struct For purely structural code

rtl For Register Transfer Level code, intended for synthesis

behav For Behavioural code, (including non-synthesisable RTL code)

bench For Test bench architectures

4. Configuration Naming

The number of characters in the configuration name should be minimised where possible. Configuration

names shall contain the top-level entity and architecture name prefixed by **cfg_.**

cfg _<entity>_<architecture>_<unique_identifier>

5. Package Naming

The basic package name shall contain up to 28 characters

The basic package body name shall be the same as the package name.

The package body shall be kept in the same file as the package.

6. Component Naming

If the use of default bindings is not desired, the following should be implemented, otherwise this section can be ignored.

Component declaration names shall be different from the entities to which they refer. Use the '**comp_**' prefix to ensure that no default bindings exist.

In the Modelsim tool, the modelsim.ini option "RequireConfigForAllDefaultBinding" may also be used to turn off default bindings for simulation, which will still occur for synthesis.

7. Instance Naming

It is recommended that identifiers for instances do not exceed 10 characters. This is because instance

names are often concatenated to form long pathnames in downstream tools. Long pathnames may cause problems further along the process flow.

8. Signal Naming

Identifiers for signals and variables shall not exceed 20 characters and can be any meaningful name.

Input signals shall prefixed with i_

Output signals shall be prefixed with o_

Internal Signals shall be prefixed with s_

Internal Variables shall be prefixed with v_

Generics shall be prefixed with g_

Constants shall be prefixed with c_

Types shall be prefixed with t_

9. Port and Generic Maps

Port maps shall use named association. All ports shall be explicitly listed, using 'OPEN' if necessary.

Generic maps shall use named association. For generics, only those values that are having their defaults overridden need be listed in the generic map.

10. Formatting Conventions

Within the declaration of an entity, component, instance, function or procedure, each of the ports, generics or signals shall be on a separate line. Each entry in a port list or generic list shall have its type shown individually.

Comma separated lists of signal names associated with one type shall not be used.

11. Vector Conventions

Vectors shall be treated consistently across their complete range. For example,

If ANY bits in a vector are to be reset, then ALL bits shall be reset.

If ANY bits in a vector are to be registered, then ALL bits shall be registered.

The index ordering (i.e. using TO or DOWNTO) of the model top-level entity port clause signals shall be identical to the one used in the datasheet or similar documentation. It is recommended to use the same index ordering in the whole model, but in case the index order is reversed within the model, this shall be clearly marked every time the index order is different w.r.t. the corresponding signal at the highest level of the hierarchy.

12. Item numbering

Items shall be numbered from 0 rather than 1 where possible as this is more natural in a digital system.

13. Comments

All VHDL source files shall contain comments to at least the following extent:

- Description of the purpose of the design unit - usually in the header.
- Description of the function of all sub-programs immediately prior to the sub-program definition.
- Description of the function of each process, immediately prior to the start of the process.

No uncommented VHDL source files shall be allowed to be issued. Temporarily commented out code shall be removed prior to issue.

Long comments shall occupy one or more lines, indented to the same level as the code they relate to. Short comments may be appended to individual lines of the code.

Comments shall be meaningful and reasonably self-contained, although they may reference other documentation or the header description. A comment for each entry in a port-list or generic-list is recommended.

14. Model Interfaces

Interfaces at the top level of an FPGA shall use the following types:

std_logic	**IEEE.std_logic_1164**
std_logic_vector	**IEEE.std_logic_1164**
std_ulogic	**IEEE.std_logic_1164**
std_ulogic_vector	**IEEE.std_logic_1164**
signed	**IEEE.numeric_std**
unsigned	**IEEE.numeric_std**

15. TYPES

Arrays intended to represent bussed signals shall always be defined as (N-1 DOWNTO 0) for an N-bit bus, where (N-1) is the most significant bit.

Enumeration types can be used for state machine state variables. For example:

TYPE t_state IS (idle, read, write, hop, skip, jump, seu1, seu2);

SIGNAL state : t_state;

If INTEGER types are to be used, for simulation efficiency reasons, always use a constrained subtype of

INTEGER. For example:

SUBTYPE t_byte IS INTEGER RANGE 0 TO 255 ;

For synthesisable RTL it is very important to constrain the range to only that required by the application.

More complex data types, such as records, shall only be used if they clearly improve readability and understanding of the code. They can also be used for compatibility with the requirement specification, but note that some synthesis tools do not support records.

Unresolved types (i.e. std_ulogic / std_ulogic_vector) are to be preferred over resolved types (std_logic /

std_logic_vector) where a multi-drive capability is not required (i.e. everywhere except top-level bidirectional pins). This is because it reduces logical errors with multiple drivers into

compilation / load errors, rather than simulation time ‘X’ generation, hence speeding up the time to discovery / resolution.

Numerical types (e.g. signed / unsigned) should be used wherever data contains a numerical value (as opposed to a bit-field). This allows arithmetic functions to be used.

Type conversions (casting) should normally be done at the port map, e.g.

PORT MAP (

input_data <= std_ulogic_vector(input_data),

signed(output_data) <= output_data);

Note that single bit std_ulogic / std_logic require no conversion (although it can be included).

16. Assertion Reports

Assertion messages shall only be used to report error conditions, unexpected simulation conditions, or for debugging. They shall not be used for general simulation output.

17. File Input and Output

All file I/O shall be done using the file type STD.TEXTIO.TEXT. No other file type shall be used. Nb This allows use of both STD.TEXTIO and STD_LOGIC_TEXTIO packages.

Simulation behaviour shall never be dependent upon the results of file output from the same simulation i.e. models are not allowed to communicate using file I/O.

18. Prohibited Language Constructs

In order to maintain deterministic behaviour across different simulators, and to maintain coherency between simulation and synthesised models, the following language features shall not be used.

19. Communication between models, using file I/O

This technique will not work consistently on different simulators or under different operating systems. The use of file output buffering in operating systems means that the output file may be completely written only at the end of the simulation. It also adds inputs and outputs to the model that are not on the entity declaration, leading to difficult to understand and un-deterministic code.

20. Non-commutative resolution functions

The VHDL Language standard does not define parameter ordering when implicitly calling resolution functions. Consequently different simulators may associate the parameters differently. This does not matter if the resolution function is commutative.

i.e. where f(a,b) = f(b,a). If the function is non-commutative the two simulators may give different results.

21. Non-associative resolution functions

Where there are more than two drivers for a resolved signal, the resolution function must also be associative. i.e. where f(a,f{b,c}) = f(b,f{a,c}) = f(c,f{a,b})

22. Shared variables

The use of shared variables is non-deterministic in all cases except for trivial situations.

23. Coding Guidelines

Guideline – Keep the clocking and reset schemes simple.

A simple clocking scheme is easier to understand, analyse and maintain. The preferred scheme uses a single global clock with positive edge-triggered flops as the only sequential devices.

Guideline – Avoid using both clock edges in the design.

This makes the clock duty cycle non-critical and simplifies scan based testing.

Guideline – Incorporate clock control and reset logic into a separate module.

All other blocks can then use standard timing analysis and scan insertion, and RTL coding exceptions will be restricted to a small module for careful review.

Guideline – Avoid hand instantiating clock buffers in the RTL code. For RTL treat clock nets as ideal nets, with no delays.

Clock resources in FPGAs or clock trees in ASICs will be technology specific. If blocks are written assuming idealised versions of the clock exist, it improves their portability.

Guideline – Avoid dynamically gated clocks for retargetable designs.

Clock gating circuits may be technology specific and timing dependent. If an exception is made for power saving, any clock gating that is implemented should be maintained in the separate clocking module and not within code intended for reuse.

Guideline – Reset polarity

The polarity of the reset signal used in RTL code should be chosen to match the reset polarity on the target library DFF cells.

Guideline – A global asynchronous reset of all registers is recommended. The reset should be released in a controlled manner to ensure safe reset recovery. This will usually entail registering the external reset input.

Guideline – When crossing asynchronous clock domains use two flip-flop stages to transfer single bits. For multi-bit synchronisers do not use multiple copies of a single bit synchroniser, use a multi-bit coding scheme such as gray code, or a reliable handshaking scheme instead.

Guideline – When crossing related, nominally phase aligned, clock domains both inputs and outputs should be registered to simplify timing requirements.

When the added clock cycle delay that this imposes cannot be tolerated, a safe synchronisation point can be defined at which logic in the faster clock domain can read from,

or write to, the slower domain. An enabling signal will be defined in the fast clock domain, with a repeat period at the slow clock frequency. This signal should be generated within the clock control module. The timing constraints that this imposes on logic in the slow clock domain shall be clearly documented.

Guideline – Infer registers. Registers are the preferred storage element for sequential design.

Guideline – Do not use latches in the design. Use of latches will complicate static timing analysis and test.

Guideline – Register sub-block outputs.

This provides numerous advantages for synthesis:

1. Downstream blocks have predictable input delays and drive strengths
2. No optimisation is needed across hierarchical boundaries.
3. It forces the designer to keep like-logic together.
4. Simplifies preservation of the hierarchy.
5. Simplifies constraints for bottom-up compilation.
6. Allows recompilation of only those levels that have changed.
7. Simplifies hierarchical floor planning.

Top-level outputs should be registered, but will probably be registered by default if this guideline is followed.

Guideline – Register the top-level inputs.

This decreases the input to clock delays; therefore, it increases the chip-to-chip frequency.

Registering inputs is also recommended for large blocks and code intended for reuse, so that these can be treated as standalone from a timing point of view.

Guideline – Avoid combinational feedback.

Use of combinational feedback can create difficulties for test and static timing analysis.

Rule – Specify complete sensitivity lists.

For a combinational process this must include every signal that is read by the process, otherwise unintended operation will result. For a sequential process it should only include the clock and an asynchronous reset (if used). Including any unnecessary signals on the sensitivity lists slows down the simulation.

Guideline –*Case* vs. *If* statements.

Do not use extremely long nested IF constructs where the built-in priority encoding adds unnecessary design complexity and potentially extends delay paths. In general, use the Case

statement for complex decoding and use the priority encoding of an IF statement for speed critical paths.

Rule – Do not use delay constants in RTL code.

Synthesised and RTL code may behave differently if this is not adhered to. A delay time may sometimes be thought necessary, e.g. mixed RTL and gate-level simulation, or RTL simulation with multiple and/or generated clock signals. In these situations, the clock timing external to the RTL block, or interface timing to/from the code may be adjusted. This will model what happens in the final chip environment.

Guideline – For synthesisable code the use of signals instead of variables is recommended to ensure that the behaviour of pre and post synthesis designs match.

If significant speed improvements can be expected by using variables, or code readability is improved, then their use may be justified. However, if used they should be used with care.

Rule – the primary intent of a variable is to imply a combinatorial signal. When a given variable is used, it shall imply either a combinatorial signal (preferred) or a multiplexed/switched signal (not preferred), but never both.

Guideline – Avoid asynchronous logic.

Asynchronous designs are more difficult to design, to verify and to characterise. Timing and hence functionality are technology dependent, limiting portability.

Guideline – Try to avoid timing exceptions such as multi-cycle and false paths.

Analysis is prone to human error, and exceptions need to be marked for each downstream design tool.

Guideline - Do not assign fixed logic values directly to port maps of instantiated components. Constants or signals with defined values may be used.

Some synthesis tools may reject code written in this way.

Guideline – Do not use glue logic at the top level of the design.

This allows stitching together of lower level blocks without a compile.

Guideline – Do not use local redefinitions of the std_logic resolution function.

Synthesis and reuse will be strongly compromised.

Guideline – Use multiplexed based buses in place of internal tristate buses.

Adhering to this guideline removes the need to ensure that only one device is driving the bus at any time

(can be difficult to achieve during power-up), avoids having to prevent the bus from floating and aids portability with limited or non-existent tristate buffer resources.

Guideline – Do not use types *bit* or *bit_vector*.

Not all simulators provide arithmetic functions for these types.

Guideline – Use constants in place of hard coded numeric values.

An exception can be made for values '0', '1', TRUE and FALSE. Constants and parameter definitions should

be kept in packages.

Guideline – Synthesis directives should not be embedded within the code.

Future users may not be aware of hidden commands, and may have different synthesis goals. Exceptions

may be required, particularly for directives to turn synthesis on or off. Where this is done the use of --

pragma translate_on and --*pragma translate_off* are recommended as these directives are commonly recognised by different synthesis tools.

Guideline – Use enumerated types for state naming in FSMs.

This allows the synthesis tool to choose the most appropriate encoding scheme for the architecture it is being mapped to. Using state names also improves readability.

Guidelines for use of cores

Instantiation of vendor supplied macros (e.g. RAMs) are inflexible, and non-portable. Therefore, wherever possible, the use of inference is preferred. Where instantiation is necessary (e.g. DCMs, buffers), put them into a top level vendor specific block.

=