|  |  |
| --- | --- |
| Title: | VHDL Coding Styles |

CHANGE HISTORY:

|  |  |  |
| --- | --- | --- |
| Revision | Description of Change | Effective Date |
| A | Initial release. | Initial Release |
|  |  |  |

[1. Introduction 4](#_Toc21517735)

[1.1.1 Purpose 4](#_Toc21517736)

[1.1.2 Scope 4](#_Toc21517737)

[2. VHDL cODING Guidelines 5](#_Toc21517738)

[2.1 Language version 5](#_Toc21517739)

[2.2 Naming 5](#_Toc21517740)

[2.2.1 Signal Names 5](#_Toc21517741)

[2.2.2 Port Name 6](#_Toc21517742)

[2.2.3 Variables 6](#_Toc21517743)

[2.2.4 Type and Record Type Names 6](#_Toc21517744)

[2.2.5 Constants and Enumerated Types 7](#_Toc21517745)

[2.2.6 Types, Constants and Packages 7](#_Toc21517746)

[2.2.7 Name Declaration Comments 8](#_Toc21517747)

[2.2.8 Process Names 8](#_Toc21517748)

[2.2.9 Architecture Names 8](#_Toc21517749)

[2.2.10 Packages 9](#_Toc21517750)

[2.2.11 Components & Instances 9](#_Toc21517751)

[2.2.12 Generated Logic 9](#_Toc21517752)

[2.2.13 State Machines 9](#_Toc21517753)

[2.2.14 Summary of Naming Prefixes 10](#_Toc21517754)

[2.2.15 Summary of Naming Suffixes 10](#_Toc21517755)

[2.3 Coding styles 11](#_Toc21517756)

[2.3.1 Whitespace 11](#_Toc21517757)

[2.3.2 Line Length 11](#_Toc21517758)

[2.3.3 Comments in General 11](#_Toc21517759)

[2.3.4 Comments on Port Signals 11](#_Toc21517760)

[2.3.5 Resets 12](#_Toc21517761)

[2.3.6 Signal Definition 12](#_Toc21517762)

[2.3.7 Active Low Signals 12](#_Toc21517763)

[2.3.8 State machines 12](#_Toc21517764)

[2.3.9 Port Record Types 13](#_Toc21517765)

[2.3.10 Process Sensitivity Lists 13](#_Toc21517766)

[2.3.11 Common Functions Like Rounding and Saturation 13](#_Toc21517767)

[2.3.12 Common Error & Status Strategy 14](#_Toc21517768)

[2.3.13 Test Features and Overrides 14](#_Toc21517769)

[2.3.14 Stale or Commented Out Code 14](#_Toc21517770)

[2.3.15 Signals in an Unknown State 14](#_Toc21517771)

[2.3.16 Use Arrays and Loops 14](#_Toc21517772)

[2.3.17 Generate Statements 15](#_Toc21517773)

[2.3.18 File Naming 15](#_Toc21517774)

[2.3.19 No Internal Tri-states 15](#_Toc21517775)

[2.3.20 Multiple Clocks 15](#_Toc21517776)

[2.3.21 RAM Enables 15](#_Toc21517777)

[2.3.22 Use of Libraries 15](#_Toc21517778)

[2.3.23 Use of Clock Edge Functions for Clocked Processes 16](#_Toc21517779)

[2.3.24 Reasonable Limits on Process, Function and File Sizes 16](#_Toc21517780)

[2.3.25 End of Lines 16](#_Toc21517781)

[2.3.26 Hold vs. Latch 16](#_Toc21517782)

[2.4 Coding Examples 17](#_Toc21517783)

[2.4.1 CPU Registers – Read/Write Control 17](#_Toc21517784)

[2.4.2 Use ‘range for Signal Declarations 18](#_Toc21517785)

[2.4.3 Use IEEE Standard Functions Where Possible 18](#_Toc21517786)

[2.4.4 For Arithmetic Functions, Use ieee.numeric\_std Package 18](#_Toc21517787)

[2.4.5 Do not Use std\_logic\_arith, std\_logic\_unsigned or std\_logic\_signed 18](#_Toc21517788)

[2.4.6 Use resize() to Extend Signals 19](#_Toc21517789)

[2.4.7 Use \*\_reduce() Functions in IEEE Packages for std\_logic\_vectors 19](#_Toc21517790)

[2.5 Productivity Tools 19](#_Toc21517791)

[2.6 Find X’s in Simulation - TODO 19](#_Toc21517792)

[3. IEEE Package Functions 20](#_Toc21517793)

[4. Appendix 23](#_Toc21517794)

[4.1 Abbreviations 23](#_Toc21517795)

# Introduction

### Purpose

The purpose of the following guidelines is to ensure that code is developed in a way that is easy to maintain and understand by different engineers.

### Scope

This document is intended to be used by engineers that are developing FPGA and ASIC code in VHDL although the general principles also apply to writing Verilog code (if you enjoy spending time debugging instead of letting the compiler do it for you. 😊.)

# VHDL CODING Guidelines

## Language version

Write code that complies with VHDL-2008. [Or whatever version supersedes it]

## Naming

**Naming is very important**. A good name helps the reader understand your code with less effort. We all make assumptions when we see a name. If our assumptions are correct, then we can read code more smoothly. If your assumptions are incorrect then when we discover that fact, we must pause and mentally adjust. Try to build names so that the most important information comes first.

Example: The data output bus from a **FIFO** in the **bs** block should be named ‘bs\_fifo\_dout’ not ‘dout\_bs\_fifo’ or ‘dout\_fifo\_bs’. Think of the name as having a hierarchy of its own from most important to least important, or most general to most specific.

 A name should indicate the purpose or function of the object and not its type.

Example: an 8-bit loadable counter used for addressing should be called cnt\_address (its function) rather than ‘cnt\_load8’ (its type).

All names shall be lower case except for constants, generics and enumerations, which shall be upper case.

### Signal Names

Use all lowercase letters and underscores "\_" to improve legibility.

Use \_i as the ending for internal nets in a block that also drive an output port. Do not use \_i for interrupts or integers. Use \_intr or \_interrupt at the end for an interrupt.

Use \_n for active low signals. (…but never use them.)

Use \_d to add to the end of a signal which has been delayed by one clock. Use d1, d2, etc., for additional pipeline delays. But in general, use an array if there are more than 2 delays.

 All clock names should begin with **clk\_** if it is followed by an alpha numeric name. The underscore is optional for **clk** followed by a number (i.e. clk100). Don't hide ‘clk’ in the middle or at the end. Use clk\_in, clk\_pci, clk\_avi, clk\_1, clk\_2, clk\_192d, clk40mhz, etc., not pciclk, mclock, sclk, etc. If a signal is a clock, then that is its most important feature and you want to know that first. What it is clocking is the next most important feature. The clock frequency is the least important piece of information and can often change during the design process so avoid using it as a part of the name whenever possible.

Within a block with only one clock you can just use the name **clk**.

To identify signals in a block that are coming from another clock domain and need to be synchronized, you can use \_clk\_name suffix to indicate a signal that is in the clk\_name domain.

Use \*\_msync for identifying metastability synchronizers. Ideally, there would be a function in a common package that would implement the metastability synchronizers. This would ensure a consistent name style that would make it easy to search and find all synchronizers.

For differential signal pairs, assign the positive polarity signal without any suffix. Have the tool insert the negative polarity signal.

### Port Name

Use \_pad and \_core for I/O block names, rather than using \_in or \_out suffixes on port names just to indicate the ports direction. It’s ok to use in and \_out if they are describing function and not just port direction such as data\_in and data\_out since they distinguish between the data going in and the data going out.

Port names across the hierarchy do not have to match exactly, but there should be consistency in the name and the differences should make sense. In general, as a name goes up the hierarchy it needs to become more specific to identify which block it is generated from.

*Opinions vary on whether it is a good idea to try to keep net names and therefore port names the same across different levels of hierarchy. I don't think that it's necessary. If it adds to readability, then fine but does a top level block really need to know that the error port on a major block at the top level of a design is ‘ps\_fifo\_xyz\_overflow\_err’ or just that it is the ps block error output.(‘ps\_err’)*

### Variables

**Use v\_ prefix for variable names**.

Don’t use variables in a process unless it makes the code simpler and easier to understand. Variables can’t be seen in the simulator waveform, so they make debugging more difficult. With today’s optimized compilers having signals don’t make the compiler slow down appreciably.

### Type and Record Type Names

**Type definitions begin with t\_ and record type definitions begin with t\_rec\_.**

Type definitions of an array for a specific purpose can use the t\_arr\_ prefix, e.g:

constant MAX\_BURST\_LEN : integer := 32; -- 16-bit words

type t\_arr\_burst is array (0 to MAX\_BURST\_LEN-1) of std\_logic\_vector(31 downto 0);

Arrays of standard logic vectors can be defined for a more general use. These can be defined in a common package and might be used for different sorts of signals in more than one place. These can be defined using the format t\_arr\_slv<bits>, where <bits> is the vector size. For example:

type t\_arr\_slv32 is array (natural range <>) of std\_logic\_vector(31 downto 0);

type t\_arr\_slv16 is array (natural range <>) of std\_logic\_vector(15 downto 0);

signal my\_arr : t\_arr\_slv32(5 downto 0);

The naming is still general to detailed. When you see t\_arr\_slv32 you think - "It’s a type, it’s an array, it’s a sort of std\_logic\_vector and the words are 32-bit".

### Constants, Generics and Enumerated Types

**Constants, Generics and Enumerations should always be UPPER CASE**.

**It is recommended to always use the ‘G\_’ prefix for generics**.

**It is recommended to always use the ‘S\_’ prefix for enumerations that are state-machine state names**.

A ‘C\_’ prefix could be used for constant names but if we restrict upper-case usage to constants, generics and enumerated type values then the ‘C\_’ is not really required.

Constant names should start by telling you something about the constant e.g.

-- Block select bits

constant BSEL\_VID : integer := 0;

constant BSEL\_SDCARD : integer := 1;

constant BSEL\_INTERRUPT : integer := 2;

-------------------------------------------------------------------

-- Register addresses

-------------------------------------------------------------------

constant ADR\_REG\_CTRL : std\_logic\_vector(15 downto 0):= 0x1000; -- address of control reg

constant ADR\_REG\_STATUS : std\_logic\_vector(15 downto 0):= 0x1800; -- address of status reg

constant ADR\_REG\_TIMER0 : std\_logic\_vector(15 downto 0):= 0x2000; -- address of...

constant ADR\_DRV\_TIMER1 : std\_logic\_vector(15 downto 0):= 0x2800; -- address of...

Enumerated types

type t\_state\_sys is (S\_SYS\_START, S\_SYS\_WAIT, S\_SYS\_SHUTDOWN, S\_SYS\_SHUT\_DONE);

There should not be hardcoded numbers ('magic numbers') embedded in the code. Instead use constants instead of putting in the code. All zero and all one values are the only exceptions.

Instead of this:

reg\_avo\_length\_pframe <= "1001110000";

Do this: define a constant at the top of the file then assign the constant in the original line.

constant FRAME\_LENGTH\_MAX : std\_logic\_vector( 9 downto 0) := conv\_std\_logic\_vector(624, 10);

reg\_avo\_length\_pframe <= FRAME\_LENGTH\_MAX;

To avoid magic numbers in the constant declaration, define the constant as an integer and set the width at the time of assignment.

constant RD\_FIFO\_ALMOST\_EMPTY : integer := 8;

if (fifo\_level <= to\_unsigned(RD\_FIFO\_ALMOST\_EMPTY, fifo\_level’length)) then

### Types, Constants and Packages

Types and constants that are only used within one file should be declared in that file (not in a separate package).

Types and constants that are used by multiple files but are confined to a major block (whose files are in one directory or sub-directories) should be put in a package file in that block’s directory.

Types and constants that are used by multiple blocks that span different directories should be put in a package that resides in a separate top-level directory (i.e. packages).

### Name Declaration Comments

Every port, signal, constant, type, etc. (except obvious ones like clock and reset) should have a comment describing the purpose or function of that object.

### Process Names

Use prefix pr\_ for process names. It is helpful if the rest of the process name refers to the output signal of the process, if there is just one, or to the overall function of the process. Spend some time picking names. This will pay off when looking at simulator or synthesis tool output transcripts when debugging.

If the clock frequency is important, do pr\_clk\_rate\_function. For example, pr\_clk\_48\_gen.

If a process is long, it helps readability to add the process name to the end statement. For example,

end process pr\_clk\_48\_gen;

### Architecture Names

**Use rtl as the architecture name for synthesizable code**.

*Do not use* design*. The name is trying to tell the reader something about the design style. Normally use* ***‘rtl’*** *(register transfer language) since this is the commonest coding style. Use* ***‘behave’*** *or* ***‘behavioral’*** *if that really is the style of code. You can use* ***‘struct’*** *or* ***‘structural’*** *for an architecture if the contents simply connect block instances together with no additional logic. If you have different architectures of RTL code, then use names like ‘rtl\_ram4k’ and ‘rtl\_ram8k’*

If the code uses vendor-specific macrocells or attributes, then you can use the architecture rtl\_vendor e.g. a file with a Xilinx DLL instantiated in it has the architecture name rtl\_xilinx.

Use behave for an architecture name that is typically not synthesizable, such as a testbench model.

Use empty for architectures that do nothing except drive their outputs to 1s or 0s. The file name for this architecture should be entity\_empty.vhd.

Use dummy for architectures that have limited functionality, i.e. a CPU that can read and write registers but nothing else. The file name for this architecture should be entity\_dummy.vhd.

When creating a new entity – drive all outputs to some default value if the functionality is not completed yet.

### Packages

**Package names should have \_pkg suffix. Package file names should be xx\_pkg.vhd**

### Components & Instances

**Use u\_xx for the instantiated name of an instance of entity xx. (This is a common convention in board level schematics and some of us are old enough to have done ASIC and FPGA designs with schematics and we are used to it. So there. The ‘u\_’ is also not typically used in any other way and makes it easy to search for instantiated components.**

In general, it is better to infer components, unless there is some specific reason why you want to make sure you get a specific component.

Try to avoid component declarations – instead, use the entity type of component instantiation:

u\_myblock: entity work.myblock;

For vendor tool generated IP, put the component declaration in either a component package or in the file in which it is instantiated, and use

u\_ram\_my\_seq: ram\_my\_seq;

Use functional IP names rather generic names. For instance, if generating a 256-word x 8-bit RAM for a sequencer, use ram\_my\_seq, rather than ram\_sp\_256x8.

Port-to-signal connections in a component instantiation should be specified with named association rather than positional association.(Added especially for Verilog-using scum. You know who you are)

### Generated Logic

**Use prefix g\_ for generate block labels**.

For example:

g\_xxx : for I in 0 to 3 generate

### State Machines

State machines should use enumerated types to define the states using t\_state or t\_state\_xxx to define the enumeration. The state variable should be named state or state\_xxx if there is more than one state machine in the file.

e.g. : type t\_state\_sys is (S\_SYS\_START, S\_SYS\_WAIT, S\_SYS\_SHUTDOWN, S\_SYS\_DONE);

### Summary of Naming Prefixes

|  |  |
| --- | --- |
| **Prefix** | **Use/Meaning** |
| u\_ | Instantiated components |
| g\_ | Generate block labels |
| pr\_ | Process names |
| v\_ | Variable names |
| t\_ | Type definition names |
| trec\_ | Record type definition names |
| clk\_ | Port or signal clock names |
| reset\_ | Port or signal reset names |
| cnt\_xx | A counter signal. E.g. cnt\_packet, cnt\_addr |
| G\_ | An entity generic |
| S\_ | A state name in a state-machine enumerated type |
| C\_ | A constant (optional) |
| tb\_ | Testbench file names |

### Summary of Naming Suffixes

|  |  |
| --- | --- |
| **Prefix** | **Use/Meaning** |
| \_pkg.vhd | Package file name |
| \_clk\_name | Signal belonging to clock domain clk\_name |
| \_d or \_d1, \_d2 | Pipeline delayed versions of a signal |
| \_msync1, \_msync2 | Metastabilizing synchronizer, or s\_xx\_d1, s\_xx\_d2 |
| \_n | Asserted low signal |
| \_i | Internal signal |
| n | Integer-type signal e.g. |

## Coding styles

In general, if the code seems to contain a lot of replication, then there is probably a better and more maintainable way of doing it.

### Whitespace

**Use 4 spaces for indentation – DO NOT use tabs.**

**Use consistent indentation throughout your code**. It can get very confusing to the reader when the indentation is incorrect.

Only put one statement per line.

Apply consistent use of blank lines. Aim for a clean look in your code

### Line Length

It is preferred to keep the line lengths to approximately 80 – 100 characters. Going over 140 characters can start to impact the readability of the code.

### Comments in General

Use them; they are as important as the code that is compiled. Make them accurate and maintain them. The only thing worse than no comments are incorrect comments. Comments like "-- architecture rtl of blocka" or “-- std\_logic\_vector(5 downto 0)” or “-- adds 1 to the count” do not add value. You can optionally put type declaration comments in component instantiations.

**Every process should have a comment.** You wrote this process for a reason. What is the reason?

Comments are meant to help others understand what the code is doing.

Do not add lots of comments that do not apply directly to the code in the file. Long descriptions probably belong in the FPGA Specification.

### Comments on Port Signals

Port signals in component packages do not need comments.

Port signals in entity declarations must have normal comments. Organize signals into functional groups. It’s good (but not required) to use a blank line between functional groups. The elements of each port declaration should align vertically in the same functional group, but each functional group may have independent alignment.

Port signals in instances do not have to have comments. However, it is often very helpful for someone new to the design to see ‘in’ or ‘out’ as a comment, especially for top-level ports.

### Resets

**Reset should be named reset or have a reset\_ prefix.**

Registers should use synchronous resets when possible.

All registers should be reset, unless there is some compelling reason not to.

Sometimes it is desirable to have a software-controlled reset that resets all the state machines and logic except for registers that are programmed by the CPU. This signal shall be named reset\_dp.

### Signal Definition

In RTL, signals should not be given initial values when the signal is defined. This design technique is not portable. If the signal is a register, initialize it using reset. It’s OK to do this in the testbench.

### Active Low Signals

**Don't use active low signals unless absolutely necessary**. If there are input pads that are active low, just invert them as soon as you can. If an IP block has an active low port, then make sure you use \_n on the end of the signal even if the IP port name doesn't. Some Xilinx memory blocks have an active low reset called reset. Drive this with a signal called reset\_n.

### State machines

Use enumerated types for the state variable or signal. Then you can see the state names in the simulator and SignalTap or ChipScope. The type name is the state variable name with a t\_ prefix. You can still get specific bit patterns for each state if you want to use them for debugging. The state names begin with S\_ (enumerations are uppercase). If there is only one state machine in a file, use state for the variable name, otherwise use state\_<name> format, where <name> describes the state machine.

-- Enumerated type for the system state variable

type t\_state\_sys is (S\_SYS\_START, S\_SYS\_WAIT, S\_SYS\_SHUTDOWN, S\_SYS\_DONE);

attribute enum\_encoding of t\_state\_sys : type is "00 01 10 11"; -- optional

signal state\_sys : t\_state\_sys;

There are two main methods of writing finite state machines (FSMs) in VHDL:

* Single synchronous process that does everything using one state variable signal.
* Two processes, one synch, one combinational, using a current state and a next state signal.

In general, **use the first method**. If practical, include registered control logic in the FSM process, so that the branch condition code is not duplicated in 2 process and so that the code is easier to understand.

Use two handshaking FSMs rather than an FSM within an FSM, unless trivial.

The main point is that the state machine is understandable and maintainable by someone other than you. Styles that make the code size smaller are generally better and more maintainable. If a single state machine is getting long and complicated, then maybe it’s better to break it up into smaller ones.

State machines should have all states defined. When using enum types and case statements have a “when others” state. This provides a way for the state machine to recover if ever gets in an undefined state.

### Port Record Types

Port record types should be used sparingly and only when they will aid in the use and understanding of the design. They should not be use in the ports of the top-level entity.

A record type for a CPU bus:

-- Record type for the CPU bus

type t\_rec\_cpubus is record

cs : std\_logic\_vector(15 downto 0);

rd : std\_logic;

wr : std\_logic;

adr : std\_logic\_vector(27 downto 0);

idat : std\_logic\_vector(31 downto 0);

odat : std\_logic\_vector(31 downto 0);

ack : std\_logic;

end record;

signal cpubus : t\_rec\_cpubus;

### Process Sensitivity Lists

Clocked processes with a synchronous reset should only have the clock in the sensitivity list.

Clocked processes with an asynchronous reset should only have the clock and the async reset in the sensitivity list before the clock. NOTE: this is stated for completeness, but in general, all registers should use synchronous resets, not asynchronous resets.

Avoid inferred latches with a complete list or use ‘process (all)’.

Write processes so that they do not need a huge sensitivity list.

### Common Functions Like Rounding and Saturation

Common functions should be put into a package so that multiple blocks can use the same functions. This will ensure that there is a consistency in the way that rounding and saturation are done.

### Common Error & Status Strategy

Implement a common way across the ASIC/FPGAs in a product to latch and report error and status conditions. Ensure that the documentation makes it clear for the software engineer to be able to follow an interupt to all of its possible causes.

In general, prevent counters, pointers, FIFOs, etc from overflowing.

In general, attempt to detect and flag error conditions.

### Test Features and Overrides

Ensure that if there are lockout mechanisms, that there are also test features which allow the lockouts to be overridden.

### Stale or Commented Out Code

Remove old code and signals that are not used.

Do not just comment out code – remove it. Configuration management should be used to go back and look at prior versions of the code.

Preferably have a tool which checks this in simulation or a Linting tool which detects this.

### Signals in an Unknown State

In general, all signals should always be driven to known state unless there’s a good reason not to be. When this cannot be avoided, then attempt to control the spread of the unknown signals by conditioning those signals with ones that are known. For example, a RAM output may be unknown, so that RAM output should not be the default value muxed onto a CPU read back bus.

### Use Arrays and Loops

Use array variables instead of replicating variables. Also use looping with these arrays instead of having replicated code. Use upper-case for loop variables. For example, use:

Constant SC\_MAX\_LINE : integer := 4;

type t\_arr\_sc\_lines is array (0 to SC\_MAX\_LINE) of std\_logic\_vector(31 downto 0);

signal arr\_sc\_lines : t\_arr\_sc\_lines;

arr\_sc\_lines(0) <= new\_line;

for I in 1 to SC\_MAX\_LINE loop

arr\_sc\_lines(I) <= arr\_sc\_lines(I – 1);

end loop;

Instead of:

signal sc\_lines0 : std\_logic\_vector(31 downto 0);

signal sc\_lines1 : std\_logic\_vector(31 downto 0);

signal sc\_lines2 : std\_logic\_vector(31 downto 0);

signal sc\_lines3 : std\_logic\_vector(31 downto 0);

sc\_lines0 <= new\_line;

sc\_line1 <= sc\_lines0;

scline2 <= sc\_line1;

scline3 <= sc\_line2;

### Generate Statements

Use generate statements to create arrays of components rather than individually replicating them. The purpose for doing this is to make the code more understandable and maintainable – if this causes the code to become convoluted due to strange port mapping of bit, then maybe a generate statement isn’t the way to go.

### File Naming

**VHDL filenames should match the entity or package name with a .vhd extension**.

If there is a file with an alternative architecture then it should append the architecture name to the basename of the file. For example, the file containing the dummy architecture for blocka would be blocka\_dummy.vhd.

In general, the architecture and entity definitions for an entity should be in one file named entity.vhd. Names of testbench files should use the ‘tb\_’ prefix.

### No Internal Tri-states

Enough said.

### Multiple Clocks

Preferably there is only one clock per entity. (Even better – only one clock in the entire FPGA.) If there is more than one clock, then the clock domains should be easily identifiable.

### RAM Enables

Enables on RAMs often provide a power down feature so the enables should always be controlled so that the RAMs are not enabled when they are not actively being read or written.

### Use of Libraries

**When using packages from a library that the source is compiled into, use “work” instead of the library name:**

use work.my\_pkg;

Don’t use:

library quark;

use quark.my\_pkg;

Work has the special meaning of the current library.

### Use of Clock Edge Functions for Clocked Processes

**Clocked processes should use the newer rising\_edge() and falling\_edge() functions** instead of the older “clk = 1 and clk’event”. The newer method is easier to read and is easier to search on to find clocked processes. It also makes it much easier to find negative edge clocked processes. Although the falling edge clocked processes should be avoided.

### Reasonable Limits on Process, Function and File Sizes

Excessively long processes, functions and/or file sizes are not very conducive to understanding or maintenance of the code. Break things up logically so they are easily understood by others.

### End of Lines

Each source code line should end with a Line Feed ASCII character (0x0A) only, instead of Carriage Return and Line Feed ASCII characters (0x0D 0x0A). Unix instead of DOS endings.

### Hold vs. Latch

When a synchronous process samples a signal at one time and clears it at another, it helps to name the signal with the suffix ‘\_hold’. Do not use ‘\_latch’. The string “latch” is misleading for a synchronous design.

## Coding Examples

Note use of whitespace and column alignment.

### CPU Registers – Read/Write Control

Use a single process to implement CPU reading and writing of registers.

Define your register addresses in constants either locally in the file, in a package for the block or in a package for the device.

 At the top of the architecture:

-- Register addresses

constant ADR\_CTLSTATUS : std\_logic\_vector (3 downto 0) := X"0";

constant ADR\_CMDIDX : std\_logic\_vector (3 downto 0) := X"1";

constant ADR\_CMDARG : std\_logic\_vector (3 downto 0) := X"2";

Register access process:

------------------------------------------------------------------------------

-- Write and read registers

------------------------------------------------------------------------------

pr\_regs\_rw : process(clk)

begin

if rising\_edge(clk) then

if (reset = '1') then

reg\_ctrlstatus <= (others=>'0');

reg\_cmdidx <= (others=>'0');

reg\_cmsarg <= (others=>'0');

regbus\_rd\_dat <= (others=>'0');

regbus\_rd\_dv <= '0';

elsif (regbus\_sel = '1') and (regbus\_wr = '1') then

case regbus\_adr is

when ADR\_CTLSTATUS => reg\_ctrlstatus <= regbus\_wr\_dat(29 downto 0);

when ADR\_CMDIDX => reg\_cmdidx <= regbus\_wr\_dat(12 downto 0);

when ADR\_CMDARG => reg\_cmdarg <= regbus\_wr\_dat;

when others => null;

end case;

regbus\_rd\_dat <= (others=>'0');

elsif (sd\_reg\_sel = '1') and (regbus\_wr = '0') then

case regbus\_adr is

when ADR\_CTLSTATUS => regbus\_rd\_dat <= reg\_ctrlstatus;

when ADR\_CMDIDX => regbus\_rd\_dat <= reg\_cmdidx;

when ADR\_CMDARG => regbus\_rd\_dat <= reg\_cmdarg;

when others => regbus\_rd\_dat <= CODE\_ERROR\_ADR; -- (from package)

end case;

regbus\_rd\_dv <= '1';

else

regbus\_rd\_dv <= ‘0’;

end if;

end if;

end process;

Outside of the register read/write process assign the register bits to individual signals.

----------------------------------------------------

-- Assign register bits (37 signals)

----------------------------------------------------

sd\_rdfifo\_flush <= reg\_ctrlstatus(29);

sd\_wrfifo\_flush <= reg\_ctrlstatus(28);

clr\_sd\_xmit\_dat\_done <= reg\_ctrlstatus(27);

clr\_sd\_rcv\_dat\_done <= reg\_ctrlstatus(26);

sd\_rate\_select <= reg\_ctrlstatus(25 downto 24);

rcvd\_dat3\_crc\_err <= reg\_ctrlstatus(23);

.

sd\_fpga\_core\_en <= reg\_ctrlstatus(0);

sd\_cmd\_go\_reg <= reg\_cmdidx(12);

sd\_cmd\_exp\_resp(31) <= reg\_cmdidx(8);

sd\_cmd\_index <= reg\_cmdidx(5 downto 0);

sd\_cmd\_argument <= reg\_cmdarg;

### Use ‘range for Signal Declarations

When creating signals that that are functionally related to another signal port (i.e. they will have the same bit width range), then use the ‘range to replicate the range of another signal or port.

entity top is

…

data\_in : in std\_logic\_vector(12 downto 0);

…

signal my\_data\_in : std\_logic\_vector(data\_in’range);

This will reduce the number of places that have to be modified if the width of the input changes.

### Use IEEE Standard Functions Where Possible

Don’t recreate a function that already exists in IEEE packages. However, see guidelines below for arithmetic functions. See appendix in this document.

### For Arithmetic Functions, Use ieee.numeric\_std Package

For all arithmetic functions use the newer ieee.numeric\_std package. It is preferred to declare all signals and ports that represent signed or unsigned values as the appropriate signed or unsigned types (except at the top level of the chip, where all ports should be std\_logic or std\_logic\_vector). Doing so will minimize the amount of type conversion and help clarify the type of value that is represented.

numeric\_std provides +, -, \*, /, rem, mod, <, >, <=, >=, =, /=, shift\_left, shift\_right, rotate\_left, rotate\_right, resize and conversion functions.

### Do not Use std\_logic\_arith, std\_logic\_unsigned or std\_logic\_signed

These packages are non-standard and were deprecated in 1993 (!!). They were vendor specific libraries (from Cadence, Mentor or Synopsys) and should never have been compiled into the IEEE libraries.

### Use resize() to Extend Signals

When assigning a shorter signal to a longer one, use the resize() function rather than hard coding a fixed length bit vector of zeros. For example, use:

rd\_dat <= resize(my\_signed\_val, 32);

rd\_dat <= resize(my\_unsigned\_val, 32);

or better yet:

rd\_dat <= resize(my\_signed\_val, rd\_dat’length);

rd\_dat <= resize(my\_unsigned\_val, rd\_dat’length);

### Use \*\_reduce() Functions in IEEE Packages for std\_logic\_vectors

The std\_logic\_misc package defines the reduce logic functions for std\_logic\_vectors as follows:

any\_bit\_set <= or\_reduce(my\_slv)

## Productivity Tools

## Find X’s in Simulation - TODO

Have a script that can find and report all X’s a number of clocks after reset. The tool would need a way to know when to stop looking, so that it doesn’t go into vendor cells. It would also need a way to indicated known X’s – i.e. a suppression file.

# IEEE Package Functions

-------------------------------------------------------------------------------

PACKAGE std\_logic\_1164 IS

-------------------------------------------------------------------------------

FUNCTION "and" ( l : std\_ulogic; r : std\_ulogic ) RETURN UX01;

FUNCTION "nand" ( l : std\_ulogic; r : std\_ulogic ) RETURN UX01;

FUNCTION "or" ( l : std\_ulogic; r : std\_ulogic ) RETURN UX01;

FUNCTION "nor" ( l : std\_ulogic; r : std\_ulogic ) RETURN UX01;

FUNCTION "xor" ( l : std\_ulogic; r : std\_ulogic ) RETURN UX01;

function "xnor" ( l : std\_ulogic; r : std\_ulogic ) return ux01;

FUNCTION "not" ( l : std\_ulogic ) RETURN UX01;

FUNCTION "and" ( l, r : std\_logic\_vector ) RETURN std\_logic\_vector;

FUNCTION "and" ( l, r : std\_ulogic\_vector ) RETURN std\_ulogic\_vector;

FUNCTION "nand" ( l, r : std\_logic\_vector ) RETURN std\_logic\_vector;

FUNCTION "nand" ( l, r : std\_ulogic\_vector ) RETURN std\_ulogic\_vector;

FUNCTION "or" ( l, r : std\_logic\_vector ) RETURN std\_logic\_vector;

FUNCTION "or" ( l, r : std\_ulogic\_vector ) RETURN std\_ulogic\_vector;

FUNCTION "nor" ( l, r : std\_logic\_vector ) RETURN std\_logic\_vector;

FUNCTION "nor" ( l, r : std\_ulogic\_vector ) RETURN std\_ulogic\_vector;

FUNCTION "xor" ( l, r : std\_logic\_vector ) RETURN std\_logic\_vector;

FUNCTION "xor" ( l, r : std\_ulogic\_vector ) RETURN std\_ulogic\_vector;

function "xnor" ( l, r : std\_logic\_vector ) return std\_logic\_vector;

function "xnor" ( l, r : std\_ulogic\_vector ) return std\_ulogic\_vector;

FUNCTION "not" ( l : std\_logic\_vector ) RETURN std\_logic\_vector;

FUNCTION "not" ( l : std\_ulogic\_vector ) RETURN std\_ulogic\_vector;

FUNCTION To\_bit ( s : std\_ulogic; xmap : BIT := '0') RETURN BIT;

FUNCTION To\_bitvector ( s : std\_logic\_vector ; xmap : BIT := '0') RETURN BIT\_VECTOR;

FUNCTION To\_bitvector ( s : std\_ulogic\_vector; xmap : BIT := '0') RETURN BIT\_VECTOR;

FUNCTION To\_StdULogic ( b : BIT ) RETURN std\_ulogic;

FUNCTION To\_StdLogicVector ( b : BIT\_VECTOR ) RETURN std\_logic\_vector;

FUNCTION To\_StdLogicVector ( s : std\_ulogic\_vector ) RETURN std\_logic\_vector;

FUNCTION To\_StdULogicVector ( b : BIT\_VECTOR ) RETURN std\_ulogic\_vector;

FUNCTION To\_StdULogicVector ( s : std\_logic\_vector ) RETURN std\_ulogic\_vector;

FUNCTION To\_X01 ( s : std\_logic\_vector ) RETURN std\_logic\_vector;

FUNCTION To\_X01 ( s : std\_ulogic\_vector ) RETURN std\_ulogic\_vector;

FUNCTION To\_X01 ( s : std\_ulogic ) RETURN X01;

FUNCTION To\_X01 ( b : BIT\_VECTOR ) RETURN std\_logic\_vector;

FUNCTION To\_X01 ( b : BIT\_VECTOR ) RETURN std\_ulogic\_vector;

FUNCTION To\_X01 ( b : BIT ) RETURN X01;

FUNCTION To\_X01Z ( s : std\_logic\_vector ) RETURN std\_logic\_vector;

FUNCTION To\_X01Z ( s : std\_ulogic\_vector ) RETURN std\_ulogic\_vector;

FUNCTION To\_X01Z ( s : std\_ulogic ) RETURN X01Z;

FUNCTION To\_X01Z ( b : BIT\_VECTOR ) RETURN std\_logic\_vector;

FUNCTION To\_X01Z ( b : BIT\_VECTOR ) RETURN std\_ulogic\_vector;

FUNCTION To\_X01Z ( b : BIT ) RETURN X01Z;

FUNCTION To\_UX01 ( s : std\_logic\_vector ) RETURN std\_logic\_vector;

FUNCTION To\_UX01 ( s : std\_ulogic\_vector ) RETURN std\_ulogic\_vector;

FUNCTION To\_UX01 ( s : std\_ulogic ) RETURN UX01;

FUNCTION To\_UX01 ( b : BIT\_VECTOR ) RETURN std\_logic\_vector;

FUNCTION To\_UX01 ( b : BIT\_VECTOR ) RETURN std\_ulogic\_vector;

FUNCTION To\_UX01 ( b : BIT ) RETURN UX01;

FUNCTION rising\_edge (SIGNAL s : std\_ulogic) RETURN BOOLEAN;

FUNCTION falling\_edge (SIGNAL s : std\_ulogic) RETURN BOOLEAN;

FUNCTION Is\_X ( s : std\_ulogic\_vector ) RETURN BOOLEAN;

FUNCTION Is\_X ( s : std\_logic\_vector ) RETURN BOOLEAN;

FUNCTION Is\_X ( s : std\_ulogic ) RETURN BOOLEAN;

END std\_logic\_1164;

-------------------------------------------------------------------------------

package std\_logic\_misc is

-------------------------------------------------------------------------------

function AND\_REDUCE (ARG: STD\_LOGIC\_VECTOR) return UX01;

function NAND\_REDUCE(ARG: STD\_LOGIC\_VECTOR) return UX01;

function OR\_REDUCE (ARG: STD\_LOGIC\_VECTOR) return UX01;

function NOR\_REDUCE (ARG: STD\_LOGIC\_VECTOR) return UX01;

function XOR\_REDUCE (ARG: STD\_LOGIC\_VECTOR) return UX01;

function XNOR\_REDUCE(ARG: STD\_LOGIC\_VECTOR) return UX01;

function AND\_REDUCE (ARG: STD\_ULOGIC\_VECTOR) return UX01;

function NAND\_REDUCE(ARG: STD\_ULOGIC\_VECTOR) return UX01;

function OR\_REDUCE (ARG: STD\_ULOGIC\_VECTOR) return UX01;

function NOR\_REDUCE (ARG: STD\_ULOGIC\_VECTOR) return UX01;

function XOR\_REDUCE (ARG: STD\_ULOGIC\_VECTOR) return UX01;

function XNOR\_REDUCE(ARG: STD\_ULOGIC\_VECTOR) return UX01;

end std\_logic\_misc;

-------------------------------------------------------------------------------

PACKAGE numeric\_std IS

-------------------------------------------------------------------------------

function "abs" (ARG: SIGNED) return SIGNED;

function "-" (ARG: SIGNED) return SIGNED;

function "+" (L, R: UNSIGNED) return UNSIGNED;

function "+" (L, R: SIGNED) return SIGNED;

function "+" (L: UNSIGNED; R: NATURAL) return UNSIGNED;

function "+" (L: NATURAL; R: UNSIGNED) return UNSIGNED;

function "+" (L: INTEGER; R: SIGNED) return SIGNED;

function "+" (L: SIGNED; R: INTEGER) return SIGNED;

function "-" (L, R: UNSIGNED) return UNSIGNED;

function "-" (L, R: SIGNED) return SIGNED;

function "-" (L: UNSIGNED;R: NATURAL) return UNSIGNED;

function "-" (L: NATURAL; R: UNSIGNED) return UNSIGNED;

function "-" (L: SIGNED; R: INTEGER) return SIGNED;

function "-" (L: INTEGER; R: SIGNED) return SIGNED;

function "\*" (L, R: UNSIGNED) return UNSIGNED;

function "\*" (L, R: SIGNED) return SIGNED;

function "\*" (L: UNSIGNED; R: NATURAL) return UNSIGNED;

function "\*" (L: NATURAL; R: UNSIGNED) return UNSIGNED;

function "\*" (L: SIGNED; R: INTEGER) return SIGNED;

function "\*" (L: INTEGER; R: SIGNED) return SIGNED;

function "/" (L, R: UNSIGNED) return UNSIGNED;

function "/" (L, R: SIGNED) return SIGNED;

function "/" (L: UNSIGNED; R: NATURAL) return UNSIGNED;

function "/" (L: NATURAL; R: UNSIGNED) return UNSIGNED;

function "/" (L: SIGNED; R: INTEGER) return SIGNED;

function "/" (L: INTEGER; R: SIGNED) return SIGNED;

function "rem" (L, R: UNSIGNED) return UNSIGNED;

function "rem" (L, R: SIGNED) return SIGNED;

function "rem" (L: UNSIGNED; R: NATURAL) return UNSIGNED;

function "rem" (L: NATURAL; R: UNSIGNED) return UNSIGNED;

function "rem" (L: SIGNED; R: INTEGER) return SIGNED;

function "rem" (L: INTEGER; R: SIGNED) return SIGNED;

function "mod" (L, R: UNSIGNED) return UNSIGNED;

function "mod" (L, R: SIGNED) return SIGNED;

function "mod" (L: UNSIGNED; R: NATURAL) return UNSIGNED;

function "mod" (L: NATURAL; R: UNSIGNED) return UNSIGNED;

function "mod" (L: SIGNED; R: INTEGER) return SIGNED;

function "mod" (L: INTEGER; R: SIGNED) return SIGNED;

function ">" (L, R: UNSIGNED) return BOOLEAN;

function ">" (L, R: SIGNED) return BOOLEAN;

function ">" (L: NATURAL; R: UNSIGNED) return BOOLEAN;

function ">" (L: INTEGER; R: SIGNED) return BOOLEAN;

function ">" (L: UNSIGNED; R: NATURAL) return BOOLEAN;

function ">" (L: SIGNED; R: INTEGER) return BOOLEAN;

function "<" (L, R: UNSIGNED) return BOOLEAN;

function "<" (L, R: SIGNED) return BOOLEAN;

function "<" (L: NATURAL; R: UNSIGNED) return BOOLEAN;

function "<" (L: INTEGER; R: SIGNED) return BOOLEAN;

function "<" (L: UNSIGNED; R: NATURAL) return BOOLEAN;

function "<" (L: SIGNED; R: INTEGER) return BOOLEAN;

function "<=" (L, R: UNSIGNED) return BOOLEAN;

function "<=" (L, R: SIGNED) return BOOLEAN;

function "<=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;

function "<=" (L: INTEGER; R: SIGNED) return BOOLEAN;

function "<=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;

function "<=" (L: SIGNED; R: INTEGER) return BOOLEAN;

function ">=" (L, R: UNSIGNED) return BOOLEAN;

function ">=" (L, R: SIGNED) return BOOLEAN;

function ">=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;

function ">=" (L: INTEGER; R: SIGNED) return BOOLEAN;

function ">=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;

function ">=" (L: SIGNED; R: INTEGER) return BOOLEAN;

function "=" (L, R: UNSIGNED) return BOOLEAN;

function "=" (L, R: SIGNED) return BOOLEAN;

function "=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;

function "=" (L: INTEGER; R: SIGNED) return BOOLEAN;

function "=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;

function "=" (L: SIGNED; R: INTEGER) return BOOLEAN;

function "/=" (L, R: UNSIGNED) return BOOLEAN;

function "/=" (L, R: SIGNED) return BOOLEAN;

function "/=" (L: NATURAL; R: UNSIGNED) return BOOLEAN;

function "/=" (L: INTEGER; R: SIGNED) return BOOLEAN;

function "/=" (L: UNSIGNED; R: NATURAL) return BOOLEAN;

function "/=" (L: SIGNED; R: INTEGER) return BOOLEAN;

function SHIFT\_LEFT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;

function SHIFT\_RIGHT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;

function SHIFT\_LEFT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;

function SHIFT\_RIGHT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;

function ROTATE\_LEFT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;

function ROTATE\_RIGHT (ARG: UNSIGNED; COUNT: NATURAL) return UNSIGNED;

function ROTATE\_LEFT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;

function ROTATE\_RIGHT (ARG: SIGNED; COUNT: NATURAL) return SIGNED;

function "sll" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;

function "sll" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;

function "srl" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;

function "srl" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;

function "rol" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;

function "rol" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;

function "ror" (ARG: UNSIGNED; COUNT: INTEGER) return UNSIGNED;

function "ror" (ARG: SIGNED; COUNT: INTEGER) return SIGNED;

function RESIZE (ARG: SIGNED; NEW\_SIZE: NATURAL) return SIGNED;

function RESIZE (ARG: UNSIGNED; NEW\_SIZE: NATURAL) return UNSIGNED;

function TO\_INTEGER (ARG: UNSIGNED) return NATURAL;

function TO\_INTEGER (ARG: SIGNED) return INTEGER;

function TO\_UNSIGNED (ARG, SIZE: NATURAL) return UNSIGNED;

function TO\_SIGNED (ARG: INTEGER; SIZE: NATURAL) return SIGNED;

function "not" (L: UNSIGNED) return UNSIGNED;

function "and" (L, R: UNSIGNED) return UNSIGNED;

function "or" (L, R: UNSIGNED) return UNSIGNED;

function "nand" (L, R: UNSIGNED) return UNSIGNED;

function "nor" (L, R: UNSIGNED) return UNSIGNED;

function "xor" (L, R: UNSIGNED) return UNSIGNED;

function "xnor" (L, R: UNSIGNED) return UNSIGNED;

function "not" (L: SIGNED) return SIGNED;

function "and" (L, R: SIGNED) return SIGNED;

function "or" (L, R: SIGNED) return SIGNED;

function "nand" (L, R: SIGNED) return SIGNED;

function "nor" (L, R: SIGNED) return SIGNED;

function "xor" (L, R: SIGNED) return SIGNED;

function "xnor" (L, R: SIGNED) return SIGNED;

function STD\_MATCH (L, R: STD\_ULOGIC) return BOOLEAN;

function STD\_MATCH (L, R: UNSIGNED) return BOOLEAN;

function STD\_MATCH (L, R: SIGNED) return BOOLEAN;

function STD\_MATCH (L, R: STD\_LOGIC\_VECTOR) return BOOLEAN;

function STD\_MATCH (L, R: STD\_ULOGIC\_VECTOR) return BOOLEAN;

function TO\_01 (S: UNSIGNED; XMAP: STD\_LOGIC := '0') return UNSIGNED;

function TO\_01 (S: SIGNED; XMAP: STD\_LOGIC := '0') return SIGNED;

# Appendix

## Abbreviations

General rule is not to abbreviate if the name is short or if it ends up cryptic.  It’s OK to not abbreviate; for example, master vs. mstr. Do not remove one character to shorten a 5+ character word if it ends up being cryptic. For instance, changing ‘burst’ to ‘brst’ has one thinking of ‘b\_reset’, ‘branch\_state’, etc. and offers no significant savings in length.

Example abbreviations:

enable : ena

disable : dis

reset : rst

block : blk

control : ctrl

status : stat

command : cmd

register : reg

definition : defn

information : info

configuration : cfg/config

revision : rev

initialization : init

select : sel

clock : clk

address : addr/adr

data : data

strobe : stb

input : in

output : out

write : wr

read : rd

array : arr

pointer : ptr

index : idx

count : cnt

counter : cntr

header : hdr

delay : dly

coefficient : coef/coeff

format : fmt

sample : smp/smpl

trigger : trig

detection : det

threshold : thresh

round : rnd

saturate : sat

result : rslt

buffer : buff

burst : burst

blank : blank

compress : cmpr

decompress : decmpr

encode : enc

decode : dec if context clear from decrement : dec

load : ld

increment : inc/incr

decrement :  dec/decr

decimate : decim

accumulator : acc

multiplier : mult

divisor : div

average : avg

interrupt : int

interval : interv

frequency : freq

period : per

amplitude : amp/ampl

random : rand not rndm

sequence : seq

period : per

count : cnt

length : len

size : size

width : width           not  wd, looks like word

filter : filter        not flt, looks like float/fleet/fillet/flat

default : default       not dflt, looks like deflate

notify : notify       not ntfy, looks like not-fully/nifty

test : tst

shift : shift          not shft

shift\_reg : sreg

time : time            perhaps use count instead

depth : depth

idle : idle

overflow : ovfl

underflow : undfl

empty : empty/mt

almostfull : af

full : full

master : mstr

slave : slave slv is standard\_logic\_vector

start : start not strt, looks like straight

stop : stop

flag : flag

data : data

error : err

locked : locked

software : sw

hardware : hw

request : req

grant : gnt

arbitrate : arb

packet : pkt

on : on

off : off

offset : ofs

wobbler : wob

calibration : cal

number : num

first : first/1

second : sec(time) or second/2(number)