DSA Lab VHDL Coding Style

V1.0

Novi Sad, 2009.

Contents

1. OBJECT NAMING CONVENTIONS

VHDL language defines various objects and language constructs that can be used to model hardware devices. In order to assure easy understanding, maintainability, verification and reusability of the VHDL models a consistent, universal style for such things as signal, variable declarations, entity declarations, component declarations, port mappings, functions and procedures should be used. Following table defines the preferred naming conventions that should be used in the new designs.

Object	Name	Example
Constant	*constant_name_c*	data_width_c
Signal	*signal_name_s*	data_out_s
Variable	*variable_name_v*	temp_v
File	*file_name_f*	log_file_f
Array type	*type_name_t_arr*	input_mem_t_arr
Record type	*type_name_t_rec*	status_t_rec
Range type	*type_name_t_range*	data_t_range
Enumeration type	*type_name_t_enum*	control_unit_t_enum
Elements of entity/arch	**Name**	**Example**
Generic	*generic_name_g*	addr_bus_width_g
Port	*port_name_i, _o, _io*	command_bus_i
Process	*process_name_p*	control_unit_p
Loop	*loop_name_l*	filter_l
Function	*function_name*	compare_response
Procedure	*procedure_name*	compare_response
Component	*component_name*	alu

2. UNIT NAMING CONVENTIONS

VHDL contains five design unit constructs (entity, architecture, package, package body and configuration) that can be independently analyzed and stored in a design library. We are concerned only with the first four.
Each file shall be named according to the unit it contains.
A file shall contain only one design unit. This minimizes the amount of recompilation required when a library unit, on which other library depends, is modified.

2.1. Entity

- Represent the interface I/O definition and generics.
- When appropriate, make use of generics for buffer sizes, bus width and all other unit parameters. This provides more readability and reusability of the code.
- Use separate line for each generic or port you declare.
- Add a comment after each declaration describing the purpose of generic/port.

Filename: *entity_name.vhd*

Example: (dac_ltc2624_rtl.vhd)

```
entity dac_ltc2624 is
        generic (
                clkcnt_value_g : std_logic_vector (27 downto 0) := x"8000000"; -- clk counter border
                depth_g :        integer range 1 to 99 := 8;                   -- sine signal 8bit quantized
                width_g :        integer range 1 to 99 := 12                   -- 12 bit sine amplitude value
                );
        port (
                ampl_cnt :    in std_logic_vector; -- amplitude counter
                btn_reset :   in std_logic;        -- reset button
                clk_in :      in std_logic;        -- 50MHz clock
                clkdv_in :    in std_logic;        -- 25MHz clock
                dac_clk :     out std_logic;       -- dac clock
                dac_cs_n :    out std_logic;       -- dac enable
                dac_data :    out std_logic;       -- dac data
                dac_reset_n : out std_logic;       -- dac reset
                freq_trig :   in std_logic         -- frequency for dac data packages
               );
end;
```

2.2. Architecture

- Architecture defines how the system behaves. This description can be in different levels of abstraction or different purpose.
- Together the entity/architecture pair represents a component.
- Use the following architecture names depending on the used abstraction level.
    - Behavioural *beh*
    - Structural *structure*
    - Register Transfer Level *rtl*
    - Functional *fun*
    - Transaction Level Modeling *tlm*

  - Testbench *tb*
- Use the company name if it is specific for a company like altera_rtl, xilinx_rtl, lattice_rtl
- Use the family name if it is family specific like xc2vp_rtl,xc9500_rtl

Filename: *entity_name_architecture_name.vhd*

Example: (modulator_rtl.vhd)
```
architecture rtl of modulator is
...
begin
...
end;
```

2.3. Package

- Provide a collection of declarations (types, constant, signals, component) or subprograms (procedures, functions).
- The subprogram bodies are not described.
- Where possible, packages approved by the IEEE should be used rather than redeveloping similar functionality.
- Packages specific to a particular CAD tool standard should not be used.
- The number of packages used by a model shall not be excessive.

Filename: *package_name_pkg.vhd*

Example: (modulator_pkg.vhd)
```
package modulator_pkg is
        type vector_t_arr is array (natural range <>) of integer;
        function init_sin_f (
                                constant depth_c : in integer;
                                constant width_c : in integer
                                )
        return vector_t_arr;
end;
```

2.4. Package Body

- Provide a complete definition of the subprograms.

Filename: *package_name_body.vhd*

Example: (modulator_pkg_body.vhd)
```
package body modulator_pkg is
        function init_sin_f (
                                depth_c : in integer;
                                width_c : in integer
                                )
        return vector_t_arr is
                variable init_arr_v : vector_t_arr(0 to (2 ** depth_c));
        begin
                for i in 0 to ((2 ** depth_c) / 2) loop -- calculate positive amplitude values
                        init_arr_v(i) := integer(round(sin((math_2_pi / real(2 ** depth_c))*
                        real(i)) * real(2 ** (width_c - 1)))) + integer(2 ** (width_c - 1) - 1);
                end loop;
```

```
                for i in ((2 ** depth_c) / 2 + 1) to (2 ** depth_c) loop -- calculate negativ amplitude
values
                        init_arr_v(i) := integer(round(sin((math_2_pi / real(2 ** depth_c))*
                        real(i)) * real(2 ** (width_c - 1)))) - integer(2 ** (width_c - 1));
                end loop;
                return init_arr_v;
        end;
end;
```

3. TYPE CONVERSIONS

Use **std_logic_arith, std_logic_unsigned/std_logic_signed** packages. These provide the essential conversion functions:

- **conv_integer** (<*signal_name*>)
  - Converts **std_logic_vector**, **unsigned**, and **signed** data types into an **integer** data type.
- **conv_unsigned** (<*signal_name*>, <*size*>)
  - Converts a **std_logic_vector**, **integer**, **unsigned** (change size), or **signed** data types into an **unsigned** data type.
- **conv_signed** (<*signal_name*>, <*size*>)
  - Converts a **std_logic_vector**, **integer**, **signed** (change size), or **unsigned** data types into a **signed** data type.
- **conv_std_logic_vector** (<*signal_name*>, <*size*>)
  - converts an **integer**, **signed**, or **unsigned** data type into a **std_logic_vector** data type.
- **ext** (<*signal_name*>, <*size*>)
  - zero extends a **std_logic_vector** to size <*size*>.
- **sxt** (<*signal_name*>, <*size*>)
  - sign extends a **std_logic_vector** to size <*size*>.

All conversion functions can take for the <*signal_name*> data-type a **std_logic_vector**, **unsigned**, **signed, std_ulogic_vector**, or **integer**. <*size*> is specified as an integer value.

4. VHDL STYLE GUIDE

4.1. Denotation

- Do not mix between VHDL coding standards for the whole project.
- Use VHDL-93.
- Everything (all VHDL keywords and all user-defined identifier) shall be in lower case.
- All user-defined identifiers shall be meaningful and have words separated by underscores, based on the English language.
- Use the same identifier name for the actual hardware that is used in the data sheet.
- Named association shall be used preferably to positional association.
- Only literals in base 2, 8, 10, and 16 shall be used.
- Extended digits in base-16 literals should be in lowercase.
- Variable width ports shall be constrained using generics.

4.2. Positioning

- Use indentation size in steps of four spaces. See any example code from this document.
- Declarative regions and blocks shall be indented by four spaces.
- Indentation level in sequential statements shall not exceed 4.
- Indented regions in sequential statements shall not have more than 60 lines.
- The TAB character shall not be used to indent, only use the space character.
- Lines should not exceed 120 characters.
- Long lines shall be broken where there are white spaces.
- Line continuations shall be indented to line-up with the first token at the same nesting level or by four spaces.
- One line shall separate concurrent statements and their descriptive comment.
- Groups of logically related statements and declaration shall be separated by one blank line.
- Unless otherwise specified, tokens shall be separated by one space.
- No space shall precede a close parenthesis or semi-colon.
- No space should surround a single quote or dot.
- Each statement shall start on a new line.
- Each declaration shall start on a new line.
- Elements in interface declarations shall be vertical aligned.
- Elements in signal, constant declarations shall be vertical aligned.
- Elements in a named association than span more than one line should be vertical aligned.
- Buffer and Linkage ports shall not be used.
- Guarded blocks and guarded signals should not be used.
- Use too many parentheses; never let the tool resolve precedence; explicitly declare precedence via parenthesis.
- Use relative path.
- Include only libraries which are really use in the design!

- For interfacing other modules use only **std_logic** and **std_logic_vector** as type.
- For arithmetic operations use library **ieee.std_logic_signed_bit.all** and **ieee.std_logic_signed.all.**
- Use preferred libraries **ieee.std_logic_1164.all**, **std.text.all** and **std.logic_textio.all**.

5. COMMENTING CODE

5.1. General Rules

- Comments should include a header template for each entity-architecture pair and for each package and package-body pair. See the example in section 5.2. The functional description field should include a brief description of the functionality of each lower block instantiated within it.
- Use comment headers for processes, functions, and procedures, as shown in section 5.3. This should be a description of the purpose of that block of code.
- Use comments internal to processes, functions, and procedures to describe what a particular statement is accomplishing. While the other two levels of commenting should always be included, this level is left to the designer to decipher what is required to convey intent. Inline comments are shown in Figure 13-6.
- Comments shall be immediately followed by the code they describe.
- Comments for port and signal declarations shall be in the same line.

5.2. Header

```
--------------------------------------------------------------------------------
-- File : modulator_rtl.vhd
-- Project : modulator
-- Creation : 25.05.2009
-- Limitations : none
-- Errors : none known
-- Simulator : Modelsim SE 6.1f
-- Synthesizer : ISE 10.1
-- Platform : Windows XP
-- Targets : Simulation, Synthesis, Implementation
-------------------------------------
-- Naming conv. : dsalab_vhdl_guide_.pdf
-------------------------------------
-- Authors : Rastislav Struharik (rasti)
-- Organization : FTN
-- Email : rasti@eunet.rs
-- Address : Trg Dositeja Obradovica 6, 21000 Novi Sad, Serbia/Europe
--------------------------------------------------------------------------------
-- Copyright Notice
-- This work is licensed under the General Public License (GPL).
--------------------------------------------------------------------------------
-- Function description
-- Frequency modulator with output for dac
--------------------------------------------------------------------------------
-- Revision History
-- Date       Author  Comments
-- 25.05.09  rasti     Created
-- 30.05.09  rasti     Added control unit
--------------------------------------------------------------------------------
```

5.3. Process, Function, and Procedure Header

```
-------------------------------------------------------------------------------------------
-- demux_proc: this process dumultiplexes the inputs and registers the
-- demultiplexed signals
-------------------------------------------------------------------------------------------
demux_proc : process (clk, reset)
begin
        …
end process;
```

5.4. Inline Comments

```
-------------------------------------------------------------------------------------------
-- demux_proc: this process dumultiplexes the inputs and registers the
-- demultiplexed signals
-------------------------------------------------------------------------------------------
demux_proc : process (clk, reset)
begin …
        if reset = '1' then
                demux <= (others => '0');
        elsif rising_edge(clk) then
        -- demultiplex input onto the signal demux
                case (sel) is
                        when '0' =>
                                demux(0) <= input;
                        when '1' =>
                                demux(1) <= input;
                        when others =>
                                demux <= (others => '0');
                end case;
        end if;
end process;
```

6. GENERAL CODING RULES FOR DIGITAL DESIGNS

6.1. Reset

- Use *reset* or *rst* for synchronous reset.
- Use *areset* or *arst* for asynchronous reset.
- Prefer the synchronous reset whenever possible.
- Do not assign an asynchronous reset to sequential statements.
- For simulation use default value assignment at signal declaration.

6.2. Clocks

- Use *clk* prefix for clock. This might include *clk_div2* (clock divided by 2), *clk_x2* (clock multiplied by 2), etc.
- Use only synchronous design techniques.
- Do not use gated clocks.
- No internal clk-generation in modules because of glitching and clock-skew related problems.
- Avoid using latches.
- Do not use buffer as delay elements.
- All block's external IOs should be registered.
- Avoid using both clock edges.
- Clock signal must be connected to global dedicated routing resources.
- Do not use clock or reset as data or as enable.
- Do not use data as clock or a reset.
- Signals that cross-different clock domains should be sampled before and after crossing domains (double sampling is preferred to minimize meta-stability).
- Use the lowest possible clock speed in each clock domain.
- Use the minimum number of clock domains.
- Use clock enables.
- Clock enables can only be inferred in a *clocked process*.
- Clock enables can be inferred explicitly by testing an enable signal. If the enable is true, the signal is updated. If enable is false, that signal will hold its current value.
- Clock enables can be implicitly inferred in two ways:
    - Not assigning to a signal in every branch of an if-then-else statement or case statement. Remember that latches will be inferred for this condition in a *combinatorial process*
    - Not defining all possible states or branches of an if-then-else or case statement.

6.3. General Signals

- Never assign to a signal in more than one process, with the exception of a three-state signal.
- Use *addr* suffix for addresses. This might include *sys_addr, up_addr*, etc.

- Use a *_n* suffix for active low signals <*signal_name*>*_n*.
- Use <*signal_name*>*_p0*, <*signal_name*>*_p1*, and so forth, to represent a pipelined version of the signal <*signal_name*>.

6.4. Buses

- Start buses at the LSB.
- Use MSB to LSB for data buses.
- Use LSB to MSB for delay lines and shift registers.
- Start counting with zero.
- Avoid using internal tri-state signals; they either no longer exist on FPGAs or come in short supply.

6.5. Finite State Machine FSM

- Generally, use two process statements for a state machine: one process for next-state decoding, and output decoding, and one for registering of state bits and outputs.
- Use a Mealy look-ahead state machine with registered outputs whenever possible, or use a Moore state machine with next-state output decoding and registered outputs to incur the minimum amount of latency.
- Use enumeration types for the states
- Use one hot encoding for FPGAs
- Use binary, gray, sequential coding for CPLDs

6.6. Memories

- Use synchronous single-port or dual-port block memories for sync read and write.
- Use asynchronous distributed RAMs for sync write and asynchronous read.

6.7. Rules for Required Instantiations for Xilinx

- Boundary Scan (BSCAN)
- Digital Clock Manager (DCM) or Delay-Locked Loop (DLL). Instantiating the DCM/DLL provides access to other elements of the DCM, as well as elimination of clock distribution delay. This includes phase shifting, 50-50 duty-cycle correction, multiplication of the clock, and division of the clock.
- IBUFG and BUFG. IBUFG is a dedicated clock buffer that drives the input of the DCM/DLL. BUFG is an internal global clock buffer that drives the internal FPGA clock and provides the feedback clock to the DCM/DLL.
- DDR registers. DDR registers are dedicated Double-Data Rate (DDR) I/O registers located in the input or output block of the FPGA.
- Startup. The startup block provides access to a Global Set or Reset line (GSR) and a Global Three-State line (GTS). The startup block is not inferred because routing a global set or reset line on the dedicated GSR resources is slower than using the abundant general routing resources.
- I/O pullups and pulldowns (pullup, pulldown).

7. PROJECT STRUCTURE

Every project should use the following file structure and SVN for the source revision control. Only the files from the /*src* folder should be under the version control.

/project_name
/literature
/result
/release
/src
/work

Content of each of the folders is:

- */literature*
  - Literature that is necessary to understand the project details. This doesn't include the documents created by the project team, only the books, articles, datasheets, user manuals and other documents explaining the theory and existing implementations of the project goals.
- */src*
  - Everything insight the source directory should be under version control, no binary files. Structure of the *src* folder is described in great detail later.
  - No version numbers in file names under revision control.
- */result*
  - This folder contains temporary files generated from the tools in subdirectories with the <*toolname*>\ like *ise91sp3, ise82sp3, edk91sp2, msim61f, pads2007, matlab2006b*.
- */release*
  - This folder contains release files within the subdirectories with some kind of design names.
- */work*
  - Temporary folder with subdirectories containing project data for various tools, for example, *xilinx* subdirectory containing Xilinx ISE project files, *modelsim* subdirectory containing ModelSim project files, etc.

Structure of the *src* folder is present in the following picture.

src\.
c\.
*.c
*.cpp
*.h
cmd\
*.make
*.bat
*.sh
doc\.
figure\.
*.jpg
*.vsd
html\.
pdf\.
hardware\.
*.sch
*.pcb
*.bom
matlab\.
modelsim\.
vhdl\.
tb\.
xilinx\.
coregen\.

Explanation of the content of every folder:

- */c*
  - Folder used to store the C, C++ source code files. This folder exists only if the project uses some microprocessor (typically MicroBlaze, PowerPC, or some other processor IP cores like 8051, 68000, etc.)
- */cmd*
  - Folder containing script files required for running implementations, compilations, simulations, etc.
- */doc*
  - This folder contains documentation files related to the project. *Figure* subdirectory is used for storing figures, *html* subdirectory is used for storing *.html files and *pdf* subdirectory is used for storing *.pdf files. If not required these subdirectories may be ommited.
- */hardware*
  - Folder containing design files related to the other hardware parts of the project apart from the FPGA design. These typically include PCB board schematics, layout and assembly files. If not required this folder can be ommited.
- */matlab*
  - Folder used to store Matlab/Simulink source files. If not used, this folder can be ommited.
- */modelsim*

  - Folder used to store ModelSim source files. These files typically include *.ini files and *.do files. If not used, this folder can be ommited.
- */vhdl*
  - Folder used to store VHDL model files for the FPGA design. Testbench files should be stored in the *tb* subfolder.
- */xilinx*
  - Folder used to store Xilinx tools source files. These files typically include *.ucf files, script and configuration files for various Xilinx tools. If some Xilinx cores are used in the design, design files generated by the CoreGenerator should be stored in the \coregen folder.

8. FILE NAME CONVENTION

- Use short but meaningful names.
- Use only lower case characters.
- Use no special characters, only alphanumeric and the underscore _ to separate two substrings are allowed, no spaces are allowed.
- Name big things first like cpu_register, not register_cpu!
- Use only English terms, avoid keywords or commands.
- Good names are part of the documentation.
- Rename names if the change the functionality.
- The structure of the file system hierarchy shall mirror the logical structure of the system being modelled. A directory should correspond to one and only one design unit.
- Do not use version numbers, if the files are under version control like SVN.