Guidelines for VHDL Coding

Patrick Loschmidt[1], Nataša Simanić[1], César Prados[2]

[1]Research Unit for Integrated Sensor Systems, Austrian Academy of Sciences

{Patrick.Loschmidt, Natasa.Simanic}@OEAW.ac.at

[2]Gesellschaft für Schwerionenforschung mbH, GSI

C.Prados@GSI.de

2009-04-21

Abstract

This is a summary of coding style rules which is intended to be a guideline for writing portable and readable VHDL code. Following these rules should result in better code quality or, at least, should make finding errors easier. This version was printed on April 21, 2009.

Contents

Changelog for rev. 327

2009-04-21

- changed postfix for pulsed signals

2009-03-03

- Author info added
- *Doxygen* chapter modified

2003-02-19

- SVN revision number / date
- *Doxygen* and Changelog chapters added
- Variables usage and placing component instantiation determined
- Naming conventions extended
- Combinational blocks template added

2009-02-08

- Minor text corrections
- Upper case naming convention changed

2008-12-19

- Initial release

1 Conventions

A lot of cross references are used because they make it easier to find other corresponding parts within the document. A link looks like "(→ 2.1)" and always belongs to the word(s) right before it.
Further, this document distinguishes between different types of rules. An explanation of these types and the appropriate symbol is shown in table 1.

symbol	type	description
(*)	important	This rule should be followed in any case.
(x)	recommended	Coding and reading is a lot easier with this rule.
(o)	practical	Your code quality will improve using the rule.
none	hints	Just giving you an idea of how it could be.

Table 1: Rule types

2 Formatting Rules

2.1 (*) Fonts

If you are not using a text mode editor, coding is much easier with a fixed pitch font. Otherwise you cannot benefit from using tabulators (→ 2.3) and reading the code will be difficult. To demonstrate the difference an example with a fixed pitch font and one without is shown in figure 1.

```
p_clock_generator: process
begin
  if (reset_n_i = '0') then
    s_clk <= '1';
  else
    wait for g_PERIOD/2;
    if s_clk = '0' then
      s_clk <= '1';
    else
      s_clk <= '0';
    end if;
  end if;
  clk_o <= s_clk;
end process p_clock_generator;
```

good example

```
p_clock_generator: process
begin
  if (reset_n_i = '0') then
    s_clk <= '1';
  else
    wait for g_PERIOD/2;
    if s_clk = '0' then
      s_clk <= '1';
    else
      s_clk <= '0';
    end if;
  end if;
  clk_o <= s_clk;
end process p_clock_generator;
```

bad example

Figure 1: Font examples

2.2 (x) Line Width

The line length of the code should not exceed 80 characters. Many editors cannot handle longer lines or you may get in trouble when printing out the source file. Remember that there may be other programmers who would want to take look at your code!
Because 80 characters are rather few, maximum of 100 characters is suggested instead. With common graphical editors it is possible to find a fixed font (→ 2.1) and an appropriate size to print the code out without a line break. No matter which line width you decide to use, no line should exceed the screen width. Otherwise it is very hard to read the code. To further improve readability, each line should contain one statement at maximum.

2.3 (*) Tabulators

Don't use so called "hard tabulators" because each editor (and printer as well) can have a different width assigned to them. Because each user uses a different editor (or printer) somebody might have problems reading your code easily.
For better readability use so called "soft tabulators", which is nothing else than a sequence of spaces. Every editor (or printer) is able to display this correctly. It is recommended to use soft tabulators with an equivalent of 2 spaces, but you can also use 2-4 spaces. The more spaces you use the easier it is to find blocks in your code, however you may run into troubles with the line width (→ 2.2).

2.4 (*) File Header

Each file in your design should contain a standardised header, where important information about the file content is stored. Further, this header should improve readability and maintenance of your project. It should provide all information you need to handle your code in a convenient way. For more details refer to figure 2.

2.5 (*) Comments

As it is true for all programming languages, each important operation and definition should have a comment above or beside it, describing operation of the statement. You will make life much easier for someone who would like to add functionality or fix a bug. Not to mention it is good for you as well, if you try to change the code after some time.
There are special elements in the code which need extended comments like the file header (→ 2.4), entities or architectures (→ 3.3), blocks or processes (→ 3.2). The following paragraphs show comments for these elements.

2.6 (*) Keywords

The VHDL-93 keywords are shown in figure 6. They should be written in lower case to distinguish them from user defined names (→ 3). Further, you should avoid the use of keywords in your signal names, entity names etc. Otherwise, you may get confused about their function when reading these words in your names or get into trouble with the compiler.

```
--------------------------------------------------------------------------------
--                                                                            --
--            company name, division and the name of the design               --
--                                                                            --
--------------------------------------------------------------------------------
--
--     unit name: <full name> (<shortname / entity name>)
--
--        author: <author name> (<email>)
--
--          date: $Date::                $:
--
--       version: $Rev::                 $:
--
--   description: <file content, behaviour, purpose, special usage notes...>
--                <further description>
--
--  dependencies: <entity name>, ...
--
--    references: <reference one>
--                <reference two> ...
--
--   modified by: $Author::               $:
--
--------------------------------------------------------------------------------
--  last changes: <date> <initials> <log>
--                                  <extended description>
--------------------------------------------------------------------------------
--          TODO: <next thing to do>
--                <another thing to do>
--
--------------------------------------------------------------------------------
```

Figure 2: Standard file header

```
--================================================================================
-- Entity declaration for <long entity name of my_entity>
--================================================================================
entity my_entity is
  generic (
    g_MYGENERIC : positive := 32 -- something which has to be configurable
  );
  port (
    -- global input signals
    clk_i     : in    std_logic; -- local bus clock
    reset_n_i : in    std_logic; -- reset =0: reset active
                                 --       =1: no reset

    -- global output signals
    led_o     : out   std_logic; -- LED =0: LED on
                                 --     =1: LED off

    -- data bus(es)
    data_b    : inout std_logic_vector (7 downto 0) -- data bus
  );
end entity my_entity;
```

Figure 3: Comments for entities

```
--================================================================================
-- architecture declaration
--================================================================================
architecture rtl of my_entity is

signal s_clk_local : std_logic; -- local clock

--================================================================================
-- architecture begin
--================================================================================
begin

end architecture rtl;
--================================================================================
-- architecture end
--================================================================================
```

Figure 4: Comments for architectures

```
--================================================================================
-- Begin of my_process
-- <description>
--================================================================================
-- read:  clk_i, reset_n_i
-- write: s_clk_local
-- r/w:   led_o
p_my_process: process (clk_i, reset_n_i)
begin
  if (clk_i'event and clk_i = '1') then -- synchronous process
    if reset_n_i = '0' then
      -- reset to default value
      led_o <= '1';
    else
      -- generate clock signal and LED output
      s_clk_local <= not s_clk_local; -- s_clk_local now runs at half of clk_i
      if s_clk_local = '1' then
        led_o <= '0' -- turn on LED if s_lclk is high
      end if;
    end if;
  end if;
end process p_my_process;
```

Figure 5: Comments for processes

```
abs, access, after, alias, all, and, architecture, array, assert, attribute
begin, block, body, buffer, bus
case, component, configuration, constant
disconnect, downto
else, elsif, end, entity, exit
file, for, function
generate, generic, group, guarded
if, impure, in, inertial, inout, is
label, library, linkage, literal, loop
map, mod
nand, new, next, nor, not, null
of, on, open, or, others, out
package, port, postponed, procedure, process, pure
range, record, register, reject, rem, report, return, rol, ror
select, severity, shared, signal, sla, sll, sra, srl, subtype
then, to, transport, type
unaffected, units, until, use
variable
wait, when, while, with
xnor, xor
```

Figure 6: VHDL 93 keywords

2.7 Syntax Highlighting

To improve readability of your code you may want to use an editor which is capable of syntax highlighting. Colour coding the keywords (→ 2.6) of VHDL and especially the comments does help you to keep an overview of the code.

3 Name Style Rules

Use only English language for all names given. It simplifies the transfer of your design to other parties involved. it will often be convenient to use standard names for signals, like reset or clock, where it does not make sense to translate them. As a consequence of this fact, consistency can only be maintained if you use English for all names. In general, you should avoid any special characters inside the name of an element because you might run into trouble with the compiler and/or synthesis tool.
The identifiers can be written in lower and upper case, according to the rules in the following sections. Nevertheless, all user defined names must be unique even if they are converted to unicase. Otherwise you might have troubles with the compiler and/or synthesis tool.

3.1 (*) Signals, Variables, Constants, Types, Generics, Files

These elements should have names containing only the letters A-Z, a-z, _, and 0-9. Do not use any other characters. Writing identifiers (FSM states, constants, generics) in upper case makes it easier to identify user defined names in the code. Further, there are different prefixes and suffixes shown in Table 2 - obligatory usage and Table 3 - optional usage. They should be used to clearly identify the type of the user defined name.
It is recommended to use the same signal or generic name through the whole design hierarchy. This helps you identifying a signal or generic at a low level which has been set at the top level entity. Further, you should propagate every generic to the top because so you are able to configure the whole design in a centralised way.

description	extension	example
variable	prefix `v_`	`v_buffer`
clock	prefix `clk_`	`clk_system_i`
alias	prefix `a_`	`a_bit5`
constant	prefix `c_`	`c_LENGTH`
type definition	prefix `t_`	`t_mytype`
generics	prefix `g_`	`g_WIDTH`
file identifier	prefix `fd_`	`fd_romcontent`
low active signal or variable	suffix `_n`	`s_reset_n`
tri-stated signal or variable	suffix `_z`	`v_databus_z`
unit input signals	suffix `_i`	`clk_i`
unit output signals	suffix `_o`	`led_o`
unit bidirectional signals	suffix `_b`	`data_b`

Table 2: Obligatory extensions

description	extension	example
internal signal	prefix `s_`	`s_state`
asynchronous signal	suffix `_a`	`s_state_a`
delayed signal	suffix `_d`n	`pulse_d2`
counter signal	suffix `_c`	`seconds_c`
pulse signal	suffix `_p`n	`done_p1`

Table 3: Optional extensions

3.1.1 (x) Origin Based Naming Convention

If several entities are connected within a larger design, sometimes it makes sense to use this type of convention, because it eases reading and understanding of internal connections.

description	extension	example
process	prefix `p_`	`p_register: process(clk_i, reset_n_i)`
block	prefix `b_`	`b_global_signals: block`
loop	prefix `l_`	`l_sync_event: for nr in 3 downto 0 loop`
generate	prefix `gen_`	`gen_my_components: generate`
function	prefix `f_`	`function f_arctan (...)`
component	prefix `cmp_`	`cmp_my_component: my_component_name`
other labels	none	`important_command:`

Table 4: Label prefixes for process, blocks, etc.

The entity port names are then made up of the entity driving the port (source) and the further name representing the meaning of the port (`<entity>_<further port name>`). The suffixes for the direction (`_i, _o, _b`) this way are only needed for the top level ports connected to pins.
Further, the signals connecting entities are named like the ports. By using this naming convention, the ports and connecting signals are named identically throughout the whole design, which makes the code more readable. E. g. `cpu_reg_rden` or `uart_reg_data`.

3.2 (*) Blocks, Processes and other Labels

You are strongly encouraged to use labels for all sequential blocks in your code. This simplifies reading and analysing log files from the compiler and/or synthesis tool. Table 4 shows the extensions for blocks, processes and other useful labels. Note that large blocks are easier to read if the specific label name is not only used at the beginning of a statement (process, block, generate) but also in connection with the closing part of a code block. (e. g. end p_register;)

3.3 (*) Entity, Architecture, Configuration

The entity name (short name) should describe the functionality of the block. The name should not count more than 15 characters (letters A-Z, a-z, _ and 0-9) because it is reused in the name of a possible configuration. The full name of the entity should be in the comment (→ 2.5) above the entity declaration.
An architecture name should be “behaviour”, “structure”, or “rtl” (register transfer logic), according to description of the system behaviour. The three parts of an architecture (declaration, begin, end) should be marked with appropriate comments (→ 2.5).
A configuration always belongs to one entity and a specific architecture of this entity. Therefore, the name should represent this togetherness as it is shown in Table 5.

description	extension	example
architecture	none	`behaviour, structure, rtl`
entity simulator	suffix `_sim`	`my_entity_sim`
entity test bench	suffix `_tb`	`my_entity_tb`
configuration	<ent.>_<arch. >`_cfg`	`my_entity_rtl_cfg`

Table 5: Extensions for entities, architectures and configurations

description	extension	example
entity	none (_)	`my_entity_.vhd`
architecture rtl	`_rtl`	`my_entity_rtl.vhd`
architecture structure	`_struc`	`my_entity_struc.vhd`
architecture behaviour	`_behav`	`my_entity_behav.vhd`
package	`_pkg`	`mypackage_pkg.vhd`

Table 6: Extensions for files

3.4 (x) Files

File names should always start with the entity name of the corresponding unit, but for compatibility to different file systems, only lower case letters should be used for names. If you have different architectures in separate files you should add a suffix representing the architecture to the file name. For details about extensions for files please see Table 6.
Referencing external files within the code should only be done by giving the *relative* path. If not, you might experience problems compiling your code on another computer or even directory, which is often the case when multiple developers work together. Note that using relative path names is tricky with some tools. Some recognize the external file relative to the location of the code, including it, and others relative to the working directory of the tool.

3.5 (x) Directories

Directory structure of each hardware design project should separate files for source code, simulation, test bench, synthesis and implementation. Figure 7 shows required directories hierarchy for a common project.
If multiple testcases are used to simulate the design, the directory for simulation files (`sim/gate_sim` and `sim/rtl_sim`) should provide subdirectories called `TC<testcase_name>`.

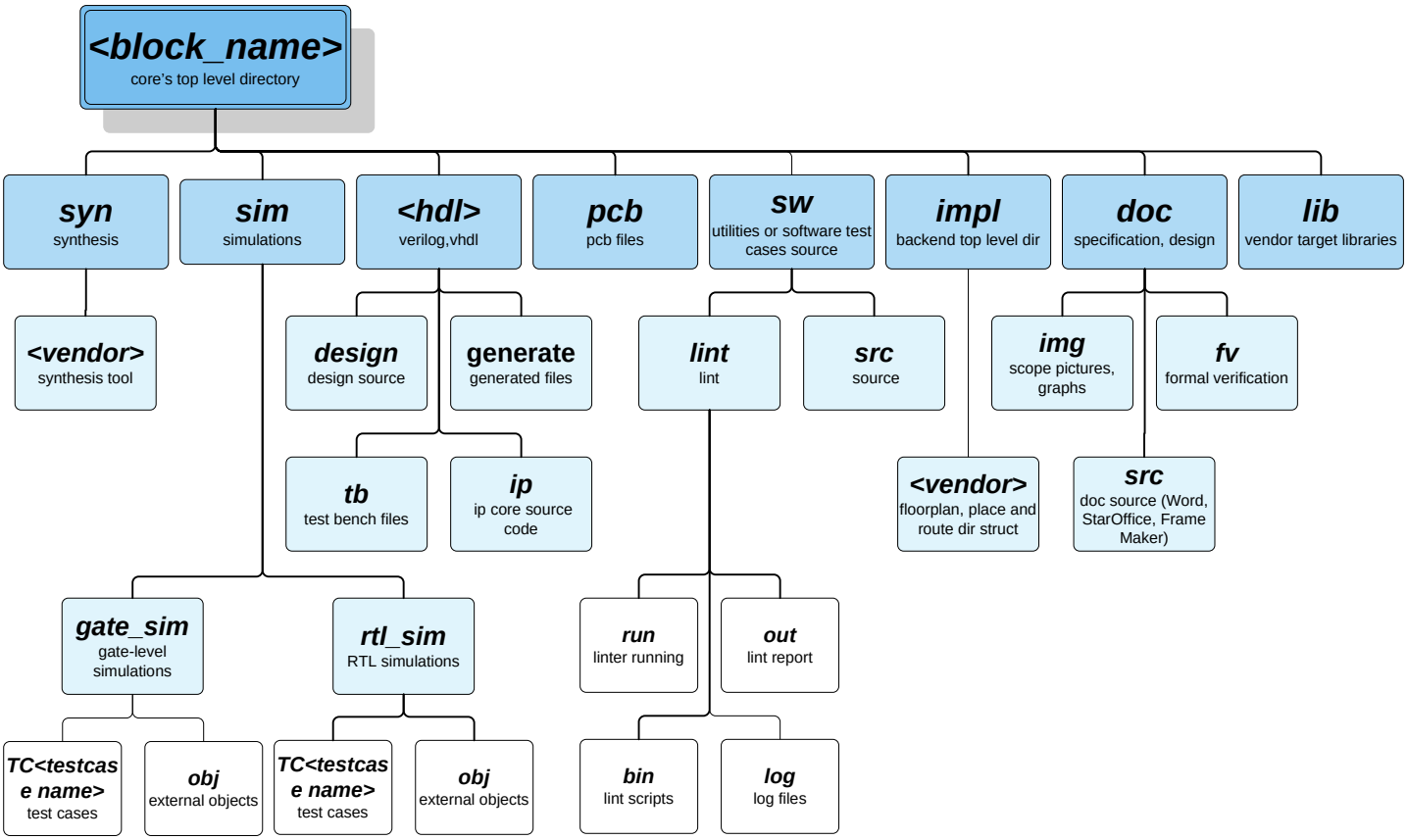

Figure 7: Project directories hierarchy

4 Coding Rules

4.1 (*) Notation for Bussed Signals

There is a need for consistent notation of bussed signals. Use only MSB to LSB notation (as shown in figure 8) for all bussed signals, variables and ports to avoid misinterpretation throughout the design flow.
There is another one, connected to this one. You should not mix “downto” and “to” for specifying the bus width through your design. In most cases you will have no problem just using “downto”.
To indicate that your signal, variable or port is a bus, you should always specify the range you want to use. Do this even if you use the full range, where it is not mandatory to clearly mark buses.

```
--=============================================================================
-- Entity declaration for my_entity
--=============================================================================
entity my_entity is
  port (
    -- global input signals
    clk_i     : in std_logic; -- local bus clock
    reset_n_i : in std_logic; -- reset =0: reset active
                              --       =1: no reset
    -- data bus(es)
    data_i    : in std_logic_vector (7 downto 0) -- data input
  );
end entity my_entity;

--=============================================================================
-- architecture declaration
--=============================================================================
architecture rtl of my_entity is

constant c_INITIAL  : std_logic_vector (2 downto 0) := "111"; -- initial value
variable v_register : std_logic_vector (7 downto 0);          -- stores data_i
```

Figure 8: Buses

4.2 (x) Bussed Ports Width Rules

You should not use buses of width one. First of all, it does not make much sense and second - you may get into troubles during placing and routing. Another problem is that buses of width one cannot be accessed with normal bus syntax in all cases. For details about possible issues please refer to [VHDL01] part 1 / 4.2.4.
Further, only compare buses of the same width, so that comparison always works properly.

4.3 (x) Top Level Module Structure

If your core is complex and has several submodules in hierarchy, it is recommended that the top level module is used for connectivity only, without any logic. It makes the design cleaner and gives an instant insight to major blocks.
Further, try to bring all memories and other hard blocks to the top level. If you need some glue logic, create separate module for that purpose.

4.4 (x) Component Instantiation

Components instantiations should be kept together, in the same part of the code, in order to have a better overview.
There are two ways to instantiate component. Either you choose to associate the ports by position or by name. Better solution is to use the latter (see figure 9) because you don't have to remember the exact sequence of the ports and you will get an error if you want to use a non-existent port. Therefore, it is more likely that the instantiated component is correctly connected.

declaration:

```
component my_buffer
  generic (
    g_DEFAULT : std_logic : = '1' -- reset value for buffer
  );
  port (
    input_i   : in  std_logic; -- signal to be double buffered
    output_o  : out std_logic; -- buffered version of the input signal
    clk_i     : in  std_logic; -- buffer clock
    reset_n_i : in  std_logic  -- buffer reset
  );
end component my_buffer;
```

instantiation:

```
cmp_nCS_buffer: my_buffer
   generic map (
     g_DEFAULT => '1'
   )
   port map (
     clk_i     => clk_i,
     reset_n_i => s_reset_n,
     input_i   => cs_n_i,
     output_o  => s_cs_n
   );
```

Figure 9: Component declaration and instantiation

4.5 (*) Reset for Sequential Blocks

Resets coming from bus lines should always be registered and deglitched. Use either a global synchronous or a global asynchronous reset for all sequential blocks. If there is no particular reason using the first is recommended. A reset is required because you need a defined state where you can begin your simulation or where your hardware starts working. This state is then the same for testing, debugging and simulation. Therefore you can rely on a consistent beginning which is not dependent on your compiler, simulator or synthesis tool. Figure 10 shows the template of two synchronous processes, one with synchronous and one with asynchronous reset.

synchronous

```
p_my_process: process (clk_i, reset_n_i)
begin
  if (clk_i'event and clk_i = '1') then
    if reset_n_i = '0' then
      -- reset to default
    else
      -- normal operation
    end if;
  end if;
end process p_my_process;
```

asynchronous

```
p_my_process: process (clk_i, reset_n_i)
begin
  if reset_n_i = '0' then
    -- reset to default
  elsif (clk_i'event and clk_i = '1') then
    -- normal operation
  end if;
end process p_my_process;
```

Figure 10: Sequential block reset

4.6 (*) One Statement per Line

Although the language allows you to write more then one statement per line (separated by ;) it is not advisable, because finding mistakes in your code is complicated. This is due to the fact that a compiler reference to a line number is no more unique, for a single statement.

4.7 (x) Finite State Machines

There is a recommended code structure for finite state machines. It splits the functionality into two processes. Therefore, it is easier to keep an overview and to combine Moore and Mealy designs in one state machine. In order to have a well coded state machine, you write it in three segments (declarative part and two main processes).

- You need to declare a signal which holds your current state first. Because the code readibility is better when using names instead of numbers for your states, using a symbolic encoded state signal is recommended. You can define encoding of these

states if you want it and it is also possible to change the encoding later, without much effort. Figure 11 shows how to declare such a signal.

- The conditional state transition are all in one synchronous process. According to the example in figure 12, for each state you can define conditions and a way of state changing by writing simple "if" or "case" statements and letting them assign the new state to the state signal.
- Very similar to the first process is the one for generating output signals. You can either directly assign your output signals or you can make them not only dependent on the state, but also on further conditions. This is the way how you can mix Moore and Mealy design into one state machine. Figure 13 shows a template for this process. You should always have "when others" in the case statement, which handles the possibility that the state machine might end up in a dead state.

```
-- SYMBOLIC ENCODED state machine: s_MY_STATE
-- ============================================
type t_my_state is (IDLE, STATE1, ... );
-- Add these two lines to force the encoding of the state machine by the synthesis tool
-- e.g., if you need the state encoding on a top level port
-- attribute ENUM_ENCODING : STRING;
-- attribute ENUM_ENCODING of t_my_state : type is "000 001 ...";
signal s_my_state : t_my_state; -- current state
```

Figure 11: Declarations for a finite state machine

4.8 (o) Input Double Buffers

If you are going to use a synchronous design, it is highly recommended to use double buffers to synchronise all asynchronous input ports. This avoids metastability and/or spikes on these signals. Figure 14 shows some code segments which you might want to use to get around these problems.

4.9 (o) Operator Precedence

Do not rely on the default operator precedence. Use brackets to specify the intended precedence in particular, which makes your code more readable and avoids misunderstanding.

```
--=============================================================================
-- Begin of My Finite State Machine
-- (state transitions)
--=============================================================================
-- read:  clk_i, s_reset_n
-- write:
-- r/w:   s_my_state
p_my_FSM_state: process (clk_i)
begin
  -- state transitions are always synchronous to the clock
  if (clk_i'event and clk_i = '1') then
    -- on synchronous reset in any state we jump to the idle state
    if (s_reset_n = '0') then
      s_my_state <= IDLE;
    else   -- there is no reset
      case s_my_state is

        -- state "IDLE"
        when IDLE =>
          -- conditional state transitions

        -- state "STATE1"
        when STATE1 =>
        ...

        -- all the other states (not defined)
        when others =>
          -- jump to save state (ERROR?!)
          s_my_state <= IDLE;
        end case;
     end if;
  end if;
end process p_my_FSM_state;
```

Figure 12: State transitions for a finite state machine

```
--================================================================================
-- Begin of My Finite State Machine
-- (output generation)
--================================================================================
-- read:  s_my_state
-- write:
-- r/w:
p_my_FSM_output: process (s_my_state)
begin
  case s_my_state is

    -- state "IDLE"
    when IDLE =>
      -- set output signals

    -- state "STATE1"
    when STATE1 =>
    ...

    -- all the other states (not defined)
    when others =>
      null;
  end case;
end process p_my_FSM_output;
```

Figure 13: Output generation for a finite state machine

```
--=============================================================================
-- Entity declaration for double buffer
--=============================================================================
entity double_buffer is
   generic (
     g_DEFAULT : in  std_logic := 0 -- reset value for internal buffers
   );
   port (
     clk_i     : in  std_logic; -- buffer clock
     reset_n_i : in  std_logic; -- reset for internal buffers
     input_i   : in  std_logic; -- input signal
     output_o  : out std_logic  -- double buffered output signal
   );
end entity double_buffer;

...
--=============================================================================
-- architecture declaration
--=============================================================================
architecture rtl of double_buffer is

signal s_buff1, s_buff2: std_logic; -- internal signal buffers (stage 1 and 2)

...
--=============================================================================
-- Begin of double buffer
--=============================================================================
-- read:  clk_i, reset_n_i, input_i, g_DEFAULT
-- write: output_i
-- r/w:   s_buff1, s_buff2
p_buffer: process (clk_i)
begin
  if (clk_i'event and clk_i = '1') then
    if reset_n_i = '0' then
      -- reset all internal buffers to default value
      s_buff1 <= g_DEFAULT;
      s_buff2 <= g_DEFAULT;
      output_o <= g_DEFAULT;
    else
      -- propagate signal through buffers (delay = 3 cycles)
      s_buff1 <= input_i;
      s_buff2 <= s_buff1;
      output_o <= s_buff2;
    end if;
  end if;
end process p_buffer;
```

Figure 14: Code segments for a double buffer

4.10 (x) The Value "don't care"

You should not use the value "don't care" because various synthesis tools may handle it in different ways. If you want to write technology and platform independent source code this cannot be accepted. Otherwise you also might have troubles because the "don't cares" may also lead to a mismatch between simulation results and the synthesised hardware. This makes formal verification more difficult.

4.11 (x) Internal Tri-State

Generally, avoid internal tri-state signals. They introduce increased power consumption, reliability problems (if tri-state line remains undriven) and requirement for an analog simulator (for timing simulation of distributed tri-state busses).
If still using it, requirements are:

- clearly structured layout
- one hot encoded enable signal
- bus keeper circuits

4.12 (*) Prefer IEEE 1076.3 over Synopsis Arithmetic Packages

In order to avoid vendor specific implementations, the use of the IEEE 1076.3 standardised packages instead of Synopsis arithmetic packages should be preferred.

```
use ieee.numeric_std.all;     -- ieee 1076.3
use ieee.std_logic_arith.all; -- synopsys
```

Example of usage is conversion between integer and std_logic_vector.
In IEEE.numeric_std package you can find `to_integer (ARG: signed, unsigned)` function. Combining it with unsigned or signed value of the vector, using `signed (ARG: std_logic_vector)` or `unsigned (ARG: std_logic_vector)` will result in the complete conversion.
Example: `to_integer (unsigned (ARG: std_logic_vector))`
The other way round works with the functions `to_unsigned (ARG, SIZE: natural)` and `to_signed (ARG: integer; SIZE: natural)`. The full conversion is done in combination with `std_logic_vector (ARG: signed, unsigned)`.
Example: `std_logic_vector (to_unsigned (ARG, SIZE: natural))`

4.13 (x) Signals and Variables - Usage and Declaration

Since changes of variable value are immediately adopted during a process (opposite to signals, which are updated after the process ends) and therefore can affect the functionality of the design, variables should be used when it is necessary to change their value more than once inside a single process cycle, but should not be used as registers. In that case, use signals instead.
Do not use default values (or initialization) for signals and variables, such an assignment can cause mismatch between synthesis and simulation. Use reset to initialize all signals and variables.
All signals or variables of type integer should be declared constrained, because standard is a 32-bit bus. If you don't want to use the full range in your design this would result in unnecessary use of resources on your hardware. Therefore, you should declare your integer signals, ports and variables always like that:

```
signal integer_signal :  integer range 0 to 16;
```

4.14 (*) Entity Port Types

If you plan to verify a certain entity using behavioural simulation on the one hand and gate level simulation on the other, it is necessary to declare all I/O signals as std_logic(_vector). This is required because the test bench for gate level simulations has to use bit lines. Consequently, the verification of both simulation models via one test bench (to avoid inconsistency) is not possible if some entity ports are not declared using elementary bit types.
Do not use buffer type ports to read output values within the code. Instead use type out and add another variable or signal and assign to it the same output value. This is because buffer type ports can't be connected to other types of ports, causing the buffer type to propagate throughout the entire design.

4.15 (o) Latches In Design

The usage of latches or flip-flops depends on the clocking scheme used. These two types of storage elements must not be used simultaneously within one design. Avoid mixing clocking schemes and thus use either latches or flip-flops only, whereas it is strongly recommended to prefer flip-flops.
The code to instantiate a transparent latch is shown in figure 15.

4.16 (o) VHDL Coding Standards - VHDL 93

Do not mix VHDL syntax constructs. VHDL 93 is not fully compatible with VHDL 87, so use VHDL 93 for the whole project.
Component declaration difference example is given in figure 16.

```
p_latch: process(en_i, data_i)
begin
  if (en_i = '1') then
    s_data <= data_i;
    -- If en_i = 0, then s_data keeps its old value,
    -- i.e. the latch is closed.
  end if;
end process p_latch;
```

Figure 15: VHDL Transparent Latch Instantiation

```
VHDL93:                          VHDL87:
component <name> is              component <name>
...                              ...
end component <name>;            end component;
```

Figure 16: VHDL Coding Standards Difference

4.17 (x) Readability, Reusability, Reliability - General Advices

- If an interface structure repeats in a design, use a record to represent it as a single signal. This reduces your code size and gives you the possibility to easily modify the interface.
- Whenever you find yourself coping and pasting some code, think of writing a function or procedure instead.
- Make use of constants and generics for buffer sizes, bus width and all other unit parameters. This provides more readability and reusability of the code.
- Try to use configuration to map entities, architectures and components (i. e. to define such mapping explicitly). That way tracing changes between different architectures can be in a single file. This can be useful to change simulation from high level to low level architectures.
- Define components and constants in a single package for each core.

5 Project Definition, Design, and Verification

5.1 Specification

Design behaviour should be described in a short and clear specification, which could be written by the verification engineer, design engineer or a third person, who should try to be as precise as possible in top level timings and protocols. The specifications should describe a testable design, avoiding special cases and usage of asynchronous clock domains if they are not really necessary.
Algorithms are not described, unless they are absolutely necessary for understanding of specifications.
Both - the verification and design engineer - should be able to arrive to the same behaviour, starting with design specifications.

5.2 RTL Design

Having a clear idea of the algorithms to be used in entities is the first prerequisite. You should implement algorithms in a way that best case=worst case, meaning that the processing time of algorithm should be fixed, no matter the inputs or state of process. This simplifies verification and time dependencies between processes. Exploiting parallelism and pipelining will increase speed of progression.
Clock boundaries should be defined and a single slow system clock should be used for most of the logic, while other domain clocks should only be used to clock peripherals.
Calculate memory size, the number of memory blocks and number of registers needed. Some algorithms require less memory size, but bigger number of memory blocks. It might be surprising when you realize that old, well known design does not fit in a brand new FPGA with just a few huge memory blocks.
Do not underestimate the memory size you need, but try to fit into a single, fairly big, FPGA. If this is not enough, use external memory.
RTL engineer should write his own testbenches as a design tool. These testbenches are highly recommended but are not considered as a part of the verification process.

5.3 Verification

The objective of verification phase is to guarantee the stability of the RTL design and its conformance with the specification requirements. Verification includes:

- Signals timing and protocols.
- Design predictability (including clock domain boundaries).
- 100 % code coverage.

If any of these goals is impossible to achieve, either the RTL design or the original specifications should be revised.

5.4 VHDL Synthesis Guidelines

VHDL for RTL should be as generic as possible. Stick to standard templates for combinational blocks (figure 17) and registered blocks (figure 10). Memory blocks can easily be inferred from VHDL templates with modern synthesis tools.

```
p_namecomb: process (input1, input2... inputn)
  variable v_var1 : var1type; -- optional
  variable v_var2 : var2type; -- optional
  ...
  variable v_varx : varxtype; -- optional

begin
  v_var1 := defaultvar1; -- mandatory if variable declared
  v_var2 := sig2;        -- mandatory if variable declared
  ...
  v_varx := defaultvarx; -- mandatory if variable declared

  ...  -- Combinational logic (any logic not including combinational feedbacks)

  sig1   <= v_var1;
  sig2   <= v_var2;
  ...
  sigx   <= v_varx;
end process;
```

Figure 17: Template for combinational blocks

Critical paths should be entirely included in a single design entity.

5.4.1 Instantiating IP Cores

IP cores may accelerate your design at the first stages and synthesize at much higher frequencies as any standard home made design. Nevertheless, experience has shown that when the design evolves, they often do not fully adapt to new requirements and a custom solution needs to be implemented.

5.4.2 Registering of Core's External I/Os

All core's external I/Os should be registered. It prevents long timing paths and allows you to meet timing constraints easier. It also allows easier verification of the entire SoC. Tri-State I/Os output enable line should also be registered.

5.4.3 Clock and Multiple Clock Domains

Design should be fully synchronous. Avoid asynchronous logic.
Try to stick to one single system clock. This simplifies the synchronisation and timing analysis. If you ever feel the necessity of clocking a part of your design with an extra clock domain, reduce this new domain to the minimum necessary. For example, if a VHDL custom CPU requires to serialize data clocked by an external source of unknown frequency, it makes sense to clock the serializer with the external clock and fixed frequency quartz as system clock. Data exchange should be done through a deterministic synchronization algorithm between the CPU and the serializer. This will result in much faster serializer and a robust CPU.
Clocks must not be "gated". If your design ever requires an internally generated clock use the FPGA clock manager.
Do not use both the rising edge and the falling edge of the same clock. Better think of using the FPGA clock manger to multiply your clock and distribute clock enable signals as required.

5.4.4 Use a Standard Entity for Memory Blocks

Keep a library with standard entity for memory blocks. If possible, try to use a standard VHDL template. It is also good practice to use the configuration when instantiating a memory block. In that case, if you ever want to change technology you just have to select a new architecture for the memory instantiation.

5.4.5 Memory Block Partitioning

Remember that FPGA's memory blocks are independent from each other. When you need to accelerate your algorithms, try to exploit the possibility of splitting your data into several independent blocks and do parallel simultaneous accesses to the memory.

5.5 VHDL for Simulation

5.5.1 Testbench Goals

VHDL testbench goals are:

- Signal timing and protocol verification.

  Testbench should check if all inputs compliant with the specification are understood by the UUT (Unity Under Test). Testbench must also check if UUT responds safely to non specified timings.

- Design predictability test (including clock domain boundaries).

  Whatever test pattern is UUT input, the output should conform to specifications. Any input not compliant with specifications should take the UUT to a specified known state.

- 100 % code coverage.

  Simulation should check correctness of every meaningful VHDL code line. RTL VHDL should be written in such a way that every line corresponds to one single statement. Either the specification or the RTL design should be revised if 100 % coverage is unreachable.

- Generating predictive test patterns.

5.5.2 Writing a Testbench

There are no hard rules for writing a testbench. Common sense is the best friend. Nevertheless, these advices may be helpful:

- Be creative.

  You are not restricted in your coding. Feel free to use records, variables, signal attributes, processes and functions.

- Base your testbench in procedures and functions.

  Identify basic operations in your UUT interface such us ReadData, WriteData etc. and write a procedure for each of them. Write abstract procedures based on your basic procedures to do meaningful tests for your UUT.

- Keep test procedures in a separate library.

  It gives you the possibility to reuse code easily.

- Write a pattern generator process for every group of signals.

- Use a rule checker process per timing rule.

  A rule checker process looks at a group of signals and a timing rule. Normally it is a good practice to write a procedure per rule and instantiate it in a single process.

- Write your own UUT behavioural model for design predictability test.

- Use non RTL constructs to speed up the design.

  If design specifications are good and RTL and behavioural models are correct, there should not be any difference between both models.

6 Using Documentation Generator - *doxygen*

Doxygen is a documentation system for several languages (in the Hardware Timing Project, intended to be used for VHDL documentation), which can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in LaTeX) from a set of documented source files. *Doxygen* is available for Linux, Mac OS X and Windows.
The operating system used to test the functionalities and configuration of *doxygen* for the present document is Windows XP Professional, it is possible to obtain slight differences in the outcome documentation if you are using other OS.
The installation of *doxygen* is widely documented in the project's website, please refer to [DXGN] if you have any doubt.
The documentation process is divided in three steps:

1. Documenting the VHDL code.
2. Configuring *doxygen*.
3. Running *doxygen* to generate the documentation.

6.1 Documenting the VHDL Code

For VHDL a comment normally starts with "--". *doxygen* will extract comments starting with:

- --!
- --! @DOXYGEN_COMMAND

There are only two types of comment blocks in VHDL: one line --! comment, representing a brief description, and multiline --! comment, where the --! prefix is repeated for each line.

```
--------------------------------------------------------------------------------
--! @brief Brief Description
--------------------------------------------------------------------------------
--! @details
--! Detailed Description
--------------------------------------------------------------------------------
entity my_entity is
  port (
    clk_i : in std_logic; --! in body description
  );
end entity my_entity;
```

Figure 18: Types of descriptions

The use of the --! @DOXYGEN_COMMAND will represent special parts of the documentation, and same commands will need input parameters to set properly their effects in the documentation.
For each code item there are three types of descriptions, as shown in Figure 18, which together form the documentation:

- Brief description
- Detailed description
- In body description

For the "In body Description", the comments are always located in front of the item that is being documented with one exception - **for ports, comment could also be placed after the item and it is then treated as a brief description for the port.** Whenever the code is commented without *doxygen* key characters "--!", it is possible to place a comment after the item (port or not) without effects in the documentation.
The document is organized in tags and subtags:

- Main Page - description of the project and general information.
- Related Pages - list of all related documentation pages, e. g. To-Do List.
- Design Unit Members:
  - Class List: a list of all design unit members with links to the Entities and Packages they belong to.
  - Design Unit Members: a list of all documented class members, with links to the class documentation for each member
- Files - list of all documented files with brief descriptions.

6.1.1 Main Page

Main Page purpose is customizing the index page in HTML or the first chapter in LaTeX. The most used commands are:

- **@mainpage [(title)]**

  If the @mainpage command is placed in a comment block, the title argument is optional and replaces the default title that *doxygen* normally generates.

- **@section [(title)] @subsection [(title)]**

  @section and @subsection are creating (sub)sections with names [(title)]. The title of the subsection should be specified as the second argument of the command.

- **@image [(title)] <file>["caption"] [<sizeindication>=<size>]**

  This command inserts an image into the documentation. The first argument specifies the output format. The second argument specifies the file name of the image. *doxygen* will look for files in paths (or files) that you specified after the IMAGE_PATH tag. The third argument is optional and can be used to specify the caption that is displayed below the image.

```
-----------------------------------------------------------------------------
--! @mainpage My Personal Index Page
-----------------------------------------------------------------------------
--! @section
--! Section Description
--! @subsection
--! Subsection Description
--! @image html file.jpg
--! @image latex file.jpg
-----------------------------------------------------------------------------
```

Figure 19: Comment block

Comment block shown in Figure 19 should be placed in the main file of the project.

6.1.2 Standard File Header

The structure for commenting a file is detailed in the Figure 20 and used commands are given in Table 7 (the command marked with "*" ends when a blank line or another sectioning command is encountered also in multiline description).

```
================================================================================
--! @file file_name.vhd
================================================================================
--! Standard library
library IEEE;
--! Standard packages
use IEEE.XXX.ALL;
use IEEE.XXX.ALL;
--! Specific packages
use work.XXX.ALL;
--------------------------------------------------------------------------------
-- --
-- company name, division and the name of the design --
-- --
--------------------------------------------------------------------------------
--
-- unit name: full name (shortname / entity name)
--
--! @brief <file content, behavior, purpose, special usage notes>
--! <further description>
--
--! @author <author name (email)>
--
--! @date <--\--\---->
--
--! @version <v.>
--
--! @details
--!
--! <b>Dependencies:</b>\n
--! <Entity Name,...>
--!
--! <b>References:</b>\n
--! <reference one> \n
--! <reference two>
--!
--! <b>Modified by:</b>\n
--! Author: <name>
--------------------------------------------------------------------------------
--! \n\n<b>Last changes:</b>\n
--! <date> <initials> <log>\n
--! <extended description>
--------------------------------------------------------------------------------
--! @todo <next thing to do> \n
--! <another thing to do> \n
--
--------------------------------------------------------------------------------
```

Figure 20: Structure for commenting a file

In absence of command for the next comment sections, we'll create the command appearance in the HTML and LaTex document:
--! Dependencies:\n
--! References:\n
--! Modified by:\n
--! \n\n Last changes:\n
doxygen inherits from HTML the text tag , where the text between the tags will be typed in bold, and the symbol \n forces a new line.

command	description
@file [<name>]	Indicates that a comment block contains documentation for a source file with name <name>.
@brief*	Starts a paragraph that serves as a brief description.
@author*	Starts a paragraph where one or more author names may be entered.
@date*	Starts a paragraph where one or more dates may be entered.
@version*	Starts a paragraph where one or more version strings may be entered.
@details	Starts the detailed description. You can also start a new paragraph (blank line), then the details command is not needed.
@todo	Starts a paragraph where a TODO item is described. The description will also add an item to a separate TODO list. The two instances of the description will be cross-referenced. Each item in the TODO list will be preceded by a header that indicates the origin of the item.

Table 7: Commands for commenting a file

6.1.3 Comments for Entities

As it has been said, the ports could be commented either in front of, or after the item, but be aware that the result in the documentation is different. The comment after the item is mandatory and represents the brief information in the documentation. Just in case additional information is needed, use the comment line in front of the item, it is even possible to extend the information to several lines.

```
--=============================================================================
--! Entity declaration for <long entity name of my_entity>
--=============================================================================
entity my_entity is
  generic (
    g_MYGENERIC : positive := 32 --! something which has to be configurable
  );
  port (
    -- global input signals
    --! Extend description of the local bus clock
    clk_i     : in std_logic; --! local bus clock
    reset_n_i : in std_logic; --! reset =0: reset active reset =1: no reset
    -- global output signals
    led_o     : out std_logic; --! LED =0: LED on LED =1: LED off
    -- data bus(es)
    data_b    : inout std_logic_vector (7 downto 0) --! data bus
  );
end entity my_entity;
```

Figure 21: Comments for entities

6.1.4 Comments for Architectures and Processes

The commenting in the architecture declaration follows the style of the entity declaration. In the process statements should be used commenting header, where the process is explained, but the statements shouldn't be commented with the purpose of documenting.

```
--============================================================================
-- ! architecture declaration
--============================================================================
architecture rtl of my_entity is
   --! Extended description of local clock \n
   signal s_lclk : std_logic; --! local clock
--============================================================================
-- architecture begin
--============================================================================
begin
--============================================================================
-- Beginning of my_process
--! Process <description>
--! read: clk_i, reset_n_i \n
--! write: s_lclk \n
--! r/w: led_o \n
--============================================================================
p_my_process: process (clk_i, reset_n_i)
begin
    if (clk_i'event and clk_i = '1') then -- synchronous process
       If reset_n_i = '0' then
          -- reset to default value
          led_o <= '1';
       else
           -- generate clock signal and LED output
           s_lclk <= not s_lclk; -- s_lclk now runs at half of clk_i
           if s_lclk = '1' then
              led_o <= '0' -- turn on LED if s_lclk is high
           end if;
       end if;
    end if;
end process p_my_process;

end architecture rtl;
--============================================================================
-- architecture end
--============================================================================
```

Figure 22: Comments for architectures and processes

6.2 Configuring *doxygen*

The executable doxygen is the main program that parses the sources and generates the documentation. Optionally, the executable doxywizard can be used, which is a graphical front-end for editing the configuration file that is used by *doxygen* and for running *doxygen* in a graphical environment.

6.2.1 Configure *doxygen* Using a *doxyfile*

doxygen uses a configuration file to determine all of its settings, *doxyfile*. Each project should get its own configuration file. If the *doxyfile* is already provided, it can be open (File/Open), and the settings will be loaded. These settings are only related to the information which *doxygen* will obtain from sources.

6.2.2 Configure *doxygen* From Scratch

- **Working directory**

  Working directory is the directory where all already commented sources are placed, from which we want to generate the documentation.

- ***doxygen* options**

  Configuration is available either via Wizard or Expert interface.

  - Wizard interface
    - Project:
      - Project name = TBD
      - Project version = TBD Versioning
      - Source code directory = <source_folder>
      - Destination directory = <doc_folder>
    - Mode:
      - Extraction mode = Document Entities only. Include cross referenced source code in the output.
      - Programming language = Optimized for VHDL output.
    - Output:
      - HTML → Plain Text
      - LaTeX → as intermediate format for PDF.
    - Diagrams: Use built-in class diagram generator
  - Expert interface
    - Project: FULL_PATH_NAMES = OFF
    - Build: GENERATE_TODOLIST = YES

* Source Browser: SOURCE_BROWSER = YES
* Input: IMAGE_PATH = path images folder
* HTML: GENERATE_TREEVIEW = FRAME

6.3 Run *doxygen*

This tag allows running documentation process, showing the configuration, saving and visualizing the output produced by *doxygen*. Documentation HTML and LaTeX files will be placed in defined <doc_folder>.

- HTML

  The HTML folder will contain the "index.html" and all other files generated by *doxygen*, which could be opened by Internet browser software to visualize the documentation.

- LaTeX

  The LaTeX folder will contain master file "refman.tex" and the rest of *.tex files that compose the document.

References

[ESA94] *R. Creasey, R. Coirault, P. Sinander*: VHDL Modelling Guidelines, ftp://ftp.estec.esa.nl/pub/vhdl/doc/ModelGuide.pdf, September 1994

[Cad00] *Gerhard R. Cadek*: HDL Coding Style Rules, http://agcad.ict.tuwien.ac.at/info/hdl_gesamt/hdlcoding/hdlcoding.htm, 10 September 2000

[OC01] *Yair Amitay, Jamil Khatib, Damjan Lampret*: OpenCores Coding Guidelines, http://www.opencores.org/cvsweb.shtml/common/opencores_coding_guidelines.pdf, 24 October 2001

[Kha00] *Jamil Khatib*: VHDL Coding Style, http://www.opencores.org/OIPC/projects/vhdl_style.html, 2000

[VHDL01] *comp.lang.vhdl*: Frequently Asked Questions And Answers, http://vhdl.org/comp.lang.vhdl, 3 November 2001

[COM10] *Peter Chambers*: The Ten Commandments of Excellent Design-VHDL Code Examples, http://www.bawankule.com/verilogcenter/files/10_2.pdf

[CERN] *Pablo Alvarez Sanchez*: VHDL guidelines, https://espace.cern.ch/ab-co-timing/hw-project/VHDL%20guidelines/Home.aspx

[DXGN] *doxygen* website, www.doxygen.org