1 VHDL STYLE GUIDE ......................................................................................................................... 3
   1.1 INTRODUCTION ............................................................................................................................ 3
   1.2 VHDL STRUCTURE .......................................................................................................................... 3
       1.2.1 Entity ....................................................................................................................................... 3
       1.2.2 Architecture ........................................................................................................................... 3
       1.2.3 Configuration ........................................................................................................................ 4
       1.2.4 Packages ............................................................................................................................... 4
2 NAMING CONVENTIONS ....................................................................................................................... 5
   2.1 GENERAL ........................................................................................................................................ 5
   2.2 SIGNALS, VARIABLES AND CONSTANTS .................................................................................. 5
   2.3 ENTITIES AND GENERICS ............................................................................................................ 5
   2.4 ARCHITECTURES .......................................................................................................................... 5
   2.5 CONFIGURATIONS ....................................................................................................................... 5
   2.6 TYPES AND SUBTYPES ............................................................................................................... 6
   2.7 PROCESSES .................................................................................................................................. 6
   2.8 FUNCTIONS AND PROCEDURES .................................................................................................. 6
   2.9 PACKAGES .................................................................................................................................... 6
3 COMPONENTS ...................................................................................................................................... 7
   3.1 STATE MACHINES .......................................................................................................................... 7
   3.2 SYNCHRONOUS COUNTERS AND REGISTERS ......................................................................... 9
   3.3 MULTIPLEXERS ............................................................................................................................. 10
   3.4 COMPLEX LOGIC .......................................................................................................................... 11
   3.5 TRI-STATE BUFFERS ....................................................................................................................... 11
4 PROJECT MANAGEMENT ................................................................................................................ 12
   4.1 FILES AND LIBRARIES ............................................................................................................... 12
   4.2 DIRECTORY STRUCTURE .............................................................................................................. 12
   4.3 SOURCE CODE CONTROL SYSTEM ............................................................................................ 13
5 SYNTHESIZABLE CODE ..................................................................................................................... 14
6 RESERVED KEYWORDS ................................................................................................................... 15
7 REFERENCES AND LITERATURE ..................................................................................................... 17
   7.1 INTERNET RESOURCES .............................................................................................................. 17
   7.2 IMS ENVIRONMENT SETTINGS .................................................................................................... 17
8 REVISION HISTORY ........................................................................................................................ 18
1 VHDL Style Guide

1.1 Introduction

1.2 VHDL Structure

Figure 1.1 shows an example of a hierarchical design. It consists of one top entity that describes the behavior of the whole system. The top level entity consists of a number of components, that are entities on a lower level.

Figure 1.1 : Hierarchical design

1.2.1 Entity

An entity represents a piece of hardware with well-defined in- and outputs that are defined in the port clause. The entity performs the function that is defined in the architecture belonging to that entity. Furthermore, some parameters can be associated with the entity, e.g. the bus width of an address bus. These parameters are defined in the generic clause.

1.2.2 Architecture

An architecture describes the function of an entity. Three kind of descriptions are possible:

- Algorithmic or Behavioral descriptions
- Register Transfer Level (RTL) descriptions
- Gate Level descriptions
The first category can be used in order to describe the behavior of the design on a high level. In general, this description can not automatically be synthesized. The RTL description takes the clock period in account and is a lower level description of the design, that can be synthesized automatically. The gate level description is the lowest description level. Describing a design with this kind of description is a time-expensive and error-sensitive design method. During a design process, it is possible to describe the entity first on an algorithmic level and if the algorithm proved to be correct, an RTL description can be made for the same entity. In this case, two different architectures are available for the same entity.

1.2.3 Configuration
During simulation, there can be the need for simulating parts of the design on an RTL level, whereas other parts are simulated on the algorithmic level. Using the configuration, the user can select which of the architectures that belong to one entity shall be used to simulate that entity. Figure 1.2 shows this graphically.

![Figure 1.2: Selection of one of three architectures for entity](image)

1.2.4 Packages
A package contains all user-defined types that are being used in entities and architectures. Also procedures and functions that are called by an architecture should be put into a package. A design can contain several different packages. If however, no user-defined types nor procedures nor functions exist for a certain design, there is no need to create a package for the design.
2 Naming conventions

2.1 General
1. All documentation, comments and identifiers should be in English
2. The source code must not exceed 80 characters per line
3. Each line must contain only one VHDL statement
4. Each line must be properly indented (2 spaces for each level of indentation)
5. No TAB characters should be used for indentation
6. All source code has to be written in lower-case, except for constant declarations
7. VHDL93 language constructs that are not in the VHDL87 language shall not be used

2.2 Signals, Variables and Constants
1. Signal names should be as short as possible, not exceeding 24 characters
2. Active-low signals should be indicated by prefixing “n_” to the signal name
3. Each signal that is declared should be commented
4. Signals that are register outputs should be indicated by suffixing “_q” to the signal name
5. Signals that are register inputs should be indicated by suffixing “_in” to the signal name
6. Signals that are latch outputs should be indicated by suffixing “_lt_q” to the signal name
7. Signals that are latch inputs should be indicated by suffixing “_lt_in” to the signal name
8. If the signal is a vector, the range is defined as: “msb downto lsb”
9. The same rules are valid for variables
10. Constants should be written in upper-case

2.3 Entities and Generics
1. The function of the entity should be commented
2. If generics are present, their function should be commented
3. The function of each port should be commented
4. The ports can only be one of the following three types: “in”, “out” or “inout”
5. The ports should be ordered by their function
6. The ports of the top-entity can only be of type std_logic or std_logic_vector

2.4 Architectures
1. The architecture name is derived from the entity name by suffixing it with “_arc”
2. If more than one architecture exists for an entity, the architecture name is suffixed with the number of the architecture
3. Each port map and generic map should use named association, with each line containing only one port or generic
4. It is not advised to mix component instantiations with processes in an architecture; an architecture should either contain components or processes

2.5 Configurations
1. A configuration should be defined which explicitly defines one of the existing architectures
2.6 Types and Subtypes
1. Index ordering in self-defined types should be **msb downto lsb**
2. Self-defined types (except for state-machine states) should be defined in a library

2.7 Processes
1. Each process has to be labeled and should be closed with the same label
2. The function of the process should be commented
3. **All signals** that are **read** in a process have to be listed in the sensitivity list, except for the "clock"-process, that is used for implementing registers. In this case, the inputs for the registers should be omitted.
4. Use parentheses in complex expressions in order to avoid ambiguities

2.8 Functions and Procedures
1. If functions and procedures are defined for a design, these should be put into one package

2.9 Packages
1. All defined types that are defined in the package should be commented
2. All procedures should be described in the package, including limitations
3 Components

3.1 State Machines

- Use two processes for a state machine, one containing state transition conditions and outputs and the other containing the Flip-flop (FF)-part.
- Documentation of the state machine should consist of a graphical bubble chart that shows all states, state transition conditions and all output signals of the state machine. If there is not enough room for text that contains the conditions and output signals, two tables should be made. The first table consists of references to the different state transitions and the conditions that are valid for that transition. The second table consists of state names and the values of all outputs in that state.
- If possible, large bubble charts should be split up into several smaller ones. This should be indicated in the bubble charts very carefully.

<table>
<thead>
<tr>
<th>State</th>
<th>mealy_out</th>
<th>more_out</th>
</tr>
</thead>
<tbody>
<tr>
<td>idle</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>state_a</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>state_b</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>state_ab</td>
<td>a &amp; b</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 3.1 : Block diagram and output signals for state machine

Figure 3.2 : State diagram of state machine


```vhdl

-- Define all states

type state is (idle, state_a, state_b, state_ab);

-- Signals used for states

signal current_state, next_state : state;

-- State transitions and outputs process

trans_out_process: process (current_state, a, b)

begin

    more_out  <= '-';               -- Set default values
    mealy_out <= '-';

    transition_condition := b & a;  -- Merge signal a and b to one signal

    case current_state is
        when idle =>
            more_out  <= '0';           -- Define outputs in idle state
            mealy_out <= '0';

        case transition_condition is
            when "00"    =>
                next_state <= idle;
            when "01"    =>
                next_state <= state_a;
            when "10"    =>
                next_state <= state_b;
            when others =>
                next_state <= state_ab;

        end case;

    when state_a =>

        more_out  <= '1';           -- Define outputs in state state_a

        case transition_condition is
            when "00"    =>
                mealy_out  <= '0';
                next_state <= idle;
            when "01"    =>
                mealy_out  <= '0';
                next_state <= state_a;
            when "10"    =>
                mealy_out  <= '1';
                next_state <= state_b;
            when others =>
                mealy_out  <= '1';
                next_state <= state_ab;

        end case;

end process;
```

when state_b =>  
  mealy_out <= '1';  -- define outputs in state state_b  
  case transition_condition is  -- define transitions  
    when "00" =>  
      next_state <= idle;  
    when "01" =>  
      next_state <= state_a;  
    when "10" =>  
      next_state <= state_b;  
    when others =>  
      next_state <= state_ab;  
  end case;  
when state_ab =>  
  more_out <= '1';  -- define outputs in state state_ab  
  case transition_condition is  -- define transitions  
    when "00" =>  
      mealy_out <= '0';  
      next_state <= idle;  
    when "01" =>  
      mealy_out <= '1';  
      next_state <= state_a;  
    when "10" =>  
      mealy_out <= '0';  
      next_state <= state_b;  
    when others =>  
      mealy_out <= '0';  
      next_state <= state_ab;  
  end case;  
end case;  
end process trans_out_process;  
ff_process:  -- flip-flop definition part; thus current_state  
process (clk, n_reset)  -- not in sensitivity list.  
begin  
  if (n_reset='0') then  
    current_state <= idle;  
  else  
    if (clk='1') and (clk'event) then  
      current_state <= next_state;  
    end if;  
  end if;  
end process ff_process;  
end state_machine_arc;  

3.2 Synchronous Counters and Registers

- Use two processes for counters and registers, one containing the FF-part and the  
  other containing the new register or counter values

![Diagram of counter]

- **counter**
  - **counter_value**
  - **n**
  - **restart**
  - **enable**
  - **clk**
  - **n_reset**
next_counter_value_process: -- determines the new counter value
process (count_q, restart, enable)
  variable add_result : std_logic_vector(n-1 downto 0); -- dummy variable
begin
  count_in <= count_q;                    -- default next value
  add_result := unsigned(count_q) + '1';  -- calculates counter + 1
  if (restart='1') then                   -- reset counter synchronously
    count_in <= (others => '0');
  else
    if (enable_counter = '1') then       -- enables counting
      count_in <= add_result;            -- increase counter
      end if;
    end if;
  end if;
end process next_counter_value_process;

ff_process : -- stores the new counter value at rising clock (Flip-flops)
process (clk, n_reset)
begin
  if (n_reset='0') then              -- during asynchronous reset, the
    count_q <= (others => '1');      -- counter is set to maximum value
  else
    if (clk='1') and (clk'event) then
      count_q <= count_in;
    end if;
  end if;
end process ff_process;

3.3 Multiplexers

![Multiplexer Diagram](image)

Figure 3.4 : Block diagram of multiplexer

mux_process: -- if select equals 1, d_1 is routed to the output of the mux
process (sel, d_0, d_1)
begin
  case sel is
    when '1' =>
      d_out <= d_1;
    when others =>
      d_out <= d_0;
  end case;
end process mux_process;
3.4 Complex Logic

- It is recommended to use the case-statement in order to keep an overview of the complex logic function; writing complex logic as if-then-else or using and-, or- and not-constructs is valid, too.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>f_abc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

```vhdl
process (a, b, c)
variable tmp : std_logic_vector(2 downto 0);  -- dummy signal
begin
  tmp := a & b & c;       -- merge signal a, b and c to dummy signal
  case tmp is             -- implements truth table
    when "001" =>
      d <= '1';
    when "010" =>
      d <= '-';
    when "110" =>
      d <= '1';
    when "111" =>
      d <= '-';
    when others =>
      d <= '0';
  end case;
end process;
```

3.5 Tri-state Buffers

```
tri_state_process:
process (n_enable, data_in)
begin
  if (n_enable='0') then    -- if low-active enable input becomes active
    data_out <= data_in;         -- input data is routed to output
  else
    data_out <= (others => 'Z'); -- otherwise output is tri-stated
  end if;
end process tri_state_process;
```

Figure 3.5: Tri-state buffer with active-low enable input
4 Project Management

4.1 Files and Libraries
1. Each file shall contain a header (using VHDL comments : -- ), containing following data:
   • File name
   • Date of creation
   • Author, including full address
   • Name of the design units in the file
   • List of known model limitations and errors, if any
   • Change list, containing date, author and a description of all changes made
   • Disclaimer
   • Copyright
   At IMS, these standard headers are available as templates, which can be obtained with the command templates, which is available in the vhdladm (VHDL administration) environment
2. Entities and architectures have to be included in one file only.
3. File names must equal the entity name, suffixed with ".vhd"
4. All configurations for the design should be listed in a single file

4.2 Directory Structure
The following directory structure should be made when making the project environment:

```
vhdl
  └── source
      └── testbench
  └── synth
```

The directory source contains all VHDL code that will be synthesized.
The directory synth contains the VHDL code that was generated by a synthesis tool.
The functionality of this code should be the same as the VHDL source-code.
The directory testbench contains the testbench with which the source-code is tested.
The testbench is also used in order to check whether the behavior of the synthesized VHDL code is the same as the behavior of the source-code.
4.3 **Source Code Control System**

It is strongly recommended to use a source code control system during the development of VHDL source files.

The Concurrent Version System (CVS) is extremely suited for managing all VHDL-files simultaneously. An important advantage of CVS is that several engineers can work on the same design simultaneously. CVS is freeware and can be obtained at http://www.cvshome.org/.

CVS is based on a central archive, in which projects are stored. Each designer can obtain a local copy of a project and work with the local copy. After that all necessary changes to the code have been made, these changes are transferred into the central archive and are from that time on available to other designers. CVS can notify all designers of changes in the central archive, so that designers can update their local copy of the project.

The first steps for using CVS are:

- Set the environment variable `$CVSROOT` that contains the path to the central CVS archive, in which all projects are being stored.
- Import existing project files into a new archive using the command `cvs import -m "message text" <Project Name> <Your Name> start`. Now a new central archive named `<Project Name>` is created.
- Anyone who wants to work on the project needs to checkout the project `<Project Name>` using the command `cvs checkout <Project Name>`. The project is extracted in the directory from which this command is called.
- After that changes have been made, these changes are committed to the central archive using the command `cvs commit -m "message text"` from the directory `<Project Name>`, in which the local copy of the project is located. *It is strongly recommended to update the project, before committing any changes.*
- Updating the project is done using the command `cvs -q update`.

For further information about available CVS commands, refer to the CVS manual "Cederqvist", which can be downloaded from the homepage www.cvshome.org.
5 Synthesizable Code

- All VHDL should be written on the Register Transfer Level (RTL), which means that all operations in the design are scheduled by a predefined clock cycle.
- For logic signals, only the std_logic and std_logic_vector are allowed
- Don't use the `wait`-statement nor the `after`-statement in a design
- Don't assign initial values to signals
- All signals that are being read in a process, should be listed in the sensitivity list of that process, except for clock processes
6 Reserved Keywords

The following keywords are reserved for VHDL and/or Verilog:

abs access after alias all always
always and architecture array assert assign
attribute begin block body buf buffer
bufif0 bufif1 bus case casex casez
cmos component configuration constant deassign default
defparam disable disconnect downto edge else
elsif end endcase endfunction endmodule endprimitive
describe endtable endtask entity event exit
file for force forever fork function
generate generic group guarded highz0 highz1
if ifnone impure in inertial initial
inout input integer is join label
large library linkage literal loop macromodule
map medium mod module nand negedge
new next nmos nor not notif0
notif1 null of on open or
others out output package parameter pmos
port posedge postponed primitive procedure process
pull0 pull1 pulldown pullup pure range
rcmos real realtime record reg register
reject release rem repeat report return
rmos rol ror rmos rtran rtranif0
rtranif1 scalared select severity shared signal
sla sll small specify specparam sra
sr1 strength strong0 strong1 subtype supply0
supply1 table task then time to
tran tranif0 tranif1 transport tri tri0
tril triand trior trireg type unaffected
units until use variable vectored wait
wand weak0 weak1 when while wire

The following keywords are reserved for XILINX:

A4 DECODE* IFD* NOR_n RAM_r TMS
AA DP ILD* OBUF_r RDBK VCC
AB FDC INV OFD_r RDCLK WAND_n
AND* FDCE IOB Or_n READBACK WOR2AND
BEL FDP IOBUF_r OSC_r ROM_r XOR_n
BSCAN FDPE M0 P_n RST
BUF* FMAP* M1 PAD_n STARTUP
CCLK *_FLAG M2 PULLDOWN TCK
CLB GND MD* PULLUP TDI
COMP IBUF* NAND_n R_nC_n TD0
For more XILINX-specific reserved keywords, see [1]
7 References and Literature

[1]:  XSI user guide appendix B, section XC4000/A/D/H primitives and macros

[2]:  R.Creasey, R.Coirault (European Space Agency). VHDL Modeling Guidelines, September 1994

7.1 Internet resources

http://www.doulos.com
http://www.eda.org
http://rassp.scra.org
http://www.vhdl-online.de
file:/gf/html/vhdl/vhdl.htm (IMS-Info system)
http://www.cvshome.org (CVS, Cederqvist)

7.2 IMS Environment Settings

Add the following lines into your .cshrc, if you’d like to use CVS, the templates command:

```
# VHDL Settings
setenv CVSROOT <Your CVS Archive Path>
source -vhdladm/environ.csh
```

You can use the command `vhdlguide` to get the latest version of this VHDL styleguide
### 8 Revision history

<table>
<thead>
<tr>
<th>Date</th>
<th>Author</th>
<th>Version</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>03.02.1999</td>
<td>C.Scherjon</td>
<td>1.0</td>
<td>Document created</td>
</tr>
<tr>
<td>01.10.2000</td>
<td>C.Scherjon</td>
<td>1.1</td>
<td>Changes in sections 2.1, 2.2, figure 3.2, section 4.1 and chapter 6</td>
</tr>
<tr>
<td>05.02.2003</td>
<td>C.Scherjon</td>
<td>1.2</td>
<td>Changes in VHDL examples regarding register instantiation, clock processes sensitivity list. Rewrote chapter about source code control; description of CVS instead of SCCS.</td>
</tr>
</tbody>
</table>