VHDL Coding Rules

Tampere University of Technology
Department of Computer Systems

Version 4.4 – Jan 2009

Index

- Purpose
- Summary of rules and guidelines
- Rules – 11 topics
- Guidelines – 14 topics

- Extra slides
  - Few additions to rules and guidelines

Purpose of VHDL Coding Rules

- Prevent harmful or unpractical ways of coding
- Introduce a common, clear appearance for VHDL
- Increase readability for reviewing purposes
- Not to restrict creativity in any way

- Bad example:

```
A_37894 :process(xR,CK ,datai , DATAO )
BEGIN
if(XR ='1' )THEN DATAO<= "1010";end if;
if(CK'event) THEN if CK = '1' THEN
for ARGH in 0
to 3 Loop DATAO(ARGH) <=datai(ARGH);
end Loop;end if;
end process;
```

About Coding Rules

- This guide has
  1. Rules
  2. Guidelines
- Both are first listed shortly and explained later in detail

Rules (1)

1. Entity ports
   - Use only modes IN and OUT with
   - names have suffixes _in or _out
   - Only types STD_LOGIC and STD_LOGIC_VECTOR
   - Use registered outputs
2. A VHDL file and the entity it contains have the same name
   - One entity+architecture per file
3. Every entity has a testbench
4. Synchronous process
   - always sensitive only to reset and clock
   - clock event is always to the rising edge
   - all control registers must be initialized in reset

Rules (2)

5. Combinatorial process's sensitivity list includes all signals that are read in the process
   - Complete if-clauses must be used. Signals are assigned in every branch.
6. Use signal naming conventions
7. Indexes of STD_LOGIC_VECTOR are defined as DOWNTO
8. Use named signal mapping in component instantiation, never ordered mapping
9. Avoid of magic numbers, use constants or generics instead
10. Use assertions
11. Write enough comments

Guidelines (1)

1. Every VHDL file starts with a header
2. Indent the code, keep lines shorter than 76 characters

3. Use descriptive names.
4. Use conventional architecture names
5. Label every process and generate-clause
6. Clock is named `clk` and async. active-low reset `rst_n`
7. Intermediate signals define source and destination blocks
8. Instance is named accroding to entity name
9. Use `FOR GENERATE` for repetitive instances
10. Guidelines for naming conventions

11. Prefer generics to package constants
12. Avoid assigning bit vector literals. Use conversion functions
13. Prefer arrays over multiple signals. Use loops.
14. Avoid variables inside processes

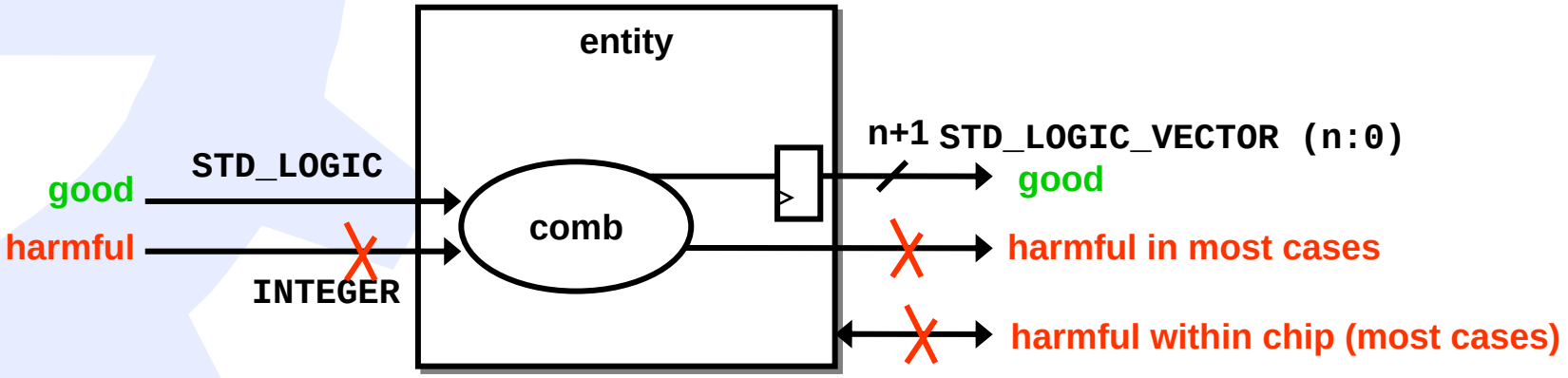
Guidelines (2)

- Avoid mixing of different coding styles (register transfer level, structural, gate level)
- Use correct spacing. Align colons and port maps
- Declare signals in consistent groups

Rules you cannot refuse

Entity ports

- Use only modes `IN` and `OUT` in the port
  - Names have corresponding post-fixes
- Use only signal types `STD_LOGIC` and `STD_LOGIC_VECTOR` in the ports
- Output of a block should always come **directly from a register**



Use only port modes `IN` and `OUT`.

Use only types `STD_LOGIC` and `STD_LOGIC_VECTOR`.

File contents and naming

- One VHDL file should contain one entity and one architecture, file named as `entityname.vhd`
- Package name should be `packagename_pkg.vhd`
- Test bench name should be `tb_entityname.vhd`

A VHDL file and the entity it contains have the same name.

Testbench

- Each entity requires at least one testbench
  - Design without a testbench is useless
- Prefer self-checking testbenches
  - Cannot assume that the verifier looks at the “magic spots” in waveform
  - (Occasionally, TB just generates few inputs to show the intended behavior in common case)
- Informs whether the test was successful or not
- There can be several test benches for testing different aspects of the design
  - Code coverage should be as high as possible
  - Verify correct functionality with different generic values

Every entity needs a testbench.

Sequential/synchronous process

- Sensitivity list of a synchronous process has always exactly two signals
  - Clock, rising edge used, named `clk`
  - Asynchronous reset, active low, named `rst_n`
- Signals that are assigned inside sync process, will become D-flip flops at synthesis
- Never give initial value to signal at declarative part
  - It is not supported by synthesis (but causes only a warning)

```
SIGNAL main_state_r : state_type := "11110000";
```

  - ➔ Assign values for control registers during reset
  - (Xilinx FPGAs may be exceptions to this rule)

Synchronous process is sensitive only to reset and clock.

Sequential/synchronous process (2)

- Correct way of defining synchronous process:

```
cmd_register : PROCESS (rst_n, clk)
BEGIN
   IF (rst_n = '0') THEN
      cmd_r <= (OTHERS => '0');
   ELSIF (clk'EVENT AND clk = '1') THEN
      cmd_r <= …;
   END IF;
END PROCESS cmd_register;
```

Clock event is always to the rising edge.

Assign values to control registers during reset.

Combinatorial/asynchronous process

- An asynchronous process must have all input signals in the sensitivity list
  - If not, simulation is not correct
  - Top-3 mistake in VHDL
  - Input signals are on the right side of assignment or in conditions (`if`, `for`, `case`)
  - Automatic check: `vcom -check_synthesis`
- If-clauses must be complete
  - Cover all cases, e.g. with `else` branch
  - All signals assigned in every branch
  - Otherwise, you'll get latches (which are evil)

Include all input signals of combinatorial process in the sensitivity list.

Combinatorial/asynch. process (2)

- An example of an asynchronous process:

```
decode : PROCESS (cmd_r, bit_in, enable_in)
BEGIN
   IF (cmd_r = match_bits_c) THEN
      match_0       <= '1';
      IF (bit_in(1) = ‘1’ and bit_in(0) = ‘0’) THEN
         match_both <= enable_in;
      ELSE
         match_both <= '0';
      END IF;
   ELSE -- else branch needed to avoid latches
      match_0       <= '0';
      match_both    <= '0';
   END IF;
END PROCESS decode;
```

- Same signal cannot be on both sides of assignment in combinatorial process
  - That would create combinatorial loop, i.e. malfunction

Combinatorial process necessitates complete if-clauses. Every signal is assigned in every if-branch.

These naming conventions are must

- General register output `signalname_r`
- Combinatorial signal `signalname`

- Input port `portname_in`
- Output port `portname_out`

- Constant `constantname_c`
- Generic `genericname_g`
- Variable `variablename_v`

Use these naming conventions.

Signal types

- Direction of bits in `STD_LOGIC_VECTOR` is **always** `DOWNTO`
- Size of the vector should be parameterized
- Usually the least significant bit is numbered as zero (not one!):

```
SIGNAL data_r : STD_LOGIC_VECTOR(datawidth_g-1
DOWNTO 0);
```

- Use package `numeric_std` for arithmetic operations

Direction of bits in a bus is always DOWNTO.

Named signal mapping in instantiations

- Always use named signal mapping, never ordered mapping

```
i_datamux : datamux
 PORT MAP (
    sel_in   => sel_ctrl_datamux,
    data_in  => data_ctrl_datamux,
    data_out => data_datamux_alu
    );
```

- This mapping works even if the declaration order of ports in entity changes

Always use named signal mapping in component instantiations, never ordered mapping.

Avoid magic numbers

- Magic number is an unnamed and/or ill-documented numerical constant
- Especially, a magic number derived from another is catastrophic
  - Eg. for-loop iterates through 2 to 30 because signal is 32b wide. What if it is only 16 bits?
- Use constants or generics instead

```
STD_LOGIC_VECTOR (data_width_g -1 DOWNTO 0) --generic
STD_LOGIC_VECTOR (data_width_c -1 DOWNTO 0) --constant
```

- States of FSM are enumerations not bit vectors
- Note that this document occasionally uses magic numbers to keep examples short

Use constants or generics instead of magic numbers.

Use assertions

- Easier to find error location
- Checking always on, not just in testbench
- Assertions are not regarded by synthesis tools ➔ no extra logic

```
assert (we_in and re_in)=0
report ”Two enable signals must not active
at the same time”
severity warning;
```

- If condition is not true during simulation,
  - the report text, time stamp, component where it happened will be printed
- Ensure that your initial assumptions hold
  - e.g. data width is multiple of 8 (bits)

Use assertions.

Comment thoroughly

- Comment the intended function
  - Especially the purpose of signals
  - Not the VHDL syntax or semantics
  - Think of yourself reading the code after a decade.
- A comment is indented like regular code
  - A comment is placed with the part of code to be commented.
- Be sure to update the comments if the code changes.
  - Erroneous comment is more harmful than not having a comment at all

Pay attention to comments

Guidelines

Include file header

- Every VHDL file starts with standard header
- Example

```
-------------------------------------------------------
-- Project : project or course name
-- Author : Aulis Kaakko (,student number)
-- Date : 2007-30-11 14:05:01
-- File : example.vhd
-- Design : Course exercise 1
-------------------------------------------------------
-- Description : This unit does function X so that…
-------------------------------------------------------
-- $Log$
-------------------------------------------------------
```

Every VHDL file starts with a standard header.

General code appearance

- VHDL code must be indented
  - Much easier to read
- Indentation is fixed inside a project
  - Comment lines are indented like regular code
- In (X)Emacs VHDL mode, use
  - `Ctrl-c Ctrl-b` to beautify buffer
  - `Ctrl-c ctrl-a Ctrl-b` to align buffer
- Maximum length of a line is 76 characters
  - In VHDL language it is very easy to divide lines
  - The commented code line should still fit to the console window
- Use blank lines to make code more readable

Use indentation. Keep lines shorter than 76 characters.

Naming in general

- **Descriptive, unambiguous names are very important**
- Names are derived from English language
- Use only characters
  - alphabets `'A' .. 'Z','a' .. 'z',`
  - numbers `'0' .. '9'` and underscore `'_'`.
  - First letter must be an alphabet
- Use enumeration for coding states in FSM
  - Do not use: `s0, s1, a, b, state0`, ...
  - Use: `idle, wait_for_x, start_tx, read_mem`, ...
- Average length of a good name is 8 to 16 characters

Use consistent and descriptive names.

Naming the architecture

- Architecture name is one of following:
- `behavioral`
  - Implies physical logic, cannot be compiled with RTL tools
- `rtl`
  - Implies physical logic, compiled with RTL tools
- `structural`
  - Implies physical connections, but not any logic

Use only conventional architecture names.

Label the processes

- Label every `process`
  - e.g. `define_next_state`, `define_output`
- Makes easier to identify part of the code implying specific logic in synthesis
- Label is written two times in the code:
  - Before and after the process
  - e.g. define_next_state: process ...

Label every process.

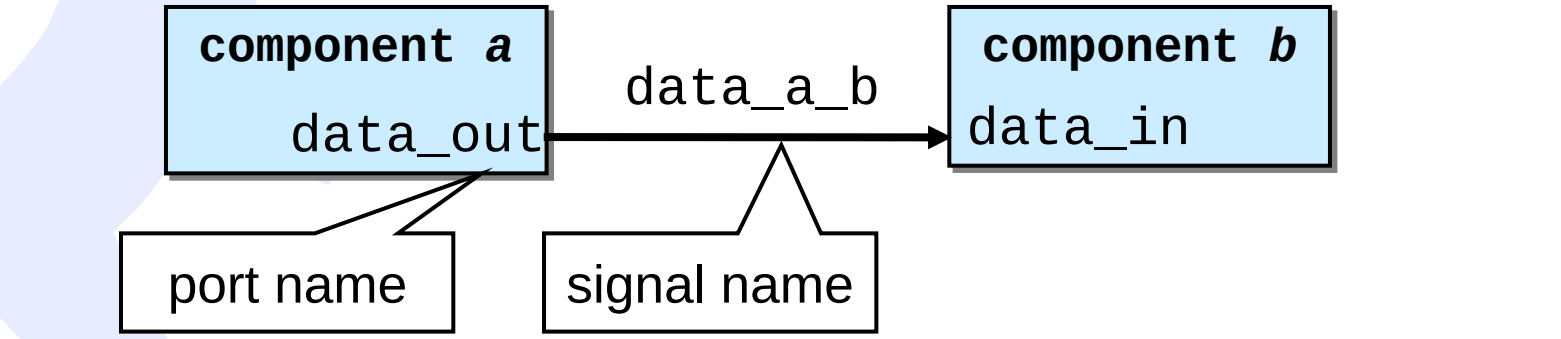
Clk and reset signals/inputs

- Active low reset is `rst_n`
  - Asynchronous set should not be used
  - A register should never have both asynchronous set and reset
- Clock signal `clk`
  - If there are more clocks the letters `"clk"` appear in every one as a postfix
- When a signal ascends through hierarchy, its name should remain the same. This is especially important for clock signals

Use names `clk` and `rst_n`.

Naming intermediate signals

- Signals from instance a to b are named: `signalname_a_b`
  - Needed in structural architectures
- They **NEVER** have "`in`" or "`out`" specifiers
  - output of a is connected to input of b
  - ➔ cannot decide which postfix to choose
- Abbreviated component names are handy when names are longer than a and b
- With multiple targets use `signalname_from_a`



Intermediate signal's name includes src and and dst.

Naming the instantances

- Component instance name is formed from the component name
- Attach prefix "`i_`" and identifier as a postfix:

```
i_componentname_id : componentname
  PORT MAP…
```

e.g.

```
i_fifo_out : entity work.fifo
  PORT MAP…
```

- This helps to track down the actual entity
  - From simulation results
  - From synthesis results
- Exceptions possible with long entity names
  - In this case, it might be best to shorten the entity name

Instance is named after the component entity.

Using for-generate

- FOR GENERATE statement is used for repetitive instantiations of the same component
- Label generate statement
- Example

```
g_datamux : FOR i IN 0 TO n_mux_c-1 GENERATE
 i_datamux : datamux
  PORT MAP (
     sel_in   => sel_in (i),
     data_in  => data_r (i),
     data_out => data_mux_alu(i)
  );
END GENERATE g_datamux;
```

- FOR GENERATE creates identifiers (running numbers) automatically to all instances

Use FOR GENERATE for repetitive instances

Recommended naming

- Between components `signalname_a_b`
- To multiple components `signalname_from_a`

- The only clock input port `clk`
- Low active reset input port `rst_n`

- Component instances `i_componentname_id`
- Generate statements `g_componentname`

Prefer these naming conventions

Prefer generics

- Basically, generic parameter is a fundamental idea in VHDL that enables design reuse, use it.
- Avoid constants (in packages or architecture)
  - if `data_width_c` is defined is package, it is impossible to have instances with different `data_width_c` values
  - ➔ E.g. This limits all adders in the system to 10 bits
  - With generics, it is possible to have different adders
- The component size should be changed with generics NOT by modifying the code.
    - When the VHDL code is reused, there should be no need to read the code except the entity definition
- If there are illegal combinations of generic values, use assertions to make sure that given generics are valid
- However, having many generic parameters, complicates verification

Prefer generics to package constants

Avoid bit vector literals

- Avoid bit vector literals
- Use conversion functions
- Bit vectors must be edited by hand if vector width changes

```
status_r <= "11110000";
```

- Width of the vector can be changed with generics but still the same number is assigned

```
status_r <= to_unsigned (err_code_c,reg_width_g);
```

Prefer conversion over bit vector literals.

Prefer arrays and loops

- Use arrays and loops instead of different names
  - Array size can be generic
  - Names (e.g `"signal_0, signal_1, ..."`) have to be modified by hand
  - Naturally, loop limits must be known at compile/synthesis time

```
priority_encoder : PROCESS (input)
BEGIN
   first <= data_width_c-1;
   FOR i IN data_width_c-2 DOWNTO 0 LOOP
      IF (input(i) = '1') THEN
         first <= i;
      END IF;
   END LOOP;
END PROCESS priority_encoder;
```

Prefer arrays over multiple separate signals and loops for repetitive operations.

Avoid variables inside processes

- Variables in processes usually make VHDL difficult to understand
  - Valid inside procedures fo functions
- Use variables only for storing intermediate values
- Only used as “short-hand notation”

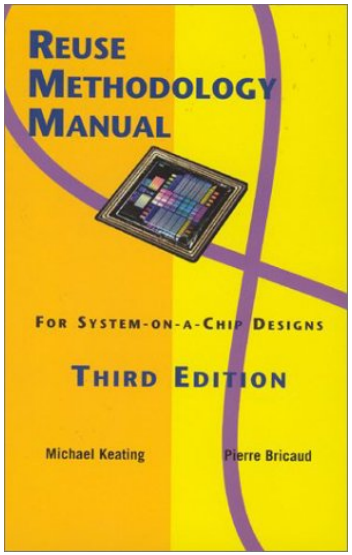
```
tmp_v           := addr_r (3)(2);
data_r (tmp_v) <= a_in (tmp_v) + b_in(tmp_v);
```

- **Never imply registers with variables**
  - Happens when you try to read the variable before its assigned

Avoid variables in synthesizable code.

Contributors

- Version 4, Dec. 2007: E. Salminen, A. Rasmus, and A. Kulmala
  - Earlier versions included also: M. Alho, K. Kuusilinna, V. Lahtinen, H. Lampinen, J. Tomberg
- See also VHDL FAQ:
  - http://www.vhdl.org/comp.lang.vhdl/FAQ1.html
- Further reading:
  - M. Keating, P. Bricaud, Reuse methodology manual: for system-on-a-chip designs, Kluwer Academic Publishers Norwell, MA, USA, 1998 / 2002, ISBN:0-7923-8175-0



The end

Extra slides

Implying logic: General

- All functionality should be contained within `rtl` architecture (leaf block)
- Every block above a leaf block up to the top level should contain only component instantiations and wiring (`structural` architecture).
- Constant values should not be routed through hierarchy
- Three-state signals are not allowed inside the chip
- If inverted clock is needed, introduce the inverted clock signal yourself
  - All clock and async. reset signals are generated in a single, dedicated block

Implying logic: General (2)

- Do not make self resetting blocks
- All timing exceptions should be contained within a single block
- Especially avoid so-called snake paths
  - Snake path is a combinational path going through many blocks of the design.
  - Time budgeting of snake paths is very difficult
- Sometimes it is useful to indicate bits that have been left off with the number of the LSB
  - LSB index is not 0
  - For example an address bus with the two LSBs left off:

```
SIGNAL addr :
STD_LOGIC_VECTOR(datawidth_g-1 DOWNTO 2);
```

Safe coding: Miscellaneous

- Avoid subtypes
- Use only `STD_LOGIC` signal states `'0'`, `'1'` and `'Z'`
  - Never refer to state `'X'` in VHDL code
- Do not embed synthesis script in the VHDL code
  - Difficult to maintain both the script and the code
- Avoid instantiating library primitives
  - If you do, keep them in a separate block
  - Consider Synopsys GTECH library components

Variables again

- Avoid variables in synthesizable code
  - Variables speed up the simulation
  - But safety is orders of magnitude more important than simulation speed
- Example need for a variable
  - XORing all bits of a vector:

```
probe_v := '0';
FOR i IN 0 TO 31 LOOP
   probe_v := probe_v XOR data_in(i);
END LOOP;
probe_out <= probe_v;
```

Ordering of entity's ports

- Ports of the entity should be grouped as:
  - Resets
  - Clocks (preferably just one)
  - Signals of group A
  - Signals of group B
  - Signals of group C
  - …
- One entity should have only one clock
- If more than one clock is necessary, minimize the number of blocks with multiple clocks
  - Place synchronization into separate entity

Ordering: Declarations I

- Component and signal declarations are ordered in groups
- One component and specific signals:
  - Declaration of component A
  - Signals the instantiations of component A drive
  - Declaration of component B
  - Signals the instantiations of component B drive
  - Declaration of component C
  - Signals the instantiations of component C drive
  - …
  - All other signals (if any)
- Order of the component instantiations should be the same than the order of the component declarations

Code appearance: Aligning I

- One statement per line
- One port declaration per line, own line also for end parenthesis
- Align the colons and port types in the entity port:

```
ENTITY transmogrifier IS
   PORT (
      rst_n      : IN STD_LOGIC;
      clk        : IN STD_LOGIC;
      we_in      : IN  STD_LOGIC;
      cmd_0_in   : IN  STD_LOGIC_VECTOR(3-1 DOWNTO 0);
      data_in    : IN  STD_LOGIC_VECTOR(5-1 DOWNTO 0);
      valid_out  : OUT STD_LOGIC;
      result_out : OUT STD_LOGIC_VECTOR(6-1 DOWNTO 0)
);
END transmogrifier;
```

- *Note: avoid magic numbers in real code*

Code appearance: Aligning II

- Align colons inside one signal declaration group:

```
-- control signals
SIGNAL select     : STD_LOGIC_VECTOR (2-1 DOWNTO 0);
SIGNAL cmd_r      : STD_LOGIC_VECTOR (32-1 DOWNTO 0);
SIGNAL next_state : state_type;


-- address and data signals
SIGNAL rd_addr  : STD_LOGIC_VECTOR (16-1 DOWNTO 0);
SIGNAL wr_addr  : STD_LOGIC_VECTOR (16-1 DOWNTO 0);
SIGNAL rd_data  : STD_LOGIC_VECTOR (32-1 DOWNTO 0);
SIGNAL wr_data  : STD_LOGIC_VECTOR (32-1 DOWNTO 0);
```

Code appearance: Aligning III

- Align the `=>` in port maps:

```
i_pokerhand : pokerhand
PORT MAP (
   rst_n     => rst_n,
   clk       => clk,
   card_0_in => card (i),
   card_1_in => card (i),
   card_2_in => card (i),
   card_3_in => card (i),
   card_4_in => card (i),
   hand_out  => hand
   );
```

- Emacs: `ctrl-c ctrl-a ctrl-b` aligns the whole buffer

Code appearance: Spacing

- Conditions are written inside parenthesis
- There is a space outside parenthesis, but not inside

```
IF (rst_n = '0') THEN
```

- There is a space after a comma, but not before:

```
digital_phase_locked_loop : PROCESS (rst_n, clk)
```

- There is a space on both sides of `=>, <=, :=, >, <, =, /=, +, -, *, /, &, AND, OR, XOR`
- E.g.

```
data_output <= ((('0' & a) + ('0' & b)) AND c);
```

Commenting example

- One-liners are used in most cases:

```
set_byte_enables : PROCESS (rst_n, clk)
BEGIN
   IF (rst_n = '0') THEN
      be_r <= (OTHERS => '0');

   ELSIF (clk'EVENT and clk = '1') THEN

      IF (state_r = lo_part_c) THEN
         -- write lower 16-bit word
         be_r <= "0011";
      ELSIF (state_r = hi_part_c) THEN
         -- write higher 16-bit word
         be_r <= "1100";
      ELSE
         be_r <= "0000"; -- idle, altenative comment place
      END IF;
   END IF;
END PROCESS set_byte_enables;
```

Commenting example (2)

- Large comment is used mostly before processes:

```
-------------------------------------------------
-- Parity bit is calculated for the DATA_INPUT signal.
-------------------------------------------------
parity_calculation : PROCESS (rst_n, clk)
BEGIN
   IF (rst_n = '0') THEN
      parity <= '0';
   ELSIF (clk'EVENT and clk = '1') THEN
      parity <= data_input(3) XOR data_input(2)
                XOR data_input(1) XOR data_input(0);
   END IF;
END PROCESS parity_calculation;
```

Numeric packages (1)

- `numeric_std` defines two new vector types and operations for them
  - IEEE standard package
  - `SIGNED` vectors represent two's-complement integers
  - `UNSIGNED` vectors represent unsigned-magnitude integers
  - Functions and operators
    - Logical : and, or, not,...
    - Arithmetic : abs, +, -, *, ...
    - Comparison : <, >, /=, ...
    - Shift : shift_left, rotate_left, sll,...
    - Conversion : see next slide
    - Misc : resize, std_match, ...
- For more detail, see also:
  - http://www.vhdl.org/comp.lang.vhdl/FAQ1.html#4.8.1

Numeric packages (2)

Types in Binary Arithmetic Operations			
		numeric_std	std_logic_arith
Argument 1	**Argument 2**	**Result**	
unsigned	unsigned	unsigned	unsigned/std_logic_vector
unsigned	integer	unsigned	unsigned/std_logic_vector
integer	unsigned	unsigned	unsigned/std_logic_vector
signed	signed	signed	signed/std_logic_vector
signed	integer	signed	signed/std_logic_vector
integer	signed	signed	signed/std_logic_vector

Differences of packages

Source: http://dz.ee.ethz.ch/support/ic/vhdl/vhdlsources.en.html

(visited 02.11.2005)

		numeric_std	std_logic_arith
Type Conversion			
std_logic_vector	-> unsigned	unsigned(*arg*)	unsigned(*arg*)
std_logic_vector	-> signed	signed(*arg*)	signed(*arg*)
unsigned	-> std_logic_vector	std_logic_vector(*arg*)	std_logic_vector(*arg*)
signed	-> std_logic_vector	std_logic_vector(*arg*)	std_logic_vector(*arg*)
integer	-> unsigned	to_unsigned(*arg*, *size*)	conv_unsigned(*arg*, *size*)
integer	-> signed	to_signed(*arg*, *size*)	conv_signed(*arg*, *size*)
unsigned	-> integer	to_integer(*arg*)	conv_integer(*arg*)
signed	-> integer	to_integer(*arg*)	conv_integer(*arg*)
integer	-> std_logic_vector	integer -> unsigned/signed ->std_logic_vector	
std_logic_vector	-> integer	std_logic_vector -> unsigned/signed ->integer	
unsigned + unsigned	-> std_logic_vector	std_logic_vector(*arg1* + *arg2*)	*arg1* + *arg2*
signed + signed	-> std_logic_vector	std_logic_vector(*arg1* + *arg2*)	*arg1* + *arg2*
Resizing			
unsigned		resize(*arg*, *size*)	conv_unsigned(*arg*, *size*)
signed		resize(*arg*, *size*)	conv_signed(*arg*, *size*)