e

Real-Time Imaging and Control

Tips

MSCV/ESIREM

Antoine Lavault
antoine.lavault@u-bourgogne.fr

VHDL Tips

General Advice

Rule 1. *VHDL is a description language, not a programming language.*
You should be able to read a VHDL file and understand what is described.

```
ARCHITECTURE structural OF top IS
    SIGNAL temp : STD_LOGIC;
BEGIN
    or1 : ENTITY work.or_gate(Behavioural)
        PORT MAP(a => a, b => b, c => temp);
    or2 : ENTITY work.or_gate(Behavioural)
        PORT MAP(a => temp, b => c, c => d);
END structural;
```

The above description should be read as:
"A structural architecture of the component top, with an internal signal temp. Inside, the component is described by two components (or_gate) where the first has its ports mapped to a,b and temp and the second to temp,c and d".

1 Components and Libraries

Rule 2. *The only acceptable IEEE package for arithmetic and comparison operations on vectors is* `numeric_std`.

`std_logic_arith`, `std_logic_signed`, `std_logic_unsigned` are not allowed. They are obsolete and should no longer be used.
Note that legacy code often uses these libraries, so it is important to understand their operation as some companies might rely on them.

Rule 3. *Whenever a library function or Intelectual Property (IP) component exists for a specific operation (e.g., addition, multiplication, shift, etc.), prefer it to a custom implementation.*

The synthesis tools easily identify library functions and lead to highly optimized IP component implementations. However, library functions are very precisely defined and will not necessarily fit specific requirements (e.g., the "+" operator does not provide a carry-out line, the output vector of multiplication has a fixed size that might not match requirements, etc...). Always carefully examine the library functions' description (the `numeric_std` package is a well-commented text file; generally, all libraries/IPs come with documentation).
The definition of personal dedicated libraries and packages for projects is a very common advanced design technique. Very complex libraries of IP components are available for advanced designs (often very expensive).

Rule 4. *When instantiating components, always prefer to specify the architecture with direct instantiation in the architecture rather than using the declare/instantiate method.*

Compared to component instantiation, it is much cleaner to read, and technically component instantiation shouldn't even work without an external file mapping components to architectures (but Vivado is smart enough to find things).

```
1  -- Good
2  ARCHITECTURE structural OF top IS
3      SIGNAL temp : STD_LOGIC;
4  BEGIN
5      or1 : ENTITY work.or_gate(Behavioural)
6          PORT MAP(a => a, b => b, c => temp);
7      or2 : ENTITY work.or_gate(Behavioural)
8          PORT MAP(a => temp, b => c, c => d);
9  END structural;
10 -- Acceptable
11 ARCHITECTURE structural OF top IS
12     COMPONENT OR_gate IS
13         PORT (
14             a, b : in STD_LOGIC;
15             c : OUT STD_LOGIC);
16     END COMPONENT;
17     SIGNAL temp : STD_LOGIC;
18 BEGIN
19     or1 OR_gate :
20     PORT MAP(a => a, b => b, c => temp);
21     or2 OR_gate :
22     PORT MAP(a => temp, b => c, c => d);
23 END structural;
```

Rule 5. *When instantiating components, always use explicit port mapping.*

This will prevent all sorts of hard-to-trace errors from getting the ports in the wrong order, particularly when adding or removing ports from components during the design process.
Also, the mapping process is of the form `entity_port => signal`.

```
-- Good
ARCHITECTURE structural OF top IS
    COMPONENT OR_gate IS
        PORT (
            a, b : in STD_LOGIC;
            c : OUT STD_LOGIC);
    END COMPONENT;
    SIGNAL temp : STD_LOGIC;
BEGIN
    or1 OR_gate :
    PORT MAP(a => a, b => b, c => temp);
    or2 OR_gate :
    PORT MAP(a => temp, b => c, c => d);
END structural;
-- Not good, DO NOT USE
BEGIN
    or1 OR_gate :
    PORT MAP(a, b, temp);
    or2 OR_gate :
    PORT MAP(temp, c, d);
END structural;
```

2 Combination Logic

Rule 6. *Process syntax should never be used to describe combinational logic.*

Processes are VHDL constructs specifically designed to implement sequential logic. While numerous examples exist in books or online resources where processes are used to implement combinational logic (usually, but not exclusively, multiplexers), this is poor design practice and should be avoided.
Important: There is a single exception to this rule that applies to FSM implementations (see the section on FSMs).

Rule 7. *When instantiating multiplexers, always use one of the two dedicated VHDL structures with a "catch-all" clause set to value 'U.'*

While a catch-all outputting a set value (in general, zeros) is acceptable, a better way to handle the case is to use the 'U' value for the `std_logic` types.

```
-- General case
O <= A WHEN SEL = "00" ELSE
     B WHEN SEL = "01" ELSE
     C WHEN sel = "10" ELSE
     D WHEN sel = "11" ELSE
     (OTHERS => 'U');
-- Mux-specific syntax
WITH sel SELECT
     O <= A WHEN "00",
     B WHEN "01",
     C WHEN "10",
     D WHEN "11",
     (OTHERS => 'U') WHEN OTHERS;
```

VHDL provides two syntactical structures for implementing multiplexers: a more general "when...else" and a specific "with...select". All multiplexers should be implemented using one of these two structures (processes should, in particular, be avoided).
To ensure that a multiplexer will be synthesized correctly, a "catch-all" clause should always be included. For synthesis only, the catch-all clause can correspond to the default output for all remaining valid control values. To improve the simulation's quality, accuracy, and usefulness, however, it is recommended to define a valid output for all valid control values and output value 'U' for all non-valid control values (note that 'U' should never be assigned to valid input combinations).
While this rule is particularly useful for beginners, there is no reason to deviate from it, even for more advanced users.

3 Sequential Logic

Rule 8. *Implement each sequential logic component in your design in a separate process.*

This rule applies especially to beginners and will help considerably to avoid generating complicated, non-synthesizable VHDL structures.
In general, even for more advanced users, it is still advisable to separate sequential components unless they share the same clock, reset, and enable signals.

Rule 9. *When testing the clock edge in a process, prefer the `rising_edge()` function within an if statement.*

There are several possible (mostly equivalent) implementations of the test for the clock edge in a process. This is the most accurate and should be preferred.

Rule 10. *Never use standard VHDL syntax to implement any form of logic operation (AND, OR, mux, etc.) on the clock signal (aka "the clock is sacred - don't ever touch it unless you know what you are doing").*

Clock gating (the technical term describing the use of logic signals to affect the clock) is an advanced logic design technique requiring specific knowledge and dedicated resources.

4 Signals/Variables/Constants

Rule 11. *When changing the size of a SIGNED or UNSIGNED vector is required, using the* `resize()` *function is preferred.*

This function is particularly useful for `SIGNED` vectors (where sign extension must be taken into account) but should be applied as a rule to `UNSIGNED` as well (it is also quite useful for parameterization). *Note: resizing from larger to smaller vectors is possible, but remember that this might cause data loss if the smaller vector is not large enough to encode the original value.*

Rule 12. *When assigning generic values to a component, prefer using* `GENERIC MAP` *at instantiation (rather than defining the value in the declaration).*

When using component instantiation, generics can be defined in the declaration. However, this is far less readable and does not allow using the same component more than once with different generic values. This rule becomes automatic when using direct instantiation (as there is no component declaration). Note that the default value inside the component should not be relied upon but always re-defined in the instantiation (even when it happens to be the same).

```
1  -- Good
2  ARCHITECTURE ckt OF my_circuit IS
3  BEGIN
4      r1 : ENTITY work.reg (Behavioral)
5          GENERIC MAP(d_size => 8)
6          PORT MAP(
7              clk => clk,
8              rst => rst,
9              d_in => inp1,
10             d_out => outp1);
11     r2 : ENTITY work.reg (Behavioral)
12     GENERIC MAP(d_size => 16)
13     PORT MAP(
14         clk => clk
15         rst => rst,
```

```
16          d_in => inp2,                                                    e
17          d_out => outp2);
18  END ckt;
19  -- Less good.
20  ARCHITECTURE ckt OF my_circuit IS
21      COMPONENT reg IS
22          GENERIC (d_size : INTEGER := 8);
23          PORT (
24              rst, clk : IN STD_LOGIC;
25              d_in : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
26              d_out : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
27      END reg;
28  BEGIN
29      r1 : reg PORT MAP(
30          clk => clk,
31          rst => rst,
32          d_in => inp1,
33          d_out => outp1);
34      -- Default d_size will be reused silently !!!
35      r2 : reg PORT MAP(
36          clk => clk,
37          rst => rst,
38          d_in => inp1,
39          d_out => outp1);
40  END;
```

5 Finite State Machines

Rule 13. *When implementing an FSM, states should always have meaningful names and be implemented as an enumerated type*

The reason should be obvious. Avoid state labels such as `S1,S2,S3...` or `A,B,C,..` and use descriptive labels (for instance `open, closed, opening...` or `reset, count, enable_MAC...`).

Rule 14. *When implementing Finite State Machines, prefer the "double-process" implementation.*

Finite state machines (FSMs) should always be implemented using a precise methodology. A few such methodologies exist, but probably the most common is the one based on two separate processes:

- the first to assign the next state at the edge of the clock,
- the second to instantiate the transition table.

Note that this second process is the only allowed exception to the rule "no processes for combinational logic" and should be used only to generate the next state value (no other signals should be assigned).

```vhdl
ARCHITECTURE arch OF garage_door_FSM IS
    TYPE state_type IS (closed, opened);
    SIGNAL state, next_state state : state_type;
BEGIN
    -- The first process defines the (synchronous) reset state and updates the state register
    state_assignment : PROCESS (clk) IS
    BEGIN
        IF rising_edge(clk) THEN
            IF (reset = '1') THEN
                state <= closed;
            ELSE
                state <= next_state;
            END IF;
        END IF;
    END PROCESS state_assignment;
    -- The second process implements the state transitions (next_state assignments ONLY).
    -- The sensitivity list includes the current state and all inputs to the FSM
    state_transitions : PROCESS (state, button, one_second) IS
    BEGIN
        CASE state IS
            WHEN closed =>
                -- ALL transitions MUST be specified
                IF (button '1' AND one_second '1') THEN
                    next_state <= opened;
                ELSE
                    next_state <= closed;
                END IF;
            WHEN opened =>
                IF (button '1' AND one_second '1') THEN
                    next_state <= closed;
                ELSE
                    next_state <= opened;
                END IF;
            WHEN OTHERS => next_state <= closed; -- "catch-all" clause for safety
        END CASE;
    END CASE;
END PROCESS state_transitions;
-- After the two processes, the outputs are assigned (without processes)
signal_open <= '1' WHEN(state = opened AND one_second = '0') ELSE
    '0';

signal_closed <= '1' WHEN(state = closed AND one_second = '0') ELSE
    '0';
```

```
44 END arch;                                                    e
```

Rule 15. *After the second process, all outputs that depend on the state should be assigned within the same entity that contains the FSM.*

The "raw" state of an FSM should never be output from the entity. This would normally require encoding the states to a vector and assigning specific binary values to each state, which prevents the tools from generating optimal FSM implementations.
Outputs depend on the state only in a Moore FSM or on the state and the inputs in a Mealy FSM, but in both cases, they represent combinational logic and should not be implemented within a process.

6 Test-benches and Simulation

Rule 16. *Always ensure that your simulation matches the final circuit as closely as possible.*

Undefined or 'X' states, particularly at startup, are common in simulations and accurately reflect the state of the circuit. Designing test-benches and/or writing code to "hide" these conditions should be avoided.
For example, all memory elements in a design will output undefined values until they are reset or assigned. This should be apparent in your simulation.

Rule 17. *Always use multiples of the clock cycle to control the timing of test-benches.*

The operating frequency of a circuit is a fundamental variable in the design cycle, likely to change. Explicit, absolute timings in wait statements can invalidate your test bench if the clock frequency changes. This rule does not apply in the (extremely rare outside of teaching labs) cases of completely combinational circuits with no clock signals.

Rule 18. *Test-bench entity/file names should always end with "_tb." In any case, it should be identifiable by its file name.*

Many tools handle test benches differently from a "standard" VHDL description; ensuring they are recognized as such is important. The convention above ensures that.
Also, it will put the source and the test-bench files next to each other in a sorted file list.