# EDC: exam

Renaud PACALET

18 June 2014

The text in black is the original one. The text in red is (very detailed) examples of the expected correct answers. Only this text was expected, in much shorter forms, nothing more. The text in blue is extra comments about the expected correct answers.

You can use any document you need. Please number the different pages of your work and indicate on each page your first and last names. Write your answers in French or in English, as you wish, but avoid mixing the languages. If some extra information or hypotheses are missing to answer a question or solve a problem, decide by yourself and write down the added hypotheses or information. If you consider a question as absurd and thus decide not to answer, explain why. If you don't have time to answer a question or solve a problem but know how to, briefly explain your ideas. The first part is a set of five questions (2 points each) and the second part is a small problem (10 points).

Important advice #1: quickly go through the document and answer first the easy parts.

Important advice #2: copying verbatim the slides of the lectures or any other provided material is not considered a valid answer.

## 1 Questions

1.1. In synthesizable VHDL there are two kinds of processes: the processes which synthesis will infer memory elements and the processes which synthesis will infer only glue logic. Summarize the different constraints on these two kinds. What rules shall they obey? Why? And what could happen if they do not?

The sensitivity list of combinatorial processes (the ones that infer only glue logic) must contain all signals that are read during the process execution and no more. They must assign their output signals in each branch of their control flow. If they use variables they must always assign them before reading them. Incomplete sensitivity lists can lead to behaviour mismatches between pre- and post-synthesis simulations, which is why most logic synthesizers issue warnings on them. Extra signals in sensitivity lists may slow down the simulation by uselessly triggering the process. Signals not assigned during an execution or variables read before being assigned force the synthesizer to infer memory elements (D flip-flops or latches) which is undesirable in combinatorial logic. This is why logic synthesizers issue warning when they infer latches.

The sensitivity list of synchronous processes (the ones that infer memory elements) must contain only the clock and the asynchronous set and reset, if any.

If the expected memory elements are latches, then, the signals that are used to compute the inputs of the latches must also be part of the sensitivity list. Output signals and variables must be assigned under control of the asynchronous set and reset (highest priority), if any, then under control of the edge (D flip-flops) or the level (latches) of the clock. Statements outside this control structure must be avoided because they prevent the correct inference of memory elements. Incomplete sensitivity lists also prevent the correct inference of memory elements and are signalled by most logic synthesizers. Superfluous signals in the sensitivity list may slow down the simulation by uselessly triggering the process.

In most cases D flip-flops are wanted and the process shall be written as shown on listing 1 (without asynchronous set/reset) or on listing 2 (with asynchronous set/reset).

```
process(clock)
  variable vregs: some_type;
begin
  if rising_edge(clock) then
    sregs <= f(some_signals, vregs);
    vregs := g(some_signals, vregs);
  end if;
end process;
```

Listing 1: Synchronous process

```
process(clock, set, reset)
  variable vregs: some_type;
begin
  if set = '1' then
    sregs <= some_value;
    vregs := some_value;
  elsif reset = '1' then
    sregs <= some_other_value;
    vregs := some_other_value;
  elsif rising_edge(clock) then
    sregs <= f(some_signals, vregs);
    vregs := g(some_signals, vregs);
  end if;
end process;
```

Listing 2: Synchronous process

Listings 3 and 4 show synchronous processes on which logic synthesizers infer latches, without and with asynchronous set/reset.

```
process(clock, some_signals)
  variable vregs: some_type;
begin
  if clock = '1' then
    sregs <= f(some_signals, vregs);
    vregs := g(some_signals, vregs);
  end if;
end process;
```

Listing 3: Synchronous process

```
process(clock, set, reset, some_signals)
  variable vregs: some_type;
begin
  if set = '1' then
    sregs <= some_value;
    vregs := some_value;
  elsif reset = '1' then
    sregs <= some_other_value;
    vregs := some_other_value;
  elsif clock = '1' then
    sregs <= f(some_signals, vregs);
    vregs := g(some_signals, vregs);
  end if;
end process;
```

Listing 4: Synchronous process

1.2. Translate in CTL the following property:

> If the signal DSI is high and DSO is low, then, on the next cycle, DSI and DSO are low and DSI cannot be high again before DSO went high. And because DSO is not guaranteed to go high in the future, it could even be that DSI cannot go high anymore.

With `not`, `and`, `or` and `imply` in decreasing order of precedence and denoted `!`, `*`, `+`, `=>`, respectively:

```
AG(DSI * !DSO => AX(!DSI * !DSO * A(!DSI W DSO)))
```

Without the weak until:

```
AG(DSI * !DSO => AX(!DSI * !DSO * !E(!DSO U (DSI * !DSO))))
```

And no, the weak until cannot be re-written:

```
A(p U q) + AG(!q)
```

because this would mean: «all executions satisfy $p$ until they satisfy $q$, which finally happens in the future» **or** «all executions never satisfy $q$», which is different from: «all executions satisfy $p$ until they satisfy $q$, which may never happen». Do you understand the difference?

1.3. What is the minimum number of processes required to model a Moore finite state machine[1]? Why? Same questions with a Mealy finite state machine?

Two processes for both Moore and Mealy state machines:

- a synchronous one to model the state register and the combinatorial logic that computes the next state from the current one and the primary inputs,
- a combinatorial one to compute the outputs from the current state and, in the Mealy case, the primary inputs.

The combinatorial logic that computes the next state from the current one and the primary inputs can also be modelled in the second process, even it is quite rare and less natural.

In some cases, the output of a Moore state machine is exactly the current state, without further combinatorial transforms. It is the case, for instance, with the `CELL` component of the Game of Life problem. In these cases and only in these cases, the state machine can be described with one single synchronous process, but, as the current state is an output, this poses the classical VHDL problem of reading a primary output. Listing 5 shows the most frequently used solution to this problem. Note that the concurrent signal assignment (line #4) is itself a process and that, technically speaking, there are thus two processes in the description.

```
1  architecture a1 of moore is
2    signal state: state_type;
3  begin
4    outputs <= state;
5    process(clk)
6    begin
7      if rising_edge(clk) then
8        state <= f(state, inputs);
9      end if;
10   end process;
11 end architecture a1;
```

Listing 5: Description of a Moore state machine with (almost) one single process

Another way is to use a variable to model the state register, as shown on listing 6. But technically speaking, two state registers shall be inferred, one for the variable and one for the output signal. As they are strictly equivalent (same inputs), most synthesizers or optimizers will remove the redundant one but this kind of description shall be avoided, if possible.

```
1  architecture a2 of moore is
2  begin
3    process(clk)
4      variable vstate: state_type;
5    begin
6      if rising_edge(clk) then
7        vstate := f(vstate, inputs);
8        outputs <= vstate;
9      end if;
10   end process;
11 end architecture a2;
```

[1]The outputs of a Moore finite state machine depend on the current state only while the outputs of a Mealy finite state machine may depend on the current state and also on the inputs

Listing 6: Description of a Moore state machine with one single process

Finally, in some very special cases of these special cases, it is possible to describe the state machine without reading the current state at all: it is when, depending on a condition on the primary inputs only, the next state is either equal to the current state or dependent on the primary inputs only. It is the case, for instance, with the `CELL` component of the Game of Life problem, as we will see. In these very special cases the state machine can be described with on single synchronous process, without variable, as shown on listing 7. It is made possible by the fact that keeping the state unchanged, in a synchronous process, does not require to access the state; not assigning it is considered by the logic synthesizer as *keep it unchanged*.

```
1  architecture a3 of moore is
2  begin
3    process(clk)
4    begin
5      if rising_edge(clk) then
6        if condition(inputs) then
7          outputs <= g(inputs);
8        end if; -- else do nothing
9      end if;
10   end process;
11 end architecture a3;
```

Listing 7: Description of a Moore state machine with one single process

1.4. The event-driven simulation algorithm (the one used by VHDL simulators) handles variables and signals differently. Explain why and what are the main differences.

Variables are local to processes (but shared variables also exist in recent flavours of VHDL), cannot be used to exchange information between concurrent processes, and are handled like in any other programming language: their assignment (`:=` in VHDL) is immediate. Signals can be used to exchange information between concurrent processes, are declared at the architecture level (they are visible in every process' scope) and their assignment (`<=` in VHDL) as no immediate effect: the value update of a signal is delayed until the end of the process execution phase (what is referred to as the *delta* cycle). The signal and its special handling are needed to simulate parallel models on sequential computers in a completely deterministic way: thanks to it, the simulation result does not depend on the order of execution of the different processes.

1.5. Assume the designer of the following VHDL code wanted to design a synthesizable D flip-flop with synchronous active low reset and asynchronous active high set[2]. What do you think of her work? Identify the errors if any and, for each of them, explain why it is an error, what undesirable effect it has and finally, write down a new VHDL code with all the errors fixed.

```
signal a, b, c, d, e: std_ulogic;
...
```

[2] A *reset* forces to '0' while a *set* forces to '1'

```
process(a, b, c, d)
begin
  if a = '0' then
    d <= '0';
  elsif b = '1' and b'event then
    d <= c;
  end if;
  if e = '1' then
    d <= '1';
  end if;
end process;
```

This design violates two important rules:

- The sensitivity list shall contain only the clock (`b`) and the asynchronous set (`e`). Having also the synchronous reset (`a`), the output (`d`), the input (`c`) but not the asynchronous set are severe bugs that will lead to simulation mismatches before and after synthesis, asynchronous reset instead of synchronous and unnecessary resumes of the process during simulation. Most logic synthesizers will issue warnings or errors on this, mainly because the set (`e`) is handled outside the control of the clock edge but is not listed in the sensitivity list.
- The handling of the reset shall be under control of the clock edge because the reset is supposed to be synchronous. Undesirable effect: asynchronous reset instead of synchronous.

A better coding would be as shown on listing 8.

```
1  process(b, e)
2  begin
3    if e = '1' then
4      d <= '1';
5    elsif b = '1' and b'event then
6      if a = '0' then
7        d <= '0';
8      else
9        d <= c;
10     end if;
11   end if;
12 end process;
```

Listing 8: Bugs fixed

We could also object that the whole process body shall be one single `if` statement instead of two. As it is, even if it were semantically correct, it could be rejected by some logic synthesizers expecting a more classical from of synchronous processes.

# 2 Design of a cellular automata

The *Game of Life* is a cellular automata invented in 1970 by John Horton Conway, a British mathematician. It could be that you already met this game because variants of

it are sometimes used as screen savers. The goal of this small problem is to design a synthesizable VHDL model of a hardware implementation of John Conway's game.

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of 2-states state machines. Each state machine communicates its current state (*live* or *dead*) to its eight neighbours and each state machine receives the current state of its eight neighbours. In the following we name *cell* the state machines and *NW,N,...,SW,W* the eight neighbours of a cell. The automata is synchronous of a global clock: on each rising edge of the global clock, each cell updates its state. The next state of a cell depends on its current state and on the number of its live neighbours:

- A live cell with zero or one live neighbour dies.
- A live cell with two or three live neighbours survives.
- A live cell with four or more live neighbours dies.
- A dead cell with exactly three live neighbours becomes a live cell.

Figure 1 illustrates the evolution of the game on a particular initial state called the *Lightweight spaceship* (LWSS). The same, $9 \times 8$, small portion of the universe is represented in each state. On the top row the evolution is decomposed, showing the dying and appearing cells between two consecutive states. The same sequence is represented on the bottom row without the intermediates. Do you understand why it is called the Lightweight spaceship?

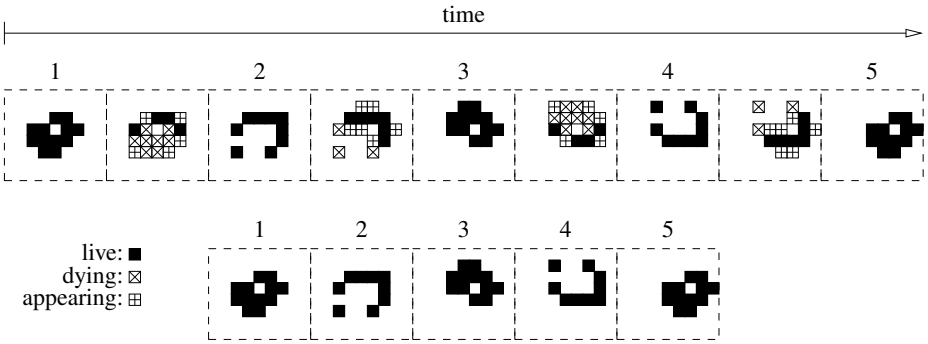


Figure 1: Five consecutive states of the Lightweight spaceship

**TODO (5 points)**: Design, in plain synthesizable VHDL (entity and architecture), a hardware cell (`CELL`). The interface of the `CELL` module is specified in figure 2 and table 1. All inputs and outputs are 1-bit.

The synchronous reset `srstn` has the highest priority (higher than the clock enable `ce`). When it is low on a rising edge of `clk` the cell dies (its current state becomes 0). The clock enable `ce` has the next highest priority: when it is low (0) the current state is frozen. When it is high the current state can change. The load command `load` has the next highest priority: when it is high (1) on a rising edge of `clk` the cell takes the state of its west (`W`) neighbour, whatever its current state and the state of its other neighbours. This is used to initialize the universe with consecutive shifts. Finally, when `srstn` and `ce` are high and `load` is low, the `CELL` works as explained above: its next state is computed from its current state and the state of its eight neighbours.

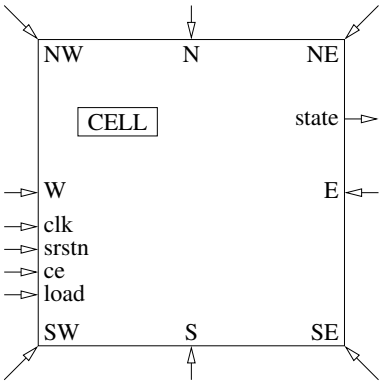


Figure 2: Interface specification of the `CELL` module

| Name | Direction | Description |
|---|---|---|
| `clk` | Input | Global clock |
| `srstn` | Input | Global synchronous active low reset (force to 0: dead) |
| `ce` | Input | Clock enable |
| `load` | Input | When set, take `W` as next state |
| `NW` | Input | Current state of the north-west neighbour (0: dead, 1: live) |
| `N` | Input | Current state of the north neighbour (0: dead, 1: live) |
| `NE` | Input | Current state of the north-east neighbour (0: dead, 1: live) |
| `E` | Input | Current state of the east neighbour (0: dead, 1: live) |
| `SE` | Input | Current state of the south-east neighbour (0: dead, 1: live) |
| `S` | Input | Current state of the south neighbour (0: dead, 1: live) |
| `SW` | Input | Current state of the south-west neighbour (0: dead, 1: live) |
| `W` | Input | Current state of the west neighbour (0: dead, 1: live) |
| `state` | Output | Current state of `CELL` (0: dead, 1: live) |

Table 1: Interface specification of the `CELL` module

The current state of a `CELL` is sent to its eight neighbours through the `state` output.

The VHDL source code of `CELL` is shown on listing 9.

```
library ieee;
use ieee.std_logic_1164.all;

entity cell is
  port(clk, srstn, ce, load: in std_ulogic;
       nw, n, ne, e, se, s, sw, w: in std_ulogic;
       state: out std_ulogic);
end entity cell;

architecture rtl of cell is
begin
  process(clk)
    variable cnt: natural range 0 to 8;
    variable v: std_ulogic_vector(0 to 7);
  begin
    if rising_edge(clk) then
      if srstn = '0' then
        state <= '0';
      elsif ce = '1' then
        if load = '1' then
          state <= w;
        else
          v := (nw, w, ne, e, se, s, sw, w);
          cnt := 0;
          for i in 0 to 7 loop
            if v(i) = '1' then
              cnt := cnt + 1;
            end if;
          end loop;
          if cnt = 3 then
            state <= '1';
          elsif cnt /= 2 then
            state <= '0';
          end if; -- elsif state = 2 do nothing
        end if;
      end if;
    end if;
  end process;
end architecture rtl;
```

Listing 9: The `CELL` VHDL source code

**TODO (2 points)**: Bonus question: what is the minimum number of processes that are needed to design `CELL`? Does this change is the reset if asynchronous?

One only, as shown in the proposed solution. This is the special case mentioned on the extended answer about the minimum number of processes required to model a finite state machine: the output is the current state and there is no need to read it. This does not change with an asynchronous reset, as shown on listing 10.

```
1  process(clk, reset)
2    variable cnt: natural range 0 to 8;
3    variable v: std_ulogic_vector(0 to 7);
4  begin
5    if reset = '1' then
6        state <= '0';
7    elsif rising_edge(clk) then
8      if ce = '1' then
9        -- ..........
```

Listing 10: The `CELL` with asynchronous reset

**TODO (3 points)**: Using instances of the `CELL` module, some glue logic and generic and generate statements, design a $r$ rows times $c$ columns universe where $r$ and $c$ are two generic parameters. Rows are numbered 0 (top) to $r-1$ (bottom) and columns are numbered 0 (left) to $c-1$ (right). The cell in row $i$ and column $j$ is denoted $(i, j)$. Your universe is a torus where:

- the north neighbour of cell $(0, j)$ is cell $(r-1, j)$,
- the left neighbour of cell $(i, 0)$ is cell $(i, c-1)$,
- ...
- the north-west neighbour of cell $(0, 0)$ is cell $(r-1, c-1)$.

Figure 3 represents such a torus universe with $r = c = 3$. The dashed arrows and boxes at the borders represent the connexions that make this universe a torus.

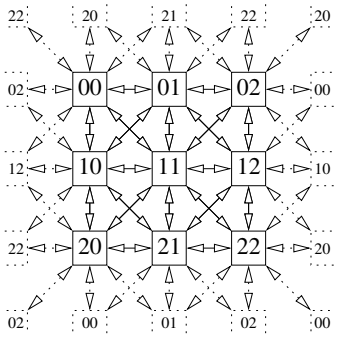


Figure 3: A $3 \times 3$ torus universe

In order to load an initial state and to read out the current state of your universe, equip it with a $r$-bits input `di` and a $r$-bits output `do`, such that:

- `do` is wired to the current state of the cells of the rightmost column,
- when `load` is high, the cells of the leftmost column are initialized with `di` instead of the states of the cells of the rightmost column.

The VHDL source code of `UNIVERSE` uses generic and generate statements as shown on listing 11. Note the handling of the loading / unloading, thanks to an extra matrix column (#$-1$), a missing `mod c` in the wiring of west inputs (line #27) and the two concurrent signal assignments (lines #30 to #32).

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity universe is
  generic(r, c: positive);
  port(clk, srstn, ce, load: in std_ulogic;
       di: in  std_ulogic_vector(0 to r - 1);
       do: out std_ulogic_vector(0 to r - 1));
end entity universe;

architecture rtl of universe is
  subtype column is std_ulogic_vector(0 to r - 1);
  type matrix is array(-1 to c) of column; -- one extra column
  signal states: matrix;
begin
  grows: for row in 0 to r - 1 generate
    gcols: for col in 0 to c - 1 generate
      icell: entity work.cell
        port map(clk, srstn, ce, load,
                 nw => states ((col-1) mod c) ((row-1) mod r),
                 n  => states  (col)           ((row-1) mod r),
                 ne => states ((col+1) mod c) ((row-1) mod r),
                 e  => states ((col+1) mod c)  (row),
                 se => states ((col+1) mod c) ((row+1) mod r),
                 s  => states  (col)           ((row+1) mod r),
                 sw => states ((col-1) mod c) ((row+1) mod r),
                 w  => states  (col-1)         (row), -- no "mod c"
                 state  => states(col)(row));
      end generate gcols;
    end generate grows;
    states(-1) <= di when load = '1' else -- loading
                  states(c - 1);          -- torus
    do <= states(c - 1); -- outputs
end architecture rtl;
```

Listing 11: The UNIVERSE VHDL source code