EDC: exam

Renaud PACALET

18 June 2014

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?

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.`

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?

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.

[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

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;
...
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;
```

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, 9×8, 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?

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.

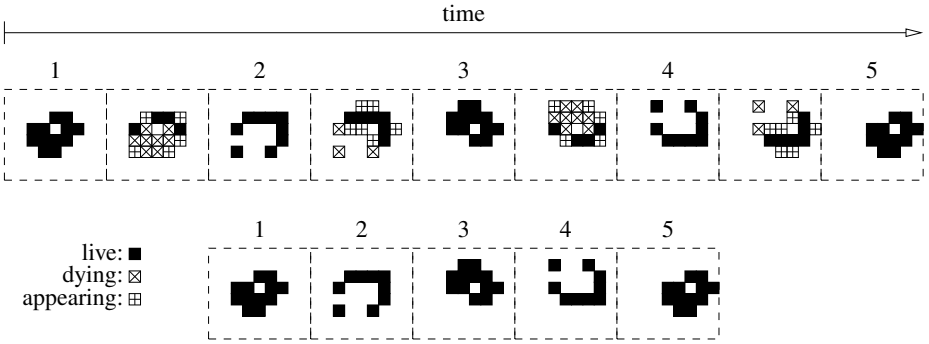

Figure 1: Five consecutive states of the Lightweight spaceship

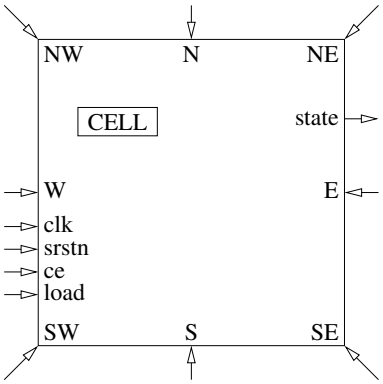

Figure 2: Interface specification of the `CELL` module

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 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.

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

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?

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.

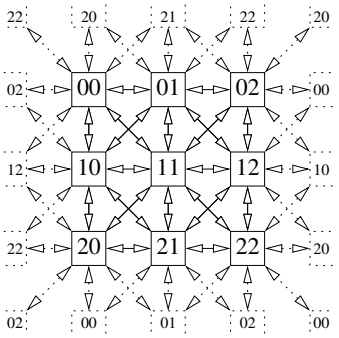

Figure 3: A 3×3 torus universe

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.

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