DigitalSystems: exam

R. Pacalet

2024-06-25

You can use any document but communicating devices are strictly forbidden. Please number the different pages of your paper and indicate on each page your first and last names. You can write your answers in French or in English, as you wish. Precede your answers with the question's number. If some information or hypotheses are missing to answer a question, add them. If you consider a question as absurd and thus decide to not answer, explain why. If you do not have time to answer a question but know how to, briefly explain your ideas. Note: copying verbatim the slides of the lectures or any other provided material is not considered as a valid answer. Advice: quickly go through the document and answer the easy parts first.

The 5 questions are worth 2 points each. The problem is worth 10 points

1 Questions

1.1 Resolved types

Why is it important not to use VHDL resolved types when unresolved types can be used instead?

1.2 Logic synthesis results

After logic synthesis and when investigating the various synthesis reports, why is it important to pay attention to the inferred latches, if any?

1.3 VHDL modeling of state machines

What are the advantages of modelling the states of a Finite-State Machine (FSM) with an enumerated type instead of numbers or vector values? Are there drawbacks?

1.4 VHDL simulation

In your opinion, what will happen when simulating the VHDL model shown on the following listing?

```
entity foo is
end entity foo;
architecture arc of foo is
  signal a, b, c, d, e: bit;
begin
  process(a, b, c)
  begin
    if a = '0' then
      d <= b xor c;
    elsif b = '1' then
      d <= not c;
    else
      d <= c;
    end if;
  end process;
  process(a, d, e)
  begin
    if a = '0' then
      c <= not (d or e);
    elsif d = '0' then
      c <= not e;
    else
      c <= d;
    end if;
  end process;
end architecture arc;
```

We first notice that signals `a`, `b` and `e` are never assigned. So they keep their initial value, that is, the leftmost value of their type declaration, that is, `'0'` for type `bit`. The model of the architecture can thus be simplified as (we replace the sensitivity lists with the equivalent final `wait` statements because it helps understanding the behavior):

```
architecture simplified of foo is
  signal c, d: bit;
begin
  process
  begin
    d <= c;
    wait on c;
  end process;
  process
  begin
    c <= not d;
    wait on d;
  end process;
end architecture simplified;
```

We denote the current simulation time as `t+n` where `t` is the physical time in nanoseconds and `n` is the number of "delta-cycles", that is, the number of simulation steps.

At the beginning of the simulation, at time `0+0`, we have `c='0', d='0'`. The first simulation step assigns `'0'` to `d` and `'1'` to `c`.

At time `0+1` signals `c` and `d` take their assigned values, we have `c='1', d='0'`. Only `c` changes and this triggers again the first process. The first process assigns `'1'` to `d`.

At time `0+2` `d` changes from `'0'` to `'1'`, we have `c='1', d='1'`. This triggers again the second process. The second process assigns `'0'` to `c`.

At time `0+3` `c` changes from `'1'` to `'0'`, we have `c='0', d='1'`. This triggers again the first process. The first process assigns `'0'` to `d`.

At time `0+4` `d` changes from `'1'` to `'0'`, we have `c='0', d='0'`, the initial situation. This triggers again the second process. The second process assigns `'1'` to `c`.

...

This simulation never ends, the time increments only by delta-cycles, the physical time remains constant and equal to `0 ns`, and signals `c` and `d` cycle through the 4 possible combinations, delta-cycle after delta-cycle. What we modeled is a combinatorial loop.

1.5 VHDL synthesis

The VHDL process shown on the listing below is intended to be:

- synthesizable,
- synchronous on the rising edge of `clk`,
- with a synchronous, active low, reset `rstn`.

```
process(clk, di, dsi, rstn)
begin
  do <= '0';
  if rstn /= '0' and clk = '0' and clk'event then
    if dsi = '1' then
      do <= di;
    else
      do <= do;
    end if;
  end if;
end process;
```

What do you think of it? 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.

For synthesizable VHDL, a synchronous process on rising edge of `clk`, with synchronous active low reset `rstn` shall have only `clk` in the sensitivity list and

its body shall have the following structure (using the **event** attribute, as in the given code, instead of the **rising_edge** function):

```
process(clk)
begin
  if clk = '1' and clk'event then
    if rstn = '0' then
      -- reset behavior
    else
      -- other behavior
    end if;
  end if;
end process;
```

So the errors and their consequences are:

- One statement (`do <= '0';`) out of the topmost `if ... end if` block: this is not allowed in a fully synchronous process. During simulations this will assign `'0'` to `do` on any value change of `clk`, `di`, `dsi` or `rstn` while, according the specifications, `do` shall change only on rising edges of `clk`. Assuming it accepts this code, the logic synthesizer will probably implement something with a different behavior from that of the given VHDL code and from the expected one.
- Other signals than `clk` in sensitivity list: any value change of `di`, `dsi` or `rstn` resumes the process while it should resume only on `clk` value changes.
- The process assigns `do` on **falling** edges of `clk` for which `rstn` is not equal to `'0'`, while, according the specifications, `do` shall change only on rising edges of `clk`. Assuming it accepts this code, the logic synthesizer will probably infer a falling-edge-triggered D-flip-flop instead of a rising-edge-triggered one.

Note: if a signal is not assigned during an execution of a process, it keeps its current value. So, assigning a signal value to itself (`do <= do`) to keep its value is useless, but it is not an error. In some cases it could even be considered as good practice because it makes the intention explicit.

If the intention was to model a rising-edge-triggered (by clock `clk`) D-flip-flop with active low synchronous reset `rstn`, input `di`, and enable `dsi`, a better code would be:

```
process(clk)
begin
  if clk = '1' and clk'event then
    if rstn = '0' then
      do <= '0';
    else
      if dsi = '1' then
        do <= di;
      end if;
    end if;
  end if;
end process;
```

An even better code would use the **rising_edge** function introduced with VHDL 2008, and the **elsif** clause instead of a second **if** block nested in the **else** clause:

```
process(clk)
begin
  if rising_edge(clk) then
    if rstn = '0' then
      do <= '0';
    elsif dsi = '1' then
      do <= di;
    end if;
  end if;
end process;
```

Note: with a recent enough synthesizer a concurrent signal assignment could also be used instead of a process:

```
do <= rstn and di when rising_edge(clk) and
                       (rstn = '0' or dsi = '1');
```

2 Problem: Design of an interrupts controller

An interrupts controller is simple in principle but its implementation in a dedicated piece of hardware poses some interesting challenges.

In this small problem you will design a simple interrupts controller that could be used in a computer system. An interrupts controller is a hardware device that receives interrupts from various devices of the system (timer, cryptographic engine...) in parallel and forwards them, one after the other to the CPU. Figure 1 represents the interrupts controller in its environment, Listing 2 and Table 1 specify its interface.

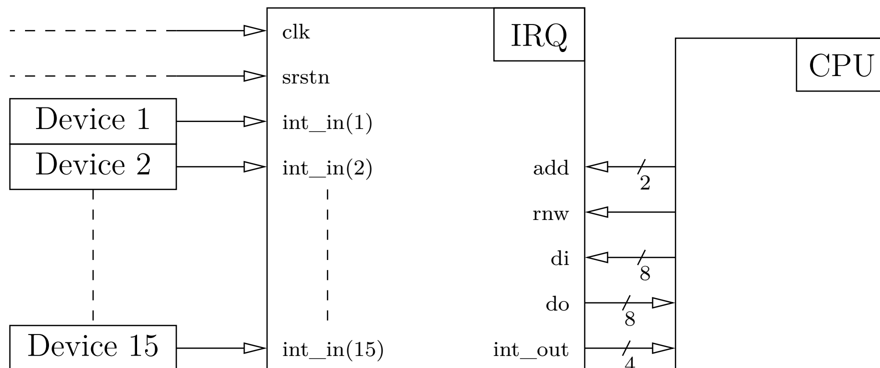

Figure 1: The interrupts controller in its environment

Listing 1: Entity of IRQ

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

entity irq is
  port(clk:      in  std_ulogic;
       srstn:    in  std_ulogic;
       rnw:      in  std_ulogic;
       add:      in  std_ulogic_vector(1 downto 0);
       di:       in  std_ulogic_vector(7 downto 0);
       do:       out std_ulogic_vector(7 downto 0);
       int_in:   in  std_ulogic_vector(15 downto 1);
       int_out:  out std_ulogic_vector(3 downto 0));
end entity irq;
```

Table 1: Interface specification of the IRQ module

Name	Size	Direction	Description
`clk`	1	Input	CLocK. The interrupts controller is synchronous on the rising edge of `clk`.
`srstn`	1	Input	Synchronous ReSeT-Not. Synchronous, active low reset. When low on a rising edge of `clk`, force the content of all internal registers to a predefined value.
`rnw`	1	Input	Read-Not-Write. When high on a rising edge of `clk` the register at address `add` is read and its value is put on `do`. When low on a rising edge of `clk` the register at address `add` is written with the value carried by input bus `di`.
`add`	2	Input	ADDress. Indicates which half of the 2 internal registers is read or written.
`di`	8	Input	Data Input. The 8-bits bus used for write operations.
`do`	8	Output	Data Output. The 8-bits bus used for read operations.
`int_in`	15	Input	INTerrupt INput. The 15 input interrupt lines from up to 15 interrupt sources.

Name	Size	Direction	Description
`int_out`	4	Output	INTerrupt OUTput. The index of the currently pending interrupt with highest priority, if any, else 0.

The **`IRQ`** device is controlled by the CPU through regular read/write operations in one of its two 16-bits registers: **`msk`** and **`pen`**, declared as **`std_ulogic_vector(15 downto 0)`**. The 2 registers are accessed by the CPU at addresses 0, 1, 2 and 3 according the memory map shown in Table 2.

Table 2: Memory map of the IRQ module

Address	Register
0	`msk(7 downto 0)`
1	`msk(15 downto 8)`
2	`pen(7 downto 0)`
3	`pen(15 downto 8)`

`pen(0)` is always `'0'`. The other bits of **`pen`** indicate which interrupts are pending. Warning, read carefully, its behavior is unusual: when **`int_in(i) = '1'`** on a rising edge of **`clk`**, **`pen(i)`** is set to `'1'`. It remains set to `'1'` even if **`int_in(i) = '0'`** on subsequent clock edges. When the CPU writes in **`pen`**, each bit for which the written value is `'0'` is unchanged (not written) and each bit for which the written value is `'1'` is forced to `'0'` (clear-on-set). The software can, for instance, read **`pen`** to get all the pending interrupts and write back the same value to force them all to `'0'`. It can also write `0001000000000100` to force **`pen(12)`** and **`pen(2)`** to `'0'` and leave the other bits unchanged.

`msk(0)` is always `'0'`. The other bits of **`msk`** form a mask that decides which pending interrupts are forwarded to the CPU and which are not. When **`msk(i) = '1'`** the interrupts signaled by **`int_in(i)`** are forwarded. When **`msk(i) = '0'`**, they are masked and thus not forwarded.

When interrupts are pending in the **`pen`** register, **`IRQ`** selects the one with the lowest index that is not masked and encodes the index on **`int_out`**. This signals the interrupt to the CPU. If no interrupts are pending or if all pending interrupts are masked, **`int_out`** carries the all zero value, which indicates to the CPU that there are no interrupts.

Important note: there is no interrupt number 0. Bits 0 of the **`pen`** and **`msk`** registers are unused. Writing in these bits has no effect and reading them always returns `'0'`. For an explanation about how the software running on the CPU can use **`IRQ`** to control up to 15 peripherals, please have a look at the end of this document. You do not need it to solve the problem.

2.1 Architecture design (5 points)

Carefully study the specification and draw a block diagram of the architecture of your **`IRQ`** module. Clearly identify and name the internal registers. Clearly

identify and name the computing elements. If possible, put a kind of pseudo-code in the symbols representing your computing elements. Name all internal signals and specify their bit-widths. Finally, decide how many VHDL processes you will use to code your **IRQ** module, which are synchronous and which are combinatorial and allocate the registers and the computing elements to one of your processes. Make all this specification work clear and easy to understand. Do not leave aspects ambiguous or confusing, the completeness and accuracy of your description is taken into account for the grading.

Our design comprises two 15 bits registers, **msk** and **pen**, and one 8 bits register, **do**; they are the rectangular blocks on Figure 2. There is an ambiguity in the specifications: what happens when a bit of **pen** must be set to **'1'** because an interrupt arrives and, at the same time, set to **'0'** because the CPU writes a **'1'** at this bit position (clear-on-set)? We decide that the set to **'1'** shall have the highest priority.

The other option would also be fine but would be a little less logical: it would sometimes miss an interrupt.

There is another ambiguity about the CPU read accesses: shall the **do** output be updated after the rising edge of **clk** or shall it combinatorially depend on **rnw** and **add**? We decide that **do** is the output of an 8 bits register which content changes only on rising edges of **clk** for which **rnw = '1'**.

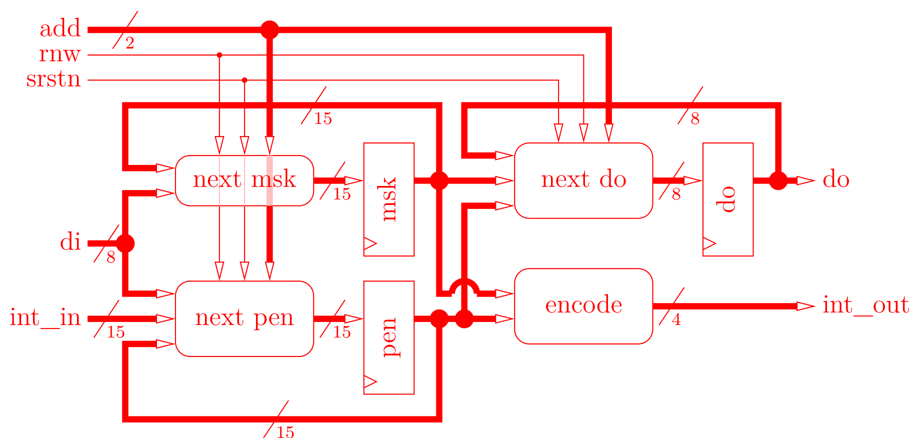

Figure 2: The block diagram

The various combinatorial parts are represented as rounded blocks on Figure 2 and have the following behaviors:

next msk: output bit number $1 \leq i \leq 15$ takes value:

- **'0'** if **srstn = '0'** else
- **di(i mod 8)** if **rnw = '0'** and **add** $= i/8$ (integer division) else
- **msk(i)**.

next pen: output bit number $1 \leq i \leq 15$ takes value:

- **'0'** if **srstn = '0'** else
- **1** if **int_in(i) = '1'** else

- `'0'` if `di(i mod 8) = '1'` and `rnw = '0'` and `add` $= 2 + i/8$ (integer division) else
- `pen(i)`.

`next do`: output bit number $0 \leq i \leq 7$ takes value:

- `'0'` if `srstn = '0'` else
- `do(i)` if `rnw = '0'` else
- `'0'` if `i = 0` and (`add` $= 0$ or `add` $= 2$) else
- `msk(i)` if `add` $= 0$ else
- `msk(8 + i)` if `add` $= 1$ else
- `pen(i)` if `add` $= 2$ else
- `pen(8 + i)`.

`encode`: computes the bitwise AND of `msk` and `pen`, and outputs the index of the rightmost `'1'`, or 0 if there are none, encoded on 4 bits.

The internal signals are only the outputs of the `msk` and `pen` registers and they have the same name as the register. We use one combinatorial process to model the `encode` part. All other parts and the registers are modeled with the same single synchronous process.

2.2 VHDL coding (5 points)

Code, in plain synthesizable VHDL 2008 the architecture of your `IRQ` module.

For the VHDL coding we split the two 15 bits registers `msk` and `pen` in two parts using aliases: the 7 right bits (`mskr`, `penr`) and the 8 left bits (`mskl`, `penl`). We also declare an alias, `di7`, for the 7 left bits of input `di`.

Listing 2: Architecture of IRQ

```
architecture rtl of irq is
  signal msk, pen: std_ulogic_vector(15 downto 1);
  alias mskr: std_ulogic_vector(7 downto 1) is msk(7 downto 1);
  alias mskl: std_ulogic_vector(15 downto 8) is msk(15 downto 8);
  alias penr: std_ulogic_vector(7 downto 1) is pen(7 downto 1);
  alias penl: std_ulogic_vector(15 downto 8) is pen(15 downto 8);
  alias di7: std_ulogic_vector(7 downto 1) is di(7 downto 1);
begin

  process(clk)
    variable a: natural range 0 to 3;
  begin
    if rising_edge(clk) then
      if srstn = '0' then
        msk <= (others => '0');
        pen <= (others => '0');
        do  <= (others => '0');
      else
        a := to_integer(add);
        if rnw = '1' then
          case a is
            when 0 => mskr <= di7;
            when 1 => mskl <= di;
            when 2 => penr <= penr and (not di7);
            when 3 => penl <= penl and (not di);
          end case;
        else
          do <= mskr & '0' when a = 0 else
                mskl       when a = 1 else
                penr & '0' when a = 2 else
                penl;
        end if;
        pen <= pen or int_in;
      end if;
    end if;
  end process;

  process(msk, pen)
  begin
    int_out <= (others => '0');
    for i in 15 downto 1 loop
      if (msk(i) and pen(i)) = '1' then
        int_out <= to_stdulogicvector(i, 4);
      end if;
    end loop;
  end process;

end architecture rtl;
```

2.3 Extra discussion: IRQ usage (no points)

`IRQ` is used by the software running on the CPU to control up to 15 different peripherals. At reset the **`msk`** and **`pen`** registers are cleared. During the boot sequence, the software will write a mask in **`msk`** (in two write operations) to enable the interrupts of the peripherals it wants to control. Each time one of these peripherals raises its interrupt line **`int_in(i)`** (for instance to indicate an error or the end of something it was doing), `IRQ` encodes the interrupt index and forwards it to the CPU. The CPU then:

- disables all interrupts, thanks to an internal flag in its own control registers,
- stores the address of the instruction it was executing,
- jumps to the corresponding software Interrupt Service Routine (ISR) that does whatever is required for this kind of event.

Usually, ISRs are non-re-entrant, that is, they must not be interrupted by themselves. The ISRs thus usually:

- store the current value of **`msk`** somewhere such that it can be later restored,
- mask all interrupts with less or equal priority,
- enable again the interrupts (still with the internal flag of the CPU),
- handle the peripheral that caused the interrupt,
- write in the **`pen`** register a value where all bits are `'0'` except bit number `i` to clear only the pending interrupt,
- restore the masks and return to the interrupted instruction.