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;
```

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.

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 1 and Table 1 specify its interface.

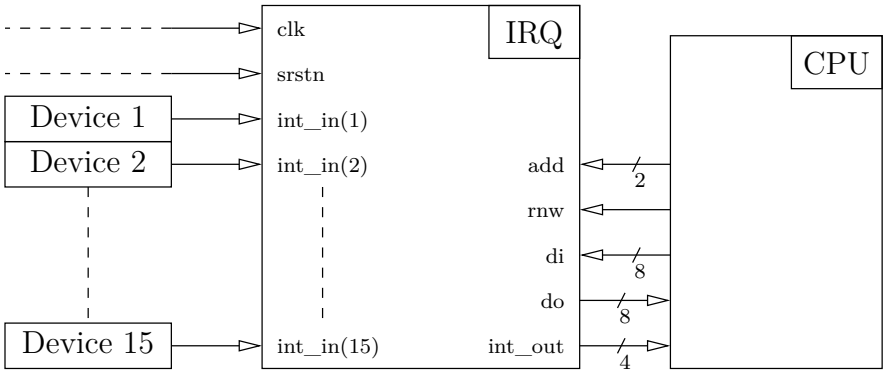

Figure 1: The interrupts controller in its environment

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.
`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.

2.2 VHDL coding (5 points)

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

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.