EDC: exam

Renaud PACALET

19 June 2012

You can use any document you need. On each page of your paper please indicate your name. Put your answers in any order but indicate first the corresponding question number. Example:

```
Question 2.1: I think this is a very good question and...
```

Advice: read every word very carefully and answer first the easy parts. You can write your answers in English or in French, as you like. Do not forget to attach your labs paperwork if you did not send it to me already or send them to me today, sharp deadline.

The five questions of the first part are worth 2 points each. If you consider a question as ambiguous explain why, then make the assumptions needed to solve the ambiguities and answer the modified question. If you consider a question as absurd explain why. If you find an error in a question explain why you think it is a error, propose a correction if it is reasonable and answer the modified question.

The second part is a small problem about an interrupts controller. It is very simple in principle but its implementation in a dedicated piece of hardware poses some interesting challenges. This part is worth 10 points.

1 Questions

1.1. What do you think of the following synchronous VHDL process? 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.

```
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;
```

1.2. What are the VHDL resolved types, what are they used for, where should they be used, where should they not be used and why?

1.3. In your opinion, how many bits of register will be inferred when synthesizing the following VHDL design? Can you draw a schematic of the synthesis result?

```
entity e is
  port(clk: in  bit;
       x:   in  bit_vector(7 downto 0);
       s:   out bit_vector(7 downto 0));
end entity e;

architecture a of e is
begin
  p: process(clk)
    variable v: bit_vector(7 downto 0);
  begin
    if clk = '1' and clk'event then
      s <= v;
      v := x;
    end if;
  end process p;
end architecture a;
```

1.4. Where do you declare VHDL variables (in which VHDL declarative region) and why?

1.5. Translate the following property from natural language to CTL: *If the START input is raised and if it is raised again before the DONE output is raised, then the ERROR signal is raised.*

2 Design of an interrupts controller

In this small problem you will design a simple interrupts controller that could be used in a computer system, like the NiosII system we saw during the labs. An interrupts controller is a hardware device that receives interrupts from various devices of the system (UART, timer,...) in parallel and forwards them, one after the other to the CPU. Figure 1 represents the interrupts controller in its environment.

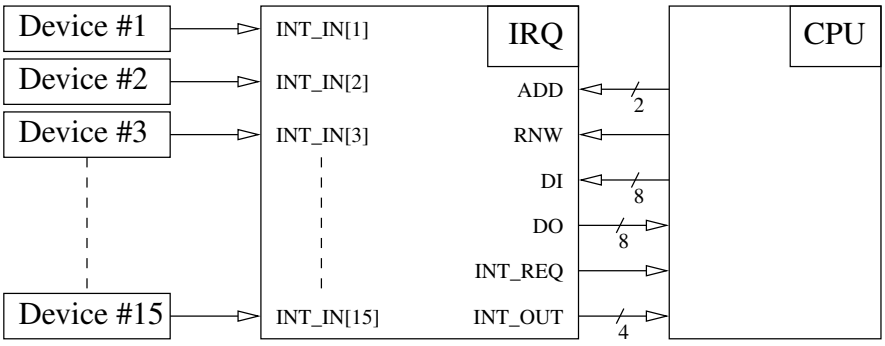

Figure 1: The interrupts controller in its environment

Its complete interface is:

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

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 of the 4 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.

Table 1: Interface specification of the IRQ module

IRQ is controlled by the CPU through regular read/write operations in one of its four 8-bits registers: MSKL(7 downto 0), MSKH(7 downto 0), PENL(7 downto 0) and PENH(7 downto 0), accessed by the CPU at addresses 0, 1, 2 and 3, respectively. MSKL and MSKH form a 16-bits MSK register, MSKL being the rightmost byte and MSKH the leftmost byte (MSK = MSKH & MSKL). Similarly PENL and PENH form the 16-bits PEN register. Bits of MSK and PEN are numbered from 15 for the leftmost bit downto 0 for the rightmost bit. MSK contains a mask that decides which incoming interrupts are forwarded to the CPU and which are not. When MSK(i) = '1' the interrupts signalled by INT_IN(i) are forwarded. When MSK(i) = '0', they are masked and thus not forwarded. PEN is the register indicating which interrupts are pending. Warning, read carefully, its behaviour is unusual: when INT_IN(i) = '1' on a rising edge of CLK, PEN(i) is set. It remains set 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 cleared (clear-on-set). The software can, for instance, read PEN to get all the pending interrupts and write back the same value to clear them all.

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 zeroes value, which indicates to the CPU that there are no interrupts.

Important note: there is no interrupt #0. Bits 0 of the PEN and MSK registers are unused. Writing in these bits have 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.

TODO (5 point): 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 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 very clear and easy to understand.

TODO (5 points): Design, in plain synthesizable VHDL the architecture of your IRQ module.

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,
- 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:

- start by masking all interrupts with less or equal priority,
- enables again the interrupts, still with the internal flag of the CPU,
- handles the peripheral,
- writes in the PEN register a value where all bits are '0' but bit #i to clear the pending interrupt,
- restores the masks and exits.