EDC: exam

Renaud PACALET

28 June 2010

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 multiplication-accumulation (MAC). MACs are used in digital signal processing, for instance when computing a dot product of two vectors or when applying a filter to a sequence of samples. 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. We consider $ABREG$, a small digital circuit, represented on figure 1. A and B are primary inputs. RA and RB are primary outputs. The two registers are regular D flip-flops. They sample their inputs on the rising edge of clock CLK. Express in CTL a temporal-logic property stating that the environment is not allowed to change both A and B in a single clock period.

```
AG(
     (A=1 * B=1 -> AX(A=1 + B=1)) *
     (A=0 * B=1 -> AX(A=0 + B=1)) *
     (A=0 * B=0 -> AX(A=0 + B=0)) *
     (A=1 * B=0 -> AX(A=1 + B=0))
);
```

1.2. We consider INC, a small digital circuit, represented on figure 2. A is a 16 bits primary input vector. SEL is a single bit primary input. DO is a 16 bits primary

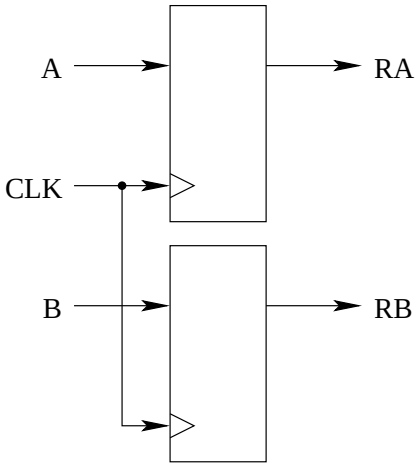

Figure 1: The $ABREG$ digital circuit

output vector. The circuit contains a 16 bits multiplexer, a 16 bits register (that samples its input on the rising edge of clock CLK) and a 16 bits incrementer (a simple combinatorial element that adds 1 to its 16 bits input and outputs the 16 least significant bits of the result). What is the minimum number of processes required to model this circuit in synthesizable VHDL? Explain your answer.

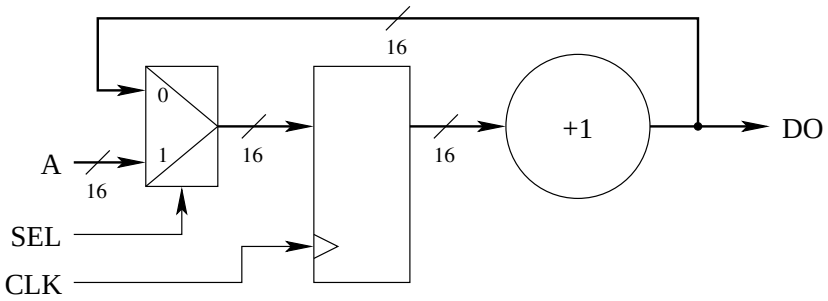

Figure 2: The INC digital circuit

Two processes, minimum, are required: because the circuit contains registers, the VHDl model will contain at least one synchronous process to model them and, possibly, some combinatorial parts which outputs are the registers' inputs. But as the primary output DO is not a register output, it cannot be assigned in this synchronous process. At least one other, combinatorial, process is required. Two processes are sufficient as illustrated by the following VHDL model:

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

entity inc is
  port(clk, sel: in std_ulogic;
       a: in unsigned(15 downto 0);
       do: out unsigned(15 downto 0));
end entity inc;

architecture rtl of inc is
  signal r: unsigned(15 downto 0);
begin

  p1: process(clk)
  begin
    if rising_edge(clk) then
      if sel = '0' then
        r <= r + 1;
      else
        r <= a;
      end if;
    end if;
  end process p1;

  p2: do <= r + 1;

end architecture rtl;
```

Note that, depending on how smart the logic synthesizer is, this model may lead to two different incrementers, which is obviously not necessary. In order to avoid this we could use a third process:

```
architecture rtl of inc is
  signal r, do_local: unsigned(15 downto 0);
begin

  p1: process(clk)
  begin
    if rising_edge(clk) then
      if sel = '0' then
        r <= do_local;
      else
        r <= a;
      end if;
    end if;
  end process p1;

  p2: do_local <= r + 1;

  p3: do <= do_local;

end architecture rtl;
```

1.3. Same question as above about $INCR$, the circuit represented on figure 3.

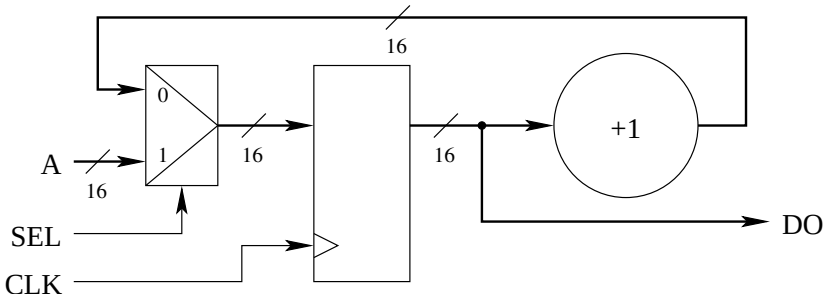

Figure 3: The $INCR$ digital circuit

DO being now a register output, and because there are no other primary outputs, one single synchronous process in sufficient:

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

entity incr is
  port(clk, sel: in std_ulogic;
       a: in unsigned(15 downto 0);
       do: out unsigned(15 downto 0));
end entity incr;

architecture rtl of incr is
  signal r: unsigned(15 downto 0);
begin

  process(clk)
  begin
    if rising_edge(clk) then
      if sel = '0' then
        r <= r + 1;
        do <= r + 1;
      else
        r <= a;
        do <= a;
      end if;
    end if;
  end process;

end architecture rtl;
```

But, here again, the synthesizer could infer twice the required minimum hardware. A solution with two processes would be:

```
architecture rtl of incr is
  signal r: unsigned(15 downto 0);
begin

  p1: process(clk)
  begin
    if rising_edge(clk) then
      if sel = '0' then
        r <= r + 1;
      else
        r <= a;
      end if;
    end if;
  end process p1;

  p2: do <= r;

end architecture rtl;
```

1.4. In your opinion, what will happen when synthesizing a design that contains the following VHDL process?

```
process(a, b)
begin
  if sel = '0' then
    d <= a;
  elsif sel = '1' then
    d <= b;
  end if;
end process;
```

The synthesizer will probably issue a warning about signal sel missing in the sensitivity list of the process. And, depending on the type of signal sel and how smart the synthesizer is, it could also issue a warning about inferred latches: if sel can take more than the two values $'0'$ and $'1'$, then d will not be assigned a value in every execution of the process. A better VHDL code for this process would thus be:

```
process(a, b, sel)
begin
  if sel = '0' then
    d <= a;
  else
    d <= b;
  end if;
end process;
```

1.5. In case a software implementation of a given algorithm is not fast enough for a given application field, what are the different possibilities you know that could be tried to solve the issue?

- Use the optimization options of the compiler
- Use a better software implementation (and, of course, the optimization options of the compiler)
- If parallelization is an option, add other CPUs and split the workload between them
- If possible, add custom instructions to the CPU, through a dedicated hardware co-processor
- If possible, add a hardware accelerator in the system, map it in the CPU's address space and rely on it to compute the algorithm

2 Design of a multiplier-accumulator

We want to use the NiosII CPU to perform some digital signal processing. The data we manipulate the most frequently are vectors of complex numbers. Each component of a vector is represented on 32 bits, the 16 leftmost bits representing the real part and the 16

rightmost bits representing the imaginary part. The real and imaginary parts are signed integers in the range $[-32767...32767]$, represented in two's complement. Reminder: the two's complement representation of an n bits integer is defined by equation (1) in which B is the integer value and b_{n-1} (b_0) is the leftmost (rightmost) bit of its two's complement representation .

$$B = -b_{n-1} \times 2^{n-1} + \sum_{i=0}^{i=n-2} b_i \times 2^i \tag{1}$$

Unfortunately the NiosII CPU is not very efficient at computing on vectors of complex numbers and the performance of a pure software implementation would not be sufficient for our needs. We thus decide to equip the NiosII CPU with a dedicated co-processor named CPXMAC. The interface complies with the co-processor interface of a NiosII CPU and is represented in figure 4 and table 1. The VHDL code for the entity of CPXMAC is given below.

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

entity cpxmac is
  generic(nwidth: natural);
  port(clk, clk_en, reset, start: in std_ulogic;
       dataa, datab: in signed(31 downto 0);
       n: in std_ulogic_vector(nwidth - 1 downto 0);
       done: out std_ulogic;
       result: out signed(31 downto 0));
end entity cpxmac;
```

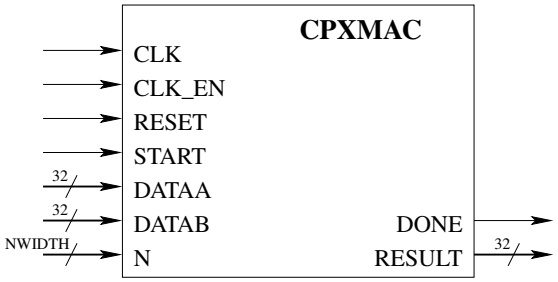

Figure 4: Interface specification of the CPXMAC co-processor

CPXMAC will allow the CPU to compute the product of two complex numbers in one cycle only. It will also allow to accumulate the consecutive results in an internal register.

CPXMAC contains a combinatorial complex multiplier (CPXMUL) which entity is given below.

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

Name	Size	Direction	Description
CLK	1	Input	Clock. CPXMAC is synchronous on the rising edge of CLK
CLK_EN	1	Input	Clock enable. When low, all internal registers are frozen and hold their current value
RESET	1	Input	Synchronous, active high reset. When high on a rising edge of CLK for which CLK_EN is also high, force the content of all internal registers to a predefined value
START	1	Input	Start computation. When high on a rising edge of CLK for which CLK_EN is also high and RESET is low, launches the command specified by N
DATAA	32	Input	Data input. A complex number represented as specified above
DATAB	32	Input	Data input. A complex number represented as specified above
N	NWIDTH	Input	Command selector. The command executed depends on the value of N when the START is given
DONE	1	Output	End of computation. When high on a rising edge of CLK for which CLK_EN is also high and RESET is low, indicates that the previous command is over and its result available on RESULT
RESULT	32	Output	Data output. The result of the last command

Table 1: Interface specification of the CPXMAC module

```
use ieee.numeric_std.all;

entity cpxmul is
  port(dataa, datab: in signed(31 downto 0);
       result: out signed(xx - 1 downto 0));
end entity cpxmul;
```

CPXMUL multiplies its two inputs and puts the result on its XX bits $result$ output. The inputs representation is the one already presented. The result is represented in a similar way: the leftmost half is the real part and the rightmost half is the imaginary part. Reminder: the product of two complex numbers is defined by equation (2) in which j represents the square root of -1.

$$(a + j \times b) \times (c + j \times d) = (a \times c - b \times d) + j \times (a \times d + b \times c) \quad (2)$$

TODO (1 point): considering the range of the values at the input side, how many bits are needed to represent the result (that is, what is the value of XX)?

The product of two integers in the range:

$$[-32767...32767] = [-2^{15} + 1...2^{15} - 1]$$

is in the range:

$$[-2^{30} + 2^{16} - 1...2^{30} - 2^{16} + 1] = [-1073676289...1073676289]$$

So, 31 bits are sufficient to represent it. According to equation (2), the real and the imaginary parts of our result are the sum (or difference) of two such products. One extra bit is thus required to accommodate the new range:

$$[2 \times (-2^{30} + 2^{16} - 1)...2 \times (2^{30} - 2^{16} + 1)] = [-2147352578...2147352578]$$

All in all, 64 bits are needed and sufficient to represent the result. $XX = 64$.

TODO (2 points): Design, in plain synthesizable VHDL the architecture of CPXMUL.

```
architecture rtl of cpxmul is
begin
  process(dataa, datab)
    variable ar, ai, br, bi: signed(15 downto 0);
    variable rr, ri: signed(31 downto 0);
  begin
    ar := dataa(31 downto 16);
    ai := dataa(15 downto 0);
    br := datab(31 downto 16);
    bi := datab(15 downto 0);
    rr := ar * br - ai * bi;
    ri := ar * bi + ai * br;
    result <= rr & ri;
  end process;
end architecture rtl;
```

CPXMAC contains an accumulator, that is, a register which size is sufficient to store the dot product of two complex vectors. The largest vectors we consider are 4096 components vectors. Reminder: the dot product of two vectors is defined by equation (3) in which $\vec{x}$ and $\vec{y}$ are two n components vectors ($1 \leq n \leq 4096$) and x_i, for any $0 \leq i < n$, is the i^{th} component of vector $\vec{x}$.

$$\vec{x} \odot \vec{y} = \sum_{i=0}^{i=n-1} x_i \times y_i \tag{3}$$

TODO (1 point): what is the required number of bits for the accumulator?

We need to accumulate up to $4096 = 2^{12}$ complex numbers which real and imaginary parts are in range:

$$[2 \times (-2^{30} + 2^{16} - 1)...2 \times (2^{30} - 2^{16} + 1)]$$

The result will thus be in range:

$$[2^{13} \times (-2^{30} + 2^{16} - 1)...2^{13} \times (2^{30} - 2^{16} + 1)] = [-2^{43} + 2^{29} - 2^{13}...2^{43} - 2^{29} + 2^{13}]$$

44 bits are thus required to represent the real part and 44 bits for the imaginary part. The accumulator will be 88 bits wide.

We want CPXMAC to offer several functions:

- MUL: single product between two complex numbers. The result is stored in the accumulator. The 32 rightmost bits of the imaginary part of the result is put on the $result$ output when this command is executed. The command takes only one cycle from the point of view of the CPU, that is, the $result$ output is combinationally dependent on the $dataa$ and $datab$ inputs. The $done$ signal is raised in the same clock period as $start$.

- MAC: multiplication-accumulation. The product of the two $dataa$ and $datab$ inputs is computed and added to the current content of the accumulator. The result is stored in the accumulator. Reminder: the sum of two complex numbers is defined by equation (4) in which j represents the square root of -1. The 32 rightmost bits of the imaginary part of the result is put on the $result$ output when this command is executed. The command takes only one cycle from the point of view of the CPU, that is, the $result$ output is combinationally dependent on the $dataa$, $datab$ and the current content of the accumulator. The $done$ signal is raised in the same clock period as $start$.

- INIT: initialization of the accumulator. When executed, this command puts zero in the accumulator. $dataa$ and $datab$ are ignored. A zero value is put on $result$. The command takes only one cycle from the point of view of the CPU, that is, the $done$ signal is raised in the same clock period as $start$.

- RDI0...RDIx: read out the imaginary part of the accumulator, 32 bits at a time. Depending on the width of the accumulator there will be more or less such commands. Important note: in case a RDI operation returns less than 32 useful bits, they are right aligned in the 32 bits result and the other bits are filled with zeros if the read value is positive or null, else with ones (sign extension). RDI0 returns the 32 rightmost bits of the imaginary part of the accumulator, RDI1 returns the 32 following bits of the imaginary part, etc. RDIx returns the leftmost bits of

the imaginary part, sign-extended to 32 bits. The command takes only one cycle from the point of view of the CPU, that is, the $done$ signal is raised in the same clock period as $start$.

- RDR0...RDRx: same as RDI0...RDIx but for the real part of the accumulator.

$$(a + j \times b) + (c + j \times d) = (a + c) + j \times (b + d) \tag{4}$$

TODO (2 points): Complete the specifications, decide how many different RDI and RDR commands are needed, decide how many bits are needed for N (that is, what is the value of $NWIDTH$), decide how to encode the different commands on the N input. Finally, draw a block diagram of the architecture of CPXMAC. Clearly define your notations.

The real and the imaginary parts of the complex values stored in the accumulator are up to 44 bits. Two different RDI operations (RDI0 and RDI1) and two different RDR operations (RDR0 and RDR1) are needed. The CPXMAC co-processor will implement a total of 7 different commands. N will be 3 bits wide ($NWIDTH = 3$), which allows up to 8 different behaviors. The coding is given in table 2.

N	Command
000	INIT
001	Reserved (unused)
010	MUL
011	MAC
100	RDR0
101	RDR1
110	RDI0
111	RDI1

Table 2: Coding of the different commands of CPXMAC

A block diagram of CPXMAC is given in figure 5. $CPXADD$ is a complex adder with a 2×32 bits input, a 2×44 bits second input and a 2×44 bits output. $SEL1$ and $SEL2$ are selectors for the two multiplexers. Altogether with the $DONE$ primary output, they are computed by the main controller, $CTRL$.

TODO (3 points): Design, in plain synthesizable VHDL the architecture of CPXMAC.

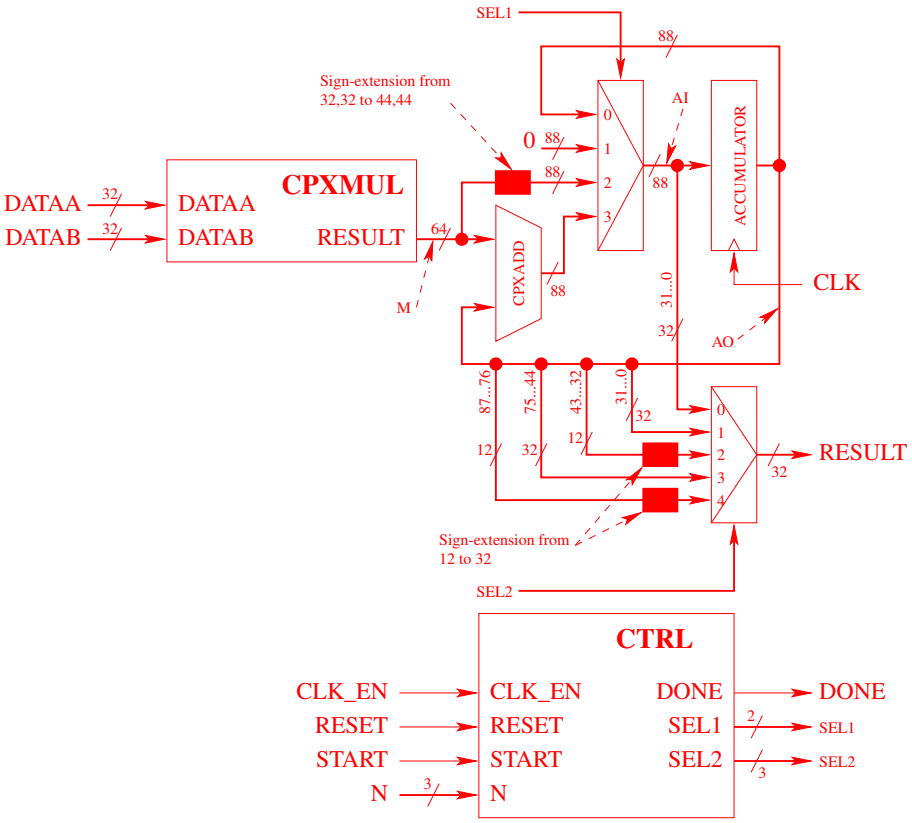

Figure 5: The block diagram of $CPXMAC$

```
architecture rtl of cpxmac is
  signal sel1: natural range 0 to 3;
  signal sel2: natural range 0 to 4;
  signal m: signed(63 downto 0); -- cpxmul output
  signal ai: signed(87 downto 0); -- accumulator input
  signal ao: signed(87 downto 0); -- accumulator output
begin
  ctrl_done: done <= start and clk_en and reset;
  ctrl_sel1: sel1 <= 0 when clk_en = '0' else
                     1 when reset = '1' else
                     1 when start = '1' and n = "000" else
                     2 when start = '1' and n = "010" else
                     3 when start = '1' and n = "011" else
                     0;
  ctrl_sel2: sel2 <= 0 when clk_en = '0' or reset = '1' else
                     1 when start = '1' and n = "110" else
                     2 when start = '1' and n = "111" else
                     3 when start = '1' and n = "100" else
                     4 when start = '1' and n = "101" else
                     0;
  with sel1 select
    ai <= ao when 0,
              (others => '0') when 1,
              resize(m(63 downto 32), 44) &
                resize(m(31 downto 0), 44) when 2,
              (m(63 downto 32) + ao(87 downto 44)) &
                (m(31 downto 0) + ao(43 downto 0)) when 3;
  with sel2 select
    result <= ai(31 downto 0) when 0,
              ao(31 downto 0) when 1,
              resize(ao(43 downto 32), 32) when 2,
              ao(75 downto 44) when 3,
              resize(ao(87 downto 76), 32) when 4;
  mul: entity work.cpxmul(rtl)
    port map(dataa => dataa, datab => datab, result => m);
  process(clk)
  begin
    if rising_edge(clk) then
      ao <= ai;
    end if;
  end process;
end architecture rtl;
```

TODO (1 points): Bonus question: We consider the dot product between two 4096 components vectors. The operands and the final result are stored in the external memory and we assume that each 32 bits read or write operation in this memory take exactly one cycle. On a NiosII CPU estimate how many cycles it would take to compute a dot product between two 4096 components vectors with and without the co-processor.

In both cases there will be 2×4096 read and 4 write operations in external memory, that is, 4100 cycles. With the co-processor, the NiosII CPU will execute one MUL

operation between the two first components, followed by 4095 MAC operations. If the loop over the 4096 components is completely unrolled (no overhead due to a loop index nor a termination test), the total number of cycles will be about $4100 + 4096 = 8196$ cycles. Without the co-processor, and assuming a very optimized assembly code, each of the 4096 complex multiplications will take about:

- 1 cycle per operand to extract the real part and to sign-extend it on 32 bits. A single 16 bits right shift is sufficient.
- 2 cycles per operand to extract the imaginary part and to sign-extend it on 32 bits. A 16 bits left shift followed by a 16 bits right shift is sufficient.
- 4 cycles for the 4 integer multiplications
- 2 cycles for the 2 integer additions / substractions

that is, a total of about 12 cycles per component, and $12 \times 4096 = 49152$ for the 4096 components. The addition of a 32 bits real part and a 44 bits real part will take 1 cycle for the 32 least significant bits, plus 1 cycle to test whether a carry propagates to the most significant bits and 1 more to increment the most significant half or to skip this operation. Same for the imaginary parts, that is, a total of 6 cycles per component and $6 \times 4096 = 24576$ for the whole vector. The grand total will thus be about:

$$8196 + 49152 + 24576 = 81924$$

The co-processor provides about a 10 times acceleration.