Computer Architecture - Lab Assignment 2

Performance evaluation of the memory hierarchy of a computer and reverse engineering of the data cache memory

The main objective of this lab assignment consists in handling different elements of the architecture of the *Basic Computer* that it is configured in the DE0-Nano board [1]. Some of these elements are responsible for implementing various levels of what we call *the Memory Hierarchy*. These levels of the memory hierarchy and their implementations on the DE0-Nano board are the following:

- The level of the memory hierarchy called *Main Memory* can be implemented with DE0-Nano using two different electronic technologies:
  - Using electronic circuits of type *SDRAM* that are located outside of the main FPGA chip of DE0-Nano board where the processor Nios II is integrated (see Figure 1) [3].
  - Using electronic circuits of type *SRAM*. In DE0-Nano exists one SRAM electronic device that is also used in this lab assignment to implement the main memory (see Figure 1). The on-chip SRAM memory is an external memory to the Nios II processor but that is integrated into the same FPGA chip where the Nios II processor is implemented.
- The level of the memory hierarchy called *Cache Memory*:
  - This level is implemented with electronics circuits of type SRAM that are in the same FPGA chip where the Nios II soft processor is integrated (see Figure 1).

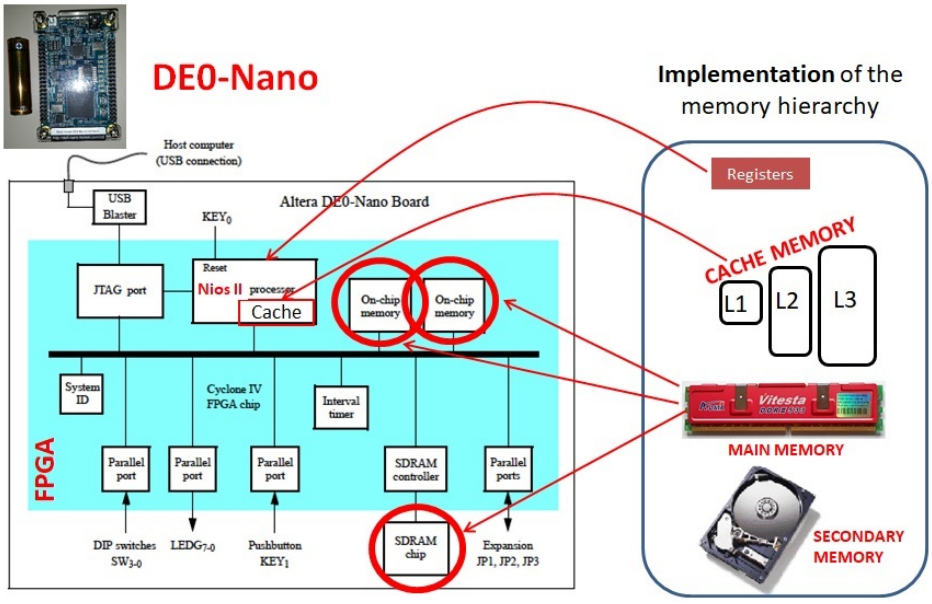

Figure 1: Implementation of the levels of the memory hierarchy for the architecture of the soft Basic Computer when is configured on the DE0-Nano board.

Additionally, three hardware versions of the Nios II processor will be used in this lab assignment. They are called **Nios II/e** (economy), **Nios II/s** (standard) and **Nios II/f** (fast), respectively [2, 4]. Each type of processor and memory cause the execution times of programs to be different.

In this lab experiment, you will measure the execution time of a same program called *Fibonacci* using different configurations of the architecture of the soft Basic Computer that are implemented in the same DE0-Nano board. These configurations are distinguished by the type of processor configured: Nios II/e, Nios II/s or Nios II/f, as well as the type of implementation activated for two levels of the memory hierarchy: cache and main memory can be implemented using on-chip SRAM or external SDRAM technologies.

The engineering methodology employed in this lab consists of four activities that involve the DE0-Nano board and the software tool called *Altera Monitor Program (AMP)*. There will be some questions that request you to justify the results experimentally obtained.

Part I. Memory hierarchy for the Nios II/e soft processor without memory cache and main memory implemented with SDRAM or SRAM technologies

The goal of this first lab exercise consists in measuring the execution time of the Fibonacci program using the external SDRAM memory of the DE0-Nano board, in addition to the Nios II/e soft processor, and the Timer input/output controller. Memory controller for the SDRAM memory, processor and Timer are integrated into the FPGA circuit of the board (see Figure 1).

At the end of the execution of the Fibonacci program, the Nios II/e processor measures the number of 33-ms. intervals that have elapsed. When the Fibonacci program ends, the terminal of the AMP tool shows the number of time intervals as you can see in Figure 2.

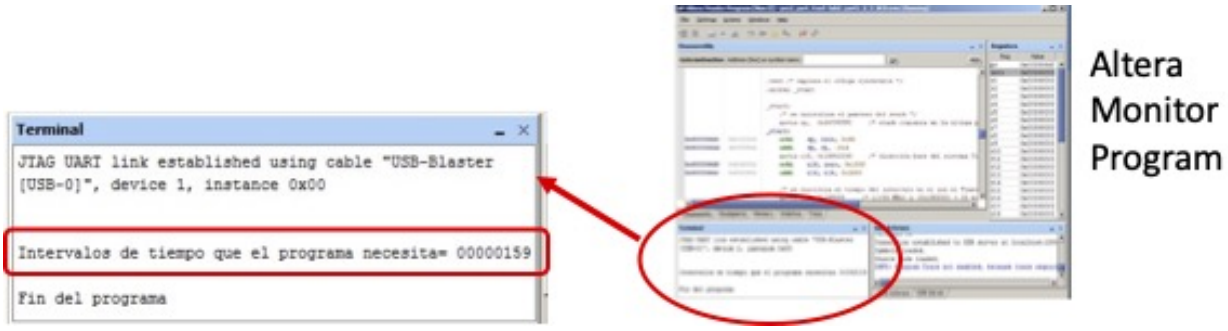

Figure 2: Visualization of the execution time of the Fibonacci program. The value of the number of 33 ms. intervals is shown on the terminal of AMP tool.

The files of assembly source code involved in this part of the lab assignment are named as follows:

- `lab2_part1_2_3_main.s`: main program (see Annex 1).
- `lab2_part1_2_3_fibo.s`: benchmark routine whose execution time is measured (see Annex 2).
- `lab2_part1_2_3_interrupts.s`: interrupt service routine that is invoked when the Timer indicates an interruption event requires attention because a 33-ms. interval has elapsed (see Annex 3).
- `lab2_part1_2_3_excepciones.s`: routine that is called from `lab2_part1_2_3_interrupts.s` for increasing the counter of 33-ms. intervals (see Annex 4).
- `lab2_part1_2_3_JTAG.s`: routine that is called from `lab2_part1_2_3_main.s` to show on the terminal of AMP the number of 33-ms. intervals that have elapsed until the end of the execution of the benchmark program (see Annex 5).
- `lab2_part1_2_3_BCD.s`: routine that is called from `lab2_part1_2_3_JTAG.s` to transform a binary code into BCD format (see Annex 6).

- `lab2_part1_2_3_div.s`: routine that is called from `lab2_part1_2_3_BCD.s` to do the integer division (see Annex 7).

The flow diagram of the benchmark program that we will use to measure the execution time is shown in Figure 3. As we can see, the Timer controller invokes the interrupt service routine every 33 ms (see `lab2_part1_2_3_main.s` and `lab2_part1_2_3_interrupts.s`).

The interrupt service routine allows to save a counter of 33-ms. intervals in one memory address whose pointer is called `COUNTER` (see `lab2_part1_2_3_excepciones.s` file). Each event indicates that one interval of 33 ms. has elapsed since the end of the previous time interval. So, `COUNTER` variable stores the numbers of 33-ms. intervals that have elapsed from the beginning to the end of the benchmark.

In parallel with the time measurement performed by the Timer controller, the main program executes the Fibonacci loop a number of times that is indicated by the constant called `ITERATIONS` (see `lab2_part1_2_3_main.s` file). When this loop ends, the content of the memory position `COUNTER` is read and its value is shown on the terminal of AMP tool (see `lab2_part1_2_3_JTAG.s` file). The binary code saved in `COUNTER` is transformed into BCD format and then, in ASCII code. For generating the BCD code, one or more divisions are needed. In these cases, one routine for divisions is invoked. This is required because the Nios II/e processor has not hardware divisors (see `lab2_part1_2_3_JTAG.s` file).

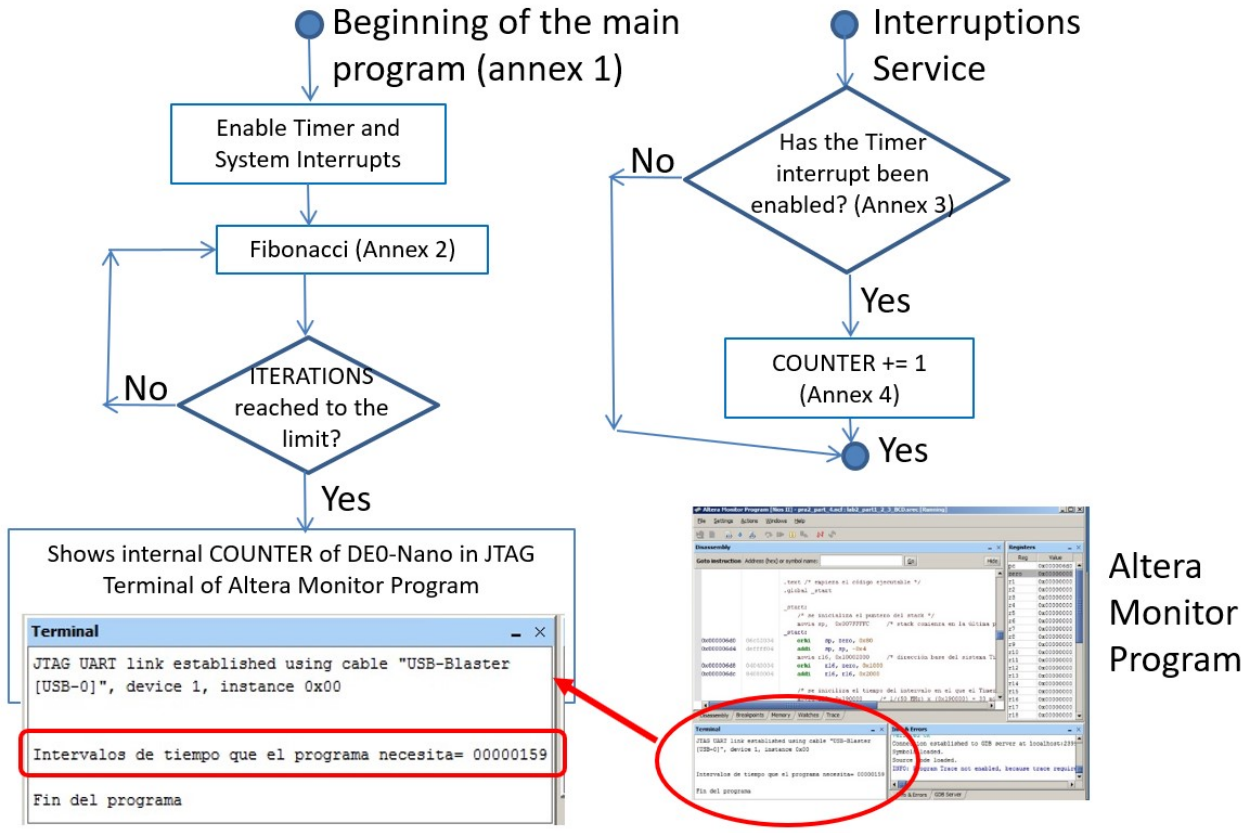

Figure 3: Flow diagram of the benchmark program for Part I. The number of 33-ms. intervals are shown on the AMP terminal.

Lab Methodology for Part I

Create a new project in the software tool called *Altera Monitor Program (AMP)* like it was done in the Lab Assignment 1. The FPGA configuration called *DE0-Nano Basic Computer* has to be selected. Additionally, the seven assembly files (`*.s`) indicated at the beginning of this lab assignment must be included in the AMP project. Furthermore, it is needed to set in the AMP project that the programs and data are stored starting at the memory address `0x400` (`_start`). This is done as follows within the AMP tool:

AMP software tool

```
[Step 1] Settings > System settings > Memory Settings >
.text { memory device = SDRAM/s1; start offset in device (hex) = 400
.data { memory device = SDRAM/s2; start offset in device (hex) = 400

Then, load the configuration into the FPGA of the DE0-Nano board:
[Step 2] Actions > Download System > Download

Afterwards, compile the programs and load the executable file into the memory:
[Step 3] Actions > Continue
```

Now, fill in Table 1 the value of execution time that is shown on the AMP terminal. This value for the execution time is the same value as the number of 33-ms. intervals that have elapsed during the Fibonacci program execution. The selected memory configuration (memory device) as indicated above means that both the instruction (`.text`) and data (`.data`) parts of the program are stored in the device with SDRAM electronic technology.

Table 1: Memory settings of the AMP project for Part I. These settings must be activated in the *Memory settings* tab of the AMP project.

Soft processor / DE0-Nano configuration	`.text`		`.data`		**Execution time**	**Speed-up**
	Memory device	**Start offset in device (hex)**	**Memory device**	**Start offset in device (hex)**		
Nios II/e / DE0-Nano Basic Computer	SDRAM	400	SDRAM	400		1×
	On-chip memory/s1	400	SDRAM	400		
	SDRAM	400	On-chip memory/s1	400		
	On-chip memory/s1	400	On-chip memory/s1	400		

Next, change in the "Memory settings" tab of the AMP project the device assigned to the main memory to hold the instruction part of the program (`.text`) to the device called *on-chip memory*. Next, recompile, link, load into memory and run the benchmark program. Now, register the execution time provided by the AMP terminal in the second entry of Table 1. Additionally, calculate the Speed-up obtained assuming that the reference time is that registered in the first entry of Table 1.

Finally, change the device assigned to the instruction (`.text`) and data (`.data`) parts of the program as indicated in entries 3 and 4 of Table 1. For each one of these two cases, obtain the execution time of the benchmark program and register them in the table. The execution times in Table 1 will be compared with those obtained in the next two parts of this lab assignment.

Next, answer the following two questions related to the work you have developed in this first experimental part.

Question 1

- Which main memory configuration provides the best performance, i.e., the configuration that achieves the lowest benchmark program execution time (see Table 1)?
- What is the reason for this configuration to be the one with the highest performance compared to the other three configurations?

Question 2

Note that the source code in `lab2_part1_2_3_main.s` has a constant declared and initialized: ITERATIONS = 500000. Using the SDRAM device assigned to the instruction (`.text`) and data (`.data`) parts of the program, change ITERATIONS from 500,000 to 100,000 in the `lab2_part1_2_3_main.s` file, recompile and load the program on the board. Then run the program and register the corresponding value displayed on the AMP terminal. Is this new performance result reasonable? Justify your answer.

Part II. Memory hierarchy for the Nios II/s soft processor with instruction cache memory and main memory implemented with SDRAM or SRAM technologies

The goal of this second hands-on activity is to measure the actual execution time of the Fibonacci program using the Nios II/s soft processor. This version of the Nios II processor family provides a higher level of performance than the Nios II/e processor since its data path is pipelined using five stages. Additionally, a first level of instruction cache memory is available. However, unlike the Nios II/f processor that will be used in Part III, the Nios II/s processor has no data cache memory.

Lab Methodology for Part II

For this hands-on activity, you will change the configuration of the DE0-Nano board as shown below using the soft SoC configuration called *DE0-Nano Computer*. Be careful not to confuse it with the configuration from Part I which is called *DE0-Nano Basic Computer*.

Choose the previous project developed in Part I and the on-chip address memory space following the following steps:

AMP software tool

`[Step 1] Settings > System Settings > - Select a system: > DE0-Nano Computer`

`[Steps 2,3,4,5] Next, perform four runs of the benchmark program used in Part I but change the memory devices that hold the instruction (.text) and data (.data) sections of the program. These benchmark runs should be performed after compiling, linking and loading the program into the board memory.`

`Table 2 shows the memory settings established in the` *`Memory settings`* `tab of the AMP monitor program. In the four memory settings of the AMP project it should be indicated that the memory address offset for both the instruction part of the program (.text) and the data part (.data) should be 400 in hexadecimal format.`

In the source code of the program (`lab2_part1_2_3_main.s`) the number of iterations must be 500000. Compile again the assembly files of the AMP project and load the executable file into the main memory of the computer configured in the DE0-Nano board. Then, run the program and fill in Table 2 with the value of the runtime displayed on the AMP terminal.

After each execution of the benchmark program, write down in Table2 the values of the execution time in number of 33 ms. intervals provided by the program through the AMP terminal. Additionally, calculate the corresponding Speed-up obtained for each memory hierarchy configuration using as reference/base time the one obtained in Part I, when using the Nios II/e processor and the `.text` and `.data` addressing space assigned to the external SDRAM device through the *DE0-Nano Basic Computer* configuration of the DE0-Nano board.

Table 2: Memory settings of the AMP project for Part II. These settings must be activated in the *Memory settings* tab of the AMP project.

Soft processor / DE0-Nano configuration	`.text`		`.data`		**Execution time**	**Speed-up**
	Memory device	**Start offset in device (hex)**	**Memory device**	**Start offset in device (hex)**		
Nios II/e / DE0-Nano Computer	SDRAM	400	SDRAM	400		
	On-chip memory/s1	400	SDRAM	400		
	SDRAM	400	On-chip memory/s1	400		
	On-chip memory/s1	400	On-chip memory/s1	400		

Then answer the following two questions related to the work you have developed in this second experimental activity.

Question 3

- Which main memory configuration provides the best performance, i.e., the configuration that achieves the shortest execution time of the benchmark program when using the *DE0-Nano Computer* soft hardware configuration of the DE0-Nano board (see Table 2)?
- What is the reason that justifies that configuration to be the one with the best performance compared to the other three configurations analyzed in Part II?

Question 4

What are the reasons that each of the four main memory configurations analyzed in Part II with the Nios II/s processor provides larger or smaller execution time (see Table 2) than the corresponding main memory configurations analyzed in Part I with the Nios II/e processor (see Table 1)?

Part III. Memory hierarchy for the NiosII/f soft processor with instruction and data cache memory and main memory implemented with SDRAM or SRAM technologies

The goal of this part consists in measuring the real execution time of the Fibonacci program using the fast Nios II/f soft processor. This member of the Nios II processor family provides higher performance level than Nios II/e and Nios II/s processors because it has a microarchitecture with hardware elements that the processor of the two previous activities does not have. The main relevant hardware characteristics of Nios II/f are as follows:

- six-stage pipelined data path
- first-level data cache memory
- first-level instruction cache memory
- dynamic branch predictor

Lab Methodology for Part III

For this part, you will change the configuration of the DE0-Nano board to implement a new soft computer. Please, follow the steps described below.

AMP software tool

```
[Step 1] Select the same AMP project used in Part II:
- Settings > System Settings >

[Step 2] Select a custom FPGA configuration:
- Select a system > <Custom System> >

[Step 3] Change the information file (nios_system.sopcinfo) and the FPGA
configuration file (DE0_Nano_Basic_Computer.sof) for the custom soft computer as
can be seen in Table 5 and Figure 4.
```

Table 3: Information and configuration files for Part III.

Board	Information file	Configuration file
DE0-Nano	`nios_system.sopcinfo`	`DE0_Nano_Basic_Computer.sof`

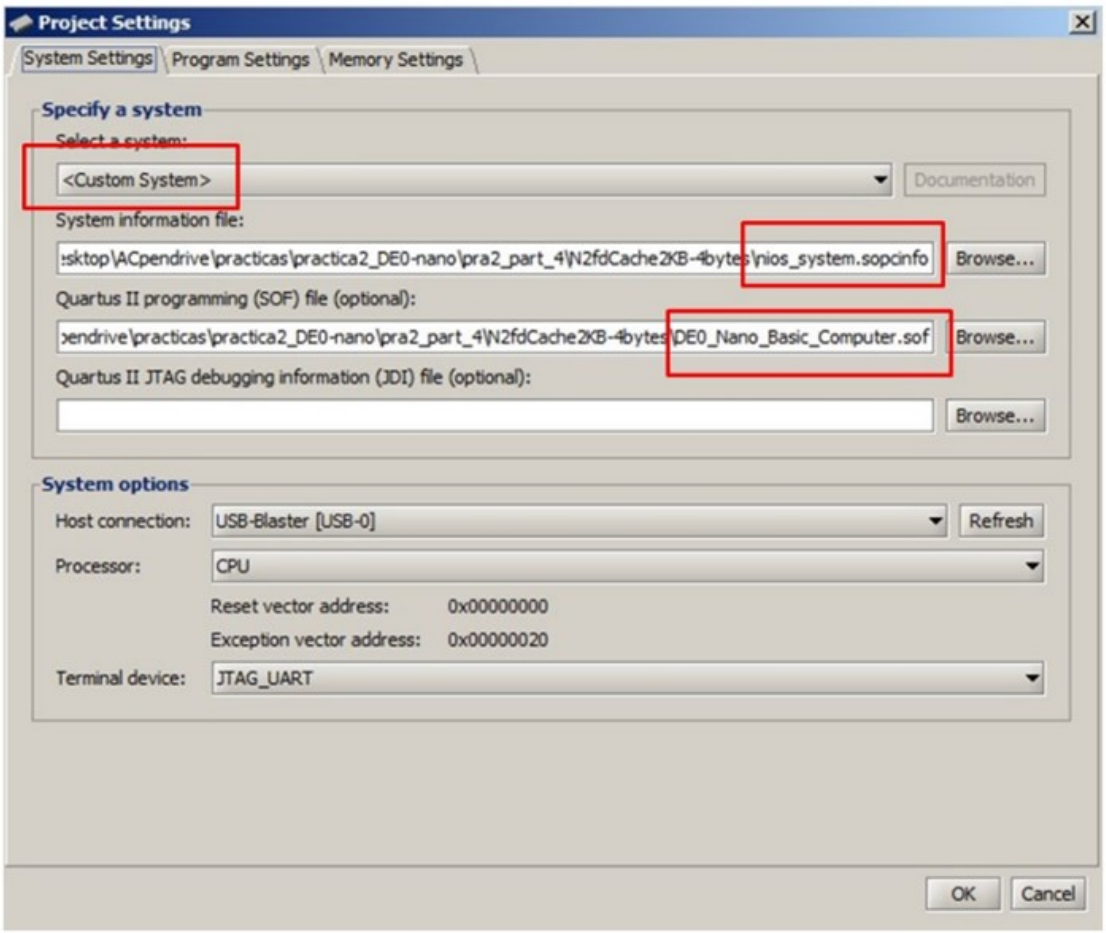

Figure 4: Part of the project where a custom computer architecture based on the Nios II/f is selected.

Additionally, one of the available memory address spaces must be selected from the two alternatives that can be chosen: on-chip or SDRAM. Furthermore, the code part (`.text`) and the data part (`.data`) must start from address `0x400`.

AMP software tool

```
Settings > System settings > Memory settings >
.text { start offset in device (hex) = 400
.data { start offset in device (hex) = 400
```

Next, perform four runs of the benchmark program used in Parts I and II but change the memory devices that hold the `.text` and `.data` parts of the program. These benchmark runs should be performed after compiling, linking and loading the program into the memory of the DE0-Nano board.

Table 4 shows the memory configurations to be set in the *Memory settings* tab of the AMP monitor program. In the four memory settings of the AMP project it should be indicated that the memory address offset for both the instruction part of the program (`.text`) and the data part (`.data`) should be 400 in hexadecimal format.

Table 4: Memory settings of the AMP project for Part III. These settings must be activated in the *Memory settings* tab of the AMP project.

Soft processor / DE0-Nano configuration	`.text`		`.data`		**Execution time**	**Speed-up**
	Memory device	**Start offset in device (hex)**	**Memory device**	**Start offset in device (hex)**		
Nios II/e / Custom System	SDRAM	400	SDRAM	400		
	On-chip memory/s1	400	SDRAM	400		
	SDRAM	400	On-chip memory/s1	400		
	On-chip memory/s1	400	On-chip memory/s1	400		

As in the previous two experimental parts, after each execution of the benchmark program, write down in Table 4 the values of the execution time in number of 33 ms. intervals provided by the program through the AMP terminal. Additionally, calculate the corresponding Speed-up obtained for each memory hierarchy configuration using as reference/base time the one obtained in Part I, when the Nios II/e processor was used and the `.text` and `.data` addressing space assigned to the external SDRAM device through the *DE0-Nano Basic Computer* configuration of the DE0-Nano board.

Finally, answer the following questions:

Question 5

- Which of the four main memory configurations analyzed in this Part III is the one that provides the best performance, i.e., the configuration that achieves the lowest benchmark program execution time (see Table 4)?
- What is the reason for this result?

Question 6

- Which of the processor-memory hierarchy configurations analyzed in Parts I, II and III provides the best performance (see Tables 1, 2, 4)?
- What are the reasons for this result?

Part IV. Discovering the microarchitecture of the data cache memory

In this part, the data cache memory of the Nios II/f soft processor will be analyzed without any knowledge of its microarchitecture. To discover the cache capacity and block size of the microarchitecture of the data cache memory of the Nios II/f processor we will use a simple program traversing a vector of bytes called `V`. For this goal, follow the following steps:

- Reduce the iteration number of the main program from 500000 to 50000 in the file `lab2_part1_2_3_main.s`.
- Modify as shown below the subroutine Fibonacci. The new function of the routine is simply to access the components of a vector of bytes `V`.

```
        ...
        movi    r4,    0
        movi    r5,    X
LOOP:   bge     r4,    r5,     END
        ldb     r0,    V(r4)
        addi    r4,    r4,     P
        br      LOOP
END:
        ...
.data
V:
.skip   65536
        ...
```

  Note that in the code above shown there are two parameters that need values to be assigned: **X** and **P**. **X** represents the number of elements of the vector `V` that are accessed with stride **P**. The stride **P** is the number of elements of the vector `V` that are in memory between two successive accesses using the `ldb` instruction. A new parameter called `E` is defined to represent the number of elements of the vector actually accessed with the `ldb` instruction. Thus, $\mathbf{X} = \mathbf{P} \times \mathbf{E}$.

- Table 5 will be used to register the execution times obtained in the same way as in the previous parts, measuring the execution times of the benchmark program. This time is mostly due to the one needed to traverse part of the elements of the vector `V` with stride `P`:
  - Stride one to one (**P** $\rightarrow$ 1: `addi r4, r4, 1`)
  - Stride two to two (**P** $\rightarrow$ 2: `addi r4, r4, 2`)
  - Stride four to four (**P** $\rightarrow$ 4: `addi r4, r4, 4`)
  - Stride eight to eight (**P** $\rightarrow$ 8: `addi r4, r4, 8`)
  - Stride sixteen to sixteen (**P** $\rightarrow$ 16: `addi r4, r4, 16`)
  - Stride thirty-two to thirty-two (**P** $\rightarrow$ 32: `addi r4, r4, 32`)

  The number of elements of `V` that are necessary to do `E` accesses will be `X` for each different stride. For example, for the first column of Table 5, the values for **X**, **E** and **P** are the following:

  - **X**$\rightarrow$128 for **E**$\rightarrow$128, **P**$\rightarrow$1 (`movi r5, 128`)
  - **X**$\rightarrow$256 for **E**$\rightarrow$128, **P**$\rightarrow$2 (`movi r5, 256`)
  - **X**$\rightarrow$512 for **E**$\rightarrow$128, **P**$\rightarrow$4 (`movi r5, 512`)
  - **X**$\rightarrow$1024 for **E**$\rightarrow$128, **P**$\rightarrow$8 (`movi r5, 1024`)
  - **X**$\rightarrow$2048 for **E**$\rightarrow$128, **P**$\rightarrow$16 (`movi r5, 2048`)
  - **X**$\rightarrow$4096 for **E**$\rightarrow$128, **P**$\rightarrow$32 (`movi r5, 4096`)

  Note in this example that for all values of **X**, the iteration number of the loop in the program is **E**=128. For this reason, $\mathbf{X} = \mathbf{P} \times \mathbf{E}$. Additionally, note that the number of access to memory and the number of `ldb` instructions executed are also 128.

- Fill in Table 5 with the data obtained in the previous point and draw a graphic as shown in Figure 5. For filling in Table 5, please follow the next procedure:

Table 5: Execution times for Part IV to discover the microarchitecture of the data cache memory.

Stride (`P`)	E=126	E=256	E=512	E=1024	E=2048	E=4096
P=1						
P=2						
P=4						
P=8						

1. Open AMP.
2. Create a new project.

CHANGE CACHE

3. Settings > System Settings > System information file + browse > `nios_system.sopcinfo`
4. Settings > System Settings > Quartus II Programming file + browse > `DE0_Nano_Basic_Computer.sof`
5. Actions > Download system.

CHANGE DATA

6. Modify the program file: `lab2_part1_2_3_main.s` to set new values for **X** and **P**.
7. Compile.
8. Load.
9. Run the program and wait for the time to appear on the AMP terminal and register its value in Table 5.

CONTINUE CHANGE DATA

CONTINUE CHANGE CACHE

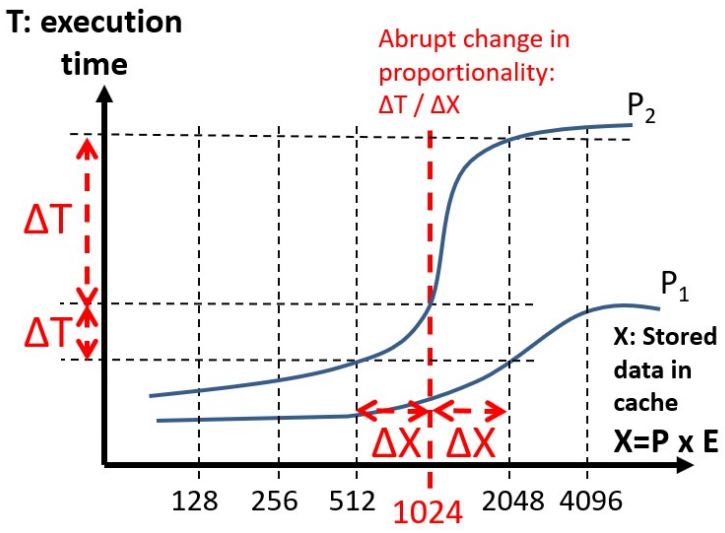

Figure 5: Relationship between the execution time of the program (`T`) and the number of elements stored in data cache, `X`. Note that `X` is obtained multiplying the number of `E` elements of the `V` vector accessed and the stride, `P`, of these accesses.

- Taking the data previously obtained in Table 5 and looking at Figure 5, deduce and discover the total capacity of the data cache memory and its block size. Please, justify all answers. Help: Note that in lectures we explained and calculated the concept *Average Memory Access Time*: $AMAT = time_{hit} + MissRate \times Penalty$.

Annexes

The assembly source code used in this lab assignment is provided in Annexes 1 to 7.

Annex 1. `lab2_part1_2_3_main.s`: this is the main program of the benchmark for all parts of the lab assignment.

Annex 2. `lab2_part1_2_3_fibo.s`: this file contains the computing routine whose execution time is measured in the main program.

Annex 3. `lab2_part1_2_3_interrupts.s`: interrupt service routine that is invoked when the Timer I/O controller asserts the interrupt request line. In this case, this event requires attention from the processor because a 33 ms. interval has finished.

Annex 4. `lab2_part1_2_3_excepciones.s`: routine that is called from `lab2_part1_2_3_interrupts.s` for increasing the counter variable that stores the number of 33-ms. intervals.

Annex 5. `lab2_part1_2_3_JTAG.s`: routine that is called from the main program `lab2_part1_2_3_main.s` to show on the terminal of AMP the number of 33-ms. intervals that have elapsed until the end of the execution of the benchmark program.

Annex 6. `lab2_part1_2_3_BCD.s`: routine that is called from `lab2_part1_2_3_JTAG.s` to transform a binary code into BCD format.

Annex 7. `lab2_part1_2_3_div.s`: routine that is called from `lab2_part1_2_3_BCD.s` to do the integer division.

References

[1] Altera. Basic Computer System for the Altera DE-0 Nano Board, Altera Corporation - University Program. `ftp://ftp.intel.com/pub/fpgaup/pub/Intel_Material/12.1/Computer_Systems/DE0-Nano/DE0_Basic_Computer.pdf`, 2012.

[2] Intel. Introduction to Altera Nios II soft processor, Altera Corporation - University Program. `https://ftp.intel.com/Public/Pub/fpgaup/pub/Teaching_Materials/current/Tutorials/Nios2_introduction.pdf`, 2019.

[3] Altera. Using the SDRAM Memory on Altera´s DE2 Board with Verilog Design, Altera Corporation - University Program. `https://ftp.intel.com/Public/Pub/fpgaup/pub/Teaching_Materials/current/Tutorials/Nios2_introduction.pdf`, 2009.

[4] Altera. Nios II Processor Reference Handbook, Altera Corporation. `https://ftp.intel.com/Public/Pub/fpgaup/pub/Teaching_Materials/current/Tutorials/Nios2_introduction.pdf`, 2014.

Annex 1

```
/*********************************************************************************
* lab2_part1_2_3_main.s
* Main program for Nios II-based Lab Assignment 2 of Computer Architecture course
* Initialize the Timer system
* Initialize and activate the interrupt system of the Nios II processor
* Execute a loop that calls Fibonacci routine and show the number of 33-ms.  intervals
* Subroutines:  PRINT_JTAG (lab2_part1_2_3_JTAG.s), FIBONACCI (lab2_part1_2_3_fibo.s),
*********************************************************************************/
.equ ITERATIONS, 500000
.text /*the executable code starts*/
.global _start
_start:
      /* the stack pointer is initialized */
      movia sp, 0x007FFFFC /* stack starts in last memory position of SDRAM */
      movia r16, 0x10002000 /* base address of the internal Timer system */
      /*the time of the interval in which the timer generates an interrupt for
      performance analysis is started*/
      movia r12, 0x190000 /* 1/(50 MHz) x (0x190000) = 33 ms.  */
      sthio r12, 8(r16) /* saves the half of the word from the initial value of timer*/
      srli r12, r12, 16 /* shifts the 16 bits value to the right */
      sthio r12, 0xC(r16) /* saves the top half word for the initial value of timer */
      /*the Timer is initialized, enabling its interrupts*/
      movi r15, 0b0111 /* START = 1, CONT = 1, ITO = 1 */
      sthio r15, 4(r16)
      /*Nios II processor interrupt is enabled*/
      movi r7, 0b011 /* the interrupt bit mask is initialized for level 0 (Timer) and level 1 (pushbuttons)*/
      wrctl ienable, r7
      movi r7, 1
      wrctl status, r7 /* Nios II interrupts are activated */
      movia r14, ITERATIONS /* initializes the Fibonacci iteration counter */
      addi r17, r0, 0 /*initializes the interval counter of the program "r17"/
LOOP:
      beq r14, r0, END /* the Fibonacci loop is executed */
      call FIBONACCI
      addi r14, r14, -1
      br LOOP
END:
      movi r7, 0
```

```
      wrctl status, r7 /* interrupt processing is disabled in Nios II */
      call PRINT_JTAG /* the number of 33-ms.  intervals is displayed on the AMP terminal */
      IDLE: br IDLE /* the main program finishes */
.data
.global COUNTER
COUNTER:
      .skip 4 /* memory addresses that save the counter of 33-ms.  intervals */
.end
```

Annex 2

```
/*****************************************************************************
* lab2_part1_2_3_fibo.s
* Subroutine:  this routine executes the Fibonacci Series computation for 8 numbers
* Called from:  lab2_part1_2_3_main.s
*****************************************************************************/
.text
.global FIBONACCI
FIBONACCI:
      subi sp, sp, 24 /* reserve space for the stack */
      stw r4, 0(sp)
      stw r5, 4(sp)
      stw r6, 8(sp)
      stw r7, 12(sp)
      stw r8, 16(sp)
      stw r9, 20(sp)

      movia r4, N /* r4 points to N address */
      ldw r5, (r4) /* r5 is the counter initialized with the value stored in N */
      addi r6, r4, 4 /* r6 points to the first Fibonacci number */
      ldw r7, (r6) /* r7 contains the first Fibonacci number */
      addi r6, r4, 8 /* r6 points to the second Fibonacci number */
      ldw r8, (r6) /* r7 contains the second Fibonacci number */
      addi r6, r4, 0x0C /* r6 points to the third Fibonacci number */
      stw r7, (r6) /* Save the third Fibonacci number */
      addi r6, r4, 0x10 /* r6 points to the fourth Fibonacci number */
      stw r8, (r6) /* Save the fourth Fibonacci number */
      subi r5, r5, 2 /* Decrease the number of values saved in 2 */
LOOP:
      beq r5, r0, STOP /* Finishes when r5 = 0 */
      subi r5, r5, 1 /* Decrement the counter */
      addi r6, r6, 4 /* Increment the list pointer */
      add r9, r7, r8 /* adds two previous numbers */
      stw r9, (r6) /* saves the result */
      mov r7, r8
      mov r8, r9
      br LOOP
STOP:
      ldw r4, 0(sp)
```

```
      ldw r5, 4(sp)
      ldw r6, 8(sp)
      ldw r7, 12(sp)
      ldw r8, 16(sp)
      ldw r9, 20(sp)
      addi sp, sp, 24 /* releases the reserved stack */
      ret
.data
N:
      .word 8 /* 8 Fibonacci Numbers */
NUMBERS:
      .word 0, 1 /* First and second numbers */
RESULT:
      .skip 32 /* Space for 8 numbers of 4 bytes */
.end
```

Annex 3

```
/********************************************************************************
* subroutine:  lab2_part1_2_3_interrupts.s
* The AMP program (Altera Monitor Program) finds out the section ".reset"
* in the memory address that is set in the Nios II hardware configuration
* by the SOPC Builder tool.
* "ax" is needed to indicate that this section is reserved and executed
********************************************************************************/
.section .reset, "ax"
movia r2, _start
jmp r2 /* jump to the main program */

/********************************************************************************
* The AMP program (Altera Monitor Program) finds out the section ".exceptions"
* in the memory address that is set in the Nios II hardware configuration
* by the SOPC Builder tool.
* "ax" is needed to indicate that this section is reserved and executed
* Subroutines:  INTERVAL_TIMER_ISR (lab2_part1_2_3_excepciones.s)
********************************************************************************/
.section .exceptions, "ax"
.global EXCEPTION_HANDLER
EXCEPTION_HANDLER:
      subi sp, sp, 16 /* reserve the stack */
      stw et, 0(sp)
      rdctl et, ctl4
      beq et, r0, SKIP_EA_DEC /* interrupt is not external */
      subi ea, ea, 4 /* ea register must be decreased by 1 instruction */
SKIP_EA_DEC:
      /* For external interruptions, so that the interrupted instruction will be executed */
      /* after eret (Exception Return) */
      stw ea, 4(sp) /* save registers in the stack */
      stw ra, 8(sp) /* required if a call has been used */
      stw r22, 12(sp)
      rdctl et, ctl4
      bne et, r0, CHECK_LEVEL_0 /* the exception is an external interrupt */
NOT_EI:
/* exception for not implemented instructions or TRAPs */
      br END_ISR
CHECK_LEVEL_0:
/* Timer has Level 0 interrupt */
```

```
      call INTERVAL_TIMER_ISR
      br END_ISR
END_ISR:
      ldw et, 0(sp) /* restore previous registers values from stack */
      ldw ea, 4(sp)
      ldw ra, 8(sp)
      ldw r22, 12(sp)
      addi sp, sp, 16
eret
.end
```

Annex 4

```
/*********************************************************************************
* lab2_part1_2_3_excepciones.s
* Subroutine that increases an interval counter of the Timer
* Called from:  lab2_part1_2_3_interrupts.s
*********************************************************************************/
.extern COUNTER
.global INTERVAL_TIMER_ISR
INTERVAL_TIMER_ISR:
      subi sp, sp, 8 /* reserved memory space for the stack */
      stw r10, 0(sp)
      stw r11, 4(sp)
      movia r10, 0x10002000 /* base address of Timer */
      sthio r0, 0(r10) /* initialize to 0 the interrupt */
      movia r10, COUNTER /* base address of the Timer interval counter */
      ldw r11, 0(r10)
      addi r11, r11, 1 /* adds the timer interval counter */
      stw r11, 0(r10)
      ldw r10, 0(sp)
      ldw r11, 4(sp)
      addi sp, sp, 8 /* release the stack */
      ret
.end
```

Annex 5

```
/*****************************************************************
* lab2_part1_2_3_JTAG.s
* Subroutines related to the display of a character on the terminal
* input parameters:
* r10 = ascii value of the character to be shown
*****************************************************************/
.extern COUNTER /* variable defined in the main program */

/*****************************************************************
Subroutine:  PRINT_JTAG
Displays on AMP JTAG terminal the contents of the external memory address COUNTER
*****************************************************************/
.global PRINT_JTAG
PRINT_JTAG:
      subi sp, sp, 24 /* stack management */
      stw r2, 4(sp)
      stw r3, 8(sp)
      stw r4, 12(sp)
      stw r10, 16(sp)
      stw r17, 20(sp)
      stw ra, 24(sp)
      movia r3, TEXT
      call WRITE_TEXT_JTAG /* show on JTAG terminal a fixed text */
      movia r17, COUNTER /* base address of the Timer interval counter */
      ldw r4, 0(r17)
      call BCD /* input:  r4= binary value, output:  r2= BCD value */
      call WRITE_VALUE_JTAG /* show on the JTAG terminal the BCD value */
      movia r3, TEXT_FIN
      call WRITE_TEXTO_JTAG /* show on the JTAG terminal a fixed text */
      ldw r2, 4(sp) /* stack management */
      ldw r3, 8(sp)
      ldw r4, 12(sp)
      ldw r10, 16(sp)
      ldw r17, 20(sp)
      ldw ra, 24(sp)
      addi sp, sp, 24
      ret

/*****************************************************************
```

```
Subroutine:  WRITE_TEXT_JTAG
Show a string of characters on the JTAG terminal
input parameter:  r3, string pointer
*****************************************************************/
.global WRITE_TEXT_JTAG
WRITE_TEXT_JTAG:
      subi sp, sp, 12
      stw r3, 4(sp)
      stw r10, 8(sp)
      stw ra, 12(sp)
BUC:
      ldb r10, 0(r3) /* loads one byte from the base address of the character string */
      beq r10, r0, WITH /* if reads a 0, it means that r3 has reached the end of the chain
      and jumps out of the loop */
      call WRITE_JTAG /* subroutine that shows the byte on the JTAG-UART terminal */
      addi r3, r3, 1 /* next byte */
      br BUC /* close the loop */
WITH:
      ldw r3, 4(sp)
      ldw r10, 8(sp)
      ldw ra, 12(sp)
      addi sp, sp, 12
      ret

/*****************************************************************
Subroutine:  WRITE_VALUE_JTAG
Show a BCD value on the JTAG terminal
input parameters:  r2, BCD value
*****************************************************************/
.global WRITE_VALUE_JTAG
WRITE_VALUE_JTAG:
      subi sp, sp, 16
      stw r2, 4(sp)
      stw r4, 8(sp)
      stw r10, 12(sp)
      stw ra, 16(sp)
      addi r4, r0, 8 /* 8 nibbles for the BCD value of the counter */
VALUE:
      andhi r10, r2, 0xf000 /* extracts the 4 more significant bytes of the BCD value */
      srli r10, r10, 28 /* r10 is shifted 28 bits to the right */
```

```
      addi r10, r10, 0x30 /* add 0x30:  BCD -> ASCII */
      call ESCRIBIR_JTAG /* shows ASCII value */
      subi r4, r4, 1 /* counter of nibbles */
      slli r2, r2, 4 /* next nibble of BCD value */
      bne r4, r0, VALUE
      ldw r2, 4(sp)
      ldw r4, 8(sp)
      ldw r10, 12(sp)
      ldw ra, 16(sp)
      addi sp, sp, 16
      ret
.global WRITE_JTAG
WRITE_JTAG:
      subi sp, sp, 12 /* the used registers are saved in the stack */
      stw r3, 4(sp)
      stw r22, 8(sp)
      stw ra, 12(sp)

      movia r22, 0x10001000 /* base address of JTAG I/O controller */
AGAIN:
/* while loop:  check if there is space to write */
      ldwio r3, 4(r22) /* read register of JTAG-UART controller */
      andhi r3, r3, 0xffff /* selects the 16 more significative bits */
      beq r3, r0, AGAIN /* WSPACE=0?  */
WRT:
/* show the character on the terminal by writing in the JTAG-UART controller */
      stwio r10, 0(r22)
END:
      ldw r3, 4(sp) /* retrieve the registers from the stack and return */
      ldw r22, 8(sp)
      ldw ra, 12(sp)
      addi sp, sp, 12
      ret
/* Data zone */
TEXT:
.asciz "\n\n Time intervals that the program needs= "
TEXT_END:
.asciz "\n\n End of the program "
.end
```

Annex 6

```
/*********************************************************************************
* lab2_part1_2_3_BCD.s
* Transform the binary code into BCD value
* Called from:  lab2_part1_2_3_JTAG.s
* Subroutine:  DIV (lab2_part1_2_3_div.s)
* input argument:  r4= binary value
* output result:  r2= BCD value
*********************************************************************************/
.text
.global BCD
BCD:
      subi sp, sp, 24 /* stack management */
      stw r3, 0(sp)
      stw r4, 4(sp)
      stw r5, 8(sp)
      stw r6, 12(sp)
      stw r10, 16(sp)
      stw r31, 20(sp) /* saved due to nested subroutines */
      beq r4, r0, END /* if binary == 0 goto END */
      addi r5, r0, 10 /* r5 = 10 for dividing the BCD value */
      add r6, r0, r0 /* i = 0 */
      add r10, r0, r0 /* r10 = 0 */
LOOP2:
      bge r0, r4, END /* while binary value > 0 */
      call DIV /* calls division with r4 = dividend, r5 = divisor; returns r3= quotient, r2= remainder
*/
      sll r2, r2, r6 /* shifts the result 4 bits to the left except for the first number */
      or r10, r10, r2 /* accumulates the result in r10 */
      addi r6, r6, 4 /* updated r6 += 4 */
      bgt r5, r3, END /* if quotient < 10 goto END */
      add r4, r3, r0 /* r4 = previous quotient */
      jmpi LOOP2 /* if quotient > = 10 goto LOOP2 */
END:
      sll r3, r3, r6 /* shifts the final quotient several 4 bits to the left */
      or r10, r10, r3 /* accumulates the result in r10 */
      add r2, r10, r0 /* puts the result in the output register r2 */
      ldw r3, 0(sp)
```

```
      ldw r4, 4(sp)
      ldw r5, 8(sp)
      ldw r6, 12(sp)
      ldw r10, 16(sp)
      ldw r31, 20(sp)
      addi sp, sp, 24 /* free the reserved stack */
      ret
.end
```

Annex 7

```
/*****************************************************************************
* lab2_part1_2_3_div.s
* Integer division for Nios II processors
* Called from:  lab2_part1_2_3_JTAG.s
* input arguments:  r4= dividend, r5= divisor
* output results:  r2= remainder, r3= quotient
*****************************************************************************/
.text
.global DIV
DIV:
      subi sp, sp, 16 /* reserve memory space for the stack */
      stw r6, 0(sp)
      stw r7, 4(sp)
      stw r10, 8(sp)
      stw r11, 12(sp)
      beq r5, r0, END /* if divisor == 0 goto END */
START:
      add r2, r4, r0 /* remainder = dividend */
      add r6, r5, r0 /* r6 = next_multiple = divisor */
      add r3, r0, r0 /* quotient = 0 */
LOOP:
      add r7, r6, r0 /* r7 = multiple = next_multiple */
      slli r6, r7, 1 /* next_multiple = left_shift(multiple,1) */
      sub r10, r2, r6 /* r10 = remainder - next_multiple */
      sub r11, r6, r7 /* r11 = next_multiple - multiple */

      blt r10, r0, LOOP2 /* if r10 < 0 goto LOOP2 */
      bgt r11, r0, LOOP /* if r11 > 0 goto LOOP */
LOOP2:
      bgt r5, r7, END /* while divisor <= multiple */
      slli r3, r3, 1 /* quotient << 1 */
      bgt r7, r2, MOVE /* if multiple <= remainder */
      sub r2, r2, r7 /* then remainder = remainder - multiple */
      addi r3, r3, 1 /* quotient += 1 */
MOVE:
      srli r7, r7, 1 /* multiple = right_shift(multiple, 1) */
      jmpi LOOP2
END:
      ldw r6, 0(sp)
```

```
      ldw r7, 4(sp)
      ldw r10, 8(sp)
      ldw r11, 12(sp)
      addi sp, sp, 16 /* free stack memory */
      ret
.end
```