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 practice consists of manipulate different elements of the architecture of the Basic Computer that it is configured in the DE0-Nano board and they are responsible for implementing various levels of what we call Memory Hierarchy. These levels and implementations of the memory hierarchy on DE0-Nano are the following:

- The level of the main memory can be implemented with DE0-Nano using two different electronic technologies:
  - With electronic circuits of type SDRAM that are located outside of the chip where is the processor (See figure 1).
  - With electronic circuits of type SRAM. In DE0-Nano exists one device SRAM that is used in this practice to implement the main memory (See figure 1).
    - The on-chip memory is an external memory to the processor but that it is in the same chip where is located the processor which is called FPGA (Field Programmable Gate Array)
- The level of the cache memory:
  - This level is implemented with electronics circuits of type SRAM that are in the same chip where is located the processor (FPGA, see figure 1).

*AC 40969 - https://www.english.ulpgc.es/

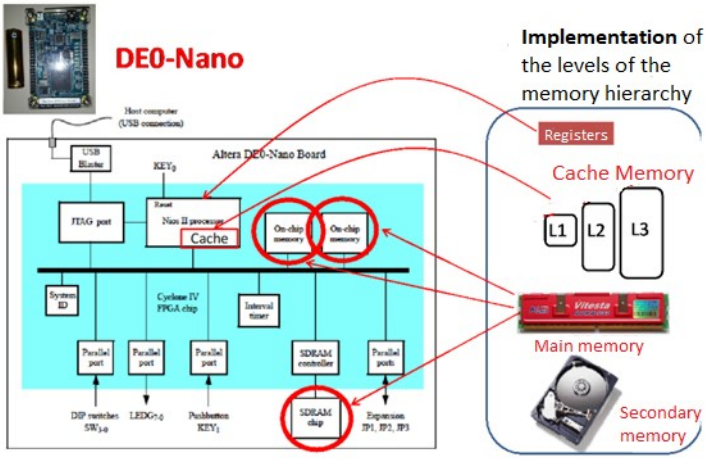

Figure 1. Implementation of the levels of the memory hierarchy of the architecture of the Basic Computer of the DE0-Nano board

In addition, in this practice it will be use two hardware versions of the NIOS II processor that are called **NIOSII/e** (economy) and **NIOSII/f** (fast), respectively. Each type of processor and memory cause that the times of the executions of programs be different.

In this practice it will be carry out the measure of time of execution of a same program called Fibonacci when it is executed with different configurations of the architecture of the Basic Computer in the DE0-Nano Board. These configurations are distinguished in the type of processor configured: NIOS II/e or NIOS II/f, as well as in the type of the implementation activated for the levels of the memory hierarchy: cache, main memory (SRAM or SDRAM).

The methodology of practices consists of four activities with the DE0-Nano board and the software environment Altera Monitor Program (AMP) of Altera. In the practical exercises it will be request think the results that you have experimentally obtained.

Exercise 1. Access to the memory SDRAM with the NIOS II/e processor (economy)

The objective of this first activity consists of measure the real time of computation of the Fibonacci program using the SDRAM memory of the DE0-Nano board, also of the NIOS II/e, and the Timer which is an input/output device that joins to the hardware components mentioned (see Figure 1).

During the execution of the Fibonacci program, the NIOS II processor counts the number of intervals of 33 ms that have passed. When the Fibonacci program finish, the terminal what is found in the AMP application shows the number of complete intervals of 33 ms that have passed, as you can see in Figure 2.

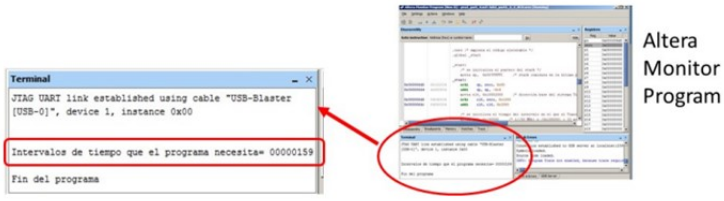

Figure 2. Visualization of the final result of the execution of the program used to measure the time of execution of the programs. The value of the number of intervals of 33 ms appears in the terminal´s application AMP.

The assembly programs involved in this activity of the practice are:

- lab2_part1_2_3_main.s: main program (see Annex 1).
- lab2_part1_2_3_fibo.s: computing benchmark routine of which aims to measure the time of computation (see Annex 2).
- lab2_part1_2_3_interrupts.s: routine which is executed when the interruption of the Timer is active (see Annex 3).
- lab2_part1_2_3_excepciones.s: routine which is called from lab2_part1_2_3_interrupts.s and that manages the counter of time of the execution (see Annex 4).
- lab2_part1_2_3_JPEG.s: routine which is called from lab2_part1_2_3_main.s to show in the terminal of AMP the number of intervals that have passed until the end of the execution of the benchmark program (see Annex 5).
- lab2_part1_2_3_BCD.s: routine which is called from lab2_part1_2_3_JTAG.s to transform a binary code in BCD (see Annex 6).
- lab2_part1_2_3_div.s: routine that is called from lab2_part1_2_3_BCD.s to perform the integer division (see Annex 7).

The flow of the benchmark program that we will use to measure the time of execution is show in Figure 3. As we can see, the computer application activates the interrupts system of the Timer in the main program (see lab2_part1_2_3_main.s).

The interrupt service routine allows maintain a counter of events in one position of memory called COUNTER (see lab2_part1_2_3_excepciones.s file). Each event consist of the indication that has passed one interval of 33 ms since the previous interval of time. So, COUNTER records the numbers of intervals of 33 ms that have passed since the Timer was activated.

Concurrently with the time measurement performed by the Timer, the main program executes the Fibonacci´s loop a number of times that is indicated by the constant called ITERATIONS (see lab2_part1_2_3_main.s file). When the loop finishes, the content of the position of the memory COUNTER is consulted and its value is show in the terminal of the application AMP (see lab2_part1_2_3_JTAG.s file). For this last process, the binary code is transformed. This binary code represents the value of COUNTER in a number encoded in BCD and, then, in an ASCII code. In the transformation to BCD code is necessary make one o various divisions, for which is include one routine of division because the NIOS II/e processor has not have the hardware to make division (see lab2_part1_2_3_JTAG.s file).

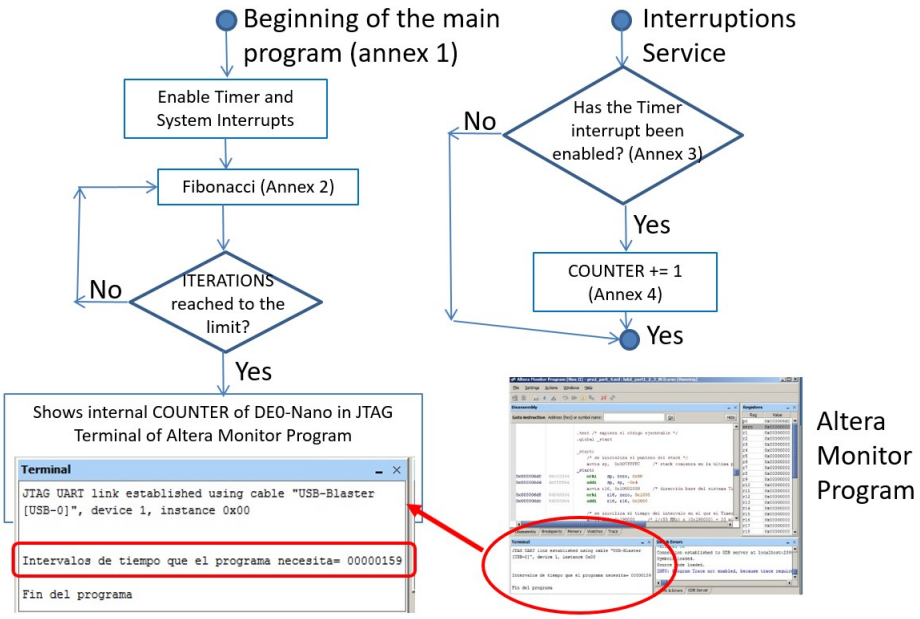

Figure 3. Flow diagram of the program of Practice 2.

Methodology of practices

Start a new project in the software environment Altera Monitor Program (AMP) in the same way made in the previous practice, selecting the configuration DE0-Nano Basic Computer, and including the seven files in assembly (*.s) indicated at the beginning of this practical activity. In addition, specify in the AMP project that the programs and data are stored starting in the direction of the space of addressing in memory 0x400 (_start). This is done as follows within AMP:

- 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 in the DE0-Nano board:

- Actions > Download System > Download

Afterwards, compile the programs and load in the memory:

- Actions > Continue

Following this practice, write down in the Table 1 the value of time of the execution that is shown in the AMP terminal. This time is the same value of the number of intervals of 33 ms that have passed during the Fibonacci program execution. The time of execution will be compared with the time that will be obtained in the next two activities of this practice. Focus on the lab2_part1_2_3_main.s program has declared and initialized a constant: ITERATIONS = 500000.

Change ITERATIONS a 100000 in the lab2_part1_2_3_main.s file, compile again the program and load it in the board. Then, execute the program and write down the value that is shown in the AMP terminal.

Question 1

Is this new result of temporary benefits reasonable? Reason and justify answer.

Exercise 2. Access to the on-chip memory with the NIOS II/e processor

The objective of this second activity consists of measure real computation time of the Fibonacci program using the SRAM memory that is located inside the chip (called on-chip) where is also the NIOS II/e processor. The difference with the exercise 1 is that is used a SRAM memory that was built with other technology and also is located physically nearer the processor, instead of using the external memory SDRAM of processor that is found welded in the DE0-Nano board.

Methodology of practices

Select the previous project in the activity 1 and the address space on-chip following the next way:

- Settings > System Settings > Memory Settings >

.text - memory device = onchip memory/s1(0x9000000 - 0x9001FFF)

.text – start offset in device (hex) = 400

.data – memory device = onchip memory/s2 (0x9000000 – 0x9001FFF)

.data – start offset in device (hex) = 400

In the source code of the program (lab2_part1_2_3_main.s file) the number of iterations must be keep at 500000.

Compile again the assembly files of the project AMP and load the executable in the DE0-Nano board. Then, run the program and write down in Table 1 the value of the runtime displayed in the AMP terminal.

Question 2

Is this new result of temporary benefits reasonable? Reason and justify answer using the diagram in Figure 1 at the beginning of this document.

Table 1. Measurement of benefits

Processor version + memory technology	Execution time	Speed-up
Nios II/e + SDRAM memory (Exercise 1)		1X
Nios II/e + on-chip memory (Exercise 2)		
Nios II/f + SDRAM memory (Exercise 3)		
Nios II/f + on-chip memory (Exercise 3)		

Exercise 3. Access to the cache memory of the NIOS II/f(fast) processor

The objective of this activity consists of measure the real computation time of the program Fibonacci using the NIOS II/f processor. This version of the NIOS II processor provides a higher level of temporary benefits because it implements different characteristics that the processor of the two previous activities does not have. NIOS II/f characteristics: a six-stage segmented data path, data cache memory and first-level instructions, and dynamic branch prediction.

Methodology of practices

For this practical activity, we will proceed to change the configuration of the board DE0-Nano following way indicated below.

Select the AMP project of the exercise 2:

- Settings > System Settings >

Select a personalized configuration:

- Select a system > <Custom System> >

Change the files of description (nios_system.sopcinfo) and the configuration of the FPGA of the Altera board (DE0_Nano_Basic_Computer.sof) as can be seen in Figure 4.

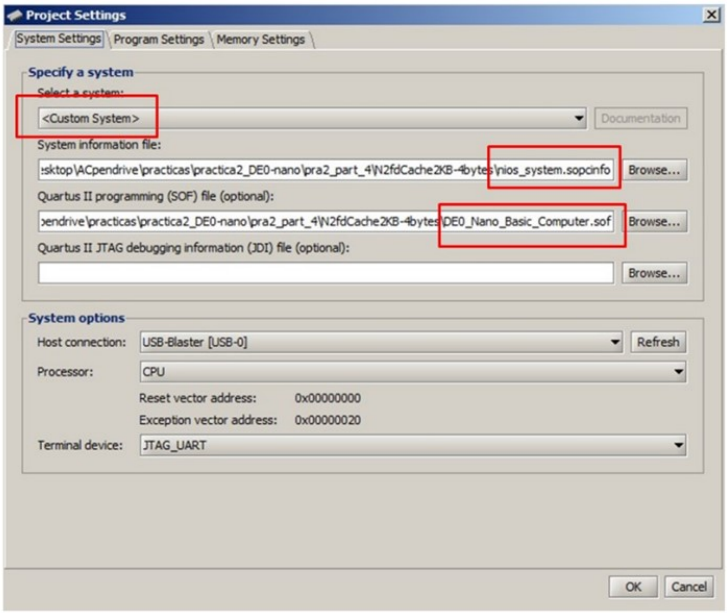

Figure 4. Part of the project where is selected a computer architecture based on the NIOS II/f.

Table 2. Files of configuration in the DE2 board

Board	SOPCINFO file	SOF file
DE0-Nano	nios_system.sopcinfo	DE0_Nano_Basic_Computer.sof

Note: This files of configuration are available in the compressed file lab2.rar (see "Material of the practice 2" in the Moodle page)

Additionally, any type of address space must be selected from the three alternatives that can be chosen (on-chip, SDRAM), with the restriction that the code part (.text) and data part (.data) start from address 0x400

-Settings > System settings > Memory settings >

.text – start offset in device (hex) = 400

.data – start offset in device (hex) = 400

Write down in the table 1 the number of intervals of 33 ms (execution time) that appear in the AMP terminal al the final of each execution of the program. Calculate the Speed-up that is obtained with the different versions of the NIOS II processor using as benchmark the Fibonacci program and considering as base system (Speed-up = 1X) to the NIOS II/e processor which address space is assigned to the external memory SDRAM of the DE0-Nano board. Finally, answer the following questions:

Question 3

What is the reason that different versions of the NIOS II/e processor provide such variation of benefits?

Question 4

What is the reason that the two versions of the NIOS II/f processor are equal in the execution time?

Question 5

What is the reason that the NIOS II/e version provides worse performance than NIOS II/f?

Exercise 4. Discovery of the internal architecture of the cache memory of data

To discover the storage capacity and block size of the architecture from the data cache memory of the NIOS II/f processor we will use a simple program traversing a vector of bytes (V). The steps to follow in this practical activity are the following:

- Reduce the iterations number of the main program from 500000 to 50000 (file: lab2_part1_2_3_main.s)
- Modify by the way that is shown below the subroutine code of Fibonacci to limit simply to traversing a vector V of bytes.

```
        ...
        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
...
```

  Observe that in the code above there are two parameters that need values: **X** and **P**. **X** represents the number of elements of the vector V that are going to be used to access them with a standard **P**. The standard **P** is the number of elements of the vector V that are in memory between two successive accesses with the ldb instruction. A new parameter is defined called E which represents the number of elements of the vector actually accessed with the ldb instruction. Thus, **X = P × E**.

- Then, Table 3 will be written with execution times values obtained in the same way as in the previous activities, 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 a jump standard (P):
  - One to one (**P** $\rightarrow$ 1: addi r4, r4, 1)
  - Two to two (**P** $\rightarrow$ 2: addi r4, r4, 2)
  - Four to four (**P** $\rightarrow$ 4: addi r4, r4, 4)
  - Eight to eight (**P** $\rightarrow$ 8: addi r4, r4, 8)
  - Sixteen to sixteen (**P** $\rightarrow$ 16: addi r4, r4, 16)
  - Thirty-two to thirty-two (**P** $\rightarrow$ 32: addi r4, r4, 32)

The number of elements V that are necessary to make the E access will be X for each one of this standards. For example, for the first column of the table 3, the **X**, **E** and **P** values are the following:

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

Observe that, in all cases of **X**, the iterations number of the loop of the program is **E**=128. For this reason, **X = P × E**. The number of access to the memory and the ldb instructions are also 128.

- Write the table 3 with the obtained data in the previous point and draw a graphic (see Figure 5). To write the table 3 follow the next procedure:

a. Execute AMP

b. New project
CHANGE CACHE

c. Settings > System Settings > System information file + browse > nios_system.sopcinfo

d. Settings > System Settings > Quartus II Programming file + browse > De0_Nano_Basic_Computer.sof

e. Actions >Download system
CHANGE DATA

f. Modify the file of program: lab2_part1_2_3_main.s to establish new values of **X** and **P**

g. Compile

h. Load

i. Execute the program wait for the time to appear in the AMP terminal and note it in the table 3.

CONTINUE CHANGE DATA

CONTINUE CHANGE CACHE

Table 3. Table that is used in the Activity 4 to collect the times measures of execution

	E: Bytes number of the vector V that are really accessed					
P: jump pattern	128	256	512	1024	2048	4096
P= 1: from 1 to 1						
P= 2: from 2 to 2						
P= 4: from 4 to 4						
P= 8: from 8 to 8						

- In base on the data obtained, deduce (discover) the total capacity of storage of the cache memory of data and size block of cache, thinking and justifying all the answers. Help: Remember that in theory classes we explained and calculated the concept access time average to memory: AMAT = timeSuccess + failure frequency x penalty

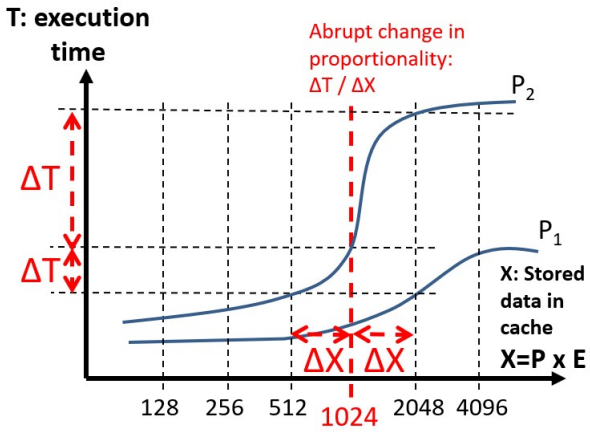

Figure 5. Graphic where is show the relationship between the execution time of the program and the number of elements accessed (E) of the vector V

These are the web pages used: [2, 4, 1, 3]

References

[1] 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.

[2] 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.

[3] Altera. Nios ii processor reference handbook, altera corporation. https://ftp.intel.com/Public/Pub/fpgaup/pub/Teaching_Materials/current/Tutorials/Nios2_introduction.pdf, 2014.

[4] 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.

Annex 1

```
/*****************************************************************************
* lab2_part1_2_3_main.s
*
* Main program of the Practice 3 of EC
*
* Initialize the Timer system of DE2
* Initialize and activate the interrumptions system of the NIOS II processor
* Execute one loop of Fibonacci and show the number of intervals of 33 ms in HEX of
* the DE2 board
*
* Subroutines: PRINT_JTAG (lab2_part1_2_3_JTAG.s), FIBONACCI (lab2_part1_2_3_fibo.s),
* ****************************************************************************/

.equ ITERATIONS, 500000

.text /*start the executable code*/

.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 miliseconds */

        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 word top half from 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 */
        wrctl ienable, r7 /* Timer) y nivel 1 (pushbuttons) */
        movi r7, 1
        wrctl status, r7 /* NIOS II interrupts are activated*/

        movia r14, ITERATIONS /* initializes the Fibonacci iterarion 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 in disabled in the NIOS II*/
        call PRINT_JTAG /* the number of 33 ms intervals is displayed in the AMP
         terminal*/
        IDLE: br IDLE /* the main program finish */
.data
.global COUNTER
COUNTER:
        .skip 4 /*memory position that saves the Timer interval counter*/ .end
```

Annex 2

```
/*****************************************************************************
* lab2_part1_2_3_fibo.s
*
* Subroutine: Executes the Fibonacci Series computation for 8 numbers
*
* Call 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 N */
        ldw r5, (r4) /* r5 is the counter initialized with N */
        addi r6, r4, 4 /* r6 points to the first Fibonacci numbers */
        ldw r7, (r6) /* r7 contains the first Fibonacci number */
        addi r6, r4, 8 /* r6 points to the first Fibonacci numbers */
        ldw r8, (r6) /* r7 contains the second Fibonacci number */
        addi r6, r4, 0x0C /* r6 points to the first Fibonacci number result */
```

```
        stw r7, (r6) /* Save the first Fibonacci number */
        addi r6, r4, 0x10 /* r6 points to the second Fibonacci number result */
        stw r8, (r6) /* Save the second Fibonacci number */
        subi r5, r5, 2 /* Decrease the counter in 2 numbers already saved */
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 /* Fibonacci Numbers */
```

```
NUMBERS:
        .word 0, 1 /* First 2 numbers */
RESULT:
        .skip 32 /* Space for 8 numbers of 4 bytes */
.end
```

Annex 3

```
/*******************************************************************************
* subroutine: lab2_part1_2_3_interrupts.s

*

* The program AMP (Altera Monitor Program) locates the section ".reset"

* in the direction of the memory of the rest that is specified in the Nios II configuration

* which is determined with SOPC Builder.

* "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 program AMP (Altera Monitor Program) located automatically the section ".ex-
ceptions"

* in the direction of the memory of the rest that is specified in the Nios II configuration

* which is determined with SOPC Builder.

* "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 /* interruption is not external */
        subi ea, ea, 4 /* ea must be decreased in 1 instruction */
        /* for external interruptions, so that */
        /* the interrupted instruction will be executed after eret (Exception Return) */

SKIP_EA_DEC:
        stw ea, 4(sp) /* save registers to the Stack */
        stw ra, 8(sp) /* is 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 interrupts */
        call INTERVAL_TIMER_ISR
        br END_ISR
```

```
END_ISR:
        ldw et, 0(sp) /* restore previous registers values */
        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
*
* Call from: lab2_part1_2_3_interrupts.s
*********************************************************************************/
.extern COUNTER
.global INTERVAL_TIMER_ISR
INTERVAL_TIMER_ISR:
        subi sp, sp, 8 /* reserved space in the stack */
        stw r10, 0(sp)
        stw r11, 4(sp)
        movia r10, 0x10002000 /* base address of Timer */
        sthio r0, 0(r10) /* inicialize to 0 the interruption */
        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

```
/*****************************************************************
* file lab2_part1_2_3_JTAG.s
* AC - Practice 2 - Exercise 1
* Subroutines related to the display of a character in the terminal
* input parameters:
* r10 = ascii value of the character to be processed
* output parameters: none
*****************************************************************/
.extern COUNTER /* variable defined in the main program */
/*
Subroutine: PRINT_JTAG
Displays in AMP JTAG terminal the contents of the external memory position 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 /* write in JTAG terminal fixed text */
```

```
        movia r17, COUNTER /* base address of the Timer interval counter */
        ldw r4, 0(r17)
        call BCD /* r4= binary value, r2= BCD value */

        call WRITE_VALUE_JTAG /* writes to JTAG terminal BCD value */
        movia r3, TEXT_FIN
        call WRITE_TEXTO_JTAG /* write in JTAG terminal 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
write a string of characters by jtag terminal
parameters: 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 1 byte from character string address */
        beq r10, r0, WITH /* if reads a 0 it means that have reached the end of the chain
        and goes out of the loop
*/
        call WRITE_JTAG /* subroutine that shows the byte by JTAG-UART */
        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
writes a value to BCD by JTAG terminal
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 nibles BCD counter */
VALUE:
        andhi r10, r2, 0xf000 /* extracts the 4 more significant bytes BCD */
        srli r10, r10, 28 /* r10 result is shifted to the right 28 bits */
        addi r10, r10, 0x30 /* addition 0x30: BCD -> ASCII */
        call ESCRIBIR_JTAG /* shows ASCII */
        subi r4, r4, 1 /* counter of nible – */
        slli r2, r2, 4 /* next nible BCD */
        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 stored in the stack */
        stw r3, 4(sp)
        stw r22, 8(sp)
        stw ra, 12(sp)

        movia r22, 0x10001000 /* base address of JTAG */
```

```
AGAIN: /* survey: check if there is space to write */
        ldwio r3, 4(r22) /* read register of JTAG-UART port */
        andhi r3, r3, 0xffff /* selects the 16 bits more significants */
        beq r3, r0, AGAIN /* ¿WSPACE=0? */

WRT: /* sends the character by writting in JTAG-UART */
        stwio r10, 0(r22)
END: ldw r3, 4(sp) /* retrieve the logs from the stack and return */
        ldw r22, 8(sp)
        ldw ra, 12(sp)
        addi sp, sp, 12

        ret

/*
* Data zone
*/ TEXT: .asciz "_n_nTime intervals that the program needs= "
TEXT_END:
.asciz "_n_nEnd of the program "
.end
```

Annex 6

```
/*****************************************************************************
* lab2_part1_2_3_BCD.s
*
* Subroutine: transform the binary code to BCD
*
* Call from: lab2_part1_2_3_JTAG.s
* Subroutine: DIV (lab2_part1_2_3_div.s)
*
* arguments: r4= binary value
* results: r2= BCD value
*
*****************************************************************************/
.text
.global BCD
BCD:
        subi sp, sp, 24 /* memory reserve in the Stack */
        stw r3, 0(sp)
        stw r4, 4(sp)
        stw r5, 8(sp)
        stw r6, 12(sp)
        stw r10, 16(sp)
        stw r31, 20(sp) /* by possible nested call */

        beq r4, r0, END /* if binary == 0 goto END */
```

```
        addi r5, r0, 10 /* r5 = 10 for divide BCD */

        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= rest */
        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 various 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 /* releases the reserved stack */
        ret
.end
```

Annex 7

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

* lab2_part1_2_3_div.s

* Entire division for NIOS II that is required when the processor does not have

* hardware of one divisor

*

* Reference:

* http://stackoverflow.com/questions/938038/assembly-mod-algorithm-on-processor-with-no-division-operator

*

* Call from: lab2_part1_2_3_JTAG.s

*

* arguments: r4= dividend, r5= divisor

* results: r2= rest, r3= quotient

*

*********************************************************************************/

.text

.global DIV

DIV:

        subi sp, sp, 16 /* reserve space in 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 /* rest = 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 = resto - 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 <= rest */
        sub r2, r2, r7 /* then rest = rest - 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 /* releases the reserved stack */
        ret
.end
```