

Computer Architecture - Lab Assignment 3

Performance evaluation of pipelined processors

Introduction

The main goal if this lab assignment consists in the performance evaluation of the pipelined processor [1] performance Nios II/f and compare with the multicycle processor Nios II/e [2]. Additionally, we analyze the effect of the performance of Nios II/f of the software technique of reorder the instructions. Finally, we propose that realize a theory exercise where we evaluate the change of the microarchitecture Nios II/f using the data obtained during the practice activity.

This practice is divided into four parts which are the following

Part 1. You will do an analyze of the instructions execute in a benchmark program to know the percentage of each one of the types of the instructions: ALU, MEMORY, JUMP.

Part 2. You will do an analysis of the performance of the multicycle processors Nios II/e and segmented Nios II/f to know in which circumstances of the execution of the program is limited by the Access to memory or by the operations ALU. Additionally, we analyze the deteriorated of the performance of these processors that introduce the instructions JUMP.

Part 3. We compare the effect of the performance of the segmented processor Nios II/f that produce the software technique of reorder the instructions.

Part 4. We propose you to evaluate the new design of the segmented processor.

The academic material for this assignment is found of the following folders:

- `benchNIO$II2021_Parte1`
- `benchNIO$II2021_Parte2`
- `benchNIO$II2021_Parte3`
- `N2fdCache512B-4bytes`

You should use the Intel DE0-Nano board that it is found in one of the laboratories of the ULPGC (see Figure 1). To do this, each student should connect this board in a computer of the laboratory and execute in a virtual machine “Altera” where you find this tool Altera Monitor Program (AMP) that allows interacting with the board.

Part 1. Analizing the mix of instructions in a benchmark and the CPI of Nios II/{e,f} processors

General description: you use a benchmark synthetic `benchNIO$II2021_Parte1` to analyze the mix of instructions of the instructions set Nios II of 32 bits. This program realizes the dot product of two vector

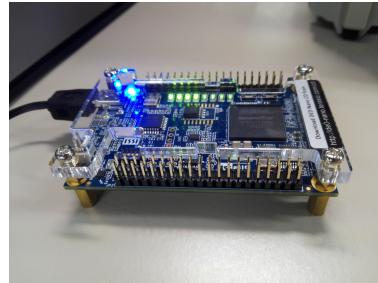


Figure 1: DE0-Nano board

many times, as indicated by the constant `ITER_BENCH` (see the subroutine `PRODUCTO_ESCALAR` in the file `producto_escalar.s`).

Objective 1: Classify the instructions, count the number of times that executes each one in the subroutine `PRODUCTO_ESCALAR` of the source code that its find in the file: `producto_scalar.s`. In this Figure 4 it shows the flowchart of the principal activities that its realize in the benchmark.

Objective 2: Now calculate the total number of instructions execute and the percentage of each one type of instruction (ALU, MEMORY, JUMP and others) writing on the Table 3.

Objective 3: Register the total number of clock cycles that execute the benchmark,for the both processors Nios II/e and Nios II/f, and calculate the CPI of the program of this processor. Important: Consider that the part of the benchmark that is not the compute kernel does not contribute a significant number of instructions to calculate the CPI.

Experimental methodology based on the multicycle processor Nios II/e:

1. Create a new Project in AMP using the configuration *DE0-Nano Basic Computer*.
2. Write the shift of the sections `.text` and `.data` inside the address space in `0x400`.
3. Compile and charge the benchmark program (file: `benchNIO$II_Parte1`) en in the DE0-Nano board.
4. Execute step by step to count the type of instructions that execute in the kernel of compute that you identify in the Figure 1 using breakpoints in the zone corresponding with the executable program.

Experimental methodology based on the pipelined processor Nios II/f:

1. Create a new Project in AMP using the configuration Custom System with the file `sopcinfo: nios_system.sopcinfo` and the `sof` configuration file: `DE0_Nano_Basic_Computer.sof` (both are in the folder: `N2fdCache512B-4bytes`).
2. Write the shift of the sections `.text` and `.data` inside the address space in `0x400`.
3. Compile and charge the benchmark program(file: `benchNIO$II_Parte1`) en in the board DE0-Nano.
4. Execute the program until receiving the message: `end of the program`.

Files of this activity:

- Main program of the benchmark: `benchNIO$II2021_Parte1.s`
- Subroutines: `DIV.s`, `JTAG2021.s`, `productoEscalar.s`, `BCD.s`, `nios_macros.s` (folder: `benchNIO$II_Parte1`).
- Configuration files for Nios II/f: `nios_system.sopcinfo`, `DE0_Nano_Basic_Computer.sof` (folder: `N2fdCache512B-4by`

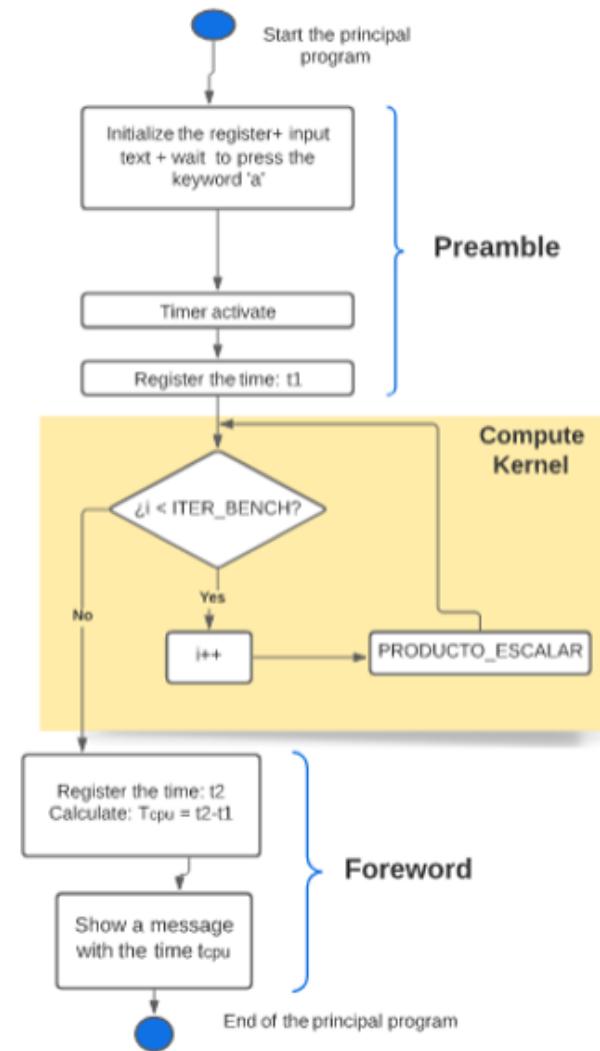


Figure 2: Flowchart of the benchmark program whose main program is found in the file `benchNIOSII2021_Parte1.s`.

ALU Instruction	Number of executions	Memory Instructions	Number of executions	Jump Instructions	Number of executions	Others Instructions	Number of executions
addi		ldw		beq		nop	
...		
...		
Total Instructions ALU		Total Instructions MEMORY		Total Instructions JUMPS		Total Instructions OTHERS	
N (total instructions executed)							
% ALU		% MEMORY		% JUMPS		% OTHERS	
CYCLES NIOS II/e				CYCLES NIOS II/f			
CPI TOTAL OF THE PROGRAM NIOS II/e				CPI TOTAL OF THE PROGRAM NIOS II/f			

Figure 3: Percentage of instructions executed by the processors Nios II/{e,f} using the subroutine `PRODUCTO_ESCALAR` found in the file named `producto_escalar.s` (folder: `benchNIOSII_Parte1`)

Question por Part 1

1. ¿In what type of program you classify the program `bechNIOSII2021_Parte1` (arithmetic, limited by memory accesses (memory-bound) o limited by jump instructions)? Justify and argue the answer.
2. Assuming that the instructions take on average a total of 6 cycles to execute in the Nios II/e processor [1], and that in the Nios II/f processor take: 1 cycle (ALU instructions), 1 cycle (memory instructions), 2 cycles (inconditional jumps) [1], obtain the theoretical CPI of the program in both processors.
3. What it is the differences that you find in the values obtained for the CPI about? ¿What circumstances do you think these differences are caused? Justify and argue the answer.

The benchmark is divided into three sections (see Figure 4): prologue, computing kernel and epilogue.

In the prologue, we wait to introduce ‘a’ into the keyboard. After that, the input/output device named Timer is configured. Then, the time is registered and identified as t_1 .

In the computing kernel, the dot product of 2 vectors with 6 components per vector is repeated many times.

In the epilogue, time is registered again and identified as t_2 . Then, the time between t_2 and t_1 is calculated and shown on the AMP terminal.

Part 2. Analizing the limitations in the relation “ALU-operations/second” for Nios II/{e,f} processors

General description: In this part you will evaluate the limit of two soft processors, Nios II/{e,f}, in the number of arithmetic operations that can be done for every second. Mainly, the bounds in the “ALU-operations/second” measurement that is enforced by the ALU functional unit, the memory hierarchy and the jump instructions will be evaluated. For this purpose, you will use a second benchmark called `bench-NIOSII2021_Parte2`.

Objective 1. Obtain the curve “ALU-operations/sec” vs. “ALUoperations/MEMORYbyte” (see Figure 4). This curve represents the performance level of the Nios II processor measured in number of arithmetic operations per time unit. In this curve, we can distinguish three zones (see Figure 4):

1. Zone 1. The peformance of the processor measured in “ALU-operations/sec” is limited by the memory accesses. So, this zone is also named *memory-bound*. This zone is characterized by the increasing in

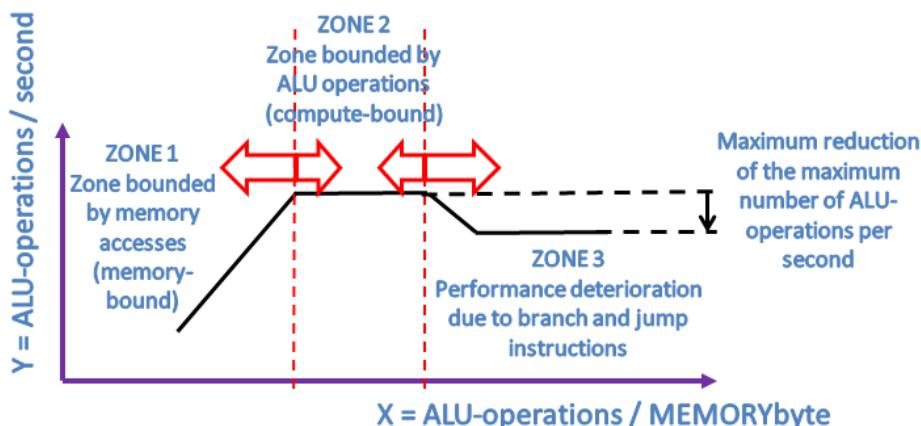


Figure 4: “roofline” curve to evaluate the bounds in the number of ALU operations per time unit.

the number of arithmetic operations per second (ALU-operations) as the number of memory bytes accessed also increases.

2. Zone 2. The performance of the processor measured in “ALU-operations/sec” is limited by the number of arithmetic operations that can be done by the processor every second. So, this zone is also named *compute-bound*. This zone is characterized by a fixed number of ALU-operations per second. This constant is the maximum performance level for the processor.
3. Zone 3. The performance of the processor measured in “ALU-operations/sec” is deteriorated by the jump and branch instructions respect to the maximum value shown in Zone 2. The third zone is characterized by a significative reduction in the maximum number of ALU-operations/sec.

Experimental methodology for Part 2 that is common to processors Nios II/{e,f}

The source code found in the file `roofline.s` is modified as explained below. This code generates the benchmark program called `benchNIOSII2021_Parte2` (see Figure 5).

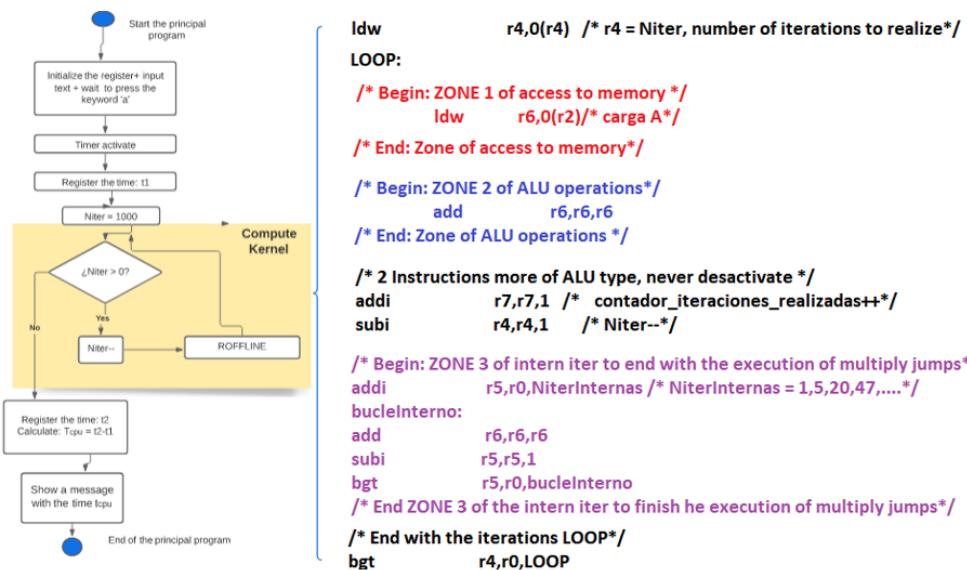


Figure 5: Flowchart of the benchmark program `benchNIOSII2021_Parte2` and internal loop of the subroutine `ROOFLINE` that is found in `roofline.s` file.

Next, the program is executed using both Nios II/{e,f} processors as many times as the inputs shown in Table 6:

- Each input of Table 6 is filled with data provided by the execution of the benchmark program `benchNIOSII2021_Parte2`. Before each run, each one of the zones marked with comments in the file `roofline.s` should be modified.
 - Modification of Zone 1: uncomment as many `ldw` instructions as the `ldw` column shows.
 - Modification of Zone 2: uncomment as many `add` instructions as the `ALU` column shows.
 - Modification of Zone 1: uncomment as many loop iterations as the `br` column shows.
- Fill in the Table 6 for the Nios II{e,f} processors.
- Represent the pairs of values (*ALU – operations/sec*, *ALU – operations/MEMORYbyte*) using two *X – Y* figures as shown in Figure 4, one for each processor, using the values obtained in Table 6.

Experimental methodology for the Nios II/e multicycle processor:

1. Create a new project in AMP tool using the configuration called *D0-Nano Basic Computer*.

Figure 6: Table to generate the “roofline” curve to measure the bound in the performance of the Nios II/{e,f} processors (see Figure 4). X coordinate represents “ALU-operations/MEMbyte”. Y coordinate represents “ALU-operations/sec”. Legends:

- *ID*: identification code for the experiment that provides a point of the curve.
- *ldw*: number of *ldw* instructions executed in LOOP (see `roofline.s`).
- *ALU*: number of ALU instructions executed in LOOP (see `roofline.s`).
- *br*: number of jump and branch instructions executed in LOOP (see `roofline.s`).
- *ALU-operations/MEMbyte*: ratio between the number of ALU instructions executed in each iteration of LOOP and the number of bytes accessed to main memory. This value is obtained combining the values of the columns: $ALU/4 * ldw$.
- *N_{iter}*: number of iterations of LOOP (see `roofline.s`).
- *N*: total number of executed instructions in the ROOFLINE subroutine (see `roofline.s` file). his value is obtained combining the values of the columns: $N_{iter} \times (ldw + ALU + br)$.
- cycles: value provided by the AMP terminal that represents the execution time of the `benchNIOSSII2021_Parte2` program.
- *t_{cpu}*: execution time of the program `benchNIOSSII2021_Parte2`, which is obtained using: $cycles/f$, where $f = 50$ MHz = 50×10^6 Hz. *t_{cpu}* it is measured in seconds.
- *ALU – operations/sec*: number of operations done in the ALU functional unit per second. This value is obtained combining the values of the columns: $N_{iter} \times ALU/t_{cpu}$.
- *CPI*: Average number of clock cycles needed to execute each instruction of the program. This value is obtained combining the values of the columns: $cycles/N$.

Id of the curve point	kernel: <code>roofline.s</code>			Iterations (N _{iter})	N (instructions execute	Nios II/{e,f}, f = 50 MHZ				
	number of iterations by iterations					Coord X operations ALU/4byteMEM (ALU/4*ldw)	cycles	t _{cpu} seg=cycles/f		
	ldw	ALU	br					Coord y operations ALU/seg (N _{iter} * ALU/t _{cpu})		
1	4	3	1	1000						
2	3	3	1	1000						
3	2	3	1	1000						
4	1	3	1	1000						
5	1	4	1	1000						
6	1	5	1	1000						
7	1	7	1	1000						
8	1	11	1	1000						
9	1	15	1	1000						
10	1	19	1	1000						
11	1	23	1	1000						
12	1	27	1	1000						
13	1	31	1	1000						
14	1	35	1	1000						
15	1	39	1	1000						
16	1	47	1	1000						
17	1	50	1	1000						
18	1	58	1	1000						
19	1	88	1	1000						
20	1	142	1	1000						
21	1	848	1	1000						
22	1	1048	1	1000						

2. Set the origin of `.text` and `.data` sections at the memory address 0x400.
3. Compile and download the benchmark program (`benchNIOsII_Parte2.elf` file) into the DE0-Nano board.
4. Execute the program until receiving the message “end of the program”.

Experimental methodology for the Nios II/f pipelined processor:

1. Create a new project in AMP tool using the configuration called *Custom system* and the files: `nios_system.sopcinfo`, `DE0_Nano_Basic_Computer.sof`.
2. Set the origin of `.text` and `.data` sections at the memory address 0x400.
3. Compile and download the benchmark program (`benchNIOsII_Parte2.elf` file) into the DE0-Nano board.
4. Execute the program until receiving the message “end of the program”.

Files involved in this part of the lab assignment:

- Benchmark program: `benchNIOsII2021_Parte2.s`, `DIV.s`, `JTAG2021.s`, `roofline.s`, `BCD.s`, `nios_macros.s` (folder: `benchNIOsII_Parte2`).
- FPGA configuration for Nios II/f processor: `nios_system.sopcinfo`, `DE0_Nano_Basic_Computer.sof` (folder: `N2fdCache512B-4bytes`).

Questions for Part 2

4. From what number of ALU-operations per MEMORYbyte accessed in memory the processors Nios II/{e, f} are bounded by ALU operations?
5. Which is the maximum number of ALU-operations per second that can execute the Nios II/e processor? And the Nios II/f processor?
6. Which is the maximum percentage of deterioration in the performance of Nios II/{e, f} processors when the jump and branch instructions are executed?
7. Are the benchmark program used in Part 1 (`benchNIOsII2021_Parte1/producto_escalar`) bounded by memory or by ALU-operations?

Part 3. Influence of instruction reordering on performance of pipelined Nios II/f processors

General description: A new synthetic benchmark program called `benchNIOsII2021_Parte3` will be used to evaluate the effect of true RAW data dependencies between load instructions and ALU instructions. This benchmark program is similar to the one used in Parts 1 and 2 of this lab assignment except that the computing kernel is modified, which is now in a file called `bypassing.s`. In this section, it is proposed to apply the instruction reordering technique to reduce the execution time of the benchmark program.

Objective 1: Edit the `bypassing.s` file and place an `add` instruction that is data-dependent on the `ldw` instruction (see *Version 1* in Figure 7). Then, measure and record the execution time of the benchmark program that is shown on the AMP terminal when the Nios II/f processor is used.

Objective 2: Edit the `bypassing.s` file and place an `add` instruction that is NOT data-dependent on the `ldw` instruction (see *Version 2* in Figure 7). Then, measure and record the execution time of the benchmark program that is shown on the AMP terminal when the Nios II/f processor is used.

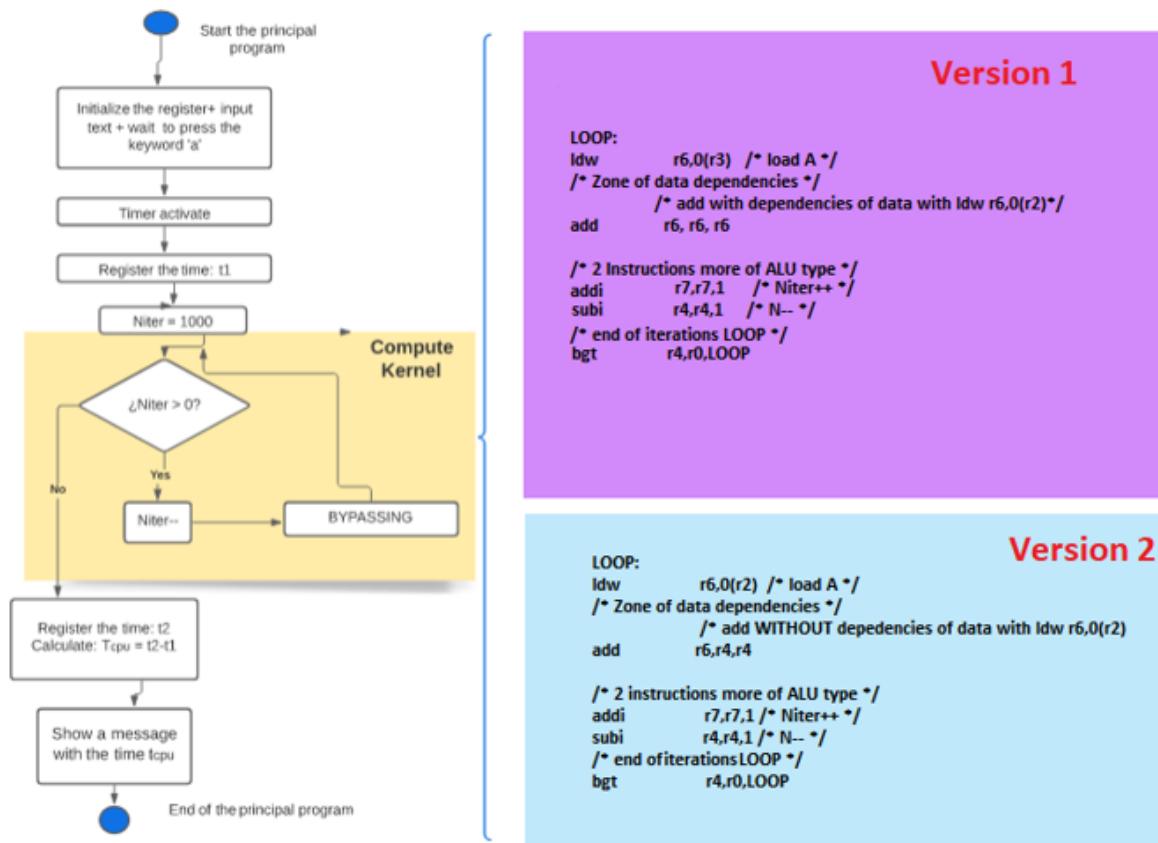


Figure 7: Flowchart of the benchmark program *benchNIOSII2021_Parte3* and internal loop of the subroutine *BYPASSING* that is found in *bypassing.s* file.

Questions for Part 3

8. Quantify the increase in the execution time of the program (t_{CPU}) in percentage (%) when there are data dependencies between `ldw` and `add` instructions respect to the reference case in which a data dependency between `ldw` and `add` instructions exists.
9. Calculate the CPI of the two versions of the benchmark program *benchNIOSII2021_Parte3*: Version 1 and 2. To do this, count only the executed instructions in the `LOOP` loop, noting that there is a constant named $N_{iter} = 1000$ in the source code of *bypassing.s*.
10. From the two CPI values from the previous question, calculate the CPI value of penalty due only to the `ldw` → `add` data-dependency in Version 1 (with true RAW dependencies) and their percentage compared to the CPI of Version 2 (no RAW dependencies). Note that: $CPI = CPI_{base} + CPI_{penalization}$.
11. Propose a new version (*Version 3*) of *Version 1* of the benchmark using instruction reordering. Note that you do not have to change the number of instructions that are executed in the `LOOP` loop.
12. Implement the new *Version 3* of the benchmark program and measure the execution time using AMP and the Nios II/f processor.
13. Calculate the speed-up of the new *Version 3* with respect to *Version 1*.
14. Calculate the speed-up of the new *Version 3* with respect to *Version 2*.
15. Based on the previous results, what conclusion regarding the improvements provided by the Nios II/f pipelined processor you can get after applying instruction reordering?

Part 4. Designing a new pipelined processor

Imagine that your boss at the department of computer architecture at Intel ask you to evaluate a possible modification of the design of the Nios II/f 6-stage pipelined processor. The modification proposes that the stage “*Execution/Calculate of address (E)*” and the stage of “*Access a memory(M)*” are unified in one pipe [2]. In this unified pipe, the ALU and the memory will work in parallel. The instructions for accessing to the data using the memory, leave the ALU inactive. In addition, the ALU instructions leave the memory inactive. This change will improve the energy efficiency.

In the Nios II instruction set architecture, the effective address of load and store instructions is calculated by adding the content of a register (**rs1**) and an immediate value (**imm**). The problem with the new design is that now no calculation is done with direction in the load or store instructions, since these instructions cannot use the ALU module.

The defenders of the new processor design try to change the architecture of the instructions set to allow only one addressing mode, called *direct addressing*. In this addressing mode, only a source register is used, which stores the value of the memory address of memory that is accessed. You cannot specify shifting using immediate values.

In Nios II, the only way to implement the direct addressing consists in providing the memory address using a register and assigning 0 to the immediate, $imm = 0$. With the proposed design, any load or store instruction that use the register addressing with imm value different from 0, need 2 instructions. In the first instruction, the values of the register and the immediate should add using an **addi** instruction. Then, this resulting address can be used in the following instruction to store or load. The load and store instructions that actually use $imm = 0$ do not require additional instructions in the new design.

Questions for Part 4

16. Your work consists in deciding the percentage of additional instructions that would have to execute under the new design of 5 pipes. This task will require to analyze the different types of load and store instructions executed in the subroutine **PRODUCTO_ESCALAR** of the benchmark **benchNIOSII2021_Parte1** that was analyzed in the Part 1 of this lab assignment.
17. Evaluate the new design depending on the increase in percentage of the number of instructions that will have to execute.
18. What design do you recommend to your Intel boss? Justify the answer quantitatively.

References

- [1] Altera. Nios II Core Implementation Details. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/nios2/n2cpu_nii51015.pdf, 2015.
- [2] Intel. Nios II Processor Reference Guide. <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/n2cpu-nii5v1gen2.pdf>, 2020.