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 Nios II/f pipelined processor [1] and compare with the Nios II/e multicycle processor [2]. Additionally, we analyze the effect on the performance of Nios II/f caused by the software technique called *instruction reordering*. Finally, we propose to realize a theoretical exercise where the modification of the Nios II/f microarchitecture is evaluated using the data obtained during this hands-on activity.

This lab is divided into four parts which are summarized in the next paragraphs.

Part 1. You will analyze of the executed instructions of a benchmark program to know the percentage of each main instruction type: ALU, memory, branch and jump.

Part 2. You will analyze the performance of the Nios II/e multicycle and Nios II/f pipelined processors to know which circumstances limit the performance of the program. Performance may be limited by the memory accesses or by the native capacity of the processor to do ALU operations. Additionally, we will analyze the reduction of performance caused by jump and branch instructions.

Part 3. You will analyze the effect on the performance of the Nios II/f pipelined processor provided by the software technique called *instruction reordering*.

Part 4. We propose you to evaluate the new design of the Nios II/f pipelined processor.

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

- `benchNIOSII2021_Parte1`
- `benchNIOSII2021_Parte2`
- `benchNIOSII2021_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, you should connect this board to a computer situated in the laboratory. Then, execute a virtual machine called *Altera* where you can find the *Altera Monitor Program (AMP)* software tool that allows interacting with the board from the host computer.

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

General description: you will use a synthetic benchmark called *benchNIOSII2021_Parte1* to analyze the mix of instructions of the 32 bits Nios II instruction set. This program calculates the dot product of two

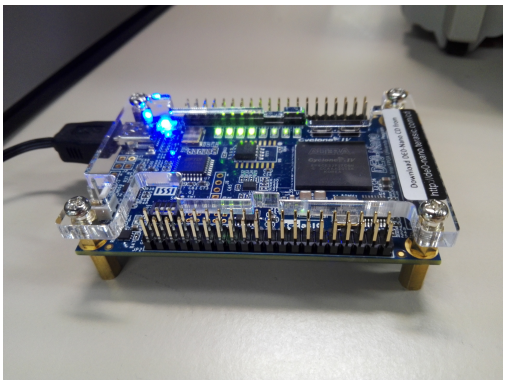

Figure 1: DE0-Nano board

vectors repetitively, as set by the constant `ITER_BENCH` (see the subroutine `PRODUCTO_ESCALAR` in the file `producto_escalar.s`).

Objective 1: Classify the instructions and count the number of times that each one executes in the subroutine `PRODUCTO_ESCALAR` of the source code found in the file: `producto_scalar.s`. Figure 2 shows the flowchart of the main activities done by this benchmark.

Objective 2: Now calculate the total number of executed instructions and register the percentage of each type of instruction (ALU, memory, jump and branchs) in Table 1.

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

Table 1: Percentage of instructions executed by the Nios II/{e,f} processors using the subroutine `PRODUCTO_ESCALAR` found in the file named `producto_escalar.s` (folder: `benchNIOSII_Parte1`).

ALU Instruction	**Number of executions**	**Memory Instruction**	**Number of executions**	**Jump Instruction**	**Number of executions**	**Other Instruction**	**Number of executions**
`addi`		`ldw`		`beq`		`nop`	
...		...		...		...	
...		...		...		...	
ALU Total Instructions		**Memory Total Instructions**		**Jump Total Instructions**		**Other Total Instructions**	
N (total number of executed instructions)							
Nios II/e cycles				**Nios II/f cycles**			
Total program CPI for Nios II/e				**Total program CPI for Nios II/f**			

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

1. Create a new Project in AMP using the configuration *DE0-Nano Basic Computer*.
2. Set `0x400` in the shift of the sections `.text` and `.data` inside the address space.
3. Compile and download the benchmark program (folder: `benchNIOSII_Parte1`) into the FPGA of the DE0-Nano board.
4. Execute step by step to count the type of instructions that execute in the computing kernel. Use breakpoints to monitor the execution of instructions. Register the values to fill in Table 1.

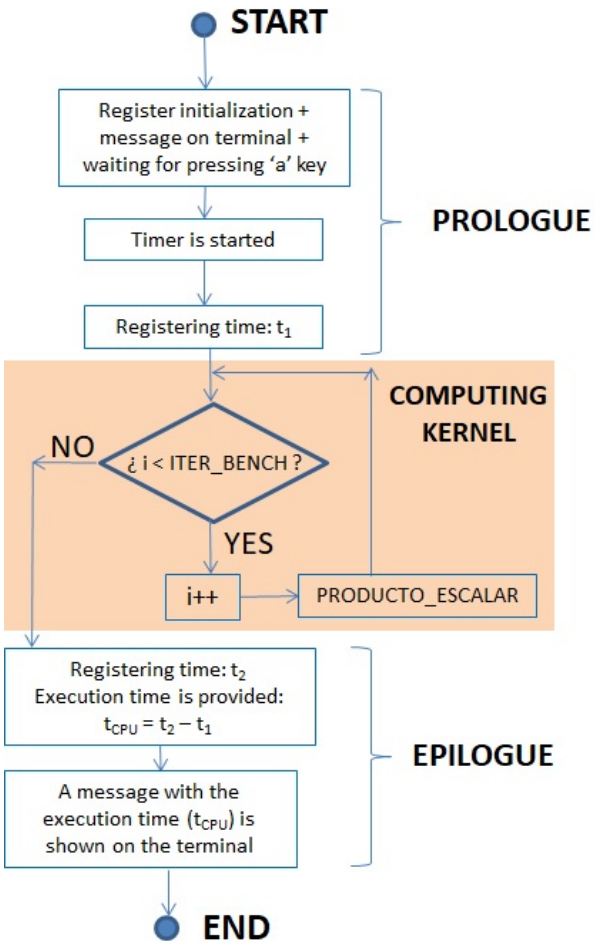

Figure 2: Flowchart of the benchmark program whose main program is found in the file `benchNIOSII2021_Parte1.s`.

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

1. Create a new project in AMP using the configuration called *Custom System* with the sopcinfo file: `nios_system.sopcinfo` and the sof configuration file: `DE0_Nano_Basic_Computer.sof` (both are in the folder: `N2fdCache512B-4bytes`).
2. Set `0x400` in the shift of the sections `.text` and `.data` inside the address space.
3. Compile and download the benchmark program (folder: `benchNIOSII_Parte1`) in the FPGA of the DE0-Nano board.
4. Execute the program until receiving the message: `end of the program`.

Files for this lab exercise:

- Main program of the benchmark: `benchNIOSII2021_Parte1.s`
- Subroutines: `DIV.s`, `JTAG2021.s`, `productoEscalar.s`, `BCD.s`, `nios_macros.s` (folder: `benchNIOSII_Parte1`).
- Configuration files for Nios II/f: `nios_system.sopcinfo`, `DE0_Nano_Basic_Computer.sof` (folder: `N2fdCache512B-4bytes`)

Question por Part 1

1. What type of program `bechNIOSII2021_Parte1` can be classified (compute-bound, memory-bound or performance deteriorated by jump and branch 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 needs: 1 clock cycle for ALU instructions, 1 clock cycle for memory instructions, 2 cycles for inconditional jumps [1], obtain the theoretical CPI of the program for both processors.
3. What is the differences that you find in the values obtained for the CPI? What events do you think these differences are caused? Justify and argue the answer.

The benchmark is divided into three sections (see Figure 2): prologue, computing kernel and epilogue.

In the prologue, the program wait for pressing the 'a' key. After that, the input/output device called *Timer* is configured. Then, the elapsed time is registered and named *t1*.

In the computing kernel, the dot product of two vectors with six elements per vector is repeated many times.

In the epilogue, time is registered again and named *t2*. Then, the time between *t2* and *t1* is calculated and shown on the AMP terminal.

Part 2. Analizing the bounds 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 performance measurement called *ALU-operations/second* that are imposed by the ALU functional unit, the memory hierarchy and the branch and jump instructions will be evaluated. For this purpose, you will use a second benchmark called *benchNIOSII2021_Parte2*.

Objective 1. Obtain the curve "ALU-operations/sec" vs. "ALUoperations/MEMORYbyte" (see Figure 3). 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 3):

1. Zone 1. The peformance of the processor measured in "ALU-operations/sec" is bounded by memory accesses. So, this zone is also named *memory-bound*. This zone is characterized by the increasing in the number of arithmetic operations per second (ALU-operations) as the number of memory bytes accessed also increases.
2. Zone 2. The peformance 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 peformance 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 Nios II/{e,f} processors:

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 4).

Next, the program is executed using both Nios II/{e,f} processors as many times as the value of the N_{iter} constant shown in Table 2.

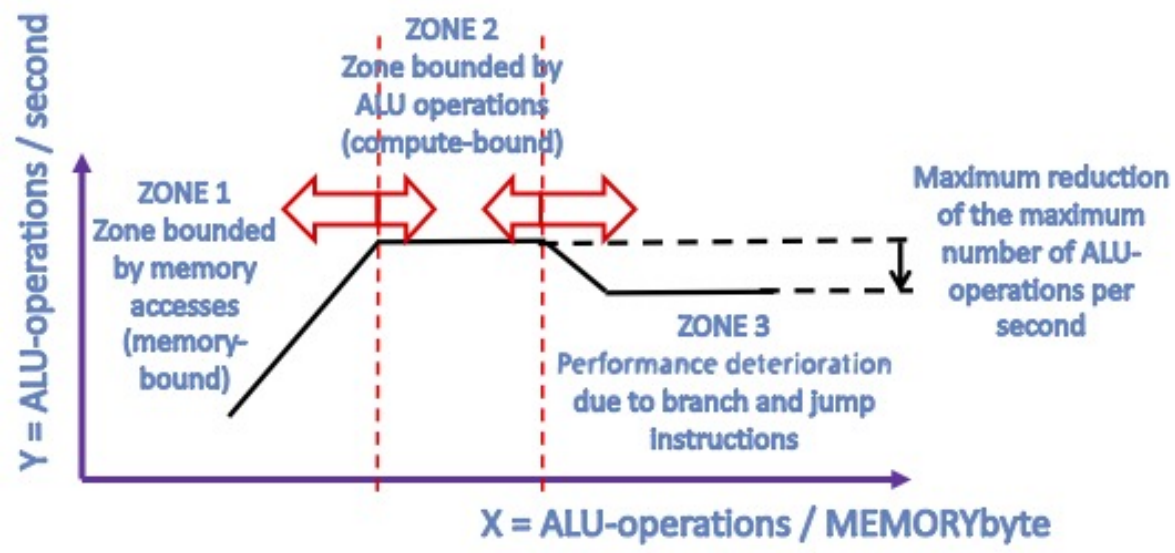

Figure 3: "roofline" curve to evaluate the bounds in the number of ALU operations per time unit.

- Each input of Table 2 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 `roofile.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 3: uncomment as many loop iterations as the *br* column shows.
- Fill in Table 2 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 3, one for each processor, using the values obtained in Table 2.

Experimental methodology for the Nios II/e multicycle processor:

1. Create a new project in AMP tool using the configuration called *DE0-Nano Basic Computer*.
2. Set `0x400` in the shift of the sections `.text` and `.data` inside the address space.
3. Compile and download the benchmark program (file: `benchNIOSII_Parte2.elf`) into the FPGA of 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 `0x400` in the shift of the sections `.text` and `.data` inside the address space.
3. Compile and download the benchmark program (file: `benchNIOSII_Parte2.elf`) into the DE0-Nano board.
4. Execute the program until receiving the message: `end of the program`.

Table 2: Table to generate the "roofline" curve to measure the bound in the performance of the Nios II/{e,f} processors (see Figure 3). X coordinate represents "ALU-operations/MEMORYbyte". 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`).
- X : $ALUoperations/MEMbyte$: 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: $X =$ `ALU` / ($4 \times$ `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: $N = N_{iter} \times$ (`ldw` + `ALU` + `br`).
- cycles: value provided by the AMP terminal that represents the execution time of the `benchNIOSII2021_Parte2` program.
- t_{cpu}: execution time of the program `benchNIOSII2021_Parte2`, which is obtained using: $cycles/f$, where $f = 50$ MHz $= 50 \times 10^6$ Hz. t_{cpu} it is measured in seconds.
- Y : $ALUoperations/sec$: number of operations done in the ALU functional unit per second. This value is obtained combining the values of the columns: $Y = N_{iter} \times$ `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: $CPI = cycles/N$.

ID	kernel: `roofline.s` number of instructions per iteration			**X**	N_{iter}	N	Nios II/{e,f}, $f = 50$ MHz			
	`ldw`	`ALU`	`br`				cycles	t_{CPU}	**Y**	CPI
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	6		1000					
19	1	88	21		1000					
20	1	142	48		1000					
21	1	848	401		1000					
22	1	1048	501		1000					

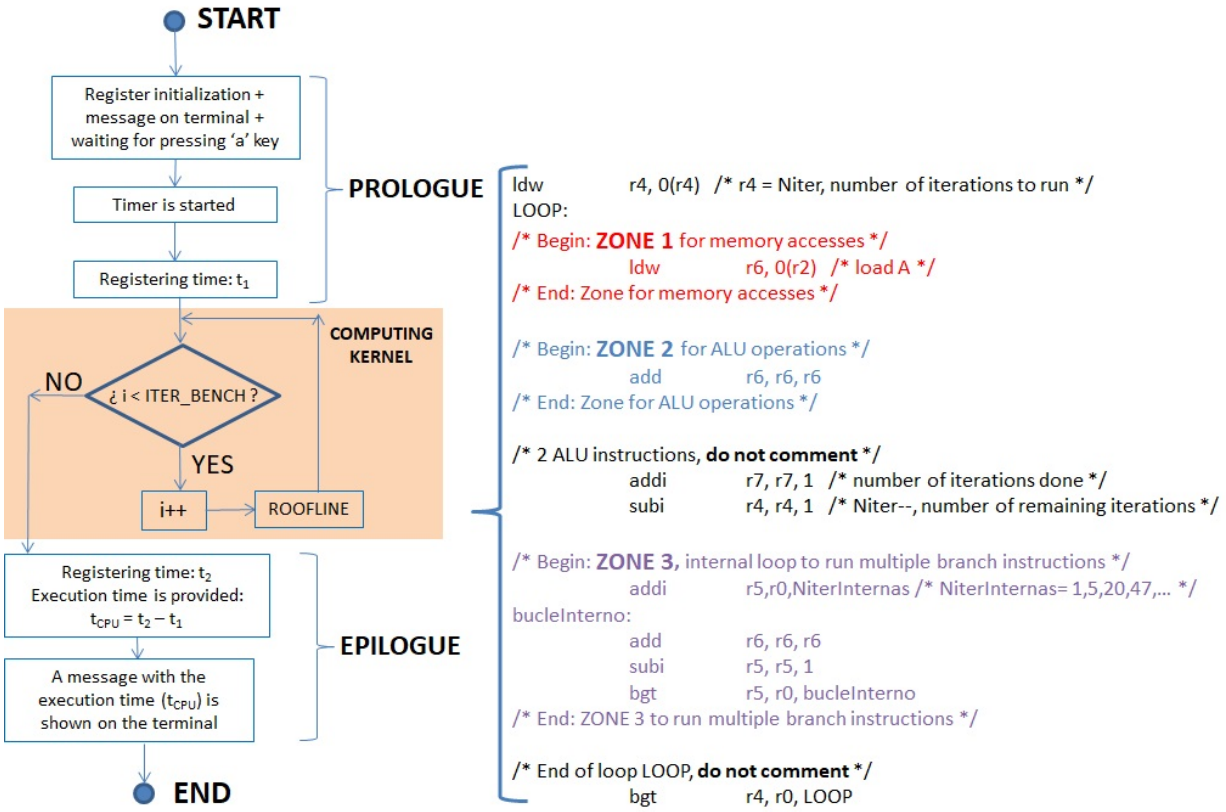

Figure 4: Flowchart of the benchmark program *benchNIOSII2021_Parte2* and internal loop of the subroutine `ROOFLINE` that is found in `roofline.s` file.

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 Nios II/f pipelined processor

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 5). Then, measure and register 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 5). Then, measure and register the execution time of the benchmark program that is shown on the AMP terminal when the Nios II/f processor is used.

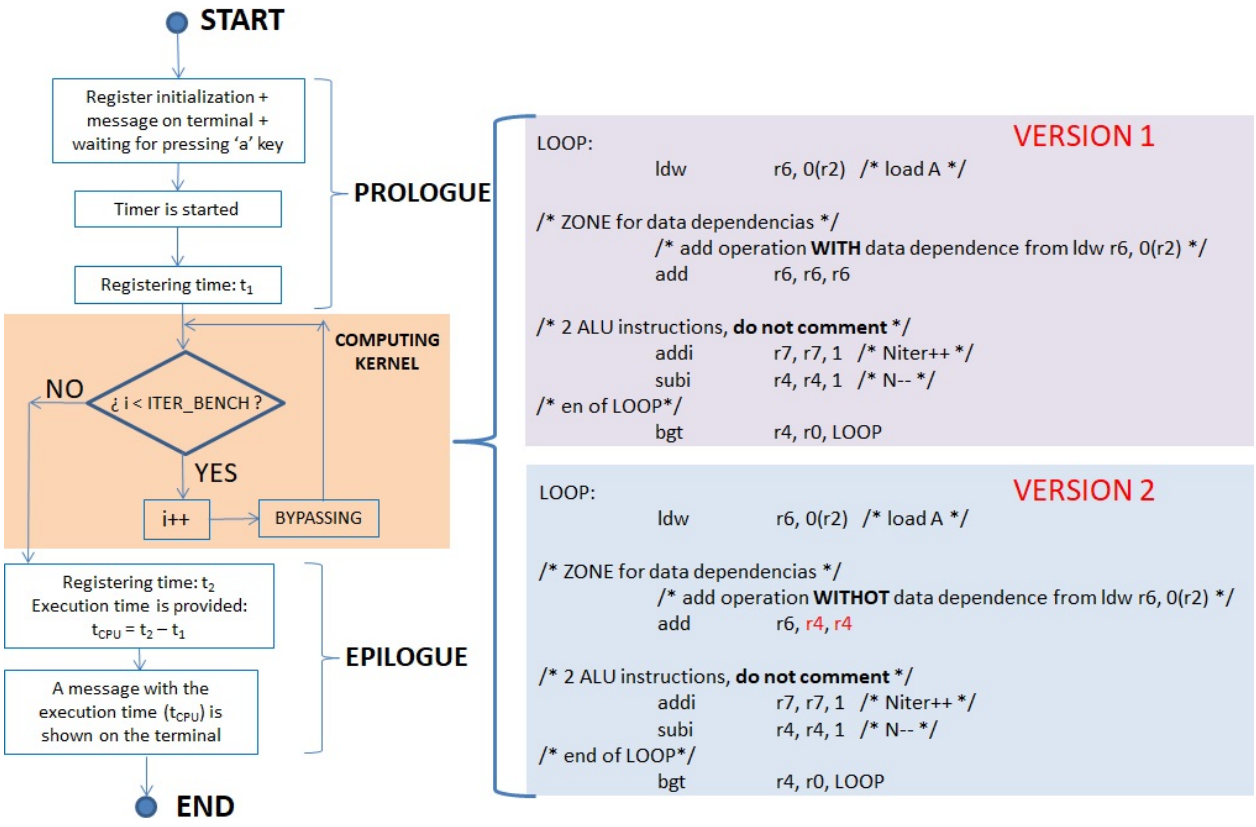

Figure 5: Flowchart of the benchmark program *benchNIOSII2021_Parte3* and internal loop of the subroutine `BYPASSING` that is found in `bypassing.s` file.

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 does not exist.

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`.

Questions for Part 3

10. From the two CPI values from the previous question, calculate the penalty CPI 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 employer at the department of computer architecture ask you to evaluate a potential modification to the design of the Nios II/f 6-stage pipelined processor. The modification proposes that the *"Execution/Address Calculation (E)"* stage and the *"Memory Access (M)"* stage are merged into a single pipeline stage [2]. In this combined stage, the ALU and memory will operate in parallel. The instructions for data access will use memory while leaving the ALU idle. In addition, the ALU instructions will leave the memory idle. These changes will improve the area and power efficiency.

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

Proponents of the new processor design advocate changing the instruction set architecture to allow only one addressing mode, called *register direct addressing*. In this addressing mode, only a source register is used, which contains the value of the memory address to be accessed. You cannot specify any offset using immediate values.

In Nios II, the only way to implement register 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 which uses the register addressing with imm value different from 0 will two instructions. In the first instruction, the values of the register and the immediate should be summed using an `addi` instruction. Then, this resulting address can be used in the following instruction to store or load. Load and store instructions which currently use $imm = 0$ will not require extra instructions on the new design.

Questions for Part 4

16. Your work consists in determining the percentage of additional instructions in the total number of instructions that would have to be executed under the new design of 5 stages. This task will require a more detailed analysis of the different types of load and store instructions executed by the `PRODUCTO_ESCALAR` subroutine of the benchmark *benchNIOSII2021_Parte1* that was used in the Part 1 of this lab assignment.
17. Evaluate the new design in terms of the percentage increase in the number of instructions that will have to be executed.
18. Which design would you advise to your employer? Justify your positition 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.