*Computer Architecture - Lab Assignment 3*

# Performance evaluation of pipelined processors

The principal objective if this lab assignment 3 consists in the evaluation of the pipelined processors Nios V/m and Nios V/g (Intel [2023a]). Additionally, we analyze the effect on the performance of the software technique based on reordering instructions. Finally, we propose to do an exercise where we theoretically evaluate a modification of the Nios V microarchitecture using the data obtained during the hands-on activity.

This assignment is divided into 4 parts which are described in the following paragraphs.

Part 1. Analysis of the usage of instructions types in a benchmark program and the CPI of Nios V/{m,g} processors.
You will do an analysis of the executed instructions in a benchmark program to found out the percentage of each type of the instruction: ALU, memory, jump/branch.

Part 2. You will do an analysis of the performance of the pipelined processors Nios V/m and Nios V/g to know in which circumstances of the execution of a benchmark program is limited by memory accesses or by ALU operations. Additionally, we will analyze the reduction in performance of these processors that is caused by the branch and jump instructions.

Part 3. We compare the effect on performance of the pipelined processors Nios V/m and Nios V/g that is caused by the software technique based on reordering instructions.

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

The material for this lab assignment includes the following three benchmarks:

- benchNIOSV2024_dotProduct
- benchNIOSV2024_roofline
- benchNIOSV2024_reordering

Two soft System-on-Chips will also be used in this lab assignment. Each SoC integrates a different Nios V soft processor. Both SoCs will be implemented in a FPGA-based board. Their names are:

- DE0-Nano Nios V/m Basic Computer
- DE0-Nano Nios V/g Basic Computer

You will use the *DE0-Nano* board (Terasic [2021]) that it is found in the Computer Architecture laboratory at the School of Computer Science of the ULPGC university (see Figure 1). You should connect this board to a desktop computer. Then, you should execute the Nios V Command Shell (Intel [2023b]) that allows to interact with the board (see Figure 2). Using this command shell, the board will be configured several times with various SoC microarchitectures. In Figure 3, the microarchitecture of two SoCs called *DE0-Nano Nios V/m Basic Computer* and *DE0-Nano Nios V/g Basic Computer* can be seen. Both configurations will activate different computers on the same DE0-Nano board.

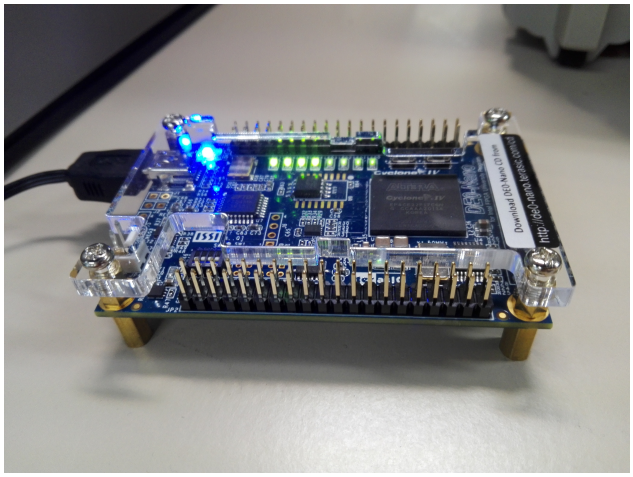

Figure 1: DE0-Nano board.

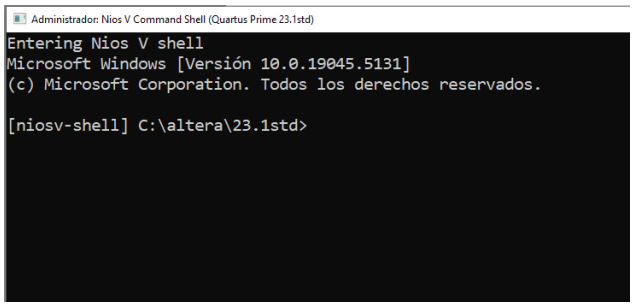


Figure 2: Nios V Command Shell (integrated into Intel/Altera Quartus Prime Standard 23.1 design suite).

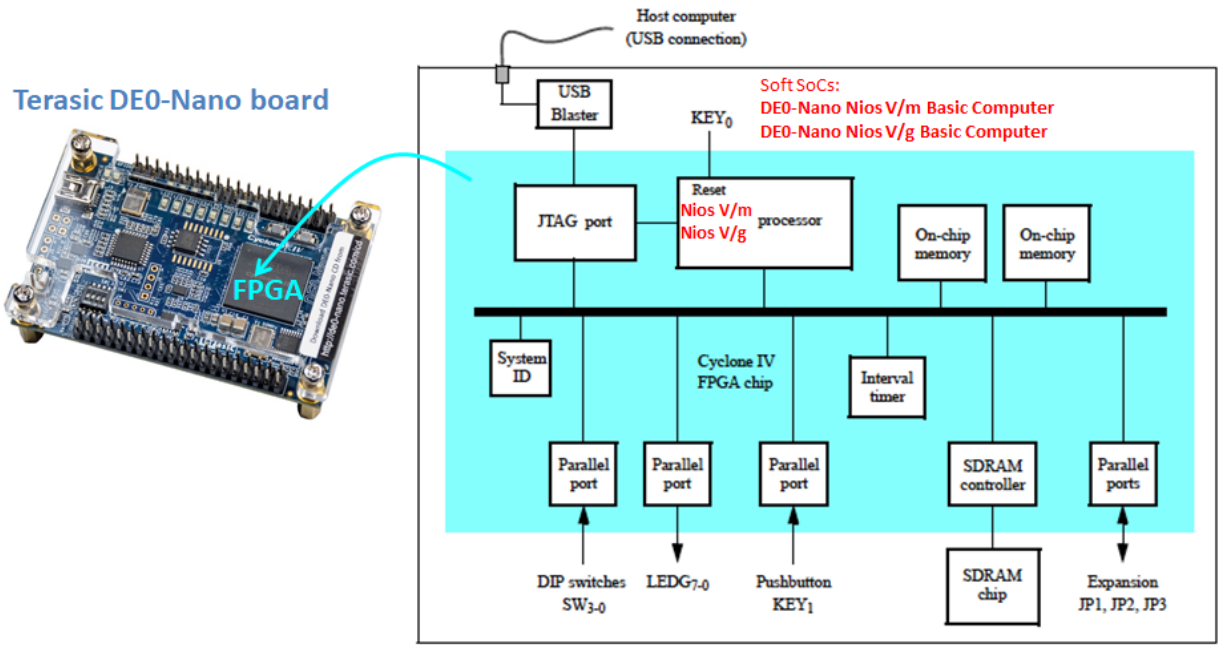


Figure 3: Microarchitectures of the soft System-on-Chips named *DE0-Nano Nios V/m Basic Computer* and *DE0-Nano Nios V/g Basic Computer*.

# Part I. Analysis of the usage of instructions types in a benchmark program and the CPI of Nios V/{m,g} processors

General description. You will use a synthetic benchmark named *benchNIOSV2024_dotProduct* to analyze the mix of RISC-V instructions for the 32-bit Nios V soft processor. This program implements the dot product of two vectors repetitively, as indicated by the constant `ITER_BENCH` (see the main program in the file called `benchNIOSV2024_dotProduct.s`).

Objective 1. Classify the instructions by counting the number of times that executes each one in the subroutine `PRODUCTO_ESCALAR` of the source code: `producto_scalar.s`. Figure 2. 4 shows the flowchart of the principal activities that the benchmark does.

Objective 2: Now calculate the total number of executed instructions and the percentage of each one type of instruction (ALU, MEMORY, JUMP and others) and fill in the Table 1.

Objective 3:Register the total number of clock cycles that execute the benchmark, for the both processors Nios V/m and Nios V/g, and calculate the CPI of the program for these processors. **Important:** Count only the executed instructions that are integrated into the compute kernel and assume that the rest of instruction do not influence significantly on the number of instructions involved in obtaining the CPI.

Hands-on method for the Nios V/m pipelined processor:

1. Create a new directory, for example: `part1` and make cd. Execute the following command in the Nios V Command Shell Window.

```
$ cd part1
$ sh
```

2. Copy the files: `benchNIOSV2024_dotProduct.s`, `productoEscalar.s`, `escribir_jtag.s`, `DIV.s`, `BCD.s`, `Makefile` in this directory.

3. Compile the benchmark program: `benchNIOSII_dotProduct`.

```
$ riscv32-unknown-elf-as.exe benchNIOSV2024_dotProduct.s -alsg -o
benchNIOSV2024_dotProduct.s.obj > benchNIOSV2024_dotProduct.s.s.log
$ riscv32-unknown-elf-as.exe productoEscalar.s -alsg -o productoEscalar.s.obj
> productoEscalar.s.log
$ riscv32-unknown-elf-as.exe escribir_jtag.s -alsg -o escribir_jtag.s.obj >
escribir_jtag.s.log $ riscv32-unknown-elf-as.exe DIV.s -alsg -o DIV.s.obj >
DIV.s.log
$ riscv32-unknown-elf-as.exe BCD.s -alsg -o BCD.s.obj > BCD.s.log
```

4. Link the benchmark program: `benchNIOSII_dotProduct`.

```
$ riscv32-unknown-elf-ld.exe -g -T linker_SDRAM.x -nostdlib -e _start -u _start
--defsym __alt_stack_pointer=0x08001F00 --defsym __alt_stack_base=0x08002000
--defsym __alt_heap_limit=0x8002000 --defsym __alt_heap_start=0x8002000
-o benchNIOSV2024_dotProduct.elf benchNIOSV2024_dotProduct.s.obj
productoEscalar.s.obj escribir_JTAG.s.obj BCD.s.obj DIV.s.obj
$ niosv-stack-report.exe -p riscv32-unknown-elf- benchNIOSV2024_dotProduct.elf
```

5. Configure the FPGA in the DE0-Nano board using the file `DE0_NanoBasic_Computer_22jul24.sof` into the board DE0-Nano.

```
$ quartus pgm.exe -c 1 -m JTAG -o "p;DE0_NanoBasic_Computer_22jul24.sof@1"
```

6. Download the file `benchNIOSV2024_dotProduct.elf` into the board DE0-Nano.

```
$ niosv-download.exe -g benchNIOSV2024_dotProduct.elf
```

7. 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. For this stage of the method, use the *OpenOCD* and *GDB* tools. For this step, open three Nios V Command Shell terminals ahd execute the following commands.

Terminal 1

```
$ cd part1
$ sh
$ openocd-cfg-gen ./niosv.cfg
$ openocd -f ./niosv.cfg --> cursor in trminal remains blinking
```

Terminal 2

```
$ cd part1
$ sh
$ riscv32-unknown-elf-gdb -ex "set arch riscv:rv32" -ex "target
extended-remote localhost:3333" -ex "file benchNIOSV2024_dotProduct.elf" -ex
"load"
Are you sure you want to change the file?  (y or n) y --> press "y"
(gdb) x 0x080000e4
(gdb) continue
(gdb) stepi
```

Terminal 3

```
$ cd part1
$ sh
$ juart-terminal.exe
```

Hands-on method for the Nios V/g pipelilned processor:

Follow the same method as previously used for Nios V/m but taking the `DE0_NanoBasic_Computer_23jul24.sof` configuration file for the DE0-Nano board.

**Question 1.**
What type of program is `benchNIOSV2024_dotProduct.elf`: arithmetic, memory, or branch/jump? Justify and argue your answer.

For Question 1, analyze the benchmark program for Part I by counting the number of instructions for each type shown in Figure 1. This benchmark is divided into three sections (see Figure 4).

1. **Preamble.** In this section, three actions are executed.
   (a) Control registers of the Nios V processors are initialized for activating the interrupt controller.
   (b) Program waits for pressing the typing key 'a' into the keyboard.
   (c) The Timer controller of the soft SoC for the DE0-Nano board is configured. Its interrupt signal is enabled and an initial count is saved in the *Counter Start Value (low, high)* timer registers.
2. **Kernel.** In this section, the dot product of two vectors with six components in each vector repeats 5,000 times. Previously, a snapshot of the time controller allows to get the initial value for time.
3. **Epilogue.** In this section, another snapshot is registered to obtain the final value for execution time. Then, final and initial values are sustracted to calculate the time interval. Finally, this hexadecimal value is translated into BCD format and shown on the terminal.

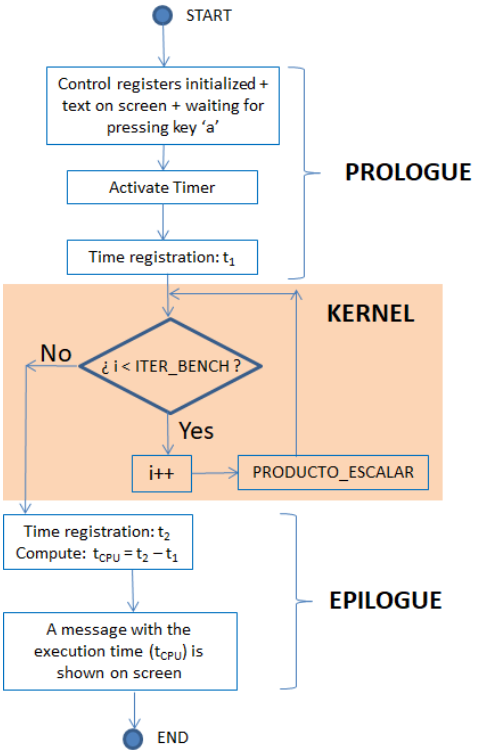


Figure 4: Flowchart of the benchmark program whose principal program is in the `benchNIOSV2024_dotProduct.s` file.

Table 1: Percentage of instructions executed by Nios V processors for the `PRODUCTO_ESCALAR` subroutine that is coded in the `producto_escalar.s` file

| **ALU instructions** | **Number of executions** | **MEMORY instructions** | **Number of executions** | **BRANCH & JUMP instructions** | **Number of executions** | **OTHER instructions** | **Number of executions** |
|---|---|---|---|---|---|---|---|
| addi | | lw | | beq | | nop | |
| ... | | ... | | ... | | ... | |
| ... | | ... | | ... | | ... | |
| **Total ALU instructions** | | **Total MEMORY instructions** | | **Total BRANCH & JUMP instructions** | | **Total OTHER instructions** | |
| **N (total number of executed instructions)** | | | | | | | |
| **% ALU** | | **% MEMORY** | | **% BRANCH & JUMPS** | | **% OTHER** | |
| Clock cycles **Nios V/m** | | | | Clock cycles **Nios V/g** | | | |
| **Total CPI of program Nios V/m** | | | | **Total CPI of program Nios V/g** | | | |

**Question 2.**
Taking the following average number of cycles per instruction for the Nios V/m and Nios V/g soft processors: 1 cycle (ALU ops), 1 cycle (memory ops), 2 cycles (jump and branch ops), calculate the theoretical CPI of the program when both processors execute the benchmark program. Justify and argue your answer.

**Question 3.**
What are the differences you found out in the values obtained for the CPIs of Nios V/m and Nios V/g processors? What are the causes for these differences? Justify and argue your answer.

# Part II. Analyzing the limitations of the 'operationsALU/second' ratio of a benchmark program on the Nios V/m and Nios V/g pipelined processors

General description: This part evaluates the limits of each Nios V/{m,g} soft processor in terms of the number of ALU operations it can perform in each unit of time. In particular, the limits measured in 'ALU operations/second' caused by the ALU functional unit of the processors, the memory hierarchy of the SoC embedded computer and the need to execute branch and jump instructions are analyzed.

In this section of the lab assignment a second benchmark named `benchNIOSV2024_roofline` is employed and a main objective is proposed.

Objective. Obtain the curve "ALU operations/second" versus "ALU operations/memory-byte" (see Figure 5). This curve is named *roofline curve* and represents the performance level of the Nios V processor measured in number of ALU operations performed per unit time.

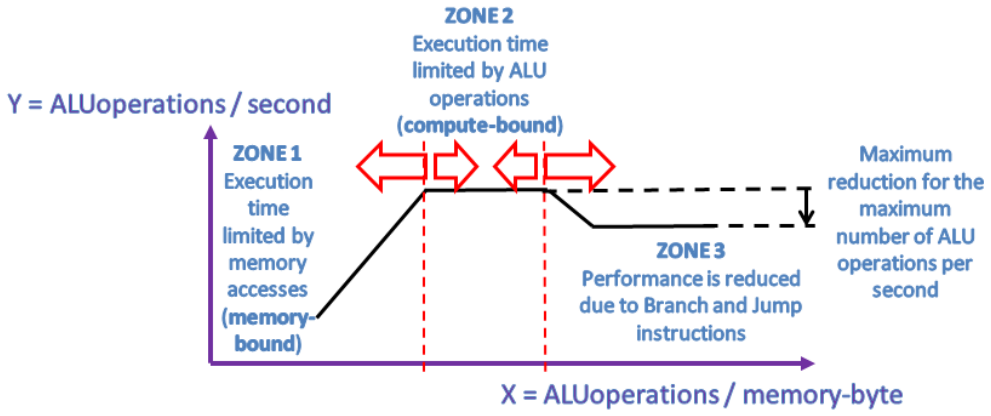


Figure 5: Roofline curve.

Three zones can be noted in the roofline curve.

1. **Zone 1**. The performance of the processor measured in "ALUoperations/sec" is limited by the memory accesses. This zone is also called *memory-bound*. It is characterized by the increase in the number of arithmetic operations (ALUoperations) per second when the number of arithmetic operations to the accessed memory bytes ratio is also increased.

2. **Zone 2**. The performance of the processor measured in "ALUoperations/sec" is limited by the number of arithmetic operations that the processor executes per second. This zone is also called *compute-bound*. It is characterized by a constant the number of ALU operations per second as the ALU operations to the accessed memory bytes ratio is increased.

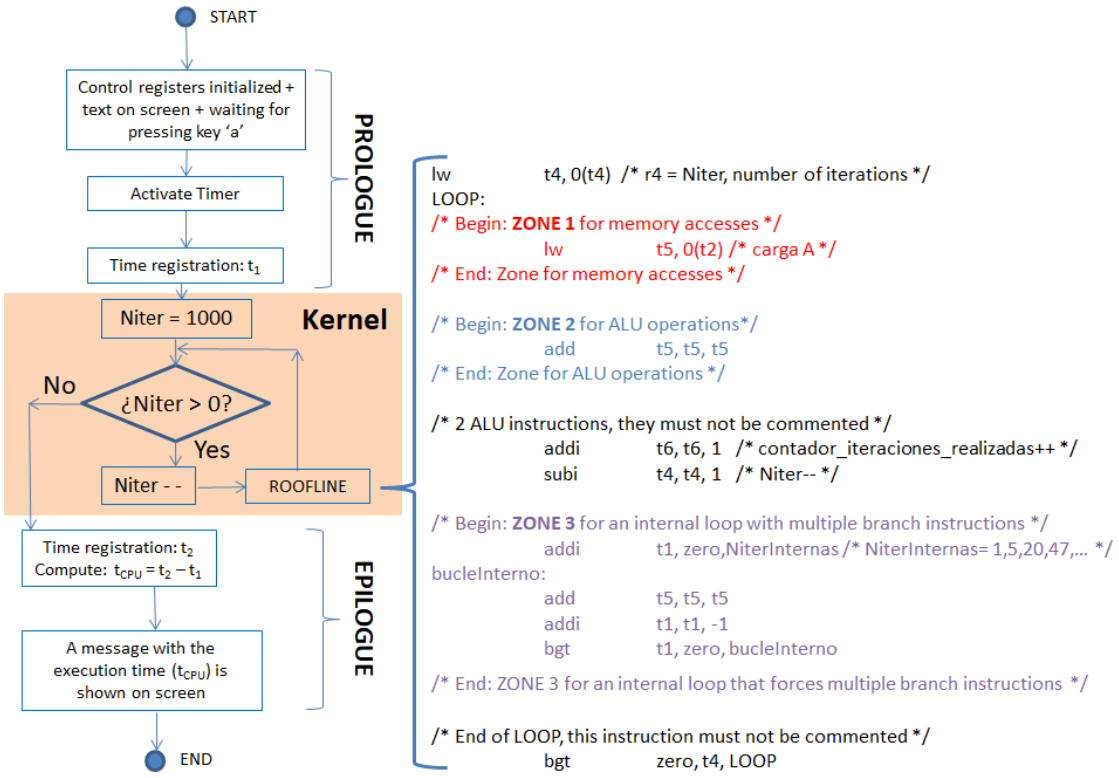


Figure 6: Flowchart for the benchmark program `benchNIOSV2024_roofline`. On the right, the main sections of the code for the kernel subroutine `ROOFLINE` are shown. This code is in `roofline.s` file.

3. **Zone 3**. The performance of the processor in ALU operations per second decay due to the Branch & Jump instructions respect to the maximum value observed in Zone 2. Zone 3 is characterized by a constant value for the ALUoperations/second ratio after the reduction observed at the beginning of this zone.

Hands-on methodology for using the Nios V/{m,g} processors at the Computer Architecture Laboratory:

- Modify the source code in the file `roofline.s` as described below. This code is part of the benchmark named `benchNIOSV2024_roofline` (see Figure 6). The benchmark program is integrated by the following files: `benchNIOSV2024_roofline.s`, `DIV.s`, `escribir_jtag.s`, `roofline.s`, `BCD.s`.
- By using the Nios V Command Shell and after compiling, linking, configuring the DE0-Nano board and downloading the `benchNIOSV2024_roofline.elf` file, the program should be executed repetitively using both processors, Nios V/{m,g}, to fill in the Table 2. The files for configuring the DE0-Nano board are: `DE0_NanoBasic_Computer_22jul24.sof` and `DE0_NanoBasic_Computer_23jul24.sof` for the Nios V/m- and Nios V/g-based SoC systems, respectively. For filling each line of Table 2, execute the benchmark program `benchNIOSV2024_roofline.elf` after modifying the code as indicated below.
  - Modification for Zone 1 of roofline curve: uncomment as many `lw` instructions as shown in the respective column of Table 2.
  - Modification for Zone 2 of roofline curve: uncomment as many `add` instructions as shown in the column named *ALU* in Table 2.
  - Modification for Zone 3 of roofline curve: uncomment as many iterations as shown in the column "br" in Table 2.
- Fill in one different Table 2 for each one of the Nios V/{m,g} soft processors.
- Draw a X-Y graph taking pairs with values (ALUoperations/sec, ALUoperations/memory-byte) from the respective columns of Table 2, as shown in Figure 5. One different roofline curve must be created for each Nios V processor.

- Draw another X-Y graph taking pairs with values (CPI, ALUoperations/memory-byte) from the respective columns of Table 2. One different CPI curve must be created for each Nios V processor.

The meanings of the terms shown in Table 2 are described below:

- ID: identification number of the table entry that provides a point in the X-Y roofline curve.
- `lw/sw`: number of executed instructions lw or sw coded in the LOOP section of the `roofline.s` source code.
- ALU: number of ALU instructions executed in the LOOP section of the `roofline.s` source code.
- `br/j`: number of instructions Jump of the iterator LOOP of the source code roofline.s
- ALUoperations/memory-byte: number of ALU instructions executed in each iteration by each byte that is accessed to the main memory. The value is determined by the values of the correspond columns: $ALU\ /\ (4 \times lw/sw)$.
- Niter: number of iterations executed in the LOOP section of the file `roofline.s`.
- $t_{cpu}$: execution time of the program `benchNIOSV2024_roofline` after applying the equation: $cycles/f$, where $f = 50\ MHz = 50\ \times\ 10^6\ Hz$. $t_{cpu}$ is measured in seconds.
- ALUoperations/sec: number of ALU operations done by the ALU functional unit per time unit. The value is obtained by combining the values of the columns: $N_{iter}\ \times\ ALU\ /\ t_{cpu}$.
- CPI: Average number of cycles for the program execution.The value is obtained by combining the columns: $cycles\ /\ N$.

**Question 4.**
What number of ALU operations per memory-byte the processors Nios V/{m, g} are "compute-bound", i.e., the program is limited by ALU operations? Which is the maximum number of ALU operations per second for the Nios V/m processor? What is the maximum number of ALU operations per second achieved by the Nios V/g processor? Justify and argue your answer.

**Question 5.**
Which is the maximum percentage of performance reduction for the Nios V/{m,g} processors when the branch instructions are executed?

**Question 6.**
Is the benchmark program used in Part 1, (`benchNIOSV2024_dotProduct` memory-bound or compute-bound? Justify and argue your answer.

# Part III. Influence of instruction reordering on performance of Nios V/m y Nios V/g pipelined processors

General description: A new synthetic benchmark program called *benchNIOSV2024_bypassing* is used to evaluate the effect of true data (RAW) dependencies between load instructions and ALU instructions. This benchmark program is similar to Parts 1 and 2 of this lab assignment except that the computation kernel has been modified and is now located in a file named `bypassing.s`. Next, we propose to apply the instruction reordering technique to reduce the execution time of the benchmark program using the Nios V/m and Nios V/g pipelined processors.

Three steps are proposed to do.

Step 1:

Table 2: Data for drawing the roofline curve.

| ID | kernel: `roofline.s` | | | **X coordinate** ALUoperations/ memory-byte (ALU / **4** * lw/sw) | Iterations ($N_{iter}$) | N (executed instructions, $N_{iter}$ * [lw/sw+ALU+ br/j]) | Nios V/{m,g}, frequency (f)= 50 MHz | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Number of executed instructions per iteration | | | | | | cycles | $t_{CPU}$ (sec=cycles / f) | **Y coordinate** ALUoperations/sec ($N_{iter}$ * ALU / $t_{CPU}$) | CPI (cycles / N) |
| | **lw/sw** | **ALU** | **br/j** | | | | | | | |
| 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 | 2 | | 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 | | | | | |

- Edit the file `bypassing.s` and put the instruction `add` that is data dependent on the instruction `lw` (see *Version 1* in Figure 7).
- Then, compile, link, configure the DE0-Nano board and download the `elf` file.
- For choosing Nios V/m or Nios V/g soft processor, modify the string `CONfile` in the `Makefile` file accordingly.
- Register the execution times of the program benchmark that is shown on the terminal of the Nios V Command Shell. You will have two execution times, one for Nios V/m and another for Nios V/g.

You can use the `Makefile` file to complete the actions proposed for this Step 1.

```
$ jtagconfig.exe
$ make
$ make configure
$ make download
$ juart-terminal.exe
```

Step 2:

- Now, modify the file `bypassing.s` and put the instruction `add` that is not data dependent on the instruction `lw` (see *Version 2* in Figure 7).
- Again, compile, link, configure the DE0-Nano board, download the `elf` file and register the execution times, one for Nios V/m and another for Nios V/g.

Step 3:

- Code a new version of the kernel that will be named *Version 3*.
- For this new version of the program, reorder the location of instruction in the LOOP of Version 1 of the file `bypassing.s` to eliminate the data dependence between the `lw` and `add` instructions. Execution time should reduce.
- Again, compile, link, configure the DE0-Nano board, download the `elf` file and register the execution times, one for Nios V/m and another for Nios V/g.

**Question 7.**
Quantify and indicate the improvement in execution time of the program ($t_{cpu}$) in terms of percentage (%) when there exist data dependencies "`ldw` $\rightarrow$ `add`" respect to the non-existence of this type of data dependency.

**Question 8.**
Calculate the CPI of the two Versions 1 and 2 for the program benchmark *benchNIOSV2021_bypassing*. For this exercise, count only the instructions executed in LOOP. Note that there exists a constant $Niter = 1000$ in the source code of the `bypassing.s` file.

**Question 9.**
From the two values of CPI obtained in Question 8, calculate the value of total CPI and penalization due to the true data dependency (RAW) and the percentage of penalization respect to total CPI for Versions 1 and 2, with and without penalizations, respectively. Tip: $CPI = CPI_{base} + CPI_{penalization}$.

**Question 10.**
From the two values of CPI obtained in Question 8, calculate the value of total CPI and penalization due to the true data dependency (RAW) and the percentage of penalization respect to total CPI for Versions 1 and 2, with and without penalizations, respectively. Tip: $CPI = CPI_{base} + CPI_{penalization}$.

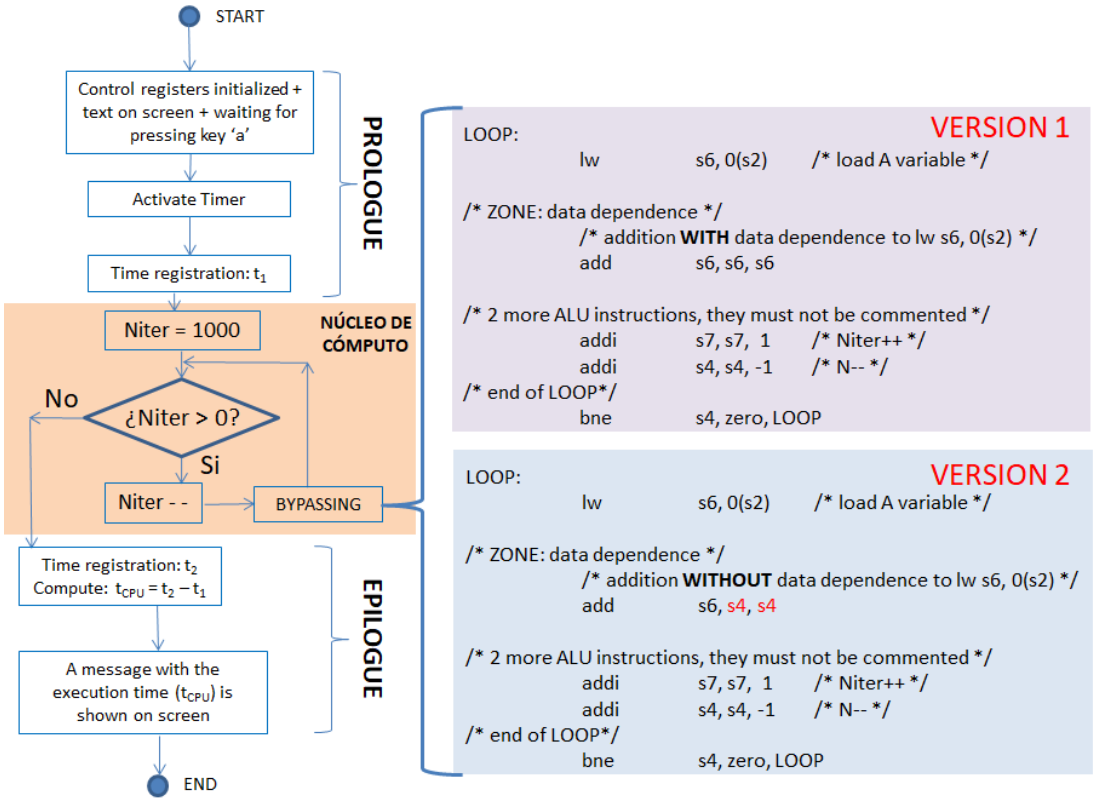


Figure 7: Flowchart for the benchmark program `benchNIOSV2024_reordering`. On the right, the main sections of the code for the kernel subroutine `BYPASSING` are shown. This code is in `bypassing.s` file.

**Question 11.**

- Calculate the speed-up of the new Version 3 respect to Version 1.
- Calculate the speed-up of the Version 2 respect to Version 1.
- Using the results of this question, what conclusion can be drawn on the performance of the Nios V pipelined processors that is related to instruction reordering technique?

# Part IV. Designing a new pipelined processor

Imagine that your boss of the Computer Architecture department at Intel ask you to evaluate a possible modification of the design of the pipelined Nios V/g processor with 5 pipes. The proposed modification consists in merging the execution phases "Execution/Calculate of address (E)" and "Access to memory (M)". Both phases will be done in the new processor in one execution phase. In this combined execution phase, the ALU unit and the memory unit work in parallel. The memory instructions leave the ALU inactive, and the arithmetic-logic instructions leave the memory unit inactive. This change will be beneficial in terms energy efficiency.

In the instructions set architecture of Nios V processors that is called *RISC-V*, the effective address to load or store is obtained by adding the content of a register (`rs1`) and an immediate value (`imm`). In this case, the problem consists in that there is no need to calculate the load or store address since memory instructions cannot access to the ALU unit.

The defenders of the new design for the Nios V processor try to change the instruction set architecture to allow only one addressing mode called *register/direct addressing*. In this addressing mode, only a source register is employed. The content of the register is the memory address that is accessed. You cannot indicate

whichever address displacement using immediate values.

In Nios V, to implement the direct addressing mode, the address is saved in a register and the value for the immediate is 0, `inm` = 0. With the proposed design, any load/store instruction uses the direct addressing mode. Immediate values different from 0, `imm` $\neq$ 0, causes two instructions to be employed. In a first instruction, the values of register and immediate should be added using an `addi` instruction. In a second instruction, the calculated address can be stored or loaded. The load/store instructions using `inm` = 0 do not require to include additional instructions in the new processor design.

**Question 12.**

- Your work consists in deciding the percentage in performance improvement, the total number of instructions that would have to execute under the new design of 4 pipes. This will require a more detailed analysis for the different types of load/store instructions executed by the `PRODUCTO_ESCALAR` subroutine of the benchmark `benchNIOSV2024_dotProduct` used in Part 1 of this lab assignment.
- Evaluate the new design depending on the increase in percentage of the executed number of instructions.
- What design do you recommend to your boss? Justify an answer quantitatively.

# References

Intel. Nios V Processor Reference Manual. Updated for Intel Quartus Prime Design Suite: 23.1. `https://cdrdv2-public.intel.com/776470/ug-683632-776470.pdf`, 2023a.

Terasic. DE0-Nano User Manual. `http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English\&No=593\&PartNo=4`, 2021.

Intel. Nios V Processor Software Developer Handbook. Updated for Intel Quartus Prime Design Suite: 23.1. `https://cdrdv2-public.intel.com/774362/ug-743810-774362.pdf`, 2023b.