INSTITUTO SUPERIOR TÉCNICO

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

COMPUTER ORGANIZATION

LEIC-A, LEIC-T

Third Lab Assignment: Instruction Level Parallelism

Version 1.0

2022/2023

1 Introduction

The main purpose of this assignment is to provide the students with a direct and close contact to the operation of a pipelined computer architecture, as well as to the techniques that are commonly applied in order to maximize the efficiency of the developed computer programs. For that purpose, a dedicated simulation environment will be adopted: the WinMIPS64 [1], version 1.57.

Due dates:

- the assignment needs to be handed in **at the beginning** of your second lab shift (week of 24/10-28/10).

1.1 Environment

WinMIPS64 is a simulator and a visual debugger of a subset of the MIPS64 Instruction Set Architecture (ISA). It was developed at the Dublin City University School of Computing (Ireland) for educational purposes and it is capable of executing small programs that comply with the supported subset of the MIPS64 ISA. A detailed description of its operation is available in the provided user manual [2].

The main reason for adopting WinMIPS64 is its educational value in helping to understand the inner workings of pipelines. It provides a graphical interface that allows users to observe the execution of instructions through the multiple stages of the pipeline. Furthermore, the user has the capability of seeing how stalls are introduced and handled by the CPU, inspecting the status of registers and memory, and controlling step by step the execution of instructions.

However, WinMIPS64 is not fully compatible with the 32-bit architecture adopted in the textbook [3] (MIPS32). As a result, the instruction set used in this assignment is slightly different from the one presented in the textbook and in the theory classes. In particular, two main differences must be considered in order to properly understand the application program introduced in Section 1.2:

- *Registers:* in WinMIPS64, all registers are 64 bits in size. There are 32 integer registers (referred to as *$0-$31*) and 32 floating-point registers (referred to as *f0-f31*).
- *Instructions:* WinMIPS64 implements the operations using the integer instructions that depicted in the following table (see [2] for more details):

MIPS64 Instruction	**Description**	**MIPS32 Equivalent**
`daddi` *reg*, *reg*, *imm*	Add 64-bit immediate	`addi` *reg*, *reg*, *imm*
`dadd` *reg*, *reg*, *reg*	Add 64-bit integers	`add` *reg*, *reg*, *reg*
`dmul` *reg*, *reg*, *reg*	Multiply 64-bit integers	`mul` *reg*, *reg*, *reg*

Two other important notes about this MIPS64 implementation:

- the pipeline has specialized execution units for multiplication and division and for addition in floating point, meaning that different instructions may spend a different number of cycles in the execution phase; note also that a structural hazard may occur if two instructions finish their execution phase on the same cycle and both try to enter the memory stage, in which case the instruction earlier in the program is given priority.
- the fact that an instruction takes longer in the execution phase may cause instructions to finish in a different order than that in the program, creating different types of potential data hazards:

RAW *read after write*, when an instruction needs to read a register that has not yet been written; this is the type of data hazards covered in the theoretical class and the only type we will analyze in this assignment.

WAW *write after write*, when an instruction later in the program tries to write to a register before another instruction earlier in the program (out of order writes).

WAR *write after read*, when an instruction needs to write to a register that another instruction earlier in the program has yet to read.

1.2 Application program

A given mathematics library makes use of the following algorithm written in pseudo-code. The outcome value is stored in the variable *mult*.

```
#define N 10
double A[N] = {1, 3, 1, 6, 4, 2, 4, 3, 9, 5};
int64 mult = A[0], i;

for(i = 1; i < N; i++) {
    mult += mult*A[i];
}
```

Figure 1 lists the MIPS64 source code that implements this mathematical function (see file `prog.s`). To provide an example of how this function is used, the program initializes the data vectors with a few sampled values. Each value is represented with a 64-bit integer.

```
            .data
A:          .word   1, 3, 1, 6, 4
            .word   2, 4, 3, 9, 5
mult:       .word   0

            .code
            daddi   $1, $0, A     ; *A[0]
            daddi   $5, $0, 0     ; $5 = 0 ;; i
            daddi   $6, $0, 10    ; $6 = N ;; N = 10
            lw      $9, 0($1)     ; $9 = A[0]  ;; mult = A[0]
            daddi   $1, $1, 8     ;

loop:       lw      $12, 0($1)    ; $12 = A[i]
            dmul    $12, $12, $9  ; $12 = $12*$9 ;; $12 = A[i]*mult
            dadd    $9, $9, $12   ; $9 = $9 + $12  ;; mult = mult+A[i]*mult

            daddi   $5, $5, 1     ; i++
            daddi   $1, $1, 8     ;
            bne     $6, $5, loop  ; Exit loop if i == N

            sw      $9, mult($0)  ; Store result
            halt
```

Figure 1: Program source code.

2 Procedure

2.1 Simple execution, without data forwarding techniques

a) Download the source code file `prog.s` from the course webpage. Copy it into your working directory and start the WinMIPS64 simulator, by executing the program `winmips64.exe`[1].

[1] In a Linux environment, this program may be executed by making use of the 'wine' platform emulator and by issuing the command: `wine winmips64.exe`

The WinMIPS64 main window is composed of a menu bar and seven frames, showing different aspects of the simulation: Pipeline, Code, Data, Registers, Statistics, Cycles and Terminal. There is also a status bar to notify the user that the simulator is running as soon as the simulation has been started.

b) Configure the simulator, in order to prevent data forwarding, by making sure that the following check boxes are **unchecked**:

```
Configure → Enable Forwarding
Configure → Enable Delay Slot
```

c) Open the downloaded program, by pursuing the following steps:

```
File → Open → prog.s
```

d) Initiate the simulation, either by issuing the command:

`Execute → Run to` (shortkey = F4)

or by running the program by single-steps:

`Execute → Single Cycle` (shortkey = F7)

e) Select an arbitrarily loop iteration (avoid the first and the last ones) of the executed program. For each instruction of such iteration represent in Table 5 the several executed stages of the pipeline: F, D, Xn, M, W. Compute the CPI while the program is executing this loop. Make sure to include in the diagram the first fetch of the next loop iteration, this is what defines the total clock cycles that a single loop iteration takes.

f) Summarize the program execution profile, by filling the table in the answer sheet at the end.

g) By analyzing the program execution, characterize the branch prediction policy that is adopted by this simulator. Justify.

2.2 Application of data forwarding techniques

a) Configure the WinMIPS64 simulator in order to activate data forwarding, by making sure that the following check box is **checked**:

```
Configure → Enable Forwarding
```

b) Repeat the previous section procedure and represent, in Table 1, the execution of the same iteration of the program loop, by representing, for each instruction, the several executed stages of the pipeline: F, D, Xn, M, W. Do not forget to represent every *Stall* that may occur.

c) Summarize the program execution profile, by filling the table in the answer sheet.

d) Evaluate the obtained *speedup*, when compared to the base setup, considered in Section 2.1.

2.3 Source code optimization: minimization of data and structural hazards

a) One common approach to reduce the still existing data and structural hazards is to apply re-order techniques [3] to the instruction sequence of the program. Keeping the simulator's data forwarding option asserted, analyze the time diagram of the previous section and apply the necessary re-ordering optimization techniques in order to minimize the Structural and Data *Stalls*. Make sure that the resulting output is kept unchanged.

b) Represent in Table 2 the execution of the selected iteration of the program loop, by representing, for each instruction, the several executed stages of the pipeline: F, D, Xn, M, W. Do not forget to represent every *Stall* that may occur.

c) Summarize the program execution profile, by filling the table in the answer sheet.

d) Compute the obtained *speedup*, when compared to the base setup, considered in Section 2.1.

2.4 Source code optimization: loop unrolling

a) One approach that is usually adopted to reduce the control hazards is to apply loop unrolling techniques [3] to the program instruction sequence. By analyzing the time diagram of the previous section, apply the loop unrolling technique in order to reduce by half the amount of resulting control hazards. Try to optimize as much as possible the body of the loop.

b) Represent in Table 3 the execution of the selected iteration of the program loop, by representing, for each instruction, the several executed stages of the pipeline: F, D, Xn, M, W. Do not forget to represent every *Stall* that may occur.

c) Summarize the program execution profile, by filling the table in the answer sheet.

d) Compute the obtained *speedup*, when compared with the base setup, considered in Section 2.1.

2.5 Source code optimization: branch delay slot

a) One alternative approach that is frequently made available by most pipeline processor implementations to reduce the control hazards penalty is based on the usage of the *Branch Delay Slot* [3]. By analyzing the time diagram of the program sequence considered in Section 2.3, repeat the application of the re-order techniques to the program considered in Section 2.3 in order to take advantage of the branch delay slot.

b) Before executing the modified file, configure the simulator in order to take advantage of the *Branch Delay Slot*, by making sure that the following check box is also **checked**:

`Configure` → `Enable Delay Slot`

c) Represent in Table 4 the execution of the selected iteration of the program loop, by representing, for each instruction, the several executed stages of the pipeline: F, D, Xn, M, W. Do not forget to represent every *Stall* that may occur.

d) Summarize the program execution profile, by filling the table in the answer sheet.

e) Compute the obtained *speedup*, when compared with the base setup, considered in Section 2.1.

References

[1] WinMIPS64. Webpage. "`http://indigo.ie/~mscott`", September 2013.

[2] Mike Scott. *WinMIPS64 Simple Tutorial*, 2008.

[3] David Patterson and John Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 5$^{\text{th}}$ edition, 2014.