Unit 2

The CPU

Labs of
Computer Architecture
v1.2

SESSION 1

Support for multitasking operating systems

Objectives

This lab focuses on two main objectives:

- Showing control switching methods between the tasks and a multitasking operating system. A well-known computer is used to study this concept: the CT, used in the Computer and Network Fundamentals course. This computer will run a very simple operating system that can execute two tasks concurrently.
- Identifying the features required for a CPU to provide support for a multitasking operating system. The CPU of the CT is not designed to support multitasking operating systems. Thus, the detected deficiencies will be used to define the requirements for a CPU to support this kind of operating systems. Finally, a real operating system on a real computer will be used to test programs that try to hack the system.

Previous knowledge and required materials

In order to get the maximum benefit from this lab session, the student is required to:

- Know the instruction set of the CT.
- Attend the lab session with a copy of the DVD with the tools provided for the course. This DVD contains a modified GNU/Linux operating system.

Session development

Before starting, you must perform the following tasks:

- ❏ Run the CT simulator.
- ❏ Configure the memory of the CT mapping 60K words in the lowest range of the address space. The balance of the address space will be free. Thus, 4K words will be available for mapping interfaces of peripheral devices.
- ❏ Map a screen interface from the address `F800h`.
- ❏ Map the interface of the lights peripheral from the address `FA00h`. Select the interrupt vector number 1 for this peripheral and check the box `Generar Int` (*Generate interrupts*) for this peripheral to interrupt the CPU. The priority level can be the default one, since this will be the only peripheral that can generate interrupts in the computer.
- ❏ Copy to your working folder the following files: `2-1os.ens`, `2-1task1-ens`, and `2-1task2.ens`.
- ❏ Copy to your working folder the CT-language assembler: `Ensambla.exe`.

1. Concurrent execution of tasks

A very simple operating system will be loaded in the memory of the CT to show how two tasks can be executed concurrently.

Firstly, the operating system will execute task 1 until an interruption is generated. Then, the operating system will take control of the CPU and will schedule task 2. Task 2 will be executed until a new interrupt is generated and the operating system takes the control of the CPU again. Then, task 1 will be scheduled again, and so on. The interrupt will be generated by pressing INT in the lights peripheral. By means of repeatedly pressing this button, you will simulate the periodical interruption generated by the timer of the system. This timer is used to ensure a maximum time period between two control switch operations performed by the operating system.

- ❏ Edit `2-1task1.ens` and `2-1task2.ens` to see their content. The appendix contains the listings of these two programs.

When the CPU executes the instruction `INT 10h`, a service call with the identifier `10h` is performed to the operating system.[1] In this case, the service routine writes on the screen the character specified by the ASCII code stored in the least

[1] The effect of executing the instruction `INT` in the CT is similar to accepting a hardware interrupt. The main difference relies on the fact that the interrupt vector is provided by software instead of by the peripheral interface.

significant byte of the register `R1`, which is loaded with the ASCII code for '1'. The most significant byte of `R1` contains the character attributes. Thus, this task will write the character '1' on the screen continually. The attribute of the character changes in each writing. Thus, the color and the background of the character will also change. Task 2 is similar to task 1, but it prints the character '2' instead.

- ❏ What memory address is task 1 loaded from? Since this task does not have data section, this address will match the initial value of the program counter, `PC`. 1
- ❏ What memory address is task 2 loaded from? 2
- ❏ Edit the `2-1os.ens` file. Read the comments at the beginning of the file to understand the functionality of the program, that is, the operating system. The listing of this file is shown in the appendix. You may notice that the assembler directive `ORIGEN 8000h` will make the CPU load the operating system in memory from the address `8000h`.
- ❏ Compile the operating system by means of the CT assembler. The `2-1os.eje` file will be generated. Several warnings appear in the compiling process. They are not important warnings, but show that the stack pointer, `R7`, has been modified by software, instead of by means of instructions managing the stack of the program.
- ❏ Load the `2-1os.eje` executable file in the CT simulator. Use the memory disassembler of the CT simulator to see how the operating system has been loaded in memory. Also, you can see the initial values of the program counter and the stack pointer.
- ❏ Compile the tasks 1 and 2 using the CT assembler. Two executable files, `2-1task1.eje` and `2-1task2.eje`, will be generated.
- ❏ The `2-1task1.eje` and `2-1task2.eje` executable files cannot be loaded directly in the CT simulator since this operation will modify the initial values of the program counter and the stack pointer of the operating system. `2-1task1.eje` and `2-1task2.eje` must be edited and their three first lines removed.[2] Save the files without these three lines with the following names: `2-1task1.mem` and `2-1task2.mem`. The `*.mem` files are files that can modify the content of memory locations of the CT simulator.
- ❏ Load the `2-1task1.mem` file from the memory address `1000h` by clicking on the memory of the CT simulator and choosing the option `Cargar desde archivo...` (*Load from file...*). Then, you can see the effect of loading this file in the memory of the computer using the CT disassembler. This loading operation is similar to the one performed by a real computer when an executable file is loaded in memory.

1

2

[2] The first three lines of an executable file for the CT contain the address from which the program is loaded in memory, the initial value for the program counter, and the initial value for the stack pointer.

- ❏ Now, it is highly recommended to save the current state of the CT simulator. Thereby, the above operations will not have to be performed again in the following tests. Save the state of the simulator to a file called `2-1state.sim`.
- ❏ Load in the memory of the CT the `2-1task2.mem` file from the address `2000h`.
- ❏ Draw in a piece of paper the organization of the address space of the CT. You must identify in the drawing the operating system, task 1, task 2 and the screen interface. To do so, you must use the addresses used to load the programs in memory and the addresses where the interface of the peripheral is mapped.
- ❏ Switch the CT simulator mode to `Run` by pressing F9. Which task is being executed? 3

3

- ❏ Press the INT button of the lights peripheral. Which task is now being executed? 4

4

- ❏ Press the INT button several more times waiting a few seconds between each click.
- ❏ In a given moment, the CT can execute the task 1 or the task 2, but never both of them at the same time. Nevertheless, if you could press the INT button each 10 ms and the CT could execute instructions at the same rate as a real computer, would you perceive that tasks 1 and 2 would be executed simultaneously even when they were running in only one CPU?[3] 5

5

- ❏ Click on the window simulator and stop the simulation by pressing F9.

2. Mechanisms for switching the control to the operating system

Switching the control to the operating system when the task calls a service provided by the former and when an interrupt is issued will be studied in this section.

The CT does not provide support for exceptions. However, an easy mechanism for managing exceptions that could be added to this computer will also be described.

[3] In modern computers something similar happens. You must take into account that the interrupt generated by the lights peripheral tries to emulate the periodical interrupt generated by the timer of a real computer. Furthermore, in a real computer it is not necessary to run a new task each time an interrupt from the timer or from a peripheral device is generated.

2.1. Switching the control to the OS in a service call

Next, the mechanism for switching the control from a task, calling the `10h` service, to the operating system is described. Furthermore, the mechanism for switching the control from the operating system to the task once the service is finished is also shown.

- ❏ Press F9 to continue the simulation process.
- ❏ If the task 2 is being executed, press INT in the lights peripheral for task 1 to be scheduled.
- ❏ You must stop the execution by pressing F9 again.
- ❏ Run the program instruction by instruction by pressing F8. Task 1 is being executed; you can follow the instructions observing the listing of this task. If the instructions that are being executed do not correspond to the instructions of the task 1, this means that the operating system is being executed while serving the `INT 10h` call. In this case, you must run the program instruction by instruction until the `IRET` instruction is executed.[4] The following instruction that must be executed is `JMP -5`, which is the result of compiling `JMP repite` in the source code of task 1.
- ❏ Run instruction by instruction by pressing F8 until you get to the `ADD R1,R1,R2` instruction. Which is the value of the program counter? [6] Which is the value of the stack pointer? [7] Write down in a piece of paper the values of registers `R0` to `R7` and `PC`.

6

7

- ❏ Press F8 again. The following instruction, `INT 10h`, is executed.[5] This instruction switches the control to the operating system to write the character specified in `R1` on the screen. This instruction results in pushing the content of `SR` and `PC` in the stack and switching the control to the first instruction referred by the interrupt vector number `10h`. Which is the new value of `PC`? [8] Try to find the address contained in `PC` in the scheme drawn in the above section. You can check that the operating system is being executed.

8

- ❏ The first instruction of the service handler is `CLI`. The goal of this instruction is disabling the interrupts to avoid accepting an interrupt while processing the service call. This may provoke a bad operation of the operating system. Then, the state of the task under execution is saved. Furthermore, the stack of the task and the stack of the operating system are switched. Press F8 to run the `MOVL R0, 0h`[6], `MOVH R0, 80h`[7], and `MOV [R0], R7` instructions. The

[4] The instruction register, `IR`, shows the instruction that has been executed.

[5] This instruction is equivalent to the `SYSCALL` instruction, mentioned in the theoretical part of the course.

[6] The compiled `MOVL R0, BYTEBAJO DIRECCION temporal` instruction in the source code.

[7] The compiled `MOVH R0, BYTEALTO DIRECCION temporal` instruction in the source code.

goal of these three instructions is saving the stack pointer of the task temporally.[8]

- ❏ Press F8 two more times. The `MOVL R7, 0h`[9] and `MOVH R7, F0h`[10] instructions are executed. Which is the new value of the stack pointer? [9] Try to find this address in the scheme drawn in the above section. The stack has been switched, from the one of the task to the one of the operating system.
- ❏ Then, a code fragment is in charge of saving the content of the registers of the task, from `R1` to `R7`, in the memory area of the operating system.[11] Press F8 several times until reaching the `MOV [R3], R0` instruction, which finishes the code fragment.
- ❏ At this point, the operations required to prepare the service have been carried out. Then, the service routine is ready to be executed. The main body of the service routine is in charge of printing the number specified in `R1` on the screen. Press F8 several times until the execution of the `MOV [R3], R4` instruction, which finishes the execution of the service routine. While you are executing these instructions, observe the screen peripheral of the CT.
- ❏ The last sequence of instructions prepares for the handler termination. To do so, it retrieves the original content of registers `R1` to `R7` of the task which called the service from the memory of the operating system.[12] Press F8 until the execution of the `MOV R7, [R0]` instruction, which prepares the stack pointer, switching from the stack of the operating system to the stack of the task. Which is the new value of the stack pointer? [10] Notice that the stack pointer points to the stack of task 1.
- ❏ Execute the next instruction, the `IRET` instruction, which switches the control back to task 1. Which is the new value of the program counter? [11] Try to find this value in the scheme drawn in the previous section. You can see that task 1 is now being executed.
- ❏ Notice how `R1` to `R7` registers contain the values they contained before the service was called. This is very important since the service should not affect the task.
- ❏ Press F9 to run the program continuously.

9

10

11

[8] Usually, real CPUs performing the stack switching operation by software store the stack pointer of the task automatically in a CPU register, speeding up the process.

[9] The compiled `MOVL R7, BYTEBAJO PTR_PILA_SO` instruction.

[10] The compiled `MOVH R7, BYTEALTO PTR_PILA_SO` instruction.

[11] The `R0` register is not saved and, thus, it cannot be used by the tasks. This register has to be sacrified to store temporal information due to limitations of the CT.

[12] Usually, a scheduling routine is invoked while processing the service in real operating systems. In this simple operating system for the CT, the task which calls the service continues its execution.

2.2. Switching the control to the OS in an interrupt

In this section, the procedure for switching the control to the operating system when an interrupt is raised by a lights peripheral device is studied. This peripheral can raise an interrupt by pressing INT. In addition, we will also study the procedure for switching the control from the operating system to the task after the interrupt has been served.

- ❏ Activate the continuous run of the CT simulator, if it is not in this state yet, by pressing F9. Check that task 1 is being executed. Press F9 again to pause the execution.
- ❏ Run the program instruction by instruction, by pressing F8. If the instructions that are now being executed by the CPU do not belong to task 1, the operating system is being executed to serve the service call invoked by the `INT 10h` instruction. In this scenario, you must press F8 several times until the `IRET` instruction, which finishes the service routine, has been executed. The next instruction to be executed by pressing F8 is `JMP -5` (the compiled result of the `JMP repite` instruction.).[13] Which is the value of the program counter? 12 Observe, in the scheme with the organization of the address space, that this value belongs to the memory area of task 1.

12

- ❏ Press INT in the lights peripheral. At this moment, an interrupt is issued. This interrupt will be accepted by the CPU since the `IF` bit of the status register is set.
- ❏ Press F8. Just after executing the following instruction of task 1, `MOVL R2, 0h`, the interrupt request will be processed. Then, the control is switched to the interrupt service routine, which belongs to the operating system. Which is the value of the program counter? 13 Observe, in the scheme of the address space drawn in the previous section, that this value belongs to the memory area of the operating system.

13

- ❏ Press F8 again. This operation triggers the execution of the first instruction of the interrupt service routine, that is, `MOVL R0, 0h` (the compiled `MOVL R0, BYTEBAJO DIRECCION temporal` instruction). As you can see in the source code of the interrupt service routine, the code which serves the service `10h` and the code which serves the interrupt differ only in its central section, the routine body.[14] Thus, the analysis carried out above is also valid for this routine. The functionality is the same since the state of the task must be saved and the stacks must be switched. Furthermore, after the body of the routine, the state of the task must be restored and the stacks must be switched again.

[13] The objective of these steps is to be sure that the service routine invoked by the task is not being executed since interrupts were disabled.

[14] The only difference outside of the routine body is that the interrupt service routine does not execute `CLI`, since the CT automatically disables the interrupts when an interrupt request is accepted.

- ❏ Press F8 several times until the execution of the `CALL +41` instruction (the compiled version of the `CALL planifica` instruction). This instruction decides the next task to be executed. In this case, the next task to be executed is task 2.
- ❏ Run the program instruction by instruction until the execution of the `IRET` instruction. Which is the value of the program counter? [14] Which is the value of the stack pointer? [15] Observe in the memory organization of your drawing that the operating system has switched the control to task 2.

[14]

[15]

2.3. Switching the control to the OS in an exception

The CT does not provide native support for exceptions. The sole situation that could raise an exception in this computer is trying to execute an invalid machine code. In this scenario, the CT simulator will generate a warning message, ignore the instruction and continue executing the following instruction of the program.

In the above scenario, a real CPU raises an exception called *invalid operation*, or similar. As a consequence, a service routine of the operating system is executed. It must be taken into account that the exception is raised by the CPU, so it knows the reason which caused the exception and where the service routine is located.

The *invalid operation* exception could be managed in the CT by making the CPU execute the service routine pointed, for instance, by the content of the memory location `0000h`, that is, the interrupt vector number 0. This exception routine would belong to the operating system and could terminate the task that was trying to execute and invalid instruction. For instance, if this operation were issued by task 1, this would mean that from that moment on task 2 would be scheduled.

3. Deficiencies in the CT

The CT has several deficiencies that do not allow it to properly support multitasking operating systems.

These deficiencies will be studied modifying the source code of task 2: malicious operations will be introduced for this task to take control of the system, affect the execution of the other task, and even halt the system.

3.1. Deficiency #1

Task 2 will execute the `CLI` instruction.

- ❏ Open the simulation file previously saved, `2-1state.sim`, in the CT simulator. Once opened, the CT simulator will have loaded the operating system and task 1. Furthermore, the memory system and the peripheral devices will be properly configured.
- ❏ Copy `2-1task2.ens` into a new file called `2-1taks2-def1.ens`. Edit the latter file and add the `CLI` instruction just before the first instruction of the program. That is, the first instruction of this task will be `CLI`. Save the file.
- ❏ Compile the `2-1task2-def1.ens` file, remove the first three lines of the executable file and save the remain lines in a file called `2-1task2-def1.mem`.
- ❏ Load the file `2-1task2-def1.mem` in the memory of the CT from the memory location `2000h`.
- ❏ Turn the simulator into the continuous execution mode (press F9).
- ❏ Press INT several times in the lights peripheral waiting several seconds between them. What has happened?

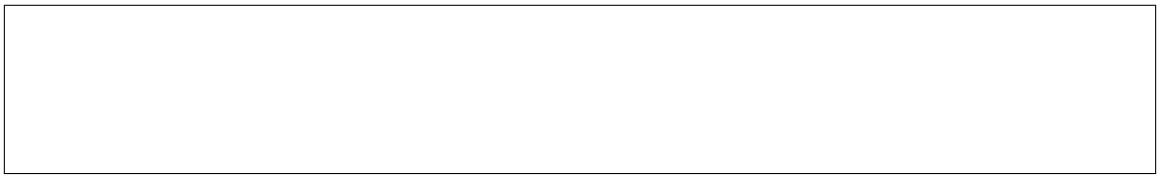
- ❏ Why do you think the interrupt service routine has not been executed? (Hint: observe the content of the `SR` register.)

- ❏ How can the CT architecture be modified to avoid this problem?

3.2. Deficiency #2

Task 2 will be modified to write in the memory area of the operating system. The value `0000h` will be written in the memory location `80CFh`.

- ❏ Load `2-1state.sim` in the CT again.
- ❏ Copy `2-1task2.ens` into a new file called `2-1taks2-def2.ens`. Edit the latter file and add the following instructions at the beginning of the loop statement: `XOR R0,R0,R0`, `MOVL R3,0CFh`, `MOVH R3,80h` and `MOV [R3],R0`. Save the file.
- ❏ Compile the `2-1task2-def2.ens` file, remove the first three lines of the executable file and save the remain lines to a file called `2-1task2-def2.mem`.
- ❏ Load the file `2-1task2-def2.mem` in the memory of the CT from the memory location `2000h`.
- ❏ Open the memory disassembler of the CT simulator and observe the instruction stored in the memory location `80CFh`. What do you think it will happen when task 2 is scheduled?



- ❏ Press F9 to turn the CT simulator into the continuous execution mode. Press INT several times, waiting a few seconds between them. Check if your prediction was correct. If you need more hints about what has happened, use the memory disassembler again to see what is the instruction stored now in the memory location `80CFh`.
- ❏ How can the CT architecture be modified to avoid this problem?

3.3. Deficiency #3

Task 2 will be modified to write inside the memory area of task 1.

- ❏ Load `2-1state.sim` in the CT again.
- ❏ Copy `2-1task2.ens` into a new file called `2-1taks2-def3.ens`. Edit the latter file and add the following instructions at the beginning of the loop statement: `MOVL R0,0FFh`, `MOVH R0, 0C0h`, `MOVL R3, 6h`, `MOVH R3, 10h` and `MOV [R3], R0`. Save the file.
- ❏ Compile the `2-1task2-def3.ens` file, remove the first three lines of the executable file and save the remain lines to a file called `2-1task2-def3.mem`.
- ❏ Load the file `2-1task2-def3.mem` in the memory of the CT from the memory location `2000h`.
- ❏ Open the memory disassembler of the CT simulator and observe the instruction stored in the memory location `1006h`. What do you think it will happen when task 2 is scheduled? Check it out.

- ❏ How can the CT architecture be modified to avoid this problem?

3.4. Deficiency #4

When the control is switched to the operating system, the values of `SR` and `PC` registers are pushed in the stack of the task that loses the control. This is an unacceptable weakness. We will see this problem.

The source code of task 2 will be modified maliciously to change the value of the stack pointer. The value `0002h` will be written in the stack pointer register.

- ❏ Copy `2-1task2.ens` into a new file called `2-1taks2-def4.ens`. Edit the latter file and add the following instructions at the beginning of the loop statement: `MOVL R7, 2h` and `MOVH R7, 0h`. Save the file.
- ❏ Compile the `2-1task2-def4.ens` file, remove the first three lines of the executable file and save the remain lines to a file called `2-1task2-def4.mem`.
- ❏ Load the file `2-1task2-def4.mem` in the memory of the CT from the memory location `2000h`.
- ❏ What do you think it will happen when task 1 is scheduled? Check it out.

- ❏ How can the CT architecture be modified to avoid this problem?

3.5. Protection in real multitasking operating systems

This section covers the protection in Linux, a real multitasking operating system, against scenarios like those detailed above. In Linux, this protection is based on the protection mechanisms provided by the CPU, which implements the AMD-64 (x86-64) architecture.

❏ Boot the PC with the course DVD inside the optical drive.

In Unix-like systems when a program terminates abruptly, a `core` file is generated in the same folder as the executable file that crashed. This file contains an image of the program memory and the content of its registers, which allows a *post mortem* analysis. However, in some systems this file is not generated by default. To set the `core` file generation when a program crashes, the following command must be run in a command-line terminal:

```
$> ulimit -c unlimited
```

❏ Copy the `2-1linux1.c` file, which is shown below, to your working folder. This program tries to disable, and enable again, the interrupts. Notice how assembly code has been inserted into a C program using `__asm__()` blocks.

```c
#include <stdio.h>
int main() {
  // Disable interrupts
  __asm__(
    "cli" /* Reset IF (bit 9) of EFLAGS */
  );

  // Enable interrupts
  __asm__(
    "sti" /* Set IF (bit 9) of EFLAGSS */
  );

  return 0;
}
```

❏ Compile and link the program with the following command:

```
$> gcc -g 2-1linux1.c -o 2-1linux1
```

The `-g` option asks the compiler to add debugging information.

❏ Run the executable file. What has happened? Why?

- ❏ The instruction which caused the program to crash can be determined by the command `gdb 2-1linux1 core`, from a command-line interface. `gdb` is the most commonly used debugger in Linux systems.
- ❏ You will see that an assembly block is referred, instead of one assembly instruction. This is because the assembly block is interpreted as a sole high-level instruction. Once this issue has been checked, you must exit the `gdb` console typing `quit`. Finally, the `core` file must be deleted with the `rm` command.
- ❏ Now, copy the `2-1linux2.c` file, whose content is listed below, to your working folder. This program attempts to write in the memory range of the Linux operating system.

```
int main() {
  // Linux is usually loaded in the range C0000000h-FFFFFFFFh
  char *p = (char *)0xFFFFE000;
  *p = 'A';

  return 0;
}
```

- ❏ Compile (adding debugging information), link and run the program. What has happened? Why?

- ❏ The instruction which caused the program to crash can be determined by the command `gdb 2-1linux2 core`. After that, you should delete the `core` file.

4. Files in your working folder

At the end of this session you will have the following files in your working folder:

- `2-1task1.eje`, `2-1task1.mem`, `2-1task2.eje`, `2-1task2.mem`, `2-1state.sim` and `2-1os.eje`.
- `2-1task2-def1.ens` to `2-1task2-def4.ens`.

- `2-1task2-def1.eje` to `2-1task2-def4.eje`.
- `2-1task2-def1.mem` to `2-1task2-def4.mem`.
- `2-1linux1.c`, `2-1linux1`, `2-1linux2.c` and `2-1linux2`.

5. Exercises

✏ Modify the `2-1task2.ens` file so that this program writes an asterisk on the upper-left corner of the screen instead of calling the `10h` service. Save the new file with the name `2-1taks2-screen.ens`. Compile and run the program in the CT simulator as usual. Is the program able to print the asterisk? Do you think this is an expected behaviour in a real multitasking operating system? How can the CT architecture be modified to avoid programs directly accessing peripheral devices?

6. Appendix

6.1. Task 1

```
1  ; Código de la tarea 1.  El resultado de compilar este fichero se
2  ; usará para generar un archivo de memoria de nombre "1-1tarea1.mem".
3  ; Para ello, bastará con borrar las 3 primeras líneas del fichero
4  ; ejecutable y renombrar el fichero obtenido usando la extensión ".mem".
5  ;
6  ; Durante su ejecución, este programa escribe contínuamente en el
7  ; periférico pantalla el número 1 que identifica la tarea 1, aunque
8  ; cada vez con un color y fondo diferente.
9
10
11         ORIGEN 1000h
12         INICIO primera
13
14         .PILA 15h
15
16         .CODIGO
17 primera:
18         MOVL R1, '1'
19         MOVH R1, 0
20 repite:
21         MOVL R2, 00h
22         MOVH R2, 01h
23         ADD R1, R1, R2
24         INT 10h
25         JMP repite
26 FIN
```

6.2. Task 2

```
1  ; Código de la tarea 2.  El resultado de compilar este fichero se
2  ; usará para generar un archivo de memoria de nombre "1-1tarea2.mem".
3  ; Para ello, bastará con borrar las 3 primeras líneas del fichero
4  ; ejecutable y renombrar el fichero obtenido usando la extensión ".mem".
5  ;
6  ; Durante su ejecución, este programa escribe contínuamente en el
7  ; periférico pantalla el número 2 que identifica la tarea 2, aunque
8  ; cada vez con un color y fondo diferente.
9
10
11         ORIGEN 2000h
12         INICIO primera
13
14         .PILA 15h
15
16         .CODIGO
17 primera:
18         MOVL R1, '2'
19         MOVH R1, 0
20 repite:
21         MOVL R2, 00h
22         MOVH R2, 01h
23         ADD R1, R1, R2
24         INT 10h
25         JMP repite
26 FIN
```

6.3. Operating system

```
1  ; Este programa contiene el código de un SO muy sencillo para el
2  ; computador von Neumann, el cual se carga a partir de la dirección
3  ; 8000h. Implementa un único servicio, el servicio 10h. Este servicio
4  ; escribe por pantalla el carácter que se le pasa a través del
5  ; registro R1. El periférico pantalla se supone ubicado a partir de la
6  ; dirección F800h.  Se supone además que hay un único dispositivo que
7  ; genera interrupciones y que tiene asociada la interrupción 1. Como
8  ; tal utilizaremos el periférico luces, ubicado a partir de la
9  ; dirección FA00h. Asimismo se supone que hay dos tareas en el
10 ; sistema, las cuales deben cargarse en memoria manualmente. Cada vez
11 ; que se produce una interrupción, se salva el estado de la tarea que
12 ; se estaba ejecutando y se recupera el estado de la otra tarea, la
13 ; cual pasa a ejecutarse.  NOTA: debido a las limitaciones de la
14 ; arquitectura de la CPU teórica, el SO modifica el registro R0 y no
15 ; lo restaura. Las tareas no deben usar este registro, pues una
16 ; transferencia de control al SO puede modificar su valor.
17
18
19 PTR_PILA_SO      EQU 0F000h ; Límite inferior de la pila del SO más 1.
20
21 ; Primera instrucción de las tareas 1 y 2 y punteros de pila
22 ; iniciales. En un sistema operativo real, estos valores se
23 ; obtendrían automáticamente de los archivos ejecutables. El PC
24 ; inicial se correspondería con el segundo valor del archivo
25 ; ejecutable, mientras que el puntero de pila inicial se
26 ; correspondería con el tercer valor.
27 ;-------------------------------------------------------
28 ;-- Para introducir más tareas hay que modificar este código
29 ;-------------------------------------------------------
30 NTAREAS          EQU 2
31 PC_TAREA_1       EQU 1000h
32 PC_TAREA_2       EQU 2000h
33 PTR_PILA_TAREA_1 EQU 101Ch
34 PTR_PILA_TAREA_2 EQU 201Ch
35
36         ORIGEN 8000h ; EL SO se ubica en los 32K más altos.
37         INICIO primera_SO
38
39         ; Variables globales del SO.
40         .DATOS
41 temporal   VALOR 0 ; Variable temporal.
42 tarea      VALOR 1 ; Índice de la tarea que se está ejecutando.
43 cursor     VALOR 0 ; Inicialmente está en la esquina superior izquierda.
44 estado_R1  VALOR NTAREAS VECES 0FFFFh ; Estado inicial de R1 a R7 de
45 estado_R2  VALOR NTAREAS VECES 0FFFFh ; las tareas. Todos pasan a almacenar
46 estado_R3  VALOR NTAREAS VECES 0FFFFh ; el valor FFFFh.
47 estado_R4  VALOR NTAREAS VECES 0FFFFh
48 estado_R5  VALOR NTAREAS VECES 0FFFFh
49 estado_R6  VALOR NTAREAS VECES 0FFFFh
50 estado_R7  VALOR NTAREAS VECES 0FFFFh
51
52         ; Código del SO.
53         .CODIGO
54 primera_SO:
55       ; Inicializar el puntero de pila del SO. La pila del SO
```

```
56        ; debe ser independiente de la pila de cada tarea.
57        MOVL R7, BYTEBAJO PTR_PILA_SO
58        MOVH R7, BYTEALTO PTR_PILA_SO
59
60        ; Instalar la rutina "escribe_car" como el servicio 10h del SO.
61        MOVL R0, 10h
62        MOVH R0, 0
63        MOVL R1, BYTEBAJO DIRECCION escribe_car
64        MOVH R1, BYTEALTO DIRECCION escribe_car
65        MOV [R0], R1
66
67        ; Instalar la rutina de servicio de la interrupción 1.
68        MOVL R0, 1
69        MOVH R0, 0
70        MOVL R1, BYTEBAJO DIRECCION rut_servicio_interr
71        MOVH R1, BYTEALTO DIRECCION rut_servicio_interr
72        MOV [R0], R1
73
74        ; Inicializar el estado de las tareas.
75        CALL inicializa_estado_tareas
76
77        ; Hacer que la primera tarea que se ejecute sea la tarea 1.
78        ; Debe notarse que tras la inicialización anterior, la pila de
79        ; la tarea 1 contiene un valor de PC que apunta a su primera
80        ; instrucción.
81        MOVL R0, BYTEBAJO DIRECCION estado_R7
82        MOVH R0, BYTEALTO DIRECCION estado_R7
83        MOV R7, [R0]
84        IRET
85
86 ;--------------------------------------------------------------------
87 ; Rutina que implementa el servicio 10h. El código ASCII del carácter
88 ; a escribir en la posición del cursor se lee del byte menos
89 ; significativo de R1. El byte más significativo de R1 contiene el
90 ; atributo.
91 PROCEDIMIENTO escribe_car
92
93        ; Para no complicar esta rutina y la de servicio de la
94        ; interrupción se inhabilitan las interrupciones durante la
95        ; ejecución de este servicio. En un SO real esto no debería
96        ; hacerse.
97        CLI
98
99        ;-------------------------------------------------------------
100       ; En el bloque siguiente se produce la conmutación de la pila
101       ; de la tarea a la del sistema operativo y se guarda el estado
102       ; de la tarea que se estaba ejecutando.
103       ; Este bloque es el mismo en todos los manejadores.
104       ;-------------------------------------------------------------
105
106       ; Almacena en la variable temporal el puntero de pila de la
107       ; tarea, para salvarlo más adelante en el lugar apropiado. Las
108       ; limitaciones de la arquitectura de la CPU teórica nos impiden
109       ; salvarlo ya directamente en dicho lugar.
110       MOVL R0, BYTEBAJO DIRECCION temporal
111       MOVH R0, BYTEALTO DIRECCION temporal
112       MOV [R0], R7
113
```

```
114         ; El nuevo puntero de pila pasa a ser el del SO. Es
115         ; decir, se lleva a cabo la commutación de la pila de la tarea
116         ; a la del SO. Debe recordarse que la interrupción no es
117         ; aceptada durante la ejecución de una interrupción anterior, ni
118         ; durante la llamada al servicio.
119         MOVL R7, BYTEBAJO PTR_PILA_SO
120         MOVH R7, BYTEALTO PTR_PILA_SO
121
122         ; Almacena en memoria el valor que tenían los registros R1 a R6
123         ; justo antes de comenzar la ejecución de la rutina de servicio
124         ; de la interrupción.
125         CALL guarda_estado_R1_a_R6
126
127         ; Aprovechando que ya hemos salvado los registros de propósito
128         ; general, podemos mover el valor salvado de R7 de la tarea
129         ; desde la variable temporal hasta el lugar apropiado.
130         MOVL R0, BYTEBAJO DIRECCION temporal
131         MOVH R0, BYTEALTO DIRECCION temporal
132         MOV R0, [R0] ; R0 = valor de R7 de la tarea.
133         MOVL R2, BYTEBAJO DIRECCION tarea
134         MOVH R2, BYTEALTO DIRECCION tarea
135         MOV R2, [R2] ; R2 = tarea
136         MOVL R3, BYTEBAJO DIRECCION estado_R7
137         MOVH R3, BYTEALTO DIRECCION estado_R7
138         ADD R3, R3, R2
139         DEC R3 ; R3 = dirección para guardar R7 de tarea.
140         MOV [R3], R0
141
142         ;-------------------------------------------------------------
143         ; Aquí comienza el cuerpo de la rutina de servicio.
144         ;-------------------------------------------------------------
145
146         MOVH R2, 0F8h
147         MOVL R2, 00h ; R2 = dir. base de la pantalla.
148         MOVL R3, BYTEBAJO DIRECCION cursor
149         MOVH R3, BYTEALTO DIRECCION cursor
150         MOV R4, [R3] ; R4 = posición del cursor.
151         ADD R2, R2, R4 ; R2 = dir. en la interfaz.
152         MOV [R2], R1 ; Escribe el carácter R1 en la interfaz.
153         INC R4 ; R4 = siguiente posición del cursor.
154         MOVL R0, 120
155         MOVH R0, 0
156         COMP R4, R0 ; Si la posición del cursor está fuera de la
157                     ; interfaz.
158         BRNZ nuevo_cursor
159         XOR R4, R4, R4 ; La nueva posición del cursor es la esquina
160                        ; superior izquierda.
161 nuevo_cursor:
162         MOV [R3], R4 ; Actualiza la variable que almacena la nueva
163                      ; posición del cursor.
164
165         ;-------------------------------------------------------------
166         ; Aquí termina el cuerpo de la rutina de servicio.
167         ;-------------------------------------------------------------
168
169         ;-------------------------------------------------------------
170         ; En el bloque siguiente se produce la conmutación de la pila
171         ; del sistema operativo a la de la tarea y se recupera el
```

```
172         ; estado de la tarea que tiene que ejecutarse a continuación.
173         ; Este bloque es el mismo en todos los manejadores.
174         ; -------------------------------------------------------------
175
176         ; Almacena en "temporal" el puntero de pila de la nueva
177         ; tarea a ejecutar.
178         MOVL R0, BYTEBAJO DIRECCION estado_R7
179         MOVH R0, BYTEALTO DIRECCION estado_R7
180         MOVL R2, BYTEBAJO DIRECCION tarea
181         MOVH R2, BYTEALTO DIRECCION tarea
182         MOV R2, [R2] ; R2 = tarea
183         ADD R0, R0, R2
184         DEC R0
185         MOV R0, [R0]
186         MOVL R1, BYTEBAJO DIRECCION temporal
187         MOVH R1, BYTEALTO DIRECCION temporal
188         MOV [R1], R0
189
190         ; Recupera el estado de los registros R1 a R6. La recuperación
191         ; depende del valor de la variable "tarea".
192         ; Nota: el registro R7 se recupera justo después.
193         CALL recupera_estado_R1_a_R6
194
195         ; El nuevo puntero de pila es el de la tarea. Es decir, se
196         ; produce la conmutación de la pila del SO a la de la tarea a
197         ; ejecutar
198         MOVL R0, BYTEBAJO DIRECCION temporal
199         MOVH R0, BYTEALTO DIRECCION temporal
200         MOV R7, [R0]
201
202         IRET ; Una vez se ejecuta esta intrucción toma el control la tarea.
203 FINP
204
205
206 ;-------------------------------------------------------------------------
207 ; Rutina encargada de inicializar el estado de las tareas
208 PROCEDIMIENTO inicializa_estado_tareas
209
210         ; Salva los registros R1 a R3 antes de ser modificados.
211         ; Nota: el registro R0 nunca se salva.
212         PUSH R1
213         PUSH R2
214         PUSH R3
215
216         ; R2 = 0001. Almacenará el nuevo valor de SR que se guardará en
217         ; la pila de cada tarea. El bit IF será 1.
218         MOVL R2, 1
219         MOVH R2, 0
220
221         ;-------------------------------------------------------------------
222         ;-- Para cada tarea debe repetirse el siguiente fragmento de código
223         ;-------------------------------------------------------------------
224
225         ; Inicializa el contenido de la pila de la tarea 1, como si se
226         ; hubiese producido una interrupción justo antes de ejecutarse
227         ; su primera instrucción.
228         MOVL R0, BYTEBAJO PC_TAREA_1
229         MOVH R0, BYTEALTO PC_TAREA_1 ; R0 = PC_TAREA_1
```

```
230         MOVL R1, BYTEBAJO PTR_PILA_TAREA_1
231         MOVH R1, BYTEALTO PTR_PILA_TAREA_1
232         DEC R1
233         MOV [R1], R2
234         DEC R1
235         MOV [R1], R0
236
237         ; Inicializa el estado salvado del registro R7 de la tarea 1.
238         MOVL R3, BYTEBAJO DIRECCION estado_R7
239         MOVH R3, BYTEALTO DIRECCION estado_R7
240         MOV [R3], R1
241
242         ; Inicializa el contenido de la pila de la tarea 2, como si se
243         ; hubiese producido una interrupción justo antes de ejecutarse
244         ; su primera instrucción.
245         MOVL R0, BYTEBAJO PC_TAREA_2
246         MOVH R0, BYTEALTO PC_TAREA_2 ; R0 = PC_TAREA_2
247         MOVL R1, BYTEBAJO PTR_PILA_TAREA_2
248         MOVH R1, BYTEALTO PTR_PILA_TAREA_2
249         DEC R1
250         MOV [R1], R2
251         DEC R1
252         MOV [R1], R0
253
254         ; Inicializa el estado salvado del registro R7 de la tarea 2.
255         INC R3
256         MOV [R3], R1
257
258         ; Recupera los registros modificados.
259         POP R3
260         POP R2
261         POP R1
262
263         RET
264 FINP
265
266
267 ;-------------------------------------------------------------------
268 ; Rutina de servicio de la interrupción 1. Esta interrupción se lanza
269 ; cada vez que el usuario pulsa el botón de generación de interrupción
270 ; del periférico luces, con la cual se trata de emular la interrupción
271 ; periódica del temporizador del sistema operativo.
272 PROCEDIMIENTO rut_servicio_interr
273
274         ;-------------------------------------------------------------
275         ; En el bloque siguiente se produce la conmutación de la pila
276         ; de la tarea a la del sistema operativo y se guarda el estado
277         ; de la tarea que se estaba ejecutando.
278         ; Este bloque es el mismo en todos los manejadores.
279         ;-------------------------------------------------------------
280
281         ; Almacena en la variable temporal el puntero de pila de la
282         ; tarea, para salvarlo más adelante en el lugar apropiado. Las
283         ; limitaciones de la arquitectura de la CPU teórica nos impiden
284         ; salvarlo ya directamente en dicho lugar.
285         MOVL R0, BYTEBAJO DIRECCION temporal
286         MOVH R0, BYTEALTO DIRECCION temporal
287         MOV [R0], R7
```

```
288
289         ; El nuevo puntero de pila pasa a ser el del SO. Es
290         ; decir, se lleva a cabo la commutación de la pila de la tarea
291         ; a la del SO. Debe recordarse que la interrupción no es
292         ; aceptada durante la ejecución de una interrupción anterior, ni
293         ; durante la llamada al servicio.
294         MOVL R7, BYTEBAJO PTR_PILA_SO
295         MOVH R7, BYTEALTO PTR_PILA_SO
296
297         ; Almacena en memoria el valor que tenían los registros R1 a R6
298         ; justo antes de comenzar la ejecución de la rutina de servicio
299         ; de la interrupción.
300         CALL guarda_estado_R1_a_R6
301
302         ; Aprovechando que ya hemos salvado los registros de propósito
303         ; general, podemos mover el valor salvado de R7 de la tarea
304         ; desde la variable temporal hasta el lugar apropiado.
305         MOVL R0, BYTEBAJO DIRECCION temporal
306         MOVH R0, BYTEALTO DIRECCION temporal
307         MOV R0, [R0] ; R0 = valor de R7 de la tarea.
308         MOVL R2, BYTEBAJO DIRECCION tarea
309         MOVH R2, BYTEALTO DIRECCION tarea
310         MOV R2, [R2] ; R2 = tarea
311         MOVL R3, BYTEBAJO DIRECCION estado_R7
312         MOVH R3, BYTEALTO DIRECCION estado_R7
313         ADD R3, R3, R2
314         DEC R3 ; R3 = dirección para guardar R7 de tarea.
315         MOV [R3], R0
316
317         ;-------------------------------------------------------------
318         ; Aquí comienza el cuerpo de la rutina de interrupción.
319         ;-------------------------------------------------------------
320
321         ; Calcula el índice de la nueva tarea a ejecutar, el
322         ; cual escribe en la variable "tarea".
323         CALL planifica
324
325         ;-------------------------------------------------------------
326         ; Aquí termina el cuerpo de la rutina de interrupción.
327         ;-------------------------------------------------------------
328
329         ;-------------------------------------------------------------
330         ; En el bloque siguiente se produce la conmutación de la pila
331         ; del sistema operativo a la de la tarea y se recupera el
332         ; estado de la tarea que tiene que ejecutarse a continuación.
333         ; Este bloque es el mismo en todos los manejadores.
334         ; ------------------------------------------------------------
335
336         ; Almacena en "temporal" el puntero de pila de la nueva
337         ; tarea a ejecutar.
338         MOVL R0, BYTEBAJO DIRECCION estado_R7
339         MOVH R0, BYTEALTO DIRECCION estado_R7
340         MOVL R2, BYTEBAJO DIRECCION tarea
341         MOVH R2, BYTEALTO DIRECCION tarea
342         MOV R2, [R2] ; R2 = tarea
343         ADD R0, R0, R2
344         DEC R0
345         MOV R0, [R0]
```

```
346         MOVL R1, BYTEBAJO DIRECCION temporal
347         MOVH R1, BYTEALTO DIRECCION temporal
348         MOV [R1], R0
349
350         ; Recupera el estado de los registros R1 a R6. La recuperación
351         ; depende del valor de la variable "tarea".
352         ; Nota: el registro R7 se recupera justo después.
353         CALL recupera_estado_R1_a_R6
354
355         ; El nuevo puntero de pila es el de la tarea. Es decir, se
356         ; produce la conmutación de la pila del SO a la de la tarea a
357         ; ejecutar
358         MOVL R0, BYTEBAJO DIRECCION temporal
359         MOVH R0, BYTEALTO DIRECCION temporal
360         MOV R7, [R0]
361
362         IRET ; Una vez se ejecuta esta intrucción toma el control la tarea.
363 FINP
364
365
366 ;-------------------------------------------------------------------------
367 ; Guarda el estado de los registros R1 a R6. El lugar en que debe
368 ; guardarlos depende del valor de la variable "tarea".
369 PROCEDIMIENTO guarda_estado_R1_a_R6
370
371         ; R6 = número de tareas.
372         PUSH R6 ; Lo salvamos antes de modificarlo.
373         MOVL R6, BYTEBAJO NTAREAS
374         MOVH R6, BYTEALTO NTAREAS
375
376         ; R0 = dirección de almacenamiento del valor de R1.
377         MOVL R0, BYTEBAJO DIRECCION estado_R1
378         MOVH R0, BYTEALTO DIRECCION estado_R1
379         PUSH R1 ; Lo salvamos antes de modificarlo.
380         MOVL R1, BYTEBAJO DIRECCION tarea
381         MOVH R1, BYTEALTO DIRECCION tarea
382         MOV R1, [R1] ; R1 = tarea
383         ADD R0, R0, R1
384         DEC R0
385
386         ; Guarda R1.
387         POP R1
388         MOV [R0], R1
389
390         ; Guarda R2.
391         ADD R0, R0, R6
392         MOV [R0], R2
393
394         ; Guarda R3.
395         ADD R0, R0, R6
396         MOV [R0], R3
397
398         ; Guarda R4.
399         ADD R0, R0, R6
400         MOV [R0], R4
401
402         ; Guarda R5.
403         ADD R0, R0, R6
```

```
404         MOV [R0], R5
405
406         ; Guarda R6.
407         ADD R0, R0, R6
408         POP R6
409         MOV [R0], R6
410
411         RET
412 FINP
413
414
415 ;--------------------------------------------------------------------
416 ; Calcula la siguiente tarea a ejecutar y almacena su índice en la
417 ; variable "tarea".
418 PROCEDIMIENTO planifica
419         PUSH R1
420         PUSH R2
421
422         ; R0 = dirección de la variable "tarea".
423         MOVL R0, BYTEBAJO DIRECCION tarea
424         MOVH R0, BYTEALTO DIRECCION tarea
425
426         ; R1 = última tarea que se estaba ejecutando.
427         MOV R1, [R0]
428
429         ; R2 = número de tareas
430         MOVL R2, BYTEBAJO NTAREAS
431         MOVH R2, BYTEALTO NTAREAS
432
433         ; última tarea = última tarea + 1
434         INC R1
435
436         ; Si última tarea > número de tareas => última tarea = 1.
437         COMP R2, R1
438         BRNS final
439         MOVL R1, 1
440         MOVH R1, 0
441 final:
442         MOV [R0], R1 ; Actualiza el valor de la variable "tarea".
443
444         POP R2
445         POP R1
446         RET
447 FINP
448
449
450 ;--------------------------------------------------------------------
451 ; Recupera el estado de los registros R1 a R6. El lugar desde el que debe
452 ; leerlos depende del valor de la variable "tarea".
453 PROCEDIMIENTO recupera_estado_R1_a_R6
454
455         ; R0 = dirección de almacenamiento del valor de R1.
456         MOVL R0, BYTEBAJO DIRECCION estado_R1
457         MOVH R0, BYTEALTO DIRECCION estado_R1
458         MOVL R1, BYTEBAJO DIRECCION tarea
459         MOVH R1, BYTEALTO DIRECCION tarea
460         MOV R1, [R1]
461         ADD R0, R0, R1
```

```
462         DEC R0
463
464         ; R6 = número de tareas.
465         MOVL R6, BYTEBAJO NTAREAS
466         MOVH R6, BYTEALTO NTAREAS
467
468         ; Recupera R1
469         MOV R1, [R0]
470
471         ; Recupera R2
472         ADD R0, R0, R6 ; R0 = dirección de almacenamiento del valor de R2.
473         MOV R2, [R0]
474
475         ; Recupera R3
476         ADD R0, R0, R6 ; R0 = dirección de almacenamiento del valor de R3.
477         MOV R3, [R0]
478
479         ; Recupera R4
480         ADD R0, R0, R6 ; R0 = dirección de almacenamiento del valor de R4.
481         MOV R4, [R0]
482
483         ; Recupera R5
484         ADD R0, R0, R6 ; R0 = dirección de almacenamiento del valor de R5.
485         MOV R5, [R0]
486
487         ; Recupera R6
488         ADD R0, R0, R6 ; R0 = dirección de almacenamiento del valor de R6.
489         MOV R6, [R0]
490
491         RET
492 FINP
493
494 FIN
```

SESSION 2

Performance improvement using pipeline

Objectives

In this lab session the MIPS64 architecture is introduced. Its microarchitecture, which is segmented in five stages, will be analyzed using a simulator. The main objectives of this lab session are:

- First contact with MIPS64 architecture.
- Understand the operation of a pipelined CPU, as well as the dependencies that may stall the pipeline.
- Know some of the techniques used to minimize the pipeline stalling.

The *WinMips64* simulator is required to develop this lab session. This simulator run on Windows, so booting the PC with the course *Live-DVD* is not required.

Previous knowledge and required materials

In order to get the maximum benefit from this lab session, the student is required to:

- Review the course materials regarding pipeline before attending the labs.

Session development

1. Pipeline hazards

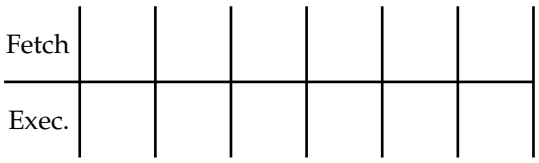
Pipeline hazards prevent the different stages of the pipeline to run in parallel. In these situations the concurrent execution is paused and the pipeline is stalled. Thus, some stages may wait until others finish their tasks. Therefore, the throughput of the pipeline will be lower than expected.

Some situations which may stall the pipeline were described in the lectures. They were exemplified with a very simplified pipelined design for the CT. In this design, the execution of every instruction is divided in two stages. Firstly, the instruction is fetched and the program counter is updated. Secondly, the instruction is executed. In this CPU, the pipeline can be stalled in two different scenarios: 1) changing the execution flow of the program; or 2) simultaneously accessing shared resources in the stages of the pipeline. Next, these scenarios are described in depth.

Changing the execution flow of a program provokes stalling the pipeline, as the next example shows.

- ❏ Fill in the schedule of the following code snippet. The execution of every instruction is divided into two stages. Each column of the schedule represents a clock cycle. You must take into account that each stage of the pipeline consumes one clock cycle, and all functional units of the CPU are replicated to avoid stalling the pipeline while accessing these resources.

```
1 MOV  R1, R3
2 JMP  +1
3 ADD  R1, R1, R2
4 XOR  R1, R1, R1
```



- ❏ Assuming that the clock frequency of the CPU of the CT is 200 MHz, how long does it take to execute one instruction, that is, what is the response time of the execution of an instruction?[1] How long does it take to execute the above snippet?[2]
- ❏ What is the throughput, measured in instructions per cycle, in the execution of the above snippet in a non-pipelined version of the CPU?[3]
- ❏ What is the throughput, measured in instructions per cycle, in the execution of the above snippet in the pipelined version of the CPU?[4]
- ❏ The maximum speedup in the throughput of a pipelined CPU with regards to the non-pipelined version of the CPU is equal to the number of stages of the pipeline. Thus, what is the speedup of the throughput in the pipelined CPU with regards to the non-pipelined CPU in the execution of the above snippet?[5]

1

2

3

4

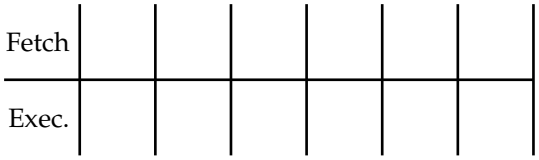
5

Notice that the `JMP +1` instruction provokes a change in the execution flow of the program, breaking its sequential execution. This breakage invalidates the fetch stage of the following instruction until the destination of the jump is computed.[1] Therefore, the pipeline is stalled. These situations are called **control hazards**.

Now, another situation which also stalls the pipeline is shown: accessing shared, non-replicated resources in the computer, such as the memory.

- ❏ Assume that there is only one memory unit in the computer where code and data are stored. Also, assume that the rest of functional units are replicated and each stage of the pipeline consumes one clock cycle. Fill in the schedule of the execution of the next code snippet.

```
1 XOR  R1, R1, R1
2 MOVL R1, 32h
3 MOV  R0, [R1]
4 ADD  R0, R0, R0
```



- ❏ If the clock frequency of the CT is 200 MHz, what is the response time in the execution of the above snippet?[6]
- ❏ What is the throughput, measured in instructions per cycle, in the execution of the above snippet?[7]
- ❏ What is the speedup in the execution of the above snippet in a pipelined CPU with regards to the execution in a non-pipelined CPU?[8]

6

7

8

In this case, the `MOV R0, [R1]` instruction accesses the memory, provoking the pipeline to stall since fetching the following instruction also requires accessing the memory.[2] These situations where the pipeline is stalled while accessing shared resources are called **structural hazards**.

In the rest of the lab session, control and structural hazards are exemplified in the MPIS64 architecture. Furthermore, a new type of hazards that does not appear in simple pipelined CPUs, like that proposed for the CPU of the CT, is shown: **data hazards**.

2. Introduction to the MIPS64 architecture

MIPS64, an extension to the MIPS32 architecture, is a RISC architecture based on load/store operations. Embedded systems, such as network devices, as well

[1] Actually, the destination of the jump can be determined from the instruction itself since it is an unconditional branch. Advances, such as the branch prediction units of modern CPUs, help to improve the performance of pipelined CPUs.

[2] In modern CPUs this is avoided by implementing a Harvard architecture approach, with a memory unit for code and another memory unit for data, concurrently accessible.

as game consoles, such as the Sony PlayStation Portable, are examples of systems using this architecture.

The MIPS64 architecture is a 64-bit architecture with a very simple instruction set. It was released in 1991 and it is considered the first 64-bit architecture. It consists of a set of 32 general purpose registers of 64 bits each, called `R0`, ..., `R31`; even though `R0` always contains 0 and `R31` is reserved for specific tasks. Also, 32 floating-point registers of 64 bits each, called `FP0`, ..., `FP31`, are available. Furthermore, this architecture provides two status registers: one for integer and another one for floating-point operations.

Each memory word is 1-byte wide and is addressed with 64 bits; the endianness in the memory access is *big-endian*, that is, data items occupying more than one memory location (1 byte) are stored in memory from the most to the least significant byte. The memory read (load) and memory write (store) operations are always carried out over general purpose registers (`Rx` or `FPx`). When using integer registers, several data sizes can be addressed: 1 byte, half-word (2 bytes), word (4 bytes), and double-word (64 bits). Floating-point registers may address one memory word in simple precision and two memory words in double precision. In any case, memory accesses must be aligned, that is, the memory address must be multiple of 4.

In this architecture, instructions are 32-bit wide. Examples of instruction types are:

- Load/store instructions. They allow memory reading and writing. A general purpose register, either integer or floating-point, is always required.
- Arithmetic and logic instructions.
- Flow control instructions.
- Floating-point instructions.

MIPS64 only provides one addressing mode, requiring a base register and an offset. That is, the memory address to be accessed is computed as the addition of the content of an integer register plus an immediate value.

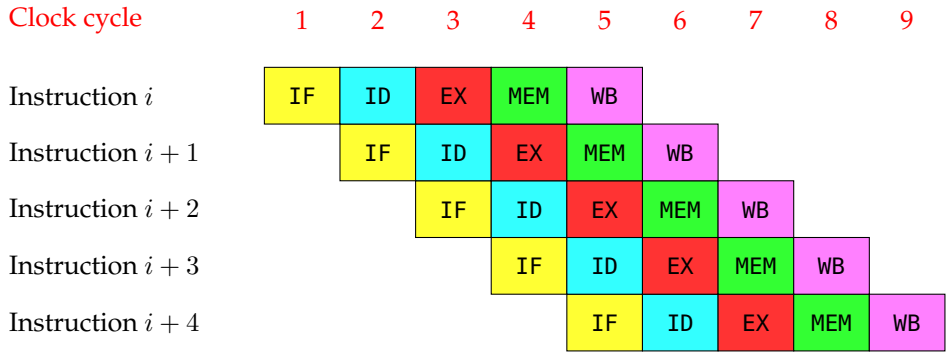
2.1. Pipeline

The MIPS64 architecture defines a pipeline based on five stages:

1. `IF`: instruction fetch and `PC` updating.
2. `ID`: instruction decode and register read.
3. `EX`: execution and effective address computation.
4. `MEM`: memory access.

5. `WB`: write back.

Next, the execution scheme is shown. When the pipeline is full, five instructions run in parallel, as many as stages.



As you already know, pipelining the execution of instructions improves the throughput of the CPU. In an ideal scenario, the number of instructions executed per second in a pipelined CPU is the number of instructions per second in the non-pipelined version of the CPU multiplied by a factor equivalent to the number of stages. Nevertheless, pipeline hazards limit this factor. In the following sections a MIPS64 simulator is used to simulate the execution of programs in this architecture, observing the operation of the pipeline.

3. WinMips64 simulator

WinMips64 allows simulating programs written for the MIPS64 architecture, showing the operation of the pipeline.

❑ Run the simulator. A window similar to that represented in figure 2.1 will show up.

As you can see, six windows appear displaying several views of the simulation process. The content of these windows is briefly described next:

- `Code` (figure 2.2): Shows the listing of the program loaded in the simulator; one instruction per line. The first column shows the offset from the base address used to load the code in the memory of the computer; the second column shows the machine code of the instruction, in hexadecimal; the third column shows the instruction mnemonic.
- `Register`: Shows the content of the CPU registers, although the control registers are omitted. General purpose registers are called `R0` ...`R31`, while floating-point registers are called `F0` ...`F31`.
- `Clock Cycle Diagram` (figure 2.3): Shows the evolution of the pipeline. Each row-column combination contains a label identifying the stage of the pipeline.

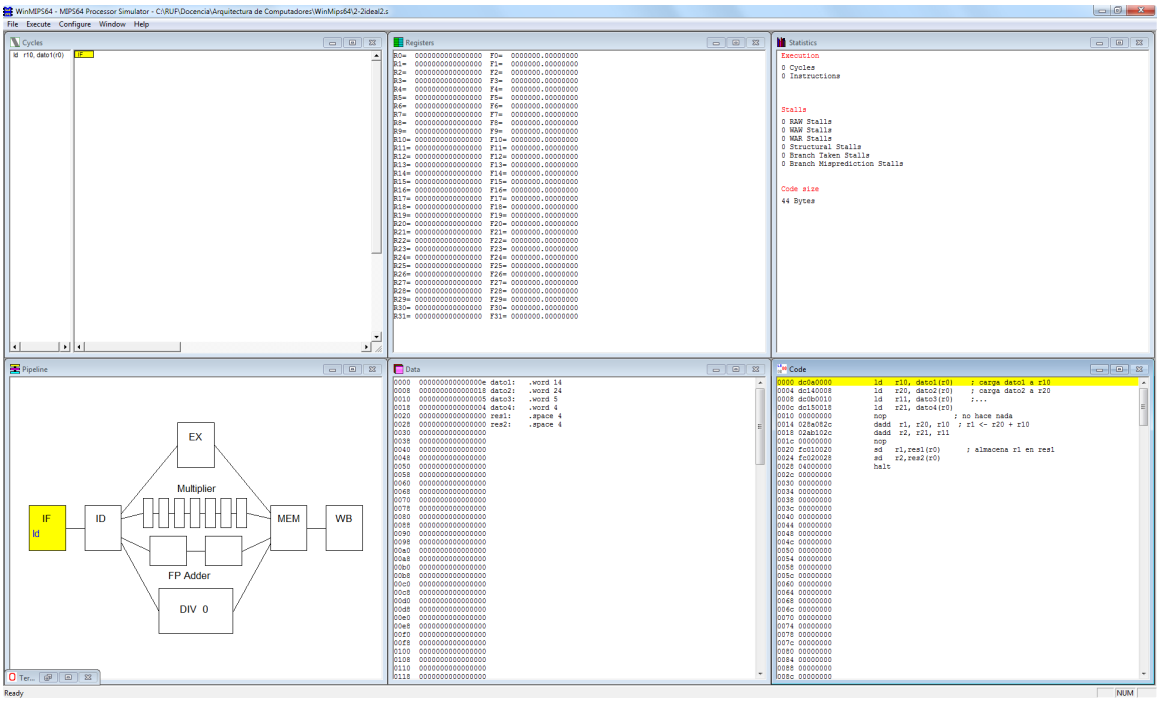


Figure 2.1: WinMips64 main window

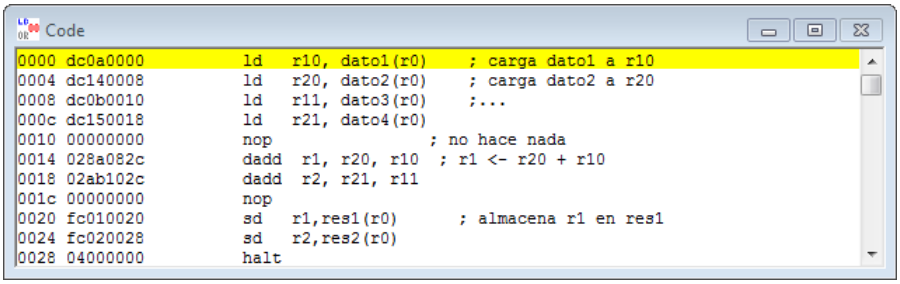


Figure 2.2: WinMips64 code window

- `Pipeline` (figure 2.4): Shows the evolution of the pipeline in the execution process. The functional units of the datapath are shown.
- `Statistics`: Display statistics of the current simulation process.

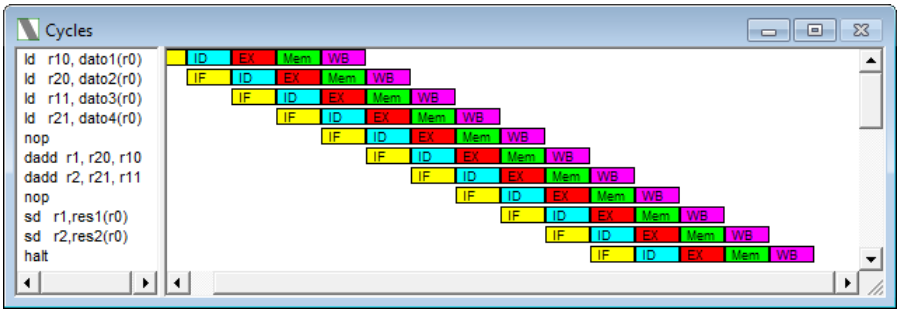


Figure 2.3: WinMips64 clock cycle window

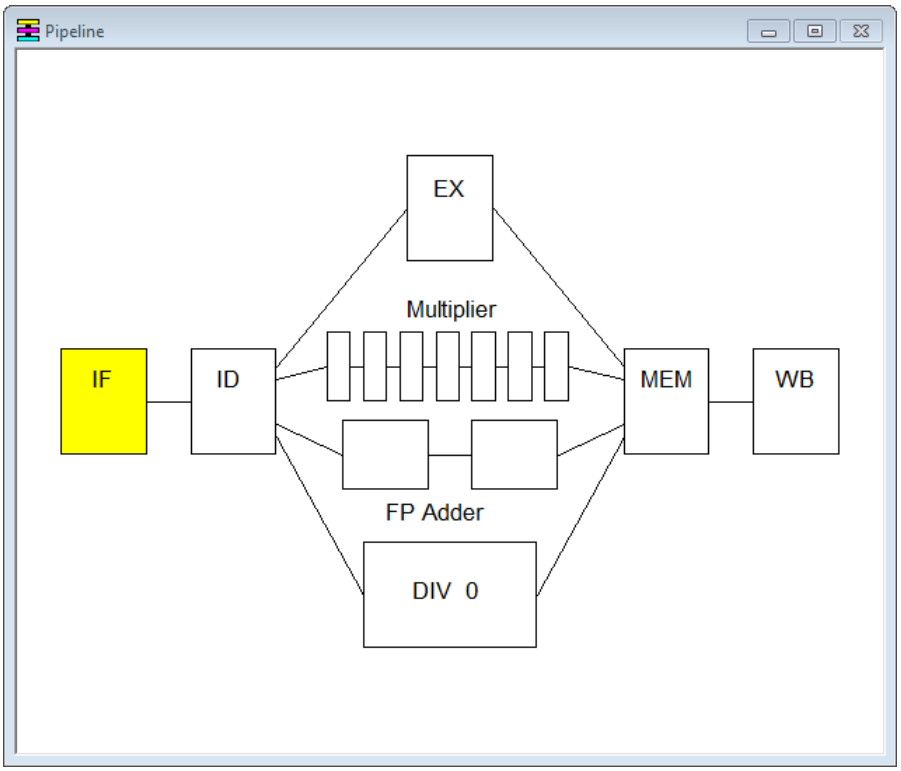


Figure 2.4: WinMips64 pipeline window

The simulation process begins by loading a source file in the simulator. To do so, you may use the `Open` option of the `File` menu. Then, the simulator allows simulating the complete execution of the program, once clock cycle or several cycles. These options can be found in the `Execute` option of the menu.

4. Pipeline basic operation

The operation of the pipeline is shown by means of a small program. This program performs two additions of four integer numbers stored in memory, and stores both results in two memory locations.

```
.data

var1:   .word 14
var2:   .word 24
var3:   .word 5
var4:   .word 4
res1:   .space 8
res2:   .space 8

.code

main:
         ld    r10, var1(r0)    ; load var1 in r10
         ld    r20, var2(r0)    ; load var2 in r20
         ld    r11, var3(r0)    ;...
         ld    r21, var4(r0)
```

```
        nop                     ; nop = no operation
        dadd  r1, r20, r10      ; r1 <- r20 + r10
        dadd  r2, r21, r11
        nop
        sd    r1,res1(r0)         ; store r1 in res1
        sd    r2,res2(r0)
        halt
```

- ❏ Edit the `2-2ideal.s` file containing the above program.
- ❏ Analyze the source code trying to understand it. The `ld r10, var1(r0)` instruction loads the value of `var1` in the `r10` register. In this case, `r0` is used as a base register with value `0`. The `dadd` instruction adds 64-bit numbers. The `sd r1, res1(r0)` instruction stores the content of `r1` in the `res1` variable. As you can see, the position of the source and destination operator changes between load and store instructions.
- ❏ Assuming a sequential execution of the stages of the pipeline, how many clock cycles are required to run the above program?[9]
- ❏ If the clock frequency of this CPU is 400 MHz, how long does it take to execute one instruction?[10] How long does it take to run the whole program?[11]
- ❏ What is the average theoretical throughput of this CPU measured in MIPS?[12]
- ❏ Assuming now that the five stages of the pipeline can run in parallel, what is the maximum theoretical throughput of this CPU expressed in MIPS?[13]
- ❏ Assuming that no instruction provokes stalling the pipeline, how many clock cycles are required to run the program?[14]

9

10

11

12

13

14

Now, the above answers will be checked by means of the WinMips64 simulator.

- ❏ Run the simulator.
- ❏ Load the `2-2ideal.s` file.
- ❏ At this point, the simulator is ready to start the simulation process. Open the Code and Clock Cycle Diagram windows if they are not open yet.
- ❏ After loading the file, the first stage of the first instruction is about to be executed. Observe the Code window. The first instruction is highlighted, which means its first stage is going to be executed. Also, the Clock Cycle Diagram window shows this fact. The first instruction goes into the pipeline, in the `IF` stage.
- ❏ Press F7 to simulate a new clock cycle. The `ld r10, var1(r0)` instruction goes into the `ID` stage, whereas a new instruction goes into the first stage of the pipeline.

- ❏ Press F7 three more times. The clock cycle #5 is being simulated (the Statistics window will display *4 Cycles* since it considers this cycle is not finished yet). At this moment, all the stages of the pipeline are working in parallel. The initial transitory time, from the moment in which the pipeline was empty until it is full, has been finished. This transitory time occurs when the CPU is rebooted. At the end of the clock cycle #5, the execution of the first instruction will be finished.
- ❏ Press F7 one more time. The first instruction has finished and the last stage of the second instruction is executed in this clock cycle.
- ❏ Press F4 to run the simulation process until the end. After the execution of the `halt` instruction, the simulator stops the simulation process. At this moment, the program has finished. How many clock cycles have been required to run the program?[15] Does this result match your previous answer?

[15]

5. Control hazards

Control hazards appear in the pipeline when the execution flow of the program changes as a consequence of calling a function, issuing an interrupt or raising an exception. Under these circunstances the sequential execution flow is broken and the CPU cannot execute the initial stages of the following instructions until the destination of the branch, interrupt or exception is resolved. Next, an example is shown.

```
.code

main:
         xor   r10, r10, r10
         daddi r20, r0, 12
         daddi r21, r0, 3
         daddi r22, r0, 2
         j     dest           ; jump (unconditional)
         nop
         xor   r20, r20, r20
dest:
         dsub  r20, r20, r21
         dadd  r22, r22, r21
         halt
```

- ❏ Edit the `2-2contr.s` file and analyze its code. It is a very simple program with an unconditional jump, `j dest`, which breaks the sequential execution flow of the program and modifies the program counter with the memory address where the machine code of the `dsub` instruction is stored in memory.
- ❏ If the pipeline was not stalled, how many clock cycles would the CPU take to execute the above program?[16]

[16]

- ❏ To check your answer with the simulator, open the `2-2contr.s` file.
- ❏ Have a look at the Code window. The column on the left is the memory address from which the machine code of each instruction is stored in memory; each instruction is encoded using 4 bytes.
- ❏ Press F7 several times until the `j dest` instruction goes into the first stage of the pipeline, `IF`, that is, until the clock cycle #5. At the end of this clock cycle the first stage of the instruction will be finished and, thus, the machine code will have been fetched.
- ❏ Look at the Clock Cycle Diagram window and read the source code of the program again. Press F7 to start executing the `ID` stage of the `j dest` instruction; the `nop` instruction goes into the first stage of the pipeline. This is because the pipeline always assumes a sequential execution of the instructions. At the end of this stage, clock cycle #6, the CPU decodes the `j dest` instruction and detects a branch in the execution flow of the program.
- ❏ Press F7 again. The CPU has identified the branch instruction and knows that the program counter must be modified (it is an unconditional jump). Thus, the execution of the `nop` instruction is interrupted and the first stage of the instruction which is the destination of the jump is executed, that is, `dsub r20, r20, r21`.
- ❏ Simulate the execution of the program until the end by pressing F4. Check that you can understand what is happening in the pipeline in each clock cycle. How many cycles has the CPU required to execute the program?[17] What is the speedup when comparing this result with that in an ideal execution without stalling the pipeline?[18]

[17]

[18]

6. Structural hazards

Structural hazards appear when several pipeline stages try to simultaneously use shared resources that are not replicated. In the example of the pipeline of the CPU of the CT these situations appear when two instructions tried to use the ALU, for instance. Nevertheless, the MIPS64 architecture is optimized to avoid these situations by means of replicating some shared resources.

The execution of floating-point arithmetic operations is carried out in the floating-point stages. The simulator allows configuring the number of clock cycles consumed by the CPU when performing a floating-point operation. Providing a value greater than one means that each floating-point operation requires several stages, since floating-point units are usually pipelined.

The structural hazards in the pipeline will be exemplified by means of a program which operates with floating-point numbers.

```
.data

var1:   .double 3.5
var2:   .double 1.7
var3:   .double 2.8
var4:   .double 0.4
res1:    .space 8
res2:    .space 8

.code

main:
         l.d   f10, var1(r0)
         l.d   f20, var2(r0)
         l.d   f11, var3(r0)
         l.d   f21, var4(r0)
         nop
         add.d f0, f20, f10       ; floating-point addition f0 <- f20 + f10
         add.d f1, f21, f11
         nop
         nop
         s.d   f0, res1(r0)       ; store f1 in res1
         s.d   f1, res2(r0)
         halt
```

- ❏ Edit the `2-2struc.s` file, analyze its content and try to understand its meaning.
- ❏ If the pipeline was not stalled while executing this program, and assuming that the functional unit for adding floating-point numbers required the same time to perform the operation that the rest of the stages, how many clock cycles would the CPU require to execute the above program?[19]
- ❏ Load the program in the simulator.
- ❏ Configure the latency of the floating-point addition unit (the number of stages in the floating-point addition operation). To do so, select `Architecture` in the `Configure` option of the menu. Specify that the floating-point unit requires 2 clock cycles (*FP Addition Latency* must be set to 2).
- ❏ Simulate the program until the `ID` stage of the `add.d f0, f20, f10` instruction has been executed. At this moment, the instruction has been fetched and decoded. Thus, the floating-point unit will be used to perform the addition.
- ❏ Press F7. The first of the two clock cycles required by the floating-point unit is executed. At the same time, the following addition is fetched and decoded.
- ❏ Press F7 one more time. The second clock cycle of the addition stage of the first instruction is carried out. At the same time, the following instruction goes into the first stage of the floating-point unit.

19

- ❏ Press F7 one more time. The execution of the first instruction which required the floating-point unit is finished. The second instruction goes into the second stage of the floating-point unit. Furthermore, the `EX` stage of the `nop` instruction is executed.
- ❏ In the following clock cycle both the `nop` and the second `add.d` instructions are ready to go into the `MEM` stage of the pipeline. Since this stage is not replicated, the `nop` is paused and the pipeline is stalled due to a structural hazard. Press F7 and check it out.
- ❏ Simulate the execution of the program until the end by pressing F4. Check that you can understand what is happening in the pipeline in each clock cycle. How many cycles has the CPU required to execute the program?[20] What is the speedup when comparing this result with that in an ideal execution without stalling the pipeline?[21]

20

21

7. Data hazards

Data hazards appear when several pipeline stages try to gain access to the same data item, such as a register or the result of an instruction.

```
.data

var:    .word 25
res:    .space 8

.code

main:
         daddi r10, r0, 5     ; r10 <- 5
         nop
         ld    r20, var(r0)   ; loar var in r20
         dadd  r1, r20, r10   ; r1 <- r20 + r10
         nop
         nop
         sd    r1, res(r0)      ; store r1 in res
         halt
```

- ❏ Edit the `2-2data.s` file, analyze its content and try to understand its meaning. It is a simple program which performs the addition of an integer number with a constant and stores the result in memory.
- ❏ If the pipeline was not stalled when executing this program, how many clock cycles would the CPU require to run it?[22]
- ❏ Load the file in the simulator.
- ❏ Be sure that the option `Enable Forwarding` in the `Configure` menu is disabled.

22

- ❏ Press F7 four times, until the clock cycle #5 of the first instruction has been finished.
- ❏ In the next clock cycle, EX stage of the `dadd r1, r20, r10` instruction, the numbers stored in `r20` and `r10` are required. However, while `r10` already contains the immediate value at the end of the clock cycle #5 (the value is written in the first half of the cycle of the WB stage), the pipeline must be stalled until the `r20` register loads its value. Thus, the pipeline is stalled due to a data hazard, and the simulator shows `RAW Stall`[3]. Press F7 to see this in the simulator.
- ❏ The value contained in `r20` after accessing the memory is known after finishing the WB stage. Thus the `dadd` instruction must wait until the load instruction is finished. Press F7 two more times to check this.
- ❏ Simulate the execution of the program until the end by pressing F4. Check that you can understand what is happening in the pipeline in each clock cycle. How many cycles has the CPU requiered to execute the program?[23] What is the speedup when comparing this result with that in an ideal execution without stalling the pipeline?[24]
- ❏ The value of `r20` after accessing the memory is already known after finishing the MEM stage, since the value from memory is already in the CPU. After finishing the WB stage the value is effectively written in `r20`. However, the MIPS64 architecture provides a functionality that enables the use of this value as an operand for an instruction even when it is not written in the destination register yet. This technique is called *forwarding* and its complexity is beyond the scope of this course. To check how this technique is performed, enable this option in the simulator and simulate the execution of the program again.

23

24

8. Exercises

✏ All the programs simulated contain `nop` instructions. This avoids additional hazards in the execution of the programs which may difficult their understanding. Try to remove these `nop` instructions and simulate the programs again. New hazard types will appear in the pipeline. Do the results obtained in these scenarios match the previous ones?

[3] `RAW` means *Read After Write* and it is a sub-type of the data hazards.