COMPUTER ORGANIZATION

Session 4: Pipelined Processor

1. Goals

- To observe how data and control hazards impact the performance of a pipelined processor, regardless of the solving technique applied, be it insertion of *nop* instructions in the software or insertion of stall cycles from the hardware.

2. Experiments: the DLX CPU

This lab is based on a pipelined processor, the DLX, which is very similar to the MIPS datapath studied in the classroom. You will simulate the execution of small chunks of DLX code and experiment with different hazard-solving techniques. Thus, you will be able to understand data and control hazards in pipelining better, how they impact performance, and their solving techniques.

The DLX assembly language is very close to MIPS', with slight syntax differences. The datapath is pipelined into the same five stages as the classroom model of the pipelined MIPS: instruction fetch, IF; instruction decode and read register file, ID; use of the ALU, EX; access to data memory, MEM; and the final write-back stage, WB.

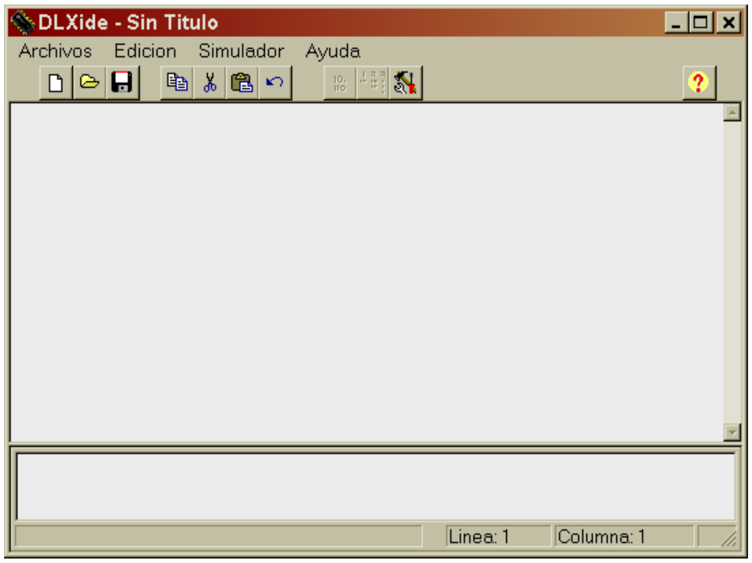
The differences between DLX and MIPS that affect the exercises in this lab are the following:

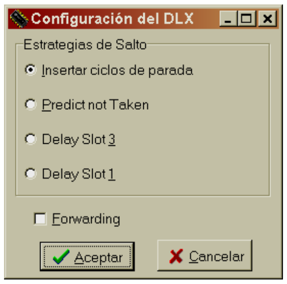
- In DLX, registers are named r1, r2... meaning the same as $1, $2... on a MIPS. The DLX register r0 is wired to zero, like $0 on a MIPS.
- Immediate values are denoted by the prefix "#". For example, the *add immediate* DLX instruction **addi r4, r1, #64** equals the MIPS instruction addi $4, $1, 64.
- In DLX, **seq r5, r4, r1** sets r5 to one if r4=r1; otherwise, r5 is set to zero (*set if equal*).
- The store instruction in DLX changes the order of the arguments compared to the MIPS syntax. For example, the MIPS sw $14, 0($3) is written **sw 0(r3), r14** on the DLX.
- A DLX program ends when the instruction **trap #0** is executed. It has the same effect as calling the exit system function on a MIPS.
- DLX comments start with the ";" character and take the rest of the line.

3. The DLXide Simulator

The DLXide simulator can be found in the lab folder in PoliformaT. The simulator is a single MS Windows executable file that does not require installation. DLXide can simulate the execution of DLX instructions cycle by cycle, showing their progress through the data path. There are separate instruction and data memories (Harvard architecture). Registers are written during the first half of the clock cycle and read during the second.

When you start DLXide, the program window with the available menus will be displayed. The following figure (left-hand side) shows the aspect of this window:




Configuración del DLX

Estrategias de Salto

[✓ Insertar ciclos de parada]

Predict not Taken

Delay Slot 3

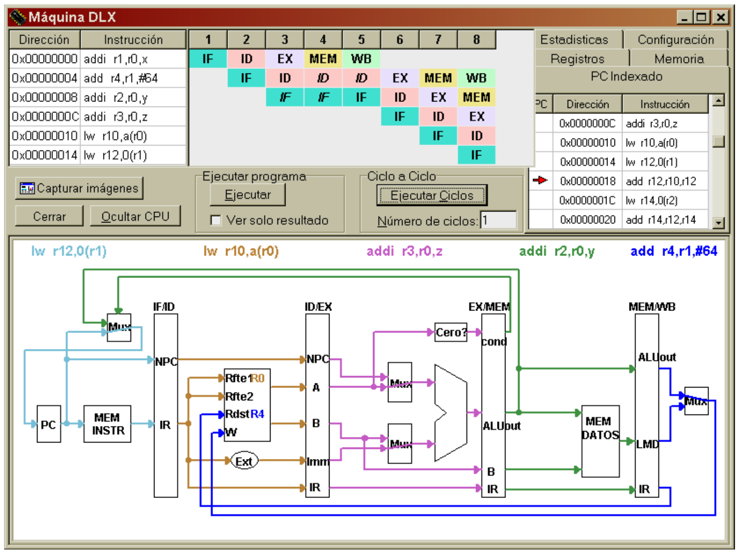
Delay Slot 1

☐ Forwarding

Enter the *Simulador* menu and the DLX Configuration option to configure the simulator. This will then open a dialogue to select one of the available strategies to solve data and control hazards. The default option is to insert stall cycles, as shown to the right. Make sure your configuration coincides.

Once configured, you can **load the file with the assembly program to simulate**. Use *Archivos*/*Abrir* and select the program file. After loading a program, it must be assembled (*Simulador* menu, *Ensamblar* option). In the event of errors, they will be displayed at the bottom of the program window. After correcting the errors, it must be reassembled. When the program is assembled without errors, it is stored in the simulated machine's memory, and the user is informed about it.

To **start a simulation**, use the *Simulador*/*Ejecutar* option. A new window will show the instruction-time diagram, the datapath, and a window with multiple tabs to allow you to inspect the CPU state. The following figure shows the simulation window after executing eight simulation steps of some program (eight clock cycles):


Máquina DLX

Dirección	Instrucción	1	2	3	4	5	6	7	8
0x00000000	addi r1,r0,x	IF	ID	EX	MEM	WB			
0x00000004	add r4,r1,#64		IF	ID	*ID*	*ID*	EX	MEM	WB
0x00000008	addi r2,r0,y			*IF*	*IF*	IF	ID	EX	MEM
0x0000000C	addi r3,r0,z						IF	ID	EX
0x00000010	lw r10,a(r0)							IF	ID
0x00000014	lw r12,0(r1)								IF

Ejecutar programa: ☐ Ver solo resultado

Ciclo a Ciclo: Número de ciclos: 1

PC Indexado

PC	Dirección	Instrucción
	0x0000000C	addi r3,r0,z
	0x00000010	lw r10,a(r0)
	0x00000014	lw r12,0(r1)
➜	0x00000018	add r12,r10,r12
	0x0000001C	lw r14,0(r2)
	0x00000020	add r14,r12,r14

The *Configuración* tab indicates the hazard-solving strategy currently selected. The *Registros* tab shows the contents of registers. The "value" fields in the registers tab are editable with a double-click. The numbering base is selectable via a secondary click. The *PC Indexado* and *Memoria tabs* allow you to inspect the instruction and data memory contents, respectively. Finally, the *Estadísticas* tab gives the number of cycles consumed, instructions executed, stall cycles inserted, and short circuits applied – a hardware technique to solve data hazards we are not exploring in ETC.

You can execute programs cycle by cycle, advance several cycles in one go (button *Ejecutar Ciclos*), or execute until a ***trap #0*** instruction (*Ejecutar*). At the end of every simulated clock cycle, DLXide updates the instruction-time diagram. When a stall cycle is inserted, the stages repeating instruction are shown in *italics* (e.g., in cycles 3, 4 and 5 of the image). The datapath scheme is also updated with every simulation step. Fetched instructions are given different colours. When no instruction is in one of the stages, the text "**–nop–**" is displayed. This must not be confused with the actual **nop** instruction, represented as "nop" (no dashes).

4. Solving data hazards using stall cycles

Let's start by running and observing the program in file *aritm1.s* below. The only purpose of this program is to exercise data hazards.

```
; aritm1.s - Various arithmetic operations
      .text
start:
1)    add  r1, r0, r0      ; r1 = 0
2)    addi r2, r0, #64     ; r2 = 64
3)    addi r3, r2, #10     ; r3 = r2 + 10 = 74
4)    sub  r4, r3, r2      ; r4 = r3 - r2 = 10
5)    trap #0              ; End of program
```

Exercise 1: Fill in the following table to identify the data hazards in this program.

	Conflicting register	number of instruction writing the register	number of instruction reading the register
Hazard 1	r2	2	3
Hazard	r3	3	4
Hazard	r2	2	4
Hazard			
…			

Exercise 2: Before using the simulator, fill in this table assuming stall cycles to solve data hazards.

Number of instructions executed	5
Number of stall cycles	4
Total number of cycles	13
CPI	9/5 = 1.8

Exercise 3: Load, assemble, and execute the code of *aritm1.s* step by step. Double-check that DLXide is configured to insert stall cycles and the *forwarding* option is disabled. Complete the instruction/cycle diagram.

Instruction\cycle	1	2	3	4	5	6	7	8	9	10	11	12	13

5. Solving data hazards with *nop* instructions

As you have just seen, data hazards affect the performance of a pipelined processor. During some cycles, instructions may be unable to progress and remain *stalled* if the insertion of stall cycles is in place. An alternative approach is to insert *nop* instructions so that conflicting instructions stay at a "safe" distance. You need to insert as many *nop* instructions as stall cycles were required in the previous exercise. This is in such a way that conflicting instructions are at a minimum distance of 3 instructions, i.e., there are at least two instructions in between any two instructions with data conflicts.

Exercise 4: Modify the code *aritm1.s*, generating a new file named *aritm1_nop.s*, which includes the appropriate *nop* instructions to eliminate all data hazards. Copy the resulting code below:

```
  .text


start:


  add  r1,r0,r0     ; r1 = 0


  addi r2,r0,#64    ; r2 = 64
  nop
  nop
  addi r3,r2,#10    ; r3 = r2 + 10 = 74
```

Exercise 5: Complete the following table assuming *aritm1_nop.s* was executed in a datapath where data hazards were solved by inserting *nop* instructions.

Number of instructions executed	9
Number of stall cycles	0
Total number of cycles	13
CPI	1 (as there are no stall cycles)

Exercise 6: Did you get a different execution time for the program? Elaborate your answer.

No, we have solved the same problem, but in another way.

6. Data hazards involving memory instructions

The previous program contained data hazards among arithmetic instructions. Data hazards can also involve sw (when we want to store data that has not yet been written in the register file) and lw (when an instruction requires a memory operand not yet written in a register). Consider the program below, available in file *mem.s*:

```
; mem.s - Multiple memory operations
        .data
A:      .word 0
B:      .word 20
C:      .word 30
D:      .word 0

        .text
start:
1)      addi r1, r0, #10    ; r1 = 10
2)      sw A(r0), r1        ; Store 10 to A
3)      lw r1, B(r0)        ; r1 = 20
4)      lw r2, C(r0)        ; r2 = 30
5)      add r3, r1, r2      ; r3 = r1 + r2 = 50
6)      sw D(r0), r3        ; Store 50 to D
7)      trap #0             ; End of program
```

Exercise 7: Load, assemble, and run this code in the DLXide simulator. Select stall cycles as the hazard-solving technique. Fill in the following table.

Number of instructions executed	7
Number of stall cycles	6
Total number of cycles	17
CPI	13/7 = 1.857

Exercise 8: Modify *mem.s* using *nop* instructions so that the program does not require the insertion of stall cycles. Save the resulting program as *mem_nop.s* and write the modified code here.

Exercise 9: Simulate the program *mem_nop.s* now and fill in the following results table.

Number of instructions executed	13
Number of stall cycles	0
Total number of cycles	17
CPI	1

7. Control Hazards

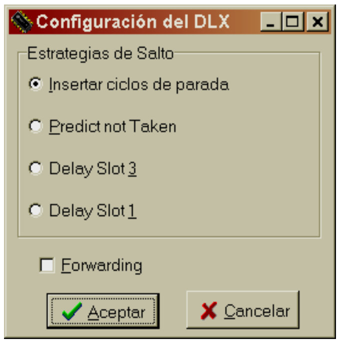
Control hazards are those caused by branch instructions. This is because branch instructions can modify the control flow of programs. The program in file *bucle1.s* operates with vectors. For all ten components of three vectors *A*, *B* and *C*, the program calculates *C[i] = 2*A[i] + B[i] + 1*. This is the code:

```
; bucle 1.s:  C[i] = 2*A[i] + B[i]+1
      .data
A:    .word  0, 1, 2, 3, 4, 5, 6, 7, 8, 9
B:    .word 10,11,12,13,14,15,16,17,18,19
C:    .space 40

      .text

1) start:   addi r1, r0, #10        ; r1 = number of iterations
2)          addi r2, r0, #0         ; r2 = offset to component of A
3)          addi r3, r0, #0         ; r3 = offset to component of B
4)          addi r4, r0, #0         ; r4 = offset to component of C

5) loop:    lw r6, A(r2)            ; read A[i]
6)          add r6, r6, r6          ; r6 = 2*A[i]
7)          lw r7, B(r3)            ; read B[i]
8)          addi r7, r7, #1         ; r7 = B[i]+1
9)          add r8, r6, r7          ; r8 = 2*A[i]+B[i]+1
10)         sw C(r4), r8            ; C[i] = r8
11)         addi r1,r1,#-1          ; r1 = r1 − 1   -> one less to process
12)         addi r2, r2, #4         ; r2 = r2 + 4   -> update offsets
13)         addi r3, r3, #4         ; r3 = r3 + 4
14)         addi r4, r4, #4         ; r4 = r4 + 4
15)         seq r5, r1, r0          ; r5 = (r1 == 0) -> check completion
16)         beqz r5, loop           ; branch if r5 == 0 (implies r1 ≠ 0)

17)         trap #0                 ; End of program
```

Exercise 10: Fill the table below with the **data** hazards in this program.

	conflicting register	Nr. of the instruction writing the register	Nr. of the instruction reading the register
Hazard 1	r6	5	6
Hazard	r7	7	8
Hazard	r7	8	9
Hazard	r8	9	10
Hazard	r5	15	16
Hazard			
Hazard			

You can now execute the code. Choose *Insertar ciclos de parada* (Insert Stall Cycles) in the simulator settings menu, as shown in the figure to the right. In this case, the simulator inserts three stall cycles for each control hazard (the branch delay slot is 3 for this processor). Choose the insertion of stall cycles to solve data hazards as well.



Estrategias de Salto: [✓ Insertar ciclos de parada] Predict not Taken / Delay Slot 3 / Delay Slot 1
Forwarding: not checked

Exercise 11: Load, assemble, and execute the program *bucle1.s* in the DLXide simulator. Obtain the results required to fill the following table:

Number of instructions executed	125
Number of stall cycles	130
Total number of cycles	259
CPI	255 / 125 = 2.04

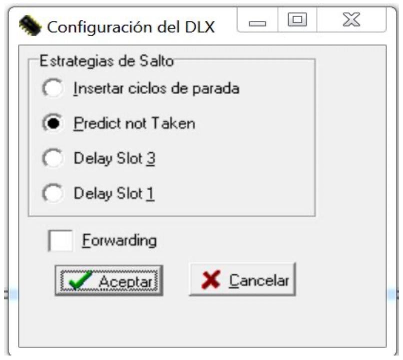
Exercise 12: Calculate how many stall cycles are inserted to solve data hazards and how many to solve control hazards. Run the code with the simulator DLXide cycle-by-cycle, identifying each stall cycle in every iteration. You should also be able to find this out analytically.

Total number of stall cycles for solving data hazards: 130 - 30 = 100 (30 comes from below)

Total number of stall cycles for solving control hazards: 10 * 3 = 30

8. Using branch prediction for solving control hazards

Control hazards have previously been resolved by inserting stall cycles, penalising performance. An alternative to solving them is the use of branch prediction techniques. The only prediction technique available in DLX is *Predict not Taken*. To use predict-not-taken, you must configure the simulator as follows:



Estrategias de Salto: Insertar ciclos de parada / [✓ Predict not Taken] / Delay Slot 3 / Delay Slot 1
Forwarding: not checked

Exercise 13: Select *predict-not-taken*, load, assemble, and execute the program *bucle1.s* in the DLXide simulator. Fill in the following table with the results:

Number of instructions executed	125
Number of stall cycles	127
Total number of cycles	256
CPI	252 / 125 = 2.016

Exercise 14: From these results, do you find this prediction technique efficient in this case? Why? Which alternative solution would have worked better?

It's better than inserting stall cycles, but it's not much more efficient.
In this case, it would have been better to assume that the jump was going to be performed instead of that it wasn't.

Extension exercises (optional) to learn more

1. Data hazard mitigation with code reordering

A practical design alternative to mitigate data hazards is code reordering. Instead of inserting *nop* instructions to separate two conflicting instructions, the reordering technique inserts instructions from the code itself, thus doing useful work and solving the hazard at the same time without incurring the penalty of increasing the number of instructions to be executed. Code reordering comes at no additional hardware cost since it is a software solution that relies on the compiler to detect hazards and find proper instructions. However, not all instructions can be reordered since they cannot come before the instructions that produce the data they consume or come after the instructions that consume the data they produce. Also, the ordering of instructions that write to the same register (or memory location) cannot be changed. In those cases, there is no other solution than inserting *nop* instructions or using a hardware technique (e.g., stall cycles). Reordering instructions to avoid hazards is usually the compiler's responsibility.

We will use code reordering with a new program located in file *aritm2.s*. The code is the following:

```
; Various arithmetic operations
; R5 = R4 – R3 + R2
; R6 = R4 + R1 – R3

      .text
start:
1)    addi r1, r0, #10  ; r1 = 10
2)    addi r2, r0, #20  ; r2 = 20
3)    addi r3, r0, #30  ; r3 = 30
4)    addi r4, r0, #40  ; r4 = 40
5)    sub r5, r4, r3    ; r5 = r4 – r3
6)    add r5, r5, r2    ; r5 = r5 + r2
7)    add r6, r4, r1    ; r6 = r4 + r1
8)    sub r6, r6, r3    ; r6 = r6 – r3
9)    trap #0           ; Fin del programa
```

The program performs two arithmetic operations, obtaining r5 and r6 from the values of registers r1 to r4.

Exercise 15: Load, assemble, and execute *aritm2.s* in the DLXide simulator. Obtain the following results:

Number of instructions executed	
Number of stall cycles	
Total number of cycles	
CPI	

Exercise 16: Modify the original file, generating a new program *aritm2_reord.s*, so that the execution time of the program is minimised, using code reordering. You can still use *nop* instructions if reordering cannot solve all data hazards. Write the resulting code below:

Ejercise 17: Load, assemble, and execute the program *aritm2_reord.s* in the DLXide simulator. Obtain the following results:

Number of instructions executed	
Number of stall cycles	
Total number of cycles	
CPI	

IMPORTANT! Once the code is executed, you should check that the execution has been correct; the final values of registers r5 and r6 must be 30 and 20, respectively.