

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, that is very similar to the MIPS datapath studied at the classroom. You will simulate the execution of small chunks of DLX code and experiment different hazard solving techniques. Thus, you will be able to better understand data and control hazards in pipelining, how they impact performance, and their solving techniques.

The DLX assembly language is very close to MIPS', with just slight differences in syntax. 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.

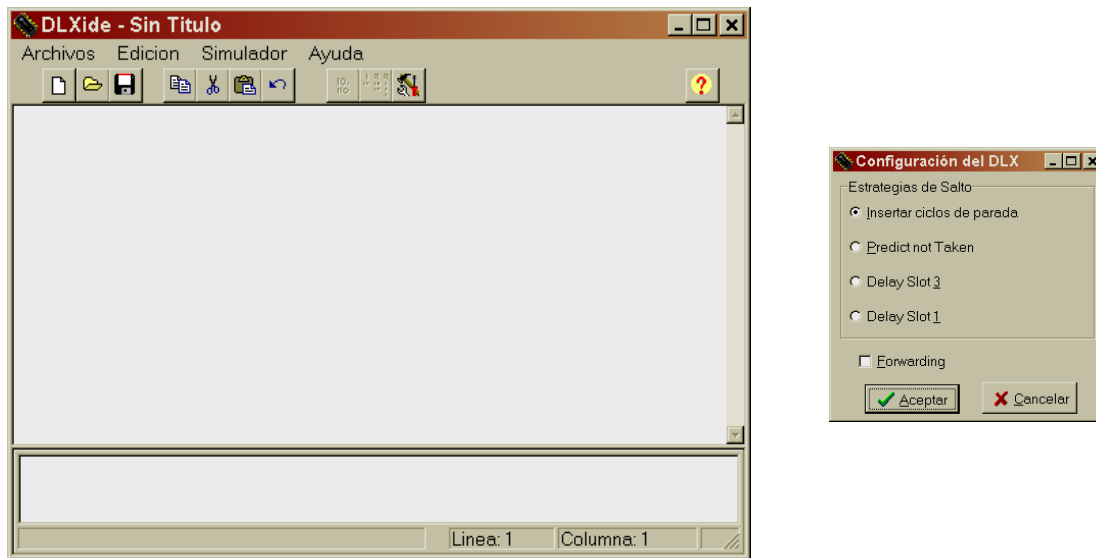
The differences between DLX and MIPS that affect the exercises in this lab are the following:

- In DLX, registers are named `r1, r2,...` meaning the same as `$1, $2, ...` on a MIPS. The DLX register `r0` is wired to zero, exactly like `$0` on a MIPS.
- Immediate values for arithmetic instructions are denoted by prefix `"#"`. For example, the DLX instruction **`addi r4, r1, # 64`** is equivalent to the MIPS `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 argument order with respect to the MIPS syntax. For example, the MIPS instruction `sw $14, 0($3)` must be 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, and it does not require installation. DLXide can simulate the execution of DLX instructions cycle by cycle, and it shows 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.

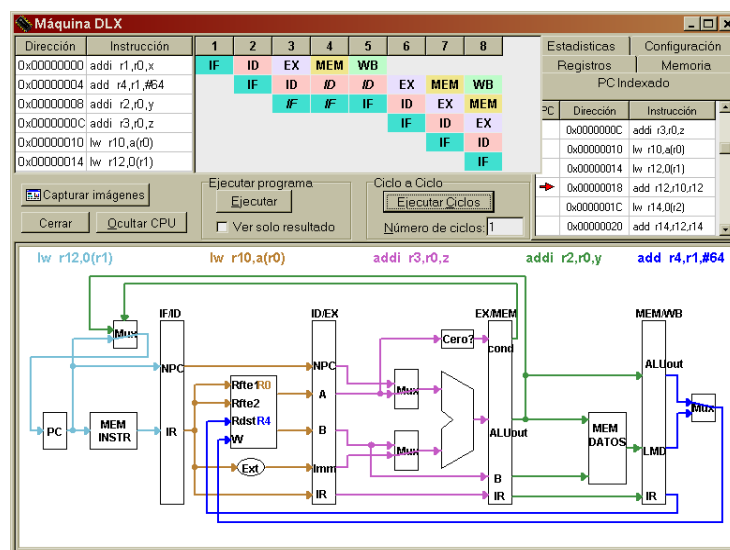
When you start DLXide, the program window will be displayed with the available menus. The following figure (left hand side) shows the aspect of this window:



To configure the simulator, enter the *Simulador* menu, DLX Configuration option. This will then open a dialog 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**. Access the *Archivos* menu, *Abrir* option, 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* menu, *Ejecutar* option. A new window will show the instruction-time diagram, the datapath, and a window with multiple tabs to allow you inspect the CPU state. The following figure shows the simulation window after executing eight simulation steps of some program (eight clock cycles):



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 secondary click. The tabs *PC Indexado* and *Memoria* allow you to inspect the contents of the instruction and data memory, 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 that we are not using in ETC.

You can execute programs cycle by cycle; or 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 there is no instruction in one of the phases, the text "**-nop-**" is displayed. Note that this must not be confused with the actual **nop** instruction, which is 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* shown below. The only purpose of this program is to serve us 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 identifying the data hazards that exist in this program.

	Conflicting register	number of instruction writing the register	number of instruction reading the register
Hazard 1			
Hazard			
Hazard			
Hazard			
...			

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

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

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

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 benefits of pipelining. During some cycles, instructions may be unable to make progress and remain *stalled* if insertion of stall cycles is in place. An alternative approach is to insert *nop* instructions so that conflicting instructions remain at a “safe” distance from each other. In this case, you will insert as many *nop* instructions as stall cycles have been occurred 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:

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	
Number of stall cycles	
Total number of cycles	
CPI	

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

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 to the register file) and `lw` (when an instruction requires a memory operand not yet written to 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	
Number of stall cycles	
Total number of cycles	
CPI	

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

7. Control Hazards

Control hazards are caused by branch instructions because they 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:

```
; 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 you find in this program.

	conflicting register	number of instruction writing the register	number of instruction reading the register
Hazard 1			
Hazard			
Hazard			
Hazard			
Hazard			
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. The simulator inserts 3 stall cycles for each control hazard in this case (the branch delay slot is 3 for this processor). Choose insertion of stall cycles for solving data hazards as well.



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	
Number of stall cycles	
Total number of cycles	
CPI	

Exercise 12: Calculate how many stall cycles are inserted to solve to data hazards and how many to solve control hazards. Run the code with the simulator DLXide cycle-by-cycle, identifying each of the stall cycles in every iteration.

Total number of stall cycles for solving data hazards:

Total number of stall cycles for solving control hazards:

8. Using branch prediction for solving control hazards

Control hazards have previously been resolved by inserting stall cycles, thereby penalizing performance. An alternative to solve 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:



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	
Number of stall cycles	
Total number of cycles	
CPI	

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

Extension exercises (optional) to learn more

1. Data hazard mitigation with code reordering

An effective design alternative to mitigate data hazards is code reordering. Instead of inserting always *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 to use a hardware technique (e.g., stall cycles). The task of reordering instructions to avoid hazards is usually the responsibility of the compiler.

We will try 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 code execution has been correct (the final values of registers r5 and r6 must be 30 and 20, respectively).