Session 4: Pipelined Processor

1. Goals

- To get familiar with the DLX Simulator for pipelined processor.
- To analyze the effect of data and control hazards on the performance of the pipelined datapath.

2. Experiments

In this session we will perform experiments on a pipelined processor very similar to the one studied in lessons. Mainly, we are going to execute small pieces of code on the pipelined processor by enabling different techniques for solving data hazards. This will allow us to better understand this problem and the different alternatives for mitigate it, such as insertion of stall cycles, insertion of NOP instructions or the use forwarding techniques[1]. We will also discuss in this session the control hazards generated by conditional or unconditional jump instructions. We will use some of the available resolution techniques for these hazards, such as branch prediction.

In the pipelined processor simulator (named **DLXide**) we will use the DLX assembler language (very similar to MIPS assembler, but with slight syntax differences). The datapath for DLX consists of five stages that correspond to the instruction fetch, decoding and searching the operands in the register file, execution in the UAL, access to data memory for both read and write, and writing back the result in the destination register. As result of a simulation we will obtain the execution time of a program on the pipelined processor measured in cycles, as well as the number of executed instructions, the number of stall cycles of the processor and the CPI index. In all cases the CPI is obtained by subtracting 4 (number of stages minus one) to the number of total execution cycles.

The differences between DLX and MIPS are the following:

- Registers are named r1, r2, ... in DLX ($1, $2, ... in MIPS). Register r0 is wired to 0.
- Immediate values for the arithmetic instructions are expressed using the prefix "#". For example, **`addi r4, r1, # 64`** in DLX is similar to `addi $4, $1, 64` in MIPS. In addition you can use a label (which will logically have a value assigned) to specify an immediate value. For example, if x = 0 then **`addi r1, r0, x`** (stores value 0 in r1).

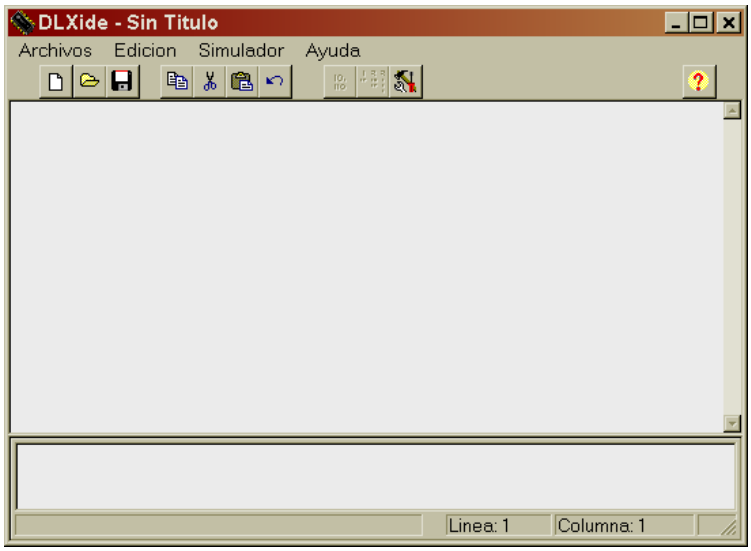
[1] Forwarding Techniques will be studied next year in Computer Architecture

- DLX instruction **`seq r5, r4, r1`** (set if equal) stores 1 in r5 if r4 = r1; otherwise write a zero in r5.
- DLX instruction **`beqz r5, loop`** (branch on equal to zero) jumps to the loop if and only if the register r5 is 0.
- The store instruction in DLX changes the order of the parameters with respect to the MIPS syntax. For example, the MIPS instruction **`sw $14, 0($3)`** must be written sw 0(r3), r14 in the DLX.
- Finally, the DLX program ends when the trap # 0 instruction is executed.

3. DLXide Simulator

This simulator is able to simulate cycle by cycle the execution of instructions of a DLX program. It supports all DLX instructions that operate with the integer register file. There is separate instruction and data memory (Harvard architecture). Registers are written and read in the first and second part of the clock cycle, respectively.

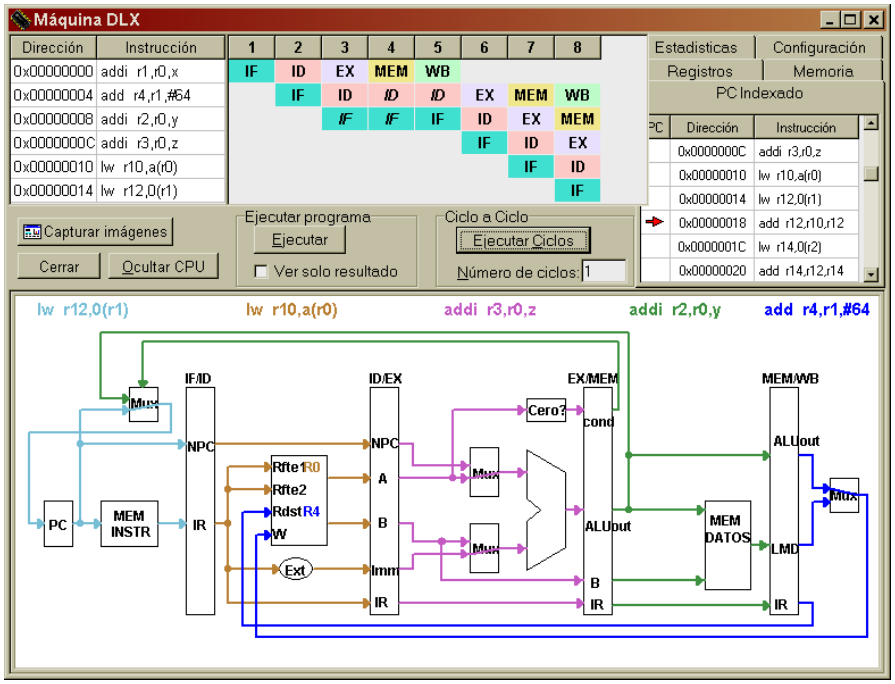
When DLXide is started a window is displayed showing the available menus. Next figure shows the aspect of this window.



To configure the simulator, you have to choice the option *Simulador* in the menu, and after that the *Configuración del DLX* option. Then, the program opens a dialog showing the strategies available to solve the data and control hazards. By default, stall cycles are inserted (first choice in the DLX Settings menu options). Therefore, the configuration should be as shown below:

In a next step you have to load the program file to be executed. To do this, select the Archivos (Files) menu, and the option *Abrir* (Open), and choose the appropriate file. After loading a program, it must be assembled (*Simulador* menu, *Ensamblar* (Assemble) option). In case any errors appear, they would be displayed at the bottom of the program window. After correcting the errors you have to reassemble the program. When the program is successfully assembled, it is stored in the memory of the simulated machine and the user is informed.

To start the simulation, you have to select the *Simulador* menu, option Ejecutar (Execute). A new window will appear in which the diagram instructions-time, the datapath, as well as a window with multiple tabs will be displayed allowing to examine the state of the machine. Next figure shows the simulation window after the execution of 8 simulation steps:



The *Configuracion* (Settings) tab allows you to remember which strategies are selected to solve the hazards. *Registros* (Register) tab allows you to examine the contents of the general purpose registers. By double clicking on the *valor* (value) field you can change the value of a register. By pressing the right button you can change the number base. The *PC indexado* (Indexed PC) and *Memoria* (Memory) tabs allow you to see the contents of the instruction and data memory, respectively. Finally, the *Estadísticas* (Statistics) tab indicates the number of clock cycles used, the number of instructions executed, the number of stall cycles needed and the bypasses applied.

The simulator allows the execution of the program cycle by cycle or advance several cycles with the option *Ejecutar Ciclos* (Execute Cycles). It also permits to execute the entire program completely until an instruction trap #0 is reached (Option *Ejecutar* –Execute-). The instruction-time diagram is updated after each clock cycle. When a stall cycle is inserted, the phases in which the same instructions are maintained are shown in italics. It also updates the data path of the machine. Each instruction is shown using a different color. When there is no instruction in one of the stages, the text -nop- is displayed. This should not be confused with an instruction nop.

4. Solving Data hazards using stall cycles

Let's start by running the program. **`aritm1.s`** shown below:

```
; Several 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
```

As appreciated, this piece of code does not perform any useful global operation, just initialize some registers but, it is a good exercise to analyze data hazards.

Exercise 1: Complete the following table identifying the data hazards that exist in the previous program causing conflict in the data path.

	Register	number of instruction writing the register	number of instruction reading the register
Data hazard num.			
Data hazard num.			
Data hazard num.			
Data hazard num.			
…			

Exercise 2: Complete the following table assuming the use of 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. Remember that the configuration of the DLX simulator must be the previous one (inserting stall cycles for solving data hazards, 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. Inserting nop instructions to solve data hazards

As it can be seen, data hazards negatively affect system performance. Specifically, during some cycles the instructions are blocked increasing the execution time. An alternative is the introduction of *nop* instructions, which allow instructions to maintain a "safe" distance so that data hazard does not become a conflict. You must enter as many instructions nop as stall cycles needed so, the instructions with data hazards maintain a minimum distance of 3 instructions.

Exercise 4: Modify the code **`aritm1.s`**, generating a new file named **`aritm1_nop.s`**, which includes the appropriate *nop* instructions to not generate any data hazard. Copy the resulting code below:

Exercise 5: Complete the following table assuming **`aritm1_nop.s`** code execution in a data path where data hazards are 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 runtime? Give a reasoning answer

6. Data hazards in memory operations

In previous sections we have been dealing with data hazards in arithmetic and logic operations. But, data hazards can also appear in memory operations because the program wants to write a data in memory that has not yet been written in the register file. A data hazard also is generated when the program has to read a value that has not been yet obtained from data memory (produced by a load instruction).

In this last section, we deal with this type of data hazards. To this end, the program **`mem.s`** shown below , is considered:

```
; Several 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        ; stores10 in 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        ; stores 50 in D
7)      trap #0             ; end
```

Exercise 7: Load, assemble, and execute the previous piece of code in the DLXide simulator. Get the results and complete the following table.

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

Exercise 8:

Modify the program **`mem.s`** generating **`mem_nop.s`**, in such a way that the stall cycles are eliminated using *nop* instructions. Write below the resulting code.

Exercise 9: Simulate now the program **`mem_nop.s`** and complete the table below with the obtained results

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

7. Control Hazards

Control hazards are caused by branch instructions because they can modify the control flow of programs. The program named **`bucle1.s`** operates with vectors (C[i] = 2*A[i] + B[i] + 1). Below you can see the code of this program:

```
; 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
start:
1)      addi r1, r0, #10        ; r1 = no. of iterations
2)      addi r2, r0, #0         ; r2 = index for  vector A
3)      addi r3, r0, #0         ; r3 = index for  vecto B
4)      addi r4, r0, #0         ; r4 = index for  vector C
bucle:
5)      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
12)     addi r2, r2, #4         ; r2 = r2 + 4
13)     addi r3, r3, #4         ; r3 = r3 + 4
14)     addi r4, r4, #4         ; r4 = r4 + 4
15)     seq r5, r1, r0          ; r5 = (r1 == 0)
16)     beqz r5, bucle          ; branch if r5 == 0
17)     trap #0                 ; End program
```

Exercise 10: Complete the table below describing data hazards causing a hardware conflict in the datapath.

	Register	number of instruction writing the register	number of instruction reading the register
Hazard no.			
Hazard no.			
Hazard no.			
Hazard no.			
Hazard no.			
Hazard no.			
Hazard no.			

Let´s execute the code. In the simulator settings the option you must choose to solve control hazards is *Insertar ciclos de parada* (Insert Stall Cycles) as shown in the figure below. The simulator inserts 3 cycles in this case (Delay slot = 3). For solving data hazards also choose the option Insert Stall Cycles.

Exercise 11: Load, assemble, and execute the program **`bucle1.s`** in the DLXide simulator. Get the results and complete the following table.

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

Exercise 12: Calculate how many stall cycles are due to data conflicts and how many to 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. The use of branch prediction for solving control hazards

In the previous exercises control hazards were solved by inserting stall cycles, thus penalizing system performance. An alternative to solve them is the use of branch prediction techniques. The only prediction technique available in DLX is *Predict not Taken.*

Let´s simulate the program **`bucle1.s`** using this technique. To this end, select the option *Predict not Taken* in the simulator settings as the figure shows:

Exercise 13: Load, assemble, and execute the program **bucle1.s** in the DLXide simulator. Get the results and complete the following table.

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

Exercise 14: Examine if all control hazards have been eliminated. Is this technique efficient in this case? Why? Which alternative solution could have been used?

Extension exercises (optional). To learn more

1. Data hazards mitigate by co reordering

An effective design alternative to mitigate data hazards, is code reordering. Instead of including instructions *nop* between the two instructions that generate a hazard the reordering technique inserts instructions belonging to the program. Thus, the penalty due to the increase of unproductive instructions is avoided.

Also, code reordering is an alternative that does not need any extra hardware. However, not all instructions in the code can be reordered because the final result cannot obviously change. When reordering is not possible then there is no other solution than inserting *nop* instructions, or using any hardware techniques (stall cycles or forwarding[2] techniques) in case the processor supports them. The task of reordering code to avoid hazards is usually made by the compiler.

Let´s practice reordering technique with the program **`aritm2.s`**.

Below you can see the code for this program:

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

        .text
```

[2] Forwarding Techniques will be studied in third year in the subject Computer Architecture

```
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             ; End
```

The program executes two arithmetic operations. Operators are in registers from r1 to r2. Results are stored in r5 and r6.

Exercise 15: Load, assemble, and execute the program **`aritm2.s`** in the DLXide simulator. Get the results and complete the following table.

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`**, in such a way that, using the reordering of the code, the minimum execution time is obtained. You can 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. Get the results and complete the following table.

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).