3^{rd} Laboratory: "STATIC INSTRUCTIONS SCHEDULING"

Computer Architecture and Engineering (3^{rd} year)
E.T.S. de Ingeniería Informática (ETSINF)
Dpto. de Informática de Sistemas y Computadores (DISCA)

Goal:

- Know, understand and apply some static instructions scheduling techniques.

Assignment:

The simulator of the MIPS computer with multicycle instructions

The simulator **mips-m** enables the execution of programs written using the MIPS assembler. It support a subset of the MIPS instruction set, including integer and floating-point instructions. The set of supported instructions is detailed in the annex of this lab.

The simulated processor does not integrate dynamic instruction scheduling. Solving data hazards is performed by inserting stalls or applying the forwarding technique, with stall insertion whenever necessary. Control hazards can be solved inserting stall, with *predict-not-taken* or using the delayed branch technique, with a *delay-slot* of one, two or three instructions. For the execution of multicycle instructions it is available a load/store unit, a multiplication operator, an adder and a comparison operator. All of them are pipelined and their latency can be configured.

The simulator executes using the following command line:

```
mips-m -s <results> -d <data-hazards> -c <control-hazards>
       -l <lat> -a <lat> -m <lat> -k <lat> -f <file.s>
```

where:

- <results> conditions how the result of the simulation will be shown. There are two options:
  - **time**: Shows in the terminal the execution. time.
  - **final**: Shows in the terminal the execution time, registers and memory content after the execution of the program.
  - **html**(*): Generates several html files with the state of the execution step by step and the final results. These results can be seen by opening the file **index.html** in a web browser. This is the default option.
- <data-hazards> Signals how data hazards are solved. There exist three options:
  - **n**: There is no logic for solving data hazards.

  - **p**: Data hazards are solved by inserting stalls.
  - **c**(*): Data hazards are solved using shorcircuits, and inserting also the necessary stalls. This is the default option.

- <control hazards> Signal how control hazards are solved. There exist three options: cómo se resuelven los riesgos de control. Hay tres opciones:

  - **p**: They are solved by inserting stalls.
  - **t**(*): They are solved using the *predict-not-taken* technique. This is the default option.
  - **3**: They are solved using the delayed branch technique, with a *delay-slot*=3.
  - **2**: They are solved using the delayed branch technique, with a *delay-slot*=2.
  - **1**: They are solved using the delayed branch technique, with a *delay-slot*=1.

- <lat> indicates the latency of the multicycle floating-point pipelined operator. It must be of at least 2 cycles:

  - *l*: Load/store operator. By default, 2 cycles.
  - *a*: Add/sub operator. By default, 4 cycles.
  - *m*: Mult/div operator. By default, 7 cycles.
  - *k*: Comparison operator. By default, 4 cycles.

- <file.s> is the name of the file containing the assembler code to execute.

Example of a MIPS program

Find following the assembler code of a loop adding an escalar value to an array stored in memory ($\vec{Z} = a + \vec{Y}$, bucle DAPY).

```
start:
        dadd r1,r0,y     ; r1 contains address y
        dadd r2,r0,z     ; r2 contains address z
        l.d f0,a(r0)     ; f0 contains a
        dadd r3,r1,#512  ; 64 elements are 512 bytes
loop:
        l.d f2,0(r1)
        add.d f4,f0,f2
        s.d f4, 0(r2)
        dadd r1,r1,#8
        dadd r2,r2,#8
        dsub r4,r3,r1
        bnez r4,loop
        nop             ; delay slot, if necessary

        trap #0         ; Program end
```

This program is stored in file `dapy.s`. It can be executed, showing its execution results in html files and solving data and control hazards through short-circuits and *predict-not-taken* respectively, using the following command:

```
mips-m -s html -d c -c t -f dapy.s
```

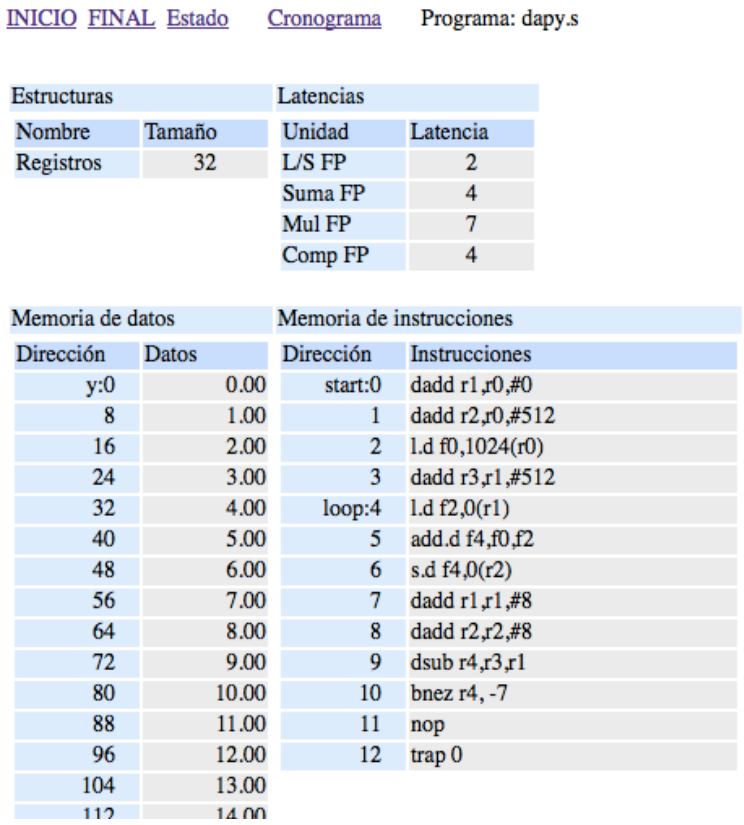
Then, opening file index.html using a web browser will show the configuration of the processor, the initial memory content and also several links enabling a navigation through results:

- INICIO. Shows the processor configuration and the initial memory content.
- FINAL. Shows performance results after executing programs, the processor configuration and the final content of memory. Checking the final memory content enables verifying the proper execution of the program.
- Estado. Shows the state of the execution unit in a given cycle, indicating which instruction is hold by each processor stage. Events and control signal that are activated in the processor (hazards, short-circuits) are also shown. Finally, it is also provided the content of registers and memory at the end of the analyzed cycle. In this page one can find links to the state pages corresponding to 1 or 5 cycles before or after the current one. The page also shows the instructions–time diagram.
- Cronograma. Shows the instructions–time diagram of the program execution. The last cycle shown corresponds to the current cycle. In this page one can find links corresponding to diagrams located at 1 or 5 cycles before or after the current one. The page also shows the state of the processor.

Figure 1 shows the content of the generated *index.html* file. It shows the size of the register file and the latencies related to the multicycle units. First, the size of the register file and the latencies of the multicycle units are shown. It is also reported the initial content of the data and instructions memory zone.

Following the link FINAL will open file *final.html*. Figure 2 shows its content in our example. First, the execution performance results are reported: execution time, executed instructions, CPI, floating point operations and floating point operations per cycle. Then, the configuration of the processor is shown and it is reported the final content of the data and instructions memory. In the right side of the figure one can see that, the resulting array starts at memory location z, thus enabling the verification of the program correctness.

If the link Estado is followed, file *estadoXXX.html* will be opened, where XXX represents the execution cycle, starting at "001". Figure 3 shows its content for cycle 23 of our example. First, they are shown the links to pages *index.html* and *final.html*. Links to files representing the state of the computer 5 cycles before ([-5]), the cycle before ([-1]), the following cycle ([+1]) and 5 cycles after ([+5]) are also available. It is also shown a link to the instructions–time diagram until the current cycle (Crono). Then, execution unit stages are shown, indicating which instruction is in each stage. Empty stages held the equivalent to a hop instruction (-nop-). Since integer and floating point registers files are separated, it is possible to have until one integer and one floating point instruction in the WB stage. Control signals activated when a hazard is detected are also shown. The short-circuits



INICIO FINAL Estado Cronograma Programa: dapy.s

Estructuras	
Nombre	Tamaño
Registros	32

Latencias	
Unidad	Latencia
L/S FP	2
Suma FP	4
Mul FP	7
Comp FP	4

Memoria de datos	
Dirección	Datos
y:0	0.00
8	1.00
16	2.00
24	3.00
32	4.00
40	5.00
48	6.00
56	7.00
64	8.00
72	9.00
80	10.00
88	11.00
96	12.00
104	13.00
112	14.00

Memoria de instrucciones	
Dirección	Instrucciones
start:0	dadd r1,r0,#0
1	dadd r2,r0,#512
2	l.d f0,1024(r0)
3	dadd r3,r1,#512
loop:4	l.d f2,0(r1)
5	add.d f4,f0,f2
6	s.d f4,0(r2)
7	dadd r1,r1,#8
8	dadd r2,r2,#8
9	dsub r4,r3,r1
10	bnez r4, -7
11	nop
12	trap 0

Figura 1: Content of file *index.html*

applied also does. Then, one can see the content of integer registers (R0 to R31), floating point registers (F0 to F31) and the floating point state register (FPSR). Finally, the content of the data memory is shown. Files containing the state of the processor enable a step by step execution of the program. This is how the assembler code that written by anyone can be debugged.

Following the link Cronograma will open the file *cronoXXX.html*, where XXX represent the execution cycle, starting at "001". Figure 4 shows its content for cycle 23 in our example. As can be seen, it is shown the instructions–time diagram corresponding to the execution of the program until the considered cycle. One can also move to one or five cycles before and after the current one. In addition, the state of the current cycle (link Estado) can be analyzed or one can directly go to the state corresponding to one of the cycles (using the links in the upper part of the instructions–time diagram).

After executing the program, check that it has been stored in the address labelled as `z` a 64 data array with the expected content. Note the execution time of the program and the resulting CPI.

Program modification using static instructions scheduling

1. *Loop unrolling*

   Basically, the *loop unrolling* technique replicates the base code of a loop several times, decreasing the number of resulting iterations.

   In our example, since the maximum number of stalls required to solve the RAW

INICIO FINAL Estado Cronograma Programa: dapy.s

Ciclos	Instrucciones	Ins. Enteras	Ins. Multiciclo	CPI	Op. CF	Op. CF/ciclo
903	390	261	129	2.32	64	0.07

Estructuras	
Nombre	Tamaño
Registros	32

Latencias	
Unidad	Latencia
L/S FP	2
Suma FP	4
Mul FP	7
Comp FP	4

Memoria de datos	
Dirección	Datos
y:0	0.00
8	1.00
16	2.00
24	3.00
32	4.00
40	5.00
48	6.00
56	7.00
64	8.00
72	9.00
80	10.00
88	11.00
96	12.00
104	13.00
112	14.00

Memoria de instrucciones	
Dirección	Instrucciones
start:0	dadd r1,r0,#0
1	dadd r2,r0,#512
2	l.d f0,1024(r0)
3	dadd r3,r1,#512
loop:4	l.d f2,0(r1)
5	add.d f4,f0,f2
6	s.d f4,0(r2)
7	dadd r1,r1,#8
8	dadd r2,r2,#8
9	dsub r4,r3,r1
10	bnez r4, -7
11	nop
12	trap 0

504	63.00
z:512	1.00
520	2.00
528	3.00
536	4.00
544	5.00
552	6.00
560	7.00
568	8.00
576	9.00
584	10.00
592	11.00
600	12.00
608	13.00
616	14.00
624	15.00
632	16.00
640	17.00
648	18.00
656	19.00
664	20.00
672	21.00
680	22.00

Figura 2: Content of file *final.html*

hazard is 3 cycles, the code of the loop $\vec{Z} = a + \vec{Y}$ must be replicated 4 times, as shown hereafter. It must be noted that some registers have been renamed to delete name dependences:

```
start:
        dadd r1,r0,y      ; r1 contains the address of y
        dadd r2,r0,z      ; r2 contains the address of z
        l.d f0,a(r0)      ; f0 contains a
        dadd r3,r1,#512   ; 64 elements are 512 bytes
loop:
        l.d f2,0(r1)
        add.d f4,f0,f2
        s.d f4,0(r2)
        l.d f6,8(r1)
        add.d f8,f0,f6
        s.d f8,8(r2)
        l.d f10,16(r1)
        add.d f12,f0,f10
        s.d f12,16(r2)
        l.d f14,24(r1)
```

INICIO FINAL [-5] [-1] [+1] [+5] Estado Crono Programa: dapy.s **Ciclo: 23**

	--/IF	IF/ID	ID/EX	EX/MEM	MEM/WB				
Instruc.	dadd r1,r1,#8	s.d f4,0(r2)	-nop-	-nop-	-nop-				
			L/S1	L/S2	FP/WB				
L/S (FP)			-nop-	-nop-	l.d f2,0(r1)				
			A1	A2	A3	A4	FP/WB		
Sum (FP)			add.d f4,f0,f2	-nop-	-nop-	-nop-	-nop-		
			C1	C2	C3	C4	CMP/WB		
Cmp (FP)			-nop-	-nop-	-nop-	-nop-	-nop-		
			M1	M2	M3	M4	M5	M6	M7	FP/WB
Mul (FP)			-nop-	-nop-	-nop-	-nop-	-nop-	-nop-	-nop-	-nop-

Señales	IFstall	IDstall	RAW
		WBFaA1	

Estado al final del ciclo

Registros

	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15
valor	0	8	520	512	504	0	0	0	0	0	0	0	0	0	0	0
	R16	R17	R18	R19	R20	R21	R22	R23	R24	R25	R26	R27	R28	R29	R30	R31
valor	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
valor	1.000000	0.000000	1.000000	0.000000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
	F16	F17	F18	F19	F20	F21	F22	F23	F24	F25	F26	F27	F28	F29	F30	F31
valor	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
	FPSR															
valor	0															

Figura 3: Content of file *estado023.html*

```
        add.d f16,f0,f14
        s.d f16,24(r2)
        dadd r1,r1,#32
        dadd r2,r2,#32
        dsub r4,r3,r1
        bnez r4,loop
        nop             ; delay slot, if necessary

        trap #0         ; Program end
```

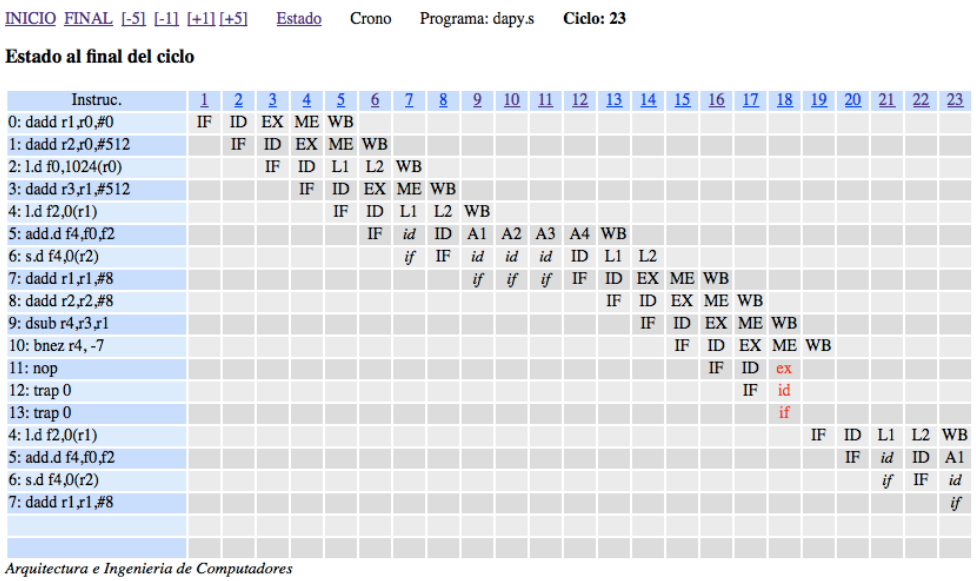
This program is stored in file `dapyu1.s`. Execute this new program:

```
mips-m -s html -d c -c t -f dapyu1.s
```

Check the correctness of the result and note the execution time. Calculate the resulting CPI. Quantify the speedup with respect to the original program.

That code can be easily modified to eliminate all data hazards, and fill the *delay slot* with a useful instruction:

```
start:
        dadd r1,r0,y      ; r1 contains the address of y
        dadd r2,r0,z      ; r2  contains the address of z
        l.d f0,a(r0)      ; f0 contains a
        dadd r3,r1,#512   ; 64 elements are 512 bytes
```



INICIO FINAL [-5] [-1] [+1] [+5] Estado Crono Programa: dapy.s **Ciclo: 23**

Estado al final del ciclo

Instruc.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0: dadd r1,r0,#0	IF	ID	EX	ME	WB																		
1: dadd r2,r0,#512		IF	ID	EX	ME	WB																	
2: l.d f0,1024(r0)			IF	ID	L1	L2	WB																
3: dadd r3,r1,#512				IF	ID	EX	ME	WB															
4: l.d f2,0(r1)					IF	ID	L1	L2	WB														
5: add.d f4,f0,f2						IF	*id*	ID	A1	A2	A3	A4	WB										
6: s.d f4,0(r2)							*if*	IF	*id*	*id*	*id*	ID	L1	L2									
7: dadd r1,r1,#8									*if*	*if*	*if*	IF	ID	EX	ME	WB							
8: dadd r2,r2,#8													IF	ID	EX	ME	WB						
9: dsub r4,r3,r1														IF	ID	EX	ME	WB					
10: bnez r4, -7															IF	ID	EX	ME	WB				
11: nop																IF	ID	ex					
12: trap 0																	IF	id					
13: trap 0																		if					
4: l.d f2,0(r1)																			IF	ID	L1	L2	WB
5: add.d f4,f0,f2																				IF	*id*	ID	A1
6: s.d f4,0(r2)																					*if*	IF	*id*
7: dadd r1,r1,#8																							*if*

Arquitectura e Ingenieria de Computadores

Figura 4: Content of file *crono023.html*

```
loop:
        l.d f2,0(r1)
        l.d f6,8(r1)
        l.d f10,16(r1)
        l.d f14,24(r1)
        add.d f4,f0,f2
        add.d f8,f0,f6
        add.d f12,f0,f10
        add.d f16,f0,f14
        s.d f4,0(r2)
        s.d f8,8(r2)
        s.d f12,16(r2)
        s.d f16,24(r2)
        dadd r1,r1,#32
        dsub r4,r3,r1
        bnez r4,loop
        dadd r2,r2,#32

        trap #0         ; Program end
```

This program is stored in file `dapyu.s`. Execute the program considering that control hazards are managed using the delayed branch technique with a *branch delay slot* of 1 instruction:

```
mips-m -s html -d c -c 1 -f dapyu.s
```

Check the correctness of the result and note the execution time. Calculate the resulting CPI. Quantify the speedup with respect to the original program.

2. *Software pipelining.*

   Basically, *software pipelining* replaces the original loop by another one where executed instructions belong to different iterations of the original loop, which eliminates hazards.

   The modified version of the loop $\vec{Z} = a + \vec{Y}$ is the following one:

```
start:
        dadd r1,r0,y     ; r1 contains the address of y
        dadd r2,r0,z     ; r2 contains the address of z
        l.d f0,a(r0)     ; f0 contains a
        dadd r3,r1,#512  ; 64 elements are 512 bytes
prepara:
        l.d f2,0(r1)
        add.d f4,f0,f2
        l.d f2,8(r1)
        dadd r1,r1,#16

loop:
        s.d f4, 0(r2)
        add.d f4,f0,f2
        l.d f2,0(r1)
        dadd r1,r1,#8
        dsub r4,r3,r1
        bnez r4,loop
        dadd r2,r2,#8

resto:
        s.d f4, 0(r2)
        add.d f4,f0,f2
        s.d f4, 8(r2)

        trap #0         ; Program end
```

   This program is stored in file `dapysp.s`. Execute the following command:

```
mips-m -s html -d c -c 1 -f dapysp.s
```

   Check the correctness of the result and note the execution time. Calculate the resulting CPI. Quantify the speedup with respect to the original program.

Development of a new program

In this section of the lab we will assume that latencies for the adder and the multiplier are of 2 and 4 cycles respectively (options `-a 2 -m 4` for running the simulator).

1. Write the conventional MIPS code for the execution of the operation $\vec{Z} = a * \vec{X} + \vec{Y}$ (DAXPY loop), being the size of arrays to process of 64 floating point numbers.

Take as reference the program stored in file `daxpy.s`.

Execute the program in the simulator. Do not forget to signal the simulator the method to solve hazards (options `-d c -c 1`) and the operator latencies. Evaluate the resulting performance.

2. Apply the *loop unrolling* technique to the developed code, reorganizing the code, when necessary in order to reduce the inserted stalls.

   Use as reference the program developed in section a), by copying it previously to another file (such as `daxpyu.s`). Write the new code and execute it. Do not forget to signal the simulator the method to solve hazards and the operator latencies. Evaluate the resulting performance and compare it with the base version.

   Have been stalls eliminated? If this is not the case, explain the causes of this situation.

Subset of MIPS instructions supported by the simulator

- Load/store

ld Rx, desp(Ry)
sd Rz, desp(Ry)

- Arithmetic, logic and shift

dadd Rx, Ry, Rz	daddi Rx, Ry, Imm
dsub Rx, Ry, Rz	dsubi Rx, Ry, Imm
and Rx, Ry, Rz	andi Rx, Ry, Imm
or Rx, Ry, Rz	ori Rx, Ry, Imm
xor Rx, Ry, Rz	xori Rx, Ry, Imm
dsra Rx, Ry, Rz	dsra Rx, Ry, Imm
dsll Rx, Ry, Rz	dsll Rx, Ry, Imm
dsrl Rx, Ry, Rz	dsrl Rx, Ry, Imm

- Comparison:

seq Rx, Ry, Rz	seq Rx, Ry, Imm
sne Rx, Ry, Rz	sne Rx, Ry, Imm
sgt Rx, Ry, Rz	sgt Rx, Ry, Imm
slt Rx, Ry, Rz	slt Rx, Ry, Imm
sge Rx, Ry, Rz	sge Rx, Ry, Imm
sle Rx, Ry, Rz	sle Rx, Ry, Imm

- Conditional branch

bnez Ry, Desp	bc1t Desp
beqz Ry, Desp	bc1f Desp

- Floating point load/store

l.d Fx, desp(Ry)
s.d Fz, desp(Ry)

- Floating point arithmetic

add.d Fx, Fy, Fz
sub.d Fx, Fy, Fz
mul.d Fx, Fy, Fz
div.d Fx, Fy, Fz

- Floating point comparison

c.eq.d Fy, Fz
c.ne.d Fy, Fz
c.lt.d Fy, Fz
c.le.d Fy, Fz
c.gt.d Fy, Fz
c.ge.d Fy, Fz

- Others

nop
trap