Col·lecció de Problemes

Estructura de Computadors

Montse Fernández
David López
Joan Manuel Parcerisa
Angel Toribio
Rubèn Tous
Jordi Tubella

Departament d'Arquitectura de Computadors
Facultat d'Informàtica de Barcelona
Quadrimestre de Primavera - Curs 2014/15

Licencia Creative Commons

Temari

1. Introducció

    1.1. Descripció jeràrquica del computador
    1.2. Mesures de rendiment
    1.3. Mesures de consum
    1.4. Llei d'Amdahl

2. Instruccions i tipus de dades bàsics

    2.1. Introducció a la MIPS ISA
    2.2. Operands en registre
    2.3. Operands en memòria
    2.4. Constants i immediats
    2.5. Representació de Naturals
    2.6. Representació d'Enters
    2.7. Representació de caràcters en ASCII
    2.8. Format de les instruccions MIPS
    2.9. Punters
    2.10. Vectors
    2.11. Strings
    2.12. Resum del tema

3. Traducció de programes

    3.1. Operacions lògiques i desplaçaments
    3.2. Sentències if-then-else i while
    3.3. Subrutines: Introducció
    3.4. Subrutines: salvar i restaurar el context
    3.5. Subrutines: el bloc d'activació
    3.6. Estructura de la memòria
    3.7. Compilació, muntatge i càrrega
    3.8. Exemple de revisió

4. Matrius

    4.1. Matrius

4.2. Accés seqüencial a vectors/matrius

5. Aritmètica d'enters i coma flotant

5.1. Overflow de suma i resta d'enters
5.2. Multiplicació entera de 32 bits amb resultat de 64 bits
5.3. Divisió entera de 32 bits amb càlcul del residu
5.4. Coma flotant: representació
5.5. Coma flotant: suma i multiplicació
5.6. Coma flotant: (no) associativitat

6. Memòria Cache

6.1. Introducció
6.2. Disseny bàsic d'una cache
6.3. Mesures de rendiment
6.4. Millores: Associativitat i Multinivell

7. Memòria Virtual

7.1. Motivacions
7.2. Traducció d'adreces
7.3. Fallo de pàgina
7.4. Traducció ràpida amb TLB
7.5. Esquema hardware: datapath/TLB/cache/memòria

8. Excepcions i interrupcions

8.1. Excepcions i interrupcions
8.2. Crides al sistema
8.3. Entrada/Sortida

Tema 1. Introducció

<1.1>① **1.1.** Volem executar un programa escrit en Java, en un ordinador MIPS, però sols disposem dels següents elements:

- Un compilador de C a llenguatge màquina x86, escrit en llenguatge màquina x86
- Un traductor binari de llenguatge màquina x86 a llenguatge màquina MIPS, escrit en llenguatge màquina MIPS
- Un intèrpret de Java, escrit en C

¿Quins passos cal fer per executar el nostre programa en Java en l'ordinador MIPS, usant solament els elements disponibles?

<1.1>② **1.2.** Volem executar un programa escrit en C, en un ordinador x86, però sols disposem dels següents elements:

- Un intèrpret de Java, escrit en llenguatge màquina MIPS
- Un compilador de C a llenguatge màquina x86, escrit en Java
- Un traductor de llenguatge màquina MIPS a llenguatge màquina x86, escrit en llenguatge màquina x86.

a) ¿Quins passos cal fer per executar el nostre programa en C en l'ordinador x86, usant solament els elements disponibles?

b) ¿Quins passos caldria fer si es vol tornar a executar el mateix programa?

<1.2>① **1.3.** Traduim un programa en alt nivell a llenguatge assemblador i en fem 2 versions: Una en MIPS, que s'executarà en els processadors P1 i P2. L'altra, en x86, que s'executarà en P3. Cada processador té les següents característiques.

Processador	Freqüència	CPI mitjá (d'aquest programa)	ISA	#Instruccions
P1	2 Ghz	1.5	MIPS	$2*10^6$
P2	2.5 Ghz	1	MIPS	$2*10^6$

a) $T_1 = N \cdot CPI \cdot 1/F = \frac{2 \cdot 10^6 \cdot 1,5}{2 \cdot 10^9} = 1,5\ ms$

$T_2 = \frac{2 \cdot 10^6 \cdot 1}{2,5 \cdot 10^9} = 0,8\ ms$ $\quad T_3 = \frac{10^6 \cdot 0,75}{3 \cdot 10^9} = 0,25\ ms$

$\frac{T_1}{T_3} = 6$ $\quad \frac{T_2}{T_3} = 3,2$

c) Podemos comparar el rendimiento de P_1 y P_2 si tenemos el CPI medio de todos los programa
P_3 no se puede comparar porque tiene un lenguaje maquina distinto (x86)

P1 Juny/2019

	CPI	f1	f2
Arim	4	$3 \cdot 10^9$	$10 \cdot 10^9$
saltos	2	$2 \cdot 10^9$	$16 \cdot 10^9$
Otros	1	$4 \cdot 10^9$	$8 \cdot 10^9$

$f = 4\ GHz$
$P = 100\ W$

$T_A + T_B + T_O = T_{exe} = 25\ s$

$E = \overset{P}{100} \cdot \overset{t}{25} = 2500\ J$

$T_A = \frac{13 \cdot 10^9 \cdot 4}{4 \cdot 10^9} = 13\ s$

$ganancia = \frac{100}{80} = 1,25$

$T_B = \frac{18 \cdot 2}{4} = \frac{36}{4} = 9\ s$

$T_O = \frac{12 \cdot 1}{4} = 3\ s$

Ley Amdhal

$f1 = 20 \cdot 10^9$ ciclos

$\frac{100}{20} = 5$ max speed up

P3 mayo/2019

	N (10^9)	CPI
A	8	7
B	6	5
C	4	4

$56 + 30 + 16 = \frac{102}{1,5} = 68\ s$

$E = 100 \cdot 68 = 6800\ J$

Como la ganancia del nuevo procesador es 2, podemos decir que reduce el tiempo a la mitad.

$f = 1,5\ GHz$
$P = 100\ W$

$\frac{10 + 12 + 10 + 2}{x} = 34 \rightarrow f = 1\ GHz$

$100 = C \cdot \alpha \cdot V^2 \cdot f \rightarrow \frac{100}{1,5 \cdot 10^9} = C \alpha V^2 = 66,6\overset{6,66 \cdot 10^{-9}}{6}$

$x = 1,5C \cdot 1,2\alpha \cdot V^2 \cdot f_N \rightarrow \frac{x}{1,8} = C \alpha V^2 = 119,88$

Processador	Freqüència	CPI mitjá (d'aquest programa)	ISA	#Instruccions
P3	3 Ghz	0.75	Intel x86	10^6

Contesta les següents preguntes raonant les respostes:

a) Quin dels tres processadors executa més ràpidament el programa? Quantes vegades és més ràpid que els altres dos?

b) Sense conèixer el nombre d'instruccions, podem calcular si el processador P1 executa el programa més ràpid que el processador P2? I més ràpid que el processador P3?

c) Amb aquestes dades, podem comparar el rendiment de P1 i P2 en general (no només per aquest programa)? I el rendiment de P1 i P3? I si el CPI fos el CPI mitjà per a tots els programes?

<1.2>② **1.4.** Tenim dos processadors diferents P1 i P2 que executen una seqüencia de 10^6 instruccions. Cada processador té les següents característiques.

Processador	Freqüència	CPI
P1	4 Ghz	1.25
P2	3 Ghz	0.75

Contesta les següents preguntes raonant les respostes:

a) Quins dels dos processadors té el millor rendiment? Quantes vegades és més ràpid?

b) Quantes instruccions pot executar el processador P2 en el temps que triga P1 en fer les 10^6 instruccions?

<1.2>② **1.5.** Tenim un processador que no disposa d'un ISA per operar amb nombres en coma flotant. L'execució d'operacions amb nombres en coma flotant s'aconsegueix per mitjà de subrutines. En base a estadístiques se sap el següent:

- El CPI promig és 3
- El 20% de les instruccions executades per qualsevol programa són instruccions que formen part de les subrutines de coma flotant

Volem avaluar la conveniència d'afegir al processador instruccions que operin amb nombres en coma flotant. Al afegir aquestes instruccions observem el següent:

- El temps de cicle del processador passa de 1 ns a 1,05 ns
- El CPI promig passa de 3 a 4
- El número promig d'instruccions executades en les subrutines de coma flotant és ara 10 vegades menor.

Contesta les següents preguntes raonant les respostes:

a) ¿Val la pena afegir instruccions de coma flotant al repertori?

b) ¿Valdria la pena, si les subrutines de coma flotant consumissin el 40% de totes les instruccions (en comptes del 20% del supòsit anterior)?

<1.2>② **1.6.** En un processador de 1Ghz executem un programa P distribuït de la següent forma

Arit	Store	Load	Branch	Total
500	50	100	50	700

a) Si les instruccions Arit triguen un 1 cicle, load i store 5 cicles, i els branch 2 cicles. Quin seria el temps d'execució de P?

b) Quin seria el CPI?

c) Si optimitzem el codi de forma que ara només realitzem 50 instruccions load, quin seria el guany obtingut? i el CPI?

<1.2>③ **1.7.** Tenim dos implementacions diferents del mateix ISA, on hi ha 4 classes d'instruccions A, B, C, D. La taula següent mostra la freqüencia de rellotge i el CPI de cada classe, per les dues implementacions.

	Clock Rate	CPI A	CPI B	CPI C	CPI D
P1	1.5Ghz	1	2	3	4
P2	2 Ghz	2	2	2	2

a) Quin seria el CPI en mitjana de cada implementació, suposant un programa amb instruccions distribuïdes de la forma : 10% Classe A, 20% Classe B, 50% Classe C i 20% Classe D?

b) Quina de les dues implementacions és mes ràpida, suposant un programa amb 10^6 instruccions?

<1.2>③ **1.8.** Considera dos processadors que executen un programa de la forma que mostra la taula següent

	Coma flotant	Enters	Load/Store	Branch	Total
a	35 s	85 s	50 s	30 s	200 s
b	50 s	80 s	50 s	30 s	210 s

Per cadascuna de les execucions, contesta les preguntes següents:

a)

$35 \cdot 0,8 = 28$

$50 \cdot 0,8 = 36$

Sa $\frac{200}{193} = 1,04$

Sb $210/200 = 1,05$

b)

c)

- 40 Br 30s no se puede reducir 40s
- 42 Br 30s no se puede reducir 40s

a) Quina és la millora de rendiment, si reduïm el temps de les operacions de coma flotant en un 20 %?

b) Quin és el percentatge de reducció del temps d'execució d'operacions d'enters que es necessitaria per aconseguir una millora del 20% en el rendiment global?

c) Podem reduïr el temps d'execució total en un 20%, millorant únicament les operacions branch?

<1.3>① **1.9.** La següent taula mostra la freqüència de rellotge (F), voltatge (V) i potència dinàmica (P) de dos processadors.

Processador	F	V	P	Càrrega capacitiva (C)
A	10 MHz	5V	2W	
B	3GHz	1V	100W	

a) Calcula la càrrega capacitiva dels processadors A i B.

b) Quina seria la potència del processador A si, sense canviar-ne el voltatge ni la capacitància, volguèssim aconseguir la mateixa freqüencia de rellotge que el processador B?

<1.3>② **1.10.** La següent taula mostra l'evolució de la freqüència de rellotge (F), voltatge (V) i potència dinàmica (P) en vuit generacions (G) de processadors Intel al llarg de 28 anys.

G	Processador	F	Ratio freq. G_N/G_{N-1}	V	P	Ratio potència G_N/G_{N-1}	C
1	80286 (1982)	12.5 MHz	-	5V	3.3W	-	
2	80386 (1985)	16 MHz	1.28	5V	4.1W	1.24	
3	80486 (1989)	25 MHz		5V	4.9W		
4	Pentium (1993)	66 MHz		5V	10.1W		
5	Pentium Pro (1997)	200 MHz		3.3V	29.1W		
6	Pentium 4 Willamette (2001)	2 GHz		1.75V	75.3W		
7	Pentium 4 Prescott (2004)	3.6 GHz		1.25V	103W		
8	Core 2 Ketsfield (2007)	2.667GHz		1.1V	95W		

a) Calcula els ratios de potència i freqüència entre les generacions consecutives. Per exemple, l'any 1985 la freqüència és va incrementar en 16/12.5 vegades, és a dir, en 1.28 vegades.

b) Quin és el canvi (ratio) més gran en la freqüència entre generacions? I en la potència?

c) Quina és la mitjana geomètrica[1] dels ratios de freqüència i potència entre les generacions consecutives.

d) Quantes vegades més gran és la freqüència de la darrera generació i de la primera? I la potència?

e) Calcula la càrrega capacitiva (C) de cada generació.

<1.3>② **1.11.** Tot i que la potència dinàmica és la principal font de disipació de la potència en una CMOS, la pèrdua produeix una disipació de la potencia estàtica, de la forma V x I_{leak}. Quant més petit és el circuit més significativa és la potència estatica. La taula següent ens mostra la dissipació de potència estàtica i dinàmica per a dues generacions de processadors.

Tecnologia	Potència dinàmica	Potència estàtica	Potència total	Voltatge	Corrent de pèrdua (I_{leak})
250nm	49W	1W	50W	3.3V	
90nm	75W	45W	120W	1.1V	

a) Troba quin percentatge de potència total dissipada correspon a la potència estàtica i dinàmica en cada generació.

b) Si la potència estàtica depén del corrent de pèrdua(I_{leak}), P = V x I_{leak}. Troba el corrent de pèrdua per a cada tecnologia.

c) En quin percentatge podríem reduir la potència total de cada processador reduint només la potència dinàmica?

<1.4>② **1.12.** Considera dos processadors A i B que executen versions diferents d'un mateix programa. Per a cada tipus d'instruccions, la taula següent mostra el nombre d'instruccions executades i els cicles per instrucció:

	Coma flotant		Enters		Load/Store		Branch	
ProcA	90 inst	2 cpi	240 inst	1,5 cpi	150 inst	6 cpi	120 inst	3 cpi
ProcB	45 inst	3 cpi	225 inst	1 cpi	135 inst	5 cpi	45 inst	2 cpi

Per a cadascuna de les execucions, contesta les preguntes següents:

a) Calcula el CPI (cicles per instrucció) promig de l'execució del programa en ambdós processadors.

1. mitjana geomètrica = $(a_0 \cdot a_1 \cdot a_2 \cdots a_{n-1})^{1/n}$

b) Suposant que els processadors A i B treballen a les freqüències F_A i F_B respectivament, indica la relació F_A/F_B per tal que els dos processadors triguin el mateix temps a executar aquest programa.

c) Si hipotèticament aconseguíssim reduir a 0 el temps d'execució del tipus d'instruccions Load/Store del processador A, quin seria el guany de rendiment (speed-up) aconseguit en aquest processador?

Tema 2. Instruccions i tipus bàsics de dades

<2.2> ① **2.1.** En els següents apartats hauràs de traduir codi C a codi MIPS. Suposa que les variables *f*, *g*, *h*, *i* i *j* són variables enteres de 32 bits i han estat assignades als registres $t0, $t1, $t2, $t3 i $t4 respectivament. Tradueix els següents fragments a llenguatge assemblador:

a)
```
f = g + h + i + j
```

b)
```
g = f + (h + 5) - i
```

<2.2> ① **2.2.** En els següents apartats hauràs de traduir codi MIPS a codi C. Suposa que les variables f, g, h, i i j són variables enteres de 32 bits i han estat assignades als registres $t0, $t1, $t2, $t3 i $t4 respectivament. Tradueix els següents fragments a llenguatge C:

a)
```
addu   $t0, $t1, $t2
addu   $t1, $t3, $t4
addu   $t1, $t1, $t1
addu   $t0, $t0, $t1
```

b)
```
addu   $t1, $t1, $t2
addu   $t3, $t3, $t4
addu   $t0, $t1, $t3
addiu  $t0, $t0, 2
addu   $t0, $t0, $t0
```

<2.3> ① **2.3.** Donada la següent declaració de dades, que s'emmagatzemarà a memòria a partir de l'adreça `0x10010000`:

```
.data
.byte  1, 2, 3, 4
.word  -1, 1, -2, 2, -3, 3
.word  0x12345678
```

Indiqueu quin serà el valor en hexadecimal dels bytes emmagatzemats a les adreces de memòria 0x1001000C i 0x1001001C (poseu NA si no es pot determinar a partir de la declaració donada).

<2.3> ①

2.4. Les variables enteres A, B, C estan ubicades a les adreces 0x10010000, 0x10010004 i 0x10010008 de memòria. Donats els següents continguts inicials de registres i de memòria, tots ells en hexadecimal:

```
$t0=0x10010000, $t2=0x10010008
```

ADRECES	CONTINGUTS (hexadecimal, byte per byte)
0x10010000	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0x10010010	00 00 00 00 EC FD 0E 0F 00 00 00 00 00 00 00 00

Determina, per a cada instrucció, quin registre o adreça de memòria es modifica, i quin és el seu contingut.

```
        .text
        lw      $t1, 0($t0)
        lw      $t3, 0($t2)
        sw      $t3, -8($t2)
        lw      $t4, 4($t0)
        sw      $t1, -4($t2)
        sw      $t4, 0($t2)
```

<2.3> ②

2.5. Donada la següent declaració:

```
        .data
A:      .byte 0, 0, 0, 0, 0
```

Escriu un fragment de no més de 8 línies de codi en assemblador tal que guardi el valor de *$t1* a l'adreça *A+1*, però sense produir cap excepció (es poden escriure valors temporals a la posició A de memòria, si cal).

<2.3> ②

2.6. Indica quin és el contingut de memòria a partir de l'adreça 0x10010000 i a nivell de byte si tenim la següent declaració de variables globals.

```
        .data
        .byte  0x10
A:      .half  -5
B:      .word  -1, 67
C:      .byte  -4, '5', 6
D:      .half  66, 67
E:      .dword 0x5799
F:      .byte  'C'
G:      .half  0x66
H:      .dword 579
```

<2.4> ①

2.7. Donades les següents declaracions de variables, emmagatzemades a memòria a partir de l'adreça 0x10010000:

```
        .data
A:      .word  5, 2
B:      .word  3
C:      .word  4, 0x0FFFF, 0x4180
D:      .word  0x10010008, 0
```

Contesteu els següents apartats, suposant que tots ells parteixen del mateix estat inicial (els càlculs d'un apartat no influeixen en els següents):

Quin és el valor final de la variable `B` després d'executar el següent codi?

```
        .text
        la      $t1, C
        lw      $t2, 12($t1)
        lb      $t1, 8($t2)
        la      $t3, B
        lb      $t4, 0($t3)
        addu    $t1, $t1, $t4
        sb      $t1, 0($t3)
```

Expliqueu textualment i de forma concisa què fa el següent fragment de codi

```
        .text
        la      $t1, A+1
        lw      $t1, 0($t1)
```

<2.4> ② **2.8.** Indica quin és el contingut de memòria a partir de l'adreça 0x10010000 i a nivell de paraula si tenim la següent declaració de variables globals.

```
        .data
        .byte  0, 1, 2
A:      .byte  5, -1
B:      .half  0x66
C:      .word  4, 0x0FDF
D:      .half  3
E:      .byte  25
F:      .byte  0x08
G:      .dword 0x10010008
```

<2.4> ② **2.9.** Un programador escriu el següent codi per fer l'assignació de 0xAAAABBBB al registre $t0:

```
        ori     $t0, $zero, 0xBBBB
        lui     $t0, 0xAAAA
```

a) És correcte? Raona la teva resposta.

b) Ara volem fer l'assignació del valor 0xFFFFFFFF al registre $t0 amb alguna de les següents pseudoinstruccions:

```
        li      $t0, 0xFFFF
        li      $t0, -1
        li      $t0, -0xFFFF
```

Quina o quines assignacions donen el resultat desitjat? Per què?

<2.6> ① **2.10.** Convertiu els següents números enters decimals als següents formats de representació binària per a 8 bits. Escriviu el resultat en hexadecimal:

	Ca1	Ca2	Signe i Magnitud	Excés a 2^7-1
-78				
125				
0				
-1				

<2.6> ① **2.11.** Completa la següent taula, on es representen en diferents codificacions binàries de 4 bits (Ca2, Ca1, Excés, Signe i magnitud), els enters decimals en l'interval [-8, 8]. Per als valors no codificables amb 4 bits, poseu NC.

N	Signe i Magnitut	Ca1	Ca2	Excés a 2^3-1
8				
7				
6				
5				
4				
3				
2				
1				
0				
-1				
-2				
-3				
-4				
-5				
-6				
-7				
-8				

<2.6> ① **2.12.** Completa la següent taula, on s'han d'especificar especificar el rang i la representació del zero, dels diferents formats de representació dels enters decimals per a n bits:

Formats representació enters en base 2	Rang	Representació del 0
Signe i Magnitud		
Complement a 1		
Complement a 2	$[-2^{n-1}, 2^{n-1} - 1]$	0 ... 00
Excés a 2^{n-1}-1	...	

<2.6> ① **2.13.** Codifica en complement a 2 els següents números enters, emprant el mínim format possible (8 o 16 bits), i expressant el resultat en hexadecimal:

a) +123

b) +255

c) -127

d) +170

e) +128

f) -230

g) -18

h) -128

i) -1

<2.6> ① **2.14.** Converteix els següents nombres enters, representats en complement a dos de 16 bits, a la representació decimal amb signe:

a) 0x0123

b) 0xfe23

<2.6> ① **2.15.** Donats els següents números en base 2, calcula el seu valor implícit en decimal. Primer, interpretant-los com a números naturals, i després com a enters en Ca2:

a) 0000 1111

b) 1000 0000

c) 1111 1111

d) 1111 1111 1111 0000

<2.6> ② **2.16.** Quin és el menor nombre enter (el més negatiu) representable en complement a 2 amb 13 bits ?

a) Representat en hexadecimal (en complement a 2)

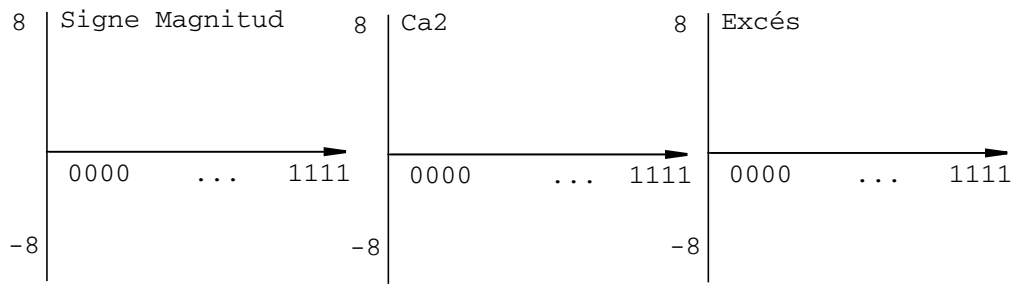
b) Representat en decimal

<2.6> ② **2.17.** Quin és el major nombre enter (positiu) representable en complement a 2 amb 13 bits ?

a) Representat en hexadecimal (en complement a 2)

b) Representat en decimal.

<2.6> ③ **2.18.** Dibuixa les següents gràfiques de forma aproximada, que mostren com es fa la correspondència entre la codificació binària en 4 bits i els valors enters decimals en l'interval [-8, 8], per a 3 sistemes de representació:



<2.7> ② **2.19.** Donades les següents declaracions:

```
        .data
u1:     .byte  '9'
u2:     .byte  '3'
d3:     .byte  0,  0
```

Cada dada *u1* i *u2* conté un caràcter ASCII que representa un dígit decimal. Escriu un programa en assemblador que sumi els naturals representats per *u1* i *u2* deixant el resultat en *d3* (que té un dígit de més per poder representar qualsevol suma sense desbordar-se). Feu servir l'algorisme típic de la suma, dígit a dígit.

<2.7> ③ **2.20.** Escriu en hexadecimal el contingut de la memòria ocupat per les següents declaracions, suposant que comencen a emmagatzemar-se a partir de l'adreça 0x10010000:

```
        .data
a:      .half  54
b:      .byte  'C'
c:      .half  0xAAAA
d:      .word  0x4F3C
e:      .byte  -18
f:      .dword 0xFFFFEEEE
```

Fes servir el següent format, canviant els zeros pels valors corresponents en hexadecimal (recordeu que per defecte està activa l'alineació automàtica de les variables a memòria):

ADRECES	CONTINGUTS (hexadecimal, byte per byte)
0x10010000	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x10010010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
	36 00 43 x AA AA xx xx 3C 4F 00 00 EE xx xx xx xx
	EE EE FF FF 00 00 00 00

<2.7> ③ **2.21.** Escriu en hexadecimal el contingut de la memòria ocupat per les següents declaracions, suposant que comencen a emmagatzemar-se a partir de l'adreça 0x10010000. Recorda que la directiva `.align 0` desactiva l'alineació automàtica de les declaracions de variables a memòria:

```
        .data
        .align 0
        .byte  24
        .align 1
B:      .byte  'A', 'C'
C:      .half  512, 0xAAAA
D:      .word  0x43, 0xF3C, 'a'
E:      .byte  -18, 'A'
```

Fes servir el següent format, canviant els zeros pels valors corresponents en hexadecimal:

ADRECES	CONTINGUTS (hexadecimal, byte per byte)
0x10010000	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x10010010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

<2.7> ③ **2.22.** Donada la següent declaració de variables en C:

```
char A = 'C';
int B = -1;
```

a) Tradueix la declaració anterior a llenguatge assemblador MIPS.

b) Especifica el valor hexadecimal del registre $t0 després d'executar el següent codi en assemblador:

```
.text
la     $t0, A
la     $t1, B
lb     $t0, 0($t0)
lw     $t1, 0($t1)
addu   $t0, $t0, $t1
```

<2.8> ① **2.23.** Convertiu les següents instruccions de llenguatge assemblador a llenguatge màquina, seguint el format de les instruccions MIPS:

Ll. Assemblador MIPS	Ll. Màquina MIPS (Bit a bit)	Ll. Màquina MIPS (Hexa)
`addu $t4, $t3, $t5`		
`addiu $t7, $t6, 25`		
`lw $t3, 0($t2)`		
`sw $t0, 0($t1)`		

<2.8> ② **2.24.** Desassembla els següents nombres binaris, escrits en hexadecimal a les corresponents instruccions MIPS, escrites en llenguatge assemblador.

a) `0xAE0BFFFC`

b) `0x8D08FFC0`

c) `0x0233a823`

d) `0x0233802A`

e) `0x3c011001`

f) `0xAE160004`

<2.8> ③ **2.25.** Estem estudiant les següents dues possibles modificacions a l'arquitectura MIPS:

1) Passar de 32 registres a 8 registres

2) Passar de 16 bits per l'operand immediat a 10 bits

Per a cadascuna d'elles contesta els següents apartats:

a) Quin impacte tindria cada modificació en la mida total d'una instrucció de tipus R?

b) I en la mida total d'una instrucció de tipus I?

<2.9> ② **2.26.** Suposem les següents declaracions en C:

```
int *m1, *m2;         /* m1 i m2 són punters a int (en memòria) */
main()
{
       int *r1, *r2;  /* r1 i r2 són punters a int (en registre) */
       ...
}
```

Suposant que els punters *r1 i r2* ocupen els registres $t1 i $t2, tradueix a assemblador MIPS les següents sentències en C, pertanyents a la funció *main*:

a) `r1 = r2;`

b) `*r1 = *r2;`

c) `m1 = m2;`

d) `*m1 = *m2;`

<2.9> ③ **2.27.** Donada la següent declaració de dades global, en C:

```
int *pdada;
```

Tradueix a una única sentència en C el conjunt d'instruccions de cada apartat:

a)
```
la     $t0, pdada
lw     $t0, 0($t0)
lw     $t1, 0($t0)
addiu  $t1, $t1, 4
sw     $t1, 0($t0)
```

b)
```
la     $t0, pdada
lw     $t1, 0($t0)
addiu  $t1, $t1, 4
sw     $t1, 0($t0)
```

c)
```
la     $t0, pdada
lw     $t0, 0($t0)
lw     $t1, 0($t0)
addiu  $t0, $t0, 4
sw     $t1, 0($t0)
```

<2.9> ③ **2.28.** Donades les següents declaracions de variables globals, en C:

```
int vec[6] = {2, 4, 0, 6, 8, 0};
int *punter;
```

Tradueix a una única sentència en C el conjunt d'instruccions de cada apartat:

a)
```
la     $t0, punter
la     $t1, vec + 8
sw     $t1, 0($t0)
```

b)
```
la     $t0, punter
lw     $t1, 0($t0)
addiu  $t1, $t1, 4
sw     $t1, 0($t0)
```

c)
```
la     $t1, vec
lw     $t3, 0($t1)
lw     $t4, 4($t1)
addu   $t3, $t3, $t4
sw     $t3, 4($t1)
```

d)
```
la     $t1, vec
la     $t0, punter
lw     $t2, 0($t0)
lw     $t3, 0($t2)
addu   $t3, $t3, 1
sw     $t3, 8($t1)
```

e)
```
la     $t0, punter
lw     $t2, 0($t0)
lw     $t3, 0($t2)
addu   $t3, $t3, 1
sw     $t3, 8($t2)
```

<2.10> ② **2.29.** Donades les següents declaracions de variables globals, en llenguatge C:

```
char *punterc;
short *punterh;
int *punteri;
long long *punterd;
```

Tradueix a assemblador MIPS les següents sentències:

a)
```
punterc++;
```

b)
```
punteri++;
```

c)
```
punterh++;
```

d)
```
punterd++;
```

e)
```
*punteri = *punteri + 5;
```

f)
```
*punterh = *punterh + 10;
```

<2.10> ② **2.30.** Donades les següents declaracions:

```
char a;
int b;
long long int c;
main()
{
       char *p;            /* punter guardat en $t0 */
       int *q;             /* punter guardat en $t1 */
       long long int *h;   /* punter guardat en $t2 */
       ...
}
```

Suposant que els punters *p*, *q* i *h* ocupen els registres $t0, $t1 i $t2, tradueix a assemblador MIPS les següents sentències en C, pertanyents a la funció *main*:

a) `q = q + 1;`

b) `a = *p;`

c) `h = &c;`

d) `b = *(q + b);`

e) `*h = *(h + b);`

f) `p[*q + 10] = a;`

g) `h = &h[*p];`

<2.10> ② **2.31.** Donades les següents declaracions en llenguatge C:

```
int dada;
int *pdada;
```

Tradueix a assemblador MIPS les següents sentències en C:

a) `pdada = &dada;`

b) `*pdada = *pdada + 1;`

c) `pdada = pdada + 1;`

d) `dada = dada - 1;`

<2.10> ③ **2.32.** Donades les següents declaracions:

```
char indx[100];            /* enters de 1 byte  */
short meitat[100];         /* enters de 2 bytes */
int val[100], vec[100];    /* enters de 4 bytes */
main()
{
       char c;
       int i, j;
       ...
}
```

Tradueix les següents sentències en C suposant que les variables *i*, *j*, i *c* ocupen els registres $t0, $t1, i $t2:

a) `i = val[5] + vec[10];`

b) `i = vec[10 + val[5]];`

2.30

a) `addiu $t1, $t1, 4`

b)
```
lb $t3, 0($t0)
la $t4, a
sb $t3, 0($t4)
```

c) `la $t2, c`

d)
```
la $t3, b
lw $t4, 0($t3)
sll $t4, $t3, 2
addu $t4, $t4, $t1
lw $t4, 0($t4)
sw $t4, 0($t3)
```

e) `*h = *(h + b)`

```
la $t3, b          $t3 = &b
lw $t3, 0($t3)     $t4 = b
sll $t3, $t3, 3    $t4 = b·8
addu $t3, $t3, $t2 $t4 = *h + b·8
lw $t3, 0($t3)
sw $t3, 0($t2)
```

f) `p[*q+10] = a`

```
lw $t3, 0($t1)
addiu $t3, $t3, 10
addu $t3, $t3, $t0
la $t4, a
lb $t5, 0($t4)
sb $t5, 0($t3)
```

g) `h = &h[*p]`

```
lb $t3, 0($t0)
sll $t3, $t3, 3
addu $t2, $t2, $t3
```

2.32

a) `i = val[5] + vec[10]`

```
la $t3, val
la $t4, vec
lw $t4, 40($t4)
lw $t3, 20($t3)
addu $t0, $t3, $t4
```

b) `i = vec[10 + val[5]]`

```
la $t3, val
la $t4, vec
lw $t3, 20($t3)
addiu $t3, $t3, 10
sll $t3, $t3, 2
addu $t4, $t3, $t4
lw $t0, 0($t4)
```

c) `c = indx[i + val[j]];`

d) `c = indx[meitat[i]];`

e) `i = meitat[vec[i] + val[j]];`

f) `i = val[meitat[i]] + vec[j];`

g) `i = vec[indx[i]] - val[j];`

h) `i = meitat[indx[val[j]]];`

<2.11> ③ **2.33.** Tradueix a assemblador MIPS el següent programa en C que escriu a l'string *minus* els caràcters ASCII que representen cada un dels dígits decimals del vector *vec*, realitzant la conversió a minúscules (suma la diferencia numèrica entre els caràcters 'a'-'A' = 32).

```
#define N 8
char minus[N+1];
unsigned int vec[N] = {70, 85, 78, 67, 73, 79, 78, 65};
void main()
{
       int i=0;
       while(i<N) {
              minus[i] = (char)vec[i] + 'a'-'A';
              i++;
       }
       minus[N+1]=0;            # Posa la marca de final de string
}
```

Tema 3. Traducció de programes

<3.1> ① **3.1.** Calcula el resultat en hexadecimal de les següents operacions lògiques bit a bit amb 16 bits:

a) `0x7654 | 0x3333` `/* or bit a bit */`

b) `0x7654 & 0x3333` `/* and bit a bit */`

c) `0x7654 << 2` `/* shift esq. 2 posicions */`

d) `0x7654 >> 3` `/* shift lògic dreta 3 pos. */`

<3.1> ① **3.2.** Sigui la següent sentència en C:

```
C = A^B;            /* fem la xor bit a bit entre A i B */
```

Tradueix l'anterior sentència a assemblador sense utilitzar la instrucció xor. Recorda que l'anterior sentència és equivalent a:

```
C = (~A & B) | (A & ~B);
```

On els operadors de C "~", "&" i "|" denoten respectivament les operacions lògiques *not*, *and* i *or* bit a bit.

<3.1> ② **3.3.** Escriu les seqüències de codi necessàries per escriure en $t1 el resultat de les següents condicions lògiques, sense fer servir instruccions de salt. El valor final de $t1 ha de ser 0 si no s'acompleix la condició, o diferent de zero altrament.

a) L'enter representat per $t1 és negatiu

b) L'enter representat per $t1 és múltiple de 4

c) El natural representat per $t1 és tal que el seu quadrat no es pot escriure amb 32 bit

d) El natural representat per $t1 és tal que la suma amb si mateix no es pot escriure amb 32 bits

e) Els bits 1, 3 i 5 de $t1 valen zero

<3.1> ③ **3.4.** El repertori de MIPS inclou 3 instruccions de desplaçament anàlogues a les estudiades, excepte que permeten desplaçar un nombre variable de posicions. El nombre de posicions a desplaçar ve donat pels 5 bits de menor pes del registre rs que apareix com a tercer operand (la resta de bits són ignorats).

```
sllv rd, rt, rs     # rd=rt<<rs (anàleg a sll)
srlv rd, rt, rs     # rd=rt>>rs (anàleg a srl)
srav rd, rt, rs     # rd=rt>>rs (anàleg a sra)
```

Fent servir alguna d'aquestes instruccions, programa una seqüència de no més de 6 instruccions (es pot fer amb 4) que serveixi per posar a 0 el bit *i*-èssim del registre $t1, deixant la resta de bits intactes, i suposant que *i* està guardat en $t2 (pots usar d'altres registres per guardar resultats temporals).

<3.1> ③ **3.5.** Fes un programa que realitzi els següents desplaçaments sobre un número de 64 bits emmagatzemat als registres $t2 (32 bits de més pes) i $t1 (32 bits de menys pes). La clau de l'exercici és calcular en un registre a part el(s) bit(s) que es desplacen d'un registre a l'altre, alineant-lo(s) a dreta o esquerra segons convingui:

a) Shift a l'esquerra 1 posició

b) Shift lògic a la dreta 1 posició

c) Shift aritmètic a la dreta 1 posició

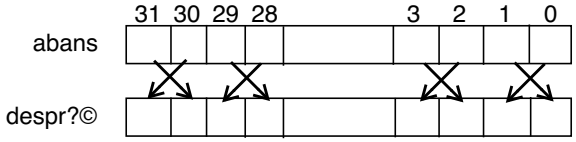
d) Fent servir les instruccions de shift variable explicades al problema anterior, fer un shift lògic a l'esquerra, *n* posicions ($n<32$), on *n* ocupa el registre $t3.

e) Shift lògic a la dreta de *n* posicions ($n<32$), on *n* ocupa $t3.

<3.1> ② **3.6.** Suposem que el registre $t0 conté dos nombres enters de 16 bits en complement a 2, un en els 16 bits de més pes, i un altre en els 16 de menys pes. Escriu una seqüència d'instruccions en assemblador MIPS que extregui aquests dos números, els representi en 32 bits extenent el signe, i els guardi en $t1 i $t2, respectivament, sense usar cap instrucció d'accés a memòria.

<3.2> ① **3.7.** Fes un programa que compti el nombre de bits actius que hi ha en el registre $t1, executant el mínim d'instruccions possible i sense fer cap *load*. El resultat s'ha de deixar en el registre $t3.

<3.2> ② **3.8.** Escriu un fragment de codi en assemblador MIPS que intercanvïi tots els bits parells amb els bits senars del registre $t0:



<3.2> ②

3.9. Suposem que denotem el valor inicial de cada bit de $t4 amb una lletra de la següent manera (en aquest exemple, el valor inicial del bit 2 és `n` i el del bit 5 és `k`):

```
$t4 = ABCD EFGH IJKL MNOP abcd efgh ijkl mnop
```

a) Escriu les instruccions necessàries perquè el bit de més pes passi a ser el de menys pes i la resta de bits es desplacin una posició cap a l'esquerra, és a dir que el contingut final del registre $t4 passi a ser:

```
$t4 = BCD EFGH IJKL MNOP abcd efgh ijkl mnopA
```

b) Seguint la mateixa notació, escriu el contingut del registre $t4 després d'executar les següents instruccions:

```
     li       $t1, 16
     addiu    $t2, $t4, 0
bucle:
     srl      $t4, $t4, 1
     addu     $t2, $t2, $t2
     addiu    $t1, $t1, -1
     bne      $t1, $zero, bucle
     or       $t4, $t4, $t2
```

<3.2> ①

3.10. El vector d'enters anomenat *Array* està ubicat a l'adreça 0x10000000 de memòria, i ha estat declarat així:

```
       .data
Array: .word 2, 4, 6, 1
```

Fes un programa en assemblador MIPS que ordeni els elements del vector del més petit al més gran, deixant el valor més petit en la posició de memòria més baixa.

<3.2> ②

3.11. Donades les següents declaracions:

```
     .data
v1:  .byte     '0', '0', '4', '1', '9'
v2:  .byte     '8', '9', '7', '2', '3'
v3:  .byte      0,   0,   0,   0,   0,   0
```

Cada un dels vectors *v1*, *v2* i *v3* conté caràcters ASCII que representen dígits numèrics (del '0' al '9'). En conjunt, cada vector representa un número natural decimal, amb les unitats en l'última posició, i el dígit de més pes en la posició inicial. Escriu un programa en assemblador que sumi els naturals representats per *v1* i *v2* deixant el resultat en *v3* (que té un dígit de més per poder representar qualsevol suma sense desbordar-se). Feu servir l'algorisme típic de la suma, dígit a dígit.

<3.2> ②

3.12. Donat el següent programa escrit en alt nivell:

```
main() {
     int a = 23;    /* guardat al registre $t0 */
     int b = 12;    /* guardat al registre $t1 */
     int i;         /* guardat al registre $t2 */
     for(i=0; i<10; i++)
             a += b;
}
```

a) Tradueix-lo a assemblador MIPS.

b) Quin és el número total d'instruccions del processador que s'executen?

c) Suposant que el processador té un clock rate = 2GHz, i que totes les instruccions tenen el mateix CPI = 2, si el nombre d'instruccions executades fossin 10, quants segons trigaria a executar-se el programa?

d) En un processador amb clock rate = 2GHz, sabent que el programa triga $5x10^{-9}$ s, quin serà el CPI del programa (cicles per instrucció)?

<3.2> ③ **3.13.** Donada la següent declaració de la variable global V (on N és una constant):

```
int v[N];
```

Calcula el temps d'execució en cicles de rellotge de cada una de les següents versions del programa que serveix per sumar els elements de V, suposant que:

- Els accessos a memòria tarden 4 cicles
- Els salts que no salten tarden 1 cicle
- Els salts que salten tarden 2 cicles
- La resta d'instruccions tarden 1 cicle

a)
```
     move    $t0, $zero          # suma
     move    $t1, $zero          # i
     move    $t2, $zero          # offset
bucle:
     slti    $t3, $t1, N
     beq     $t3, $zero, fi
     la      $t4, V
     addu    $t4, $t4, $t2
     lw      $t4, 0($t4)
     addu    $t0, $t0, $t4
     addiu   $t2, $t2, 4
     addiu   $t1, $t1, 1
     b       bucle
fi:
```

b)
```
     move    $t0, $zero          #suma
     move    $t1, $zero          # i
bucle:
     slti    $t2, $t1, N
     beq     $t2, $zero, fi
     la      $t3, V
     sll     $t2, $t1, 2
     addu    $t2, $t3, $t2
     lw      $t2, 0($t2)
     addu    $t0, $t0, $t2
     addiu   $t1, $t1, 1
     b       bucle
fi:
```

c)
```
     move    $t0, $zero          #suma
     move    $t1, $zero          # i
```

```
bucle:
     la       $t3, V
     sll      $t2, $t1, 2
     addu     $t2, $t2, $t3
     lw       $t2, 0($t2)
     addu     $t0, $t0, $t2
     addiu    $t1, $t1, 1
     slti     $t3, $t1, N
     bne      $t3, $zero, bucle
```

d)
```
     move     $t0, $zero          #suma
     li       $t1, N-1            # i
bucle:
     la       $t3, V
     sll      $t2, $t1, 2
     addu     $t2, $t3, $t2
     lw       $t2, 0($t2)
     addu     $t0, $t0, $t2
     addiu    $t1, $t1, -1
     slt      $t3, $t1, $zero
     beq      $t3, $zero, bucle
```

e)
```
     move     $t0, $zero          #suma
     li       $t1, N              # i
     la       $t2, V
bucle:
     lw       $t3, 0($t2)
     addu     $t0, $t0, $t3
     addiu    $t2, $t2, 4
     addiu    $t1, $t1, -1
     bne      $t1, $zero, bucle
```

f) Programeu una nova versió que disminueixi el temps d'execució de l'apartat e).

<3.2> ③ **3.14.** Donades dues variables enteres *a* i *b*, ja inicialitzades i ocupant els registres $t1 i $t2 respectivament:

a) Programa en C una seqüència d'instruccions que multipliqui a * b utilitzant un bucle while i operacions de suma. El resultat s'ha de deixar a la variable *c*, ubicada al registre $t3.

b) Tradueix-lo a assemblador MIPS.

c) Quin és el número màxim d'instruccions MIPS que poden arribar a executar-se en aquest programa?

d) En un processador amb clock rate = 2GHz i suposant que totes les instruccions que has fet servir utilitzen 2 cicles de rellotge, quants segons trigaria a executar-se el programa?

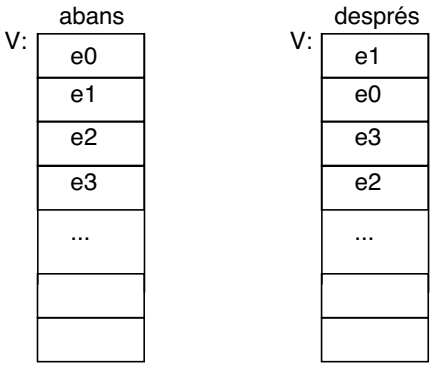
e) Imagina't ara que en comptes d'utilitzar sumes es fa servir una única operació de multiplicació que s'explicarà al Tema 5. Imagina't que aquesta opera-

ció utilitza 100 cicles de rellotge. Quants segons trigaria a executar-se el programa (d'una única instrucció)? Quin serà el CPI del programa?

<3.2> ③ **3.15.** Donada la següent declaració en C (on N és una constant):

```
int V[2*N];
```

Escriu un fragment de codi en assemblador MIPS que intercanvïi cada element que ocupa un índex parell amb l'element següent, d'índex senar:



<3.2> ③ **3.16.** Tradueix a llenguatge assemblador MIPS el següent programa en C, suposant que les variables *a*, *b*, *c* ja han estat inicialitzades i ocupen els registres *$t1*, *$t2* i *$t3* respectivament:

```
main() {
     int a, b, c, i;       /* i s'emmagatzema en $t0  */
     ...                   /* inicialització de les variables */
     if (a>0)
     {
              i = b;
              if (((i-4) < a) && (i>0))
                     b = b+c;
              else if (c>0)
                     b = b-c;
     }
     else
              a = a+b;
}
```

<3.2> ③ **3.17.** Tradueix a assemblador MIPS la sentència *if* del següent programa, que converteix un dígit hexadecimal (representat per un caràcter ASCII guardat a la variable global *c*) al seu valor binari equivalent i l'escriu a la variable global *num*, que és un enter de 8 bits. Suposem que el valor de *c* és un dels caràcters {'0'..'9', 'A'..'F', 'a'..'f'}.

```
char c, num;

main()
{
```

3.16

```
        ble $t1, $zero, else 1
if 1:
        move $t0, $t2
        addiu $t4, $t0, -4
        bge $t4, $t1, else 2
        ble $t0, $zero, else 2
if 2:
        addu $t2, $t2, $t3
        ble $t3, $zero, endif 2
elseif 2:
        subu $t2, $t2, $t3
endif 2:
        b endif 1
else 1:
        addu $t1, $t1, $t2
endif 1:
```

```
        ...                             /* inicialitzacions */

        if ((c>='A') && (c<='F'))
                num = c - 'A' + 10;
        else
                if ((c>='a') && (c<='f'))
                        num = c - 'a' + 10;
                else
                        num = c - '0';
}
```

<3.2> ③ **3.18.** Tradueix a assemblador MIPS el següent programa escrit en alt nivell:

```
int D[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
main() {
        int a = 0;              /* guardat al registre $t0 */
        int b = 12;             /* guardat al registre $t1 */
        while(a < 10) {
                D[a] = b + a;
                a += 1;
        }
}
```

<3.2> ③ **3.19.** Tradueix a assemblador MIPS el següent programa escrit en alt nivell, suposant que les variables globals ja han estat inicialitzades:

```
#define N 30;
int V[N], suma, elems;
main() {
        int i;
        ...             /* inicialització de les variables */
        i=0;
        elems = 0;
        suma = 0;
        while (i<N) {
                if (V[i] > 5) {
                        suma = suma + V[i];
                        elems++;
                }
                i++;
        }
}
```

<3.3> ② **3.20.** La instrucció MIPS `jalr` serveix per cridar a subrutines. És anàloga a la instrucció `jal`, amb dues diferències: `jal` codifica l'adreça de salt dins la pròpia instrucció (escrivim una etiqueta) i guarda l'adreça de retorn en $ra, mentre que `jalr` especifica l'adreça de salt en el registre rs i guarda l'adreça de retorn en rd:

```
jal etiqueta        # escriu @retorn en $ra i salta a etiqueta
jalr rd, rs         # escriu @retorn en rd i salta a rs
```

Suposem que s'executa el següent fragment de codi MIPS, un xic extravagant:

```
                la      $ra, etiq
                addiu   $ra, $ra, -4
        etiq:   jalr    $ra, $ra
```

3.19

```
        move $t0, $zero
        move $t1, $zero
        move $t2, $zero
        la $t3, v
        li $t4, 5
        li $t5, 30
        bge $t0, $t5, endwhile
while:
        sll $t6, $t0, 2
        lw $t6, 0($t6)
        ble $t6, $t4, endif
if:
        addu $t2, $t2, $t6
        addiu $t1, $t1, 1
endif:
        addiu $t0, $t0, 1
        b while
endwhile:
```

Quina és la resposta correcta?

a) La instrucció `jalr` s'executa 1 vegada

b) La instrucció `jalr` s'executa 2 vegades

c) La instrucció `jalr` s'executa 3 vegades

d) La instrucció `jalr` s'executa infinites vegades (el programa entra en un bucle infinit i no acaba)

e) Sense saber el contingut d'altres posicions de memòria no es pot saber quants cops s'executa la instrucció `jalr`.

<3.4> ① **3.21.** Escriu en C i en assemblador MIPS una funció tal que, donats dos valors enters de 32 bits passats per valor com a entrada, retorni el mínim dels dos.

<3.4> ② **3.22.** Tradueix a assemblador MIPS la següent funció recursiva:

```
int pellnumber(int n) {
     if (n < 2)
             return n;
     else
             return 2*pellnumber(n-1) + pellnumber(n-2);
}
```

<3.4> ③ **3.23.** Tradueix la següent subrutina en C a assemblador MIPS. Recorda que la traducció de la crida a una funció recursiva segueix exactament les mateixes regles que la crida a una funció qualsevol.

```
int maxv(int vector[], int length)
{
     int temp;

     if (length == 1)
             temp = vector[0];
     else
     {
             temp = maxv(vector, length-1)
             if  (vector[length-1] > temp)
                    temp = vector[length-1];
     }
     return temp;
}
```

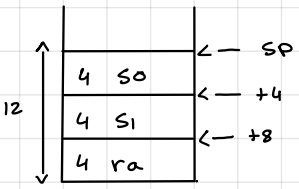
<3.4> ③ **3.24.** Tradueix a assemblador MIPS la següent subrutina:

```
int strlen(char s[])
{
     int i=0;

     while (s[i] != '\0')        /* Nota: '\0' equival a 0 */
             i++;
     return i;
}
```

3. 22

Que registros guardare?

$ra, $a0 → $s0
$v0 → $s1



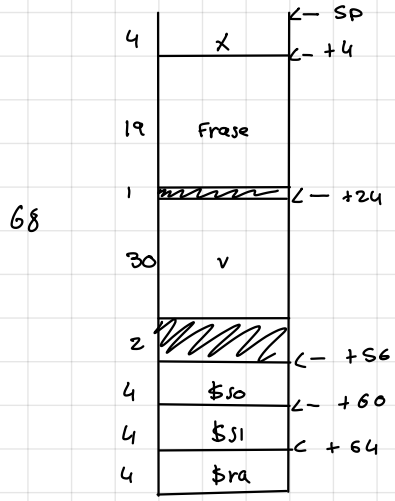
$ra, seguro llama a otra Función
$a0 (n), porque me llega como parametro de la Función
$v0, resultado de la 1ra Función porque hay 2 llamadas seguidas

Codigo MIPS:

```
pellnum: addiu $sp, $sp, -12
         sw $s0, 0($sp)
         sw $s1, 4($sp)
         sw $ra, 8($sp)
         move $s0, $a0
         li $t2, 2
         bge $s0, $t2, else
if:
         move $v0, $s0
         b end
else:
         addiu $a0, $s0, -1
         jal pellnumber
         sll $s1, $v0, 1
         addiu $a0, $s0, -2
         jal pellnumber
         addu $v0, $v0, $s1
end:
         lw $s0, 0($sp)
         lw $s1, 4($sp)
         lw $ra, 8($sp)
         addiu $sp, $sp, 12
         jr $ra
```

Pregunta 5 parcial abril 2022

a) BA?



b)

```
addiu $sp , $sp , -68
sw    $s0 , 56 ($sp)
sw    $s1 , 60 ($sp)
sw    $ra, 64 ($sp)
```

c)

```
addu $t3, $a1, $a2
lb $t4, 0($t3)
lw $t5, 0($t1)
addu $t5, $t5, $sp
sb $t4, 4($t5)
```

<3.4> ③ **3.25.** Donat el següent fragment de programa escrit en C:

```
void examen (char vi[], char vo[])
{
     int i=0;

     while (vi[i] != '\0')        /* Nota: '\0' = 0*/
     {
              if (v[i] != '  ')
                     vo[i] = vi[i] - 32;
              else
                     vo[i] = vi[i];
              i++;
     }
     vo[i] = '\0';
}

main()
{
     examen (vect1, vect2);
}
```

a) Sabent que *vect1* i *vect2* són variables globals, tradueix a assemblador MIPS la funció *main*.

b) Tradueix a assemblador MIPS la subrutina *examen*.

<3.4> ③ **3.26.** Donades les següents declaracions de variables i funcions en C, tradueix a assemblador MIPS la subrutina *subr3*.

```
char V[7];
int s3(char v1[], int tf, int nf, char c);

int subr3(int param)
{
     int ret, tam = 7;

     ret = s3(V, tam, param, 'D');
     return ret + tam + param;
}
```

<3.4> ③ **3.27.** Donades les següents declaracions en C:

```
int proc(int vector[], int *x1);

char exam (int v[], int a, int b)
{
     int i, *p;
     ...            /* Sentències dels apartats del problema */
}
```

a) Tradueix a assemblador MIPS el següent fragment de codi, que es troba dins la funció *exam*, suposant que les variables locals *i* i *p* ocupen $t0 i $t1.

```
p = &v[a+5];
i = proc(v,p);
```

b) Tradueix a MIPS la següent sentència condicional, que també es troba dins la funció *exam*, suposant que la variable local *i* ocupa el registre $t0:

```
if ((i > 0) && (i < 10))
        i = 10;
else
        i = 0;
```

c) Tradueix a assemblador MIPS el següent fragment de codi, que es troba dins la funció *exam*, suposant que les variables locals *i* i *p* ocupen $t0 i $t1:

```
p = v + b;            /* Atenció: aritmètica de punters! */
for (i=0; i<10; i++)
{
        *p = 0;
        p++;
}
```

<3.4> ③ **3.28.** Donades les següents declaracions, tradueix a assemblador MIPS la funció *s1*:

```
int s2 (int c, long long *d, long long w[]);
int s1 (int x, long long v[], int *p)
{
     long long *k;
     k = &v[x];
     x = s2(*p, k, v);
     return *p + x;
}
```

<3.4> ③ **3.29.** Donades les següents accions en C:

```
void swap(int *a, int *b)
{
     int temp;

     temp = *a;
     *a = *b;
     *b = temp;
}

void examen(int vector[], char c, int j)
{
     swap(&vector[j+1], &vector[j]);
}
```

a) Tradueix a assemblador MIPS la subrutina *swap.*

b) Tradueix a assemblador MIPS la subrutina *examen.*

<3.4> ③ **3.30.** Tradueix a assemblador MIPS la funció *func1*.

```
unsigned int func1(unsigned int v, unsigned int *p)
{
     if (v==0)
              *p = 0;
     else
              *p = func1(func1(v-1, p), p+1) - 1;

     return *p;
}
```

<3.4> ③ **3.31.** Tradueix a assemblador MIPS la funció *func2*.

```
char func2(short vec[], short a, int *b)
{
     short loc;
     if ((*b) > 3)
              loc = a;
     else
     {
              loc = vec[*b];
              *b = func2(vec+1, loc+1, b+1);
     }
     return loc;
}
```

<3.4> ③ **3.32.** Sabent que la funció f té la següent capçalera en C:

```
     int     f (int n);
```

i la següent traducció a assemblador MIPS:

```
f:   move    $v0, $zero
buc: andi    $t1, $a0, 1
     addiu   $v0, $v0, $t1
     srl     $a0, $a0, 1
     bne     $a0, $zero, buc
     li      $t0, 32
     subu    $v0, $t0, $v0
     jr      $ra
```

a) Quin valor retorna la funció f per a n=0x6789ABCD ?

b) Explica textualment què és el que calcula la funció f.

<3.5> ① **3.33.** Donades les següents declaracions, tradueix a assemblador MIPS la funció *subr1*.

```
int subr2(int a, char *b, char c, int d);

int subr1(char v[], int e2, int *var, char p)
{
     int valor;
     valor = subr2(*var, &v[e2], p, e2);
     return valor + e2;
}
```

<3.5> ① **3.34.** Donades les següents declaracions de variables globals i de la funció A:

```
char x;
int z;
char w[20];

char A(char i, int k, char *v) {
    int r[10];
    char c;
    c = v[3];
    r[k] = i;
    return c;
}
```

a) Traduir a assemblador la sentència:

```
x = A(x,z,w);
```

b) Dibuixar el bloc d'activació d'una crida a la funció A

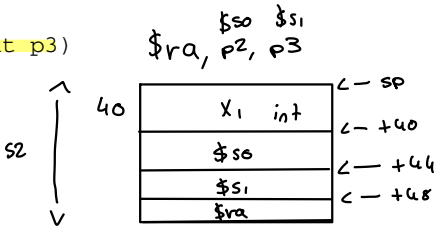
c) Traduir a assemblador la funció A:

<3.5> ② **3.35.** Tradueix a assemblador MIPS la funció *A*.

```
int B(short *x, short y[], int z, char m);
int A(int a[], short b[], char c)
{
    int p1=0;
    short p2[5];

    p1 = B(b, p2, p1, 'A');
    p2[p1+a[2]] = b[p1];
    if (c=='z')
            p1 = p1 - 3;
    else
            p1 = 2;
    return p1;
}
```

<3.5> ③ **3.36.** Tradueix a assemblador la següent funció (escrivint punts suspensius per a les sentències desconegudes):

```
void sub(int p1[10], int p2, int p3)
{
    int x1[10], x2;
    ...
    x1[p2] = p1[x2] + p3;
    ...
    sub (x1, p2, p1[3]);
    ...
    x2 = p2 + p3;
    ...
}
```



```
sub:    addiu $sp, $sp, -52
        sw $s0, 40($sp)
        sw $s1, 44($sp)
        sw $ra, 48($sp)
        move $s0, $a1
        move $s1, $a2
        ...
        sll $t1, $t0, 2
        addu $t1, $a0, $t1
        lw $t1, 0($t1)
        addu $t1, $t1, $s1
        sll $t2, $s0, 2
        addu $t2, $sp, $t2
        sw $t1, 0($t2)
        ...
        lw $a2, 12($a0)
        move $a0, $sp
        move $a1, $s0
        jal sub
        ...
        addu $t0, $s0, $s1
        ...
        lw $s0, 40($sp)
        lw $s1, 44($sp)
        lw $ra, 48($sp)
        addiu $sp, $sp, 52
        jr $ra
```

Tema 4. Matrius

<4.1> ① **4.1.** Donades les següents declaracions en C, on `NF` i `NC` són constants:

```
int mat[NF][NC];
int f() {
     int i, j;
     ... /* Aquí va el codi de cada apartat */
}
```

Escriu el codi MIPS pertanyent a la funció *f* per calcular les adreces dels següents elements fent servir el mínim nombre d'instruccions, i deixant el resultat en $v0. Suposem que *i* i *j* estan emmagatzemades en $t0 i $t1, respectivament.

a) `mat[3][11]`

b) `mat[i][11]`

c) `mat[3][j]`

d) `mat[i][j]`

e) `mat[i+5][j-1]`

f) `mat[i*10+4][6]`

<4.1> ① **4.2.** Tradueix a llenguatge assemblador MIPS la següent funció, on `N` és una constant:

```
void func(int mat[][N], int i, int j) {
     if (i>j)
              mat[i][j] = mat[j][i];
}
```

<4.2> ① **4.3.** Donada la següent declaració d'una variable global (on N és una constant):

```
int M[N][N];
```

Tradueix a assemblador MIPS el següent fragment de codi en C, usant el mínim nombre d'instruccions, i suposant que les variables *i* i *suma* ocupen els registres $t0, $t1:

```
suma += M[i][i+1] - M[i+1][i];
```

4.1

a) @mat[3][11] —> @mat +12NC + 44
b) @mat[i][11] —> @mat + (i*NC + 11)*4
c) @mat[3][j] —> @mat + (3*NC + j)*4
d) @mat[i][j] —>@mat + (i*NC + j)*4
e) @mat[i + 5][j - 1] —> @mat[i][j] + (5*NC-1)*4
f) @mat[i*10+4][6]

d) la $t7, mat
li $t2, NC
mult $t0, $t2
mflo $t2 #t2 <— i*NC
addu $t2, $t2, $t1 #t2 <— i*NC+j
sll $t2, $t2, 2 #t2 <— (i*NC+j)*4
addu $t2, $t2, $t7 #t2 <—mat +(i*NC+j)*4
lw $v0, 0($t2)

e) d)
lw $v0, 20*NC-4($t2)

4.3

la $t7, M
li $t2, N
mult $t2, $t1
mflo $t2 #t2 <— i*N
addu $t2, $t2, $t1 #t2 <— i*N+i
sll $t2, $t2, 2 #t2 <— (i*N+i)*4
addu $t2, $t2, $t7 #t2 <—mat+ (i*N+i)*4
lw $t3, 4($t2) #t3 <— @M[i][i+1]
lw $t4, 4*N($t2) #t4 <— @M[i+1][i]
subu $t3, $t3, $t4
addu $t0, $t0, $t3

Cuando nos movemos por columnas solo hace falta sumar el desplazamiento * tamaño.
Però las filas tenemos que sumar el (desplazamiento*nº columnas)*tamaño

4.4

lb $t3, -3*10+5($t0)

<4.1> ① **4.4.** Sigui la següent declaració en C:

```
char mat[6][10];
```

Si suposem que el registre $t0 és un punter que apunta a l'element `mat[i][j]`. És possible modificar $t0 amb una sola instrucció perquè apunti a `mat[i-3][j+5]`? Si la resposta és que sí, escriu-la.

<4.2> ① **4.5.** Sigui la següent declaració de la variable global *mat*:

```
long long mat[10][50];
```

Suposant que inicialment $t0 conté l'adreça de l'element `mat[4][45]`, a quin element de la matriu *mat* apunta $t0 després d'executar la següent instrucció?

```
addiu   $t0, $t0, 1384
```

<4.1> ① **4.6.** Donades les següents declaracions:

```
void f(int n) {
     int j;
     int b[8][10];
     ...
     /* Aquí va la sentència incògnita */
     ...
}
```

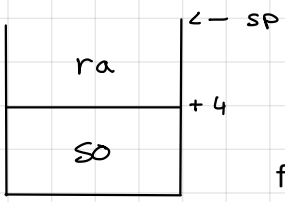
El següent fragment de codi correspon a la traducció a assemblador MIPS de la *sentència incògnita* que apareix al cos de la funció anterior. Determineu quina és aquesta sentència i expresseu-la en C, tenint en compte que $t0 conté la variable *j*.

```
     move    $t0, $zero
bucle:
     slti    $t1, $t0, 10
     beq     $t1, $zero, final
     li      $t1, 10
     mult    $t1, $a0            # $hi:$lo <- $t1 * $a0
     mflo    $t1                 # $t1 <- $lo
     addu    $t1, $t1, $t0
     sll     $t1, $t1, 2
     addu    $t1, $t1, $sp
     sw      $zero, 0($t1)
     addiu   $t0, $t0, 1
     b       bucle
final:
```

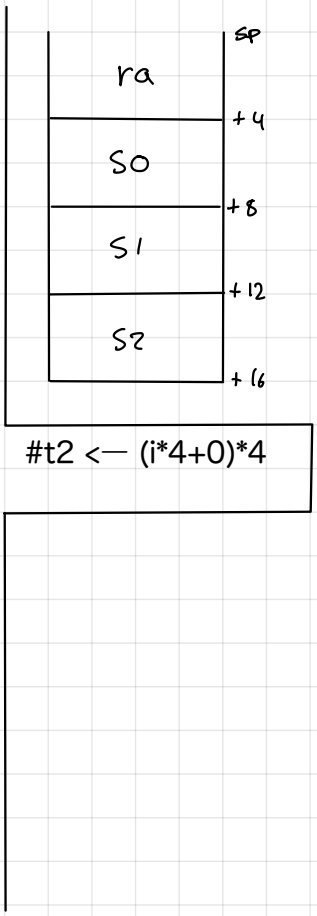
<4.2> ① **4.7.** Donades les següents declaracions en C:

```
int mati[5][4];

void func(int veci[4]) {
int j;
     for (j=0; j<4; j++)
             veci[j] = j;
}

void main() {
int i;
     for (i=0; i<5; i++)
             func(&mati[i][0]);
}
```

4.7




```
main:
        addiu $sp, $sp, -8
        sw $ra, 0($sp)
        sw $s0, 4($sp)
        move $s0, $zero
for:
        li $t1, 5
        bge $t0, $t1, end
        la $t7, mat
        sll $t2, $t0, 4
        addu $a0, $t2, $t7
        jal func
        addiu $t0, $t0, 1
        b for
end:
        lw $ra, 0($sp)
        lw $s0, 4($sp)
        addiu $sp, $sp, 8
        jr $ra
```




```
main:
        PRÓLOGO
        move $s0, $zero
        li $s1, 5
        la $s2, mat
for:    bge $s0, $s1 end
        move $a0, $s2
        jal func
        addiu $s0, $s0, 1
        addiu $s2, $s2, 16
        b for
end:
        EPÍLOGO
        jr $ra
```

a) Tradueix a MIPS la subrutina *func* utilizant accés seqüencial al vector *veci* i suposant que la variable *j* ocupa el registre $t0.

b) Tradueix a MIPS la subrutina *main* utilitzant accés seqüencial a la matriu *mati*. Fes atenció a quins registres fas servir per guardar la variable *i* i el punter.

<4.2> ① **4.8.** Donades les següents declaracions en C (on `N` és una constant):

```
void func(int A[N][N]) {
     int i, j, suma=0;
              ... /* aquí va la sentència de cada apartat */
     }
```

Tradueix a MIPS les següents sentències utilitzant la tècnica d'accés seqüencial per als accessos a la matriu A, suposant que pertanyen a la funció *func*, i que les variables *i*, *j*, i *suma* ocupen els registres $t0, $t1, $t2:

a)
```
for (i=0; i<N; i++)
         suma += A[3][i];
```

b)
```
for (i=0; i<N; i++)
         suma += A[i][4];
```

c)
```
for (i=0; i<N; i++)
         suma += A[i][i];
```

d)
```
for (i=0; i<N; i+=3)        /* Atenció: la i va de 3 en 3 */
         suma += A[i][N-1-i];
```

<4.2> ① **4.9.** Donades les següents declaracions en C:

```
#define N 100
char vecchar[N];
int matint[N][N];
long long matlong[N][N];

void func() {
     int i;
     ... /* aquí va la sentència de cada apartat */
}
```

Tradueix a MIPS les següents sentències utilizant accés seqüencial, suposant que pertanyen a la funció *func*, i que la variable *i* ocupa el registre $t0:

a)
```
for (i=1; i<N; i+=2)        /* Atenció, la i va de 2 en 2 */
         vecchar[i] = N-i;
```

b)
```
for (i=0; i<N; i++)
         matint[i][0] = 0;
```

c)
```
i=4;
while (matint[3][i] != 0) {
         matint[3][i]--;
         i+=3;
}
```

d)
```
for (i=N-1; i>=0; i--)
         matlong[i][4] = i;  /* Atenció a la part alta! */
```

<4.2> ② **4.10.** Siguin les següents declaracions en C:

```
int z[5][3];
int vector[4];

void func(int i, int j) {
     int k;
     ... (aquí van les sentències dels apartats)
}
```

a) Tradueix a MIPS de la següent sentència, pertanyent a la funció *func*.

```
vector[i] = z[4][j] ;
```

b) El codi assemblador MIPS que es mostra més avall s'ha obtingut al traduir la següent funció, pertanyent al cos de la funció *func*, implementant la tècnica d'accés seqüencial.

```
for (k=0; k<5; k++)
        vector[2] += z[k][2];
```

Completa les quatre declaracions inicials de constants per tal que la traducció sigui correcta:

```
        .set offset_vector, ?
        .set offset_z, ?
        .set max_dist, ?
        .set stride, ?

        .text
func:
        la     $t2, vector + offset_vector
        la     $t3, z + offset_z
        addiu  $t6, $t3, max_dist
for:
        beq    $t3, $t6, fifor
        lw     $t4, 0($t2)
        lw     $t5, 0($t3)
        addu   $t4, $t4, $t5
        sw     $t4, 0($t2)
        addiu  $t3, $t3, stride
        b      for
fifor:
        jr     $ra
```

<4.2> ② **4.11.** Donades les següents declaracions en C:

```
#define N 10
int mat[N][N];
```

Tradueix a assemblador MIPS el codi següent fent servir la tècnica d'accés seqüencial, usant un sol punter.

```
i=0;
do {
     mat[i][i] = mat[i][i+1] - mat[i+1][i];
     i += 2;
} while (i<N);
```

4.11

```
        la $t7, mat
        li $t1, N           #t1 <— 10
        li $t0, 0
do:     mult $t0, $t1
        mflo $t2            #t2 <— i*10
        addu $t2, $t2, $t0  #t2 <— (i*10+i)
        sll $t2, $t2, 2     #t2 <— (i*10+i)*4
        addu $t2, $t2, $t7  #t2 <— mat+ (i*10+i)*4
        lw $t3, 4($t2)
        lw $t4, 4*N($t2)
        subu $t5, $t3, $t4
        sw $t5, 0($t2)
        addiu $t0, $t0, 2
        blt $t0, $t1, do
```

```
        move $t0, $zero
        la $t2, mat
        li $t1, N
do:     lw $t4, 4($t2)
        lw $t5, 4*N($t2)
        subu $t4, $t4, $t5
        sw $t4, 0($t2)
        addiu $t2, $t2, 88
        addiu $t0, $t0, 2
        blt $t0, $t1, do
```

<4.2> ②

4.12. Donades les següents declaracions en C:

```
void examen (short p1[100], int p2) {
     int i;
     short *p;
     ... /* aqui van les sentències del cos de la subrutina */
}
```

a) Tradueix a llenguatge C el següent fragment de codi assemblador MIPS, amb una única sentència en C, sabent que forma part del cos de la subrutina *examen*.

```
addiu    $t2, $a1, 3
sll      $t2, $t2, 2
addu     $t2, $a0, $t2
sw       $zero, 0($t2)
```

b) El següent fragment incomplet de codi, escrit en C i traduït al costat a assemblador MIPS, pertany al cos de la subrutina *examen*. Aquest codi utilitza la tècnica d'accés seqüencial per recórrer el vector *p1* inicialitzant tots els elements amb el valor zero. Completa les 3 sentències que falten en C, així com les corresponents línies en assemblador, suposant que les variables locals *i* i *p* es guarden als registres $t0, $t1.

Codi en C

```
p = [          ] ;

for (i=0; i<100; i++)
{
     [          ] ;
     [          ] ;
}
```

Codi equivalent en MIPS

```
     [          ]
     move     $t0, $zero    ; i=0
     li       $t2, 100
bucle:
     bge      $t0, $t2, fibucle
     [          ]
     addiu    $t0, $t0, 1   ; i++
     b        bucle
fibucle:
```

<4.2> ②

4.13. La següent funció en C implementa un recorregut seqüencial de la variable *matriu* amb el punter *p*. Suposem que N i M són constants.

```
int matriu[N][M];
void func() {
     int i, *p;

     p = matriu;
     for (i=0; i<N; i++) {
             *p = i;
             p = p + M + 1;
     }
}
```

a) Enumera els elements de la matriu que s'escriuen i indica quin valor s'assigna a cada un.

b) Tradueix a MIPS el bucle anterior, usant la tècnica d'accés seqüencial i suposant que *i* i *p* ocupen els registres $t0, $t1.

<4.2> ③ **4.14.** La funció *fila_dispersa* escriu els elements de la fila *i* de la matriu dispersa[1] representada pels vectors *A*, *IA* i *JA* (que rep com a paràmetres), en les corresponents posicions de la matriu *mat* (que és una variable global):

```
int mat[N][M];
void fila_dispersa(int *A, int *IA, int *JA, int i) {
     int k;
     for (k=IA[i]; k<IA[i+1]; k++)
              mat[i][JA[k]] = A[k];
}
```

a) Dels següents accessos a memòria que fa aquest bucle, ¿quins es poden traduir fent servir la tècnica d'accés seqüencial, i quins sols es poden traduir amb accés aleatori?

```
IA[i], IA[i+1], JA[k], mat[i][JA[k]], A[k]
```

b) Tradueix la funció *fila_dispersa* usant la tècnica d'accés seqüencial per als recorreguts que ho permetin, suposant que *k* es guarda al registre $t0.

---

1. En algunes aplicacions de la informàtica es fan servir matrius molt grans on la majoria dels elements valen zero. Se les anomena matrius disperses. Representar-les com les matrius convencionals (denses) pot significar un gran cost en espai d'emmagatzematge. Per aquesta raó es fan servir mètodes de compressió basats en guardar solament els elements no nuls de la matriu original.

   Suposem que es vol guardar la matriu *mat*[N][M], que sabem que és dispersa. En un vector *A* es guarden els elements no nuls, en el mateix ordre en què apareixen a la matriu *mat*, és a dir per files. La llargada d'aquest vector és igual al nombre d'elements no nuls de *mat*. En un altre vector *JA,* de la mateixa llargària, es guarda la columna a la qual pertany cada element. I finalment, en un altre vector *IA*, de N+1 elements, es guarda l'índex del vector *A* on es troba emmagatzemat el primer element no nul de cada fila de *mat*. L'últim element (*IA*[N]) conté el nombre total d'elements no nuls de *mat* (és a dir la llargada de *A* i *JA*). Així doncs, podem dir que, per a cada fila *i* de *mat*, els seu elements no nuls es troben guardats al vector *A*, en posicions consecutives entre la posició *IA*[*i*] i la posició *IA*[*i+1*] (no inclosa).

Tema 5. Aritmètica d'enters i coma flotant

<5.1> ① **5.1.** Converteix a decimal els números 0x0D34 i 0xBA1D

a) Suposant que representen números naturals

b) Suposant que representen números enters en complement a 2

c) Suposant que representen números enters en signe-magnitud

<5.1> ① **5.2.** Siguin els números: A=0x0D34, B=0xDD17, C=0xBA1D, D=0x3617

a) Quant val, en hexadecimal, la suma A+B si suposem que representen números naturals de 16 bits?

b) Ídem per a la suma C+D

c) Quant val, en hexadecimal, la suma A+B si suposem que són enters en format signe-magnitud de 16 bits?

d) Ídem per a la suma C+D

<5.1> ① **5.3.** Siguin els números:A=0xBA7C, B=0x241A, C=0xAADF, D=0x47BE

a) Quant val, en hexadecimal, la resta A-B si suposem que representen números naturals de 16 bits?

b) Ídem per a la resta C-D

c) Quant val, en hexadecimal, la resta A-B si suposem que són enters en format signe-magnitud de 16 bits?

d) Ídem per a la resta C-D

<5.1> ① **5.4.** Dadas las siguientes operaciones con naturales, codifica los operandos en binario con 8 bits, y realiza la operación en binario. Convierte el resultado a base diez, e indica si la operación ha producido desbordamiento (carry):

a) `200 + 50`

b) `25 + 34`

c) `254 + 3`

d) `128 - 20`

<5.1> ① **5.5.** Dadas las siguientes operaciones con enteros, codifica los operandos en Ca2 con 8 bits, y realiza la operación en binario. Halla el valor implícito del resultado interpretado como entero en Ca2, expresándolo en decimal, e indica si la operación ha producido desbordamiento entero (overflow):

a) `(-25) + (-34)`

b) `(-25) - (-34)`

c) `(+127) - (-20)`

d) `(-128) - (-20)`

<5.1> ② **5.6.** La condición de desbordamiento (overflow) de la suma de dos números naturales *a* y *b* de 32 bits es fácil de comprobar habiendo realizado la suma *s*, pues en ese caso el valor de s (incorrecto) resulta ser menor que cualquiera de los dos sumandos. En efecto, debido al desbordamiento se cumple que $s = a+b-2^{32}$. Puesto que se cumple $b < 2^{32}$, se cumple también que $a+b-2^{32} < a$, es decir $s < a$ (análogamente se demuestra $s < b$). Basándote en esta propiedad, haz un programa que, dadas dos variables naturales de 32 bits almacenadas en $t1 y $t2, calcule si su suma ($t0 = $t1 + $t2), una vez realizada, ha producido desbordamiento (carry), en cuyo caso debe guardar un 1 en $t3, o bien un 0 en caso contrario. El programa no debe contener ninguna instrucción de salto.

<5.1> ③ **5.7.** La condición de desbordamiento (overflow) de la suma de dos números naturales de 32 bits *a* y *b* también se puede calcular antes de realizar la suma: la condición es $a+b > 2^{32}-1$. Lo cual equivale a: $a+b > a+\overline{a}$. Lo cual equivale a: $b > \overline{a}$. Basándote en esta propiedad, haz un programa que, dadas dos variables naturales de 32 bits almacenadas en $t1 y $t2, calcule anticipadamente si su suma produciría desbordamiento (pero sin calcularla), en cuyo caso debe guardar un 1 en $t3, o bien un 0 en caso contrario. El programa no debe contener ninguna instrucción de salto.

<5.1> ③ **5.8.** La condición de desbordamiento (overflow) de dos números enteros consiste en determinar si ambos son del mismo signo y además la suma es de signo opuesto. Basándote en esta propiedad, haz un programa que, dadas dos variables enteras de 32 bits almacenadas en $t1 y $t2, calcule si su suma ($t0 = $t1 + $t2), una vez realizada, ha producido desbordamiento, en cuyo caso debe guardar un 1 en $t3, o bien un 0 en caso contrario. El programa no debe contener ninguna instrucción de salto.

<5.1> ② **5.9.** Donada la següent declaració en C:

```
long long x, y;
```

Tradueix a MIPS les següents sentències: suma, resta i comparació en doble precisió:

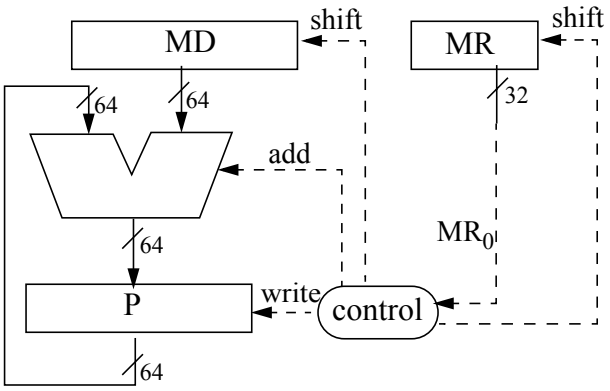
a) `x = x + y;`

b) `x = x - y;`

c) `if (x > y)  x = 1;`

<5.2> ①

5.10. Donat el següent diagrama del multiplicador seqüencial de nombres naturals (X*Y de 32 bits) estudiat a classe, el qual calcula el Producte amb 64 bits, completa l'algorisme iteratiu que en descriu el funcionament cicle a cicle:




```
MD63:32 =     ;
MD31:0  =     :
MR      =     ;
P       =     ;

for (i=1;      ; i++)
{









}
```

<5.2> ②

5.11. Suposant el circuit del problema 5.10, descriu els passos necessaris per a la multiplicació dels nombres naturals de 6 bits X (multiplicand) i Y (multiplicador), calculant en cada pas el valor dels registres P, `MD` i `MR`, en binari:

a) Suposant X=101000, Y=010011

iteració	Passos	P	MD	MR
v.inicial		`000000 000000`	`000000 101000`	`010011`
1				
2				
3				
4				
5				
6				

b) Suposant X=110110, Y=000100

iteració	Passos	P	MD	MR
v.inicial		000000 000000	000000 110110	000100
1				
2				
3				
4				
5				
6				

<5.2> ③ **5.12.** Seguint el mateix algorisme que el circuit multiplicador estudiat (veure problema 5.10), i sense usar cap instrucció de multiplicació, escriu un programa en assemblador MIPS que calculi el producte de dos números naturals de 32 bits guardats als registres $t0 i $t2. El resultat de 64 bits s'ha de guardar als registres $t3 (la part baixa) i $t4 (la part alta). Podeu completar el següent algorisme i traduir-lo:

```
MD63:32 =  ;  //$t1
MD31:0  =  ;  //$t0
MR      =  ;  //$t2
P       =  ;  //$t4, $t3

for (i=1;       ; i++)
{


}
```

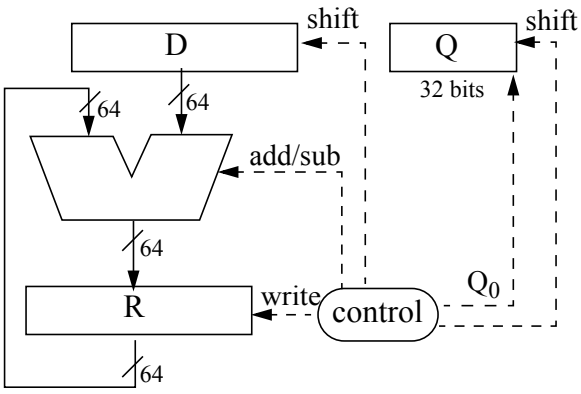
<5.2> ② **5.13.** Siguin `x,y,z` variables de 32 bits emmagatzemades a $t1, $t2 i $t3 respectivament. Escriu un programa en MIPS que calculi `z=x*y` i que escrigui en $t0 el valor 0 si el resultat `z` és correcte o bé el valor 1 si s'ha produït desbordament. El programa no ha de fer servir cap instrucció de salt. Suposarem les següents declaracions:

a) `unsigned int x, y, z;`

b) `int x, y, z;`

<5.3> ①

5.14. Donat el següent diagrama que representa el divisor seqüencial de nombres naturals de 32 bits amb restauració estudiat a classe i que realitza la divisió X/Y, calculant alhora el quocient i el residu, completa l'algorisme iteratiu que en descriu el funcionament cicle a cicle:




```
D63:32 =      ;
D31:0  =      ;
Q      =      ;
R63:32 =      ;
R31:0  =      ;
for (i=1;       ; i++)
{




}
```

<5.3> ②

5.15. Suposant el circuit del problema 5.14, descriu els passos necessaris per a la divisió dels nombres naturals de 6 bits X (dividend) entre Y (divisor), calculant en cada pas el valor dels registres `R`, `D` i `Q`, en binari:

a) Suposant X=101000, Y=010011

iteració	Passos	Q (Quocient)	D (Divisor)	R (Dividend)
v. inicial		`000000`	`010011 000000`	`000000 101000`
1				
2				
3				
4				
5				
6				

b) Suposant X=010101, Y=100100

iteració	Passos	Q (Quocient)	D (Divisor)	R (Dividend)
v.inicial		000000	100100 000000	000000 010101
1				
2				
3				
4				
5				
6				

<5.3> ③ **5.16.** Seguint el mateix algorisme del circuit divisor estudiat (veure problema 5.14), i sense usar cap instrucció de multiplicació ni divisió, escriu un programa en assemblador MIPS que calculi el quocient i el residu de la divisió de dos números naturals de 32 bits guardats als registres $t3 i $t1. El quocient s'ha de guardar al registre $t2 i el residu al registre $t3. Podeu completar el següent algorisme i traduir-lo:

```
D31:0  =  ; //$t0
D63:32 =  ; //$t1
Q      =  ; //$t2
R31:0  =  ; //$t3
R63:32 =  ; //$t4
for (i=1;     ; i++)
{


}
```

<5.3> ① **5.17.** Completeu el contingut que han de tenir els registres operands de les següents instruccions de forma que esdevinguin coherents després de la seva execució. S'ha de mirar l'execució de cada instrucció de forma individual.

a) `mult $t1, $t2`

`$hi = 0xFFFFFFFF  $lo = 0xFFFFFFFA`

`$t1 = 0x00000002  $t2 =` [ ]

b) `div $t1, $t2`

`$hi =` [ ] `$lo =` [ ]

`$t1 = 0x00000005  $t2 = 0xFFFFFFFE`

<5.3> ① **5.18.** Indica en qué casos concretos se puede producir overflow después de ejecutar una división entre dos enteros en complemento a 2.

<5.3> ② **5.19.** Un determinat processador no té la instrucció de multiplicació en el seu joc d'instruccions perquè no disposa d'un multiplicador d'enters. En cada multiplicació d'enters, el processador realitza una crida a una subrutina que executa un total de 200 instruccions. Volem estudiar la conveniencia d'afegir un multiplicador d'enters al disseny del processador, per fer servir una única instrucció de multiplicació.

Per avaluar aquesta modificació del disseny utilitzem un conjunt de programes de prova on substituirem les crides a la funció de multiplicar per la nova instrucció de multiplicació. Hem mesurat experimentalment que, un cop fetes les substitucions, un 0,5% de les instruccions executades son de multiplicació, les quals tenen un CPI de 40, mentre que la resta tenen en promig un CPI de 3 (també les instruccions de la subrutina de multiplicació):

a) Si no hi ha cap més canvi en el disseny, quin speedup podem esperar del nou processador?

b) Desgraciadament, ens adonem que el nou multiplicador afecta al temps de cicle de rellotge, i ens obligarà a reduir la freqüència, que inicialment era de 2 GHz. Quina seria la mínima freqüència de rellotge acceptable per al nou processador per tal que no resulti més lent que l'original?

<5.4> ① **5.20.** En una arquitectura Von Neumann, els grups de bits no tenen un significat específic per ells mateixos. El seu significat depèn totalment de com es facin servir. Suposem els valors `A=0x24A60004` i `B=0xAFBF0000`. Contesta les següents preguntes fent diverses suposicions sobre A i B:

a) Si són nombres naturals, quin és el seu valor en decimal?

b) Si són nombres enters en complement a 2, quin és el seu valor en decimal?

c) Si són nombres en coma flotant de simple precisió, quin és el seu valor en decimal?

d) Si són instruccions MIPS, quina instrucció representen, en assemblador?

<5.4> ② **5.21.** Converteix a decimal els següents valors suposant que codifiquen nombres en el format de simple precisió. Un cop resolt, comprova els resultats amb el simulador MARS: obre la pestanya "Coproc 1", fes doble clic en un registre, i introdueix-hi el teu resultat en decimal, comprovaràs si correspon al valor hexadecimal original.

a) `0x3fb08000`

b) `0x80000000`

c) `0xb0fc0000`

d) `0xc2968000`

<5.4> ① **5.22.** Suposant que *f* i *g* són variables de tipus float (coma flotant en simple precisió), i estan emmagatzemades als registres $f12 i $f14 del coprocessador aritmètic CP1, tradueix a assemblador les següents sentències escrites en C. Tingues en compte que el llenguatge assemblador no admet nombres decimals fraccionaris en qualsevol context. Comprova que el teu codi es pot compilar sense errors i que dóna el resultat correcte en el simulador MARS:

a) `f = 3.14;`

b) `f = g;`

c) `f = g + 3.14;`

<5.4> ① **5.23.** Converteix els següents nombres decimals al format de coma flotant de simple precisió, expressant el resultat en hexadecimal. Determina en cada cas si es comet error per pèrdua de precisió en la seva representació i, en cas afirmatiu, calcula aquest error expressant-lo en decimal.

a) `340`

b) `44,4`

c) `-255,125`

d) `-10,75`

e) `6,5`

<5.4> ② **5.24.** Donats els números decimals `A=-1609,5` i `B=-938,8125`

a) Converteix-los al format de coma flotant de simple precisió, expressant el resultat en hexadecimal, i calcula l'error comès en la conversió, per pèrdua de precisió, expressat en decimal.

b) Converteix-los al format de coma flotant de doble precisió, expressant el resultat en hexadecimal, i calcula l'error comès en la conversió, per pèrdua de precisió, expressat en decimal.

<5.4> ① **5.25.** La següent taula conté una llista de números binaris que representen nombres reals en coma flotant, en el format IEEE 754 de simple precisió. Marca amb una **X** la casella corresponent al tipus de valor de cada un d'ells, d'acord amb la notació:

NRM = normalitzat; DNRM = denormalitzat; 0 = zero; INF = infinit

NAN = "Not a Number" (resultats d'operacions invàlides)

signe	exponent	mantissa	NRM	DNRM	0	INF	NAN
0	0000 0000	110 0010 0000 1110 1110 1011					
0	0000 0000	000 0000 0000 0000 0000 0000					
0	0010 0100	000 0000 0000 0000 0000 0000					
1	1111 1111	000 0000 0000 0000 0000 0000					
0	0010 0100	110 0010 0000 1110 1110 1011					
1	0000 0000	000 0000 0000 0000 0000 0000					
0	1111 1111	101 0001 0001 0000 1001 0100					

<5.4> ① **5.26.** En el format de coma flotant IEEE-754 de simple precisió

a) Quin és el major nombre positiu representable? (en notació exponencial i en hexadecimal).

b) Quin és el menor nombre positiu no nul representable en format normalitzat? (en notació exponencial i en hexadecimal).

c) Quin és el menor nombre positiu no nul representable en format no normalitzat? (en notació exponencial i en hexadecimal).

<5.5> ① **5.27.** Suposem que `$f2=0x417ac000`, `$f4=0x3f140000`, i que executem la instrucció MIPS: `add.s $f6,$f2,$f4`. Suposant que s'arrodoneix al més pròxim (al parell en el cas equidistant):

a) Calcular a mà, seguint l'algorisme de suma de nombres en coma flotant, el valor final de `$f6` en hexadecimal ?

b) Es produeix algun error de precisió en el resultat ?

c) Converteix a decimal el valor final de `$f6`.

<5.5> ② **5.28.** Suposem que `$f2=0xC076c000`, `$f4=0x3eca8000, i que` executem la instrucció: `add.s $f6,$f2,$f4`. Suposant que s'arrodoneix al més pròxim (al parell en el cas equidistant):

a) Calcular a mà, seguint l'algorisme de suma de nombres en coma flotant, el valor final de `$f6` en hexadecimal ?

b) Es produeix algun error de precisió en el resultat ?

c) Converteix a decimal el valor final de `$f6`.

<5.5> ② **5.29.** Suposem que `$f2=0x42000000` i `$f4=0x3d800000`, i que executem la instrucció: `mul.s $f6, $f2, $f4`. Suposant que s'arrodoneix al més pròxim (al parell en el cas equidistant) ¿quin és el valor final de `$f6` en hexadecimal?

<5.5> ① **5.30.** Tradueix a assemblador MIPS la subrutina *absdif*:

```
float absdif (float a, float b)
{
     if (a>b)
             return a-b;
     else
             return b-a;
}
```

<5.5> ② **5.31.** Donades les següents declaracions de funcions:

```
float mitjana (float p[]);

float variancia (float vec[])
{
     int i;
     float m, q;
     float vquadrats[100];
```

```
    for (i=0; i<100; i++)
            vquadrats[i] = vec[i] * vec[i];

    q = mitjana(vquadrats);
    m = mitjana(vec);
    return q - m * m;
}
```

a) ¿Quines variables locals serà convenient guardar en el bloc d'activació i quines és més convenient guardar en registres? Quins registres caldrà preservar a la pila abans de cada una de les crides a la funció *mitjana*?

b) Tradueix a assemblador MIPS la funció *variancia*.

<5.5> ③ **5.32.** Els processadors NV3x de NVIDIA usen un format de coma flotant de 16 bits anomenat "half", similar al de simple precisió, excepte que l'exponent té 5 bits (excés a 15) i la mantissa té 10 bits (amb bit ocult). Converteix els següents numeros decimals a la representació en coma flotant "half" binària, donant el resultat en hexadecimal.

a) A= $-1{,}278*10^3$, B= -3,90625

b) A = $2{,}3109375*10^1$, B = $6{,}391601562*10^{-1}$.

c) A = $6{,}18*10^2$, B = $5{,}796875*10^1$

d) A=$-1{,}6360*10^4$, B=$1{,}6360*10^4$, i C=$1{,}0*10^0$

e) A=$2{,}865625*10^1$, B=$4{,}140625*10^{-1}$, i C=$1{,}2140625*10^1$.

f) A=$1{,}5234375*10^{-1}$, B=$2{,}0703125*10^{-1}$, i C=$9{,}96875*10^1$.

g) A=$-2{,}7890625*10^1$, B=$-8{,}088*10^3$, i C=$1{,}0216*10^4$.

<5.5> ③ **5.33.** Suposem els números A i B en coma flotant, en el format "half" de 16 bits explicat al problema 5.32. Suposem que tenim un sumador amb un bit de guarda, un d'arrodoniment i un de "sticky", i que arrodonim al més pròxim (al parell en el cas equidistant).

a) Calcula a mà la suma A+B suposant que A=0xE4FE i que B=0xB640, seguint el mateix algorisme que usa el hardware i donant el resultat en hexadecimal i també en decimal. Quina és la precisió del resultat?

b) Ídem amb els números A =0x4DC7 i B = 0x391D.

<5.5> ③ **5.34.** Suposem els números A i B en coma flotant, en el format "half" de 16 bits explicat al problema 5.32. Suposem, també, que arrodonim al més pròxim (al parell en el cas equidistant).

a) Calcula a mà el producte A*B, suposant que A=0x45A9, i que B=0x484C, seguint el mateix algorisme que usa el hardware i donant el resultat en hexadecimal i també en decimal. Quina és la precisió del resultat?

b) Ídem amb els números A = 0x60D4 i B = 0x533E.

<5.6> ② **5.35.** Suposem els números A, B i C en coma flotant, en el format "half" de 16 bits explicat al problema 5.32. Suposem que tenim un sumador amb un bit de guarda, un d'arrodoniment i un de "sticky", i que arrodonim al més pròxim (al parell en el cas equidistant).

a) Suposant que A=0xF3FD, B=0x73FD, i C=0x3C00, calcula a mà (A + B) + C. Després calcula també A + (B + C). Dóna els resultats en hexadecimal i en decimal. Es pot afirmar que (A + B) + C = A + (B + C)?

b) Fes la mateixa comprovació amb els números A=0x4F2A, B=0x36A0, i C=0x4A12.

<5.6> ③ **5.36.** Suposem els números A, B i C en coma flotant, en el format "half" de 16 bits explicat al problema 5.32. Suposem que tenim un sumador amb un bit de guarda, un d'arrodoniment i un de "sticky", i que arrodonim al més pròxim (al parell en el cas equidistant).

a) Suposant que A=0x30E0, B=0x32A0, i C=0x563B, calcula a mà A * (B + C). Després calcula també (A * B) + (A * C). Dóna els resultats en hexadecimal i en decimal. Es pot afirmar que A * (B + C) = (A * B) + (A * C)?

b) Fes la mateixa comprovació amb els números A=0xCEF9, B=0xEFE6, i C=0x70FD.

<5.5> ③ **5.37.** Suposem els números A i B en coma flotant, en el format "half" de 16 bits explicat al problema 5.32. Suposem, també, que arrodonim al més pròxim (al parell en el cas equidistant).

a) Calcula a mà el quocient A/B, suposant que A=0xEA60, i que B=0xE000. Calcula la divisió, seguint el mateix algorisme que usa el hardware, donant el resultat en hexadecimal i també en decimal.

b) Ídem amb els números A =0xCD98 i B =0x6400.

<5.5> ③ **5.38.** Suposem els números A i B en coma flotant, en el format "half" de 16 bits explicat al problema 5.32. Suposem que tenim un sumador amb un bit de guarda, un d'arrodoniment i un de "sticky", i que arrodonim al més pròxim (al parell en el cas equidistant):

a) Escriu un programa en assemblador MIPS que calculi la suma A=A+B, suposant que A i B estan guardats en $t1 i $t2.

b) Escriu un programa en assemblador MIPS que calculi el producte A=A*B, suposant que A i B estan guardats en $t1 i $t2, i que indiqui si s'ha produït overflow o underflow.

Tema 6. Memòria Cache

<6.2> ① **6.1.** Considereu que la jerarquia de memòria d'un sistema computador està format per una memòria principal de 64 bytes i una memòria cache de 16 bytes, amb correspondència directa i on els blocs són de 4 bytes. La unitat d'adreçament de la memòria es el byte.

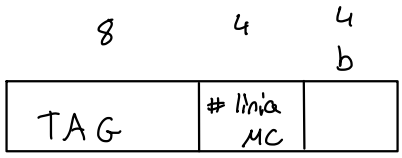
a) Donada l'adreça 0x39 de memòria principal indiqueu

- quin és l'índex de la memòria cache que li correspon
- quina és l'etiqueta que li correspon
- quin és el byte que li correspon dins el bloc

b) Considereu la següent seqüència de referències de lectura a blocs de memòria principal: 0, 1, 0, 3, 12, 10, 3, 7, 12, 15, 3. Indiqueu per a cada referència si es produeix un encert o una fallada a memòria cache.

<6.2> ① **6.2.** Disposem d'un processador de 16 bits (amb bus d'adreces de 16 bits) amb una memòria cache que té les següents característiques:

- Correspondència directa
- Mida total: 256 bytes
- Mida bloc: 16 bytes 2^b b = numero de bits
- Escriptura immediata sense assignació



a) Ompliu la següent taula a partir de la seqüència de referències donades.

tipus	adreça (hex)	etiqueta (hex)	índex MC (hex)	Encert/ Fallada	#bytes llegits MP	#bytes escrits. MP	lectura dades MC (Si/No)	escript. dades MC (Si/No)	
R	4534	45	3	F F	16 16	– –	Sí Sí	No	No
R	4568	45	6	F F	16 16	– –	Sí Sí	No	No
W	13A4	13	A	F F	– 16	2 –	No No	No	Sí
W	13A8	13	A	F A	– –	2 –	No No	No	Sí
R	3560	35	6	F F	16 16	– –	Sí Sí	No	No

0 39*
3 45
6 45 → 35
A 13* → 60*

Consejos: Siempre que hay escritura escribir MP Writhe though
Escritura i Fallo No leer MP sin asignación
Lectura i Fallo Siempre leer MP

tipus	adreça (hex)	etiqueta (hex)	índex MC (hex)	Encert/ Fallada	#bytes llegits MP	#bytes escrits. MP	lectura dades MC (Si/No)	escript. dades MC (Si/No)	
W	453C	45	3	A A	– –	2 –	No No	Sí	Sí
W	60A0	60	A	F F	– 16	2 16	No No	No	Sí
R	453C	45	3	A A	– –	– –	Sí Sí	No	No
W	3900	39	0	F F	– 16	2 –	No –	No	Sí
R	A238	A2	3	F F	16 16	– –	Sí Sí	No	No

* **b)** Ompliu ara la mateixa taula, suposant que la la MC té una política d'escriptura retardada amb assignació. Copy back + write alocate

c) Indiqueu per cada política:

- taxa de fallades
- número de bytes llegits d'MP
- número de bytes escrits a MP

<6.2> ② **6.3.** El siguiente programa multiplica una matriz `A(32x32)` por un vector `B(32)` produciendo como resultado un vector `C(32)`:

```
char A[32][32],B[32],C[32];
main() {
int i,j;
      ...
      for (i=0; i<32; i++)
         for (j=0; j<32; j++)
            C[i] = A[i][j] * B[j] + C[i]
}
```

Los elementos de `A`, `B` y `C` son bytes. Todos los elementos de `C` han sido previamente inicializados a cero. `A` está almacenada a partir de la dirección `0` de memoria (direcciones `0..1023`). `B` está almacenada justo a continuación de `A` (direcciones `1024..1055`) y `C` justo a continuación de `B` (direcciones `1056..1087`). Las variables `i,j` están almacenadas en registros del procesador.

El computador dispone de una memoria cache de correspondencia directa que almacena 4 bloques de 32 bytes cada uno, con escritura inmediata. Suponiendo que al inicializarse la ejecución del bucle anterior la cache no tiene ningún dato, calcula la tasa de aciertos de la memoria cache.

<6.2> ③ **6.4.** Se desea realizar una nueva versión de un procesador que dispone de una memoria cache de correspondencia directa con capacidad para almacenar 4 bloques de una palabra (de 32 bits). Debido a los avances tecnológicos, la memoria cache de

la nueva versión podrá tener el doble de capacidad que la cache de la anterior versión. Se barajan dos alternativas:

- Alternativa 1: doblar el número de bloques (es decir, 8 bloques de una palabra).
- Alternativa 2: doblar el tamaño de las líneas (es decir, 4 bloques de 2 palabras).

Se pide:

a) Una secuencia de referencias a palabras de memoria principal tal que la tasa de aciertos de la alternativa 1 sea mayor que la de la alternativa 2.

b) Una secuencia de referencias a palabras de memoria principal tal que la tasa de aciertos de la alternativa 2 sea mayor que la de la alternativa 1.

c) Razonar qué características de los programas se explota en mayor medida en cada una de las dos alternativas.

<6.3> ② **6.5.** Una possible mesura de rendiment d'una memòria cache es l'anomenada taxa de tràfic. Aquesta taxa es defineix com el quocient entre el nombre de bytes que es transfereixen des de/cap a la memòria principal en un sistema que disposa de cache i el nombre de bytes que es transfereixen des de/cap a la memòria principal en un sistema sense cache.

Suposem que totes les referències a memòria són de paraules de 32 bits. Es demana que calculeu la taxa de tràfic en funció de la taxa d'encerts (h), la mida del bloc en bytes (B), el percentatge d'escriptures (pe) i el percentatge de blocs reemplaçats que han estat modificats (pm) en els dos casos següents:

- escriptura immediata sense assignació
- escriptura retardada amb assignació

<6.3> ② **6.6.** Es vol definir la política d'escriptura de la memòria cache d'un determinat processador. Es consideren les alternatives: (1) escriptura immediata sense assignació i (2) escriptura retardada amb assignació.

Mitjançant simulació s'han obtingut les següents mesures:

- percentatge d'escriptures (pe): 20%
- percentatge de blocs modificats sobre el total de blocs reemplaçats (pm): 33.33%
- taxa d'encerts cas (1): 0.9
- taxa d'encerts cas (2): 0.85

El temps d'accés a memòria cache en cas d'encert (t_h) és de 10 ns. La lectura o escriptura d'un bloc de memòria principal (t_{block}) requereix 100 ns.

Es demana:

a) Calculeu el temps mitjà d'accés a memòria (t_{am}) en ambdues alternatives.

WT + WnA

100 accesos

20 escrituras = 10 = 200 ns

80 lecturas

72 hit = 720 ns

8 miss = 960 ns

total = 1880 ns / 100 = 18,8 ns

CB + WA

100 accesos

85 aciertos = 850 ns

15 Fallos

10 limpia x (th + tB + th) = 1200 ns

5 sucia x (th + 2tB + th) = 1100

total = 3150 ns

T_{am} = 31,5 ns

Formula = $h \cdot th + m(\%suc \cdot tsf_{suc} + \%miss\ tsf_{miss})$

b) Indiqueu quina alternativa seria la més ràpida per a un programa que només fes lectures.

<6.3> ③ **6.7.** Tenim una CPU amb una cache en què hem observat les característiques següents quan executa una col.lecció de programes representatius:

- CPI_{ideal} (CPI suposant que tots els accessos a memòria són encerts a la cache): 1.5 cicles/instr.
- Temps de cicle (t_c): 10 ns
- Nombre de referències per instrucció (nr): 1.6
- Cache d'instruccions i dades separades
- Cache de dades d'escriptura retardada amb assignació
- Les característiques de les dues caches són les següents:

Característica	Memòria cache	
	d'Instruccions	de Dades
Nombre de referències a memòria per instrucció (nr)	1	0.6
Percentatge d'escriptures per referència (pe)	-	40%
Percentatge de blocs modificats sobre tots els reemplaçats (pm)	-	20%
Taxa de fallades (m)	4%	10%
Penalització (t_p) en reemplaçar un bloc no modificat	10 cicles	15 cicles
Penalització (t_p) en reemplaçar un bloc modificat	-	20 cicles
Temps de servei en cas d'encert (t_h)	1cicle	1cicle

a) Quin serà el temps mitjà d'accés a memòria (t_{am}) en cicles?

b) I el temps mitjà d'execució d'una instrucció (t_{exe}) en ns?

<6.3> ③ **6.8.** Tenim una CPU amb una cache en què hem observat les característiques següents quan executa una col.lecció de programes representatius:

- CPI_{ideal} (CPI suposant que tots els accessos a memòria són encerts a la cache): 1.8 cicles/instr.
- Temps de cicle (t_c): 10 ns
- Nombre de referències per instrucció (nr): 1.4
- Mida del bloc: 8 bytes

6.7 a) $t_{amI} = 0,96 \cdot 1 + 0,04(1+10) = 1,44$ ciclos $= 14,4$ ns

$t_{amD} = 0,9 \cdot 1 + 0,1(0,8 \cdot 15 + 0,2 \cdot 20 + 1) = 2,7$ ciclos $= 27$ ns

b) $t_{am} = \dfrac{1\, t_{am_I} + 0,6\, t_{amD}}{1,6} = 19,125$ ns

c) $T_{exe} = I \cdot CPI \cdot T_c$

$CPI_I + CPI_M \rightarrow n_r \cdot m \cdot t_p$

$CPI_{M_I} = 1 \cdot 0,04 \cdot 10 = 0,4$ ciclos

$CPI_{M_D} = 0,6 \cdot 0,1 \cdot (0,2 \cdot 20 + 0,8 \cdot 15) = 0,96$

$\Big\} 1,36$

16

- Política d'escriptura retardada amb assignació
- Temps de servei en cas d'encert (t_h): 1 cicle
- Temps de servei de la memòria principal per llegir/escriure blocs (t_{block}): 15 cicles
- Percentatge d'escriptures (pe): 15%
- Percentatge de blocs modificades (pm): 20%
- Taxa d'encerts (h): 90%

a) Quin serà el temps de penalització (t_p, en cicles) en cas que el bloc estigui modificat? I en el cas que no ho estigui?

b) Quin serà el temps mitjà d'accés a memòria en cicles (t_{am})?

c) Quin serà el temps mitjà d'execució d'una instrucció en ns.?

<6.3> ③ **6.9.** A l'hora de dissenyar una memòria cache d'instruccions de 8 KB s'està dubtant entre les següents mides de bloc, i se n'ha medit la taxa de fallades per una col.lecció de programes representatius:

- 8 bytes: 16%
- 16 bytes: 10%

Es demana que:

a) Calculeu el temps mitjà d'accés a memòria (t_{am}) sabent que:

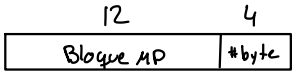
- El temps d'accés de les caches en cas d'encert (t_h) és 1 cicle
- El temps de servei de la memòria per servir blocs de 8 bytes és de 6 cicles, mentre que el temps per servir blocs de 16 bytes és de 12 cicles

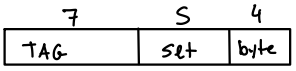
b) Quant ocuparan totes les etiquetes en cada una de les dues possibilitats sabent que:

- MP té 2^{32} bytes
- Les caches són de correspondència directa
- La unitat mínima d'accés a memòria és el byte

<6.4> ① **6.10.** Disposem d'un processador de 16 bits (paraules i adreces de 16 bits) amb una memòria cache que té les següents característiques:

- Correspondència associativa de 2 vies 2 way 2^1
- Mida total: 1024 bytes 2^5
- Mida bloc: 16 bytes 2^4
- Política d'escriptura immediata sense assignació WT + WnA
- Algorisme de reemplaçament: LRU





Ompliu la següent taula a partir de la seqüència de referències donades, on a la columna tipus *R byte* indica lectura d'1 byte, *R word* és lectura de 2 bytes, *W byte* és escriptura d'1 byte i *W word* és escriptura de 2 bytes. La mida de les lectures i escriptures s'ha d'especificar en bytes.

tipus	adr (hex)	#bloc MP	#conjunt MC	encert/ fallada	lectura d'MP			escriptura en MP		
					si/no	adr	mida	si/no	adr	mida
R byte	A930	A93	13	Miss	Si	A930	16	No	–	–
R word	B930	B93	13	Miss	Si	B930	16	No	–	–
W byte	A972	A97	17	Miss	No	–	–	Si	A972	1
W word	A932	A93	17	Hit	No	–	–	Si	A932	2
W byte	C935	C93	13	Miss	No	–	–	Si	C935	1
R word	C934	C93	13	Miss	Si	C930	16	No	-	–
W byte	B976	B97	17	Miss	No	–	–	Si	B976	1
W word	B936	B93	13	Miss	No	–	–	Si	B936	2
R byte	B938	B93	13	Miss	Si	B938	16	No	–	–
R word	A978	A97	17	Miss	Si	A978	16	No	–	–

<6.4> ①

6.11. Suposem que tenim un processador amb una memòria cache de dades amb les següents característiques:

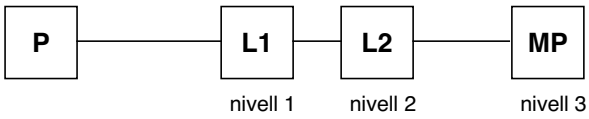
- 64 conjunts 2^6
- 4 blocs per conjunt 2^2 $= 2^{13}$
- 32 bytes per bloc 2^5
- paraules de 4 bytes
- algorisme de reemplaçament LRU

Sobre aquest sistema de memòria s'executen 2 versions diferents d'una mateixa aplicació:

```
int A[128][1024];/* emmagatzemada a partir de l'adreça 0 */

/* versió A */
sumA = 0;
for (i=0; i<128; i++)
        for (j=0; j<1024; j++)
               sumA = sumA + A[i][j];

/* versió B */
sumA = 0;
for (j=0; j<1024; j++)
        for (i=0; i<128; i++)
               sumA = sumA + A[i][j];
```

Indiqueu quantes fallades hi ha a la cache de dades per a cada una de les dues versions. Considereu que les variables i, j i sumA estan guardades en registres.

<6.4> ② **6.12.** El subsistema de memòria d'un determinat computador està organitzat en tres nivells:



Les característiques del processador i de cada nivell són les següents:

- **P**: La unitat d'adreçament es el byte.
- **L1**: Memòria cache de nivell 1 amb 4 blocs de 4 bytes cada un.

  Correspondència directa.
- **L2**: Memòria cache de nivell 2 amb 16 blocs de 4 bytes cada un.

  Correspondència associativa de 2 vies. Algorisme de reemplaçament LRU.
- **MP**: Memòria principal amb capacitat de 1 Mbyte.

El funcionament del sistema s'explica a continuació. L1 rep les peticions a memòria que genera P. En cas de fallada a L1, primer es comprova si el bloc es troba a L2. Si hi és, es copia aquest bloc a L1 sense necessitat d'accedir a MP. Si no hi és tampoc a L2, s'accedeix a MP i es copia a les dues caches. D'aquesta manera, L2 actua com una cache per a les peticions de memòria que genera L1.

Per avaluar el rendiment del subsistema de memòria, definim, per a cada nivell i, una taxa d'encerts h_i = núm. encerts a la L_i / núm. peticions a la L_i

Donada la següent seqüència de 28 adreces de lectura a memòria:

```
0, 5, 10, 12, 34, 0, 66, ... (que es repeteix 3 vegades més)
```

a) Suposant que les caches estan inicialment buides, calculeu les taxes h_1 i h_2

b) Calculeu de nou les taxes h_1 i h_2 suposant que ara canviem l'algorisme de reemplaçament de L2: usarem l'algorisme FIFO, que consisteix en reemplaçar el bloc que fa més temps que s'ha portat a la cache.

<6.4> ② **6.13.** Es vol dissenyar una memòria cache multinivell (L1 i L2, essent L1 la memòria cache més propera al processador). Es coneixen les següents dades (es consideren sols les lectures):

- El temps de servei d'L1 en cas d'encert és t_{h1}. En cas de fallada cal afegir-li una penalització consistent en el temps de servei d'L2.
- El temps de servei d'L2 en cas d'encert és t_{h2}. En cas de fallada cal afegir-li una penalització (t_{p2}) consistent en el temps d'accés a MP.
- m_1: taxa de fallades a L1.

6. 12

	0	5	10	12	34	0	66
linea	0	1	2	3	0	0	0
set	0	1	2	3	0	0	0

FIFO

		L1	L2	L1	L2	L1	L2
0	(0)						
1	(1)						
2	(2)						
3	(3)						
0	(0)						
0	(0)						
0	(0)						

LRU

		L1	L2	L1	L2	L1	L2
0	(0)	F	F	F	H	F	H
1	(1)	F	F	H	-	H	-
2	(2)	F	F	H	-	H	-
3	(3)	F	F	H	-	H	-
0	(0)	F	F	F	F	F	F
0	(0)	F	H	F	H	F	H
0	(0)	F	F	F	F	F	F

- m_2: taxa de fallades a L2 (taxa local = fallades en L2/accessos a L2)

Es demana que obtingueu l'expressió pel temps mitjà d'accés a memòria (t_{am}) expressat en cicles, en funció de t_{h1}, t_{h2}, t_{p2}, m_1 i m_2.

<6.4> ② **6.14.** Es disposa d'un processador de 8 bits i d'una cache amb els següents paràmetres:

- Nombre de blocs: 4
- Mida del bloc: 8 bytes
- Correspondència completament associativa
- Algorisme de reemplaçament: LRU
- Política d'escriptura retardada amb assignació
- Temps de servei en cas d'encert (per lectures i escriptures) (t_h): 1 cicle
- Temps de servei de la memòria principal (t_{block}): 10 cicles

Es demana:

a) Temps de servei en cas de fallada de lectura a una línia no modificada.

b) Temps de servei en cas de fallada de lectura a una línia modificada.

c) Temps de servei en cas de fallada d'escriptura a una línia no modificada.

d) Temps de servei en cas de fallada d'escriptura a una línia modificada.

e) Suposant que la cache és inicialment buida i que el processador genera la següent llista d'adreces de byte (en decimal, lectura=L, escriptura=E)

```
2L, 9L, 16L, 2L, 33E, 42E, 33L, 2L, 17L, 27E, 43L, 3L, 44L,
46L, 3L, 14L, 17L, 6L, 35L
```

- Indica, per a cada accés, si produeix encert, fallada amb reemplaçament (indicant l'índex del bloc reemplaçat), o fallada sense reemplaçament.
- Calcula el temps total de servei de la llista d'accessos.

6.14

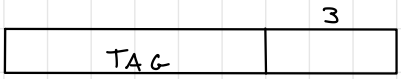
a) $t_h + t_{block} + t_h = 12$ ciclos

b) $t_h + 2t_{block} + t_h = 22$ ciclos

c) $t_h + t_{block} + t_h = 12$ ciclos

d) $t_h + 2t_{block} + t_h = 22$ ciclos

e)



$600 + 66 + 9 = 675$ ciclos

$12 \cdot 5 + 3 \cdot 22 + 9 =$

1	1	1	3	1	1	3	3	3	2	2	3	3	3	3	1	2	3	1
a)	a)	a)	H	c)	c)	H	H	H	d)	b)	H	H	H	H	a)	b)	H	a)
0	1	2	0	4*	5*	4*	0	2	3*	5	0	5	5	0	1	2	0	4
	0	1	2	0	4*	5*	4*	0	2	3*	5	0	0	5	0	1	2	0
		0	1	2	0	0	5*	4*	0	2	3*	3*	3*	3*	5	0	1	2
				1	2	2	2	5*	4*	0	2	2	2	2	3*	5	5	1
					1				5*	4*					2	3*		5

Tema 7. Memòria Virtual

<7.3> ① **7.1.** Es disposa d'un sistema amb memòria virtual paginada, amb les característiques següents:

- Espai d'adreces dels programes (grandària de la memòria virtual)= 16 Mbytes
- Espai de memòria física= 16 Kbytes
- Mida de les pàgines = 256 bytes

Cada programa pot tenir a memòria física un màxim de 3 pàgines simultàniament. En cas que una d'aquestes 3 pàgines hagués de ser sacrificada per deixar lloc a una altra, es fa servir l'algorisme de reemplaçament LRU.

Es demana:

a) Indica les dimensions de la taula de pàgines de cada programa, tenint en compte que cada entrada de la taula té un bit de presència (indica si la pàgina corresponent està carregada a memòria física) i un bit de modificació (indica si la pàgina corresponent ha estat modificada durant la seva estada a la memòria física).

b) A la taula següent apareix una llista d'adreces virtuals (en hexadecimal) que són generades pel processador. Omple la taula indicant, per a cadascuna de les adreces, el número de pàgina en hexadecimal corresponent (columna A), si l'accés produeix una fallada de pàgina o no (columna B), i, en cas que la fallada de pàgina hagi produït un reemplaçament, el número de pàgina reemplaçada (columna C). Considera que inicialment no hi ha cap pàgina carregada a memòria física.

Adreça (hex)	A Núm. pàgina (hex)	B Fallada de pàgina (SI o NO)	C Núm. pàgina reemplaçada (hex)
000023			
000101			
000102			
000200			
000010			
000111			
000416			
000520			
000101			

<7.3> ② **7.2.** Un sistema experimental es gestiona amb memòria virtual sota paginació amb algorisme de reemplaçament LRU. Aquest sistema té 64 KB de memòria virtual i 8 KB de memòria física. Les pàgines tenen 1 KB de mida. La taula de pàgines està carregada sempre a memòria física, ocupant l'últim marc de pàgina.

Suposem que una certa aplicació és ubicada dins l'espai lògic en les següents adreces:

- El codi del programa se situa en el rang d'adreces: 0x0000 - 0x03FF
- Les dades del programa se situen en el rang d'adreces: 0x0400 - 0x47FF

Dins les dades del programa hi ha una matriu que en alt nivell ha estat declarada com:

```
int MAT[32][256];
```

Aquesta matriu s'emmagatzema a memòria a partir de l'adreça base 0x0400. Tingues en compte que els enters es codifiquen en 16 bits. La resta de variables del programa, així com la pila, s'emmagatzemen a partir de l'adreça 0x4400.

Considerem el següent fragment del codi de l'aplicació:

```
for (i=0; i<32; i++)
     for (j=0; j<256; j++)
          MAT[i][j] = i+j;
```

on:

- El compilador reserva dos registres del processador per a les variables `i` i `j`.
- La taula de pàgines abans de començar a executar l'anterior codi indica que a memòria física només tenim la pàgina corresponent al codi de l'aplicació.

Es demana contestar els següents apartats, que fan referència a l'execució d'aquest fragment de codi:

a) Quins elements de MAT es porten a memòria física en la primera fallada de pàgina que es produeix?

b) Indica, justificant-ho raonadament, el nombre total de fallades de pàgina que es produiran.

c) En alguns llenguatges, com ara FORTRAN, els elements de les matrius es guarden en posicions consecutives però *per columnes* en lloc de *per files*. Considerant aquesta altra manera d'emmagatzemar les matrius, quins elements de MAT es portarien a memòria física en la primera fallada de pàgina?

d) Considerant l'emmagatzematge per columnes de l'apartat anterior, indica, justificant-ho raonadament, el nombre total de fallades de pàgina que es produirien.

<7.3> ③ **7.3.** Considerem un computador amb processador MIPS com l'estudiat a classe, que té una memòria cache de dades amb les següents característiques:

- associativa per conjunts de 4 vies (és a dir, de 4 blocs per conjunt)
- 64 conjunts
- 32 bytes per bloc
- algorisme de reemplaçament LRU

Considerem també que té un TLB de dades amb les següents característiques:

- completament associatiu
- 32 entrades
- mida de pàgina: 8 KB
- algorisme de reemplaçament LRU

Sobre aquest sistema s'executen 2 versions diferents d'una mateixa aplicació:

```
int A[128][1024];/* emmagatzemada a partir de l'adreça 0 */

/* versió A */
sumA = 0;
for (i=0; i<128; i++)
        for (j=0; j<1024; j++)
               sumA = sumA + A[i][j];

/* versió B */
sumA = 0;
for (j=0; j<1024; j++)
        for (i=0; i<128; i++)
               sumA = sumA + A[i][j];
```

Suposant que les variables `i`, `j`, i `sumA` s'emmagatzemen en registres, indica quantes fallades hi ha a la cache i al TLB, per a cada versió.

<7.3> ③ **7.4.** Tenim un processador amb memòria virtual basada en paginació. El sistema de memòria virtual té les següents característiques:

- 16 bits d'adreça lògica, i 15 bits d'adreça física
- mida de pàgines: 8 KB
- reemplaçament LRU

El contingut inicial de la taula de pàgines es mostra a continuació, on P és el bit de presència, D és el bit de pàgina modificada i PPN el número de pàgina física.

	P	D	PPN
0	0	0	-
1	0	0	-
2	1	0	0
3	1	0	1
4	1	0	2
5	0	0	-
6	0	0	-
7	0	0	-

A continuació també es mostra el contingut inicial de la memòria, en la qual la pàgina física 1 (lògica 3) és la que fa menys temps que ha estat accedida, la següent és la 0 (lògica 2) i la que fa més temps és la 2 (lògica 4).

pàgina física	pàgina lògica
0	2
1	3
2	4
3	-

La següent taula mostra una seqüència de referències a memòria (E: escriptura/ L: lectura). Emplena la taula indicant, per cada referència el número de pàgina lògica (VPN) i el número de pàgina física resultant de la traducció (PPN). Indica també amb una creu (X) si es produeix fallada de pàgina, si es llegeix del disc, i si s'escriu al disc. Indica també, en cas de reemplaçar una pàgina, el VPN i PPN de la pàgina reemplaçada. Per últim, indica també el contingut final de la taula de pàgines i de la memòria física.

adr. lògica (hex)		VPN (hex)	PPN (hex)	fallada de pàgina	lectura disc	escriptura disc	pàg. reemplaçada	
							VPN (hex)	PPN (hex)
E	F458							
E	8666							
L	1BBF							
E	5C44							
L	6600							
L	4000							

<7.4> ① **7.5.** Considerem el següent codi escrit en assemblador del MIPS, en què a l'etiqueta A li correspon l'adreça 0x10010000:

```
     la      $s0, A
     li      $s1, 0
     li      $s2, 512000
for: bge     $s1, $s2, fi
     sll     $t0, $s1, 2
     addu    $t0, $s0, $t0
     lw      $t1, 0($t0)
     lw      $t2, 8192($t0)
     addu    $t2, $t2, $t1
     sw      $t2, 8192($t0)
     sw      $t2, 16384($t0)
     addiu   $s1, $s1, 512
     b       for
fi:
```

Suposant que el sistema de memòria té les pàgines de mida 8 KB i que utilitza un TLB de 4 entrades (completament associatiu, i amb reemplaçament LRU), respon les següents preguntes:

a) Per a cadascun dels accessos a dades de memòria (instruccions en negreta) i per a les primeres 5 iteracions del bucle, indica a quina pàgina de la memòria virtual s'està accedint.

b) Indica la quantitat de fallades de TLB en tot el bucle.

c) Respon a les mateixes preguntes anteriors, suposant ara que el sistema de memòria té les pàgines de mida 4 KB.

<7.4> ② **7.6.** Considerem un sistema amb memòria virtual que té pàgines de 4 KB. A més, té un TLB de 4 entrades completament associatiu, on s'utilitza un reemplaçament LRU.

Considerem els següents continguts inicials del TLB (per al funcionament de l'algorisme LRU, considerem que les tres entrades vàlides han estat recentment accedides en el mateix ordre que ocupen dins el TLB):

TLB		
V	VPN	PPN
1	11	12
1	7	4
1	3	6
0	9	0

Considerem també els següents continguts inicials de la taula de pàgines. Per a noves pàgines assignades a MP, s'assignaran números de pàgina física (PPN) correlatius, a partir del número major (és a dir, els números 13, 14, etc.).

	TP	
	P	PPN
0	1	5
1	0	-
2	0	-
3	1	6
4	1	9
5	1	11
6	0	-
7	1	4
8	0	-
9	0	-
10	1	3
11	1	12

a) Per a cada una de les dues llistes de referències d'adreces virtuals que es mostren a continuació, indica quin serà l'estat final del TLB i de la taula de pàgines (TP). A més, per a cada referència, indica si es produeix una fallada al TLB, i si es produeix una fallada de pàgina.

- Llista 1: `4095, 31272, 15789, 15000, 7193, 4096, 8912`
- Llista 2: `9452, 30964, 19136, 46502, 38110, 17653, 47480`

b) Repeteix l'apartat anterior però en lloc de considerar les pàgines de 4 KB considera-les ara de 16 KB.

c) Considerant els paràmetres següents de dos sistemes que gestionen memòria virtual, indica quina serà la mida de la taula de pàgines en cada cas.

Mida adreces virtuals	Mida pàgina	Mida entrades TP
32 bits	4 KB	4 bytes
64 bits	16 KB	8 bytes

<7.4> ③ **7.7.** Considerem el següent codi escrit en assemblador del MIPS, en què a l'etiqueta A li correspon l'adreça 0x10010000.

```
     la      $s0, A
     li      $s1, 0
     li      $s2, 4096
for: bge     $s1, $s2, fi
     addu    $t0, $s1, $s1
     addu    $t0, $s0, $t0
     lh      $t1, 0($t0)
     lh      $t2, 1024($t0)
     addu    $t2, $t2, $t1
     sh      $t2, 1024($t0)
     sh      $t2, 4096($t0)
     addiu   $s1, $s1, 1
     b       for
fi:
```

Suposant que el sistema de memòria té les pàgines de mida 1 KB i que té un TLB de 4 entrades (amb reemplaçament LRU), respon les següents preguntes:

a) Indica el nombre de pàgines diferents accedides per a cadascun dels accessos a dades de memòria (instruccions en negreta).

b) Calcula el nombre de fallades de TLB en tot el bucle.

c) Es voldria redissenyar el TLB per minimitzar el nombre de fallades en aquest programa. Indica quin és el nombre mínim de fallades de TLB que podem aconseguir amb el millor disseny. Indica també quin seria el mínim nombre d'entrades de TLB necessàries per obtenir aquest resultat.

d) Repeteix l'apartat anterior, però en lloc de considerar les pàgines de 1 KB considera-les ara de 4 KB