Solucionari
de la 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

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?

Només necessitaríem compilar l'intèrpret de Java escrit en C, a llenguatge màquina MIPS per executar un programa escrit en Java. El problema és que tenim un compilador de C a x86, escrit en LM x86. Per tant hauríem de fer la següent seqüència en una màquina MIPS:

- *En la màquina MIPS traduirem el binari x86 del compilador de C a llenguatge màquina MIPS (però el compilador seguirà generant codi x86).*
- *En la màquina MIPS compilarem l'intèrpret de Java escrit en C amb el nou compilador de C. Obtindrem un intèrpret de Java per x86.*
- *En la màquina MIPS traduirem el binari per x86 de l'intèrpret de Java a llenguatge màquina MIPS.*

En la màquina MIPS interpretarem el programa Java amb el nou intèrpret de Java per MIPS

<1.2>① **1.3** Traduïm 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	$2*10^6$
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?

$Texec = I * CPI * Tc$

$Texec\ P1 = 2 * 10^{6} * 1.5 * 0,5 * 10^{-9} = 1,5 * 10^{-3}\ s$

$Texec\ P2 = 2 * 10^{6} * 1 * 1/2,5 * 10^{-9} = 0,8 * 10^{-3} s$

$Texec\ P3 = 10^{6} * 0,75 * 1/3 * 10^{-9} = 0,25 * 10^{-3} s$

El Processador P3 és el que executa més ràpidament el programa. L'executa 6 vegades més ràpid que P1 i 3.2 vegades més ràpid que P2.

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?

*Sense conèixer el nombre d'instruccions del programa, només podem comparar P1 amb P2 ja que s'han programat per la mateixa ISA i per tant executen exactament les mateixes instruccions. Serà més ràpid el que tardi menys en executar una instrucció (CPI*tc segons).*

$Texec/I\ P1 = 1.5 * 0.5 * 10^{-9} = 0,75 * 10^{-9}\ s$

$Texec/I\ P2 = 1 * 1/2,5 * 10^{-9} = 0,4 * 10^{-9}\ s$

P2 és més ràpid que P1.

Respecte de P3, no podem afirmar res, ja que utilitza una ISA diferent (diferent nombre i tipus d'instruccions)

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?

P1 i P2 es poden comparar per a aquest programa, però no en general, ja que per a un altre programa pot variar el CPI (el que dona la taula és el CPI mitjà per a aquest programa) . Si el CPI fos general "per a tots els programes", aleshores podríem comparar P1 i P2.

Tant si tenim com si no un CPI mitjà per a tots els programes no podem comparar P1 i P3, ja que utilitzen ISAs diferents i desconeixem la quantitat relativa d'instruccions entre les dos ISAs

<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?

$$T_{exec} = (500 \times 1 + 150 \times 5 + 50 \times 2) \times 10^{-9} = 1350 \times 10^{-9} s$$

b) Quin seria el CPI?

$$CPI = \frac{(500 \times 1 + 150 \times 5 + 50 \times 2)}{500 + 150 + 50} = \frac{1350}{700} = 1,93 \text{cpi}$$

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

$$T_{exec} = (500 \times 1 + 100 \times 5 + 50 \times 2) \times 10^{-9} = 1100 \times 10^{-9} s$$

$$\text{speedup} = 1350 \times 10^{-9} / 1100 \times 10^{-9} = 1,23 \text{vegades}$$

$$CPI = \frac{(500 \times 1 + 100 \times 5 + 50 \times 2)}{500 + 150 + 50} = \frac{1100}{650} = 1,69 \text{cpi}$$

<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.

$$P = C \times V^2 \times F \rightarrow C = \frac{P}{V^2 \times F}$$

Per tant, la càrrega capacitiva és: $C_A = 8 \cdot 10^{-9}$, $C_B = 3,3 \cdot 10^{-8}$

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

$$P_A = C \times V^2 \times F = 8 \times 10^{-9} \times 5^2 \times 3 \times 10^9 = 600 w$$

<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.

$$CPI_A = \frac{(90 \times 2 + 240 \times 1,5 + 150 \times 6 + 120 \times 3)}{90 + 240 + 150 + 120} = \frac{1800}{600} = 3\text{cpi}$$

$$CPI_B = \frac{(45 \times 3 + 225 \times 1 + 135 \times 5 + 45 \times 2)}{45 + 225 + 135 + 45} = \frac{1125}{450} = 2,5\text{cpi}$$

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.

$$T_{execA} = 1800 \times 1/F_A$$

$$T_{execB} = 1125 \times 1/F_B$$

$$1800 \times 1/F_A = 1125 \times 1/F_B$$

$$F_A = \frac{1800}{1125} \times F_B = 1,6 \times F_B$$

La freqüència de A ha de ser 1,6 vegades més alta que la de B.

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?

Llei d'Amdhal:

$$S = \frac{1}{\frac{Px}{Sx} + (1 - Px)}$$

$$\lim_{Sx \to \infty} S = S_{\max} = \frac{1}{1 - Px}$$

*Donat que la millora parcial és màxima (reduïm el temps a 0) i que la part millorada és Px = (150*6)/(90*2+240*1.5+150*6+120*3) = 900/1800 = 0.5 aleshores:*

```
Smax = 1 / (1-0.5) = 2 vegades
```

Tema 2. Instruccions i tipus bàsics de dades

<2.1> 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).

0x10010000	01	0x1001000A	00	0x10010014	FD
0x10010001	02	0x1001000B	00	0x10010015	FF
0x10010002	03	0x1001000C	FE	0x10010016	FF
0x10010003	04	0x1001000D	FF	0x10010017	FF
0x10010004	FF	0x1001000E	FF	0x10010018	03
0x10010005	FF	0x1001000F	FF	0x10010019	00
0x10010006	FF	0x10010010	02	0x1001001A	00
0x10010007	FF	0x10010011	00	0x1001001B	00
0x10010008	01	0x10010012	00	0x1001001C	78
0x10010009	00	0x10010013	00	0x1001001D	56

<2.2> ③ **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)
```

```
        .text
la     $t1, C              #$t1 = 0x1001000C
lw     $t2, 12($t1)        #$t2 = 0x10010008
lb     $t1, 8($t2)         #$t1 = 0xFFFFFFFF
la     $t3, B              #$t3 = 0x10010008
lb     $t4, 0($t3)         #$t4 = 0x00000003
addu   $t1, $t1, $t4       #$t1 = 0x00000002
sb     $t1, 0($t3)         #M[0x10010008] = 0x02. B val 2
```

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

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

```
.text
la     $t1, A+1            #$t1= 0x10010001
lw     $t1, 0($t1)         #Produeix un error d'alineament
                           #Excepció:Adreça no múltiple de 4
```

<2.1> ② **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 Magnitud	Ca1	Ca2	Excés a 2^3-1
8	NC	NC	NC	1111
7	0111	0111	0111	1110
6	0110	0110	0110	1101
5	0101	0101	0101	1100
4	0100	0100	0100	1011
3	0011	0011	0011	1010
2	0010	0010	0010	1001
1	0001	0001	0001	1000
0	0000/1000	0000/1111	0000	0111
-1	1001	1110	1111	0110
-2	1010	1101	1110	0101
-3	1011	1100	1101	0100

N	Signe i Magnitud	Ca1	Ca2	Excés a 2^3-1
-4	1100	1011	1100	0011
-5	1101	1010	1011	0010
-6	1110	1001	1010	0001
-7	1111	1000	1001	0000
-8	NC	NC	1000	NC

<2.3> ① **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)
```

```
*pdada = *pdada + 4;
```

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

```
pdada = pdada + 1;
```

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

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

<2.3> ① **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;`

```
addiu $t1, $t1, 4
```

b) `a = *p;`

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

c) `h = &c;`

```
la $t2, c
```

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

```
la   $t3, b
lw   $t6, 0($t3)
sll  $t4, $t6, 2
addu $t5, $t1, $t4
lw   $t5, 0($t5)
sw   $t5, 0($t3)
```

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

```
la   $t3, b
lw   $t3, 0($t3)
sll  $t3, $t3, 3
addu $t4, $t3, $t2
lw   $t5, 0($t4)
sw   $t5, 0($t2)
lw   $t5, 4($t4)
sw   $t5, 4($t2)
```

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

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

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

```
lw    $t3, 0($t0)
sll   $t3, $t3, 3
addiu $t2, $t3, $t2
```

Tema 3. Traducció de programes

<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.

Primera solució: Aquest bucle executa 32 iteracions de 5 instruccions cada una. En total, executa 2+32·5 = 162 instruccions

```
     li      $t2, 32
     move    $t3, $zero
bucle:
     andi    $t0, $t1, 1
     addu    $t3, $t3, $t0
     srl     $t1, $t1, 1
     addiu   $t2, $t2, -1
     bne     $t2, $zero, bucle
```

Solució millorada: Aquest bucle executa tantes iteracions com bits hi ha en $t1 si descontem els zeros inicials de més pes, (p.ex. si $t1 = 0x00100010, fa 21 iteracions). En total, executa 2+21·5 = 107 instruccions

```
     move    $t3, $zero
     beq     $t1, $zero, fi
bucle:
     andi    $t0, $t1, 1
     addu    $t3, $t3, $t0
     srl     $t1, $t1, 1
     bne     $t1, $zero, bucle
fi:
```

Solució òptima: Existeix una solució encara més eficient, que té igualment 4 instruccions per iteració però que tan sols fa tantes iteracions com bits actius té $t1 (p.ex. si $t1 = 0x00100010, fa sols 2 iteracions, és a dir que executaria 2+2·4 = 10 instruccions). Us convidem a buscar-la.

<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.

Expressem primer el problema en alt nivell:

```
carry = 0; i=4;
do {
      /* valor binari de la suma d'un dígit i el carry anterior */
      suma = (V1[i] - '0') + (V2[i] - '0') + carry;
      /* Càlcul del carry: si suma>9, carry=1; sino carry=0 */
      carry = (suma>9);
      if (carry)
               suma = suma - 10;
      V3[i+1] = suma + '0';
      i--;
} while (i>=0);
V3[0] = carry + '0';
```

A continuació el traduïm a MIPS:

```
      move     $t0, $zero          # carry = 0
      la       $s1, v1
      la       $s2, v2
      la       $s3, v3
      li       $t1, 4              # i = 4
      li       $t2, 9              # immediat per a les comparacions
do:
      addu     $t7, $s1, $t1       # @V1[i]
      lbu      $t7, 0($t7)         # V1[i]
      addu     $t8, $s2, $t1       # @V2[i]
      lbu      $t8, 0($t8)         # V2[i]
      addu     $t7, $t7, $t8       # suma= V1[i]+V2[i]
      addu     $t7, $t7, $t0       #     = V1[i]+V2[i]+carry
      addiu    $t7, $t7, -96       #     = V1[i]+V2[i]+carry-'0'-'0'
      slt      $t0, $t2, $t7       # carry = (9 < suma)
      beq      $t0, $zero, seguir
      addiu    $t7, $t7, -10       # suma = suma-10
seguir:
      addiu    $t7, $t7, 48        # suma+'0'
      addu     $t8, $s3, $t1       # @V3[i]
      sb       $t7, 1($t8)         # V3[i+1] = suma+'0'
      addiu    $t1, $t1, -1        # i--
      bge      $t1, $zero, do      # while (i>=0);

      addiu    $t0, $t0, 48        # carry + '0'
      sb       $t0, 0($s3)         # V3[0] = carry + '0'
```

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

```
     la      $t1, c
     lbu     $t1, 0($t1)          # llegim c de la memòria
     li      $t0, 'A'
     blt     $t1, $t0, else1      # salta si c<'A'
     li      $t2, 'F'
     bgt     $t1, $t2, else1      # salta si c>'F'
     addiu   $t1, $t1, 10-'A'     # sumem immediats[1]
     b       fi_if
else1:
     li      $t0, 'a'
     blt     $t1, $t0, else2      # salta si c<'a'
     li      $t2, 'f'
     bgt     $t1, $t2, else2      # salta si c>'f'
     addiu   $t1, $t1, 10-'a'     # sumem immediats[1]
     b       fi_if
else2:
     addiu   $t1, $t1, -'0'       # sumem immediats[1]
fi_if:
     la      $t0, num
     sb      $t1, 0($t0)          # guardem resultat a memòria
```

1. *Nota: L'assemblador Mars no admet expressions amb constants per als operands immediats. No obstant, com que la majoria d'assembladors les admeten i com que simplifiquen la lectura dels programes, en EC admetrem aquestes expressions en les solucions d'exàmens.*

<3.4> ③ **3.31.** Tradueix a assemblador MIPS la funció *func2*[1].

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

Observem que després de la crida a func2, es necessiten les variables loc *i* b. *Per tant caldrà mantenir-les en registres $s per garantir que no es modifiquin.* loc *ocuparà el registre $s0 i copiarem* b *en $s2, just abans de la crida.*

Les regles de l'ABI estipulen que tota funció ha de preservar el valor previ dels registres $s. Per tant, si modifiquem $s0 i $s2, caldrà salvar els seus valors previs a la pila just a l'inici de la funció i restaurar-los just al final.

```
func2:
     # Salvar registres: $ra, $s0(loc) i $s2($a2=b)
     addiu   $sp, $sp, -12
     sw      $s2, 0($sp)          # salvar $s2
     sw      $s0, 4($sp)          # salvar $s0
     sw      $ra, 8($sp)          # salvar $ra

     lh      $t0, 0($a2)          # desreferència: *b
     li      $t1, 3
     ble     $t0, $t1, else       # salta si (*b) <= 3
     move    $s0, $a1             # loc = a
     b       fi_if
else:
     sll     $t0, $t0, 1
     addu    $t0, $t0, $a0        # @vec[*b]
     lh      $s0, 0($t0)          # loc = vec[*b]

     # Salva $a2, passa els 3 paràmetres i fa la crida
     move    $s2, $a2             # salvem $a2 en $s2
     addiu   $a0, $a0, 2          # $a0 = vec+1
     addiu   $a1, $s0, 1          # $a1 = loc+1
     addiu   $a2, $s2, 2          # $a2 = b+1
     jal     func2
     sh      $v0, 0($s2)          # *b = func2(...)
fi_if:
     move    $v0, $s0             # return loc

     # Restaura registres i retorna
     lw      $s2, 0($sp)
     lw      $s0, 4($sp)
     lw      $ra, 8($sp)
     addiu   $sp, $sp, 12
     jr      $ra
```

1. Aquest enunciat no coincideix totalment amb l'original de la col·lecció, ja que hem corregit algunes inconsistències de tipus, fent que tots els tipus enters siguin *short*.

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]`

```
#mat + 3*NC*4 + 11*4
la   $v0, mat + 3*NC*4 + 11*4
```

b) `mat[i][11]`

```
#mat + i*NC*4 + 11*4
la    $v0, mat + 11*4
li    $t2, NC*4
mult  $t2, $t0
mflo  $t2
addu  $v0, $v0, $t2
```

c) `mat[3][j]`

```
#mat + 3*NC*4 + j*4
la    $v0, mat + 3*NC*4
sll   $t2, $t1, 2
addu  $v0, $v0, $t2
```

d) `mat[i][j]`

```
#mat + (i*NC + j)*4
la    $v0, mat
li    $t2, NC
mult  $t2, $t0
mflo  $t2
addu  $t2, $t1, $t2
sll   $t2, $t2, 2
addu  $v0, $v0, $t2
```

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

```
#mat + ((i+5)*NC + j-1)*4
#(mat + 5*NC*4 - 4) + (i*NC + j)*4

la    $v0, mat + 5*NC*4 - 4
li    $t2, NC
mult  $t2, $t0
mflo  $t2
addu  $t2, $t1, $t2
sll   $t2, $t2, 2
addu  $v0, $v0, $t2
```

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

```
#mat + ((i*10+4)*NC + 6)*4 = mat + i*40*NC + 16*NC + 24
la    $v0, mat + 16*NC + 24
li    $t2, 40*NC
mult  $t2, $t0
mflo  $t2
addu  $v0, $v0, $t2
```

<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.

```
#@mat[i][j] = @mat + i*10 + j = @mat + i*10 + j
#@mat[i-3][j+5] = @mat + (i-3)*10 + j+5 = @mat + i*10 - 30 +
j + 5
addiu $t0, $t0, -25
```

<4.3> 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 utilizant 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:

g)

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

```
        li    $t0, 0 #i
        li    $t3, N
        addiu $t4, $a0, 3*N*4  #A[3][0]  Punter 1º element
for:    bge   $t0, $t3, fi_for
        lw    $t5, 0($t4)
        addu  $t2, $t2, $t5
        addiu $t4, $t4, 4  #Stride
        addiu $t0, $t0, 1
fi_for:
```

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

```
        li    $t0, 0 #i
        li    $t3, N
        addiu $t4, $a0, 4*4   #A[0][4]  Punter 1º element
for:    bge   $t0, $t3, fi_for
        lw    $t5, 0($t4)
        addu  $t2, $t2, $t5
        addiu $t4, $t4, 4*N  #Stride
        addiu $t0, $t0, 1
fi_for:
```

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

```
        li    $t0, 0 #i
        li    $t3, N
        addiu $t4, $a0, 0     #A[0][0]  Punter 1º element
for:    bge   $t0, $t3, fi_for
        lw    $t5, 0($t4)
        addu  $t2, $t2, $t5
        addiu $t4, $t4, (N+1)*4  #Stride
        addiu $t0, $t0, 1
fi_for:
```

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

```
        li    $t0, 0 #i
        li    $t3, N
        addiu $t4, $a0, (N-1)*4  #A[0][N-1]  Punter 1º element
for:    bge   $t0, $t3, fi_for
        lw    $t5, 0($t4)
        addu  $t2, $t2, $t5
        addiu $t4, $t4, (N-1)*12  #Stride
        addiu $t0, $t0, 3
fi_for:
```

<4.3> 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 */
}
```

k) 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)
```

```
p1[(p2+3)*2] = 0;
```

l) 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 =  &p1[0];
;

for (i=0; i<100; i++)
{
      *p = 0;
                                  ;
      p = p + 1;

                                  ;
```

Codi equivalent en MIPS

```
      move    $t1, $a0
      move    $t0, $zero    ; i=0
      li      $t2, 100
bucle:
      bge     $t0, $t2, fibucle
      sw      $zero, 0($t1)
      addiu   $t1, $t1, 2
      addiu   $t0, $t0, 1   ; i++
      b       bucle
fibucle:
```

Tema 5. Aritmètica d'enters i coma flotant

<5.2> 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?

```
     A = 0x0D34 = 0000 1101 0011 0100
     B = 0xDD17 = 1101 1101 0001 0111
A + B           = 1110 1010 0100 1011 = 0xEA4B
```

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

```
A = 0x0D34 = 0 000110100110100
B = 0xDD17 = 1 101110100010111
```

Són de signe contrari: la magnitud del resultat és la diferència de les magnituds i el signe és igual al del operand amb major magnitud (en aquest cas negatiu, ja que la magnitud de B és major):

```
magn(B) =    101110100010111
magn(A) =    000110100110100
resta   =    100111111100011
A + B   =  1 100111111100011 = 0xCFE3
```

Hi ha un procediment alternatiu, encara que no entra en el temari d'EC [1]

---

1. *Procediment basat en la suma de nombres en complement a 1*

(a) Convertir els números al format de complement a 1 (sols cal complementar els bits de la magnitud en cas que el signe sigui negatiu).

```
A = 0 000110100110100 = 0000110100110100
B = 1 101110100010111 = 1010001011101000
```

(b) Sumar els números en complement a 1: se sumen els números en binari i s'incrementa el resultat en una unitat en cas de produir-se un carry:

```
                  suma = 1011000000011100 (carry = 0)
A + B =  suma + carry = 1011000000011100
```

(c) Convertir el resultat novament al format de signe i magnitud (complementar els bits de la magnitud, en cas que el signe sigui negatiu):

```
A + B                 = 1100111111100011 = 0xCFE3
```

<5.1> ③ **5.8.** La condición de desbordamiento (overflow) de la suma 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.

La condició de desbordament es pot expressar algebraicament així (queda com a exercici per a l'alumne demostrar que les dues expressions són equivalents):

$$Overflow = \bar{A}_{31} \cdot \bar{B}_{31} \cdot S_{31} + A_{31} \cdot B_{31} \cdot \bar{S}_{31} = \overline{(A_{31} \oplus B_{31})} \cdot (A_{31} \oplus S_{31})$$

El següent programa, usant instruccions xor, nor, *i* and, *calcula la segona expresió no sols per al bit 31, sinó per a tots els bits. Al final, sols cal desplaçar el bit 31 a la posició 0 per mitjà de la instrucció* srl:

```
addu    $t0, $t1, $t2       # La suma S
xor     $t3, $t1, $t2       # A xor B
nor     $t3, $t3, $t3       # not (A xor B)
xor     $t4, $t1, $t0       # A xor S
and     $t3, $t3, $t4
srl     $t3, $t3, 31
```

<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:

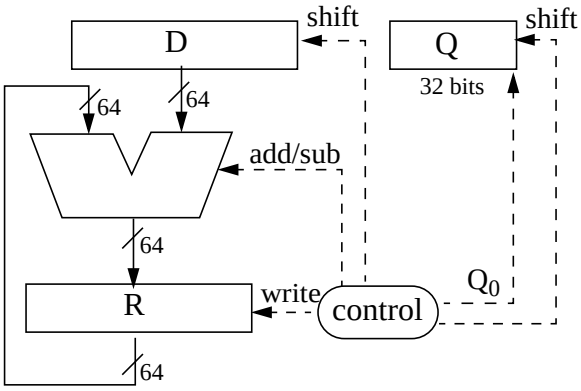
a) `x = x + y;`

```
la      $t0, x
la      $t1, y
lw      $t2, 0($t0)         # part baixa de x
lw      $t3, 4($t0)         # part alta de x
lw      $t4, 0($t1)         # part baixa de y
lw      $t5, 4($t1)         # part alta de y
addu    $t2, $t2, $t4       # suma parts baixes
sltu    $t6, $t2, $t4       # carry de la suma
addu    $t3, $t3, $t5       # suma parts altes
addu    $t3, $t3, $t6       # suma part alta i carry
sw      $t2, 0($t0)
sw      $t3, 4($t0)
```

Nota: El càlcul del carry de les parts baixes és per definició el desbordament de la suma de naturals, i aquest es pot calcular comprovant si el resultat de la instrucció addu *és menor que un qualsevol dels seus operands. La clau de l'exercici és adonar-se que cal usar* sltu *i no pas* slt

<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, cal-

culant alhora el quocient i el residu, completa l'algorisme iteratiu que en descriu el funcionament cicle a cicle:




```
D63:32  = Y ;
D31:0   = 0 ;
Q       = 0 ;
R63:32  = 0 ;
R31:0   = X ;
for (i=1; i<=32; i++)
{
   D = D>>1;
   R = R-D;
   if (R>=0)
       Q=(Q<<1)|0x1;
   else
   {
       R = R+D;
       Q = Q<<1;
   }
}
```

<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 i Residu
valor inicial	Q=0 $D_{11:6}$=**Y**; $D_{5:0}$=0 $R_{11:6}$=0; $R_{5:0}$=**X**	000000	**010011** 000000	000000 **101000**
1	D=D>>1 R=R-D R<0->R=R+D,Q=Q<<1	000000	001001 100000	110111 001000 000000 101000
2	D=D>>1 R=R-D R<0->R=R+D,Q=Q<<1	000000	000100 110000	111011 111000 000000 101000
3	D=D>>1 R=R-D R<0->R=R+D,Q=Q<<1	000000	000010 011000	111110 010000 000000 101000
4	D=D>>1 R=R-D R<0->R=R+D,Q=Q<<1	000000	000001 001100	111111 011100 000000 101000
5	D=D>>1 R=R-D R>0->Q=Q<<1|0x01	000001	000000 100110	000000 000010
6	D=D>>1 R=R-D R<0->R=R+D,Q=Q<<1	000010	000000 010011	111111 101111 000000 000010

<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 ?

Per a cada operand, l'expressem en binari, decodifiquem l'exponent en excés a 127 i afegim el bit ocult a la mantissa:

$f2 = 0x417ac000
= 0 10000010 11110101100000000000000
= $1{,}111101011 \cdot 2^{130-127} = 1{,}111101011 \cdot 2^{3}$

$f4 = 0x3f140000
= 0 01111110 00101000000000000000000
= $1{,}00101 \cdot 2^{126-127} = 1{,}00101 \cdot 2^{-1}$

Igualem l'exponent de $f4 al de $f2 (que és el major desl dos):

$= 0{,}000100101 \cdot 2^{3}$

Afegim els bits de guarda GRS *i sumem:*

$$\begin{array}{rl} & \text{GRS} \\ & 1{,}11110101100000000000000\ 000 \cdot 2^{3} \\ + & 0{,}00010010100000000000000\ 000 \cdot 2^{3} \\ \hline = & 10{,}00001000000000000000000\ 000 \cdot 2^{3} \end{array}$$

Normalitzem i trunquem els bits GRS*:*

$= 1{,}00000100000000000000000 \cdot 2^{4}$

Codifiquem l'exponent en excés a 127, i eliminem el bit ocult

$f6 = $1{,}000001 \cdot 2^{131-127}$
= 0 10000011 00000100000000000000000
= 0x41820000

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

No, ja que al truncar els bits GRS *no es perd informació*

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

Expressem $f6 en forma de part entera i fraccionària separades per la coma:

$f6 = $1{,}000001_2 \cdot 2^{4} = 10000{,}01_2$

Convertim cada part per separat a la base decimal:

$10000_2 = 16_{10}$
$0{,}01_2 = 1{,}0 \cdot 2^{-2} = 1/4 = 0{,}25_{10}$

I obtenim el resultat:

$= 16{,}25_{10}$

<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;
}
```

En ser una rutina uninivell i no modificar cap dels registres $s, no es necessita salvar cap registre (tampoc $ra).

```
absdif:
     c.lt.s   $f14, $f12          # bit de condició = (b<a)
     bc1f     else                # salta si bit de condició fals
     sub.s    $f0, $f12, $f14     # resultat = a-b
     b        fisi
else:
     sub.s    $f0, $f14, $f12     # resultat = b-a
fisi:
     jr       $ra
```

<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[1]. 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?

Expressar en binari, decodificar exponent en excés a 15, afegir bit ocult

A = 0xE4FE = 1 11001 0011111110
= -1,0011111110 • 2^{25-15}
= -1,0011111110 • 2^{10}
B = 0xB640 = 1 01101 1001000000
= -1,1001000000 • 2^{13-15}
= -1,1001000000 • 2^{-2}

Afegir bits de guarda GRS *i igualar l'exponent de B al de A, que és el major: es perden quasi tots els bits! sols se'n conserven 2 en els bits* RS

GRS
A = -1,0011111110 000 • 2^{10}
B = -0,0000000000 011 • 2^{10}

Sumar mantisses (al ser A i B del mateix signe, sumem):

GRS
A = 1,0011111110 000 • 2^{10}
B = 0,0000000000 011 • 2^{10}

A + B = 1,0011111110 011 • 2^{10}

Normalitzar resultat

A + B = 1,0011111110 011 • 2^{10}

Arrodonir cap al més pròxim: 011 *arrodoneix avall (truncar)*

A + B = 1,0011111110 • 2^{10}

Codificar l'exponent en excés a 15 i eliminar bit ocult

= 1 11001 0011111110 = 0xE4FE

Convertim a decimal A, B i la suma arrodonida:

A = -1,0011111110 • 2^{10} = -10011111110_2 = -1278_{10}
B = -1,1001• 2^{-2} = -11001 · 2^{-6} = -25/64 = $-0{,}390625_{10}$
A+B (arrod) = 1,0011111110 • 2^{10} = -10011111110_2 = -1278_{10}
A+B (exact) = -1278 -0,390625 = -1278,390625
Error = (-1278) - (-1278,390625) = 0,390625

1. 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).

Tema 6. 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
- 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	Fallada	16	-	Si	Si
R	4568	45	6	Fallada	16	-	Si	Si
W	13A4	13	A	Fallada	-	2	No	No
W	13A8	13	A	Fallada	-	2	No	No
R	3560	35	6	Fallada	16	-	Si	Si
W	453C	45	3	Encert	-	2	No	Si
W	60A0	60	A	Fallada	-	2	No	No
R	453C	45	3	Encert	-	-	Si	No
W	3900	39	0	Fallada	-	2	No	No
R	A238	A2	3	Fallada	16	-	Si	Si

Explicació addicional: El número de blocs que té la MC és 16 (256 bytes / 16 bytes/bloc). Com que els blocs tenen 16 bytes calen 4 bits de l'adreça per especificar un byte dins el bloc que s'accedeix (dígit hexadecimal de menys pes de l'adreça). Com que hi ha 16 blocs, l'índex de MC vindrà especificat pels següents 4 bits (segon dígit hexadecimal per la dreta). L'etiqueta vindrà donada pels 8 bits de més pes (els 2 dígits hexadecimals de més pes).

b) Ompliu ara la mateixa taula, suposant que la política és escriptura retardada amb assignació.

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	Fallada	16	-	Si	Si
R	4568	45	6	Fallada	16	-	Si	Si
W	13A4	13	A	Fallada	16	-	No	Si
W	13A8	13	A	Encert	-	-	No	Si
R	3560	35	6	Fallada	16	-	Si	Si
W	453C	45	3	Encert	-	-	No	Si
W	60A0	60	A	Fallada	16	16	Si	Si
R	453C	45	3	Encert	-	-	Si	No
W	3900	39	0	Fallada	16	.	No	Si
R	A238	A2	3	Fallada	16	16	Si	Si

c) Indiqueu per cada política:

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

	Escriptura immediata sense assignació	Escriptura retardada amb assignació
Taxa de fallades	8/10 = 0.80	7/10 = 0.70
#byte llegits de MP	4 blocs * 16 bytes = 64 bytes	7 blocs * 16 bytes = 112 bytes
#bytes escrits a MP	5 words * 2 bytes = 10 bytes	2 blocs * 16 bytes = 32 bytes

<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.

En primer lloc cal veure com es mapegen les estructures de dades del programa dins la MC. La matriu A té una adreça base que es mapeja al primer byte de la primera entrada de la MC. Cada bloc de la MC té 32 bytes, que és just la mida d'una fila de la matriu A. Per tant, la primera fila d'A es mapeja de forma sencera a la primera entrada de la MC. La segona fila d'A a la segona entrada, i així successivament. Com que tenim 32 files a la matriu A i 32 entrades a la MC, l'última fila de la matriu es mapeja a l'última entrada de la MC.

El vector B està ubicat a memòria després de la matriu A. Aquest vector es mapeja a la primera entrada de la MC.

El vector C està ubicat a memòria després del vector B i, per tant, es mapeja a MC a la segona entrada.

Ara cal fer un exercici d'abstracció per veure com es comporten tots els accessos a memòria referents a aquestes estructures de dades. Primer veurem quin comportament van tenint els accessos inicials i quan observem una repetició en aquest comportament podrem arribar a deduir el comportament global.

L'ordre dels accessos a memòria és:

- *Lectura de A[i][j]*
- *Lectura de B[j]*
- *Lectura de C[i]*
- *Escriptura a C[i]*

Comencem analitzant els accesos que es produeixen a la primera iteració del bucle exterior (i=0):

Quan j=0: M M M H, on s'indica encert (H) o fallada (M) en els 4 accessos a memòria que es fan en aquesta iteració, respectant l'ordre de realització dels accessos.

Quan j=1: M M H H

...

Quan j=31: M M H H

És a dir, en aquesta iteració del bucle exterior (i=0) s'observa que hi ha conflicte en l'accés a la fila 0 d'A i al vector B. En quan al vector C només hi ha la fallada inicial i tots els accessos posteriors són encerts.

Anàlisi de la segona iteració del bucle exterior (i=1): Hi ha conflicte en l'accés entre la fila A[1] i el vector C.

En total tenim que l'accés a A genera 32 M's; l'accés a B genera 32 H's; l'accés de lectura a C genera 32 M's i l'accés d'escriptura genera 32 H's.

Anàlisi d'i=2: No hi ha cap conflicte. Únicament tenim la fallada inicial per carregar la fila A[2] a la MC.

En total, A: 1 M i 31 H's; B: 32 H's; C lectura: 32 H's; C escriptura: 32 H's

Els accessos quan i=3 es comporten com a la fila anterior.

Els accessos quan i=4 es comporten com quan i=0, excepte per un cas, que és la primera lectura de C, que no genera una fallada perquè ja es troba present a MC.

En total, A: 32 M's; B: 32 M's; C lectura: 32 H's; C escriptura: 32 H's

A partir d'aquí ja es va repetint sempre el comportament observat 4 iteracions abans. Per tant, el comportament per les primeres 4 iteracions es repeteix 8 vegades, excepte que en el primer grup de 4 iteracions hi ha un encert menys.

En definitiva, el número d'encerts global que s'obté en l'execució d'aquest parell de bucles imbricats és:

$$(64 + 64 + 127 + 127) * 8 - 1 = 3055$$

El número toal d'accessos a memòria és:

$$32*32*4 = 4096$$

La taxa d'encerts és h = 3055/4096 = 0,75

<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.

Comencem analitzant el cas d'escriptura immediata sense assignació. En aquest cas tenim un temps de servei diferent quan hi ha una fallada en una lectura ($2*t_h + t_{block} = 120$ ns.) de qualsevol altre situació ($t_h = 10$ ns.). Haurem de calcular la mitjana ponderada en cada cas. Una lectura que falli succeeix en un $0.8*0.1 = 0.08 = 8\%$ dels accessos. La resta de situacions succeeixen en el 92% dels accesos. El temps d'accés mitjà a memòria és $t_{am} = 0.08 * 120$ ns. $+ 0.92 * 10$ ns. $= 18.80$ ns.

Quan s'utilitza una política d'escriptura retardada amb assignació tenim els casos següents, que generen un temps de servei diferent per accés a memòria:

Encerts (siguin de lectura o escriptura): $t_h = 10$ ns.

Fallades en què s'elimina un bloc que havia estat modificat: $2*t_h + 2*t_{block} = 220$ ns.

Fallades en què s'elimina un bloc que no havia estat modificat: $2*t_h + t_{block} = 120$ ns.

El temps d'accés mitjà a memòria és $t_{am} = 0.85*10$ ns. $+ 0.15*0.3333*220$ ns. $+ 0.15*0.6666*120$ ns. $= 31.50$ ns.

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

Es comportarà millor l'alternativa que tingui major taxa d'encerts, que és l'escriptura immediata sense assignació

<6.4> ① **6.11.** Suposem que tenim un processador amb una memòria cache de dades amb les següents característiques:

- 64 conjunts
- 4 blocs per conjunt
- 32 bytes per bloc
- 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.

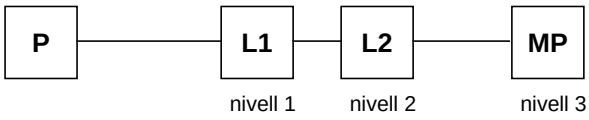
Com que hi ha 32 bytes per bloc i les paraules són de 4 bytes tenim que cada bloc conté 8 paraules.

La matriu A té 128 files i 1024 columnes i la seva adreça base es mapeja al primer byte del primer conjunt de la MC. Cada fila de la matriu ocupa 128 blocs. Els primers 64 blocs es mapegen a cadascun dels 64 conjunts que té la MC. Els segons 64 blocs de la fila es tornen a mapejar als 64 conjunts de la MC.

*En la versió A del codi es fa un recorregut per files de la matriu. Quan es recorre una fila, s'accedeixen els 128 blocs que ocupa. A cada bloc hi ha una fallada inicial, que carrega tot el bloc al conjunt corresponent, i els 7 accesos posterios són encerts. Al final de l'execució de cada fila la MC conté les 2 darreres files que s'han accedit a la matriu (cada fila ocupa 2 vies i la MC té 4 vies). El nombre de fallades en aquesta versió és 1 fallada a cada bloc dels 128 que hi ha en una fila per les 128 files de la matriu. Per tant, 1*128*128 = 16384 fallades.*

*En la versió B del codi es fa un recorregut per columnes de la matriu. Quan es recorre una columna s'accedeix a blocs que es mapegen tots al mateix conjunt de la MC. Com que la MC té 4 vies, només es guarden al conjunt corresponent els blocs de les 4 darreres files que s'han accedit. Quan comencem una nova columna mai trobarem a MC el bloc que conté la informació. En aquesta versió tots els accesos a la matriu provoquen una fallada. En total, hi ha 128*1024 = 131072 fallades.*

<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

Tenim que les adreces són de 20 bits per poder adreçar 1 MB de la MP.

A L1 tenim que aquest 20 bits es descomposen en: els dos bits de menys pes per indicar el desplaçament a nivell de byte dins un bloc; els següents dos bits especifiquen l'índex del bloc de la L1 que s'accedeix; i els darrers 16 bits, els de més pes, corresponen a l'etiqueta.

A L2 tenim 2 bits per especificar el desplaçament dins el bloc; a continuació tenim 3 bits que especifiquen l'index del conjunt que s'accedeix; i els 15 bits restants corresponen a l'etiqueta.

La següent taula mostra els encerts i fallades a cada nivell de MC desglossats per referència.

Adreça (dec i bin, darrers 8 bits)	#bloc L1	Encert/ Fallada L1	#conjunt L2	Encert/ Fallada L2
0 (...00000000)	0	Fallada	0	Fallada
5 (...00000101)	1	Fallada	1	Fallada
10 (...00001010)	2	Fallada	2	Fallada
12 (...00001100)	3	Fallada	3	Fallada
34 (...00100010)	0	Fallada	0	Fallada
0 (...00000000)	0	Fallada	0	Encert
66 (...01000010)	0	Fallada	0	Fallada
0 (...00000000)	0	Fallada	0	Encert
5 (...00000101)	1	Encert	-	-
10 (...00001010)	2	Encert	-	-
12 (...00001100)	3	Encert	-	-
34 (...00100010)	0	Fallada	0	Fallada
0 (...00000000)	0	Fallada	0	Encert
66 (...01000010)	0	Fallada	0	Fallada
0 (...00000000)	0	Fallada	0	Encert
5 (...00000101)	1	Encert	-	-
10 (...00001010)	2	Encert	-	-
12 (...00001100)	3	Encert	-	-
34 (...00100010)	0	Fallada	0	Fallada
0 (...00000000)	0	Fallada	0	Encert
66 (...01000010)	0	Fallada	0	Fallada

A partir d'aquest moment el comportament ja es va repetint per la mateixa seqüència d'adreces. Així, ja no es mostra per la tercera repetició de la seqüència.

En total hi ha 28 accessos a memòria. Tots ells van a L1. De tots aquests accessos només 19 van a L2, els que generen fallada a L1.

A L1 la taxa d'encerts és $h_1 = 9/28 = 0.32$

A L2 la taxa d'encerts és $h_2 = 7/19 = 0.37$

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).

Com que la mida de les pàgines és de 256 B necessitem 8 bits per especificar un desplaçament de pàgina.

El número de bits de les adreces virtuals és 24, que són els que es necessiten per adreçar l'espai d'adreces virtual de 16 MB. Com que hi ha 8 bits de desplaçament dins la pàgina, queden 16 bits per al VPN.

El número de bits de les adreces físiques és 14 (espai de 16 KB). El número de bits per al PPN és 6.

La taula de pàgines conté tantes entrades com VPN's té el sistema de memòria. En aquest cas 2^{16} entrades = 64 K entrades.

Cada entrada de la taula de pàgines conté el PPN corresponent al VPN de l'entrada més els bits de control, que són el de presència (P) i el de pàgina modificada (D). En toal, hi ha 8 bits per entrada.

La mida total de la TP és, doncs, de 64 KB.

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)	LRU
000023	0	SI	-	0
000101	1	SI	-	0, 1
000102	1	NO	-	0, 1
000200	2	SI	-	0, 1, 2
000010	0	NO	-	1, 2, 0
000111	1	NO	-	2, 0, 1
000416	4	SI	2	0, 1, 4
000520	5	SI	0	1, 4, 5
000101	1	NO	-	4, 5, 1

<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.

Aquest exercici complmenta l'exercici 6.11. Allà ja es va veure quin es el número de fallades que es produeixen en l'accés a la memòria cache.

Les pàgines tenen una mida de 8 KB, amb el que permeten emmagatzemar 2 files de la matriu. Tenint en compte que l'inici de la matriu correspon a l'inici d'una pàgina, la matriu ocupa un total de 64 pàgines. El TLB té 32 entrades que contenen la traducció de pàgines virtuals a pàgines físiques.

En la versió A del codi es fa un recorregut per files de la matriu. L'emmagatzematge de la matriu a memòria i el recorregut coincideixen. D'aquesta manera, només hi haurà una fallada de TLB quan es faci l'accés a l'element de la matriu emmagatzemat al principi d'una pàgina. La resta d'accesos als elements de la mateixa pàgina seran encerts de TLB. Com que la matriu ocupa 64 pàgines, el número total de fallades és 64.

*En la versió B del codi es fa un recorregut per columnes de la matriu. Quan s'accedeix a la primera columna es fa referència ja a totes les pàgines que contenen la matriu. Com que hi ha 64 pàgines i el TLB només pot emmagatzemar la traducció de 32 pàgines, quan s'acaba el recorregut de la columna ja s'han reemplaçat les entrades del TLB corresponents a les primeres pàgines. Per aquest motiu, quan s'inicia cada columna l'accés a cada nova pàgina generarà una fallada de TLB. En total, el nújmero de fallades de TLB és 64*1024 = 65536 fallades.*

<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.

*A continuació es mostra una taula per les 5 primeres iteracions del bucle, on per cada iteració s'indiquen les adreces de memòria que es referencien i les pàgines associades a aquestes adreces. Cal tenir en compte que a cada iteració s'incrementa el contingut de $t0 en 512*4 = 2048. Aquest és el registre base de l'adreça que s'utilitza en totes les instruccions d'accés a memòria. Això vol dir que no hi haurà un canvi en la pàgina que s'accedeix en una instrucció fins al cap de 4 iteracions.*

	1a iter.		2a iter.		3a iter.		4a iter.		5a iter.	
	@	pg	@	pg	@	pg	@	pg	@	pg
lw	A+0	0	A+2048	0	A+4096	0	A+6144	0	A+8192	1
lw	A+8192	1	A+10240	1	A+12288	1	A+14336	1	A+16384	2
sw	A+8192	1	A+10240	1	A+12288	1	A+14366	1	A+16384	2
sw	A+16384	2	A+18432	2	A+20480	2	A+22528	2	A+24576	3

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

Al primer cicle de 4 iteracions es fan referència a 3 pàgines diferents. Cada accés a una pàgina nova genera una fallada de TLB.

Als següents cicles de 4 iteracions s'accedeix a 2 pàgines que ja s'havien accedit a l'anterior cicle i a 1 pàgina nova. Com que el TLB té 4 entrades de capacitat llavors pot emmagatzemar durant tot el cicle les traduccions de les 3 pàgines que tracta.

En total hi ha 250 cicles de 4 iteracions en el programa.

*En total, el número de fallades de TLB és 3 * 1*249 = 252 fallades.*