

En C	Tipus	MIPS .data	#bits
char c; unsigned char c;	enter o caràcter ASCII natural	.byte 0	8
short s; unsigned short s;	enter natural	.half 0	16
int i; unsigned int i;	enter natural	.word 0	32
char *p; int *p; short *p; long long *p;	punter (natural)	.word 0	32
long long l; unsigned long long l;	enter natural	.dword 0	64

Mida dades	Instruccions d'accés	Significat	Restriccions d'alignement
8 bits	lb \$1, -100(\$2)	Load byte	Cap restricció
	lbu \$1, -100(\$2)	Load byte unsigned	
	sb \$1, -100(\$2)	Store byte	
16 bits	lh \$1, -100(\$2)	Load halfword	Adreces múltiples de 2
	lhu \$1, -100(\$2)	Load halfword unsigned	
	sh \$1, -100(\$2)	Store halfword	
32 bits	lw \$1, -100(\$2)	Load word	Adreces múltiples de 4
	sw \$1, -100(\$2)	Store word	

La variable punter *p* es pot inicialitzar en la seva pròpia declaració, indicant l'adreça de la variable a la que ha d'apuntar:

```

.data
var: .byte 'e'
p:   .word var
      # p 'apunta' a la variable "var" ja que conté la seva adreça

```

O bé, es pot inicialitzar en el codi:

```

.data
var: .byte 'e'
punter:.word 0

.text
la    $s0, var
la    $s1, punter
sw    $s0, 0($s1)

```

```
@v[i] = @v + (i * mida_d'un_element)
```

Crides al sistema operatiu del MIPS (Syscall)

Una crida al sistema és el mecanisme usat per una aplicació o un processador per sol·licitar un servei al sistema operatiu. MIPS posa a disposició del programador un determinat conjunt de crides al serveis del sistema operatiu, principalment per a operacions d'entrada/sortida.

En aquesta pràctica usarem aquestes crides per mostrar en pantalla resultats de l'execució.

Per fer servir les crides al sistema, el programa ha de seguir els següents passos:

- Pas 1. Copiar l'identificador del servei al registre \$v0.
- Pas 2. Copiar els arguments als registres específics. En les crides que farem servir únicament s'usa \$a0.
- Pas 3. Executar la instrucció *syscall*.
- Pas 4. Recuperar el valors de retorn, si és el cas, en els registres específics.

Les principals crides que usarem en aquestes pràctiques i que l'alumne ha de saber són ¹:

Syscall	Identificador	Arguments	Exemples
print integer	\$v0 = 1	\$a0 = Enter a imprimir	li \$v0, 1 li \$a0, 18 syscall # Mostra l'enter 18 en pantalla
print string	\$v0 = 4	\$a0 = Adreça inicial del string a imprimir	li \$v0, 4 la \$a0, string syscall # Mostra el string en pantalla
print character	\$v0 = 11	\$a0 = Caràcter a imprimir	li \$v0, 11 li \$a0, 'E' syscall # Mostra el caràcter 'E' en pantalla

Exercici 1.8: Traduïu a assemblador MIPS el següent programa escrit en C, omplint les caselles en blanc. Considereu que la variable *i* es guardarà al registre \$s0:

C	Assemblador MIPS
<pre>char cadena[6]={-1,-1,-1,-1,-1,-1}; unsigned int vec[5]={5, 6, 8, 9, 1}; void main() { int i=0; while (i < 5) { cadena[i]=vec[4-i] + '0'; i++; } cadena[5]=0; print_string(cadena); // consulteu lectura prèvia // main retorna al codi de startup }</pre>	<pre>.data cadena: .byte -1, -1, -1, -1, -1, -1 vec: .word 5, 6, 8, 9, 1 .globl main main: li \$s0, 0 while: li \$t0, 5 bge \$s0, \$t0, fi la \$t1, cadena la \$t2, vec sll \$t3, \$s0, 2 subu \$t2, \$t2, \$t3 lw \$t3, 16(\$t2) addiu \$t3, \$t3, 0x30 addu \$t1, \$t1, \$s0 sb \$t3, 0(\$t1) addiu \$s0, \$s0, 1 b while la \$t0, cadena sb \$zero, 5(\$t0) li \$v0, 4 move \$a0, \$t0 syscall jr \$ra</pre>

```
int mat1[5][6];
char mat2[3][5];
long C mat3[2][2];
int mat4[2][3] = {{2, 3, 1}, {2, 4, 3}};
```

```
.data
.align 2
mat1: .space 120
mat2: .space 15
    .align 3
mat3: .space 32
mat4: .word 2, 3, 1, 2, 4, 3
```

```
moda:
    addiu $sp, $sp, -60
    sw $s0, 40($sp)
    sw $s1, 44($sp)
    sw $s2, 48($sp)
    sw $s3, 52($sp)
    sw $ra, 56($sp)

    li $t0, 0
    li $t1, 10
    move $t2, $sp
for:   beq $t0, $t1, fi
        sw $zero, 0($t2)
        addiu $t2, $t2, 4
        addiu $t0, $t0, 1
        b for

fi:    move $s0, $a0
        move $s1, $a1
        li $s2, 0
        li $s3, '0'

for2:  beq $s2, $s1, fi2
        move $a0, $sp
        addu $a1, $s0, $s2
        lb $a1, 0($a1)
        addiu $a1, $a1, -48
        addiu $a2, $s3, -48
        jal update
        addiu $s3, $v0, '0'
        addiu $s2, $s2, 1
        b for2

fi2:   move $v0, $s3
        lw $s0, 40($sp)
        lw $s1, 44($sp)
        lw $s2, 48($sp)
        lw $s3, 52($sp)
        lw $ra, 56($sp)
        addiu $sp, $sp, 60
        jr $ra
```

```
update:
    addiu $sp, $sp, -16
    sw $s0, 0($sp)
    sw $s1, 4($sp)
    sw $s2, 8($sp)
    sw $ra, 12($sp)

    move $s0, $a0
    move $s1, $a1
    move $s2, $a2
    jal nofares

    sll $t0, $s1, 2
    addu $t0, $s0, $t0
    lw $t1, 0($t0)
    addiu $t1, $t1, 1
    sw $t1, 0($t0)

    sll $t2, $s2, 2
    addu $t2, $s0, $t2
    lw $t2, 0($t2)

    ble $t1, $t2, else
    move $v0, $s1
    b fi3

else: move $v0, $s2

fi3:  lw $s0, 0($sp)
        lw $s1, 4($sp)
        lw $s2, 8($sp)
        lw $ra, 12($sp)
        addiu $sp, $sp, 16
        jr $ra
```

```
char w[32] = "8754830094826456674949263746929";
/* digit ascii més freqüent */

main()
{
    resultat = moda(w, 31);
    print_character(resultat); /* consultar lectura prèvia sessió 1 */
}
char moda(char *vec, int num)
{
    int k, histo[10];
    char max;

    for (k=0; k<10; k++)
        histo[k] = 0;

    max = '0';
    for (k=0; k<num; k++)
        max = '0' + update(histo, vec[k]-'0', max-'0');

    return max;
}
void nofares();
char update(int *h, char i, char imax)
{
    nofares();
    h[i]++;
    if (h[i] > h[imax])
        return i;
    else
        return imax;
}
```

```

int mat1[5][6];
int mat4[2][3] = {{2, 3, 1}, {2, 4, 3}};
int col = 2;

main()
{
    mat1[4][3] = subr(mat4, mat4[0][2], col);
    mat1[0][0] = subr(mat4, 1, 1);
}

int subr(int x[][3], int i, int j)
{
    mat1[j][5] = x[i][j];
    return i;
}

int mat[4][6] = { {0, 0, 2, 0, 0, 0},
                  {0, 0, 4, 0, 0, 0},
                  {0, 0, 6, 0, 0, 0},
                  {0, 0, 8, 0, 0, 0} };
int resultat;

main()
{
    resultat = suma_col(mat);
}

int suma_col(int m[][6])
{
    int i, suma = 0;
    int *p;

    p = &m[0][2]; /* inicialitzar el punter */
    for(i = 0; i < 4; i++) {
        suma += *p; /* accés a m usant el punter */
        p += 6; /* incrementar el punter */
    }

    return suma;
}

```

```

.data
.align 2
mat1: .space 120
mat4: .word 2, 3, 1, 2, 4, 3
col: .word 2

.text
.globl main

main:
    addiu $sp, $sp, -4
    sw $ra, 0($sp)

    la $a0, mat4
    lw $a1, 8($a0)
    la $a2, col
    lw $a2, 0($a2)
    jal subr

    la $t0, mat1
    sw $v0, 108($t0)

    la $a0, mat4
    li $a1, 1
    move $a2, $a1
    jal subr

    la $t0, mat1
    sw $v0, 0($t0)

    lw $ra, 0($sp)
    addiu $sp, $sp, 4
    jr $ra

```

```

.data
mat: .word 0, 0, 2, 0, 0, 0
      .word 0, 0, 4, 0, 0, 0
      .word 0, 0, 6, 0, 0, 0
      .word 0, 0, 8, 0, 0, 0

resultat: .word 0

.suma_col:
    move $t0, $zero
    move $t1, $a0
    addiu $t1, $t1, 8

    move $t2, $zero
    li $t3, 4

    .text
    .globl main

main:
    addiu $sp, $sp, -4
    sw $ra, 0($sp)

    la a0, mat
    jal suma_col

    la $t0, resultat
    sw $v0, 0($t0)

    lw $ra, 0($sp)
    addiu $sp, $sp, 4
    jr $ra

```

```

subr:
    la $t0, mat1
    li $t1, 6
    mult $a2, $t1
    mflo $t1
    addiu $t1, $t1, 5
    sll $t1, $t1, 2
    addiu $t0, $t0, $t1

    li $t1, 3
    mult $a1, $t1
    mflo $t1
    addiu $t1, $t1, $a2
    sll $t1, $t1, 2
    addu $t1, $t1, $a0

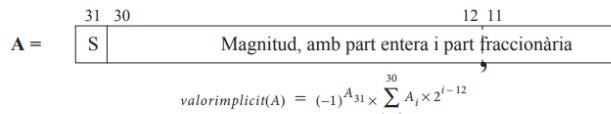
    lw $t1, 0($t1)
    sw $t1, 0($t0)

    move $v0, $a1
    jr $ra

```

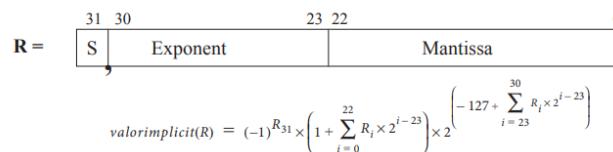
El format de coma fixa (format inicial)

El format de coma fixa del número A és el següent: té 32 bits, el bit 31 conté el *signe* (0 per als positius) i la resta de bits del 30 al 0 representen la *magnitud*, codificada en el sistema posicional de base 2, amb 19 bits a l'esquerra de la coma (part entera) i 12 bits a la dreta (part fraccionària):



El format de coma flotant IEEE-754 de simple precisió (format final)

El format de coma flotant del número R, de simple precisió definit a l'estàndard IEEE-754 també té 32 bits: el bit 31 conté el signe (0 per als positius); els 23 bits de menys pes contenen la mantissa, en base 2, fraccionària, normalitzada, amb bit ocult i coma a la dreta d'aquest bit; i els restants 8 bits (del 23 al 30) contenen l'exponent, un enter codificat en excés a 127:



```
descompon:
    slt $t0, $a0, $zero
    sw $t0, 0($a1)
    sll $a0, $a0, 1
```

```
bne $a0, $zero, else
    li $t0, 0
    b endif
else: li $t0, 18
```

```
while: blt $a0, $zero, fi_w
    sll $a0, $a0, 1
    addiu $t0, $t0, -1
    b while
fi_w: sra $a0, $a0, 8
    li $t1, 0xFFFF
    and $a0, $a0, $t1
    addiu $t0, $t0, 127
```

```
endif: sw $t0, 0($a2)
    sw $a0, 0($a3)
    jr $ra
```

referència en cas d'encert (t_h) més el temps de penalització per resoldre la referència en accedir al següent nivell de la jerarquia de memòria (t_p):

$$t_{\text{accés}} = t_h + t_p$$

$$\text{CPI}_{\text{ideal}} = n_{\text{cicles_ideal}} / n_{\text{ins}}$$

$$t_{\text{exe}} = t_c \cdot n_{\text{ins}} \cdot \text{CPI}_{\text{real}} = t_c \cdot (n_{\text{ins}} \cdot \text{CPI}_{\text{ideal}} + n_{\text{fallades}} \cdot t_p)$$

```
void descompon(int cf, int *s, int *e, int *m) {
    int exp;
    *s = (cf < 0);                                // si és negatiu val 1, sinó 0
    cf = cf << 1;                                   // elimina el signe
    if (cf == 0)                                     // el número és un +0 o un -0
        exp = 0;
    else {
        exp = 18;
        while (cf >= 0) {
            cf = cf << 1;
            exp--;
        }
        cf = (cf >> 8) & 0x7FFFFFF;                // alinear i eliminar bit ocult
        exp = exp + 127;                            // codificar en excés a 127
    }
    *e = exp;                                       // guarda exponent resultant
    *m = cf;                                         // guarda mantissa resultant
}
```

Mida de la MC. La MC s'organitza en blocs que contenen un cert nombre de paraules. El producte del nombre de blocs pel nombre de paraules per bloc estableix la capacitat de la MC quant a dades emmagatzemables de l'espai adreçable pel processador.

Algorisme d'emplaçament. Especifica en quina entrada de la MC s'ubica el bloc de MP que conté la paraula de memòria que accedeix el processador.

- Correspondència directa:** A cada bloc de MP li correpon una entrada fixa de MC.
- Completament associativa:** Cada bloc de MP pot anar a qualsevol entrada de MC.
- Associativa per conjunts:** La MC s'organitza en conjunts, que contenen un cert nombre de blocs cada un. A cada bloc de MP li correpon un conjunt fix de MC, i el bloc pot anar a qualsevol de les entrades que té el conjunt.
- LRU:** S'elimina el bloc que fa més temps que no es referencia.

Política d'escriptura. Estableix com es gestionen les escriptures a memòria. D'una banda cal decidir si convé copiar a la MC el bloc de MP quan l'escriptura provoca una fallada i d'altra banda cal mantenir la coherència de les dues memòries (MC i MP). La combinació d'aquests dos aspectes ens dóna les següents alternatives:

- Escriptura immediata amb assignació:** En cas de fallada en una escriptura es copia el bloc que s'accedeix a MC. Es realitza l'escriptura tant a MC com a MP. Aquesta política és l'única que es considera en el simulador MARS.
- Escriptura immediata sense assignació:** En cas de fallada en una escriptura no es porta el bloc a MC i l'escriptura es realitza únicament a MP. Si l'escriptura genera un encert la paraula s'escriu tant a MC com a MP.
- Escriptura retardada amb assignació:** En cas de fallada en una escriptura es porta el bloc a MC. L'escriptura es realitza únicament a MC. La MP s'actualitzarà quan el bloc on es realitza l'escriptura s'hagi de reemplaçar, en un moment posterior de l'execució del programa. Per a aquest efecte, s'utilitza el bit D (bit de bloc modificat).

t_p	Immediata amb assignació	Immediata sense assignació	Retardada amb assignació
Lectura - Encert	0	0	0
Lectura - Fallada	$t_{\text{block}} + t_h^1$	$t_{\text{block}} + t_h^1$	$t_{\text{block}} + t_h^2$ $t_{\text{block}} + t_h^1$
Escriptura - Encert	0^3	0^3	0^5
Escriptura - Fallada	$t_{\text{block}} + t_h^1$	0^4	$t_{\text{block}} + t_h^2$ $t_{\text{block}} + t_h^1$

```

int V1[16], M[16][16], V2[16];
main() {
    int i,j,tmp;

    for (i=0; i<16; i++) {
        tmp = 0;
        for (j=0; j<16; j++)
            tmp += M[i][j] * V2[j];
        V1[i] = tmp;
    }
}

```

Figura 5.4: Programa que realitza el producte d'una matriu per un vector.

El fitxer **s5d.s** conté aquest programa traduït a assemblador MIPS.

Determina mitjançant el simulador MARS quina és la taxa d'encerts que s'obté executant el programa de la figura 5.4. Considera que els blocs de la memòria cache són de 4 paraules. Calcula aquesta taxa per a diferents mides de la cache completant la següent taula (dins de cada fila suposem que la mida total no canvia, sols l'associativitat):

Capacitat de la cache	Correspondència directa	Associativa de grau 2	Associativa de grau 4	Associativa de grau 8	Completament Associativa
8 blocs	43%	81%	78%	73%	73%
16 blocs	65%	86%	86%	86%	86%
32 blocs	76%	86%	86%	86%	86%
64 blocs	82%	86%	86%	86%	86%
128 blocs	86%	86%	86%	86%	86%

Exercici 5.13: El codi que hi ha a continuació correspon a una versió optimitzada del programa de la figura 5.4, on s'hi ha aplicat l'optimització anomenada "tiling". L'índex k itera els 4 blocs de la matriu (amb 4 columnes per bloc). Completa les caselles buides amb les sentències d'alt nivell que contenen els accessos a les estructures de dades.

```

int V1[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
int M[16][16], V2[16];

main()
{
    int i,j,k,tmp;

    for (k=0; k<4; k++)
        for (i=0; i<16; i++)
        {
            tmp = 0;
            for (j=0; j<4; j++)
            {
                tmp += M[i] [k*4+j] * V2 [k*4+j];
            }
            V1[i] += tmp;
        }
}

```

```

V1: .space 64
M: .word 0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1
     .word 0,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0
     .word 1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0
     .word 0,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0
     .word 1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0
     .word 0,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0
     .word 1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0
     .word 0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1
     .word 0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1
     .word 0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1
     .word 0,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0
     .word 1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0
     .word 0,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0
     .word 1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0
     .word 0,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0
     .word 0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1
     .word 0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1
     .word 1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0
     .word 0,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0
     .word 1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0
     .word 0,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0
     .word 0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1
     .word -5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10

.text
.globl main

main:
    move $t0, $zero
    li $t1, 4
    move $t3, $zero

fork:
    bge $t0, $t1, ffork
    li $t2, 16
    move $t3, $zero

fori:
    bge $t3, $t2, ffori
    move $t5 $zero
    move $t6, $zero

fforj:
    bge $t6, $t1, fforj
    sll $t7, $t0, 2
    addu $t7, $t6, $t7
    la $s0, M
    la $s1, V2
    sll $s2, $t3, 4
    addu $s2, $s2, $t7
    sll $s2, $s2, 2
    addu $s0, $s0, $s2
    lw $s0, 0($s0)
    sll $s3, $t7, 2
    addu $s1, $s3, $s1
    lw $s1, 0($s1)
    mult $s0, $s1
    mflo $s0
    addu $t5, $t5, $s0
    addiu $t6, $t6, 1
    b forj

fforj:
    la $s4, V1
    sll $s5, $t3, 2
    addu $s4, $s5, $s4
    lw $s6, 0($s4)
    addu $s6, $s6, $t5
    sw $s6, 0($s4)
    addiu $t3, $t3, 1
    b fori

ffori:
    addiu $t0, $t0, 1
    b fork

ffork:
    jr $ra

```

Capacitat de la cache	Correspondència directa	Associativa de grau 2	Associativa de grau 4	Associativa de grau 8	Completament Associativa
8 blocs	52%	84%	87%	87%	87%
16 blocs	70%	88%	88%	88%	87%
32 blocs	79%	88%	88%	88%	89%
64 blocs	84%	89%	89%	89%	89%
128 blocs	89%	89%	89%	89%	89%

Variables Globales Ensamblador

Tipo C	MIPS	Tamaño
(<i>unsigned</i>) char	.byte	1 byte
(<i>unsigned</i>) short	.half	2 bytes
(<i>unsigned</i>) int / long	.word	4 bytes (1 word)
(<i>unsigned</i>) long long	.dword	8 bytes (2 words)

```
unsigned char a;
short x = 13;
char b = -1, c = 10;
int y = 0x13457AB2;
long long d = 0x12038034a3f00001;
```

Declaraciones en C

```
.data
a: .byte 0
x: .half 13
b: .byte -1
c: .byte 10
y: .word 0x13457AB2
d: .dword 0x12038034a3f00001
```

Declaraciones en MIPS

Las Primeras Instrucciones

addu \$t1, \$t2, \$t3 # \$t1 = \$t2 + \$t3

- En **addu**, los operandos sólo pueden ser registros

addiu \$t1, \$t2, 29 # \$t1 = \$t2 + 29

- El inmediato (29) es un número entero ca2 de 16 bits.

Instrucciones de suma y resta

addu rd, rs, rt	$rd = rs + rt$	
addiu rd, rs, imm16	$rd = rs + \text{SignExt}(imm16)$	Inmediato de 16 bits en ca2
subu rd, rs, rt	$rd = rs - rt$	

Instrucciones load / store		
lw rt off16(rs)	$rt = M_w[rs + \text{SignExt}(off16)]$	Load word
lh rt off16(rs)	$rt = \text{SignExt}(M_h[rs + \text{SignExt}(off16)])$	Load half (extiende signo)
lhu rt off16(rs)	$rt = M_h[rs + \text{SignExt}(off16)]$	Load half (extiende ceros)
lb rt off16(rs)	$rt = \text{SignExt}(M_b[rs + \text{SignExt}(off16)])$	Load byte (extiende signo)
lbu rt off16(rs)	$rt = M_b[rs + \text{SignExt}(off16)]$	Load byte (extiende ceros)
sw rt off16(rs)	$M_w[rs + \text{SignExt}(off16)] = rt$	Store word
sh rt off16(rs)	$M_h[rs + \text{SignExt}(off16)] = rt_{15:0}$	Store Half
sb rt off16(rs)	$M_b[rs + \text{SignExt}(off16)] = rt_{7:0}$	Store Byte

Repertorio instrucciones de desplazamiento

sll, srl, sra, sllv, srav, srav		
sll rd, rt, shamt	$rd = rt \ll shamt$	
srl rd, rt, shamt	$rd = rt \gg shamt$	Inserta 0's a la izquierda
sra rd, rt, shamt	$rd = rt \gg shamt$	Extiende signo a la izquierda
sllv rd, rt, rs	$rd = rt \ll rs_{4:0}$	
srlv rd, rt, rs	$rd = rt \gg rs_{4:0}$	Inserta 0's a la izquierda
srav rd, rt, rs	$rd = rt \gg rs_{4:0}$	Extiende signo a la izquierda

- Las instrucciones **sllv**, **srlv** y **srav** permiten que el desplazamiento sea calculado.
 - El desplazamiento son los 5 bits de menor peso del registro rs ,

Instrucciones de Comparación en MIPS

- ❑ MIPS sólo implementa la función “<”
 - Devuelve un 0 si FALSO
 - Devuelve un 1 si CIERTO

slt, sltu, slti, sltiu		
slt rd, rs, rt	$rd = rs < rt$	Comparación de Enteros
sltu rd, rs, rt	$rd = rs < rt$	Comparación de Naturales
slti rt, rs, imm16	$rt = rs < \text{SignExt}(imm16)$	Comparación de Enteros
sltiu rt, rs, imm16	$rt = rs < \text{SignExt}(imm16)$	Comparación de Naturales

Otros Saltos condicionales relativos al PC (macros)

- ❑ Es muy habitual necesitar instrucciones de salto con condiciones del tipo $>$, $<$, \geq , \leq
- ❑ Usaremos macros

blt, bgt, ble, bge, bltu, bgtu, bgeu, bleu		
blt rs, rt, label	si ($rs < rt$) saltar a label	<code>slt \$at, rs, rt bne \$at, \$zero, label</code>
bgt rs, rt, label	si ($rs > rt$) saltar a label	<code>slt \$at, rt, rs bne \$at, \$zero, label</code>
bge rs, rt, label	si ($rs \geq rt$) saltar a label	<code>slt \$at, rs, rt beq \$at, \$zero, label</code>
ble rs, rt, label	si ($rs \leq rt$) saltar a label	<code>slt \$at, rt, rs beq \$at, \$zero, label</code>

- ❑ Saltos para enteros, para naturales usaremos las macros **bltu, bgtu, bgeu, bleu**

Saltos condicionales relativos al PC (branch)

¿Cómo codificamos el salto a una etiqueta determinada?

- ❑ Codificamos la distancia a saltar respecto al PC en un inmediato de 16 bits.

El PC es un registro interno que apunta a la instrucción que se está ejecutando.

- ❑ La distancia se codifica en número de instrucciones
- ❑ La distancia se calcula respecto al PC de la siguiente instrucción al salto ($PC_{UP} = PC + 4$)
- ❑ Permite saltar en un rango de $[-2^{15}..2^{15}-1]$ instrucciones de distancia respecto a PC_{UP} .

beq, bne y la macro b		
beq rs, rt, label	si ($rs == rt$) $PC = PC_{UP} + \text{SigExt}(\text{offset16} * 4)$	
bne rs, rt, label	si ($rs != rt$) $PC = PC_{UP} + \text{SigExt}(\text{offset16} * 4)$	
b label	$PC = PC_{UP} + \text{SigExt}(\text{offset16} * 4)$	<code>beq \$0, \$0, label</code>

Subrutinas – Llamada y retorno

- ❑ Instrucciones que usamos para llamar y retornar de subrutinas: **jal, jalr, jr**

j, jr		
jal target	$PC = \text{target}, \$ra = PC_{UP}$	Jump and Link, modo pseudodirecto
jalr rs, rd	$PC = rs, rd = PC_{UP}$	Jump and Link, modo registro
jr rs	$PC = rs$	Jump, modo registro

- ❑ La dirección de retorno es el PC de la instrucción que hay después de la llamada ($PC_{UP} = PC + 4$)

Subrutinas – Variables Locales

- Las **variables locales** se declaran dentro de una función y se crean y se destruyen con cada llamada.
- Las **variables locales** se han de inicializar de forma explícita, sino el valor es indeterminado.

¿Dónde guardamos las variables locales?

- Si son variables escalares se guardan en registros :
 - Floats de simple precisión: **\$f0-\$f31**
 - El resto: **\$t0-\$t9, \$s0-\$s7, \$v0-\$v1**
- Algunas variables se han de almacenar en memoria [en la pila (stack)]:
 - Si son de tipo estructurado (vectores, matrices, tuplas, ...)
 - Si una variable local **v** aparece con **&v**
 - Si nos quedamos sin registros

```
int f(...) {  
    int i, sum = 0;  
    char frase[100];  
    ...  
}
```

Subrutinas Multinivel – Ejemplo paso a paso

PASO 3. Programar la rutina

```
multi:  
    addiu $sp,$sp,-12  
    sw $s0,0($sp)  
    sw $s1,4($sp)  
    sw $ra,8($sp)  
    move $s0,$a2  
  
    addu $s1,$a0,$a1  
    move $a0,$a2  
    move $a1,$s1  
    jal mcm  
    addu $t0,$s0,$s1  
    addu $v0,$v0,$t0  
  
    lw $s0,0($sp)  
    lw $s1,4($sp)  
    lw $ra,8($sp)  
    addiu $sp,$sp,12  
    jr $ra
```

PRÓLOGO

EPÍLOGO

```
int multi(int a, int b, int c) {  
    int d, e;  
    d = a + b;  
    e = mcm(c, d);  
    return c + d + e;  
}
```

BLOQUE de ACTIVACIÓN



Instrucciones para Multiplicar Enteros en MIPS

- Instrucción en MIPS para multiplicar 2 números enteros de 32 bits

```
mult rs, rt # $hi:$lo = rs * rt
```



- El resultado se guarda en los registros **\$hi** y **\$lo**

- Son **registros especiales**. No se pueden usar en las instrucciones que hemos visto.
- Para acceder a estos registros necesitamos instrucciones específicas.

```
mflo rd    # rd = $lo  
mfhi rd    # rd = $hi
```

Instrucciones para Dividir en MIPS

- Instrucción en MIPS para dividir 2 números de 32 bits

```
div  rs, rt # $lo = rs/rt; $hi = rs%rt enteros  
divu rs, rt # $lo = rs/rt; $hi = rs%rt naturales
```

- Los 2 resultados se guardan en los registros **\$hi** y **\$lo**

- Son registros especiales. No se pueden usar en las instrucciones que hemos visto
- Para acceder a estos registros necesitamos instrucciones especiales

```
mflo rd    # rd = $lo  
mfhi rd    # rd = $hi
```

Estándar IEEE-754

- Simple Precisión (32b)



- Doble Precisión (64b)



Signo: 1 bit (0, positivo; 1, negativo)

Exponente: 8 bits / 11 bits

- Codificado en exceso a 127 / 1023
- Permite comparar magnitudes con un comparador de naturales

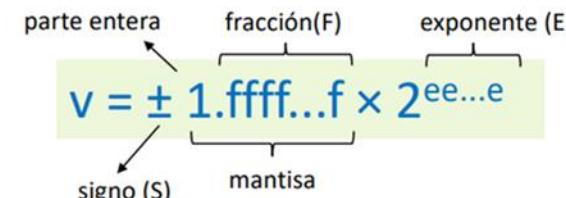
Fracción: 23 bits / 52 bits

- Parte fraccionaria de la mantisa

Parte entera = 1

- Bit oculto implícito, no se representa

Coma Flotante en base 2



$$v = (-1)^S \times (1+0.F) \times 2^E$$

Un forma más compacta.

- Formato: Signo, Exponente, Fracción

$$\text{S EEEEEEEE FFFFFFFFFFFFFFFFFFFF}$$

- Signo:** 0 = positivo, 1 = negativo
- Exponente:** entero representado "en exceso"
- Mantisa:** NORMALIZADA
 - ✓ La parte entera siempre vale 1, es implícita y no se representa (bit oculto)
 - ✓ Sólo se codifica la fracción F

IEEE-754, Modos de Redondeo

Regla práctica para redondear al más próximo:

- Calculamos el resultado con algunos bits extra de precisión
- Examinamos el **primer bit extra de la mantisa** y tenemos 3 casos:

a) El bit es 0, redondeamos al anterior (v_0):

$$v = 1.\underset{\text{xxxxx}}{\text{xxxxx}}\dots0011 \quad 0000101$$

$$v_0 = 1.\underset{\text{xxxxx}}{\text{xxxxx}}\dots0011$$

b) El bit es 1, y el resto no son todo ceros, redondeamos al siguiente (v_1):

$$v = 1.\underset{\text{xxxxx}}{\text{xxxxx}}\dots0011 \quad 1010110$$

$$+ 0.00000\dots0001$$

$$v_1 = 1.\underset{\text{xxxxx}}{\text{xxxxx}}\dots0100$$

c) El bit es 1, y el resto son todos cero

$$v = 1.\underset{\text{xxxxx}}{\text{xxxxx}}\dots0011 \quad 1000000$$

v es equidistante de v_0 y v_1 , redondeamos al que sea par

$$v_0 = 1.\underset{\text{xxxxx}}{\text{xxxxx}}\dots0011$$

$$v_1 = 1.\underset{\text{xxxxx}}{\text{xxxxx}}\dots0100 \leftarrow \text{"par"}$$

Conversión de binario (coma flotante) a base 10

Queremos saber que valor tiene $v = 0x45814140$

1. Escribirlo en binario

$$v = 0100\ 0101\ 1000\ 0001\ 0100\ 0001\ 0100\ 0000$$

2. Identificamos los 3 campos: signo, exponente, fracción

$$v = 0\ 10001011\ 000001010000101000000$$

3. Exponente. Lo pasamos a decimal y le restamos el exceso (127)

$$10001011 = 139$$

$$E = 139 - 127 = 12$$

4. Componer el número: signo, bit oculto, fracción y exponente

$$v = +1.000000101000010100000 \times 2^{12}$$

5. Punto Fijo. Mover la coma 12 posiciones a la derecha y eliminar ceros

$$v = +1000000101000.0010100000$$

6. Convertir la parte entera a base 10

$$1000000101000 = 1 \cdot 2^{12} + 1 \cdot 2^5 + 1 \cdot 2^3 = 4136$$

7. Convertir la fracción a base 10 (1º desplazaremos la coma)

$$0.00101 = 101 \times 2^{-5} = 5/32 = 0,15625$$

8. Finalmente, juntar la parte entera y la fracción:

$$v = 4136,15625$$

Subrutinas con Coma Flotante

- ❑ Paso de parámetros y resultados
 - Parámetros en **\$f12** y **\$f14**
 - Resultado en **\$f0**
 - Registros Seguros: del **\$f20** al **\$f31**

NOTA: La mezcla de parámetros en coma flotante con enteros, sigue en MIPS una reglas muy complejas que no estudiaremos en EC. Sólo estudiaremos un caso: cuando tengamos exclusivamente 1 o 2 parámetros de tipo **float**.

- ❑ Ejemplo de subrutina

```
float func(float x) {  
    if (x < 1.0)  
        return x * x;  
    else  
        return 2.0 - x;  
}
```

Subrutinas con Coma Flotante, ejemplo de traducción

```
.data  
uno: .float 1.0  
.text  
...  
func: la $t0,uno  
      lwc1 $f16,0($t0)      # f16 = 1.0  
      c.lt.s $f12,$f16       # ¿x<1.0?  
      bclt else              # si false  
      mul.s $f0,$f12,$f12    # x*x  
      b end  
else: add.s $f16,$f16,$f16 # f16 = 2.0  
      sub.s $f0,$f16,$f12    # 2.0-x  
end:  jr $ra
```

```
float func(float x) {  
    if (x < 1.0)  
        return x * x;  
    else  
        return 2.0 - x;  
}
```

Calculamos la constante 2.0, dejamos el resultado en \$f0 y acabamos

❑ Comparación

Simple Precisión (float)	Doble Precisión (double)	Comportamiento (float)
c.eq.s fs,ft	c.eq.d fs,ft	cc = (fs == ft)
c.lt.s fs,ft	c.lt.d fs,ft	cc = (fs < ft)
c.le.s fs,ft	c.le.d fs,ft	cc = (fs <= ft)

- El resultado de la comparación se escribe en un **bit de condición (cc)** de un registro interno

❑ Instrucciones de salto

Instrucción	Comportamiento
bclt etiq	salta si cc = TRUE (1)
bclf etiq	salta si cc = FALSE (0)

Algoritmos de Emplazamiento

❑ Memoria Cache ASOCIATIVA por CONJUNTOS (X-Asociativa, X-way)

- La Cache tiene S conjuntos ($S=2^s$) y X líneas por Conjunto
- Un Bloque de MP se almacena en 1 Conjunto de MC, pero dentro del Conjunto en cualquier línea

❑ Memoria Cache DIRECTA

- La Cache tiene D líneas ($D=2^d$)
- Un bloque de MP se almacena en 1 línea de MC

❑ Memoria Cache COMPLETAMENTE ASOCIATIVA

- La Cache tiene un solo conjunto de M líneas, M puede ser cualquier valor.
- Un Bloque de MP se almacena en cualquier línea de MC

@	#línea MP	#byte
n-b		b

MC Asociativa por Conjuntos

TAG	#set MC	#byte
n-s-b	s	b

MC Directa

TAG	#línea MC	#byte
n-d-b	d	b

MC Completamente Asociativa

TAG	#byte
n-b	b

Operaciones a realizar en un acceso a Cache

```
0x0003 =  
0000 0000 0000 0011 =  
000000000000 00 011  
TAG = 0x000  
línea = 0x0  
byte = 0x3  
Bloque MP = 0x0000
```

Lectura 0x0003: **MISS**
Lectura 0x0004
Lectura 0x0009
Lectura 0x0045
Lectura 0x0047

Operaciones a realizar en un acceso a Cache

```
0x0009 =  
0000 0000 0000 1001 =  
000000000000 01 001  
TAG = 0x000  
línea = 0x1  
byte = 0x1  
Bloque MP = 0x0001
```

Lectura 0x0003: **MISS**
Lectura 0x0004: **HIT**
Lectura 0x0009: **MISS**
Lectura 0x0045
Lectura 0x0047

Operaciones a realizar en un acceso a Cache

```
0x0004 =  
0000 0000 0000 0100 =  
000000000000 00 100  
TAG = 0x000  
línea = 0x0  
byte = 0x4  
Bloque MP = 0x0000
```

Lectura 0x0003: **MISS**
Lectura 0x0004: **HIT**
Lectura 0x0009
Lectura 0x0045
Lectura 0x0047

V	Etiqueta	DATOS MC						
0	1	0x000	00	03	05	08	F3	87
1	0							AD
2	0							
3	0							

Operaciones a realizar en un acceso a Cache

```
0x0047 =  
0000 0000 0100 0101 =  
000000000010 00 111  
TAG = 0x002  
Linea = 0x0  
byte = 0x7  
Bloque MP = 0x0008
```

Lectura 0x0003: **MISS**
Lectura 0x0004: **HIT**
Lectura 0x0009: **MISS**
Lectura 0x0045: **MISS**
Lectura 0x0047: **HIT**

Operaciones a realizar en un acceso a Cache

0x0003 =
 0000 0000 0000 0111 =
 000000000000 00 111
 TAG = 0x000
 línea = 0x0
 byte = 0x3
 Bloque MP = 0x0000

Escr 0x0003: **MISS**
 Lect 0x0004:
 Escr 0x0007:
 Lect 0x001F:
 Escr 0x001A:

V	Etiqueta	DATOS MC								
0	1	0x002	1C	54	00	00	FF	AA	99	5E
1	1	0x000	45	67	89	87	67	90	99	34
2	0									
3	0									

Bloque MP	DATOS MP							
0x0000	00	03	05	FF	F3	87	AD	F4
0x0001	45	67	89	87	67	90	99	34
0x0002	80	00	23	80	DE	ED	90	54
0x0003	99	AA	DA	80	EF	ED	12	67
0x0004	90	78	AE	10	FE	4E	44	23
0x0005	71	32	F5	10	FE	4E	55	23
0x0006	1A	44	F2	11	FE	64	66	F3
0x0007	1B	23	FF	EE	FF	53	88	7F
0x0008	1C	54	00	00	FF	AA	99	5E
0x0009	BB	34	00	00	34	55	98	4E
0x000A	B4	45	00	00	55	44	45	1E
0x000B	B5	78	00	00	FF	FF	56	12
0x000C	B6	67	56	00	00	CC	DE	E1
...
0x1FFF	FF	FF	FF	FF	FF	FF	FF	FF

Write Through + Write NO Allocate

Operaciones a realizar en un acceso a Cache

0x0007 =
 0000 0000 0000 0111 =
 000000000000 00 111
 TAG = 0x000
 línea = 0x0
 byte = 0x7
 Bloque MP = 0x0000

Escr 0x0003: **MISS**
 Lect 0x0004: **MISS**
 Escr 0x0007: **HIT**
 Lect 0x001F:
 Escr 0x001A:

V	Etiqueta	DATOS MC								
0	1	0x000	00	03	05	FF	F3	87	AD	00
1	1	0x000	45	67	89	87	67	90	99	34
2	0									
3	0									

Bloque MP	DATOS MP							
0x0000	00	03	05	FF	F3	87	AD	00
0x0001	45	67	89	87	67	90	99	34
0x0002	80	00	23	80	DE	ED	90	54
0x0003	99	AA	DA	80	EF	ED	12	67
0x0004	90	78	AE	10	FE	4E	44	23
0x0005	71	32	F5	10	FE	4E	55	23
0x0006	1A	44	F2	11	FE	64	66	F3
0x0007	1B	23	FF	EE	FF	53	88	7F
0x0008	1C	54	00	00	FF	AA	99	5E
0x0009	BB	34	00	00	34	55	98	4E
0x000A	B4	45	00	00	55	44	45	1E
0x000B	B5	78	00	00	FF	FF	56	12
0x000C	B6	67	56	00	00	CC	DE	E1
...
0x1FFF	FF	FF	FF	FF	FF	FF	FF	FF

Write Through + Write NO Allocate

Operaciones a realizar en un acceso a Cache

0x0004 =
 0000 0000 0000 0100 =
 000000000000 00 100
 TAG = 0x000
 línea = 0x0
 byte = 0x4
 Bloque MP = 0x0000

Escr 0x0003: **MISS**
 Lect 0x0004: **MISS**
 Escr 0x0007:
 Lect 0x001F:
 Escr 0x001A:

Bloque MP	DATOS MP							
0x0000	00	03	05	FF	F3	87	AD	F4
0x0001	45	67	89	87	67	90	99	34
0x0002	80	00	23	80	DE	ED	90	54
0x0003	99	AA	DA	80	EF	ED	12	67
0x0004	90	78	AE	10	FE	4E	44	23
0x0005	71	32	F5	10	FE	4E	55	23
0x0006	1A	44	F2	11	FE	64	66	F3
0x0007	1B	23	FF	EE	FF	53	88	7F
0x0008	1C	54	00	00	FF	AA	99	5E
0x0009	BB	34	00	00	34	55	98	4E
0x000A	B4	45	00	00	55	44	45	1E
0x000B	B5	78	00	00	FF	FF	56	12
0x000C	B6	67	56	00	00	CC	DE	E1
...
0x1FFF	FF	FF	FF	FF	FF	FF	FF	FF

Write Through + Write NO Allocate

Operaciones a realizar en un acceso a Cache

0x001F =
 0000 0000 0001 1111 =
 000000000000 11 111
 TAG = 0x000
 línea = 0x11
 byte = 0x7
 Bloque MP = 0x0003

Escr 0x0003: **MISS**
 Lect 0x0004: **MISS**
 Escr 0x0007: **HIT**
 Lect 0x001F:
 Escr 0x001A:

Bloque MP	DATOS MP							
0x0000	00	03	05	FF	F3	87	AD	00
0x0001	45	67	89	87	67	90	99	34
0x0002	80	00	23	80	DE	ED	90	54
0x0003	99	AA	DA	80	EF	ED	12	67
0x0004	90	78	AE	10	FE	4E	44	23
0x0005	71	32	F5	10	FE	4E	55	23
0x0006	1A	44	F2	11	FE	64	66	F3
0x0007	1B	23	FF	EE	FF	53	88	7F
0x0008	1C	54	00	00	FF	AA	99	5E
0x0009	BB	34	00	00	34	55	98	4E
0x000A	B4	45	00	00	55	44	45	1E
0x000B	B5	78	00	00	FF	FF	56	12
0x000C	B6	67	56	00	00	CC	DE	E1
...
0x1FFF	FF	FF	FF	FF	FF	FF	FF	FF

Write Through + Write NO Allocate

Operaciones a realizar en un acceso a Cache

$0x001A =$
 $0000\ 0000\ 0001\ 1010 =$
 $000000000000\ 11\ 010$
 $TAG = 0x000$
 $\text{l\'inea} = 0x11$
 $\text{byte} = 0x2$
 Bloque MP = 0x0003

Escr 0x0003: **MISS**
 Lect 0x0004: **MISS**
 Escr 0x0007: **HIT**
 Lect 0x01F: **MISS**
 Escr 0x01A: **HIT**

	V	Etiqueta	DATOS MC							
0	1	0x000	00	03	05	FF	F3	87	AD	00
1	1	0x000	45	67	89	87	67	90	99	34
2	0									
3	1	0x000	99	AA	11	80	EF	ED	12	67

Bloque MP	DATOS MP							
0x0000	00	03	05	FF	F3	87	AD	00
0x0001	45	67	89	87	67	90	99	34
0x0002	80	00	23	80	DE	ED	90	54
0x0003	99	AA	11	80	EF	ED	12	67
0x0004	90	78	AE	10	FE	4E	44	23
0x0005	71	32	F5	10	FE	4E	55	23
0x0006	1A	44	F2	11	FE	64	66	F3
0x0007	1B	23	FF	EE	FF	53	88	7F
0x0008	1C	54	00	00	FF	AA	99	5E
0x0009	BB	34	00	00	34	55	98	4E
0x000A	B4	45	00	00	55	44	45	1E
0x0008	B5	78	00	00	FF	FF	56	12
0x000C	B6	67	56	00	00	CC	DE	E1
...
0x1FFF	FF	FF	FF	FF	FF	FF	FF	FF

Write Through + Write NO Allocate

Operaciones a realizar en un acceso a Cache

$0x000A =$
 $0000\ 0000\ 0000\ 1010 =$
 $000000000000\ 01\ 010$
 $TAG = 0x000$
 $\text{l\'inea} = 0x01$
 $\text{byte} = 0x2$
 Bloque MP = 0x0001

Escr 0x000D: **HIT**
 Lect 0x000A: **HIT**
 Escr 0x002E:
 Lect 0x037:

	V	DB	Etiqueta	DATOS MC							
0	1	0	0x000	00	03	05	FF	F3	87	AD	00
1	1	1	0x000	45	67	89	87	FF	90	99	34
2	0	0									
3	1	0	0x000	99	AA	11	80	EF	ED	12	67

Copy Back + Write Allocate

Operaciones a realizar en un acceso a Cache

$0x000D =$
 $0000\ 0000\ 0000\ 1100 =$
 $000000000000\ 01\ 100$
 $TAG = 0x000$
 $\text{l\'inea} = 0x01$
 $\text{byte} = 0x4$
 Bloque MP = 0x0001

Escr 0x000D: **HIT**
 Lect 0x000A:
 Escr 0x002E:
 Lect 0x037:

Bloque MP	DATOS MP							
0x0000	00	03	05	FF	F3	87	AD	00
0x0001	45	67	89	87	67	90	99	34
0x0002	80	00	23	80	DE	ED	90	54
0x0003	99	AA	11	80	EF	ED	12	67
0x0004	90	78	AE	10	FE	4E	44	23
0x0005	71	32	F5	10	FE	64	66	F3
0x0006	1A	44	F2	11	FE	56	12	12
0x0007	1B	23	FF	EE	FF	53	88	7F
0x0008	1C	54	00	00	FF	AA	99	5E
0x0009	BB	34	00	00	34	55	98	4E
0x000A	B4	45	00	00	55	44	45	1E
0x0008	B5	78	00	00	FF	FF	56	12
0x000C	B6	67	56	00	00	CC	DE	E1
...
0x1FFF	FF	FF	FF	FF	FF	FF	FF	FF

Copy Back + Write Allocate

Operaciones a realizar en un acceso a Cache

$0x002E =$
 $0000\ 0000\ 0010\ 1110 =$
 $000000000001\ 01\ 110$
 $TAG = 0x001$
 $\text{l\'inea} = 0x01$
 $\text{byte} = 0x6$
 Bloque MP = 0x0005

Escr 0x000D: **HIT**
 Lect 0x000A: **HIT**
 Escr 0x002E: **MISS**
 Lect 0x037:

Bloque MP	DATOS MP							
0x0000	00	03	05	FF	F3	87	AD	00
0x0001	45	67	89	87	67	90	99	34
0x0002	80	00	23	80	DE	ED	90	54
0x0003	99	AA	11	80	EF	ED	12	67
0x0004	90	78	AE	10	FE	4E	44	23
0x0005	71	32	F5	10	FE	4E	55	23
0x0006	1A	44	F2	11	FE	64	66	F3
0x0007	1B	23	FF	EE	FF	53	88	7F
0x0008	1C	54	00	00	FF	AA	99	5E
0x0009	BB	34	00	00	34	55	98	4E
0x000A	B4	45	00	00	55	44	45	1E
0x0008	B5	78	00	00	FF	FF	56	12
0x000C	B6	67	56	00	00	CC	DE	E1
...
0x1FFF	FF	FF	FF	FF	FF	FF	FF	FF

Copy Back + Write Allocate

Operaciones a realizar en un acceso a Cache

0x0037 =
0000 0000 0011 0111 =
000000000001 10 111
TAG = 0x001
línea = 0x02
byte = 0x7
Bloque MP = 0x0006

Escr 0x000D: HIT
Lect 0x00A: HIT
Escr 0x002E: MISS
Lect 0x0037: MISS

V	DB	Etiqueta	DATOS MC								
0	1	0	0x000	00	03	05	FF	F3	87	AD	00
1	1	1	0x001	71	32	F5	10	FE	4E	55	23
2	1	0	0x001	1A	44	F2	11	FE	64	66	F3
3	1	0	0x000	99	AA	11	80	EF	ED	12	67

Bloque MP	DATOS MP							
0x0000	00	03	05	FF	F3	87	AD	00
0x0001	45	67	89	87	FF	90	99	34
0x0002	80	00	23	80	DE	ED	90	54
0x0003	99	AA	11	80	EF	ED	12	67
0x0004	90	78	AE	10	FE	4E	44	23
0x0005	71	32	F5	10	FE	4E	55	23
0x0006	1A	44	F2	11	FE	64	66	F3
0x0007	1B	23	FF	EE	FF	53	88	7F
0x0008	1C	54	00	00	FF	AA	99	5E
0x0009	BB	34	00	00	34	55	98	4E
0x000A	B4	45	00	00	55	44	45	1E
0x000B	B5	78	00	00	FF	FF	56	12
0x000C	B6	67	56	00	00	CC	DE	E1
...
0x1FFF	FF	FF	FF	FF	FF	FF	FF	FF

Copy Back + Write Allocate

Políticas de Escritura

Premisa: las escrituras, finalmente, se han de hacer en Memoria Principal.

¿Cuándo se actualiza la Memoria Principal?

WRITE THROUGH (escritura a través o escritura inmediata)

- Se actualizan simultáneamente la MC y la MP.
- El coste del acceso es el tiempo de acceso a MP, pero se puede reducir el tiempo de escritura utilizando buffers.
- La MP siempre está actualizada.

COPY BACK (escritura diferida)

- En una escritura sólo se actualiza MC.
- Para cada línea se añade un bit de control (dirty bit) que indica si la línea ha sido modificada o no.
- Se actualiza la MP cuando la línea (modificada) ha de ser reemplazada.
- Las escrituras son rápidas (velocidad de MC).
- El tiempo de penalización en caso de fallo aumenta.
- Durante un tiempo existe una inconsistencia entre MP y MC.

Políticas de Escritura

Premisa: las escrituras, finalmente, se han de hacer en Memoria Principal.

¿Qué hacer en caso de fallo en escritura?

WRITE ALLOCATE (con migración/asignación en caso de fallo)

- Se trae la línea de MP a MC y después se realiza la escritura.

WRITE NO ALLOCATE (sin migración/asignación en caso de fallo)

- La línea NO se trae a MC. Esto obliga a realizar la escritura directamente en MP.

Normalmente se utiliza:

- COPY BACK + WRITE ALLOCATE
- WRITE THROUGH + WRITE NO ALLOCATE