

Todos los ISAs tienen:

Instrucciones aritmético/lógicas:

- $OP_d \leftarrow OP_1 (?) OP_2$

Instrucciones de Acceso a Memoria:

- $OP_d \leftarrow MEM[dir]$
- $MEM[dir] \leftarrow OP_f$

Instrucciones de ruptura de secuencia:

- `if (OP1 cond OP2) goto L`

Los operandos (OP_x) pueden ser registros del procesador, memoria, o constantes.

Los operadores son los que podemos encontrar en cualquier LAN:

- ✓ **aritméticos**: +, -, *, /, %
- ✓ **bit-wise**: ~, |, &, ^, >>, <<
- ✓ **lógicos**: !, ||, &&
- ✓ **relacionales**: !=, ==, >, <, >=, <=
- La dirección con la que accedemos a memoria puede ser algo muy simple (un registro), o requerir un cálculo más complejo (modos de direccionamiento)
- Las condiciones de salto han de permitir implementar las sentencias condicionales e iterativas de un LAN convencional.

Compilación

- Compilar es traducir un programa escrito en un Lenguaje X a un programa equivalente escrito en un lenguaje Y.
- Normalmente un compilador traduce un programa escrito en un Lenguaje de Alto Nivel a un programa equivalente escrito en un Lenguaje Máquina específico.
- Ejemplos de Lenguajes que siempre se compilan: C/C++, Fortran, Pascal, Cobol, ...
- Características
 - Sólo se ha de compilar 1 vez, se puede ejecutar muchas veces.
 - Si cambiamos de ISA (y/o ABI) se ha de recomilar.
 - Un compilador es una aplicación **MUY COMPLEJA** (100.000's de líneas de código).
 - El código generado por un compilador puede ser **MUY EFICIENTE**.



Operator	Description
&	bitwise AND
	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right
~	one's complement

&&	Operador and (y)
	Operador or (o)
!	Operador not (no)

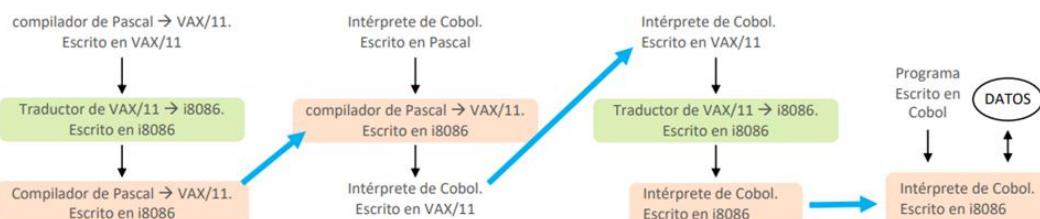
Problema

Tenemos:

- Una ordenador con una CPU i8086.
- Un compilador de Pascal → VAX/11. Escrito en VAX/11
- Un traductor de VAX/11 → i8086. Escrito en i8086
- Un interprete de Cobol. Escrito en Pascal.

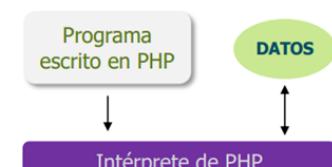
Examen EC
abril 1988

¿Qué pasos hay que seguir para ejecutar un programa escrito en Cobol.



Interpretación

- Un intérprete tiene como entrada un programa escrito en un Lenguaje de Alto Nivel y lo ejecuta directamente.
- Primero comprueba que no haya errores sintácticos, igual que un compilador, y luego ejecuta el código.
- Es muy común: Lenguajes de Alto Nivel: BASIC, PHP, Phyton, Perl, ...; el Shell de Linux; Máquina Virtual de Java; una CPU es un intérprete de Lenguaje Máquina.
- Características
 - Se ha de "traducir" en cada ejecución.
 - Un mismo código puede ejecutarse en múltiples plataformas sin cambios.
 - Un intérprete es una aplicación más simple que un compilador.
 - La ejecución es más **INEFICIENTE**.
 - Es más fácil/rápido depurar programas interpretados.



RISC vs CISC

Complex Instruction Set Computer (CISC)

- ❑ Juego de instrucciones grande y complejo.
- ❑ Instrucciones de longitud variable
- ❑ Modos de direccionamiento complejos
- ❑ Cualquier instrucción puede acceder a memoria.
- ❑ Cada instrucción se convierte en 1..n microoperaciones (inst. RISC?)
- ❑ Ejemplos: x86

Reduced Instruction Set Computer (RISC)

- ❑ Juego de instrucciones pequeño y sencillo.
- ❑ Instrucciones de longitud fija
- ❑ Formatos de instrucción y modos de direccionamiento sencillos
- ❑ Acceso a memoria con instrucciones específicas: load y store
- ❑ Ejecutadas directamente por hardware
- ❑ Ejemplos: MIPS, ARM, RISC-V

Palabra (Word)

- ❑ Dato que tiene el tamaño nativo de la arquitectura
- ❑ Normalmente, el tamaño de la palabra es igual al tamaño de registro
- ❑ MIPS: 32 bits (4 bytes)
- ❑ Ejemplos: **0x234AB6708, 0x00F30001**
- ❑ Si almacenamos **0x00F30001** en la dirección **0x234AB6708**, significa que estamos ocupando los bytes de memoria: **0x234AB6708, 0x234AB6709, 0x234AB670A y 0x234AB670B**

Endianness (ordenación de bytes)

0x00F30001

00	F3	00	01
----	----	----	----

byte 3 byte 2 byte 1 byte 0

- ❑ **Little-endian:** byte de menor peso en la dirección más baja

0x234AB6708	01
0x234AB6709	00
0x234AB670A	F3
0x234AB670B	00

- ❑ **Big-endian:** byte de mayor peso en la dirección más baja

0x234AB6708	00
0x234AB6709	F3
0x234AB670A	00
0x234AB670B	01

Registros

Número	Nombre	Descripción
\$0	\$zero	Valor 0, read only
\$1	\$at	Reservado
\$2, \$3	\$v0, \$v1	Retornos de Subrutinas
\$4, \$5, \$6, \$7	\$a0, \$a1, \$a2, \$a3	Argumentos de Subrutinas
\$8, \$9, \$10, \$11, \$12, \$13, \$14, \$15	\$t0, \$t1, \$t2, \$t3, \$t4, \$t5, \$t6, \$t7	Valores Temporales
\$16, \$17, \$18, \$19, \$20, \$21, \$22, \$23	\$s0, \$s1, \$s2, \$s3, \$s4, \$s5, \$s6, \$s7	Variables Locales
\$24, \$25	\$t8, \$t9	Valores Temporales
\$26, \$27	\$k0, \$k1	Reservado SO
\$28	\$gp	Global Pointer
\$29	\$sp	Stack Pointer
\$30	\$fp	Frame Pointer
\$31	\$ra	Return Address

Un ejemplo de traducción C → MIPS

```
.data
V: .word -1, -2, -3, -4, -5, -6, -7, -8, -9, -10 # vector V
.text
.globl main
main: li $t0, 0                                # $t0 = 0 (suma = 0)
      li $t1, 0                                # $t1 = 0 (i = 0)
      li $t2, 10                               # $t2 = 10
loop: slt $t3, $t1, $t2                         # $t3 = $t1 < $t2
      beq $t3, $zero, end
      la $t3, V                                # $t3 = @inicial de V
      sll $t4, $t1, 2                           # $t4 = 4*i (cada elemento ocupa 4 bytes)
      addu $t3, $t3, $t4                         # @V[i] = @inicial V + i*4
      lw $t3, 0($t3)                            # $t3 = V[i]
      addu $t0, $t0, $t3                         # suma = suma + V[i]
      addiu $t1, $t1, 1                          # i++
      b loop                                 # salto incondicional
end:  jr $ra
```

Código MIPS que suma los elementos de un vector de enteros

Variables Locales vs Variables Globales

Variables Globales

- Se declaran fuera de cualquier función y pueden ser accedidas desde cualquier sitio
- Se mantienen en memoria durante toda la ejecución del programa
- Se almacenan en posiciones fijas de memoria

Variables Locales

- Sólo se pueden acceder dentro del bloque donde se declaran
- Se crean al inicio de la ejecución del bloque y dejan de existir cuando finaliza
- Se reserva espacio de almacenamiento de forma dinámica (memoria o registro)

```
int V[10] = {-1, -2, ...};
void main() {
    int suma = 0;
    int i = 0;
    ...
}
```

V es una variable Global

suma e i son variables Locales

```
.data
V: .word -1, -2, ...
.text
...
li $t0, 0 # $t0 = 0 (suma = 0)
li $t1, 0 # $t1 = 0 (i = 0)
...

```

Variables Globales Ensamblador

Tipo C	MIPS	Tamaño
(unsigned) char	.byte	1 byte
(unsigned) short	.half	2 bytes
(unsigned) int / long	.word	4 bytes (1 word)
(unsigned) 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

Alineación de Memoria

Existen instrucciones específicas para leer/escribir

(load/store) de Memoria

- Datos de 4 bytes (word): lw y sw
- Datos de 2 bytes (half): lh, lhu y sh
- Datos de 1 byte (byte): lb, lbu y sb

En MIPS los accesos a Memoria han de estar alineados

- Las instrucciones lw y sw sólo pueden acceder a direcciones múltiplo de 4.
- Las instrucciones lh, lhu y sh sólo pueden acceder a direcciones múltiplo de 2.
- Las instrucciones lb, lbu y sb no tienen restricciones

```
.data
y: .word 0x13457AB2
.text
la $t0, y      #t0=@y
lw $t1, 0($t0)
lw $t1, 1($t0)
```

La primera instrucción lw funciona perfectamente. La segunda lw provoca un error de alineamiento que hará que el programa aborte.

Variables Globales Ensamblador

```
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 directivas .half, .word y .dword fuerzan la alineación.

a:	00	-
x:	-	-
b:	0D	-
c:	00	-
d:	FF	01
c:	0A	00
-	-	F0
-	A3	-
y:	B2	34
-	7A	80
-	45	03
-	13	12

@ alineada a 8

Declaración y Alineación de Variables Globales

- **Alineación Automática.** Por defecto las variables globales se ubican en direcciones múltiplo de su tamaño.
- Las directivas **.half**, **.word** y **.dword** fuerzan la alineación correcta.

Directivas .byte .half .word .dword

.byte n	Reserva 1 byte y lo inicializa a n
.half n	Reserva 1 halfword (2 bytes) alineado a una dirección múltiplo de 2 y lo inicializa a n.
.word n	Reserva 1 word (4 bytes) alineado a una dirección múltiplo de 4 y lo inicializa a n.
.dword n	Reserva 1 doubleword (8 bytes) alineado a una dirección múltiplo de 8 y lo inicializa a n.

Declaración y Alineación de Variables Globales

- Para alinear de forma explícita podemos utilizar la directiva **.align n**

Directivas .align .space

.align n	Ubica el próximo dato en una dirección múltiplo de 2^n
.space m	Reserva m bytes y los inicializa a 0

```
short x;
int v[100];
```

Declaración en C

```
x: .half 0      # alineado a 2 bytes
.align 2
v: .space 100*4 # alineado a 4 bytes
```

Declaración en MIPS

Representación de Números Enteros en ca2

- Representación de enteros en **complemento a 2** (ca2):

- X, valor entero que representamos (valor implícito)
- X_u , valor natural asociado (valor explícito)
- Representación
 $X = (x_{n-1}x_{n-2} \dots x_2x_1x_0)$, vector de bits, $x_i \in \{0, 1\}$
- Función de interpretación:
 $X = X_u - x_{n-1} \cdot 2^n$
- Función de representación:

$$X_u = \begin{cases} x & \text{si } x \geq 0 \\ x + 2^n & \text{si } x < 0 \end{cases}$$

Los valores enteros están "desordenados"

El bit x_{n-1} indica el signo

$x_3x_2x_1x_0$	X_u	X
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

Se utiliza el mismo +/- que con números naturales

Representación de Números Enteros en ca2

- El rango de representación de enteros, de n bits, en complemento a 2 (ca2) es:

$$X \in [-2^{n-1} \dots 2^{n-1}-1]$$

- Hay un valor más en negativo que en positivo

n	Mínimo (-2 ⁿ⁻¹)	Máximo (2 ⁿ⁻¹ -1)
8	-128	127
16	-32.768	32.767
32	-2.147.483.648	2.147.483.647
64	-9.223.372.036.854.775.808	9.223.372.036.854.775.807

Representación Números Enteros en Signo y Magnitud

X ₄ X ₃ X ₂ X ₁ X ₀	X _u	X	X ₄ X ₃ X ₂ X ₁ X ₀	X _u	X
00000	0	0	10000	16	0
00001	1	1	10001	17	-1
00010	2	2	10010	18	-2
00011	3	3	10011	19	-3
00100	4	4	10100	20	-4
00101	5	5	10101	21	-5
00110	6	6	10110	22	-6
00111	7	7	10111	23	-7
01000	8	8	11000	24	-8
01001	9	9	11001	25	-9
01010	10	10	11010	26	-10
01011	11	11	11011	27	-11
01100	12	12	11100	28	-12
01101	13	13	11101	29	-13
01110	14	14	11110	30	-14
01111	15	15	11111	31	-15



Los valores están "ordenados"

El bit x_{n-1} ES el signo

Hay 2 ceros

¿Se utiliza?

El +/- es muy ineficiente

Representación de Números Enteros en exceso a 2ⁿ⁻¹-1

- Representación de enteros en exceso a 2ⁿ⁻¹-1:

- Función de interpretación:

$$X = X_u - (2^{n-1} - 1)$$

- Función de representación:

$$X_u = X + (2^{n-1} - 1)$$

Los valores están "ordenados"

El número más pequeño (-15) corresponde con el 0...000, y el número más grande (16) corresponde con el 1...111.

Exceso a 15

X ₄ X ₃ X ₂ X ₁ X ₀	x _u	x	X ₄ X ₃ X ₂ X ₁ X ₀	x _u	x
00000	0	-15	10000	16	1
00001	1	-14	10001	17	2
00010	2	-13	10010	18	3
00011	3	-12	10011	19	4
00100	4	-11	10100	20	5
00101	5	-10	10101	21	6
00110	6	-9	10110	22	7
00111	7	-8	10111	23	8
01000	8	-7	11000	24	9
01001	9	-6	11001	25	10
01010	10	-5	11010	26	11
01011	11	-4	11011	27	12
01100	12	-3	11100	28	13
01101	13	-2	11101	29	14
01110	14	-1	11110	30	15
01111	15	0	11111	31	16

Representación de Números Enteros en exceso a 2ⁿ⁻¹-1

- El rango de representación de enteros en exceso a 2ⁿ⁻¹-1 es

$$X \in [-(2^{n-1}-1) \dots 2^{n-1}]$$

- Hay un valor más en positivo que en negativo

n	Mínimo (-(2 ⁿ⁻¹ -1))	Máximo (2 ⁿ⁻¹ +1)	
8	exceso a 127	-127	128
16	exceso a 2 ¹⁵ -1	-32.767	32.768
32	exceso a 2 ³² -1	-2.147.483.647	2.147.483.648
64	exceso a 2 ⁶⁴ -1	-9.223.372.036.854.775.807	9.223.372.036.854.775.808

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

Operación típica: Inicializar un Registro con un literal

li (pseudoinstrucción)		
li rdest, imm16	$rdest = \text{sigext}(imm16)$	addiu rdest, \$zero, imm16
li rdest, imm32	$rdest = imm32$	lui \$at, hi(imm32) ori rdest, \$at, lo(imm32)

- Mucho cuidado con las macros! MARS (esimulador de MIPS que usaremos en el Laboratorio) siempre trabaja con literales de 32 bits

```
li $t1, 0xFFFF # Mars lee 0x0000FFFF
```

Operación típica: Copia entre Registros

```
addiu $t1, $t2, 0 # $t1 = $t2
```

- Alternativa:

```
addu $t1, $zero, $t2 # $t1 = $t2
```

- Alternativa (pseudoinstrucción o macro):

```
move $t1, $t2 # equivale a addu $t1, $t2, $zero
```

- Las macros se expanden en 1 o más instrucciones MIPS
- Facilitan la lectura y Depuración

move (pseudoinstrucción)

move rdest, rsrc	$rdest = rsrc$	addu rdest, rsrc, \$zero
-------------------------	----------------	---------------------------------

Cómo gestionamos las constantes

- En C es muy habitual definir constantes:

```
#define N 10  
#define M -67
```

C

- También podemos usar constantes en MIPS:

```
.eqv N 10  
.eqv M -67  
.data  
a: .word 0  
.text  
.globl main  
main: li $t0, N # $t0 = N  
      addiu $t0, $t0, M # $t0 = N + M
```

MIPS

Y ¿Cómo accedemos a Memoria?

- Cuando accedemos a memoria podemos:
 - LEER de Memoria (load): $dat \leftarrow M[dir]$
 - ESCRIBIR en Memoria (store): $M[dir] \leftarrow dat$
- Lo primero que necesitamos es la dirección, usaremos la macro **la** (load address)

<pre> .data num: .word 0text ... la \$t0,num # t0 ← &num = 0x010010000 </pre>	MIPS
la (pseudoinstrucción)	
<pre> la rdest, etiq rdest = @etiq lui \$at, hi(@etiq) ori rdest, \$at, lo(@etiq) </pre>	

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

Operaciones típicas con Caracteres

- Conversión Mayúsculas/Minúsculas
 - $C_{\text{MIN}} \leftarrow C_{\text{MAY}} + 32;$
- Conversión Dígito ASCII / valor Numérico
 - $\text{CaracterDigito} \leftarrow \text{Valor} + 48;$

No es buena idea utilizar los valores de la codificación directamente.

char A, b, n; int x;

C

```

A = b + ('A'-'a'); // MAY ← MIN
b = A + ('a'-'A'); // MIN ← MAY
x = n - '0';       // VAL ← DIG
n = x + '0';       // DIG ← VAL

```

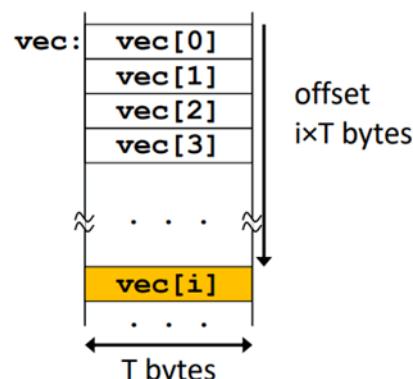
No es necesario conocer los códigos ASCII

Algo equivalente se puede hacer en MIPS

Vectores, acceso aleatorio

- Para acceder al elemento **vec[i]** hemos de calcular su dirección

$$\text{@vec}[i] = \text{@vec}[0] + i * \text{tam}$$
 siendo **tam**, el tamaño en bytes de los elementos del vector **vec**.



```

int vec[100];
...
x = vec[i]; // x,i en $t1,$t2

```

```

la $t0,vec      # @vec      MIPS
sll $t3,$t2,2   # i*4
addu $t0,$t0,$t3 # @vec+i*4
lw $t1,0($t0)   # x=vec[i]

```

Strings

- Un string es un vector de caracteres de tamaño variable.
- Se puede implementar de muchas formas:
 - En Java, con una tupla que tiene un entero con el tamaño y un vector de caracteres
 - **En C, un vector de caracteres con un centinela (el carácter '\0' = 0)**

```
char cadena[20] = "Una frase";
char cadena[20] = { 'U', 'n', 'a', ' ', 'f',
                   'r', 'a', 's', 'e', '\0' };
```

```
cadena: .ascii "Una frase" # sin centinela
          .space 11           # el centinela y 10 0's
cadena: .asciz "Una frase" # incluye el centinela
          .space 10
```

Declaraciones
Equivalentes

MIPS

C

Punteros

- Si el puntero es una **variable global**:

```
int *p1;
short *p2;
char *p3;
```

C

Un puntero es una dirección.
Siempre ocupa 32 bits (1 word),
independientemente de que sea un
puntero a un int, un char o un short.

- Se traduce a MIPS como:

```
.data
p1: .word 0
p2: .word 0
p3: .word 0
```

MIPS

- Si el puntero es una **variable local**, no hay que reservar espacio, se almacenará en un registro.

Punteros

- ¿Qué es un puntero?

Un puntero es una variable que contiene una dirección de memoria.

- Los punteros en MIPS ocupan 32 bits.
 - Si el puntero p contiene la dirección de la variable v, decimos que p apunta a v.
- En C, como cualquier variable, puede ser global o local

```
int *p1;      // variable global
void main() {
    int *p2;  // variable local
    ...
}
```

C

Punteros

- Uno de los errores más típicos en C es usar un puntero que no está inicializado.

Para inicializar un puntero utilizamos el operador de C: &. Este operador,
delante de una variable, nos devuelve la dirección de esa variable.

- La inicialización del puntero puede estar en la declaración:

```
int v[100];
char a = 'E';
char *q = &a;
int *p = &v[12];
```

C

MIPS

```
.data
v: .space 400
a: .byte 'E'
q: .word a      # q = &a
p: .word v+48  # p = &v[0]+12*4
```

Punteros

- ¿Cómo puedo acceder al dato apuntado por un puntero?

En C, el operador * delante de un puntero, devuelve el valor de la variable a la que apunta el puntero. Operando indirección (desreferència en català)

```
int *pgl;           C
void f() {
    int tmp; // en $t0
    tmp = *pgl;
    ...
    *pgl = 17;
}
```

```
.data
pgl: .word 0          MIPS
...
f: la $t1, pgl      # $t1 = &pgl
    lw $t1, 0($t1) # $t1 = pgl
    lw $t0, 0($t1) # $t0 = *pgl
    ...
    li $t2, 17      # $t2 = 17
    sw $t2, 0($t1) # *pgl = 17
```

Punteros

- Aritmética de punteros

Sumar un entero N, a un puntero p, que apunta a variables de tamaño T bytes, da como resultado un nuevo puntero que apunta a $(p + N \cdot T)$

```
char *p1;
short *p2;
int *p3;
long long *p4;
...
p1 += 3;
p2 = p2 + 3;
p3 += 3;
p4 = p4 + 3;           C
```

```
.data
p1: .word 0 # en $t1
p2: .word 0 # en $t2
p3: .word 0 # en $t3
p4: .word 0 # en $t4
...
addiu $t1, $t1, 3
addiu $t2, $t2, 3*2
addiu $t3, $t3, 3*4
addiu $t4, $t4, 3*8           MIPS
```

Relación entre punteros y vectores en C

En C, un vector es un puntero que apunta al primer elemento del vector.

```
int vec[100];           C
int *p;
main() {
    p = vec
    ...
    p[8] = 10;
    ...
    *(p+i) = 0;
    p[i] = 0;
}
```

Expresión correcta

Acceso al elemento 8 del vector vec

Expresiones Equivalentes.
Aritmética de punteros

Desplazamientos lógicos a la izquierda y a la derecha

- Instrucción **sll** (Shift Left Logical)

sll rd, rt, shamt # rd = rt << shamt

Añade 0's por la derecha

- Instrucción **srl** (Shift Right Logical)

srl rd, rt, shamt # rd = rt >> shamt

Añade 0's por la izquierda

shamt = shift amount, natural de 5 bits (0..31)

- ¿Qué pasa si trabajamos con números enteros en ca2?

0xFF (-1) << 1 = 0xFE (-2)
0x0F (15) << 2 = 0x3C (60)
0xF9 (-7) << 1 = 0xF2 (-14)

0xFF (-1) >> 1 = 0x7F (127)
0x0F (15) >> 2 = 0x03 (3)
0xF9 (-7) >> 1 = 0x7C (124)

sll multiplica por 2

srl NO divide por 2 los números negativos

Repertorio instrucciones de desplazamiento

Repertorio instrucciones lógicas bit a bit

sll, srl, sra, sllv, srlv, 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 ,

and, or, xor, nor, andi, ori, xori		
and rd, rs, rt	$rd = rs \& rt$	
or rd, rs, rt	$rd = rs rt$	
xor rd, rs, rt	$rd = rs \wedge rt$	
nor rd, rs, rt	$rd = \sim(rs rt)$	
andi rt, rs, imm16	$rt = rs \& \text{ZeroExt}(imm16)$	imm16 es un natural
ori rt, rs, imm16	$rt = rs \text{ZeroExt}(imm16)$	imm16 es un natural
xori rt, rs, imm16	$rt = rs \wedge \text{ZeroExt}(imm16)$	imm16 es un natural

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

Traducción de la negación booleana: !v

- Negación booleana:

```
c = !a; // 0 si a es CIERTO(≠0), 1 si a es FALSO(0)
```

```
sltiu $t2,$t0,1 #si $t0 es falso(0) (0<1) $t2=1 (CIERTO)
                  #si $t0 es cierto(≠0) (≠0>=1) $t2=0 (FALSO)
```

- Si el valor a negar está normalizado, podemos usar:

```
xori $t2, $t0, 1 # complementa el bit 0 (0→1, 1→0)
```

- Con la negación booleana podemos implementar:

```
c = !(a < b); // (a ≥ b)
c = !(a > b); // (a ≤ b)
```

Traducción de las operaciones booleanas && y ||

- Normalización de un booleano (en \$t0):

```
sltu $t0, $zero, $t0 # si $t0 es falso(0) (0<0) $t0=0  
                      # si $t0 es cierto(≠0) (0<≠0) $t0=1
```

- AND (&&) booleana

```
c = a && b; // a, b y c en $t0, $t1, $t2 respectivamente
```

```
sltu $t0, $zero, $t0 # Normaliza a  
sltu $t1, $zero, $t1 # Normaliza b  
and $t2, $t0, $t1    # c = a && b
```

- OR (||) booleana

```
c = a || b; // a, b y c en $t0, $t1, $t2 respectivamente
```

```
or $t2, $t0, $t1      # c = a | b  
sltu $t2, $zero, $t2 # Normaliza c
```

Sólo hay que normalizar el resultado

Traducción de las comparaciones ==, !=

- Supondremos que a, b, y c son enteros almacenados en \$t0, \$t1 y \$t2 respectivamente.

```
c = a == 0;      → sltiu $t2, $t0, 1 # a<1 (natural)
```

```
c = a != 0;     → sltu $t2, $zero, $t0 # 0<a (natural)
```

```
c = a == b;     → sub $t2, $t0, $t1 # t2=a-b  
c = (a-b)==0
```

```
c = a != b;     → sub $t2, $t0, $t1 # t2=a-b  
c = (a-b)!=0
```

Traducción de las comparaciones >, <, >=, <=

- Supondremos que a, b, y c son enteros almacenados en \$t0, \$t1 y \$t2 respectivamente.

`c = a < b;` → `slt $t2, $t0, $t1 # a<b`

`c = a > b;` → `slt $t2, $t1, $t0 # b<a, a>b`

`c = a <= b;` → `slt $t4, $t1, $t0 # b<a, a>b
sltiu $t2, $t4, 1 # !(a>b), a<=b`

`c = a >= b;` → `slt $t4, $t0, $t1 # a<b
sltiu $t2, $t4, 1 # !(a<b), a>=b`

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		
<code>beq rs, rt, label</code>	si ($rs==rt$) $PC = PC_{UP} + \text{SigExt}(\text{offset}16*4)$	
<code>bne rs, rt, label</code>	si ($rs!=rt$) $PC = PC_{UP} + \text{SigExt}(\text{offset}16*4)$	
<code>b label</code>	$PC = PC_{UP} + \text{SigExt}(\text{offset}16*4)$	<code>beq \$0, \$0, label</code>

Otros Saltos condicionales relativos al PC (macros)

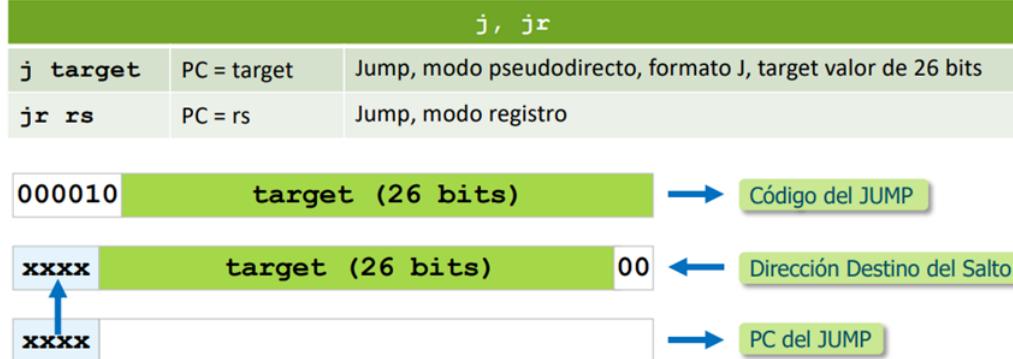
- Es muy habitual necesitar instrucciones de salto con condiciones del tipo $>$, $<$, \geq , \leq
- Usaremos macros

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

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

Saltos incondicionales “lejanos”

- Con los saltos bne y beq, sólo podemos saltar a una distancia limitada
 - [$-2^{15}..2^{15}-1$] instrucciones de distancia respecto a la instrucción de salto



- Hay 2 instrucciones “cercañas” **jal, jalr**, las explicaremos cuando hablemos de subrutinas.

Traducción Sentencia IF-THEN-ELSE

- Ejemplo de traducción, suponiendo que x, y, max son enteros y están en \$t0, \$t1 y \$t2 respectivamente.

C

```
if (x>y)
    max = x;
else
    max = y;
```

MIPS

```
ble $t0,$t1,else    # salta si x<=y
if:
    move $t2,$t0      # max ← x
    b endif
else:
    move $t2,$t1      # max ← y
endif:
```

Traducción Sentencia WHILE

- Ejemplo de traducción, división entera, supondremos que dd, dr y q son enteros y están en \$t1, \$t2 y \$t3 respectivamente.

C

```
q = 0;
while (dd >= dr) {
    dd = dd - dr;
    q++;
}
```

MIPS

```
move $t3, $zero      # q = 0
while:
    blt $t1,$t2,end  # si falso end
    subu $t1,$t1,$t2  # dd = dd-dr
    addiu $t3,$t3,1   # q++
    b while
end:
```

Traducción Sentencia FOR

- Ejemplo de traducción, suma de los elementos de un vector, supondremos que sum e i son enteros y están en \$t1 y \$t2 respectivamente, y la dirección de v[0] está en \$t0.

```
sum = 0;
for (i=0; i<N; i++)
    sum = sum + V[i];
```

C

```
move $t1,$zero      # sum = 0
move $t2,$zero      # i = 0
li $t3,N

for:
    bge $t2,$t3,end  # si i>=N end
    lw $t4,0($t0)      # t4 = v[i]
    addu $t1,$t1,$t4    # sum += v[i]
    addiu $t0,$t0,4     # t0=&v[i+1]
    addiu $t2,$t2,1     # i++
    b for
end:
```

MIPS

Traducción Sentencia SWITCH

```
int month; int days;
switch(month) {
    case 2: days = 28; break;
    case 4: days = 30; break;
    case 6: days = 30; break;
    case 9: days = 30; break;
    case 11: days = 30; break;
    default: days = 31;
}
```

Ejemplo sin fallthrough

```
int month; int days;
switch(month) {
    case 2: days = 28; break;
    case 4:
    case 6:
    case 9:
    case 11: days = 30; break;
    default: days = 31;
}
```

Ejemplo con fallthrough

Traducción Sentencia DO-WHILE

```
do {
    CUERPO-DO
} while (cond)
```

Modelo

```
int ContA(char v[]) {
    int i = 0, cont = 0;
    do {
        if (v[i] == 'a') cont++;
        i++;
    } while (v[i] != '.');
    return cont;
}
```

Ejemplo

do:
CUERPO-DO
evaluar condición
saltar si CIERTO a do
end:

PATRÓN de TRADUCCIÓN

SIEMPRE se ejecuta la 1^a iteración.

Subrutinas - Terminología

- Parámetros
 - Valor
 - Referencia
- Variables locales
- Invocación
- Retorno resultado
- Cuerpo subrutina

En C todos los parámetros son "por valor"

```
int DOT(int v1[], int v2[], int N) {
    int i, sum;
    sum = 0;
    for (i=0; i<N; i++)
        sum += v1[i] * v2[i];
    return sum;
}
```

```
void PDOT(int M[10][10], int *p) {
    int i;
    *p = 0;
    for (i=0; i<10; i++)
        *p += DOT(&M[0][0], &M[i][0], 10);
}
```

C

Subrutinas – Llamada y retorno

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

j, jr		
jal target	PC = 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 – Paso de Parámetros y Resultados

- Pasaremos Parámetros y Resultados de la forma más rápida posible: **REGISTROS**

Registros	Uso
\$a0, \$a1, \$a2, \$a3	Paso de Parámetros (argumentos)
\$v0, \$v1	Retorno resultado (usaremos sólo \$v0)
\$ra	Dirección retorno subrutina

- Los parámetros a una función se pasan en los registros \$a0–\$a3 **EN ORDEN**
 - Empezando por la izquierda el primero en \$a0, el segundo en \$a1, etc
 - Los parámetros de coma flotante se pasan en \$f12 (1er) y \$f14 (2º), máximo 2.
- El resultado se devuelve en el registro \$v0 (si es como flotante en \$f0)
- Si el parámetro/resultado tiene menos de 32 bits, se ha de extender adecuadamente.
- Por simplicidad, en EC supondremos que hay 4 parámetros como máximo, no usaremos longs, ni doubles, ni structs, etc

Subrutinas - Ejemplo

```
int suma2(int a, int b) {
    return a+b;
}

void main(){
    int x,y,z; // en t0,t1,t2
    ...
    z = suma2(x,y)
    ...
}
```

```
main:
    ...
    move $a0,$t0 # pasamos x
    move $a1,$t1 # pasamos y
    jal suma2
    move $t2,$v0 # z = resultado
    ...

summa2:
    addu $v0,$a0,$a1
    jr $ra
```

MIPS

- Como conocemos el protocolo (ABI), podemos llamar a **suma2** sin necesidad de saber cómo está hecha.

Subrutinas – Tipos de Parámetros

- Tradicionalmente existen 2 tipos de parámetros
 - Parámetros **POR VALOR**. La función recibe una copia del parámetro y se puede modificar sin que afecte al parámetro en la rutina que llama a la función.
 - Parámetros **POR REFERENCIA**. Las modificaciones que se realizan en la función afectan directamente al parámetro real. Para implementar este tipo de parámetros se pasa la dirección del parámetro, para que la función pueda acceder al dato.
- En C, todos los parámetros son **por VALOR**.
- Pero, si el parámetro es un **puntero**, equivale a tener un parámetro **POR REFERENCIA**.

Subrutinas – Variables Locales

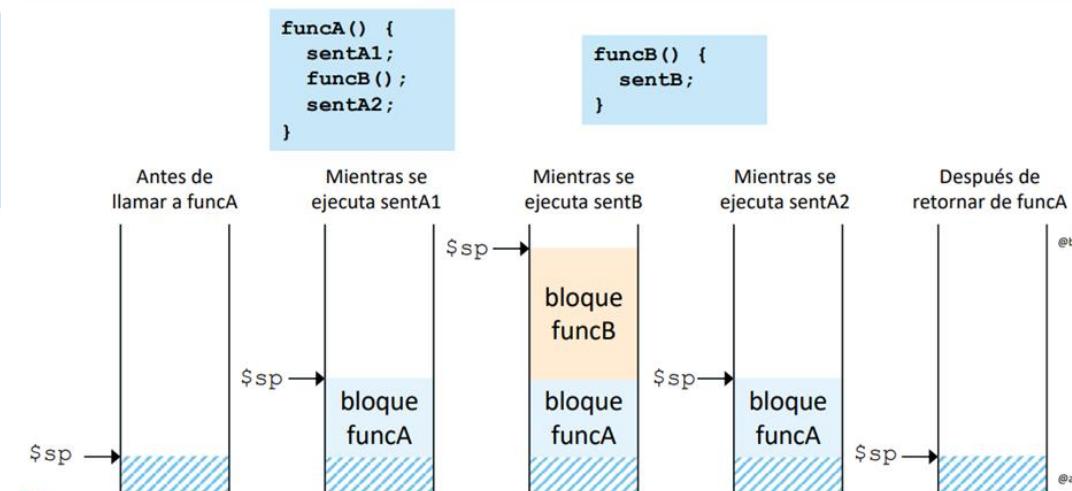
- Las **variables locales** se declaran dentro de una función y se crean y se destruyen con cada llamada.
- Las **variables locales** se han de inicializar de forma explícita, sino el valor es indeterminado.

¿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];  
    ...  
}
```

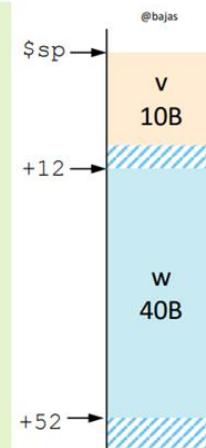
Subrutinas – La Pila y el Bloque de Activación



Subrutinas – Ejemplo

```
char func(int i){  
    char v[10];  
    int w[10], k;  
    ...  
    return v[w[i]+k];  
}
```

```
func:  
    #reserva espacio para v y w, 52B  
    addiu $sp,$sp,-52  
    ...  
    # @w[i] = sp+12 + i*4  
    sll $t4,$a0,2          # i*4  
    addu $t4,$t4,$sp        # sp+i*4  
    lw $t4,12($t4)          # w[i]  
    addu $t4,$t4,$t0        # w[i]+k  
  
    # @v[t4] = sp + t4  
    addu $t5,$sp,$t4        # sp + t4  
    lb $v0,0($t5)           # ret v[]  
  
    # liberar espacio pila  
    addiu $sp,$sp,52  
    jr $ra
```



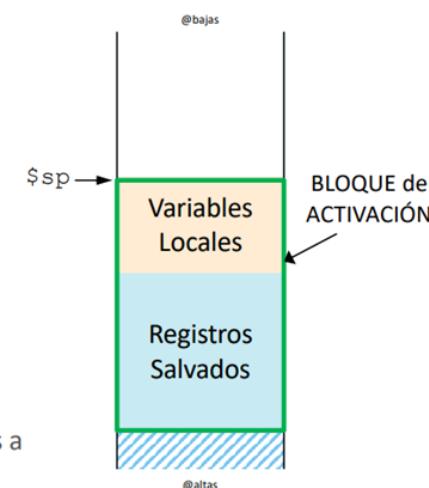
Subrutinas – Guardar y Restaurar en la PILA

Guardar y Restaurar

- Al escribir una subrutina, si utilizamos algún registro seguro, hemos de guardar una copia de él en la pila.
- En las subrutinas hemos de incluir:
 - Prólogo.** Reservamos espacio para las variables globales y salvamos los registros seguros.
 - Epílogo.** Restauramos los registros seguros y devolvemos el espacio de la pila.

PILA (Stack Frame)

- En el **BLOQUE de ACTIVACIÓN** tendremos:
 - Variables Locales**, ordenadas igual que en el programa y alineadas según su tipo.
 - Registros Salvados**, sin orden prefijado y alineados a 4 (todos los registros ocupan 4 bytes).



Subrutinas Multinivel – Ejemplo paso a paso

PASO 1. Registros Seguros

- Hay que poner en registros seguros aquellos datos que tienen un valor útil ANTES de una llamada y que se utilizan DESPUÉS de la llamada.
 - c** es un parámetro que se utiliza después de la llamada a **mcm**.
 - d** se calcula antes de llamar a **mcm** y se utiliza después.
 - Si no hacemos nada, dejamos **c** y **d** en registros temporales, nadie garantiza que **mcm** no modifique esos valores.
 - Pondremos **c** en **\$s0** y **d** en **\$s1**.

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

C

Registros Temporales	Registros Seguros
\$t0-\$t9	\$s0-\$s7
\$v0-\$v1	\$sp
\$a0-\$a3	\$ra
\$f0-\$f19	\$f20-\$f31

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
```

PRÓLOGO

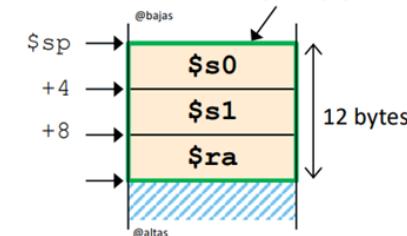
```
addu $s1,$a0,$a1
move $a0,$a2
move $a1,$s1
jal mcm
addu $t0,$s0,$s1
addu $v0,$v0,$t0

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

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



Subrutinas Repaso

Invocar a una subrutina

- Paso de parámetros
 - Los parámetros se pasan de izquierda a derecha en los registros **\$a0-\$a3**, por orden.
 - Tendremos como máximo 4 parámetros
- Llamada a la subrutina
 - jal SubName**
- Recoger/Usar resultado
 - El resultado siempre vendrá en **\$v0**

Antes de escribir la subrutina

- Evaluar Registros Seguros
- Dibujar Bloque de Activación

Escribir la subrutina

- Reservar espacio en la pila para el BA
 - Restar a **\$sp** el tamaño del BA
- Salvar registros
 - Incluyendo siempre **\$ra** [multinivel]
- Mover parámetros a registros seguros
- CUERPO de la SUBRUTINA
 - El resultado siempre se deja en **\$v0**
- Restaurar registros
- Eliminar espacio variables locales
 - Sumar a **\$sp** el tamaño del BA
- Retorno de subrutina
 - jr \$ra**

Subrutinas Repaso

Antes de escribir la subrutina

- Evaluar Registros Seguros
- Dibujar Bloque de Activación
 - En la parte alta estarán las variables locales que no puedan ir en registros.
 - En la parte baja guardaremos **\$ra** [multinivel] y los registros seguros que utilicemos.
 - Para cada elemento indicaremos la distancia desde el inicio del BA.
 - Indicaremos el tamaño total del BA.
 - ✓ Siempre ha de ser múltiplo de 4.

Variables Locales

- Siempre que podamos las variables locales se guardarán en registros.
 - \$t0-\$t9**
- Se almacena en la pila:
 - matrices vectores, tuplas, ...
 - Las variables locales con **&** .
 - variables para las que no hay registros suficientes.
- Los datos que van a la pila se almacenan en orden, alineados siguiendo los mismos criterios que las variables globales.
- El bloque de activación siempre ha de estar alineado a 4 bytes.

Instrucciones para Multiplicar Enteros en MIPS

- Para representar la multiplicación de 2 enteros de n y m bits necesitamos n+m bits.
- Caso de 2 números x e y de n bits en ca2
 - Rango x, y: $(-2^{n-1} \dots 2^{n-1}-1)$
 - Valor máximo $x*y$: $(-2^{n-1}) * (-2^{n-1}) = 2^{2n-2}$

n	Mínimo	Máximo	Min x*y	Max x*y
8	-128	127	-16.256	16.384
16	-32.768	32.767	-1.073.709.056	1.073.741.824

Necesitamos 16 bits
Necesitamos 32 bits

Declaración de Matrices

- En C, las dimensiones de las matrices han de ser constantes o literales

```
int Mat[nFil][nCol];
int MM[2][3]={{-1,2,3},{0,-4,7}};
```

- Si son variables globales, en MIPS quedaría así:

```
.data
.align 2
Mat:.space nFil*nCol*4
MM: .word -1, 2, 3, 0, -4, 7
```

```
MM: .word -1, 2, 3
     .word 0, -4, 7
```

¿“más fácil de ver”?

Instrucciones para Multiplicar Enteros en MIPS

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



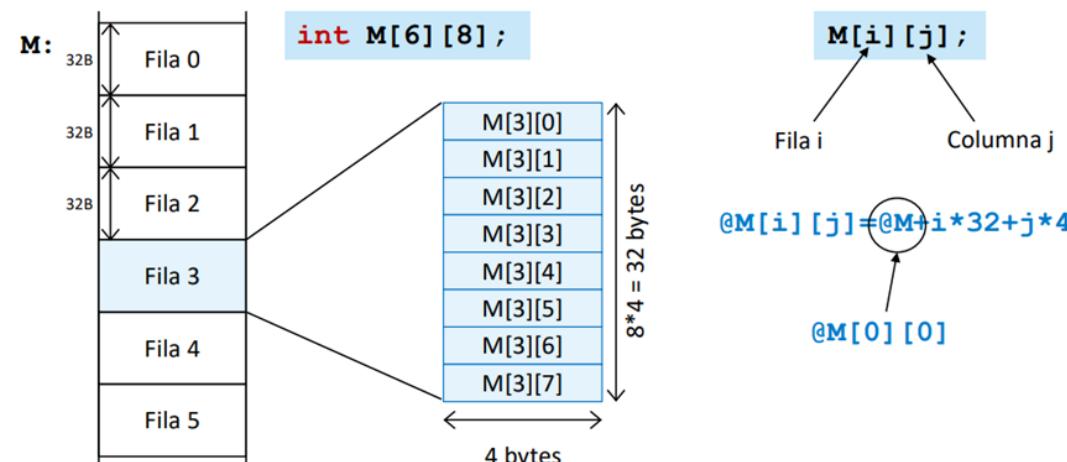
- El resultado se guarda en los registros \$hi y \$lo

- 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
```

Acceso aleatorio a un elemento

- En C, las matrices se almacenan por FILAS en posiciones consecutivas de memoria



Acceso aleatorio a un elemento

- En general, en C, si tenemos una matriz M de elementos de tipo XXX

```
XXX M[nFil][nCol];
```

- Para acceder al elemento de la fila i, columna j

```
M[i][j]
```

- Hay que calcular su dirección de la siguiente forma:

```
@M[i][j] = @inicioM + (i*nCol + j)*tam
```

siendo tam, el tamaño en bytes de los elementos de tipo XXX

Optimización: Acceso Secuencial

```
void clear(int V[], int N) {  
    for (int i=0; i<N; i++)  
        V[i] = 0;  
}
```

- Estamos recorriendo TODOS los elementos del vector.
- No es necesario calcular la dirección del V[i] en cada iteración

```
@V[i] = @inicioV + i*4
```

- Si tenemos la dirección de V[i] es muy fácil calcular la dirección de V[i+1]

```
@V[i+1] = @V[i] + stride
```

- El stride depende de tipo de los elementos del vector, si son int el stride es 4.

Ejemplo de traducción de un Acceso Aleatorio

- El código se simplifica mucho si alguno de los índices es constante
- Supongamos que Mat es global y que i, j, k están en \$t0, \$t1, \$t2

```
k = Mat[i][5];
```

```
# @Mat[i][5] = @inicioMat + (i*nCol + 5)*4  
# @Mat[i][5] = @inicioMat + i*nCol*4 + 5*4
```

```
la $t3, Mat          # t3 ← @Mat[0][0]  
li $t4, nCol*4  
mult $t0,$t4          # i*nCol*4  
mflo $t4              # t4 ← i*nCol*4  
addu $t4,$t4,$t3      # t4 ← @Mat[i][0]  
lw $t2,20($t4)        # t2 ← Mat[i][5]
```

Solución equivalente

Optimizando

```
void clear(int V[], int N) {  
    for (int i=0; i<N; i++)  
        V[i] = 0;  
}
```

```
clear: move $t0,$zero  
for: bge $t0,$a1,end  
     sll $t1,$t0,2  
     addu $t2,$a0,$t1  
     sw $zero,0($t2)  
     addiu $t0,$t0,1  
     b for  
end: jr $ra
```

6 inst/iter
2 saltos/iter
6×N+3 inst

```
CL1: move $t0,$zero  
      move $t1,$a0  
for: bge $t0,$a1,end  
     sw $zero,0($t1)  
     addiu $t0,$t0,1  
     addiu $t1,$t1,4  
     b for  
end: jr $ra
```

5 inst/iter
2 saltos/iter
5×N+4 inst

```
CL2: move $t1,$a0  
      sll $t2,$a1,2  
      addu $t3,$a0,$t2  
for: bgeu $t1,$t3,end  
     sw $zero,0($t1)  
     addiu $t1,$t1,4  
     b for  
end: jr $ra
```

4 inst/iter
2 saltos/iter
4×N+5 inst

```
CL3: move $t1,$a0  
      sll $t2,$a1,2  
      addu $t3,$a0,$t2  
      bgeu $t1,$t3,end  
do: sw $zero,0($t1)  
     addiu $t1,$t1,4  
     bltu $t1,$t3,do  
end: jr $ra
```

3 inst/iter
1 salto/iter
3×N+5 inst

Ejemplo de Revisión. Acceso Aleatorio

- Traducir a MIPS la función sumC (recorre la columna col)

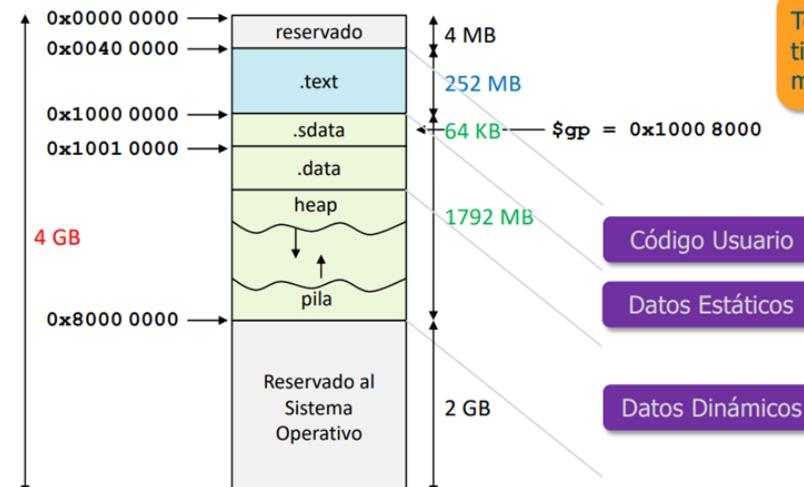
```
short sumC(short M[][NC],
           int col, int NF) {
    int i;
    short suma = 0;
    for (i=0; i<NF; i++)
        suma = suma + M[i][col];
    return suma;
}
```

$$@M[i][col] = @M + (i*NC+col)*2$$

Instrucciones Ejecutadas: $8 \times NF + 7$

```
sumC: move $v0,$zero      # suma←0
      move $t0,$zero      # i←0
      li $t1,NC*2          # NC*2
      sll $t2,$a1,1         # col*2
      addu $t2,$a0,$t2      # @M+col*2
      for: bge $t0,$a2,end  # si i>=NF end
            mult $t0,$t1      # i*NC*2
            mflo $t3
            addu $t3,$t2,$t3    # @M[i][col]
            lh $t4,0($t3)       # t4 ← M[i][col]
            addu $v0,$v0,$t4      # suma+M[i][col]
            addiu $t0,$t0,1       # i++
            b for
      end: jr $ra
```

Estructura de la Memoria en MIPS



Ejemplo de Revisión 2

```
XX: li $t7,M
    li $t6,N
    move $t0,$zero      # i←0
    fori: bge $t0,$t7,endi  # si i>=M goto endi
    move $t1,$zero      # j←0
    forj: bge $t1,$t6,endj  # si j>=N goto endj
    mult $t0,$t6          # i*N
    mflo $t5
    addu $t5,$t5,$t1      # i*N+j
    sll $t5,$t5,2          # (i*N+j)*4
    addu $t2,$a2,$t5      # t2←@B[i][j]
    lw $t2,0($t2)          # t2←B[i][j]
    addu $t3,$a1,$t5      # t3←@A[i][j]
    lw $t3,0($t3)          # t3←A[i][j]
    addu $t4,$t3,$t2      # t4←A[i][j]+B[i][j]
    addu $t2,$a0,$t5      # t2←@A[i][j]
    sw $t4,0($t2)          # A[i][j]←t4
    addiu $t1,$t1,1         # j++
    b forj
  endj: addiu $t0,$t0,1      # i++
    b fori
  endi: jr $ra
```

Instrucciones Ejecutadas $\approx (14 \times N + 4) \times M$

```
void XX(int C[M][N], int A[M][N], int B[M][N]) {
    int i,j;
    for (i=0; i<M; i++)
        for (j=0; j<N; j++)
            C[i][j] = A[i][j] + B[i][j];
}
```

```
XX: li $t7,M*N
    move $t4,$a0          # t4←@C[0][0]
    move $t5,$a1          # t5←@A[0][0]
    move $t6,$a2          # t6←@B[0][0]
    move $t0,$zero          # i←0
    for: bge $t0,$t7,end   # si i>=M*N goto end
    lw $t2,0($t6)          # t2 ← B[i][j]
    lw $t3,0($t5)          # t3 ← A[i][j]
    addu $t1,$t3,$t2      # t1←A[i][j]+B[i][j]
    sw $t1,0($t4)          # C[i][j] ← t1
    addiu $t0,$t0,1          # i++
    addiu $t6,$t6,4
    addiu $t5,$t5,4
    addiu $t4,$t4,4
    b for
  end: jr $ra
```

Instrucciones Ejecutadas $\approx 10 \times N \times M$

Almacenamiento Estático

- Acceso a variables globales de la sección **.data**

- Necesitamos cargar la dirección en un registro, usamos "**3 instrucciones**"

```
la $t0, etiquetaVarGlobal
lw $t1, 0($t0)
```

la es una macro que se traduce a 2 instrucciones

- La sección **.sdata** (small data)

- Tamaño: 2^{16} bytes (64 KB) [Muy pequeño]
- Se utiliza para guardar variables globales en C
- El registro **\$gp** (global pointer) apunta a una posición fija al medio de **.sdata**
- Podemos acceder a las variables globales con **1 instrucción**, a offsets $< 2^{15}$ del **\$gp**

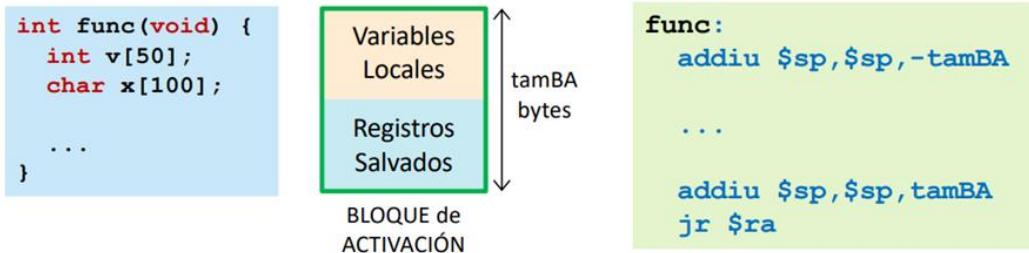
```
lw $t1, offsetVarGlobal($gp)
```

Almacenamiento Dinámico

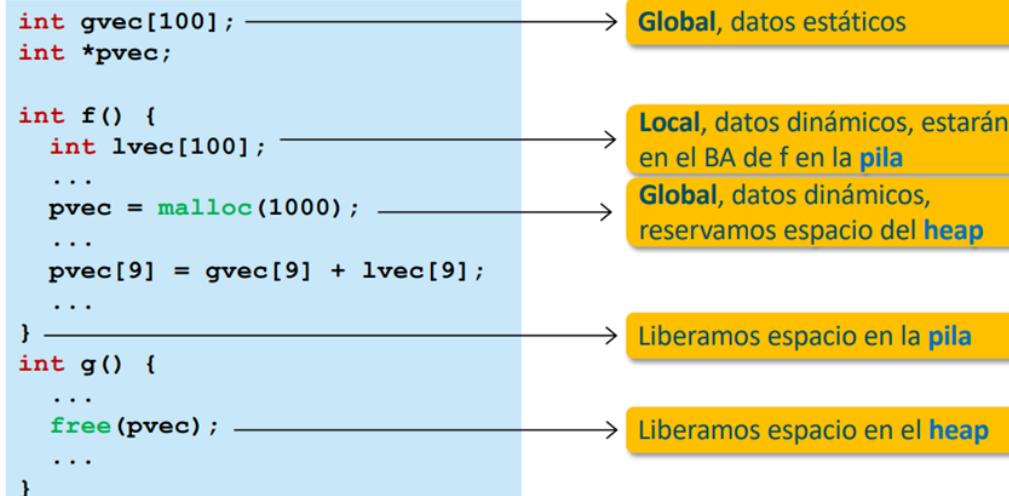
Almacenamiento dinámico: La variable ocupa un espacio de memoria temporal

□ La sección Pila (stack)

- Guarda el Bloque de Activación de las subrutinas: variables locales y registros seguros.
- Crece hacia direcciones bajas
- Crecer/decrecer dinámicamente. Reservar/Liberar espacio al inicio/final de las subrutinas.



¿Dónde se almacenan las diferentes variables?



Almacenamiento Dinámico

Almacenamiento dinámico: La variable ocupa un espacio de memoria temporal

□ La sección .heap

- Guarda las variables que se crean y destruyen explícitamente.
- Crecer hacia direcciones altas
- Reservar/Liberar espacio con llamadas al SO: `malloc()` y `free()`

```
int main(void) {
    int *ptr, numB;
    ...
    numB = 1000*sizeof(int);
    ptr = malloc(numB);
    ...
    free(ptr);
}
```

```
ptr = malloc(numB);
if (ptr == NULL)
    error&exit
```

```
free(ptr);
ptr = NULL;
```

¡CONTROL de ERRORES!

Compilación, Ensamblado, Enlazado y Carga

Pasos a seguir para ejecutar un programa:

□ Compilación

- Traduce un código fuente escrito en un Lenguaje de Alto Nivel a Ensamblador.

□ Ensamblado

- Traduce de un código escrito en Ensamblador a Lenguaje Máquina (**fichero objeto**).

□ Enlazado

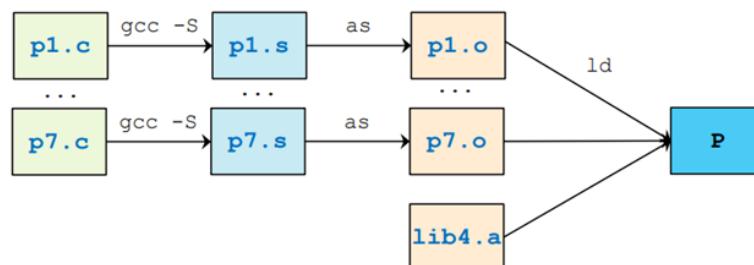
- Normalmente en una aplicación tenemos múltiples ficheros objeto. El **enlazador** (**linker**) se encarga de generar un único **fichero ejecutable**. A veces al enlazador se le llama **montador**.

□ Carga

- Para **ejecutar** un programa hay que copiar en memoria el **fichero ejecutable**, de eso se encarga el **cargador** (**loader**) que es una parte del **Sistema Operativo**.

Compilación Separada

- Estructurar el código en diversos módulos:
 - Facilita la gestión de proyectos complejos y fomenta la reutilización de código (**libraries**)
 - Sólo requiere compilación parcial cuando se modifica un módulo.
- Los módulos (ficheros) se pueden compilar y ensamblar por separado.
- Los diferentes módulos se enlazan para generar el fichero ejecutable.



¿Qué puede haber en un file.s?



Carga en Memoria

- Comando de invocación del programa

```
> filtrador fileIN.bmp fileOUT.bmp
```

- **filtrador**, es el fichero ejecutable.
- **fileIN.bmp** **fileOUT.bmp**, son los dos parámetros de entrada al programa.
- ¿Qué le llega al ejecutable?, 2 variables:
 - **int argc**, en este caso valdría 3.

```
argc: .word 3
```

- **char *argv[]**, es un vector de punteros a char, en este caso un vector con 3 posiciones y la siguiente información:

```
x1: .asciiz "filtrador"  
x2: .asciiz "fileIN.bmp"  
x3: .asciiz "fileOUT.bmp"  
argv: .word x1, x2, x3
```

- ¿Qué hace **startup()**?
 1. Pasa los parámetros al **main()** (en los registros **\$a0** y **\$a1**) [argc y argv].
 2. Llama a la rutina **main()**
 3. Cuando **main()** retorna, **startup()** invoca la rutina **exit()** del SO para liberar los recursos asignados al programa.

```
void main(int argc, char *argv[] {  
    ...  
}
```

C

Depende de nuestro objetivo, tenemos 2 métricas diferentes

❑ TIEMPO de EJECUCIÓN (Execution Time)

- T_{exe} . Tiempo transcurrido desde el inicio de una tarea hasta su finalización
- Rendimiento = $1/T_{exe}$

❑ PRODUCTIVIDAD (Throughput)

- Tareas completadas por unidad de tiempo

Tiempo de Ejecución

- ❑ El tiempo de ejecución normalmente incluye:

$$T_{EXE} = T_{CPU} + T_{E/S} + T_{resto}$$

- T_{CPU} . Tiempo de CPU (CPU time). El tiempo que consumimos ejecutando el programa.
- $T_{E/S}$. Tiempo de E/S. El tiempo que pasamos esperando el disco o la red (E/S en general)
- T_{resto} . Resto de tiempo que consume el SO, esperando a conseguir CPU, ...

Rendimiento y Speedup

- ❑ El rendimiento se evalúa utilizando el Tiempo de Ejecución:

$$\text{Rendimiento} = 1/T_{EXE}$$

- ❑ Speedup (Aceleración, Ganancia de Rendimiento):

$$\text{Speedup} = \frac{\text{Tiempo ORIGINAL}}{\text{Tiempo MEJORADO}} = \frac{\text{Rendimiento MEJORADO}}{\text{Rendimiento ORIGINAL}}$$

Ejemplo:

- $T_{original} = 10\text{s}$
- $T_{mejorado} = 5\text{s}$
- Speedup = $10/5 = 2$

- ❑ Las mejoras se pueden producir a muchos niveles:

- En el programa (algoritmos, estructuras de datos, lenguajes de programación, ...)
- En el compilador (optimización de bucles, de llamadas, ...)
- En la microarquitectura (ALUs, pipeline, cache, especulación, predictores, ...)
- En la tecnología (frecuencia de reloj, tamaño transistores, ...)

Factores que influyen en T_{EXE}

- ❑ Podemos expresar el Tiempo de ejecución de la siguiente forma:

$$T_{EXE} = n_{ciclos} \cdot T_c = n_{ciclos} / f_{clock}$$

siendo:

- n_{ciclos} = Número total de ciclos de reloj que tarda la ejecución del programa
- T_c = Tiempo de ciclo (o Periodo de reloj)
- f_{clock} = Frecuencia de reloj = $1 / T_c$

¿En el mundo real
cómo podemos
saber el número de
ciclos que tarda un
programa?

- ❑ ¿Cómo podemos mejorar el Tiempo de ejecución?

- Reduciendo el número de ciclos
- Aumentando la frecuencia de reloj (reduciendo el tiempo de ciclo)

Reducción del número de ciclos

- El número de ciclos se puede expresar como:

$$n_{\text{ciclos}} = N \cdot CPI$$



$$T_{\text{EXE}} = N \cdot CPI \cdot T_c$$

, siendo:

- N, el número total de instrucciones ejecutadas
- CPI, el promedio de ciclos por instrucción

- Cada tipo de instrucción i , tiene un CPI_i diferente

$$n_{\text{ciclos}} = \sum CPI_i \cdot n_i$$

, siendo:

- CPI_i , ciclos que tarda una instrucción de tipo i
- n_i , número total de instrucciones del tipo i ejecutadas

¿Cómo reducimos n_{ciclos} ?

- Reduciendo N
 - Mejorando el compilador
- Reduciendo el CPI
 - Mejorando la microarquitectura, usando instrucciones más rápidas (mult por sll)

Ley de Amdahl

- ¿Cuál es el MÁXIMO SPEEDUP TEÓRICO que podemos conseguir?

$$S_G = \frac{T_0}{T_M} = \frac{1}{(1 - P) + \frac{P}{S}}$$

$$S_{\max} = \lim_{S \rightarrow \infty} \frac{1}{(1 - P) + \frac{P}{S}} = \frac{1}{1 - P}$$

Ley de Amdahl. El máximo Speedup (S_{\max}) que podemos conseguir mejorando una parte de una tarea está limitado por el tiempo de ejecución que representa esta parte en el total (P).

P	20%	50%	75%	90%	99%
S_{\max}	1,25	2	4	10	100

Ley de Amdahl

- Supongamos que tenemos

- una tarea que se ejecuta en un Tiempo T_0 ,
- que una fracción (P) de esa tarea puede ser mejorada, en consecuencia el resto ($1-P$) quedará igual,
- y que el speedup de la fracción mejorada es S .

Expresión formal del sentido común.

- ¿Cuál es el speedup de la tarea completa?

$$T_0 = (1 - P) \cdot T_0 + P \cdot T_0$$

$$T_M = (1 - P) \cdot T_0 + P \cdot T_0 / S$$

$$S_G = \frac{T_0}{T_M} = \frac{T_0}{(1 - P) \cdot T_0 + \frac{P \cdot T_0}{S}} = \frac{1}{(1 - P) + \frac{P}{S}}$$

Disipación de Potencia

- En una CPU tenemos dos tipos de Potencia Disipada

- Potencia Dinámica (P_d)**. Causada por la conmutación de los transistores.
- Potencia Estática (P_s)**. También conocida como "Leakage Current" (corrientes de fuga).

$$P = P_d + P_s$$

- Potencia y Energía

- La Energía se mide en unidades de trabajo: **Julios**
- Potencia es la energía consumida por unidad de tiempo: **Watios (Julios/seg)**
- Si la potencia es constante:

$$\text{Energía} = \text{Potencia} \times \text{tiempo}$$

Potencia Dinámica vs Potencia Estática

❑ Potencia Dinámica (P_d):

$$P_d = \alpha \cdot C \cdot V^2 \cdot f_{clock}$$

- Normalmente ($\alpha \cdot C$) aparecen juntos en un solo factor C.

❑ Potencia Estática (P_e):

$$P_e = I_{leak} \cdot V$$

siendo,

- I_{leak} : Corriente parásita que circula por los transistores en circuito abierto (A, Amperios)

❑ Potencia Total (W):

$$P = P_d + P_e$$

❑ Energía consumida en un tiempo t, (J, julios):

$$E = P \cdot t$$

Algunas Técnicas para Reducir el Consumo

❑ Clock Gating.

- Inhabilitan partes del chip para reducir el α de la potencia dinámica

❑ Dynamic Voltage and Frequency Scaling (DVFS).

- Bajan selectivamente el voltaje y la frecuencia de una parte del circuito que puede ir más lenta.

❑ Power Gating.

- Permite reducir la potencia dinámica y estática, apagando completamente (cortando la corriente) de partes del chip cuando no hay que usarlos.

- c) Si el procesador dissipa 3 watts (Joules per segon) sigui quina sigui la instrucció executada, quin ha estat el consum total del processador en Joules de cada versió del programa?

$$\text{Consumo} = \text{Energía} = P \cdot t$$

$$E(AA) = 3 \cdot 5,5 \text{ ms} = 16,5 \cdot 10^{-3} \text{ J}$$

$$E(AS) = 3 \cdot 5 \text{ ms} = 15 \cdot 10^{-3} \text{ J}$$

$$\begin{aligned} N(AA) &= 5 \cdot 10^6 \text{ instrucciones} \\ N(AS) &= 7,5 \cdot 10^6 \text{ instrucciones} \\ CPI(AA) &= 2,2 \text{ c/i} \\ CPI(AS) &= 1,33 \text{ c/i} \\ T(AA) &= 5,5 \text{ ms} \\ T(AS) &= 5 \text{ ms} \end{aligned}$$

Potencia Dinámica

- ### ❑ Consumo de energía producido por los ciclos de carga/descarga de los transistores (comutación)

$$P_d = \alpha \cdot C \cdot V^2 \cdot f_{clock}$$

siendo,

- P_d : Potencia Dinámica (Watios, W)
- α : Fracción de transistores que comutan por ciclo
- C: Capacitancia (Faradios, F)
- V: Voltaje (Voltios, V)
- f_{clock} : Frecuencia de reloj (ciclos/s)

Problema Rendimiento y Consumo

Tenim un codi en MIPS que treballa sobre matrius. Hem fet dues versions del codi, una usant accés aleatori (AA) i una altra usant accés seqüencial (AS). El total d'instruccions utilitzades a cada versió les podeu trobar a la següent taula:

	Load / Store	Mult	Salts que salten	Salts que no salten	Altres instruccions
AA	500.000	250.000	250.000	250.000	3.750.000
AS	500.000	0	500.000	500.000	6.000.000

Suposem que a la nostra arquitectura, cada instrucció de memòria costa 5 cicles, cada multiplicació 16 cicles, els salts 1 o 2 cicles depenen de si salta (2) o no salta (1) i la resta d'instruccions 1 cicle.

- a) Quin seria el CPI de cada versió del programa? Quin seria el speed-up de la versió d'accés seqüencial respecte a la d'accés aleatori?

$$N(AA) = 500.000 + 250.000 + 250.000 + 250.000 + 3.750.000 = 5 \cdot 10^6 \text{ instrucciones}$$

$$\text{Ciclos (AA)} = 500.000 \cdot 5 + 250.000 \cdot 16 + 250.000 \cdot 2 + 250.000 \cdot 1 + 3.750.000 \cdot 1 = 11 \cdot 10^6 \text{ ciclos}$$

$$N(AS) = 500.000 + 0 + 500.000 + 500.000 + 6.000.000 = 7,5 \cdot 10^6 \text{ instrucciones}$$

$$\text{Ciclos (AS)} = 500.000 \cdot 5 + 0 + 500.000 \cdot 2 + 500.000 \cdot 1 + 6.000.000 \cdot 1 = 10 \cdot 10^6 \text{ ciclos}$$

$$\text{CPI(AA)} = 11 \cdot 10^6 / 5 \cdot 10^6 = 2,2 \text{ c/i}$$

$$\text{CPI(AS)} = 10 \cdot 10^6 / 7,5 \cdot 10^6 = 1,33 \text{ c/i}$$

- b) Quant de temps, en segons, triga en executar-se cada programa si el rellotge va a una freqüència de 2GHz?

$$T_{exe} = N \cdot CPI \cdot T_c = N \cdot CPI / Freq$$

$$N(AA) = 5 \cdot 10^6 \text{ instrucciones}$$

$$N(AS) = 7,5 \cdot 10^6 \text{ instrucciones}$$

$$T(AA) = 5 \cdot 10^6 \cdot 2,2 / 2 \cdot 10^9 = 5,5 \cdot 10^{-3} \text{ s} = 5,5 \text{ ms}$$

$$T(AS) = 7,5 \cdot 10^6 \cdot 1,33 / 2 \cdot 10^9 = 1 \cdot 10^{-3} \text{ s} = 1 \text{ ms}$$

$$\text{CPI(AA)} = 2,2 \text{ c/i}$$

$$\text{CPI(AS)} = 1,33 \text{ c/i}$$

Problema Rendimiento y Consumo

- d) Ens diuen que es pot dissenyar una arquitectura alternativa on podriem reduir el temps de cicle de les operacions de multiplicació. Quin és el màxim nombre de cicles de multiplicació admissible per a la nova arquitectura per tal que el codi AA sigui al menys igual de ràpid que el codi AS?

Necesitamos que $N(AA) \cdot CPI(AA) = N(AS) \cdot CPI(AS)$

$$N(AS) \cdot CPI(AS) = 10 \cdot 10^6 \text{ ciclos}$$

$$\begin{aligned} 10 \cdot 10^6 &= (0,5 \cdot 5 + 0,25 \cdot CPI_M + 0,25 \cdot 2 + 0,25 \cdot 1 + 3,75 \cdot 1) \cdot 10^6 \\ 10 &= 2,5 + 0,5 + 4 + 0,25 \cdot CPI_M \Rightarrow CPI_M = 3/0,25 = 12 \text{ c/i} \end{aligned}$$

$$\begin{aligned} N(AA) &= 5 \cdot 10^6 \text{ instrucciones} \\ N(AS) &= 7,5 \cdot 10^6 \text{ instrucciones} \\ CPI(AA) &= 2,2 \text{ c/i} \\ CPI(AS) &= 1,33 \text{ c/i} \\ T(AA) &= 5,5 \text{ ms} \\ T(AS) &= 5 \text{ ms} \end{aligned}$$

Un processador ha estat dissenyat per poder funcionar correctament sols a les següents combinacions de freqüències i voltatges d'alimentació (les freqüències estan expressades en forma de fracció per facilitar la simplificació dels càlculs, sense la calculadora):

combinació	voltatge (V)	freqüència (GHz)
1	1,0	6/3
2	1,1	7/3
3	1,2	8/3
4	1,3	9/3

Suposem que la potència estàtica consumida és zero. Sabent que la potència dinàmica consumida amb la combinació número 1 és de $P_1 = 60\text{W}$, es demana:

- a) Quines són les potències dinàmiques consumides en les combinacions 2, 3 i 4, en watts?

Sabemos que la Potencia dinámica es $P = \alpha \cdot C \cdot V^2 \cdot F$

Desconocemos $\alpha \cdot C$, pero sabemos que $P_1 = 60 \text{ W}$,

$$\text{En consecuencia, } \alpha \cdot C = P / (V^2 \cdot F) = 60 / (1^2 \cdot 2 \cdot 10^9) = 30 \cdot 10^{-9} \text{ F}$$

Usando este valor tenemos que:

- $P_2 = 30 \cdot 10^{-9} \cdot 1,1^2 \cdot (7/3) \cdot 10^9 = 84,7 \text{ W}$
- $P_3 = 30 \cdot 10^{-9} \cdot 1,2^2 \cdot (8/3) \cdot 10^9 = 115,2 \text{ W}$
- $P_4 = 30 \cdot 10^{-9} \cdot 1,3^2 \cdot (9/3) \cdot 10^9 = 152,1 \text{ W}$

Quina és la combinació amb màxima freqüència i voltatge a la qual podrà funcionar la CPU sense sobrepassar el màxim consum permès pel dissipador, que són 120W?

- $P_1 = 60 \text{ W}$
- $P_2 = 30 \cdot 10^{-9} \cdot 1,1^2 \cdot (7/3) \cdot 10^9 = 84,7 \text{ W}$
- $P_3 = 30 \cdot 10^{-9} \cdot 1,2^2 \cdot (8/3) \cdot 10^9 = 115,2 \text{ W}$
- $P_4 = 30 \cdot 10^{-9} \cdot 1,3^2 \cdot (9/3) \cdot 10^9 = 152,1 \text{ W}$

- b) (0,5 pts) Quin és el guany de rendiment (o speedup) que s'obté amb la combinació número 4 respecte de la combinació número 1, executant el mateix programa?

Mismo programa y misma CPU, implica mismo número de instrucciones y CPI. Sólo cambia la frecuencia.

$$\text{Speedup} = T_1 / T_4 = (N_{\text{ciclos}} / F_1) / (N_{\text{ciclos}} / F_4) = 9/6 = 1,5$$

- c) (0,5 pts) Amb la combinació número 1, quin és el temps d'execució (en segons) d'un programa que executa 10^{10} instruccions i que té un CPI promig de 4?

$$T_{\text{exe}} = N \cdot CPI \cdot T_c = N \cdot CPI / F$$

$$T_{\text{exe}} = 10^{10} \cdot 4 / 2 \cdot 10^9 = 40 / 2 = 20 \text{ s}$$

Overflow de la Suma y Resta

- ❑ Algunos Lenguajes de Alto Nivel requieren que el programa genere una excepción si se produce un overflow.
- ❑ MIPS implementa por hardware esta detección.

Instrucciones	Acción al detectar overflow
<code>add, addi, sub</code>	Genera una excepción
<code>addu, addiu, subu</code>	Ignora

- ❑ En C, el overflow se ignora.
 - Si necesitamos calcularlo, hay que hacerlo por programa

si $c = a + b$ son int, hay overflow si $\sim(a_{31} \wedge b_{31}) \& (a_{31} \wedge c_{31})$

a y b tienen el mismo signo el resultado tiene signo diferente

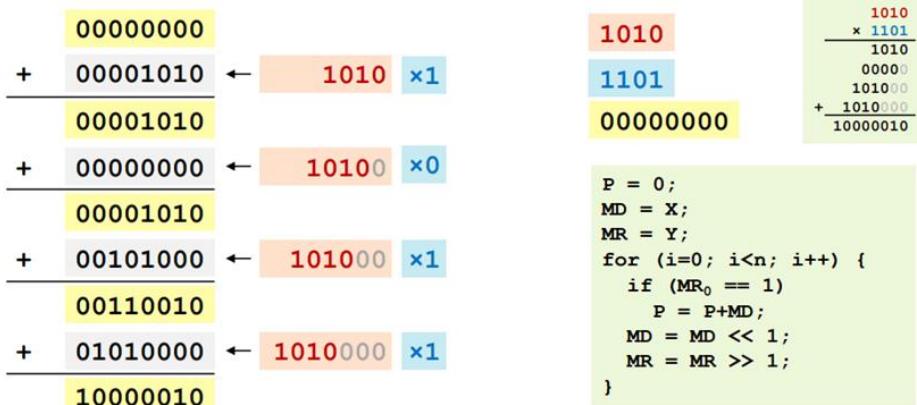
Overflow de la Suma y Resta de Naturales

- ❑ Queremos saber cómo detectar que el resultado de una operación de +/- con números naturales n bits no es representable en n bits, se ha producido un **overflow**.
- ❑ Rango de un número natural de n bits: $0 \dots 2^{n-1}$
- ❑ En hardware es muy simple, es el bit de acarreo del último sumador:
 - Lo llamamos **carry** para la suma
 - Lo llamamos **borrow** para la resta
- ❑ Se puede detectar por software, sabiendo que:
 - $c = a + b$, se produce carry (overflow) si $c < a$ (naturales) [$b = c - a$]
 - $c = a - b$, se produce borrow (overflow) si $a < b$ (naturales)

```
addiu $t2,$t0,$t1    # c = a+b
sltu $t3,$t2,$t0     # $t3=carry
```

Algoritmo Secuencial de Multiplicación de naturales

- Si usamos sumadores convencionales, hemos de hacer la multiplicación en serie.



División de enteros

- El signo del resto requiere una explicación.
- Siempre se ha de cumplir:

$$\text{Dividendo} = \text{Cociente} \times \text{Divisor} + \text{Resto}$$

- Para comprender cómo se determina el signo del resto:

Dividendo	Divisor	Cociente	Resto	Comprobación
7	2	3	1	$7 = 3 \times 2 + 1$
-7	2	-3	-1	$-7 = (-3) \times 2 + (-1)$
7	-2	-3	1	$7 = (-3) \times (-2) + 1$
-7	-2	3	-1	$-7 = 3 \times (-2) + (-1)$

- Alguien podría hacer: $-7/2 = -4$ y $-7\%2 = 1$, cumple que $(-4 \times 2 + 1) = -7$
- Pero entonces: $-(x/y) \neq (-x)/y$

Cociente negativo si Dividendo y Divisor tienen signos contrarios.

Resto y Dividendo han de tener el mismo signo

Overflow en MIPS al Multiplicar

- En C y otros Lenguajes de Programación, el resultado de multiplicar 2 números de 32 bits se guarda en 32 bits. Los 32 bits altos se ignoran.
- Si queremos saber si se ha producido overflow en MIPS, hemos de comprobar manualmente las siguientes condiciones:
 - multu** tiene overflow si **\$hi** $\neq 0$
 - mult** tiene overflow si **\$hi** no es la extensión de signo de **\$lo**
 - Si bit 31 de **\$lo** es 1, **\$hi** ha de ser -1, sino overflow
 - Si bit 31 de **\$lo** es 0, **\$hi** ha de ser 0, sino overflow

Divisiones y Desplazamientos (srl y sra)

- Comentamos que es posible sustituir algunas divisiones por desplazamientos a la derecha: **srl** y **sra**.
- Podemos hacerlo cuando tengamos divisores potencia de 2 (2, 4, 8, 16, ...).
- Dividiendo naturales (con divisor 2^x), **srl** da el mismo resultado que **divu**.
- Dividiendo enteros (con divisor 2^x), **sra**, NO da el mismo resultado que **divu**, si el dividendo es negativo.
- En general, traduciremos siempre las divisiones y módulos (/ y %) a **div** y **divu**, dependiendo del tipo (int o unsigned)

Overflow / Errores en la división

- Si el divisor es 0, el resultado es indefinido. En C una división por cero es **undefined behaviour** y puede acabar en una excepción.
- La división de naturales nunca puede provocar un overflow.
- La división de enteros, sólo tiene un caso de overflow:
 - Si dividimos el número entero más pequeño por -1, el resultado es **no representable**
 - Con 32 bits: $-2^{31} / -1 = 2^{31}$, no representable en 32 bits.

Instrucciones para Dividir en MIPS

- Instrucción en MIPS para dividir 2 números de 32 bits

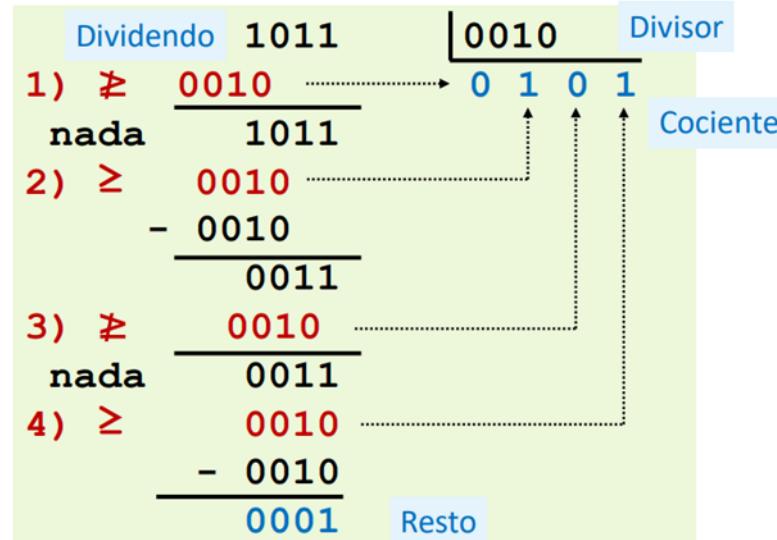
```
div  rs, rt # $lo = rs/rt; $hi = rs%rt enteros  
divu rs, rt # $lo = rs/rt; $hi = rs%rt naturales
```

- 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
```

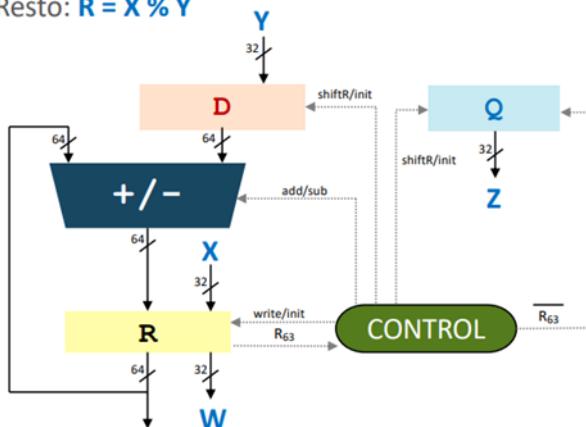
Algoritmo de División en base 2



Círcuito para la División de naturales

- División de Naturales de 32 bits “con restauración”.

- Cociente: $Q = X / Y$
- Resto: $R = X \% Y$



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

Punto Fijo

- ¿Cómo representamos números reales y/o fraccionarios?
 - Imprescindibles para la física, la ingeniería, ...
- Primera idea: En **PUNTO FIJO**
 - Unos cuantos bits para la parte entera y otros para la parte fraccionaria
 - Ejemplo con 8 bits:

eeeeefff → parte entera y parte fraccionaria
10101.110 =
 $= 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} =$
 $= 16 + 4 + 1 + 0,5 + 0,25 = 21.75$

- El rango es bastante limitado
- La distancia entre los números representables es equidistante.

Coma Flotante

En COMA FLOTANTE (base 10)

- También conocida como notación científica o exponencial

$$v = \pm m \times 10^e \rightarrow m = \text{mantisa}, e = \text{exponente}$$

- Rango mucho más grande que en punto fijo.
- Pero la distancia entre los números representables no es equidistante.

Notación científica NORMALIZADA (base 10)

$$v = \pm m \times 10^e \rightarrow \text{tal que } 1.\text{xxx} \leq m \leq 9.\text{xxx}$$

- La parte entera ha de tener 1 solo dígito, entre 1 y 9.
- Ejemplos:
 - $2.34 \times 10^6 \rightarrow \text{NORMALIZADO}$
 - $0.234 \times 10^7 \rightarrow \text{NO NORMALIZADO}$

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

$$v = \pm 1.\text{ffff...f} \times 2^{\text{ee...e}}$$

parte entera fracción(F) exponente (E)
signo (S) mantisa

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

Un forma más compacta.

Formato: Signo, Exponente, Fracción

$$S \text{ EEEEEEEE } FFFFFFFFFFFFFFFFFFFF$$

- 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
- Si dedicamos más bits a **exponente**, tendremos **MAYOR RANGO**
- Si dedicamos más bits a **fracción**, tendremos **MAYOR PRECISIÓN**

IEEE-754, valores representables

Mantisa

$$1.000\dots0 \leq \text{mantisa} \leq 1.111\dots1$$

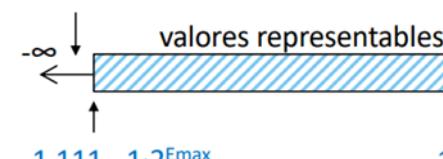
Exponente

$$E_{\min} \leq E \leq E_{\max}$$

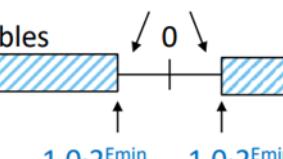
Resultados fuera de rango (**overflow**): $E > E_{\max}$

Resultado "poco fiables" (**underflow**): magnitud $< 1.0 \cdot 2^{E_{\min}}$

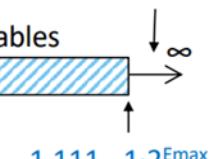
overflow



underflow



overflow



IEEE-754, Error de precisión por redondeo

- Por ejemplo, si queremos representar el racional $1/10$ en simple precisión:



- Si lo redondeamos a v_0 (eliminando bits)

- Error de precisión ($\epsilon = |v-v_0|$)

IEEE-754, Codificaciones especiales

CERO

- No hay ninguna combinación válida de fracción y mantisa que de cero.
- Usaremos una codificación especial para el cero
 - $E = 000\dots0$
 - $F = 00000\dots0$
- Teniendo en cuenta el bit de signo, tenemos dos codificaciones para el cero: $+0$ y -0
- En simple precisión el cero es:

s00000000000000000000000000000000

INFINITO (“Inf”)

- Sigue algunas reglas básicas de operación. Algunos ejemplos:
 - $\frac{1}{0} = +\infty$, $\frac{1}{\infty} = 0$, $x + \infty = \infty$,
- Idea muy útil que permite evitar el overflow en algunas expresiones.
 - Por ejemplo, $y = \frac{1}{1+\frac{100}{x}}$, $\frac{100}{x}$ generaría un overflow si $x \rightarrow 0$
 - Si usamos el infinito, $\frac{100}{x} = \infty$ y resulta que $y = 0$.
- Usaremos una codificación especial para el infinito (Inf y $-\text{Inf}$)
 - $E = 1111\dots1$
 - $F = 00000\dots0$
- En simple precisión el infinito es:

s11111111000000000000000000000000

IEEE-754, Error de precisión por redondeo

- Margen del error absoluto en el intervalo (v_0, v_1) :

- $\epsilon_{\max} = |v_1 - v_0|$

- Margen del error absoluto en simple precisión

- $v_0 = m \times 2^E$

- $v_1 = (m+2^{-23}) \times 2^E$

- $\epsilon_{\max} = |v_1 - v_0| = (m+2^{-23}) \times 2^E - m \times 2^E = 2^{E-23}$

- A veces, el error absoluto no nos dice nada:

- $\epsilon = 1 \text{ metro}$

- ✓ Es aceptable si estamos calculando la distancia entre el sol y la tierra.
- ✓ Inaceptable si estamos calculando la longitud de un puente.

IEEE-754, Error de precisión por redondeo

- Error relativo: $\eta = \epsilon / |v|$

- Margen del error relativo en simple precisión

- $\eta_{\max} < \frac{\epsilon_{\max}}{|v_0|} = \frac{2^{E-23}}{m \cdot 2^E} = \frac{2^{-23}}{m}$

- Y como $m \geq 1.0$, nos queda que

- $\eta_{\max} < 2^{-23} = 1 \text{ ULP}$ (Unit in the Last Place)

NOT A NUMBER (“NaN”)

- Representa un **resultado inválido**, en algunas operaciones:
 - $\sqrt{-1} = NaN$, $\infty - \infty = NaN$, $\frac{\infty}{\infty} = NaN$, $\log(-1) = NaN$, etc.
 - En general, cualquier operación con un NaN, da como resultado NaN
- El programa puede comprobar si el resultado es válido o inválido.
 - En una cadena de cálculos podemos diferir la comprobación hasta el final.
- Codificación de NaN
 - $E = 1111\dots1$
 - $F \neq 00000\dots0$
- En simple precisión el NaN es:

s11111111XXXXXXXXXXXXXXXXXXXX

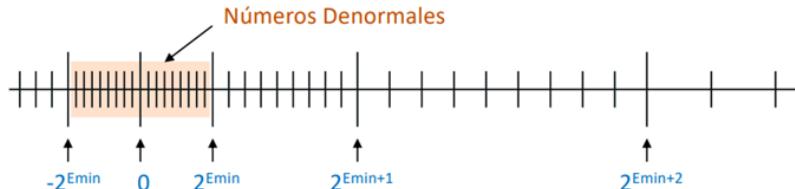
algún $X \neq 0$

IEEE-754, Codificaciones especiales

DENORMALES

- Un resultado muy pequeño, del tipo $|v| < 2^{\text{Emin}}$ no es fiable, ya que si lo redondeamos a 0 el error de precisión es enorme:

$$\circ \epsilon = |v - 0| = |v| \quad \eta = \frac{|v-0|}{|v|} = 1$$



- En estos casos podemos mejorar la precisión admitiendo que existan números **NORMA
NORMALIZADOS** o **DENORMALES** en el rango $(-2^{\text{Emin}}, 0, 2^{\text{Emin}})$:

$$y = (-1)^s \times 0.FFFF...F \times 2^{\text{Emi}}$$

IEEE-754, Resumen de formatos

The diagram illustrates the IEEE 754 floating-point number representation. It shows the bit fields for sign (S), exponent (E), and fraction (F) across three categories: Cero, Normalizado, and Denormal. The exponent field E is divided into three ranges: $E = 000\dots0$ (0 to 2²⁴-2), $resto$ (2²⁴-2 to 2³²-25), and $E = 111\dots1$ (2³²-25 to 2²⁴). The fraction field F is divided into $F = 000\dots0$ (0 to 2²⁴-2) and $resto$ (2²⁴-2 to 2³²-25). The resulting values are Cero, Infinito, Normalizado, Denormal, and NaN. An arrow points from the text "#valores de cada tipo en precisión simple (32b)" to the boundary between the $resto$ range of the exponent and the $resto$ range of the fraction.

	$E = 000\dots0$	$resto$	$E = 111\dots1$
$F = 000\dots0$	Cero		Infinito
$resto$	Denormal	Normalizado	NaN

#valores de cada tipo
en precisión simple (32b)

IEEE-754, Codificaciones especiales

DENORMALES

- ❑ Admitimos números **NO NORMALIZADOS o DENORMALES** en el rango $(-2^{\text{Emin}}, 0, 2^{\text{Emin}})$:

$$v = (-1)^s \times 0.FFFF...F \times 2^{E_{\text{min}}}$$

- En simple precisión, el margen superior de error absoluto es:

$$\circ \quad \mathcal{E} = |v_1 - v_0| = 2^{E_{\min} - E}$$

- ## □ Codificación de Denormalización

○ E = 000...

- $F \neq 000000\dots$

- En simple precisión es

¡Atención!

Si operamos con denormales:

- El bit oculto implícito es 0.
 - El exponente implícito es E_{min} .

s00000000x

algún $x \neq 0$

Resumiendo

IEEE-754

- #### ☐ **float** (precisión simple, 32b, 4B)

- 1 bit de signo, 8 bits de exponente (exceso 127) y 23 bits de mantisa
 - Rango:

$$\checkmark (+) 1.18 \cdot 10^{-38} \leq x \leq 3.4 \cdot 10^{38}$$

- ✓ aproximadamente 7 dígitos decimales de precisión

- #### ■ **double** (precisión doble, 64b, 8B)

1 bit u

Rango:

- aproximadamente 15 dígitos decimales de precisión.

IEEE-754, Modos de Redondeo

❑ Hacia Cero (Truncado):

- $0 < v_0 < v < v_1$
- $v \rightarrow v_0$
- Es el más fácil de implementar, sólo hay que eliminar los bits que sobran (truncar)
- El margen de error en simple precisión es:
 - ✓ $\epsilon_{\max} < |v_1 - v_0| = 2^{-23}$
 - ✓ $\epsilon_{\max} < 2^{-23}$
 - ✓ $\eta_{\max} < 2^{-23} = 1 \text{ ULP}$

❑ Hacia +Infinito:

- $v_0 < v < v_1$
- $v \rightarrow \max(v_0, v_1)$

❑ Hacia -Infinito:

- $v_0 < v < v_1$
- $v \rightarrow \min(v_0, v_1)$

❑ Este tipo de redondeo se utiliza cuando trabajamos con aritmética de intervalos:

- Altura media = $1,75 \pm 5 \text{ cm}$

❑ Hacia el más próximo:

- Es el método usado por defecto, porque da el menor error posible
- Hacia el par si v está justo en medio
- $v_0 < v < v_1$, si v es equidistante es cuando se produce el máximo error
 - ✓ $\epsilon_{\max} < |v_1 - v_0|/2$
 - ✓ $\epsilon_{\max} < 2^{-24}$
 - ✓ $\eta_{\max} < 2^{-24} = 0.5 \text{ ULP}$

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):

$$v = 1.\underset{\text{xxxxx}}{\dots}0011 \quad 0000101$$
$$v_0 = 1.\underset{\text{xxxxx}}{\dots}0011$$

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

$$v = 1.\underset{\text{xxxxx}}{\dots}0011 \quad 1010110$$
$$+ 0.00000\dots0001$$
$$v_1 = 1.\underset{\text{xxxxx}}{\dots}0100$$

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

$$v = 1.\underset{\text{xxxxx}}{\dots}0011 \quad 1000000$$

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

$$v_0 = 1.\underset{\text{xxxxx}}{\dots}0011$$
$$v_1 = 1.\underset{\text{xxxxx}}{\dots}0100 \leftarrow \text{"par"}$$

Conversión de base 10 a base 2

Queremos representar $v = -1029,68$

1. Convertir la parte entera, por divisiones sucesivas

$$1029 = 10000000101 \text{ (necesitamos 11 bits)}$$

2. Convertir la fracción, por productos sucesivos:

$$0,68 \times 2 = 1,36 \rightarrow 1$$

$$0,36 \times 2 = 0,72 \rightarrow 0$$

$$0,72 \times 2 = 1,44 \rightarrow 1$$

...

◦ ¿Cuántos bits calculamos? → 13 bits para completar los 24 bits de la mantisa

$$0,68 = 0,1010111000010 \text{ (13 bits)}$$

y algunos bits extra para decidir cómo redondeamos

$$0,68 = 0,101011100001010001 \text{ (18 bits)}$$

Conversión de binario (coma flotante) a base 10

Queremos saber que valor tiene $v = 0x45814140$

- Escribirlo en binario

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

- Identificamos los 3 campos: signo, exponente, fracción

$$v = 0\ 10001011\ 00000010100000101000000$$

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

$$10001011 = 139$$

$$E = 139 - 127 = 12$$

- Componer el número: signo, bit oculto, fracción y exponente

$$v = +1.\ 00000010100000101000000 \times 2^{12}$$

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

$$v = +1000000101000.00101000000$$

- Convertir la parte entera a base 10

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

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

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

- Finalmente, juntar la parte entera y la fracción:

$$v = 4136,15625$$

Operaciones en Coma Flotante: Suma y Resta

- Primero recordaremos la suma que ya conocemos (base 10):

- Formato: mantisa normalizada 4 dígitos: $z.zzz \times 10^{zz}$

- Sumar: $9,999 \times 10^1 + 1,680 \times 10^{-1}$

- ¿Cómo lo hacemos?

$$\begin{array}{r} 9,999 \times 10^1 \\ + 1,680 \times 10^{-1} \\ \hline = 11,679 \times 10^1 \end{array} \quad \leftarrow \text{ ¡Así, NO se hace!}$$

- Igualar exponentes (al mayor, =1), alinear mantisas y sumar:

$$\begin{array}{r} 9,99900 \times 10^1 \\ + 0,01680 \times 10^1 \\ \hline = 10,01580 \times 10^1 \end{array}$$

- Normalizar:

$$= 1,001580 \times 10^2$$

- Redondear a 4 dígitos:

$$= 1,002 \times 10^2$$

Conversión de base 10 a base 2

- Juntar la parte entera y la fracción ($11+18 = 29$ bits)

$$1029,68 = 10000000101,101011100001010001$$

- Normalizar, mover la coma hacia la izquierda para que la mantisa sea 1,xxx

$$1029,68 = 1,0000000101101011100001010001 \times 2^{10}$$

- Redondeamos al más próximo usando los bits extra,
→ redondeamos al siguiente

$$1029,68 = 1,00000001011010111000011 \times 2^{10}$$

- Codificar el exponente en exceso a 127

$$E = 10 + 127 = 137 = 10001001$$

- Juntar Signo, Exponente y Fracción

$$1\ 10001001\ 0000001011010111000011$$

- Expresar el número en hexadecimal

$$-1029,68 = 0xC480B5C3$$

Conversión de base 10 a base 2

- Podemos calcular el error de precisión ($\epsilon = |v-v_0|$)

Restando el valor redondeado y el exacto

$$\begin{aligned} \epsilon &= 1,00000001011010111000011 \times 2^{10} \\ &- 1,0000000101101011100001010001 \times 2^{10} \\ &= 0,00000000000000000000000000000001111 \times 2^{10} \end{aligned}$$

Normalizamos, movemos la coma 25 posiciones a la derecha

$$\begin{aligned} \epsilon &= 00000000000000000000000000000001,111 \times 2^{10-25} \\ &= 1,111 \times 2^{-15} \end{aligned}$$

Lo pasamos a decimal (no se pedirá sin calculadora),

$$\epsilon = 1,875 \times 2^{-15} = 5,722 \times 10^{-5}$$

- En MIPS, tenemos el CP1 (coprocesador 1) que es la Unidad de coma Flotante
 - Banco de registros propio. 32 registros de 32 bits: \$f0 ... \$f31
 - O bien 16 registros dobles: \$f0-\$f1, \$f2-\$f3 ... \$f30-\$f31 (se identifican con el par)
 - Todas las operaciones aritméticas en coma flotante se realizan con registros \$fxx.
 - Puede operar con floats (32 bits, 4B) o doubles (64 bits, 8B).
 - Dispone de un registro de control adicional para configurar el redondeo, gestionar excepciones, comparaciones, ...
 - Dispone de instrucciones especiales para mover datos entre los registros de la CPU y los registros del coprocesador

Declaraciones MIPS de Coma Flotante

- Declaración de variables GLOBALES en coma flotante

Tipo C	MIPS	Tamaño
float	.float	4 bytes (1 word)
double	.double	8 bytes (2 words)

- Las directivas float y double alinean a 4 y 8 bytes respectivamente

- Ejemplo

```
float v[2] = {3.1416, -3.5E2};
double x = 3E350, y;
```

Declaraciones globales en C

```
.data
v: .float 3.1416, -3.5E2
x: .double 3E350
y: .double 0.0
```

Declaraciones en MIPS

Instrucciones MIPS de Coma Flotante

- Acceso a Memoria

Simple Precisión (float)	Doble Precisión (double)	Comportamiento
lwcl ft, offset(rs)	ldcl ft, offset(rs)	load de memoria
swcl ft, offset(rs)	sdc1 ft, offset(rs)	store en memoria

```
lwcl $f3, 40($t2)      # $f3 = M[$t2+40], leemos 4 bytes
swcl $f4, 10($t3)      # M[$t3+10] = $f4, escribimos 4 bytes

ldcl $f2, 40($t2)      # $f3:$f2 = M[$t2+40], leemos 8 bytes
sdc1 $f8, 10($t3)      # M[$t3+10] = $f9:$f8, escribimos 8 bytes
```

- Aritméticas

Simple Precisión (float)	Doble Precisión (double)	Comportamiento (float)
add.s fd,fs,ft	add.d fd,fs,ft	fd = fs + ft
sub.s fd,fs,ft	sub.d fd,fs,ft	fd = fs - ft
mul.s fd,fs,ft	mul.d fd,fs,ft	fd = fs * ft
div.s fd,fs,ft	div.d fd,fs,ft	fd = fs / ft

- Movimiento de registros

Instrucción	Comportamiento
mfcl rt, fs	rt = fs
mtcl rt, fs	fs = rt
mov.s fd, fs	fd = fs

- 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)

Subrutinas con Coma Flotante

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

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

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**.

Asociatividad de la Suma en Coma Flotante

- ❑ La suma de números en coma flotante **NO CUMPLE** la propiedad **ASOCIATIVA**:

$$x + (y + z) \neq (x + y) + z$$

- ❑ Un ejemplo muy simple: $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, $z = 1.0$
 - $x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0) = -1.5 \times 10^{38} + 1.5 \times 10^{38} = 0.0$
 - $(x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0 = 0.0 + 1.0 = 1.0$

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
    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

- ❑ Las diferencias no tienen porque ser muy grandes, pero hay cosas a evitar

```
if (suma == 1.0)
    ...
```

- ❑ Es una construcción “peligrosa”. Si hay que comprobar un resultado, hay que calcular el error máximo aceptable y ver si el resultado está dentro de ese intervalo: $1.0 \pm \epsilon$

```
if (abs(suma-1.0) <= epsilon)
    ...
```

Memorias de Semiconductores

Tipos de Memoria de Semiconductores:

- **Memoria Estática** (SRAM, Static RAM). Cada celda de memoria equivale a 1 biestable (6-8 transistores). En comparación con las DRAM son **rápidas**, tienen un **alto consumo**, **pequeñas** (poca capacidad) y **caras**.

→ Memoria Cache

- **Memoria Dinámica** (DRAM, Dynamic RAM). Cada celda se comporta como un condensador (1-1.x transistores). En comparación con las SRAM son **lentas**, tienen un **bajo consumo, grandes** (mucha capacidad) y **baratas**. **Problema del refresco**.

→ Memoria Principal

Propiedades de los Programas

Localidad Temporal

- Si accedemos a una posición de memoria, es muy probable que se vuelva a acceder a la misma posición en un futuro cercano.

```
X3D(int v[], int w[]) {  
    int vert, polig, color;  
    ...  
    for (i=0; i<5000; i++) {  
        v[i] = w[i] | 0x01;  
        v[i] = v[i] * vert;  
        if (v[i] != 0)  
            w[i] = polig / v[i];  
        v[i] = w[i] + color;  
    }  
    ...  
}
```

¿Por qué? Por los bucles.

- **Instrucciones**. Dentro de un bucle se accede repetidamente a las mismas instrucciones.
- **Datos**. Dentro de un bucle se accede repetidamente a las mismas variables.

Propiedades de los Programas

Localidad Espacial

- Si accedemos a una posición de memoria, es muy probable que se acceda a posiciones próximas en un futuro cercano.

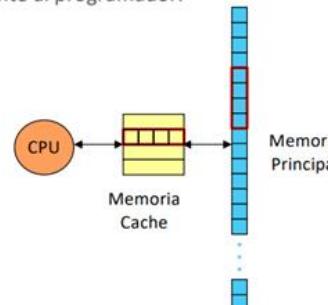
```
X3D(int v[], int w[]) {  
    int vert, polig, color;  
    ...  
    for (i=0; i<5000; i++) {  
        v[i] = w[i] | 0x01;  
        v[i] = v[i] * vert;  
        if (v[i] != 0)  
            w[i] = polig / v[i];  
        v[i] = w[i] + color;  
    }  
    ...  
}
```

¿Por qué?

- **Instrucciones**. Las instrucciones se ejecutan en secuencia.
- **Datos**. Los vectores y matrices se suelen recorrer completos en secuencia; los parámetros y variables locales suelen estar en el bloque de activación de la subrutina..

Principios de Funcionamiento de las Memorias Cache

- **Memoria Cache**: memoria pequeña y rápida que almacena una parte del contenido de una memoria más grande y lenta. La memoria cache se encargará de que la información que se almacene sea útil. Esta memoria es transparente al programador.



- **Objetivo**:
 - **Velocidad** de la memoria cache.
 - **Capacidad** de la memoria principal.
 - **Coste** de la memoria principal más un porcentaje razonable.

- Esta solución es posible debido a la **localidad de los programas**. La cache retiene información recientemente usada e información cercana a la recientemente usada.

- **Conceptos**:
 - Acierto / Fallo
 - Línea / Bloque de memoria
 - Algoritmos emplazamiento
 - Algoritmos reemplazo
 - Políticas de Escritura
 - Evaluación

Terminología y Funcionamiento

```
for (i=0; i<200; i++)  
    sum = sum + V[i];
```

```
la $t0,V  
move $t1,$zero  
move $t2,$zero  
for: bge $t1,$t2,end  
lw $t3,0($t0)  
addu $t2,$t2,$t3  
addu $t0,$t0,4  
addu $t1,$t1,1  
b for  
end:
```

Terminología y Funcionamiento

```
for (i=0; i<200; i++)  
    sum = sum + V[i];
```

```
la $t0,V  
move $t1,$zero  
move $t2,$zero  
for: bge $t1,$t2,end  
lw $t3,0($t0)  
addu $t2,$t2,$t3  
addu $t0,$t0,4  
addu $t1,$t1,1  
b for  
end:
```

- **Referencia a Memoria**. Acceso a Memoria
- **Acierto (HIT)**. El dato accedido por la CPU **ESTÁ** en Memoria Cache.

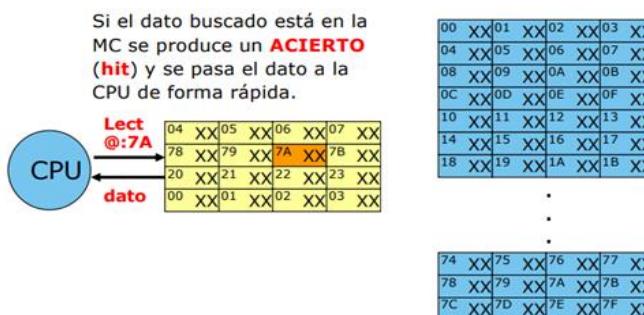
- **Fallo (MISS)**. El dato al que intenta acceder la CPU **NO ESTÁ** en Memoria Cache.

- ¿Qué hacemos en caso de FALLO?
 - Hay que traer el dato que provoca el fallo a Memoria Cache.
 - No sólo traemos ese dato, traemos un **bloque de datos** (p.e. 16 bytes).

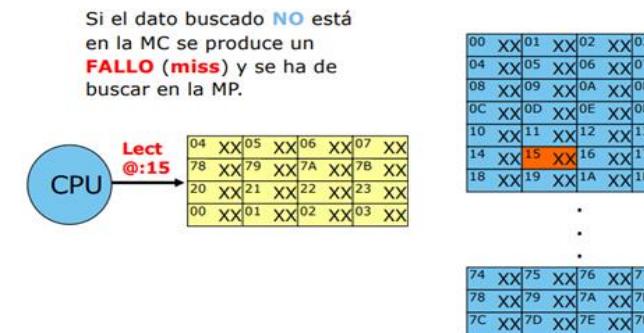
V[0] V[1] V[2] V[3] V[4] V[5] ...

- Accedemos a V[0] ⇒ **FALLO**
 - Traemos a MC Bloque de datos: V[0] – V[3] (16 bytes)
- Accedemos a V[1] ⇒ **ACIERTO**
- Accedemos a V[2] ⇒ **ACIERTO**
- Accedemos a V[3] ⇒ **ACIERTO**
- Accedemos a V[4] ⇒ **FALLO**
 - Traemos a MC Bloque de datos: V[4] – V[7] (16 bytes)
- Accedemos a V[5] ⇒ **ACIERTO**
- ...

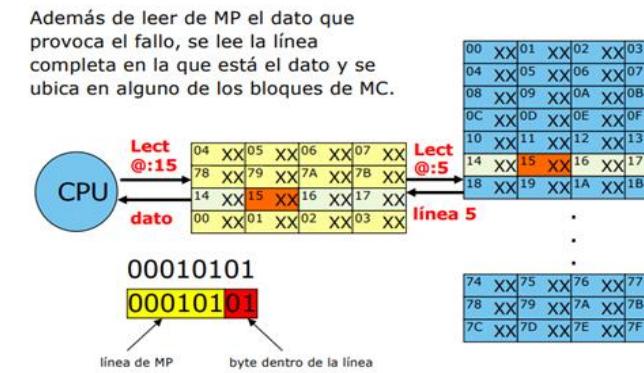
LOCALIDAD
ESPACIAL



Principios de Funcionamiento de las Memorias Cache



Principios de Funcionamiento de las Memorias Cache



la...	mo...	mo...	bg...	lw...	add...	add...	add...	b...
-------	-------	-------	-------	-------	--------	--------	--------	------

```

la $t0,V
move $t1,$zero
move $t2,$zero
for: bge $t1,$t2,end
      lw $t3,0($t0)
      addu $t2,$t2,$t3
      addu $t0,$t0,4
      addu $t1,$t1,1
      b for
end:

```

- ❑ fetch "la..." ⇒ **FALLO**
 - Traemos a MC Bloque de inst: "la ... bge" (16 bytes)
- ❑ fetch "move ..." ⇒ **ACIERTO**
- ❑ fetch "move ..." ⇒ **ACIERTO**
- ❑ fetch "bge ..." ⇒ **ACIERTO**
- ❑ fetch "lw ..." ⇒ **FALLO**
 - Traemos a MC Bloque de inst: "lw ... add" (16 bytes)
- ❑ fetch "add ..." ⇒ **ACIERTO**
- ❑ fetch "add ..." ⇒ **ACIERTO**
- ❑ fetch "add ..." ⇒ **ACIERTO**
- ❑ fetch "b ..." ⇒ **FALLO**
 - Traemos a MC Bloque de inst: "b ..." (16 bytes)
- ❑ fetch "bge ..." ⇒ **ACIERTO**
- ❑ fetch "lw ..." ⇒ **ACIERTO**
- ❑ ...

LOCALIDAD TEMPORAL

LOCALIDAD ESPACIAL

Terminología y Funcionamiento

```

for (i=0; i<200; i++)
    sum = sum + V[i];

```

```

la $t0,V
move $t1,$zero
move $t2,$zero
for: bge $t1,$t2,end
      lw $t3,0($t0)
      addu $t2,$t2,$t3
      addu $t0,$t0,4
      addu $t1,$t1,1
      b for
end:

```

- ❑ **Referencia a Memoria.** Acceso a Memoria
- ❑ **Acierto (HIT).** El dato accedido por la CPU **ESTÁ** en Memoria Cache.
- ❑ **Fallo (MISS).** El dato al que intenta acceder la CPU **NO ESTÁ** en Memoria Cache.
- ❑ **Tasa de aciertos (hit ratio).** Porcentaje de aciertos.
 - $h = \# \text{aciertos} / \# \text{referencias}$
- ❑ **Tasa de fallos (miss ratio).** Porcentaje de fallos.
 - $m = \# \text{fallos} / \# \text{referencias}$
 - $m = 1 - h$

Problema, calculad la tasa de fallos (m)

```

for (i=0; i<200; i++)
    sum = sum + V[i];

```

```

la $t0,V
move $t1,$zero
move $t2,$zero
for: bge $t1,$t2,end
      lw $t3,0($t0)
      addu $t2,$t2,$t3
      addu $t0,$t0,4
      addu $t1,$t1,1
      b for
end:

```

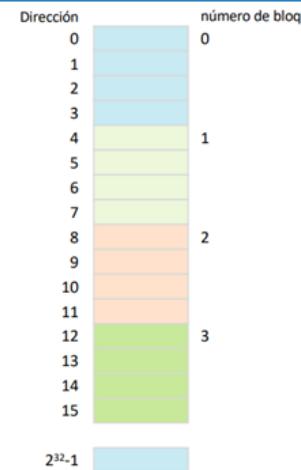
- ❑ **Accesos a Memoria:**
 - A instrucciones: 1205
 - A datos: 200
- ❑ **Fallos de Cache** (consideraremos 16 bytes por línea)
 - Instrucciones: 3 fallos
 - Datos: 50 fallos (1 de cada 4)
- ❑ **Tasa de fallos:**
 - $m_i = 3 / 1205 = 0,0025 = 0,25\%$
 - $m_D = 50 / 200 = 0,25 = 25\%$
 - $m = 53 / 1405 = 0,038 = 3,8\%$

Principios de Funcionamiento de las Memorias Cache

- En cualquier Memoria Cache hay que definir:
 - Algoritmo de Emplazamiento.** Determina en qué bloques de MC puede colocarse una línea. Determina, también, dónde hay que buscar un dato.
 - Algoritmo de reemplazo.** Determina qué línea se ha de eliminar de la cache para dejar espacio a una nueva línea.
 - Políticas de escritura.** determina cómo se hacen las escrituras. En cualquier caso, al final siempre se ha de escribir en MP.
- Han de ser algoritmos hardware:
 - Algoritmos sencillos.
 - Algoritmos rápidos.

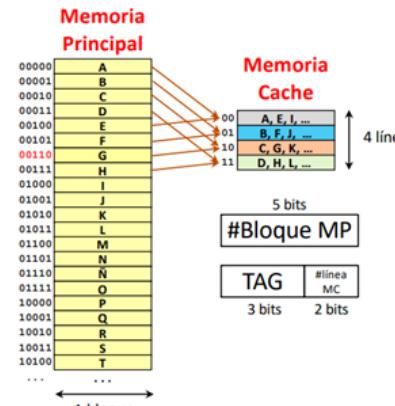
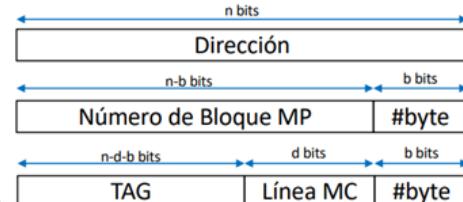
Organización de la MP

- La Memoria Principal (MP) se organiza en bloques de tamaño fijo ($B = 2^b$ bytes)
- Dada una dirección,
 - ¿Cómo averiguamos el número de bloque?
Número de bloque = Dirección / Tamaño bloque
 - ¿Cuál es el byte (offset, desplazamiento) dentro del bloque?
Byte dentro del bloque = Dirección % Tamaño bloque
- Como el tamaño de bloque siempre es potencia de 2, este cálculo es muy simple:



Cache Directa

- En una Memoria Cache, podemos almacenar D bloques de Memoria Principal.
- La forma más sencilla de implementar una Memoria Cache es una **Memoria Cache Directa**
⇒ Un Bloque de MP se almacena siempre en la misma línea (o bloque) de Memoria Cache
- Para que sea fácil de implementar, hacemos que el número de bloques de una Memoria Cache Directa sea potencia de dos ($D = 2^d$).

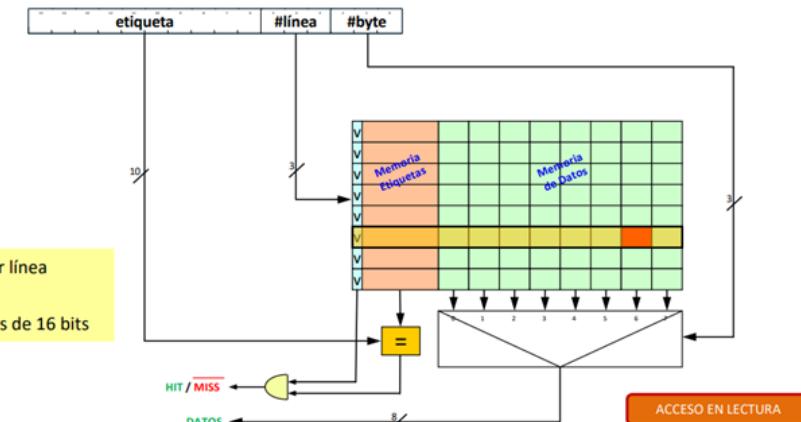


Cache Directa

- Podemos ver una Memoria Cache Directa como una tabla
- En cada entrada de la tabla tenemos 1 línea de cache.
- La información que almacenamos de cada línea es:
 - DATOS**, contiene los 2^b bytes del bloque
 - ETIQUETA**, información necesaria para identificar el bloque de MP
 - BIT de VALIDEZ**, indica si la línea está ocupada

#línea MC	V	Etiqueta	DATOS
0			
1			
2			
3			

Cache Directa



Problema

- Suponiendo un tamaño de bloque $B = 64$ bytes y una cache directa con 32 bloques. Calculad número de línea de MP, la etiqueta, número de línea de MC y desplazamiento para las direcciones de memoria:

26 bits	6 bits
Número de Bloque MP	#byte
21 bits	5 bits

- A = 0x100101C0 = 0001 0000 0000 0001 0000 0001 1100 0000 =
= 00 0100 0000 0000 0100 0000 0111 00 0000 = 0x0400407, 0x00
= 00010 0000 0000 0010 0000 0 0111 00 0000 = 0x20020, 0x07, 0x00
- A = 0x1001060F = 0001 0000 0000 0001 0000 0110 0000 1111 =
= 00 0100 0000 0000 0100 0001 1000 00 1111 = 0x0400418, 0x0F
= 00010 0000 0000 0010 0000 1 1000 00 1111 = 0x020020, 0x18, 0xF

Problema

Disponemos de un sistema de Memoria con direcciones de 32 bits y una MC directa de 64Kbytes y 32 bytes por línea.

1. Dada una dirección de memoria indicad, mediante un dibujo, qué bits se utilizan para seleccionar el byte dentro de la línea, qué bits se utilizan para seleccionar la línea de memoria cache, y qué bits forman la etiqueta.
2. ¿Cuántos bits ocupa la memoria de etiquetas (tags)?

Tamaño línea: 32 bytes por línea (2^5)

Número de líneas: 2048 líneas (2^{11})

Tamaño cache: #(bytes por línea) × #líneas = 64 Kbytes

TAG	#línea	byte
16 bits	11 bits	5 bits

Tamaño Memoria TAGS: 2048 líneas × 16 bits = 32.768 bits

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.

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

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).

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
```

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
Lectura 0x0009
Lectura 0x0045
Lectura 0x0047

Operaciones a realizar en un acceso a Cache

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							
2	0							
3	0							

Operaciones a realizar en un acceso a Cache

```
0x0047 =  
0000 0000 0100 0101 =  
000000000010 00 111  
TAG = 0x002  
línea = 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 0011 =
 0000000000 00 011
 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									

0x0004 =
 0000 0000 0000 0100 =
 0000000000 00 100
 TAG = 0x000
 línea = 0x0
 byte = 0x4
 Bloque MP = 0x0000

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

V	Etiqueta	DATOS MC								
0	1	0x000	00	03	05	FF	F3	87	AD	F4
1	1	0x000	45	67	89	87	67	90	99	34
2	0									
3	0									

0x0007 =
 0000 0000 0000 0111 =
 0000000000 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									

Write Through + Write NO Allocate

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 =
 0000000000 11 111
 TAG = 0x000
 línea = 0x11
 byte = 0x7
 Bloque MP = 0x0003

Escr 0x0003: **MISS**
 Lect 0x0004: **MISS**
 Escr 0x0007: **HIT**
 Lect 0x001F: **MISS**
 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 =
 0000000000 11 010
 TAG = 0x000
 línea = 0x11
 byte = 0x2
 Bloque MP = 0x0003

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

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

0x000D =
 0000 0000 0000 1100 =
 000000000000 01 100
 TAG = 0x000
 línea = 0x01
 byte = 0x4
 Bloque MP = 0x0001

Escr 0x000D: HIT
 Lect 0x000A:
 Escr 0x002E:
 Lect 0x0037:

	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

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
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

Operaciones a realizar en un acceso a Cache

0x002E =
 0000 0000 0010 1110 =
 000000000001 01 110
 TAG = 0x001
 línea = 0x01
 byte = 0x6
 Bloque MP = 0x0005

Escr 0x000D: HIT
 Lect 0x000A: HIT
 Escr 0x002E: MISS
 Lect 0x0037:

	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	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

0x000A =
 0000 0000 0000 1010 =
 000000000000 01 010
 TAG = 0x000
 línea = 0x01
 byte = 0x2
 Bloque MP = 0x0001

Escr 0x000D: HIT
 Lect 0x000A: HIT
 Escr 0x002E:
 Lect 0x0037:

	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

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
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

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

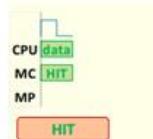
Escr 0x000D: HIT
 Lect 0x000A: HIT
 Escr 0x002E: MISS
 Lect 0x0037: MISS

	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

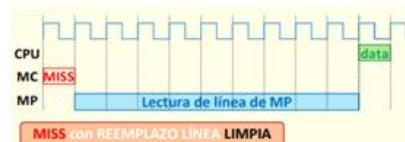
Copy Back + Write Allocate

Evaluación

Tiempo de acceso a la cache

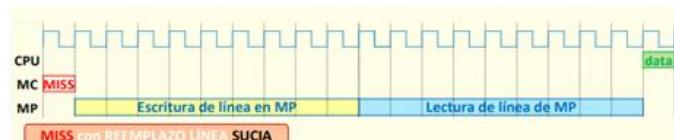


$$t_{acceso} = t_h$$



$$t_{acceso} = t_h + t_p \quad t_p = t_{bloq} + t_h$$

Copy Back + Write Allocate



$$t_{acceso} = t_h + t_p \quad t_p = t_{bloq} + t_{bloq} + t_h$$

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_p	Immediata amb assignació	Immediata sense assignació	Retardada amb assignació
Lectura - Encert	0	0	0
Lectura - Fallada	$t_{block} + t_h^1$	$t_{block} + t_h^1$	bloc modif.: $2 * t_{block} + t_h^2$ bloc no mod.: $t_{block} + t_h^1$
Escriptura - Encert	0^3	0^3	0^5
Escriptura - Fallada	$t_{block} + t_h^1$	0^4	bloc modif.: $2 * t_{block} + t_h^2$ bloc no mod.: $t_{block} + t_h^1$

Impacto de la cache en el Rendimiento

$$t_{am} = t_h + m \cdot t_p$$

$$T_{exe} = N \cdot (CPI_{ideal} + m \cdot t_p \cdot n_r) \cdot T_c$$

- ❑ No podemos mejorar sólo un componente (CPU o MC) [Ley del Amdahl]
- ❑ Hay múltiples técnicas para mejorar el rendimiento de la CPU.
- ❑ También existen técnicas para mejorar el rendimiento de la cache.
 - Si queremos mejorar la $m \Rightarrow$ ASOCIATIVIDAD
 - Si queremos mejorar $t_p \Rightarrow$ CACHES MULTINIVEL

Impacto de la cache en el Rendimiento

- ❑ El tiempo de Ejecución de un programa se modelaba así:

$$T_{exe} = N \cdot CPI \cdot T_c$$

- N , número de instrucciones
- CPI , ciclos en media por instrucción
- T_c , tiempo de ciclo del procesador

- ❑ Si tenemos una cache, el modelo cambia:

$$T_{exe} = N \cdot (CPI_{ideal} + CPI_{mem}) \cdot T_c = N \cdot (CPI_{ideal} + m \cdot t_p \cdot n_r) \cdot T_c$$

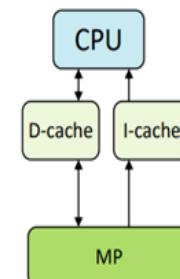
- CPI_{ideal} , ciclos en media por instrucción suponiendo una cache perfecta (sin fallos)
- CPI_{mem} , son los ciclos que perdemos por tener una cache imperfecta ($m \neq 0$)
 - ✓ m , tasa de fallos
 - ✓ t_p , tiempo de penalización en caso de fallo
 - ✓ n_r , número de referencias por instrucción

Suponemos que el fetch de instrucciones se realiza sin fallos de cache.

Problema

- ❑ Dado un sistema con caches separadas para datos e instrucciones:

- Cache de Instrucciones, $m_i = 2\%$
- Cache de Datos, $m_d = 4\%$
- $t_p = 100$ ciclos
- CPI ideal = 2,5 ciclos/inst
- 36% de las instrucciones son loads o stores.



Calculad el CPI real.

$$CPI_{memI} = n_r \cdot m_i \cdot t_p = 1 \cdot 0,02 \cdot 100 = 2 \text{ ciclos}$$

$$CPI_{memD} = n_r \cdot m_d \cdot t_p = 0,36 \cdot 0,04 \cdot 100 = 1,44 \text{ ciclos}$$

$$CPI_{real} = CPI_{ideal} + CPI_{mem} = 2,5 + 2 + 1,44 = 5,94 \text{ ciclos}$$

$$CPI_{ideal} = n_{cicles_ideal} / n_{ins}$$

Mejoras en el rendimiento: Asociatividad

```
for (i=0; i<200; i++)
    sum = sum + V[i];
```

Líneas de cache de 16B

m = 25%

LOCALIDAD
ESPACIAL

- ❑ Accedemos a V[0] ⇒ **FALLO**
 - Traemos a MC Bloque de datos: V[0] – V[3] (16 bytes)
- ❑ Accedemos a V[1] ⇒ **ACIERTO**
- ❑ Accedemos a V[2] ⇒ **ACIERTO**
- ❑ Accedemos a V[3] ⇒ **ACIERTO**
- ❑ Accedemos a V[4] ⇒ **FALLO**
 - Traemos a MC Bloque de datos: V[4] – V[7] (16 bytes)
- ❑ Accedemos a V[5] ⇒ **ACIERTO**
- ❑ ...

Memoria Cache Asociativa por Conjuntos

- ❑ Una Cache 2-Asociativa (2-way) está compuesta de 2 Vías
- ❑ Cada vía puede verse como una cache directa.
- ❑ La línea X de todas las vías forma el Conjunto X
- ❑ 1 Bloque de MP se guarda en 1 Conjunto de MC, pero dentro del conjunto lo puede hacer en cualquiera de las vías.



- ❑ ¿Cómo seleccionamos la vía? ⇒ **ALGORITMO de REEMPLAZO**
- ❑ El número de conjuntos siempre ha de ser potencia de 2
- ❑ El número de vías puede ser cualquiera.
 - Podemos tener caches: 3-asociativa, 7-asociativa, 12-asociativa, ...

Ejemplo de comportamiento

Características de la Memoria Cache

- ❑ Asociativa por Conjuntos
- ❑ Tamaño de Cache: 128 bytes
- ❑ Tamaño de línea: 16 bytes
- ❑ Número de Conjuntos: 4
- ❑ Líneas por Conjunto: 2
- ❑ Algoritmo de reemplazo: LRU
- ❑ Política de escritura: WT + WnA
- ❑ Direcciones de 16 bits



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

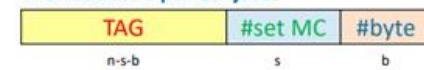
- La Cache tiene D líneas ($D=2^d$)
- Un bloque de MP se almacena en 1 línea de MC

Memoria Cache COMPLETAMENTE ASOCIATIVA

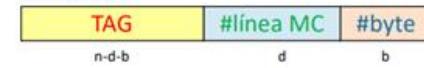
- La Cache tiene un solo conjunto de M líneas, M puede ser cualquier valor.
- Un Bloque de MP se almacena en cualquier línea de MC



MC Asociativa por Conjuntos



MC Directa



MC Completamente Asociativa



Algoritmos de Reemplazo

Reemplazo Aleatorio

Reemplazo FIFO (First In First Out)

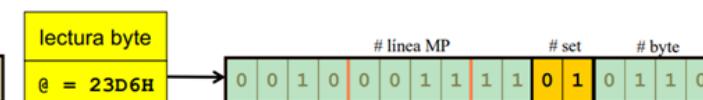
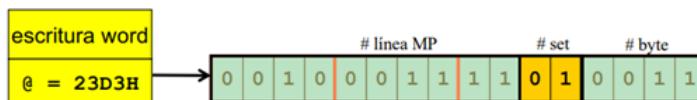
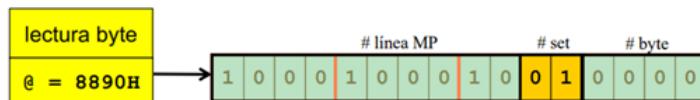
Reemplazo LRU (Least Recently Used)

- De entre todas las líneas candidatas a ser reemplazadas, se selecciona la que lleva más tiempo en la cache sin ser utilizada.
- Este algoritmo da buenos resultados. Teniendo en cuenta el comportamiento de los programas, parece la opción más lógica.
- Sin embargo, es muy costoso de implementar si el grado de asociatividad es alto. El coste de implementar este algoritmo es $n!$ (siendo n el grado de asociatividad).
- Normalmente se implementa un algoritmo PseudoLRU. Un algoritmo LRU ha de mantener información de en qué orden se ha accedido a todas las líneas de un conjunto. En un algoritmo pseudoLRU sólo se mantiene parte de esa información.

set 0	set 1	set 2	set 3
EC8 1	EC5 0	EC6 0	EC7 1
AB4 0	889 1	AB2 1	- 0

set 0	set 1	set 2	set 3
EC8 1	EC5 1	EC6 1	EC7 1
AB4 0	889 0	AB2 0	- 0

set 0	set 1	set 2	set 3
EC8 1	EC5 0	EC6 0	EC7 1
AB4 0	23D 1	ABA 1	- 0



# accesos	1	tasa fallos	1.0	#bytes leídos MP	16
# fallos	1	tasa aciertos	0.0	#bytes escritos MP	0

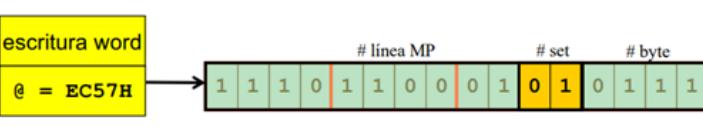
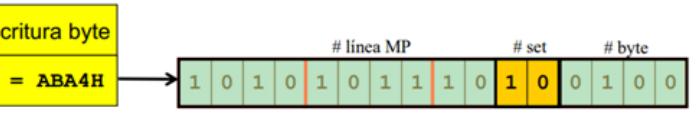
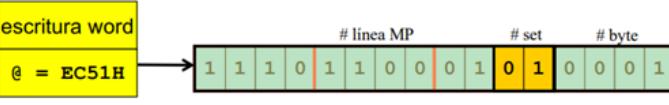
# accesos	4	tasa fallos	0.5	#bytes leídos MP	16
# fallos	2	tasa aciertos	0.5	#bytes escritos MP	5

# accesos	7	tasa fallos	0.71	#bytes leídos MP	48
# fallos	5	tasa aciertos	0.29	#bytes escritos MP	6

set 0	set 1	set 2	set 3
EC8 1	EC5 1	EC6 0	EC7 1
AB4 0	889 0	AB2 1	- 0

set 0	set 1	set 2	set 3
EC8 1	EC5 1	EC6 1	EC7 1
AB4 0	889 0	AB2 0	- 0

set 0	set 1	set 2	set 3
EC8 1	EC5 1	EC6 0	EC7 1
AB4 0	23D 0	ABA 1	- 0



# accesos	2	tasa fallos	0.5	#bytes leídos MP	16
# fallos	1	tasa aciertos	0.5	#bytes escritos MP	2

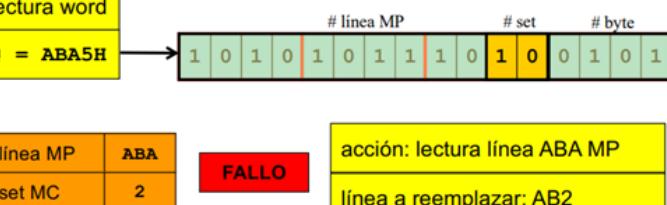
# accesos	5	tasa fallos	0.6	#bytes leídos MP	16
# fallos	3	tasa aciertos	0.4	#bytes escritos MP	6

# accesos	8	tasa fallos	0.62	#bytes leídos MP	48
# fallos	5	tasa aciertos	0.38	#bytes escritos MP	8

set 0	set 1	set 2	set 3
EC8 1	EC5 1	EC6 1	EC7 1
AB4 0	889 0	AB2 0	- 0

set 0	set 1	set 2	set 3
EC8 1	EC5 1	EC6 0	EC7 1
AB4 0	889 0	ABA 1	- 0

set 0	set 1	set 2	set 3
EC8 1	EC5 1	EC6 1	EC7 1
AB4 0	23D 0	ABA 0	- 0



# accesos	3	tasa fallos	0.33	#bytes leídos MP	16
# fallos	1	tasa aciertos	0.67	#bytes escritos MP	3

# accesos	6	tasa fallos	0.66	#bytes leídos MP	32
# fallos	4	tasa aciertos	0.34	#bytes escritos MP	6

# accesos	9	tasa fallos	0.55	#bytes leídos MP	48
# fallos	5	tasa aciertos	0.45	#bytes escritos MP	8

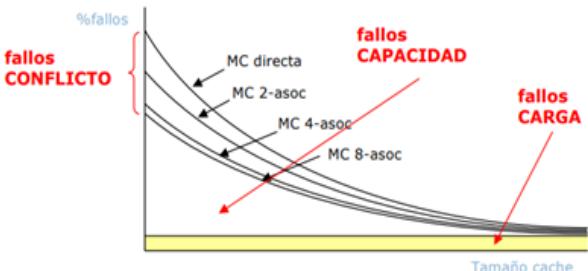
Tipos de Fallos

Los fallos de cache pueden dividirse en tres categorías:

- ❑ **CARGA** (compulsory). Se producen la primera vez que se accede a una posición de memoria.
- ❑ **CONFLICTO**. Se producen cuando varias líneas se mapean en el mismo lugar de la MC (sólo aplica en MC directas y asociativas por conjuntos, en una MC Completamente Asociativa no puede haber fallos por conflicto, por definición).
- ❑ **CAPACIDAD**. Todas las líneas que necesita un programa no caben en la Memoria Cache.

¿Cómo se calculan?

Ayuda a entenderlo



Ejemplo de tipos de fallos

```
int A[8], B[8];
int i, s=0;
for (i=0; i<8; i++)
    s = s + A[i] + B[i];
```

- Cache Directa
- Líneas de 8 bytes (2 words)
- 4 Líneas
- Procesador 32 bits

MC		
V	Etiquetas	Datos
0	1	A[0] A[1]
1	0	
2	0	
3	0	

MP #bloque		
	A:	B:
MISS CARGA →	A: [A[0]] 0	B: [B[0]] 4
	A: [A[1]] 1	B: [B[1]] 5
	A: [A[2]] 2	B: [B[2]] 6
	A: [A[3]] 3	B: [B[3]] 7
	A: [A[4]] 4	
	A: [A[5]] 5	
	A: [A[6]] 6	
	A: [A[7]] 7	
	B: [B[0]] 4	
	B: [B[1]] 5	
	B: [B[2]] 6	
	B: [B[3]] 7	
	B: [B[4]] 8	
	B: [B[5]] 9	
	B: [B[6]] 10	
	B: [B[7]] 11	

Ejemplo de tipos de fallos

```
int A[8], B[8];
int i, s=0;
for (i=0; i<8; i++)
    s = s + A[i] + B[i]; ;
```

- Cache Directa
- Líneas de 8 bytes (2 words)
- 4 Líneas
- Procesador 32 bits

MC		
V	Etiquetas	Datos
0	1	B[0] B[1]
1	0	
2	0	
3	0	

MP #bloque
A[0] 0
A[1] 1
A[2] 2
A[3] 3
A[4] 4
A[5] 5
A[6] 6
A[7] 7
B[0] 4
B[1] 5
B[2] 6
B[3] 7
B[4] 8
B[5] 9
B[6] 10
B[7] 11

Ejemplo de tipos de fallos

```
int A[8], B[8];
int i, s=0;
for (i=0; i<8; i++)
    s = s + A[i] + B[i]; ;
```

- Cache Directa
- Líneas de 8 bytes (2 words)
- 4 Líneas
- Procesador 32 bits

MC		
V	Etiquetas	Datos
0	1	A[0] A[1]
1	0	
2	0	
3	0	

MP #bloque
A[0] 0
A[1] 1
A[2] 2
A[3] 3
A[4] 4
A[5] 5
A[6] 6
A[7] 7
B[0] 4
B[1] 5
B[2] 6
B[3] 7
B[4] 8
B[5] 9
B[6] 10
B[7] 11

Ejemplo de tipos de fallos

```
int A[8], B[8];
int i, s=0;
for (i=0; i<8; i++)
    s = s + A[i] + B[i]; ;
```

- Cache Directa
- Líneas de 8 bytes (2 words)
- 4 Líneas
- Procesador 32 bits

MC		
V	Etiquetas	Datos
0	1	B[0] B[1]
1	0	
2	0	
3	0	

MP #bloque
A[0] 0
A[1] 1
A[2] 2
A[3] 3
A[4] 4
A[5] 5
A[6] 6
A[7] 7
B[0] 4
B[1] 5
B[2] 6
B[3] 7
B[4] 8
B[5] 9
B[6] 10
B[7] 11

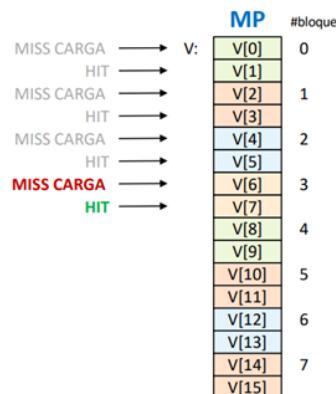
Ejemplo de tipos de fallos

```
int V[16], i, s=0;
for (i=0; i<16; i++)
    s = s + V[i];
for (i=0; i<16; i++)
    s = s + V[i]*3;
```

- Cache Completamente Asociativa
- 4 Líneas de 8 bytes (2 words)
- Procesador 32 bits

MC

V	Etiquetas	Datos	
0	1	V[0]	V[1]
1	1	V[2]	V[3]
2	1	V[4]	V[5]
3	0	V[6]	V[7]



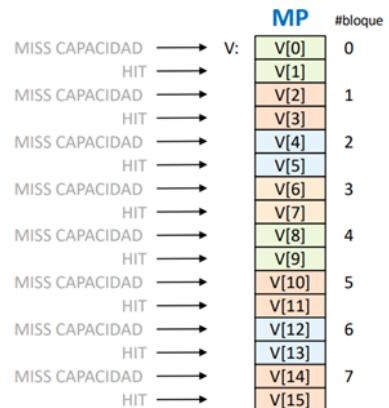
Ejemplo de tipos de fallos

```
int V[16], i, s=0;
for (i=0; i<16; i++)
    s = s + V[i];
for (i=0; i<16; i++)
    s = s + V[i]*3;
```

- Cache Completamente Asociativa
- 4 Líneas de 8 bytes (2 words)
- Procesador 32 bits

MC

V	Etiquetas	Datos	
1	1	V[8]	V[9]
2	1	V[10]	V[11]
3	1	V[12]	V[13]
4	1	V[14]	V[15]



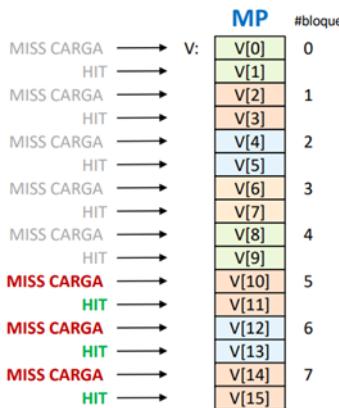
Ejemplo de tipos de fallos

```
int V[16], i, s=0;
for (i=0; i<16; i++)
    s = s + V[i];
for (i=0; i<16; i++)
    s = s + V[i]*3;
```

- Cache Completamente Asociativa
- 4 Líneas de 8 bytes (2 words)
- Procesador 32 bits

MC

V	Etiquetas	Datos	
1	1	V[8]	V[9]
2	1	V[10]	V[11]
3	1	V[12]	V[13]
4	1	V[14]	V[15]



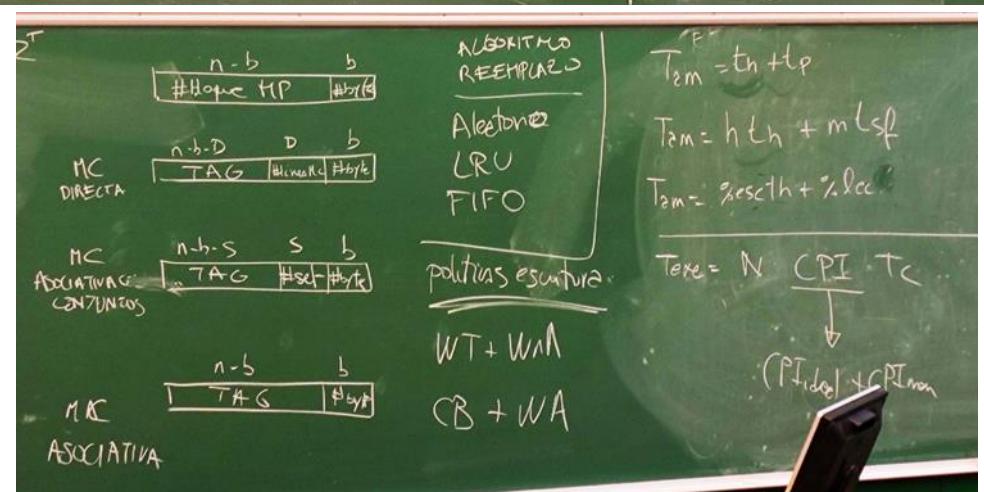
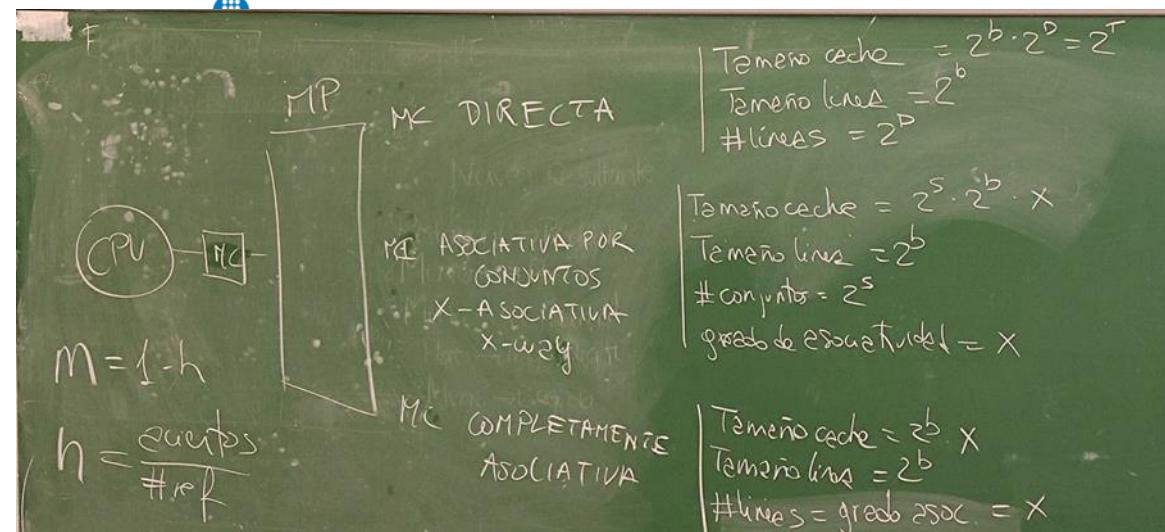
Ejemplo de tipos de fallos

```
int V[16], i, s=0;
for (i=0; i<16; i++)
    s = s + V[i];
for (i=0; i<16; i++)
    s = s + V[i]*3;
```

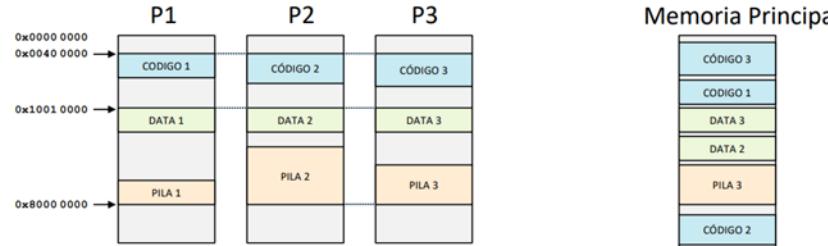
- Cache Completamente Asociativa
- 4 Líneas de 8 bytes (2 words)
- Procesador 32 bits

MC

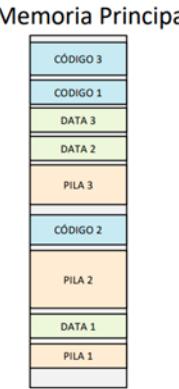
V	Etiquetas	Datos	
1	1	V[0]	V[1]
2	1	V[10]	V[11]
3	1	V[12]	V[13]
4	1	V[14]	V[15]



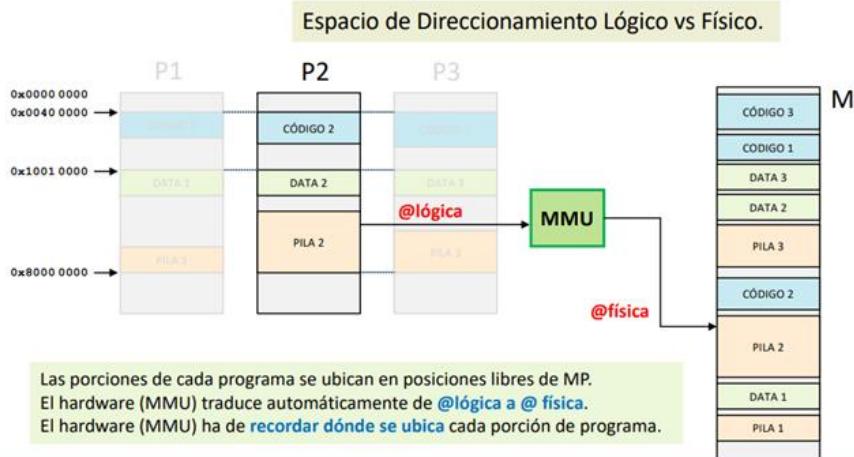
Necesidad de Reubicación



Para ejecutarlos, es necesario **REUBICARLOS**.
Hay que **modificar** todas las direcciones.

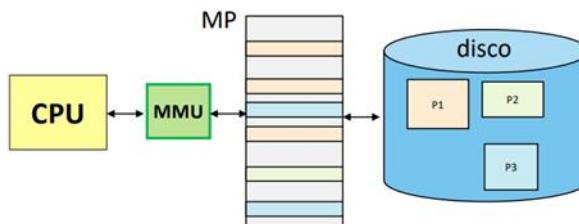


Traducción de Direcciones



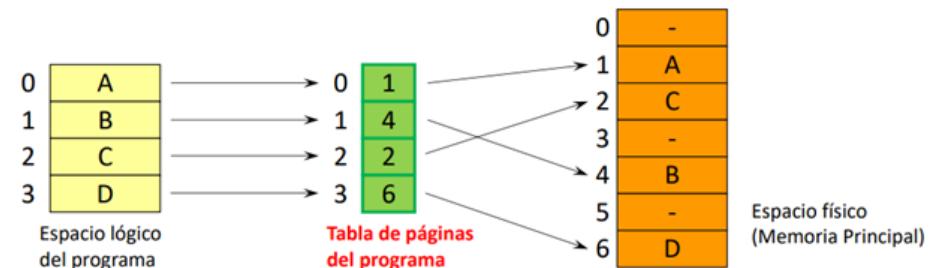
Programas parcialmente cargados en Memoria Principal

- Los programas residen íntegramente en el disco.
- Las porciones de programa accedidas recientemente se guardan en MP.
- La Memoria Virtual se encarga de mover datos entre disco y MP, explotando la localidad.
- Similar a MC -- MP, pero ahora la gestión la realiza el Sistema Operativo.



Memoria Virtual Paginada

- El espacio lógico se divide en bloques de tamaño fijo ⇒ **PÁGINAS**
- Un tamaño razonable de página sería entre 4 y 32 KB
- El espacio físico (MP) se divide en **marcos** de tamaño una página (**frames**, tramas).
- Los programas se trocean en páginas (están en disco)
- Una página puede colocarse en **CUALQUIER** marco de página de MP (correspondencia completamente asociativa)
- Las páginas se copian desde disco a MP **cuando son referenciadas**
- Hace falta una estructura de datos para saber qué hay en cada marco de página ⇒ **TABLA DE PÁGINAS**



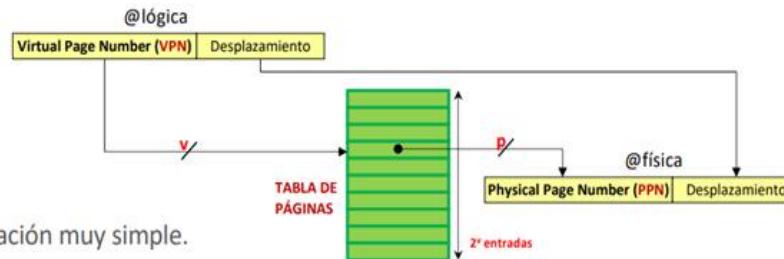
Subdivisión de la dirección

- Dada una dirección lógica A, ¿a qué página (**VPN**, Virtual Page Number) pertenece?
 - Por ejemplo, con direcciones de 32 bits, A=0x10010004 y el tamaño de página es 4 KB (2^{12} bytes)
- | Virtual Page Number (VPN) | Desplazamiento |
|---------------------------|----------------|
| 20 bits | 12 bits |
- A=0x10010004 = **0001 0000 0000 0001 0000** | **0000 0000 0100** , VPN = 0x10010, off = 0x004
- Dada una dirección física AF, ¿a qué frame (**PPN**, Physical Page Number) pertenece?
 - Por ejemplo, con direcciones de 28 bits, AF=0x0301004 y el tamaño de página es 4 KB (2^{12} bytes)

Physical Page Number (PPN)	Desplazamiento
16 bits	12 bits

○ AF=0x0301004 = **0000 0011 0000 0000 0001** | **0000 0000 0100** , PPN = 0x0301, off = 0x004

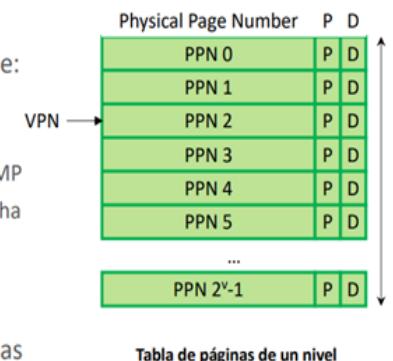
Paginación: implementación hardware de la traducción



- ❑ Reubicación muy simple.
- ❑ Permite protección de páginas.
- ❑ Cálculo rápido de la dirección (no hay operaciones aritméticas).
- ❑ Fragmentación.
- ❑ Página físicas y virtuales tienen el mismo tamaño.
- ❑ VPN y PPN pueden tener longitud diferente.
- ❑ En la mayoría de sistemas se cumple: $2^v > 2^p$.

Implementación de la Tabla de Páginas

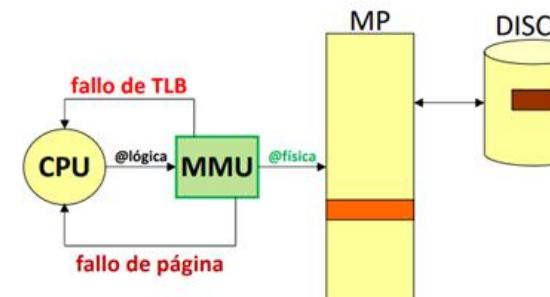
- ❑ Cada proceso tiene su propia Tabla de Páginas.
- ❑ El contenido de la TP lo gestiona el SO.
- ❑ La tabla de páginas se indexa con el VPN
- ❑ Cada entrada de la tabla de páginas (PTE) contiene:
 - Bit de Presencia P. Vale 1 si la entrada es válida y la página está en MP.
 - PPN. Si P=1, indica el frame dónde está la página en MP
 - Bit de Modificación D. Si D=1, significa que la página ha sido modificada (por un store).
 - Otros bits con permisos diversos: READ_ONLY, READ/WRITE, EXEC, bits para gestionar LRU
- ❑ Si la página no está en memoria, la tabla de páginas contiene su ubicación en disco



Memoria virtual

La Memoria Virtual permite:

- ❑ Ejecutar un programa con espacio lógico > espacio físico
- ❑ Ejecutar un programa parcialmente cargado en Memoria
- ❑ Proteger el espacio de direcciones de los programas de ser accedido por otros programas



Paginación bajo demanda

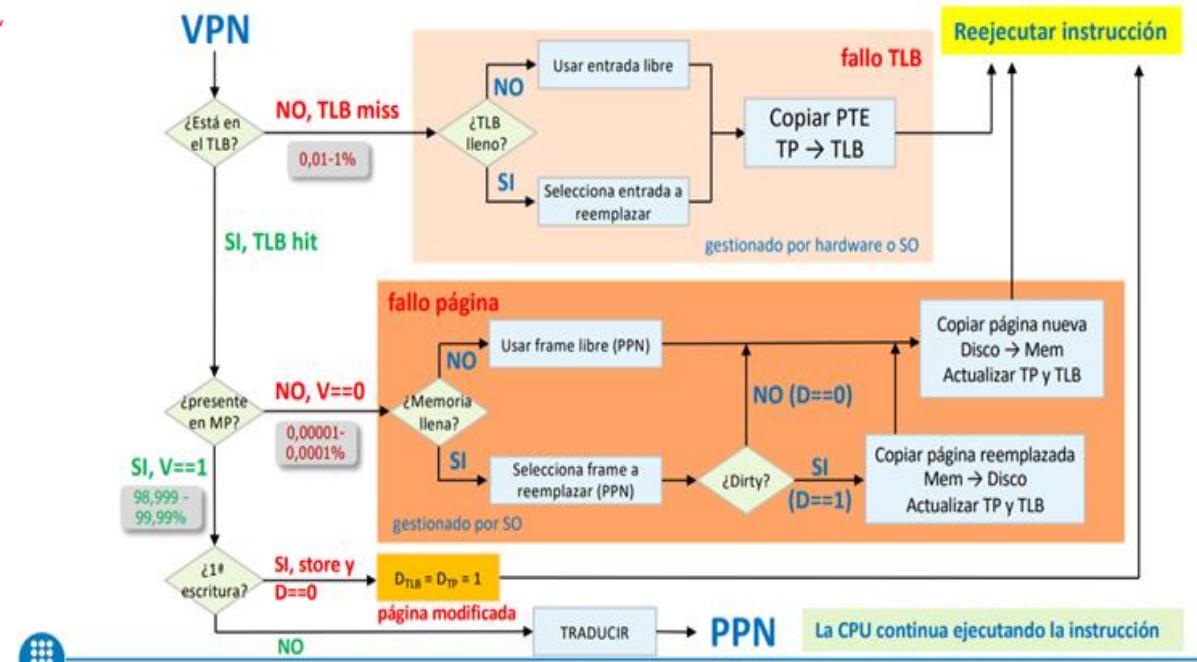
La TABLA de PÁGINAS tiene un bit de presencia

```
if (fallo de página) {
    Reemplazar página(D): MP → DISCO
    Cargar la página solicitada: DISCO → MP
}
```

¿Qué se hace con las escrituras?

- COPY BACK + WRITE ALLOCATE

Fallos de TLB y Fallos de Página

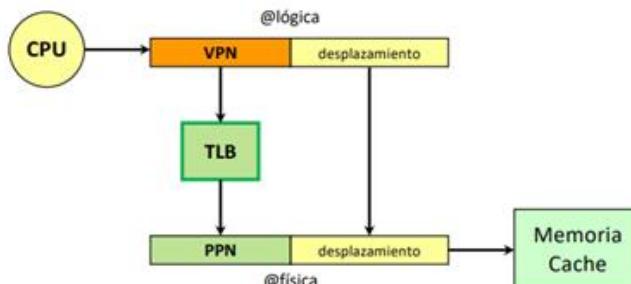


Protección y Memoria virtual

- ❑ Cada página física está asignada a un único proceso y no aparece en las tablas de páginas de otros procesos.
 - El mecanismo de traducción hace imposible que un proceso acceda a páginas de otro.
- ❑ Pero, ¿qué pasaría si un proceso intenta modificar su tabla de páginas?
 - La tabla de páginas se guardan en el espacio de direcciones reservado al SO.
 - En MIPS, son las direcciones con el bit 31 igual a 1.
- ❑ ¿Cómo accede el SO a ese espacio de direcciones?
 - Los procesadores tienen 2 modos de funcionamiento: usuario y sistema
 - Sólo cuando está funcionando en modo sistema tiene acceso al TLB y tabla de páginas.
- ❑ Protección contra escritura. Resulta conveniente prohibir la escritura en algunas páginas.
 - Por ejemplo, páginas de código, no se permite código automodificable
 - Se añade un bit de permiso de escritura (bit E) en la TP y TLB
 - Si un proceso intenta escribir en una página con el bit E=0, se produce una excepción y el SO aborta el programa.

Juntando Memoria Virtual y Memoria Cache

- ❑ Traducción **antes** de acceder a Memoria Cache



- ❑ Memoria Cache de direcciones físicas
- ❑ Lento: un acceso a memoria necesita un acceso TLB + acceso MC
- ❑ En este último caso, ¿cuándo se efectúa la traducción de direcciones lógicas a físicas, antes o después de acceder a la Memoria Cache?
- ❑ Tres posibilidades:
 - Traducción **antes** de acceder a Memoria Cache
 - Traducción **después** de acceder a Memoria Cache
 - Traducción y acceso a Memoria Cache **simultáneos**

Tipología de excepciones e interrupciones

- ❑ **EXCEPCIONES** (interrupciones síncronas), provocadas por la ejecución de una instrucción.
 - Excepciones por violación de una restricción.
 - ✓ Intento de ejecutar una instrucción privilegiada
 - ✓ Intento de ejecutar una instrucción inválida
 - ✓ Overflow aritmético, división por cero, acceso a una dirección no alineada
 - Excepciones relacionadas con la Memoria Virtual
 - ✓ Fallo de página, Fallo de TLB, violación de protecciones
 - Causadas expresamente por el programador
 - ✓ Llamada al Sistema Operativo (invocadas con un syscall)
 - ✓ Relacionadas con debug: traps, breakpoint, ...

- ❑ **INTERRUPCIONES** (interrupciones asíncronas), provocadas por un evento externo.

- Peticiones de servicio de un dispositivo de E/S.
 - ✓ Alguien ha pulsado una tecla
 - ✓ Ha acabado una operación de E/S programada
 - ✓ Ha llegado un paquete de datos por la red
- Temporizador
- Errores Hardware

Rutina de Servicio de Excepciones (RSE)

- ❑ **RSE** (exception handler)
 - Código del Sistema Operativo que se encarga de gestionar las excepciones e interrupciones.
- ❑ Al detectar una excepción o interrupción
 1. Se finaliza o cancela la instrucción en curso
 2. Se salta a la RSE
 - ✓ Se escribe en el PC la dirección inicial de la RSE
 3. Se ejecutan las instrucciones de la RSE
 4. Se aborta o se reanuda la ejecución del programa interrumpido

Cronología de una excepción

1. Detección de la excepción o interrupción

- Decodificando la instrucción en curso: código operación ilegal, instrucción privilegiada, syscall
- En la MMU: fallo de página, fallo de TLB
- En la ALU: dirección no alineada, overflow, división por cero
- Un dispositivo de E/S: ha activado una señal externa INT (interrupción)

2. La Unidad de Control de la CPU decide

- Si es una excepción, aborta la instrucción en curso. Inhibe la escritura en registros, memoria, etc.
- En caso de interrupción, espera a que acabe la instrucción en curso.

3. Salta a la RSE (Rutina de Servicio a Excepciones o Exception Handler)

- Es el código de SO que hace el tratamiento de excepciones e interrupciones.
- ¿Cómo se salta? ⇒ Poniendo en el PC la dirección inicial de la RSE
- Hay que guardar la @ de la instrucción interrumpida

4. Al acabar la RSE

- Puede retornar al programa o abortarlo, dependiendo del tipo de excepción

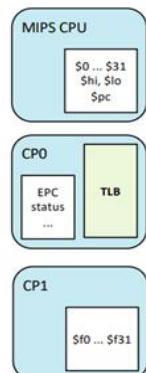
Tipos de Excepciones e Interrupciones

Excepciones por la violación de una restricción

- ◻ Excepciones producidas al ejecutar una instrucción que viola una restricción:
 - Overflow aritmético
 - Acceso a Memoria NO alineado
 - División por cero
 - Instrucción reservada
- ◻ La instrucción se interrumpe sin escribir en registros o memoria
- ◻ Normalmente es un error y la RSE aborta la ejecución del programa
- ◻ En algunos casos la RSE puede resolver el problema
 - Por ejemplo, si el SO emula por software una instrucción, al acabar la emulación, se da por ejecutada la instrucción y se salta a la siguiente instrucción a la que ha provocado la excepción

El coprocesador de Sistema CP0

- ◻ Controla el tratamiento de excepciones y la traducción con el TLB
- ◻ Añade instrucciones adicionales
 - **Instrucciones Privilegiadas**, sólo se pueden ejecutar en modo sistema
- ◻ Tiene un banco de registros específicos, que sólo son accesibles por las instrucciones privilegiadas.
 - Por ejemplo, para copiarlos de CP0 a CPU o viceversa:
 - ✓ `mfc0 rt, rd_c0 # rt ← rd_c0`
 - ✓ `mtc0 rt, rd_c0 # rd_c0 ← rt`



Tipos de Excepciones e Interrupciones

Excepciones relacionadas con la Memoria Virtual

- ◻ Se producen durante la traducción de la @lógica a @física
- ◻ Fallo de TLB
 - En MIPS, se resuelve por software (lo normal es por hardware)
 - La RSE, copia la entrada de la TP en una entrada libre del TLB
- ◻ Fallo de página
 - La RSE copia la página de disco a un frame libre de la MP, actualiza TP y TLB
 - Si es necesario se reescribe la página (sucia) reemplazada en disco
- ◻ En ambos casos, al acabar la RSE se salta a la instrucción que ha provocado la excepción
 - Se ha de ejecutar de nuevo.
- ◻ Violación de protección
 - La RSE aborta la ejecución del programa

Excepciones de trap y llamadas al Sistema Operativo

- ◻ Las excepciones de trap se usan al depurar un programa
 - Ejecución instrucción a instrucción, entre medias entra el debugger
 - Uso de breakpoints
- ◻ Se utiliza una instrucción especial del ISA para acceder a un servicio del Sistema Operativo.
 - Instrucción **syscall** en MIPS
 - Es similar a una llamada a una subrutina
 - ✓ Con paso de parámetros y retorno de resultados
 - Al finalizar la RSE, continúa ejecutando el programa a partir de la instrucción siguiente a la que produjo la excepción (**syscall**)

Interrupciones

- ◻ Causadas por la activación de la señal INT (petición de interrupción) desde algún dispositivo de Entrada/Salida
- ◻ En este caso, la petición se atiende al final de la ejecución de la instrucción en curso.
 - La petición queda registrada en un registro de peticiones pendientes
 - Cuando se atiende, la RSE puede transferir datos entre la CPU y el dispositivo
 - Las peticiones pueden necesitar mucho tiempo, el SO puede planificar otro proceso mientras se termina la operación de E/S.
- ◻ Al acabar la RSE
 - el programa continúa la ejecución a partir de la siguiente instrucción

El coprocesador de Sistema CP0

EPC (\$14)

Dirección de la instrucción interrumpida

Registro EPC (Exception Program Counter, \$14)

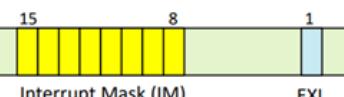
- Registro de Lectura / Escritura
- Antes de invocar la RSE, el procesador guarda el valor actual del PC en EPC
- Es la **dirección de retorno** de la RSE
- ❑ ¿Dónde apunta el PC que se guarda en EPC?
 - En una excepción, teniendo en cuenta que la instrucción en curso se aborta:
 - ✓ ⇒ El PC no se incrementa, guardamos la dirección de la instrucción en ejecución.
 - En una interrupción, teniendo en cuenta que la instrucción en curso se acaba:
 - ✓ ⇒ El PC se incrementa, y guardamos la dirección de la siguiente instrucción

¿Porqué no se puede guardar la dirección de retorno en \$ra?



El coprocesador de Sistema CP0

Status (\$12)



Registro Status (\$12)

❑ Bit EXL (Exception Level), tiene 2 funciones:

- Modo Usuario / Modo Sistema
 - ✓ Si EXL = 0, el procesador está en modo usuario
 - ✓ Si EXL = 1, el procesador está en modo sistema, tiene acceso al espacio de direcciones de SO y puede ejecutar instrucciones privilegiadas.
- Si EXL=1, **todas las interrupciones están inhibidas** y se ignoran (pero no las excepciones).

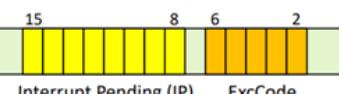
❑ 8 bits de IM (Interrupt Mask)

- 1 bit para cada tipo de interrupción
- Si el bit vale 1, las interrupciones de ese tipo están habilitadas.



El coprocesador de Sistema CP0

Cause (\$13)



Registro Cause (\$13), sólo lectura

❑ 5 bits de ExcCode, codifica la causa de la excepción

❑ 8 bits de IP (Interrupt Pending)

- Un bit para cada tipo de interrupción
- Cuando se recibe una petición de interrupción, se activa el bit correspondiente
- El bit permanece a 1 mientras el dispositivo mantenga la señal de petición activada.

Acciones del Hardware en una Excepción

1. Cuando el hardware detecta una ...
 - ... **excepción**, la instrucción en curso se aborta, no escribe en registros o memoria
 - ✓ No se incrementa el PC
 - ... **interrupción**, finaliza la instrucción en curso
 - ✓ Se comprueban las peticiones pendientes (IPi=1) y habilitados (IMi=1)
 - ✓ Si hay más de una, se selecciona la más prioritaria
2. Guarda el PC en el **registro EPC**
3. Escribe la causa en el campo ExcCode del **registro Cause**
4. Pone a 1 el **bit EXL** (modo sistema, interrupciones inhibidas)
5. Escribe en el **PC** la dirección de la Rutina de Servicio (en MIPS, hay 2 diferentes):
 - **RSE** (genérica), PC = 0x800000180
 - **TLBmiss** (sólo fallos de TLB), PC = 0x80000000

Acciones del Software en una Excepción

1. La RSE ha de preservar el estado del programa interrumpido
 - Salva en la pila **TODOS LOS REGISTROS** (\$1 - \$31) excepto \$k0, \$k1
 - No es necesario salvar los registros de coma flotante (\$f0 - \$f31)
 - ✓ Sólo se salvan si se invoca una rutina en donde se modifiquen
2. Identificar la causa (**ExcCode** en el registro Cause)
3. Saltar a una rutina específica que puede ...
 - ... abortar el programa
 - ... bloquearlo temporalmente, dando paso a otros procesos
 - ... o solucionar el problema y continuar la ejecución del proceso normalmente.
4. En caso de **syscall** hay que sumar 4 a EPC, para que apunte a la siguiente instrucción
5. Restaurar todos los registros salvados en la pila
6. Retornar, usando la instrucción **eret** (exception return)
 - Pone EXL=0 (modo usuario, interrupciones permitidas)
 - Copia EPC en el PC (salta a la dirección de retorno)

Un ejemplo sencillo de RSE (1)

```
.ktext 0x80000180
RSE:
# Salvar Registros
addiu $sp,$sp,-128
sw $1,0($sp)
sw $2,4($sp)
sw $3,8($sp)
...
sw $31,128($sp)

# Pasar parámetros Cause y EPC
...
```

Salvar TODOS los registros, incluyendo \$hi y \$lo, pero NO \$k0, \$k1, \$sp y \$0.

¿Porqué no salvamos \$0?

Un ejemplo sencillo de RSE (2)

```
.ktext 0x80000180
RSE:
# Salvar Registros

# Pasar parámetros Cause y EPC
mfc0 $a0,$13
mfc0 $a1,$14
jal handler_dispatcher
mtc0 $v0,$14

# Restaurar Registros
...
```

Pasamos Registro Cause y EPC como parámetros en \$a0, \$a1

La rutina handler_dispatcher invocará al handler que corresponga.

Devuelve como resultado la dirección de retorno que se copia en EPC

Un ejemplo sencillo de RSE (3)

```
.ktext 0x80000180
RSE:
# Salvar Registros
# Pasar parámetros Cause y EPC

# Restaurar Registros
sw $1,0($sp)
sw $2,4($sp)
...
sw $31,128($sp)
addiu $sp,$sp,128

# Retorno
```

En un sistema real, estaríamos usando la pila del SO y no la de usuario.

Un ejemplo sencillo de RSE (4)

```
.ktext 0x80000180
RSE:
# Salvar Registros
# Pasar parámetros Cause y EPC
# Restaurar Registros

# Retorno
eret
```

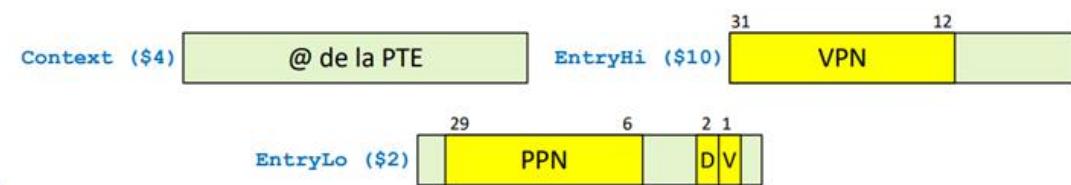
Retornamos al programa de usuario.
Ponemos EXL=0 y PC = EPC

Fallo de TLB en MIPS

- ❑ En MIPS, los fallos de TLB causan una excepción
 - Las trata una rutina de Sistema Operativo (software)
- ❑ Son bastante frecuentes ⇒ conviene tratarlas eficientemente
 - RSE genérica (@ = 0x80000180), demasiado compleja y lenta
 - Se tratan con la rutina TLBmiss (@ = 0x80000000)
 - ✓ Muy rápida (13 ciclos)
 - ✓ No salva registros en la pila, sólo usa \$k1 (reservado por el SO)
- ❑ Registros del CPO para gestionar el TLB

Fallo de TLB en MIPS

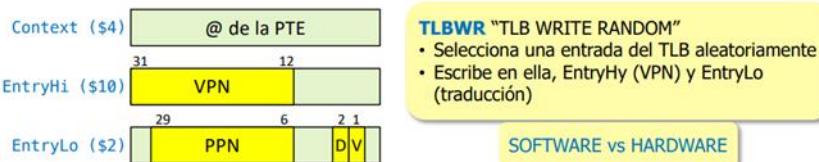
- ❑ Cuando el hardware detecta un fallo de TLB ...
 - ... copia el PC en EPC, y pone el bit EXL=1
 - ... escribe en ExcCode el código TLBL (fetch o load) o TLBS (store)
- ❑ Además ...
 - ... escribe en el PC la dirección de la rutina TLBmiss (0x80000000)
 - ... escribe la dirección de la entrada de la TP (PTE) en el **registro Context**
 - ... escribe los 20 bits del VPN en el **registro EntryHi**



Fallo de TLB en MIPS

- ❑ Ejemplo de rutina TLBmiss

```
.ktext 0x80000180
TLBmiss:
    mfc0 $k1,$4    # copia Context (@PTE) en $k1
    lw $k1,0($k1)  # lee la PTE (traducción) en $k1
    mtc0 $k1,$2    # copia la traducción a EntryLo
    tlbwr          # escribe EtryHi|EntryLo en el TLB
    eret           # Retorna el programa
```



El TLB en MIPS

Función del bit V del TLB

- ❑ Es una copia del bit de presencia (P) de la TP
- ❑ Indica si la página está en MP o sólo en disco
- ❑ No interviene para determinar si es fallo de TLB o no
 - El fallo de TLB sólo depende de si encuentra el VPN o no
- ❑ El bit V interviene cuando reemplazamos una entrada del TLB
 - El algoritmo da preferencia a ocupar entradas con V=0

Función del bit D del TLB

- ❑ En un **fallo de página**, se trae la página de disco a MP.
 - Además, se inicializa el bit D en la TP (0 si es un load, 1 si es un store)
 - El bit D indica si la página en MP se ha modificado respecto al disco
 - ✓ La escritura de páginas de MP usa la política de escritura retardada
- ❑ En un **fallo de TLB**, se copia la PTE en el TLB, incluyendo el bit D.
- ❑ Cuando un store acierta en el TLB
 - Si D=0 (primer store en la página) se activa D=1
 - También se ha de poner D=1 en la TP
 - ✓ La escritura del bit D del TLB tiene política de escritura inmediata
 - Para poner D=1 en la TP se genera la excepción de "página modificada"
 - ✓ El SO comprueba los permisos de escritura y actualiza el bit D

La Excepción de Llamada al Sistema Operativo

❑ El Sistema Operativo

- Proporciona acceso seguro y eficiente a los recursos compartidos
 - ✓ Memoria Física
 - ✓ Dispositivos de Entrada/Salida
 - ✓ Tiempo de CPU (ejecución concurrente de múltiples procesos)
- ❑ La gestión de los recursos se realiza en **Modo Sistema** (EXL=1)
 - Permite acceder al espacio de memoria (código y datos) reservado al SO
 - ✓ Espacio reservado al SO: **0x80000000 – 0xFFFFFFFF**
 - ✓ Si el usuario accede a este espacio ⇒ excepción (ExcCode = AdEL, AdES)
 - Permite ejecutar instrucciones privilegiadas
 - ✓ Como las que operan en el CPO
 - ✓ Si el usuario ejecuta una instrucción privilegiada ⇒ excepción (ExcCode = CpU)

La Excepción de Llamada al Sistema Operativo

- ❑ El acceso a los servicios del SO se hace con una excepción
 - En MIPS, con la instrucción **syscall**
 - La Excepción (ExcCode = Sys) cambia a modo sistema (EXL = 1) y salta a la RSE (que es parte del SO)
- ❑ Funciona como una llamada a una subrutina
 - Se pone en \$v0 el **código del servicio** solicitado
 - ✓ El SO tiene una subrutina para cada servicio
 - Los **parámetros**, en \$a0-\$a3 (\$f12,\$f14 los de CF)
 - Se hace la llamada con **syscall** (no con jal)
 - El **valor de retorno** (si lo hay) se devuelve en \$v0 (o \$f0 en CF)
- ❑ A diferencia de las subrutinas
 - La excepción salta a un punto de entrada único del SO (la RSE)
 - La excepción activa el modo sistema e inhibe interrupciones.

```
.data
cadena: .asciiz "Esto es una frase\n"
.text
...
li $v0,4      #llamada 4: print_string(char *)
la $a0,cadena
syscall
li $v0,10     #llamada 10: exit()
syscall
```

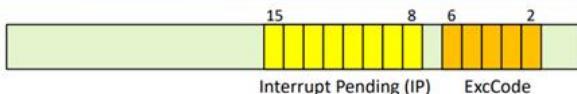
Las Interrupciones de Entrada/Salida

- ❑ Los dispositivos de Entrada/Salida sólo son accesibles por el SO
 - Los programas de usuario puede acceder solicitándolo al SO, por medio de una llamada al sistema ([syscall](#))
- ❑ Las operaciones de E/S tienen latencias enormes (si las comparamos con la velocidad de la CPU). Por ello la comunicación entre CPU y dispositivos de E/S requiere [sincronización](#).
- ❑ Dos tipos de [Sincronización](#)
 - **Por Encuesta**, el programa espera que el dispositivo esté listo, consultando repetidamente su estado (espera activa)
 - **Por Interrupciones**, el SO ejecuta otras cosas mientras que el dispositivo esté ocupado (permite optimizar el uso de la CPU)
 - ✓ Cuando el dispositivo está preparado, avisa a la CPU, activando una señal de [petición de interrupción \(INT\)](#)

```
do {  
    estado = ConsultarEstadoDisp();  
} while (estado != READY);
```

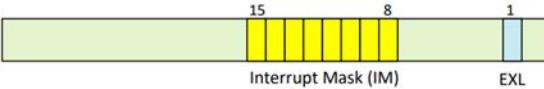
Las interrupciones de E/S en MIPS

Cause (\$13)



- ❑ Cuando un dispositivo finaliza una operación de E/S o requiere atención
 - Sigue la petición, activando la señal INT (asíncrono)
 - La instrucción en curso no se interrumpe
 - La petición queda pendiente y anotada en el bit IP, asociado al dispositivo de E/S
- ❑ Cuando acaba la instrucción en curso se incrementa el PC
 - El procesador comprueba si hay peticiones pendientes en el campo IP
 - Si hay alguna bit IP_i = 1, se genera una excepción (ExcCode = Int)
- ❑ Cuando la RSE hace el tratamiento de la interrupción
 - Lo notifica al dispositivo (señal INTA)
 - El dispositivo desactiva la petición (señal INT), el bit IP_i se pone a 0

Status (\$12)



- ❑ Cuando se invoca la RSE
 - Se pone el bit EXL=1, la rutina se ejecutará en modo sistema
 - Con EXL=1, **todas las interrupciones están deshabilitadas** y se ignoran.
- ❑ El SO puede deshabilitar las interrupciones de un dispositivo (i)
 - Poniendo a 0 el bit correspondiente del campo IM en el registro Status (bit IM_i)
- ❑ Si el dispositivo (i) solicita una interrupción, sólo se produce una excepción
 - Si IM_i = 1 y EXL = 0
- ❑ Identificación del dispositivo por software
 - La RSE Comprueba los campos IP e IM para determinar que dispositivo atiende
 - El orden en que se comprueba establece el orden de prioridades

Preguntas TEST ¿cierto o falso? (exámenes 2016-2019)

1. Si el bit EXL vale 1, las interrupciones serán ignoradas. 1- Cierto
2. La excepción por acceso no alineado a memoria puede ser inhibida a través del campo Interrupt Mask. 2- Falso
3. Una excepción no puede ser atendida hasta que la instrucción en curso haya finalizado. 3- Cierto
4. Un acceso a memoria nunca cambiará el estado de la tabla de páginas si se produce un acierto al TLB en una entrada con el bit de validez a 1. 4- Falso
5. La rutina RSE de tratamiento de excepciones de MIPS sigue las reglas del ABI que se establecen para programar subrutinas 5- Falso
6. En el MIPS se detecta que un acceso a memoria provoca un fallo de página consultando el bit V en el TLB. 6- Cierto
7. Si el acceso a datos de una instrucción produce un acierto en el TLB, pero el bit V vale 0, entonces la instrucción causará una excepción por fallo de página. 7- Cierto
8. Una misma instrucción puede causar durante su ejecución 2 fallos de página. 8- Cierto
9. Un fallo de TLB no implica que se produzca un fallo de página. 9- Cierto
10. Cuando se produzca un fallo de página, habrá uno o dos accesos a disco dependiendo del valor del Dirty Bit de la página a reemplazar. 10- Cierto
11. Al inicio de la rutina genérica de servicio a excepciones de MIPS (RSE), ésta sólo ha de salvar aquellos registros seguros que se modifique durante la ejecución de la RSE. 11- Falso
12. Un programa en modo usuario puede copiar un registro cualquiera de la CPU en el coprocesador CPO utilizando la instrucción mtc0. 12- Falso
13. En el MIPS, el campo IM (Interrupt Mask) del registro Status usado en la gestión de interrupciones, sirve para indicar las peticiones de interrupción que no han de ser atendidas cuando se acabe la ejecución de la instrucción actual. 13- Cierto
14. La primera vez que se escribe en una página se genera una excepción, el tratamiento de la cual consiste en escribir la página modificada en disco. 14- Falso
15. Cuando buscamos la traducción de un número de página (VPN) en el TLB, para que se produzca un acierto, hay que encontrar la entrada con el mismo VPN y el bit de presencia V=1. 15- Falso
16. Las escrituras del bit de Dirty del TLB siguen una política de escritura inmediata. 16- Cierto
17. En MIPS, una instrucción **lw** nunca puede producir una excepción por acceso no alineado de la dirección 17- Cierto
18. Los fallos de TLB en MIPS provocan una excepción y se ejecuta la RSE genérica. 18- Falso
19. En MIPS, cuando se acaba de ejecutar la RSE, se vuelve a la instrucción siguiente a la que ha provocado la excepción o la interrupción. 19- Falso
20. La sincronización de los dispositivos de Entrada/Salida en un computador, se gestiona exclusivamente por interrupciones. 20- Falso
21. En MIPS, la traducción de direcciones virtuales a físicas al ejecutar una instrucción **lw** puede producir hasta 4 excepciones. 21- Cierto