

Grupo ARCOS

uc3m | Universidad **Carlos III** de Madrid

Tema 3 (II)

Fundamentos de la programación en ensamblador

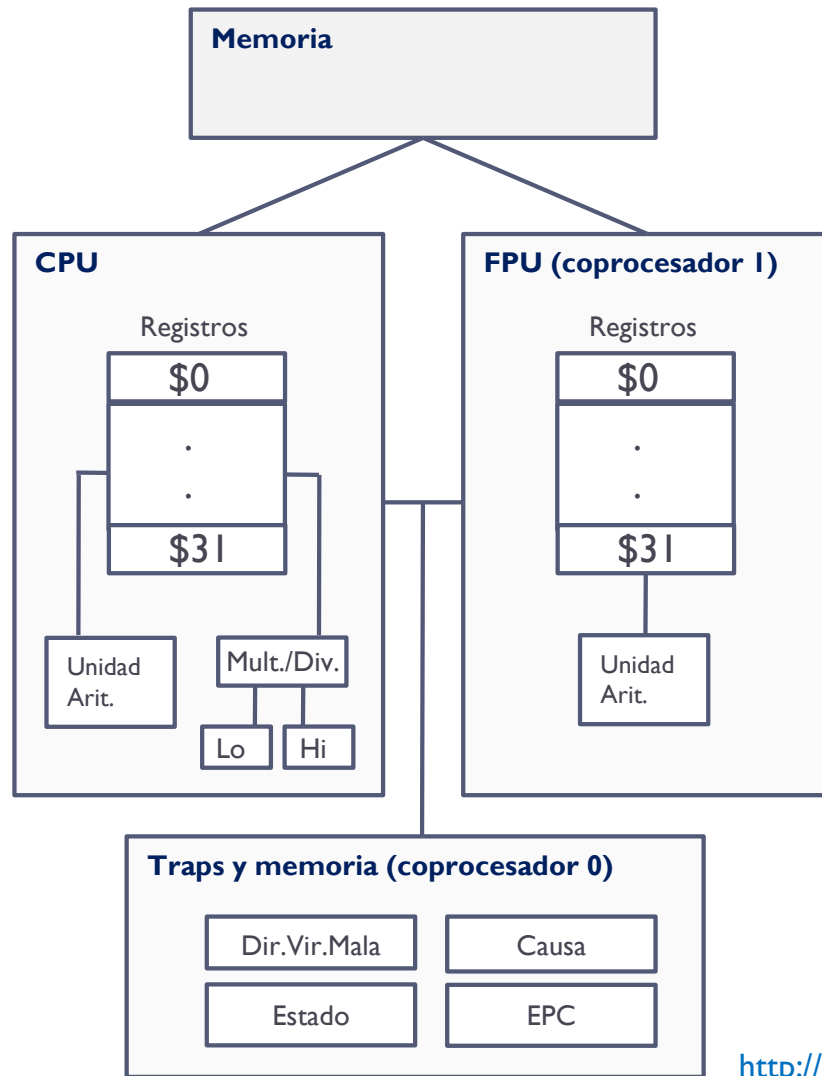
Estructura de Computadores
Grado en Ingeniería Informática



Contenidos

- ▶ Fundamentos básicos de la programación en ensamblador
- ▶ Ensamblador del MIPS 32, modelo de memoria y representación de datos
- ▶ Formato de las instrucciones y modos de direccionamiento
- ▶ Llamadas a procedimientos y uso de la pila

Arquitectura del MIPS 32



► MIPS 32

- Procesador de 32 bits
- Tipo RISC
- CPU + coprocesadores auxiliares

► Coprocesador 0

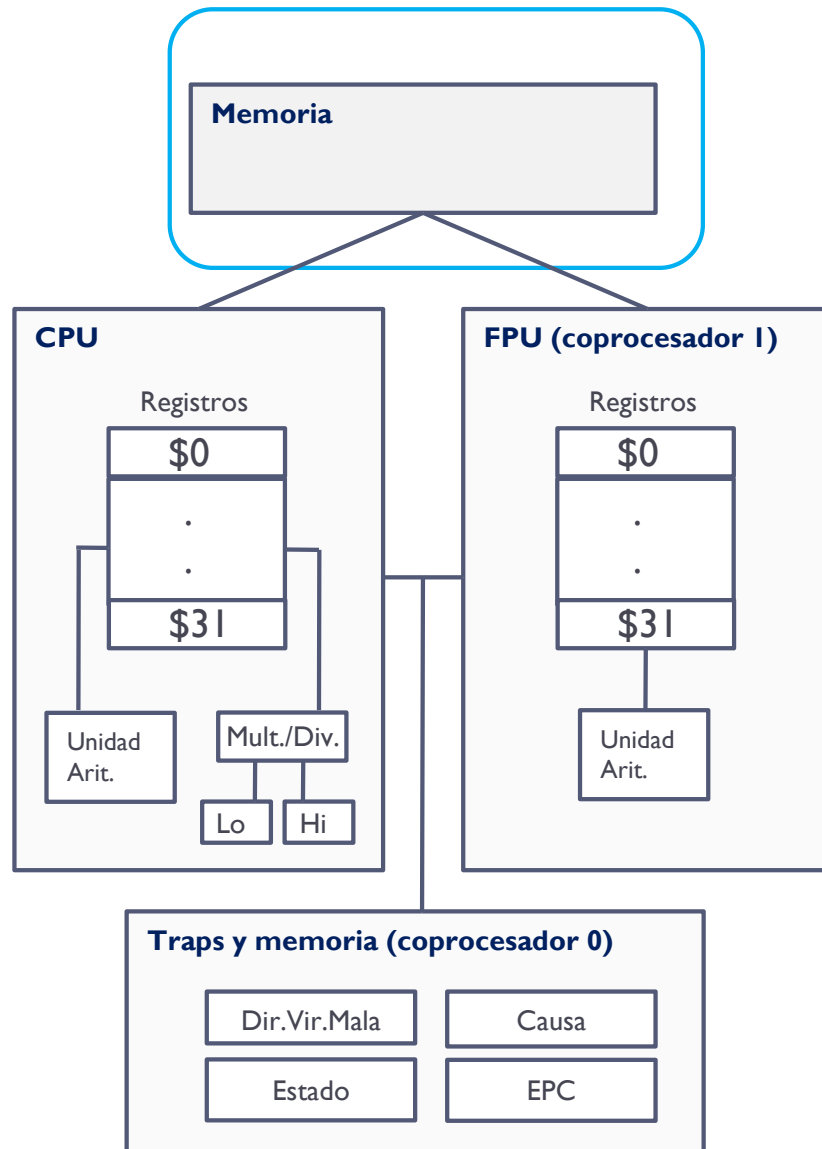
- excepciones, interrupciones y sistema de memoria virtual

► Coprocesador 1

- FPU (Unidad de Punto Flotante)

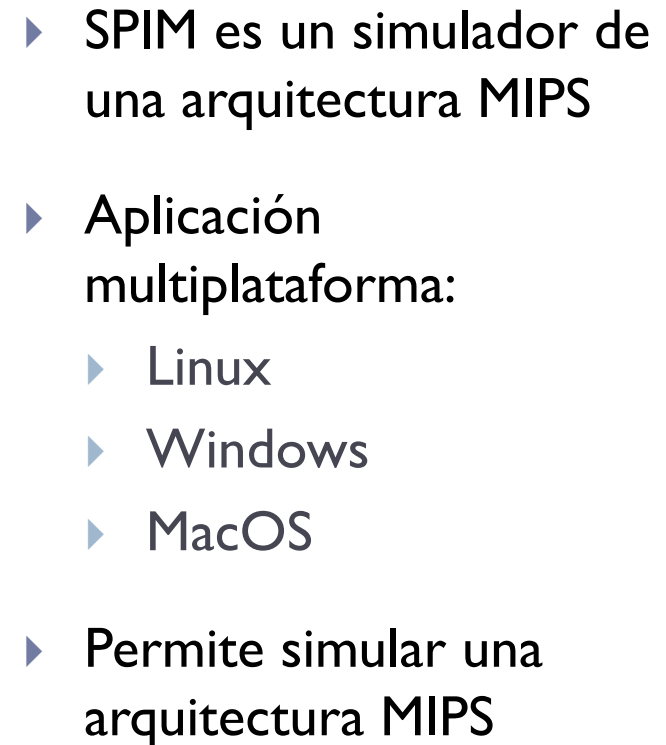
[http://es.wikipedia.org/wiki/MIPS_\(procesador\)](http://es.wikipedia.org/wiki/MIPS_(procesador))

Arquitectura del MIPS 32



- Direcciones de memoria de 2 bits
- 4 GB direccionables

<http://spimsimulator.sourceforge.net>



Ejercicio

- ▶ Instalar el simulador SPIM que se va a utilizar en las prácticas.
 - ▶ <http://spimsimulator.sourceforge.net>
- ▶ Probar en el simulador pequeños programas en ensamblador

Programa almacenado en QtSpim

memoria

```
PC = 0
EPC = 0
Cause = 0
BadVAddr = 0
Status = 3000fff0
HI = 0
LO = 0
R0 [r0] = 0
R1 [at] = 0
R2 [v0] = 0
R3 [v1] = 0
R4 [a0] = 0
R5 [a1] = 0
R6 [a2] = 7ffffe00
R7 [a3] = 0
R8 [t0] = 0
R9 [t1] = 0
R10 [t2] = 0
R11 [t3] = 0
R12 [t4] = 0
R13 [t5] = 0
R14 [t6] = 0
R15 [t7] = 0
R16 [s0] = 0
R17 [s1] = 0
R18 [s2] = 0
R19 [s3] = 0
R20 [s4] = 0
R21 [s5] = 0
R22 [s6] = 0
R23 [s7] = 0
R24 [s8] = 0
R25 [s9] = 0
R26 [k0] = 0
R27 [k1] = 0
R28 [gp] = 10008000
R29 [sp] = fffffdfe
R30 [s8] = 0
R31 [ra] = 0

[00400000] 8fa40000 lw $4, 0($29) ; User Text Segment [00400000]..[00440000]
[00400004] 27a50004 addiu $5, $29, 4 ; 183: lw $a0 0($sp) # argc
[00400008] 24a60004 addiu $6, $5, 4 ; 184: addiu $a1 $sp 4 # argv
[0040000c] 00010800 sll $2, $4, 2 ; 185: addiu $a2 $a1 4 # envp
[00400010] 00c23021 addu $6, $6, $2 ; 186: sll $v0 $a0 2
[00400014] 0c000000 jal 0x00000000 [main] ; 187: addu $a2 $a2 $v0
[00400018] 00000000 nop ; 188: jal main
[0040001c] 3402000a ori $2, $0, 10 ; 189: nop
[00400020] 0000000c syscall ; 190: li $v0 10
; 191: syscall # syscall 10 (exit)

[80000180] 0001d821 addu $27, $0, $1 ; Kernel Text Segment [80000000]..[80010000]
[80000184] 3c019000 lui $1, -28672 ; 90: move $k1 $at # Save $at
[80000188] ac220200 sw $2, 512($1) ; 92: sw $v0 $1 # Not re-entrant and we can't trust $sp
[8000018c] 3c019000 lui $1, -28672 ; 93: sw $a0 $2 # But we need to use these registers
[80000190] ac240204 sw $4, 516($1) ; 95: mfc0 $k0 $13 # Cause register
[80000194] 401a6800 mfc0 $26, $13 ; 96: srl $a0 $k0 2 # Extract ExCoDe Field
[80000198] 001a2082 srl $4, $26, 2 ; 97: andi $a0 $a0 0x1f
[8000019c] 3084001f andi $4, $4, 31 ; 101: li $v0 4 # syscall 4 (print_str)
[800001a0] 34020004 ori $2, $0, 4 ; 102: la $a0 __m1_
[800001a4] 3c049000 lui $4, -28672 [__m1_] ; 103: syscall
[800001a8] 0000000c syscall ; 105: li $v0 1 # syscall 1 (print_int)
[800001ac] 34020001 ori $2, $0, 1 ; 106: srl $a0 $k0 2 # Extract ExCoDe Field
[800001b0] 001a2082 srl $4, $26, 2 ; 107: andi $a0 $a0 0x1f
[800001b4] 3084001f andi $4, $4, 31 ; 108: syscall
[800001b8] 0000000c syscall ; 110: li $v0 4 # syscall 4 (print_str)
[800001bc] 34020004 ori $2, $0, 4 ; 111: andi $a0 $k0 0x3c
[800001c0] 3344003c andi $4, $26, 60 ; 112: lw $a0 __excp($a0)
[800001c4] 3c019000 lui $1, -28672
[800001c8] 00240821 addu $1, $1, $4
[800001cc] 8c240180 lw $4, 384($1)
[800001d0] 00000000 nop ; 113: nop
[800001d4] 0000000c syscall ; 114: syscall
[800001d8] 34010018 ori $1, $0, 24 ; 116: bne $k0 0x18 ok_pc # Bad PC exception requires special checks
[800001dc] 143a0008 bne $1, $26, 32 [ok_pc-0x800001dc] ; 117: nop
[800001e0] 00000000 nop ; 119: mfc0 $a0 $14 # EPC
[800001e4] 40047000 mfc0 $4, $14 ; 120: andi $a0 $a0 0x3 # Is EPC word-aligned?
[800001e8] 30840003 andi $4, $4, 3
[800001ec] 10040004 beq $0, $4, 16 [ok_pc-0x800001ec] ; 122: nop
[800001f0] 00000000 nop ; 124: li $v0 10 # Exit on really bad PC
[800001f4] 3402000a ori $2, $0, 10 ; 125: syscall
[800001f8] 0000000c syscall ; 128: li $v0 4 # syscall 4 (print_str)
[800001fc] 34020004 ori $2, $0, 4 ; 129: la $a0 __m2_
[80000200] 3c019000 lui $1, -28672 [__m2_] ; 130: syscall
[80000204] 3424000d ori $4, $1, 13 [__m2_] ; 132: srl $a0 $k0 2 # Extract ExCoDe Field
[80000208] 0000000c syscall ; 133: andi $a0 $a0 0x1f
[8000020c] 001a2082 srl $4, $26, 2 ; 134: bne $a0 0 ret # 0 means exception was an interrupt
[80000210] 3084001f andi $4, $4, 31 ; 135: nop
[80000214] 14040002 bne $0, $4, 8 [ret-0x80000214] ; 145: mfc0 $k0 $14 # Bump EPC register
[80000218] 00000000 nop ; 146: addiu $k0 $k0 4 # Skip faulting instruction
[8000021c] 401a7000 mfc0 $26, $14 ; 148: mfc0 $k0 $14
[80000220] 275a0004 addiu $26, $26, 4 ; 153: lw $v0 $1 # Restore other registers
[80000224] 409a7000 mfc0 $26, $14 ; 154: lw $a0 $2
[80000228] 3c019000 lui $1, -28672 ; 157: move $at $k1 # Restore $at
[8000022c] 8c220200 lw $2, 512($1) ; 160: mfc0 $k0 $13 # Clear Cause register
[80000230] 3c019000 lui $1, -28672
[80000234] 8c240204 lw $4, 516($1)
[80000238] 001b0821 addu $1, $0, $27
[8000023c] 400a6800 mfc0 $26, $13
```

programa almacenado en memoria

direcciones

código de operación

instrucciones

pseudo-instrucciones

Datos en QtSpim

```
PC = 4194336
EPC = 0
Cause = 0
BadVAddr = 0
Status = 805371664
HI = 1
LO = 78
R0 [r0] = 0
R1 [at] = 0
R2 [v0] = 10
R3 [v1] = 0
R4 [a0] = 6
R5 [a1] = 2147482984
R6 [a2] = 2147483000
R7 [a3] = 0
R8 [t0] = 9
R9 [t1] = 36
R10 [t2] = 9
R11 [t3] = 157
R12 [t4] = 2
R13 [t5] = 1
R14 [t6] = 0
R15 [t7] = 0
R16 [s0] = 0
R17 [s1] = 0
R18 [s2] = 0
R19 [s3] = 0
R20 [s4] = 0
R21 [s5] = 0
R22 [s6] = 0
R23 [s7] = 0
R24 [t8] = 0
R25 [t9] = 0
R26 [k0] = 0
R27 [k1] = 0
R28 [gp] = 268468224
R29 [sp] = 2147482980
R30 [s8] = 0
R31 [ra] = 4194328

User data segment [10000000]..[10040000]
[10000000]..[1000ffff] 00000000
[10010000] 0000000c 00000019 00000024 00000010 .....$.....
[10010010] 0000001c 00000025 0000007c 0000009c .....%.....
[10010020] 0000009d 00000009 00000000 00000000 .....
[10010030]..[1003ffff] 00000000

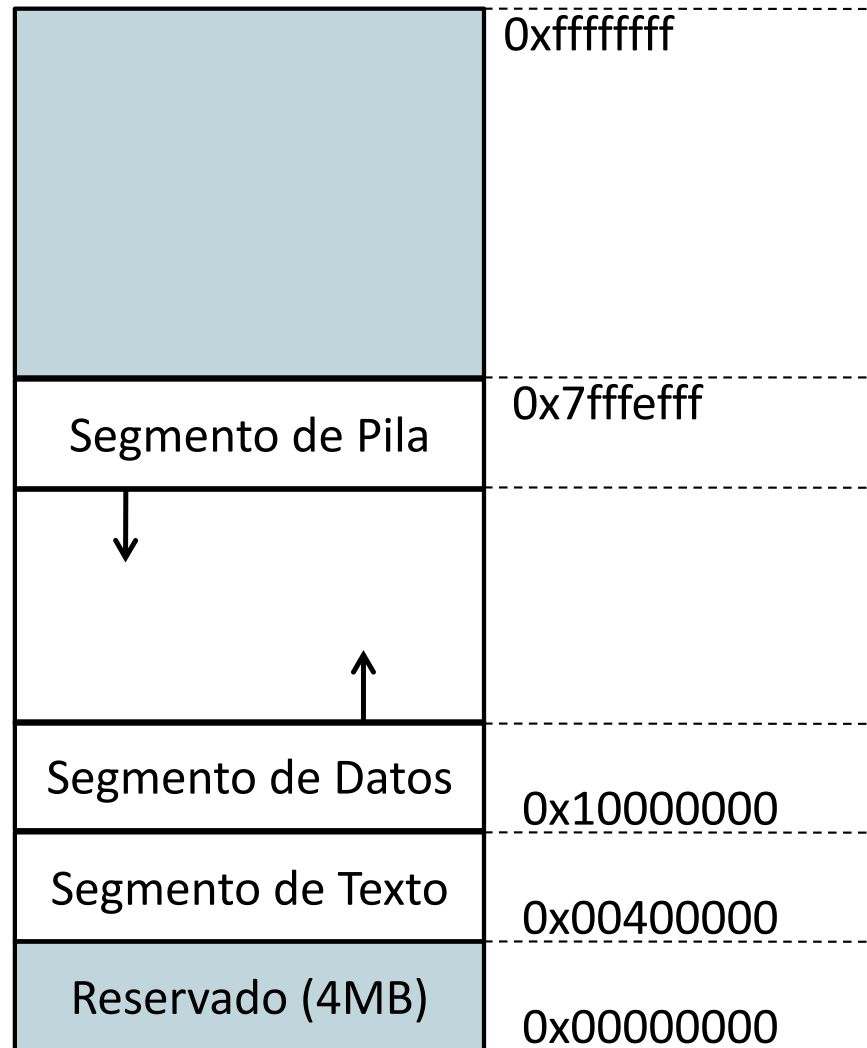
User Stack [7ffffd64]..[80000000]
[7ffffd64] 00000003 7ffffdf8 7ffffdf5 .....
[7ffffd70] 7ffffdb1 00000000 7fffffc7 7fffffa1 .....
[7ffffd80] 7fffff8e 7fffff33 7fffff11 .....3.....
[7ffffd90] 7ffffece 7ffffe75 7ffffe4d .....v.....
[7ffffda0] 7ffffe42 7ffffe34 00000000 7fffff7f B...4.....
[7ffffdb0] 6d6f4333 61747570 65726f64 50532f73 3Compu...adores/SP
[7ffffdc0] 2d2d4d49 5f63652f 63617270 61636974 IM-.../ec...practica
[7ffffdd0] 616d5f32 69726574 325f6e61 2d303130 2ma...terial...2010-
[7ffffde0] 31313032 2531765f 2656a65 69636963 2011...v1...ejercici
[7ffffdf0] 732e326f 00656400 6573552f 662f7372 o2.s...de.../Users/f
[7ffffe00] 78696c65 6f72442f 786f6270 6d6f682f elix/Dropbox/hom
[7ffffe10] 4f442f65 434e4543 752f4149 2f6d3363 e/DOCE...CIA/uc3m/
[7ffffe20] 64617247 49492d6f 7473452f 74637572 Gra...do.../Estruc...t
[7ffffe30] 00617275 5f435058 47414c46 78303d53 ura.XPC...F...AGS=0x
[7ffffe40] 53550030 663d5245 78696c65 43505800 0.USER=Felix.XPC
[7ffffe50] 5245535f 45434956 4d414e5f 6f633d45 _SERVICE_NAME=co
[7ffffe60] 616c2e6d 2e737572 70737471 372e6d69 m.larus.qtspim.7
[7ffffe70] 32313130 53494400 59414c50 72702f3d 0112.D...S...LAY=.../pr
[7ffffe80] 74617669 6d742f65 6f632f70 70612e6d i...vate/tmp/com...ap
[7ffffe90] 2e656c70 6e75616c 2e646863 7963696e ple.laun...chd.nicy
[7ffffea0] 5a366e6e 6f2f6d47 6d2e6f72 736f6361 nn6ZGM/or...g.macos
[7ffffeb0] 67726f66 71782e65 74726175 00303a7a forge.xquartz:0.
[7ffffec0] 4e474f4c 3d454d41 696c6566 41500078 LOGNAME=Felix.PA
[7ffffed0] 2f3d4854 2f272f75 3a6e6962 6e69622f TH=.../usr/bin/...bin
[7ffffee0] 73752f3a 62732f72 2f3a6e69 6e696273 :.../usr/sbin/...sbin
[7ffffef0] 48535300 5455415f 4f535f48 2f3d4b43 .SSH_AU...TH_SOCK=/
[7fffff00] 76697270 2f657461 2f706d74 2e6d6f63 private/tmp/com...
[7fffff10] 6c707061 616c2e65 68636e75 756c2e64 apple.laun...chd.lu
[7fffff20] 5855355a 76767244 73694c2f 656e6574 Z5UXD...rv...L...isten
[7fffff30] 41007372 656c7070 6275505f 5f627553 rs.Apple...pubsub
[7fffff40] 6b636f53 525f7465 65646e65 702f3d72 Socket...Render=.../p
[7fffff50] 61766972 742f6574 632f706d 612e6d6f rivate/tmp/com...a-
[7fffff60] 656c7070 75616c2e 6468636e 5a454a2e pple.laun...chd.JEZ
[7fffff70] 73656e37 2f647971 646e6552 53007265 7ne...sqd/Render.S
[7fffff80] 4c4c4548 69622f3d 61622f6e 48006873 HEL...L.../bin/bash.H
[7fffff90] 3d454d4f 6573552f 662f7372 78696c65 OME=.../Users/felix
[7fffffa0] 435f5f00 53555f46 545f5245 5f545845 _...CF...USER...TEXT=
[7fffffb0] 4f434e45 474e4944 3178303d 303a3646 E...NCODING=0x1F6:0
[7fffffc0] 303a3078 54003878 4944504d 762f3d52 x0:0x8...TH...DIR=.../v
[7fffffd0] 662f7261 65646c65 342f7372 32702f66 ar/fold...ers/4f/p2
[7fffffe0] 64346d66 70307377 78337776 7a306474 fm4dws0p...v...3...td0z
[7ffffff0] 62726d6c 30306836 70673030 002f542f lmr...b...h...0...0...gp/T/..
```

direcciones

datos en Hex

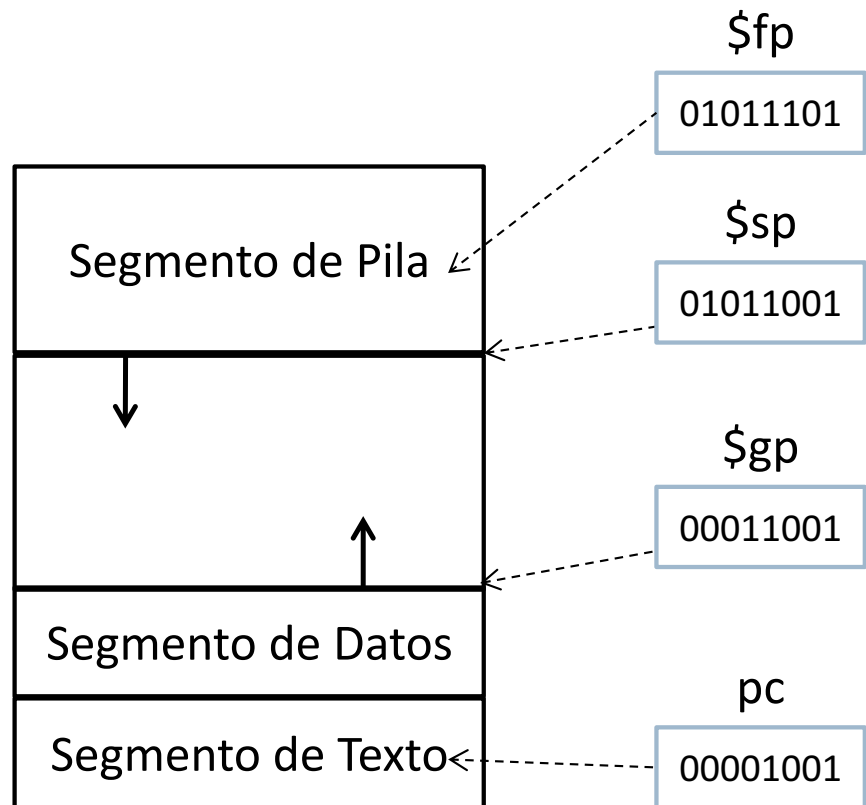
Datos (ASCII)

Memoria en QtSpim



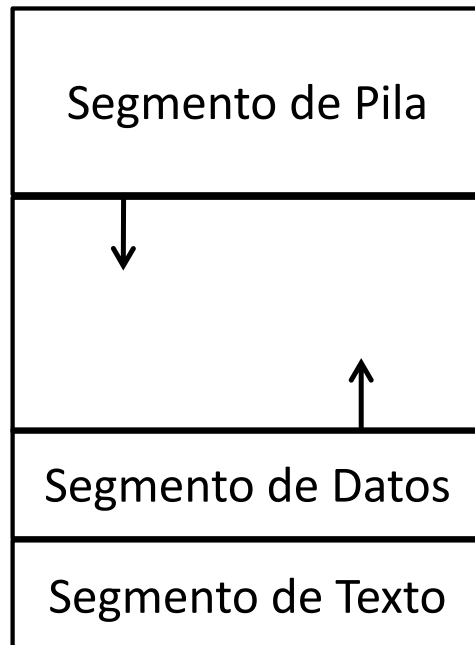
- ▶ Hay 4 GB de memoria direccionables en total
- ▶ Parte de esa memoria la utilizan los distintos segmentos de un proceso
- ▶ Otra parte de la memoria está reservada:
 - ▶ Un mini-sistema operativo reside en los primeros 4 MB de memoria

Mapa de memoria de un proceso



- ▶ Los procesos dividen el espacio de memoria en segmentos lógicos para organizar el contenido:
 - ▶ Segmento de pila
 - ▶ Variables locales
 - ▶ Contexto de funciones
 - ▶ Segmento de datos
 - ▶ Datos estáticos
 - ▶ Segmento de código (texto)
 - ▶ Código

Ejercicio

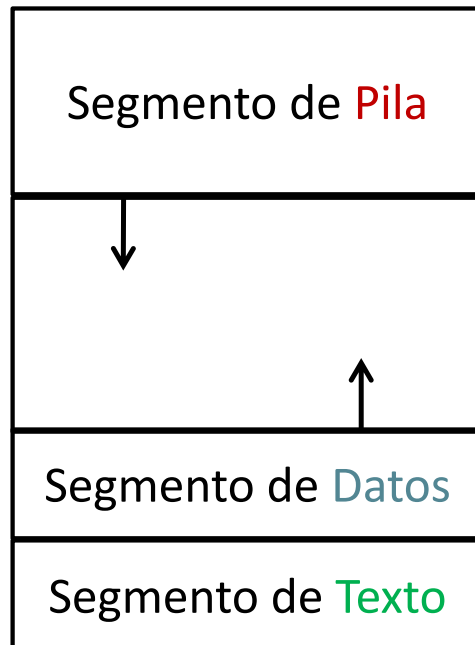


```
// variables globales
int a;

main ()
{
    // variables locales
    int b;

    // código
    return a + b;
}
```

Ejercicio (solución)

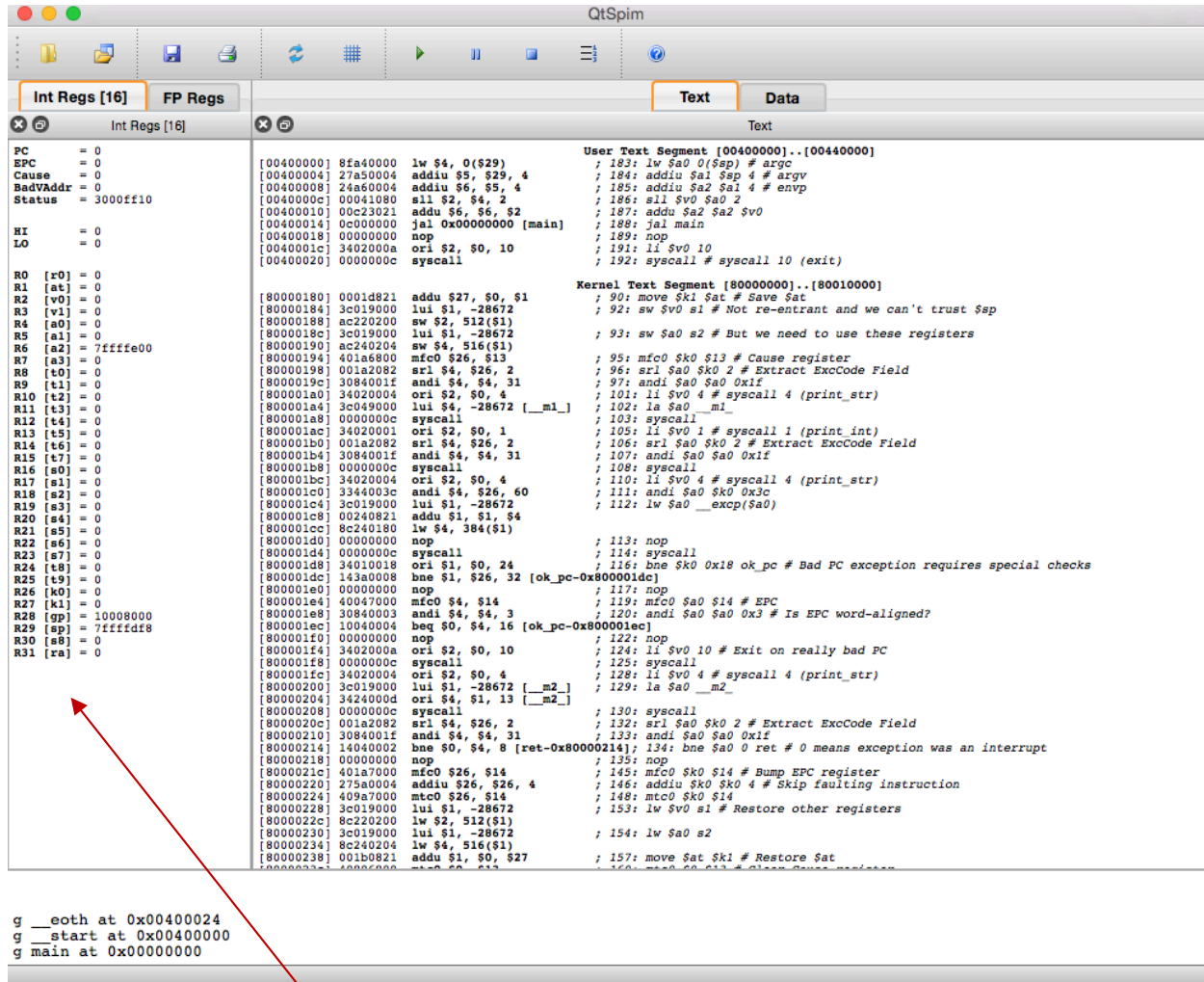


```
// variables globales
int a;

main ()
{
    // variables locales
    int b;

    // código
    return a + b;
}
```

Registros en QtSpim



The screenshot shows the QtSpim MIPS simulator interface. The 'Int Regs [16]' window is open, displaying the state of the 16 integer registers. The registers are listed on the left, and their values are shown on the right. A red arrow points from the word 'Registros' in the bottom right to the 'Int Regs [16]' window.

```
PC = 0
EPC = 0
Cause = 0
BadVAddr = 3000ff10
Status = 3000ff10

HI = 0
LO = 0

R0 [r0] = 0
R1 [at] = 0
R2 [v0] = 0
R3 [v1] = 0
R4 [a0] = 0
R5 [a1] = 0
R6 [a2] = 7ffffe00
R7 [a3] = 0
R8 [t0] = 0
R9 [t1] = 0
R10 [t2] = 0
R11 [t3] = 0
R12 [t4] = 0
R13 [t5] = 0
R14 [t6] = 0
R15 [t7] = 0
R16 [s0] = 0
R17 [s1] = 0
R18 [s2] = 0
R19 [s3] = 0
R20 [s4] = 0
R21 [s5] = 0
R22 [s6] = 0
R23 [s7] = 0
R24 [t8] = 0
R25 [t9] = 0
R26 [k0] = 0
R27 [k1] = 0
R28 [gp] = 10008000
R29 [sp] = 7ffffdf8
R30 [s8] = 0
R31 [ra] = 0
```

g __eoth at 0x00400024
g __start at 0x00400000
g main at 0x00000000

Registros

Banco de registros (enteros) del MIPS 32

Nombre registro	Número	Uso
zero	0	Constante 0
at	1	Reservado para el ensamblador
v0, v1	2, 3	Resultado de una rutina (o expresión)
a0, ..., a3	4, ..., 7	Argumento de entrada para rutinas
t0, ..., t7	8, ..., 15	Temporal (<u>NO</u> se conserva entre llamadas)
s0, ..., s7	16, ..., 23	Temporal (se conserva entre llamadas)
t8, t9	24, 25	Temporal (<u>NO</u> se conserva entre llamadas)
k0, k1	26, 27	Reservado para el sistema operativo
gp	28	Puntero al área global
sp	29	Puntero a pila
fp	30	Puntero a marco de pila
ra	31	Dirección de retorno (rutinas)

- ▶ Hay 32 registros
 - ▶ 4 bytes de tamaño (una palabra)
 - ▶ Se nombran con un \$ al principio
- ▶ Convenio de uso
 - ▶ Reservados
 - ▶ Argumentos
 - ▶ Resultados
 - ▶ Temporales
 - ▶ Punteros

Banco de registros del MIPS 32

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

- Hay 32 registros
 - ☐ 4 bytes de tamaño (una palabra)
 - ☐ Se nombran con un \$ al principio
- Convenio de uso
 - ☐ Reservados
 - ☐ Argumentos
 - ☐ Resultados
 - ☐ Temporales
 - ☐ Punteros

	16
	17
	18
	19
	20
	21
	22
	23
	24
	25
	26
	27
	28
	29
	30
	31

Banco de registros del MIPS 32

0		\$zero		16
1				17
2		Valor cableado a cero No puede modificarse		18
3				19
4				20
5				21
6				22
7				23
8				24
9				25
10				26
11				27
12				28
13				29
14				30
15				31

Banco de registros del MIPS 32

0		\$zero			16
1					17
2					18
3					19
4					20
5					21
6					22
7					23
8		\$t0		\$t8	24
9		\$t1		\$t9	25
10		\$t2			26
11		\$t3			27
12		\$t4			28
13		\$t5			29
14		\$t6			30
15		\$t7			31

Banco de registros del MIPS 32

0		\$zero		\$s0		16
1				\$s1		17
2			Valores guardados	\$s2		18
3				\$s3		19
4				\$s4		20
5				\$s5		21
6				\$s6		22
7				\$s7		23
8		\$t0		\$t8		24
9		\$t1		\$t9		25
10		\$t2				26
11		\$t3				27
12		\$t4				28
13		\$t5				29
14		\$t6				30
15		\$t7				31

Banco de registros del MIPS 32

0		\$zero		\$s0		16
1				\$s1		17
2		\$v0		\$s2		18
3		\$v1		\$s3		19
4		\$a0		\$s4		20
5		\$a1		\$s5		21
6		\$a2		\$s6		22
7		\$a3		\$s7		23
8		\$t0		\$t8		24
9		\$t1		\$t9		25
10		\$t2				26
11		\$t3				27
12		\$t4				28
13		\$t5	Paso de parámetros y Gestión de subrutinas	\$sp		29
14		\$t6		\$fp		30
15		\$t7		\$ra		31

Banco de registros del MIPS 32

0		\$zero		\$s0		16
1		\$at		\$s1		17
2		\$v0		\$s2		18
3		\$v1		\$s3		19
4		\$a0		\$s4		20
5		\$a1		\$s5		21
6		\$a2		\$s6		22
7		\$a3		\$s7		23
8		\$t0		\$t8		24
9		\$t1		\$t9		25
10		\$t2		\$k0		26
11		\$t3		\$k1		27
12		\$t4		\$gp		28
13		\$t5		\$sp		29
14		\$t6		\$fp		30
15		\$t7		\$ra		31

Otros

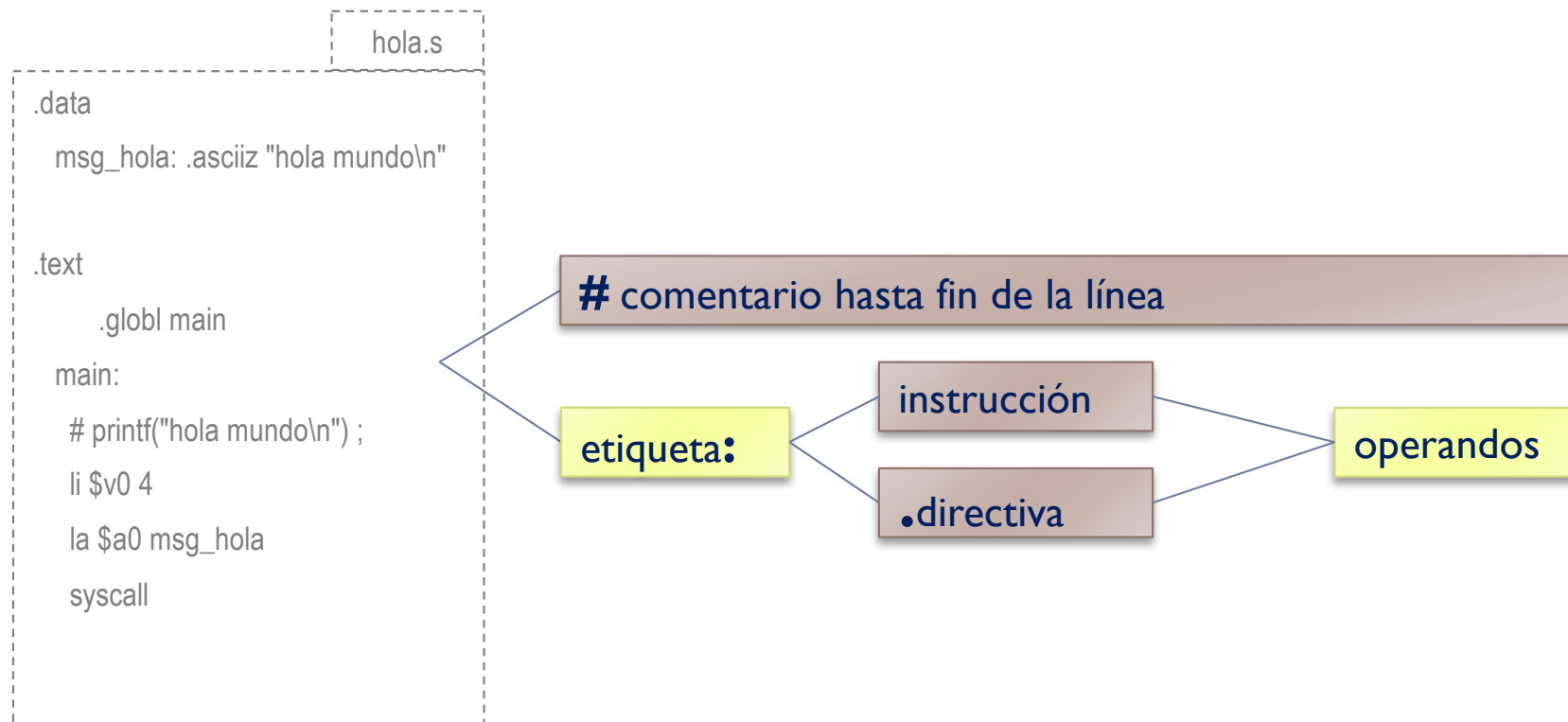
Ejemplo: Hola mundo...

hola.s

```
.data
    msg_hola: .asciiz "hola mundo\n"

.text
    .globl main
main:
    # printf("hola mundo\n") ;
    li $v0 4
    la $a0 msg_hola
    syscall
```

Ejemplo: Hola mundo...



Ejemplo: Hola mundo...

hola.s

.data

```
msg_hola: .ascii "hola mundo\n"
```

.text

```
.globl main
```

```
main:
```

```
    # printf("hola mundo\n") ;
```

```
    li $v0 4
```

```
    la $a0 msg_hola
```

```
    syscall
```

Ejemplo: Hola mundo...

hola.s

.data

```
msg_hola: .ascii "hola mundo\n"
```

.text

.globl main

main:

```
# printf("hola mundo\n") ;
```

```
li $v0 4
```

```
la $a0 msg_hola
```

```
syscall
```

etiqueta: representa la dirección de memoria donde comienza la función main

comentarios

instrucciones

Ejemplo: Hola mundo...

hola.s

.data

msg_hola: **.ascii** "hola mundo\n"

segmento de datos

.text

.globl main

main:

printf("hola mundo\n") ;

li \$v0 4

la \$a0 msg_hola

syscall

msg_hola: representa la dirección de memoria donde comienza la cadena

segmento de código

QtSpim

Banco de registros

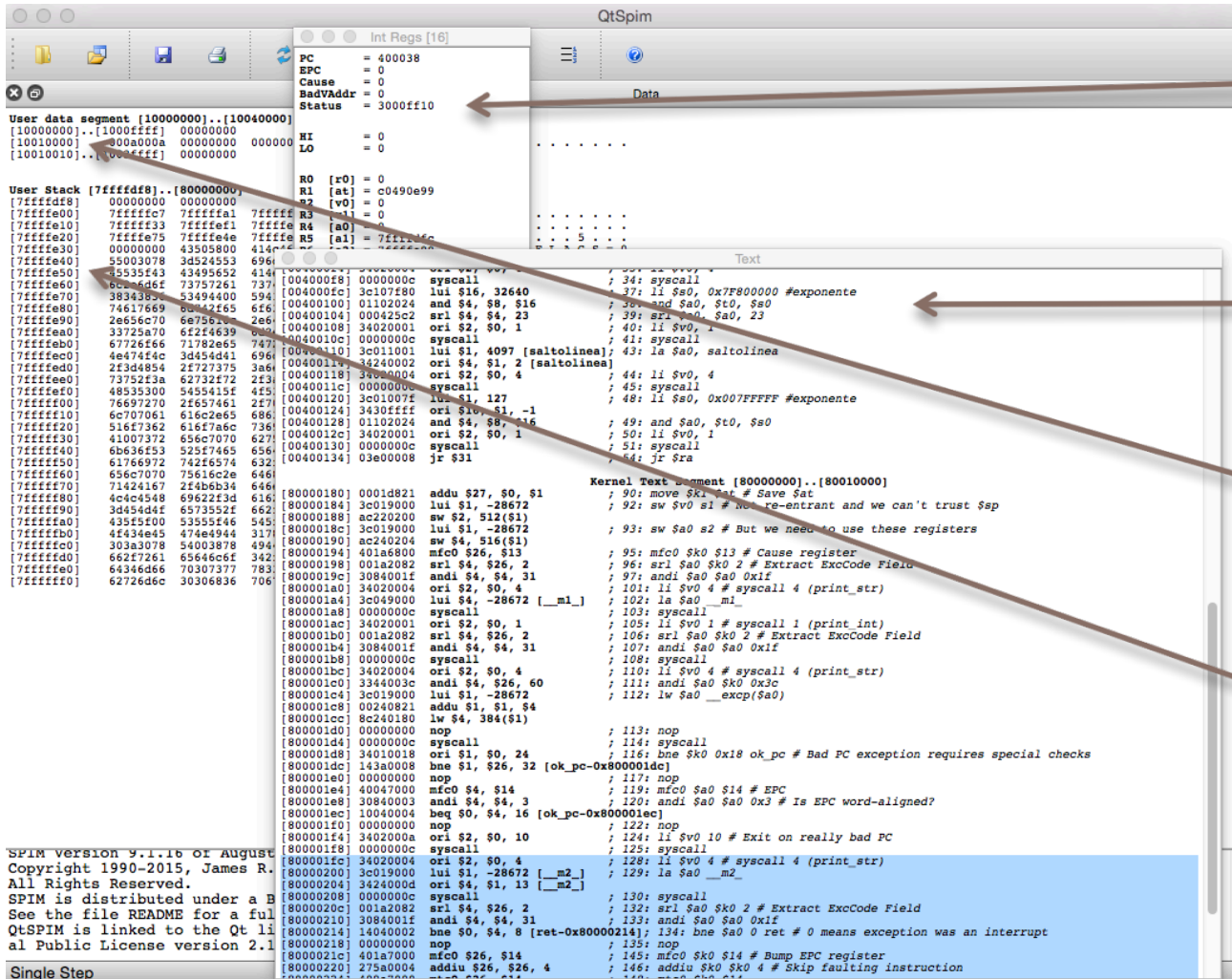
\$0, \$1, \$2, ...

$\$f_0, \f_1, \dots

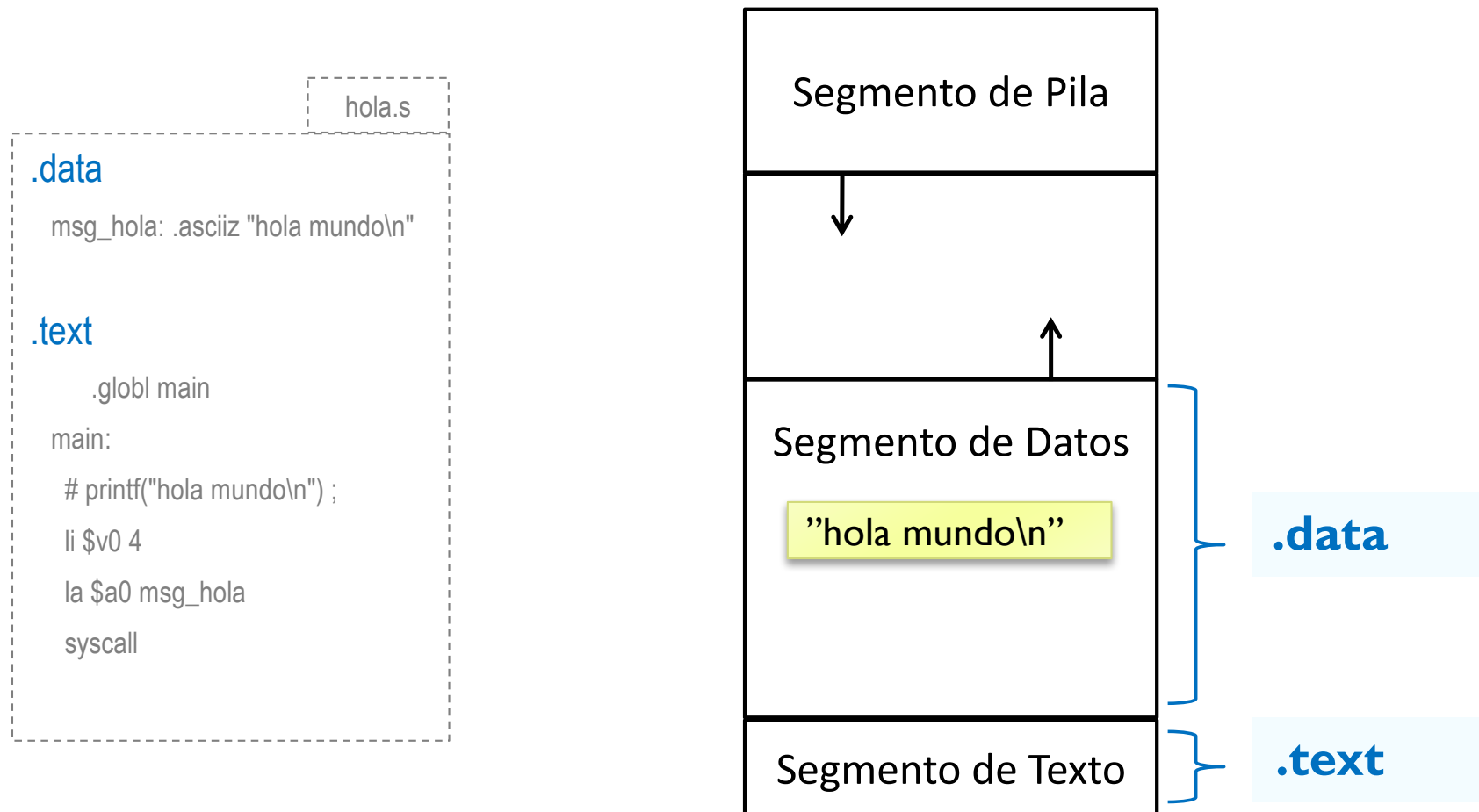
Segmento de código

Segmento de datos

Segmento de pila



Programa en ensamblador: directivas de ensamblador (de preproceso)



Programa en ensamblador:

directivas de ensamblador

Directivas	Uso
.data	Siguientes elementos van al segmento de dato
.text	Siguientes elementos van al segmento de código
.ascii <i>“tira de caracteres”</i>	Almacena cadena caracteres NO terminada en carácter nulo
.asciiz <i>“tira de caracteres”</i>	Almacena cadena caracteres terminada en carácter nulo
.byte 1, 2, 3	Almacena bytes en memoria consecutivamente
.half 300, 301, 302	Almacena medias palabras en memoria consecutivamente
.word 800000, 800001	Almacena palabras en memoria consecutivamente
.float 1.23, 2.13	Almacena float en memoria consecutivamente
.double 3.0e21	Almacena double en memoria consecutivamente
.space 10	Reserva un espacio de 10 bytes en el segmento actual
.extern <i>etiqueta n</i>	Declara que <i>etiqueta</i> es global de tamaño <i>n</i>
.globl <i>etiqueta</i>	Declara <i>etiqueta</i> como global
.align <i>n</i>	Alinea el siguiente dato en un límite de 2^n

Definición de datos estáticos

etiqueta (dirección) tipo de dato (directiva) valor

```
.data
cadena : .asciiz "Hola mundo\n"
i1: .word 10      # int i1=10
i2: .word -5      # int i2=-5
i3: .half 300     # short i3=300
c1: .byte 100     # char c1=100
c2: .byte 'a'     # char c2='a'
f1: .float 1.3e-4 # float f1=1.3e-4
d1: .double .001  # double d1=0.001

# int v[3] = { 0 , -1, 0xffffffff }; int w[100];
v: .word 0, -1, 0xffffffff
w: .word 400
```

Llamadas al sistema en QtSpim

- ▶ El simulador incluye un pequeño “sistema operativo”
 - ▶ Ofrece 17 servicios.
- ▶ Invocación:
 - ▶ Código de servicio en \$v0
 - ▶ Otros parámetros en registros concretos
 - ▶ Invocación mediante instrucción máquina **syscall**

Llamadas al sistema en QtSpim

Servicio	Código de llamada (\$v0)	Argumentos	Resultado
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer en \$v0
read_float	6		float en \$f0
read_double	7		double en \$f0
read_string	8	\$a0=buffer, \$a1=longitud	
sbrk	9	\$a0=cantidad	dirección en \$v0
exit	10		

Llamadas al sistema en QtSpim

Servicio	Código de llamada (\$v0)	Argumentos	Resultado
print_char	11	\$a0 (código ASCII)	
read_char	12		\$v0 (código ASCII)
open	13	Equivalente a \$v0 = open(\$a0, \$a1, \$a2)	descriptor de fichero en \$v0
read	14	Equivalente a \$v0 = read (\$a0, \$a1, \$a2)	bytes leídos en \$v0
write	15	Equivalente a \$v0 = write(\$a0, \$a1, \$a2)	bytes escritos en \$v0
close	16	Equivalente a \$v0 = close(\$a0)	0 en \$v0
exit2	17	Termina el programa y hace que spim devuelva el código de error almacenado en \$a0	

Ejemplo: Hola mundo...

hola.s

.data

msg_hola: .asciiz "hola mundo\n"

.text

.globl main

main:

printf("hola mundo\n") ;

li \$v0 4

la \$a0 msg_hola

syscall

Servicio	Código de llamada	Argumentos
<u>print_int</u>	1	\$a0 = <u>integer</u>
<u>print_float</u>	2	\$f12 = <u>float</u>
<u>print_double</u>	3	\$f12 = <u>double</u>
<u>print_string</u>	4	\$a0 = <u>string</u>

instrucción de
llamada al sistema

Ejercicio

. . .

```
int valor ;
```

. . .

```
readInt(&valor) ;
```

```
valor = valor + 1 ;
```

```
printInt(valor) ;
```

. . .

Servicio	Código de llamada	Argumentos	Resultado
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer en \$v0
read_float	6		float en \$f0
read_double	7		double en \$f0
read_string	8	\$a0=buffer, \$a1=long.	
sbrk	9	\$a0=cantidad	dirección en \$v0
exit	10		

Ejercicio (solución)

. . .

```
int valor ;
```

. . .

```
readInt(&valor) ;
```

```
valor = valor + 1 ;
```

```
printInt(valor) ;
```

. . .

Servicio	Código	Argumentos	Resultado
print_int	1	\$a0 = integer	
read_int	5		integer en \$v0

. . .

```
# readInt(&valor)
```

```
li $v0 5
```

```
syscall
```

```
sw $v0 valor
```

```
# valor = valor + 1
```

```
add $a0 $v0 1
```

```
sw $v0 valor
```

```
# printInt
```

```
li $v0 1
```

```
syscall
```

Ejercicio

```
int x = 10;
int y = 20;
main() {
    print_string("La suma de ");
    print_int(x);
    print_string(" y de ");
    print_int(y);
    print_string(" es ");
    print_int(x+y);
    print_character("\n");
}
```

Ejercicio (solución)

```
int x = 10;
int y = 20;
main() {
    print_string("La suma de ");
    print_int(x);
    print_string(" y de ");
    print_int(y);
    print_string(" es ");
    print_int(x+y);
    print_character("\n");
}
```

```
.rdata
m1:    .asciiz "La suma de "
m2:    .asciiz " y de "
m3:    .asciiz " es "
salto: .ascii "\n"

.data
x:     .word 10
y:     .word 20

.text
.globl main
main:
    la $a0, m1
    li $v0, 4
    syscall # print_string(m1)
    lw $a0, x
    li $v0, 1
    syscall # print_int(x)
```

```
la $a0, m2
li $v0, 4
syscall # print_string(m2)
lw $a0, y
li $v0, 1
syscall # print_int(y)
la $a0, m3
li $v0, 4
syscall # print_string(m3)
lw $t0, x
lw $t1, y
add $a0, $t0, $t1 # $a0= x + y
li $v0, 1
syscall # print_int($a0)
lb $a0, salto
li $v0, 11
syscall # print_character(salto)
jr $ra
```

Instrucciones y pseudoinstrucciones

- ▶ Una instrucción en ensamblador se corresponde con una única instrucción máquina
 - ▶ Ocupa 32 bits en el MIPS 32
 - ▶ `addi $t1, $t0, 4`
- ▶ Una pseudoinstrucción se puede utilizar en un programa en ensamblador pero no se corresponde con ninguna instrucción máquina
 - ▶ Ej: `li $v0, 4`
`move $t1, $t0`
- ▶ En el proceso de ensamblado se sustituyen por la secuencia de instrucciones máquina que realizan la misma funcionalidad.
 - ▶ Ej.: `ori $v0, $0, 4` sustituye a: `li $v0, 4`
`addu $t1, $0, $t2` sustituye a: `move $t1, $t2`

Otros ejemplos de pseudoinstrucciones

- ▶ Una pseudoinstrucción en ensamblador se puede corresponder con varias instrucciones máquina.

- ▶ `li $t1, 0x00800010`

- ▶ No cabe en 32 bits, pero se puede utilizar como pseudoinstrucción.

- ▶ Es equivalente a:

- `lui $t1, 0x0080`

- `ori $t1, $t1, 0x0010`

QtSpim

Int Regs [16]

PC = 0
EPC = 0
Cause = 0
BadVAddr = 0
Status = 3000fff10
HI = 0
LO = 0
R0 [r0] = 0
R1 [at] = 0
R2 [v0] = 0
R3 [v1] = 0
R4 [a0] = 0
R5 [a1] = 0
R6 [a2] = 7ffffe00
R7 [a3] = 0
R8 [t0] = 0
R9 [t1] = 0
R10 [t2] = 0
R11 [t3] = 0
R12 [t4] = 0
R13 [t5] = 0
R14 [t6] = 0
R15 [t7] = 0
R16 [s0] = 0
R17 [s1] = 0
R18 [s2] = 0
R19 [s3] = 0
R20 [s4] = 0
R21 [s5] = 0
R22 [s6] = 0
R23 [s7] = 0
R24 [s8] = 0
R25 [t9] = 0
R26 [k0] = 0
R27 [k1] = 0
R28 [gp] = 10008000
R29 [sp] = 7ffffdf8
R30 [s8] = 0
R31 [ra] = 0

Text

User Text Segment [00400000]..[00440000]
; 183: lw \$a0 0(\$sp) # argc
; 184: addiu \$a1 \$sp 4 # argv
; 185: addiu \$a2 \$a1 4 # envp
; 186: sll \$v0 \$a0 2
; 187: addu \$a2 \$a2 \$v0
; 188: jal main
; 189: nop
; 191: li \$v0 10
; 192: syscall # syscall 10 (exit)

Kernel Text Segment [80000000]..[80010000]
; 90: move \$k1 \$at # Save \$at
; 92: sw \$v0 \$1 # Not re-entrant and we can't trust \$sp
; 93: sw \$a0 \$2 # But we need to use these registers
; 95: mfc0 \$k0 \$13 # Cause register
; 96: srl \$a0 \$k0 2 # Extract ExCoDe Field
; 97: andi \$a0 \$a0 0x1f
; 101: li \$v0 4 # syscall 4 (print_str)
; 102: la \$a0 __m1_
; 103: syscall
; 105: li \$v0 1 # syscall 1 (print_int)
; 106: srl \$a0 \$k0 2 # Extract ExCoDe Field
; 107: andi \$a0 \$a0 0x1f
; 108: syscall
; 110: li \$v0 4 # syscall 4 (print_str)
; 111: andi \$a0 \$k0 0x3c
; 112: lw \$a0 __excp(\$a0)
; 113: nop
; 114: syscall
; 116: bne \$k0 0x18 ok_pc # Bad PC exception requires special checks
; 117: nop
; 119: mfc0 \$a0 \$14 # EPC
; 120: andi \$a0 \$a0 0x3 # Is EPC word-aligned?
; 122: nop
; 124: li \$v0 10 # Exit on really bad PC
; 125: syscall
; 128: li \$v0 4 # syscall 4 (print_str)
; 129: la \$a0 __m2_
; 130: syscall
; 132: srl \$a0 \$k0 2 # Extract ExCoDe Field
; 133: andi \$a0 \$a0 0x1f
; 134: bne \$a0 0 ret # 0 means exception was an interrupt
; 135: nop
; 145: mfc0 \$k0 \$14 # Bump EPC register
; 146: addiu \$k0 \$k0 4 # Skip faulting instruction
; 148: mfc0 \$k0 \$14
; 153: lw \$v0 \$1 # Restore other registers
; 154: lw \$a0 \$2
; 157: move \$at \$k1 # Restore \$at
; 160: mfc0 \$k0 \$13 # Clear Cause register

g __eoth at 0x00400024
g __start at 0x00400000
g main at 0x00000000

programa almacenado
en memoria

direcciones

código binario

instrucciones

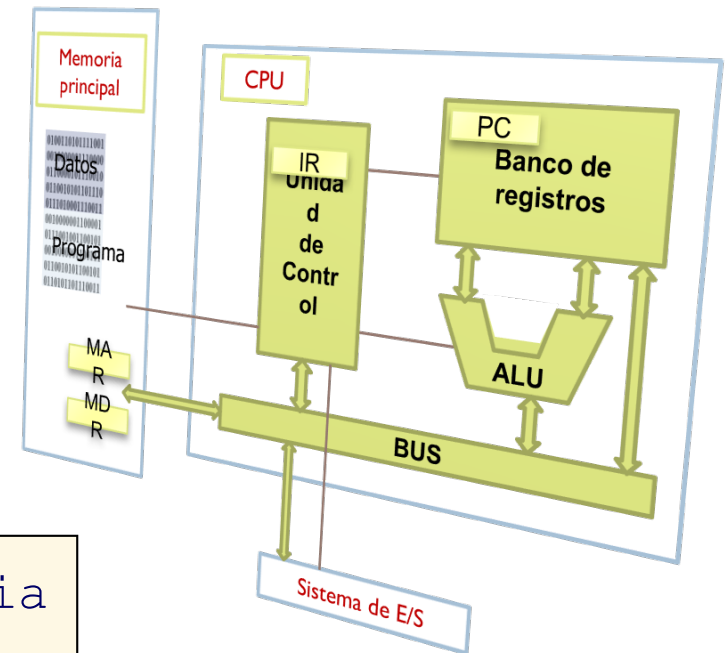
pseudo-instrucciones

Formato de las instrucciones de acceso a memoria

lw
sw
lb
sb
lbu

Registro, dirección de memoria

- **Número** que representa una dirección
- **Etiqueta** simbólica que representa una dirección
- **(registro)**: representa la dirección almacenada en el registro
- **num(registro)**: representa la dirección que se obtiene de sumar num con la dirección almacenada en el registro



Formatos de las instrucciones de acceso memoria

- ▶ `lbu $t0, 0x0F000002`
 - ▶ Direccionamiento directo. Se carga en \$t0 el byte almacenado en la posición de memoria 0x0F000002
- ▶ `lbu $t0, etiqueta`
 - ▶ Direccionamiento directo. Se carga en \$t0 el byte almacenado en la posición de memoria `etiqueta`
- ▶ `lbu $t0, ($t1)`
 - ▶ Direccionamiento indirecto de registro. Se carga en \$t0 el byte almacenado en la posición de memoria almacenada en \$t1
 - ▶ `lbu $t0, 80($t1)`
 - ▶ Direccionamiento relativo. Se carga en \$t0 el byte almacenado en la posición de memoria que se obtiene de sumar el contenido de \$t1 con 80

Transferencia de datos bytes

- ▶ Copia un **byte** de **memoria** a un **registro** o viceversa

- ▶ Para bytes:

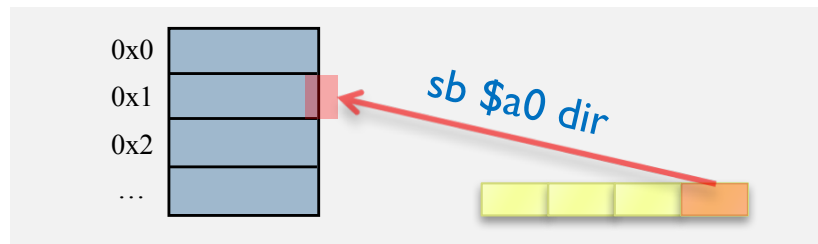
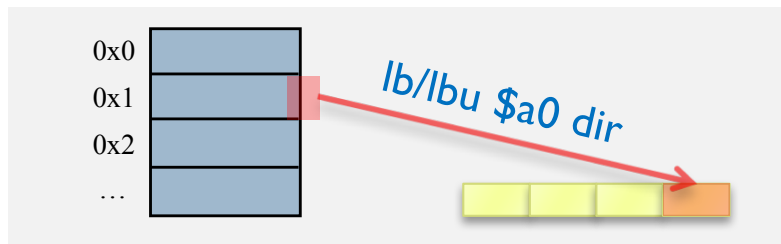
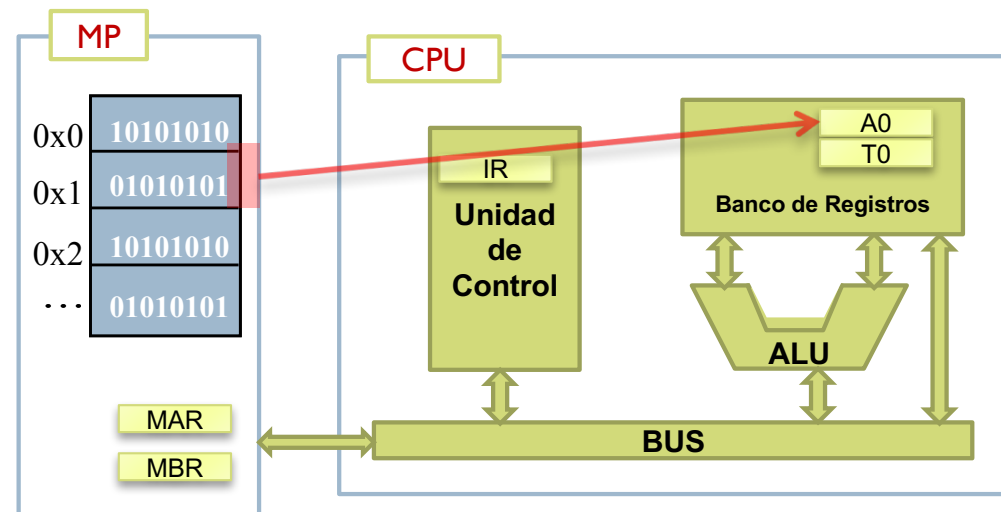
- ▶ Memoria a registro

lb \$a0, dir

lbu \$a0, dir

- ▶ Registro a memoria

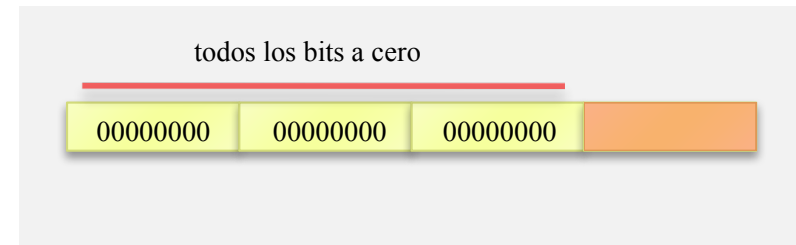
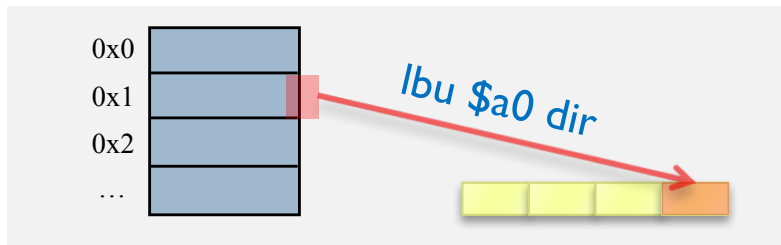
sb \$t0, dir



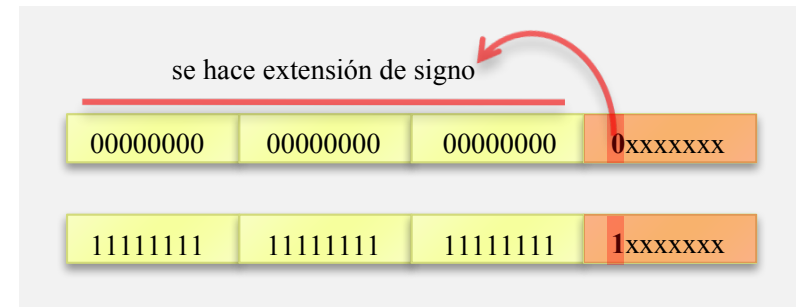
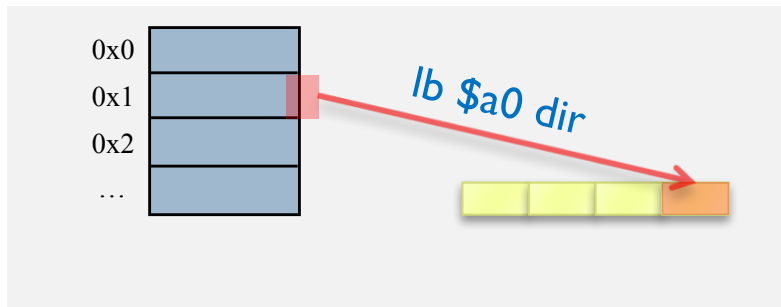
Transferencia de datos

Extensión de signo

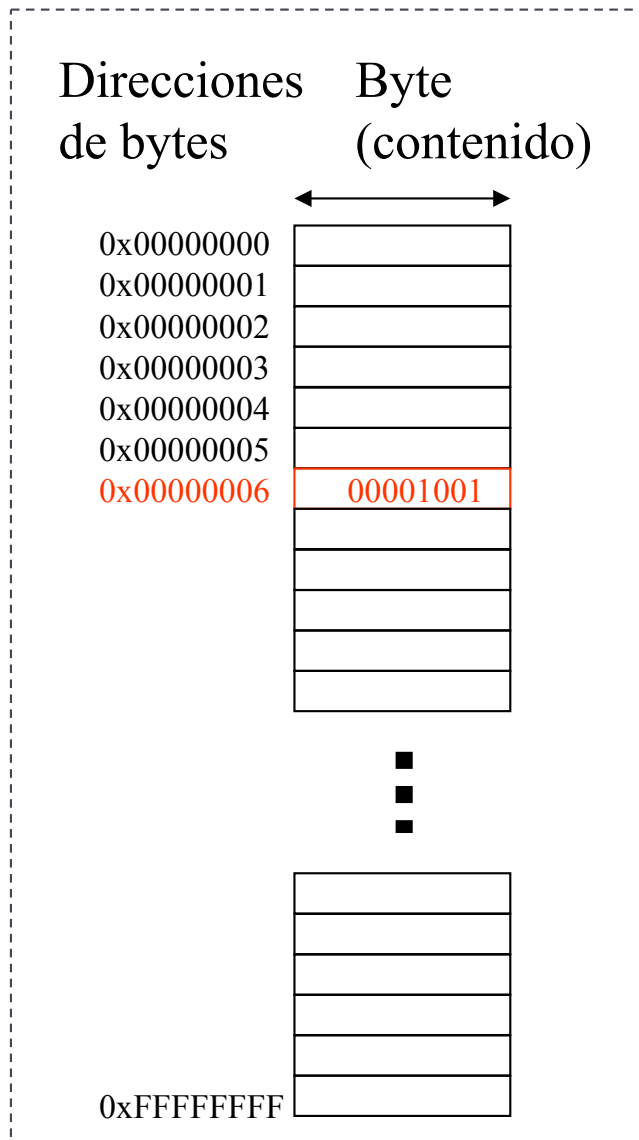
- ▶ Hay dos posibilidades a la hora de traer un byte de memoria a registro:
- ▶ A) Transferir **sin signo**, por ejemplo: `lbu $a0, dir`



- ▶ B) Transferir **con signo**, por ejemplo: `lb $a0, dir`



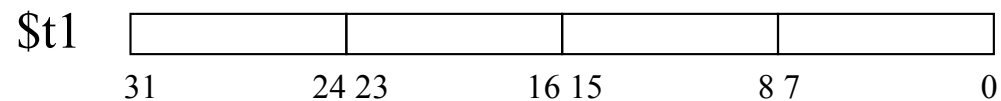
Acceso a bytes con lb



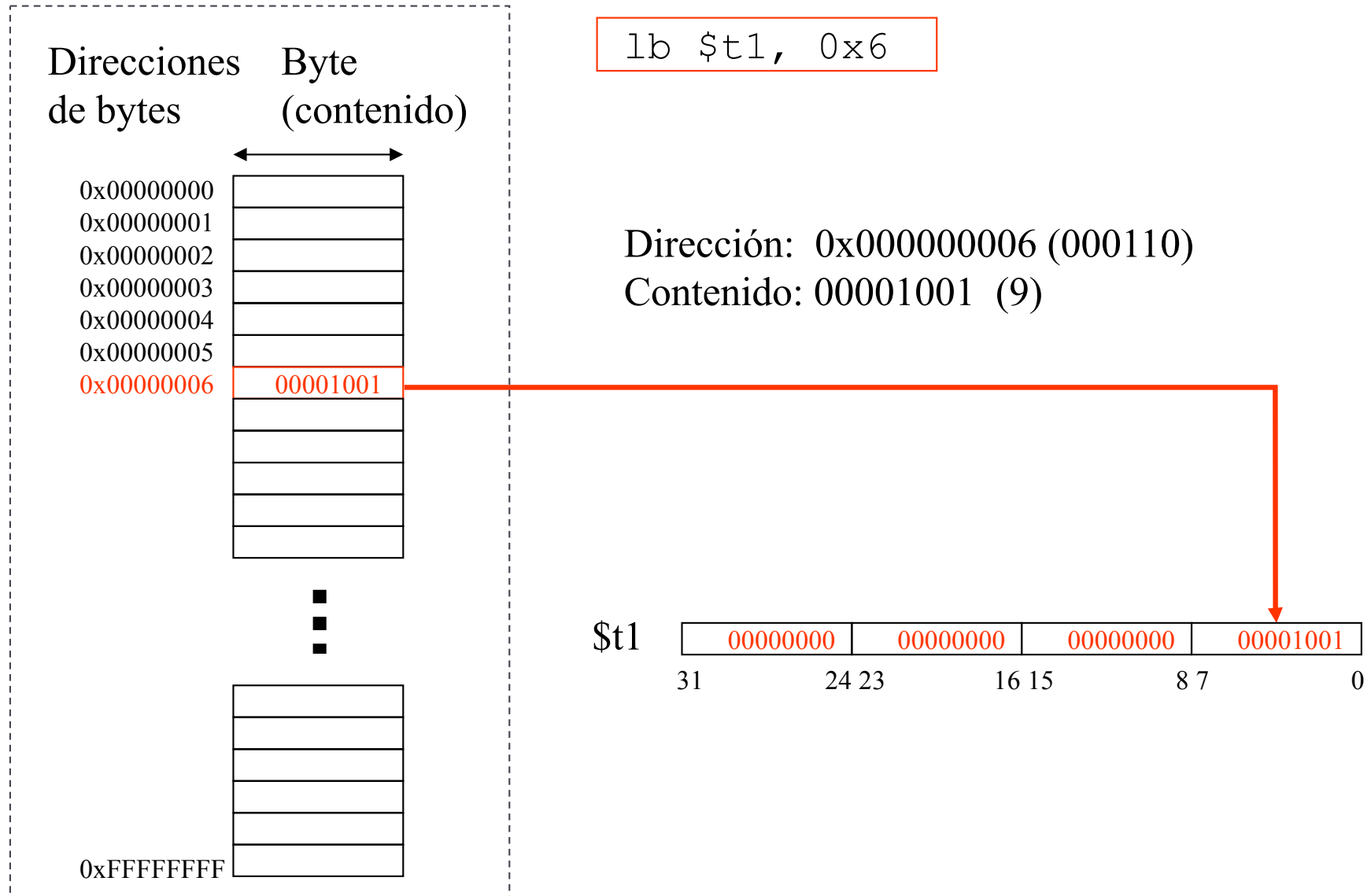
```
lb $t1, 0x6
```

Dirección: 0x000000006 (000110)

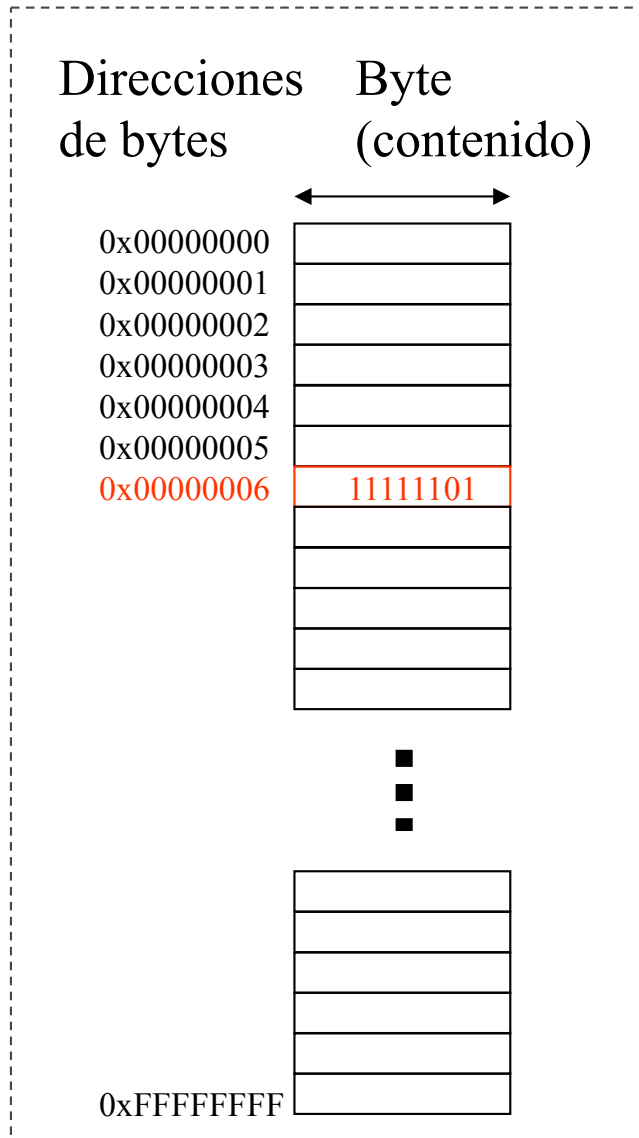
Contenido: 00001001 (9)



Accesso a bytes con lb



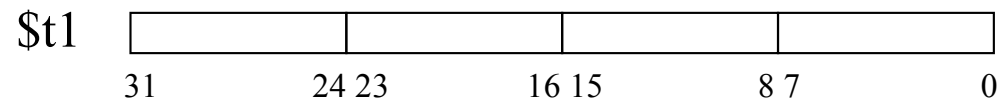
Accesso a bytes con lb



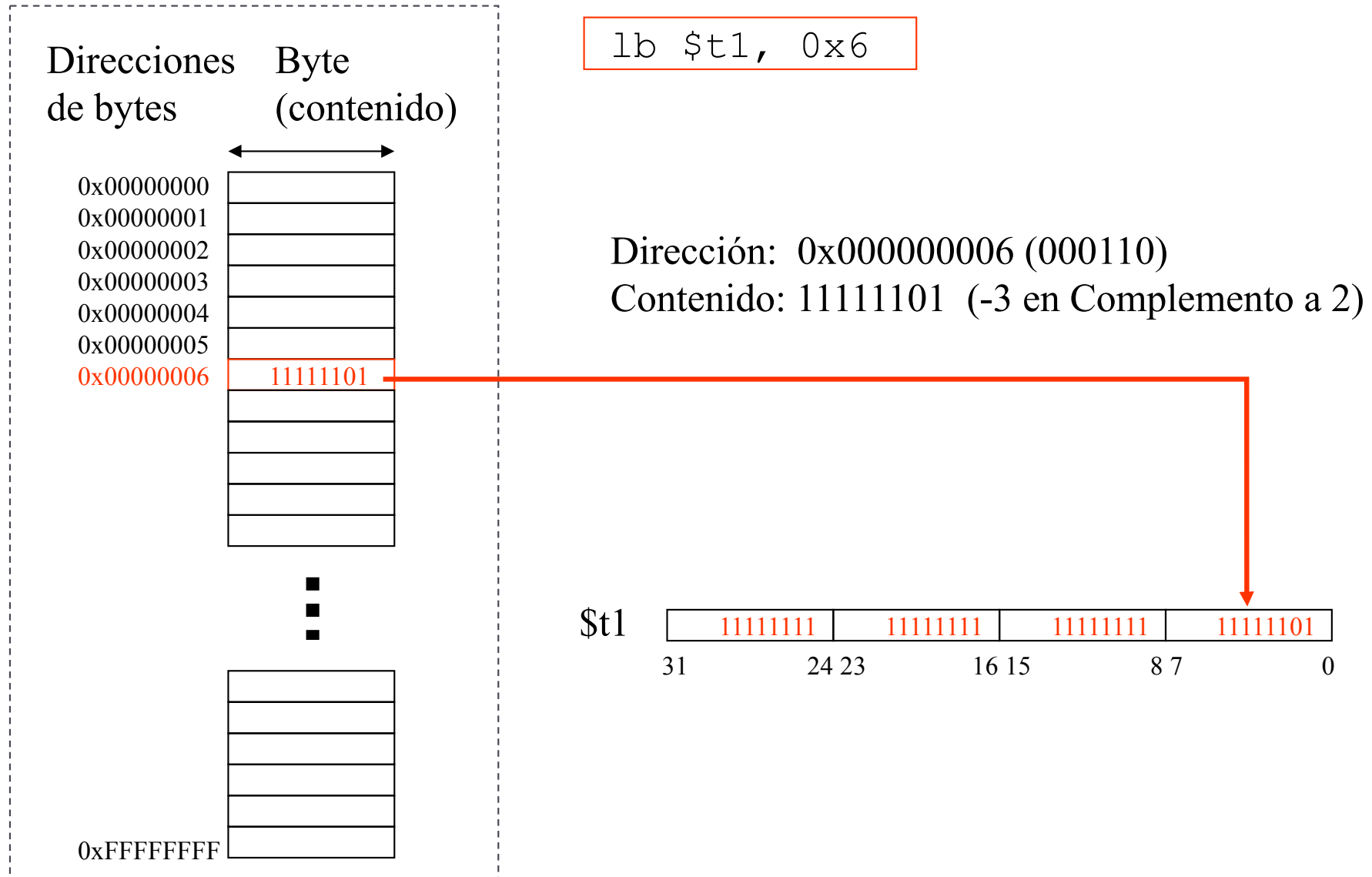
1b \$t1, 0x6

Dirección: 0x000000006 (000110)

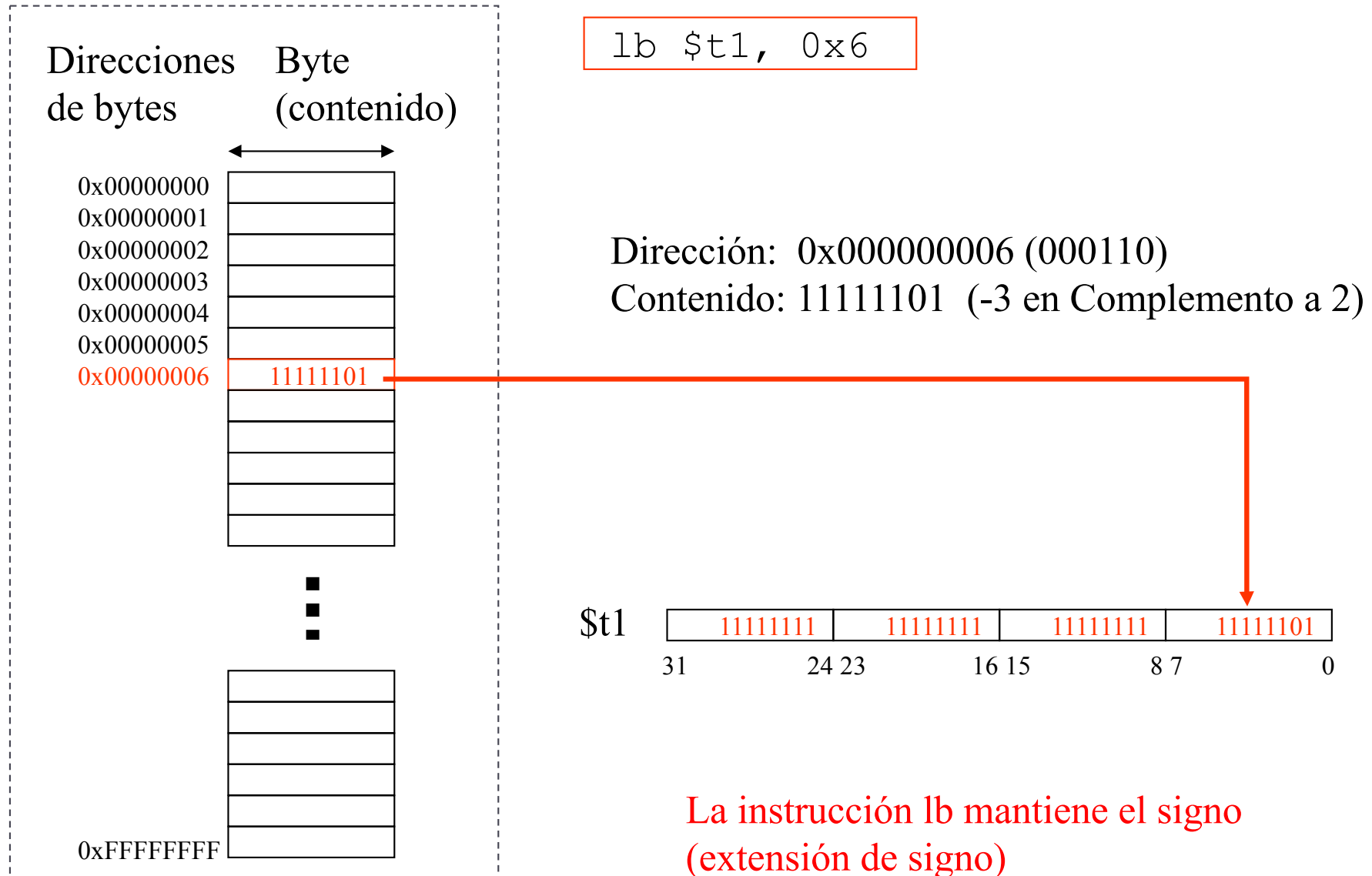
Contenido: 1111101 (-3 en Complemento a 2)



Accesso a bytes con lb

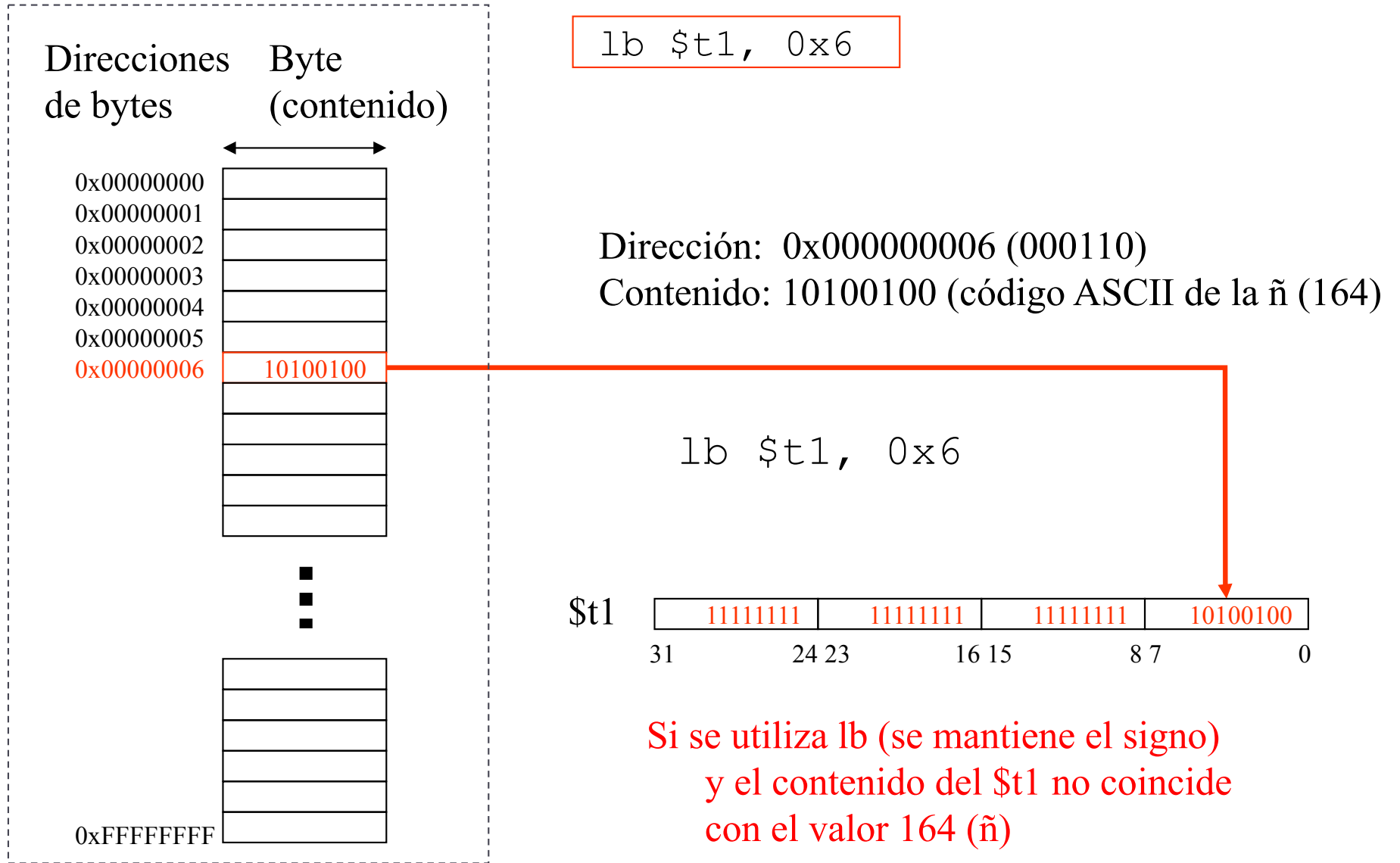


Acceso a bytes con lb

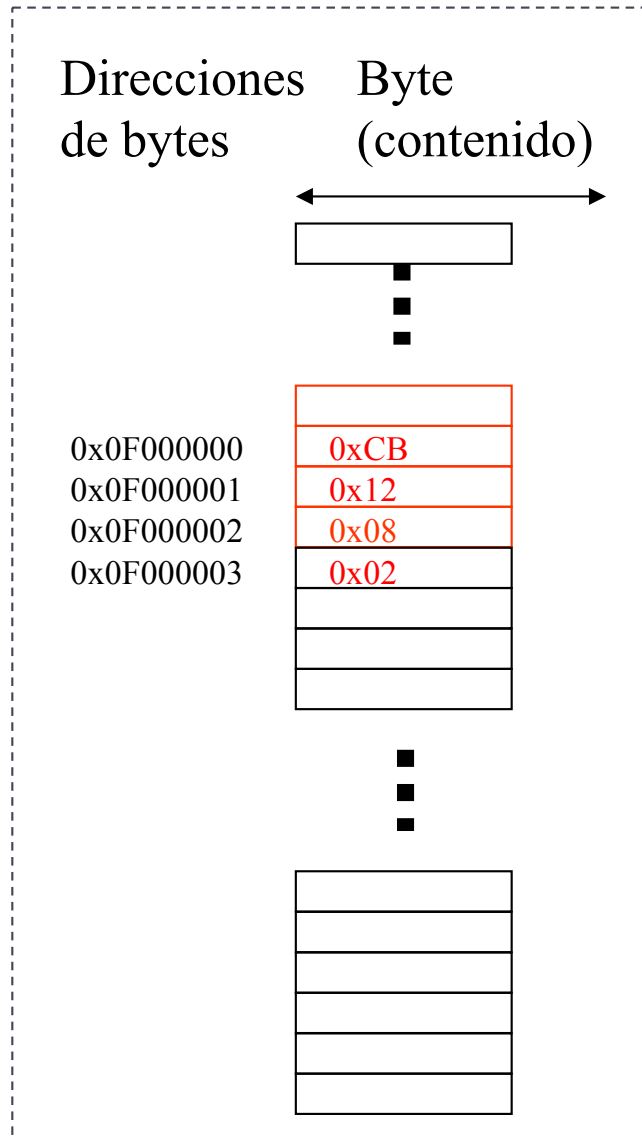


Acceso a bytes con lb

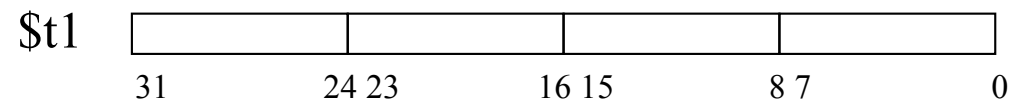
problemas accediendo a caracteres



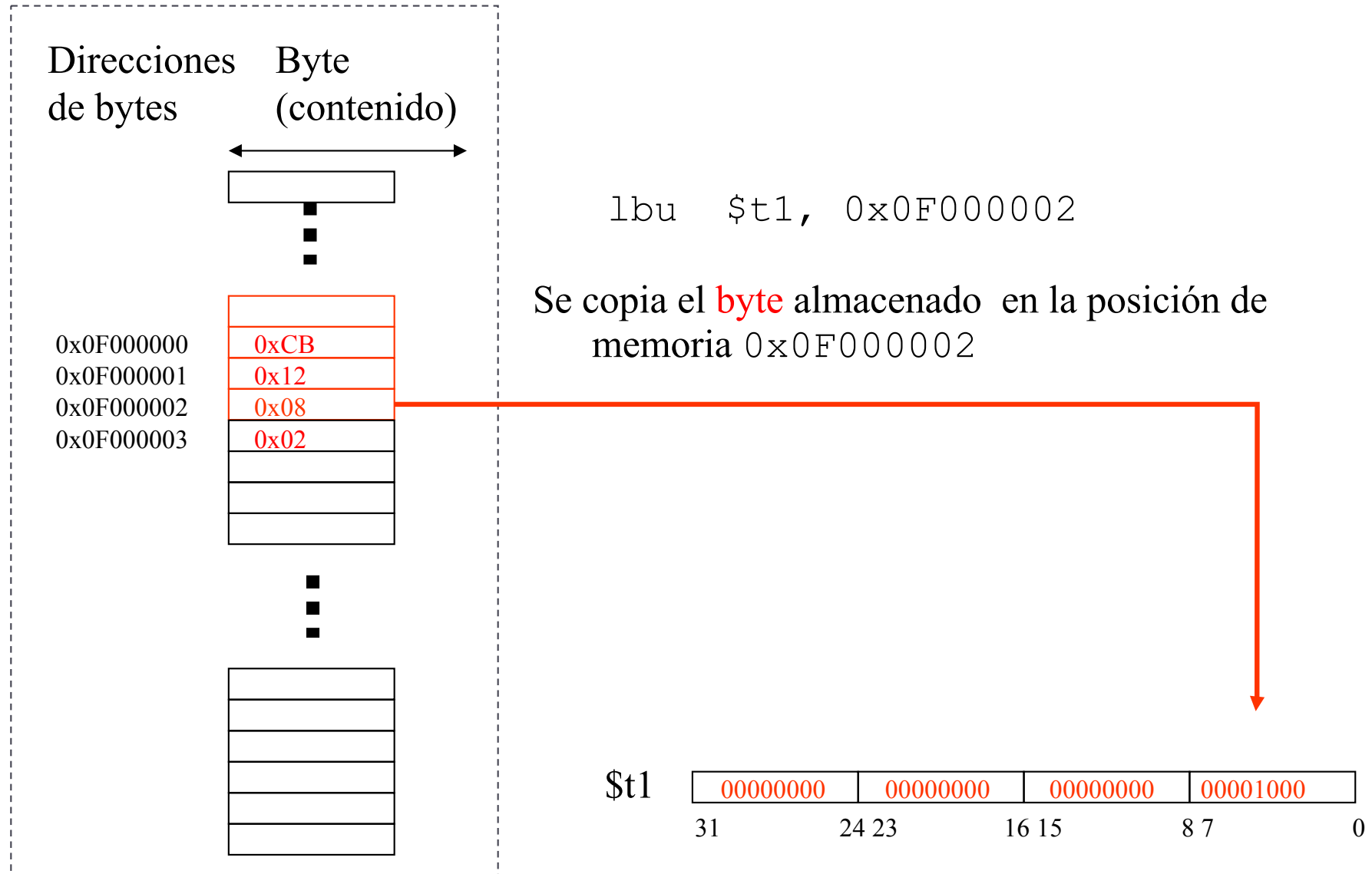
Acceso a bytes con lbu (unsigned)



```
lbu $t1, 0x0F000002
```

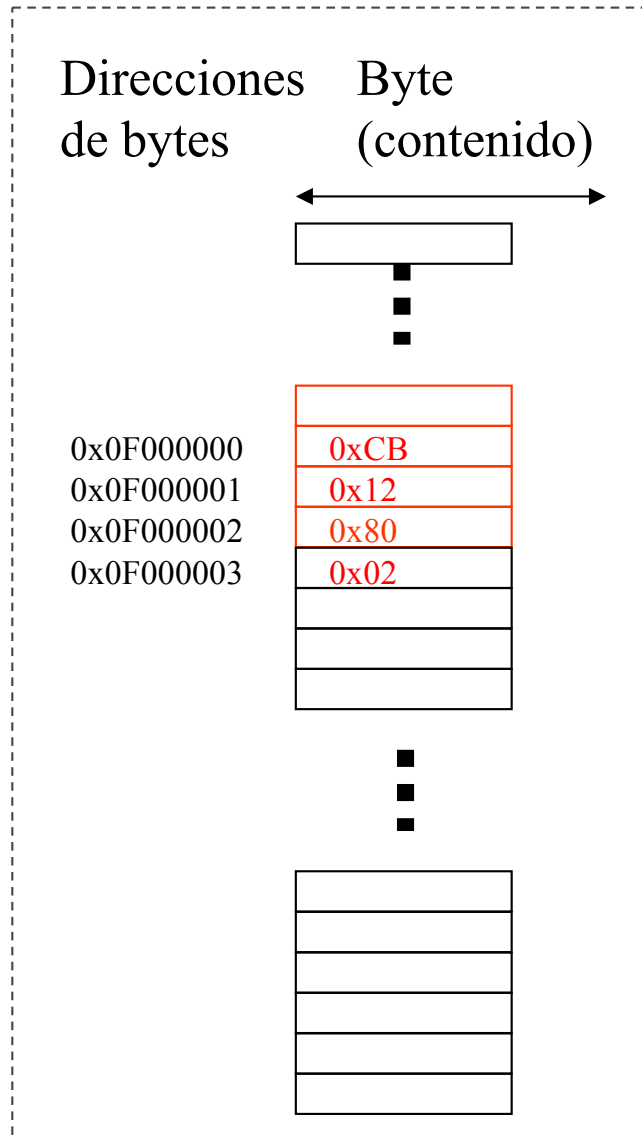


Acceso a bytes con lbu (unsigned)

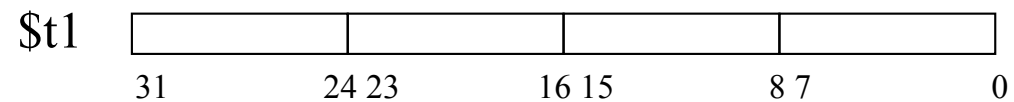


Acceso a bytes con lbu (unsigned)

No extiende el signo

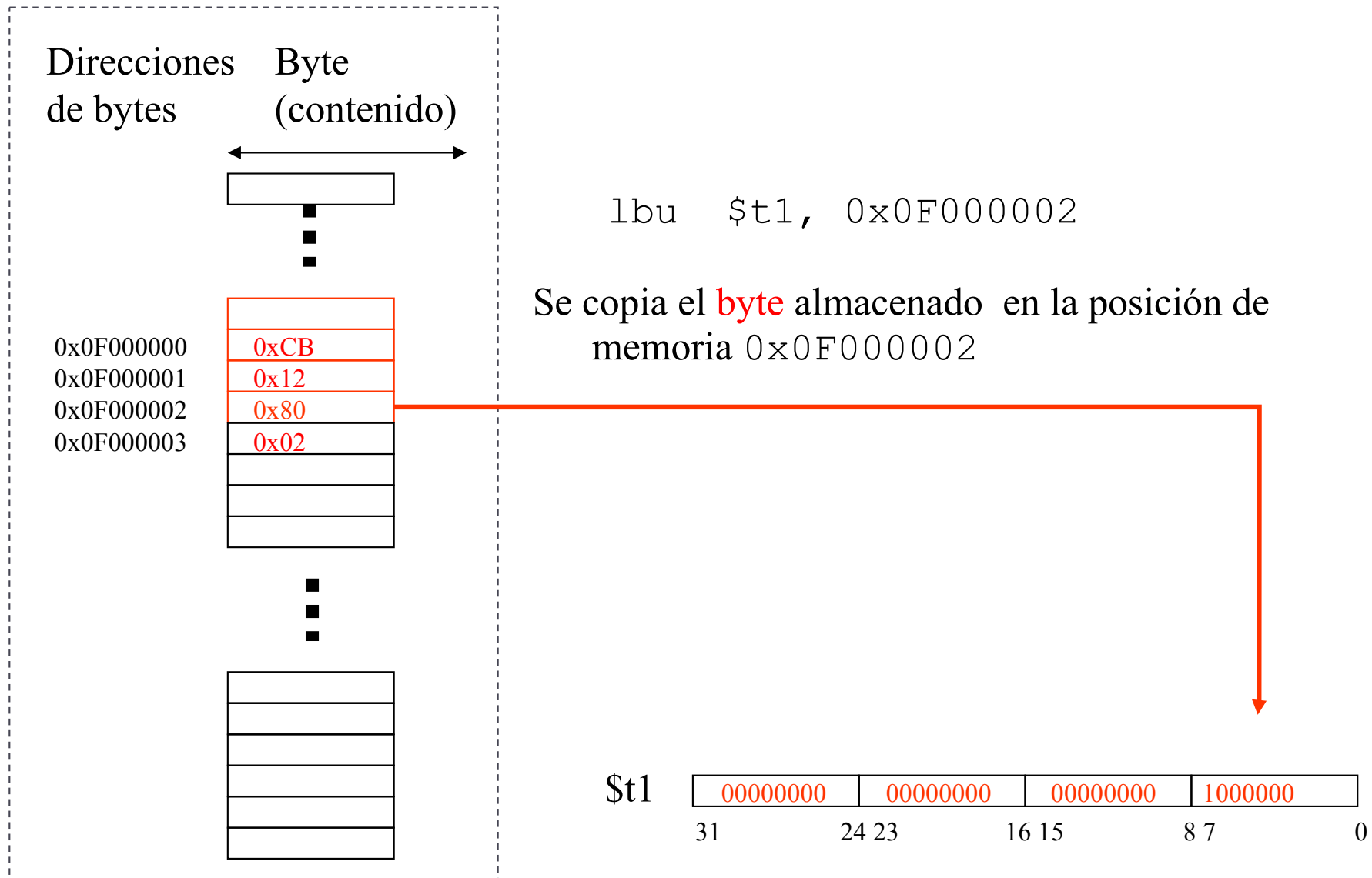


```
lbu $t1, 0x0F000002
```

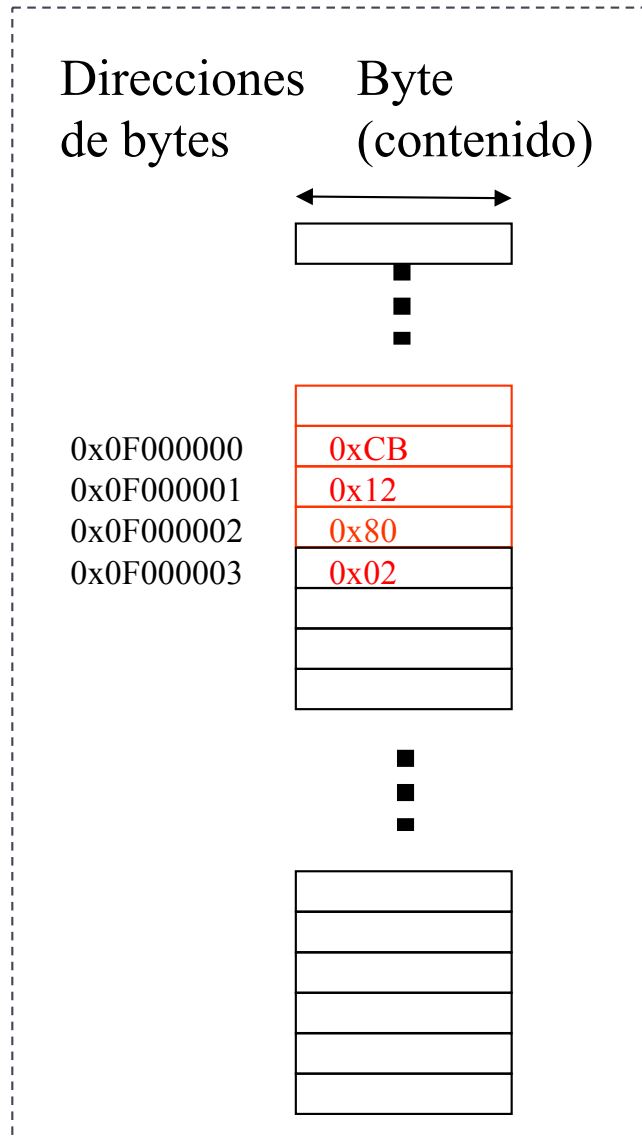


Acceso a bytes con lbu (unsigned)

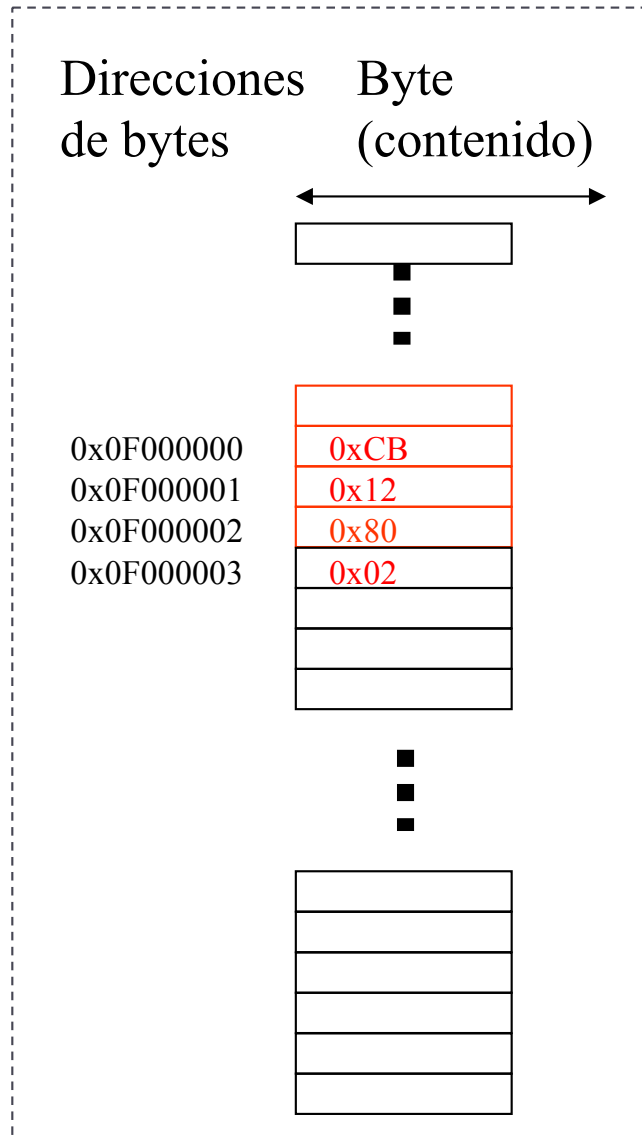
No extiende el signo



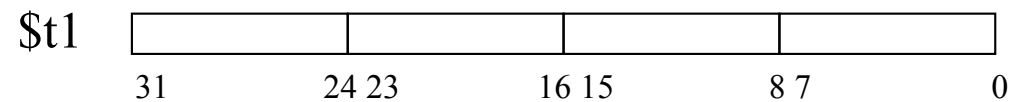
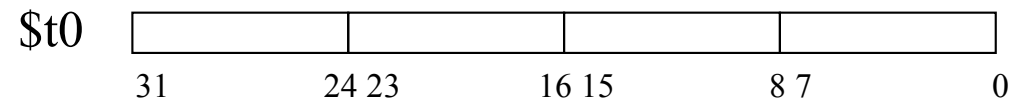
Ejemplos de uso la y lbu



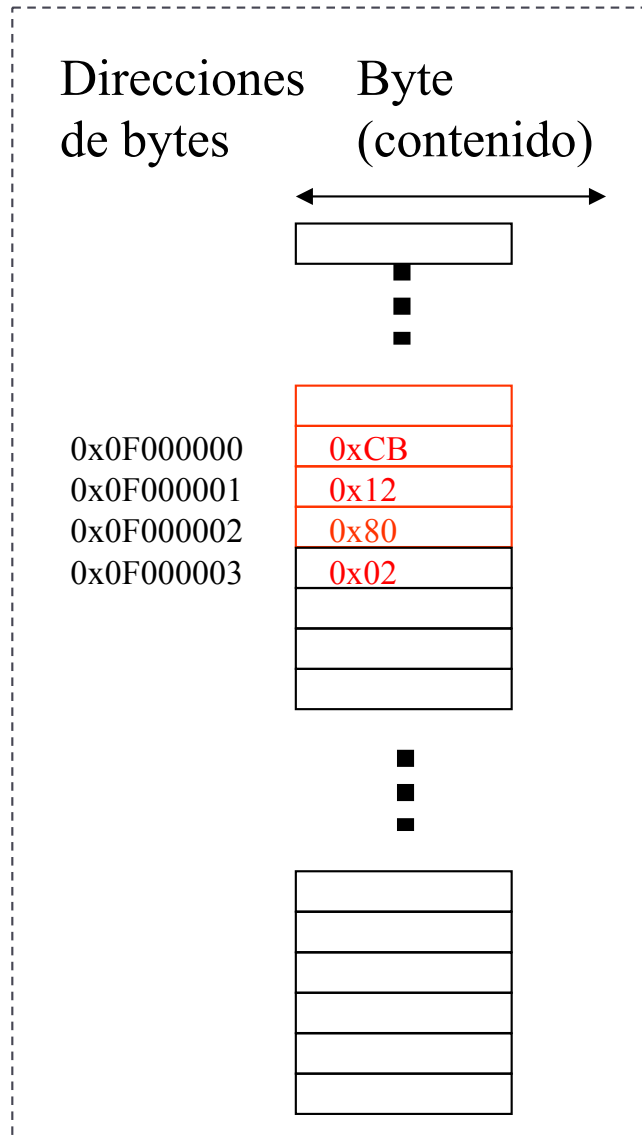
Ejemplos de uso la y lbu



```
la $t0, 0x0F000002
```

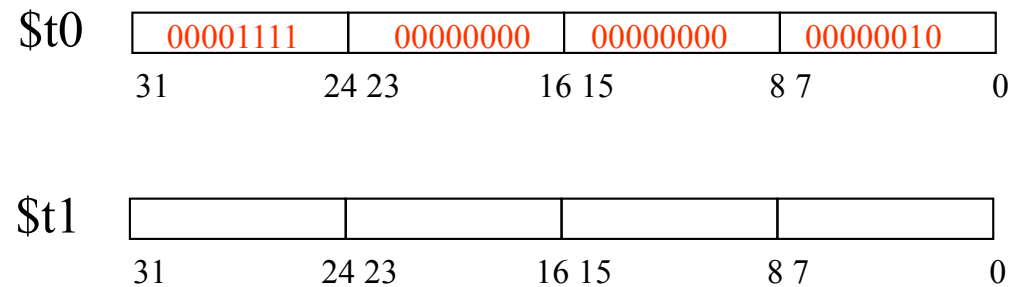


Ejemplos de uso la y lbu

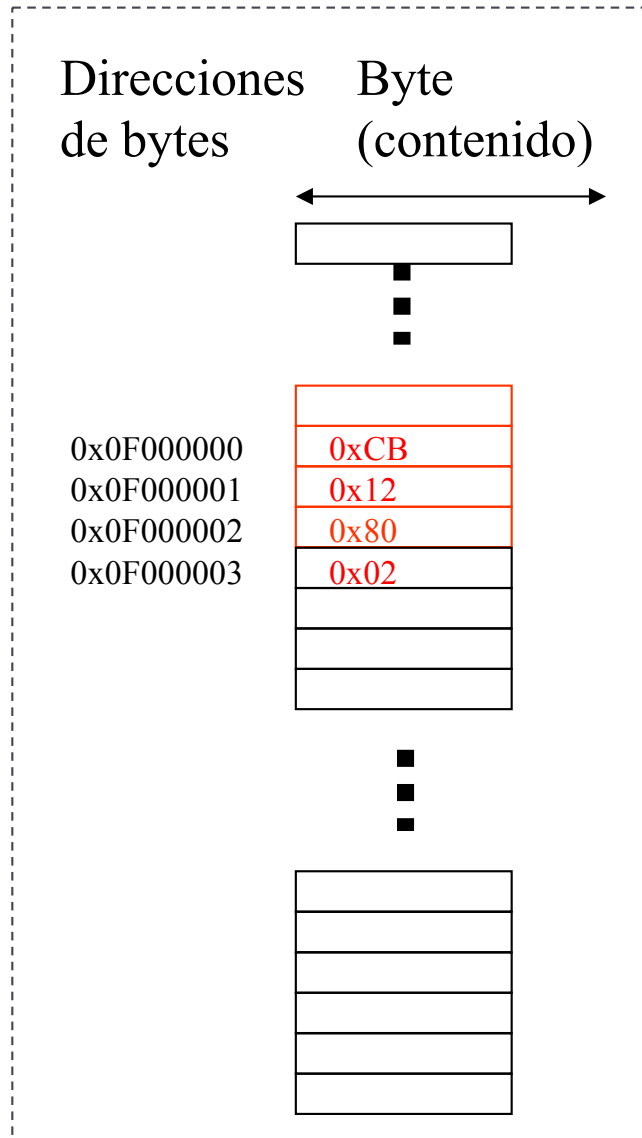


```
la $t0, 0x0F000002
```

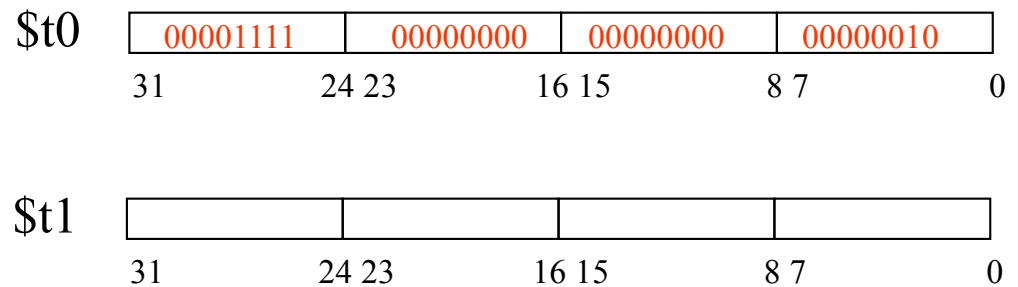
Se copia la dirección,
no el contenido



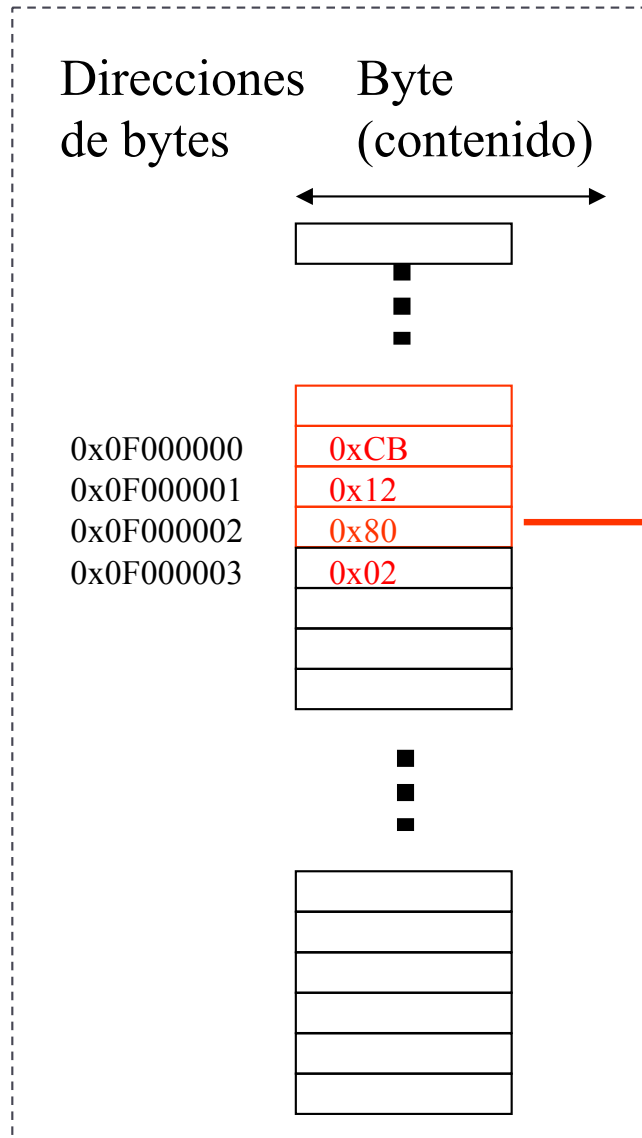
Ejemplos de uso la y lbu



```
la $t0, 0x0F000002  
lbu $t1, ($t0)
```



Ejemplos de uso la y lbu



```
la $t0, 0x0F000002
```

```
lbu $t1, ($t0)
```

Se copia el **byte** almacenado en la posición de memoria cuya dirección está almacenada en \$t0 sin extensión de signo (lbu)

\$t0

00001111	00000000	00000000	00000010
31	24 23	16 15	8 7

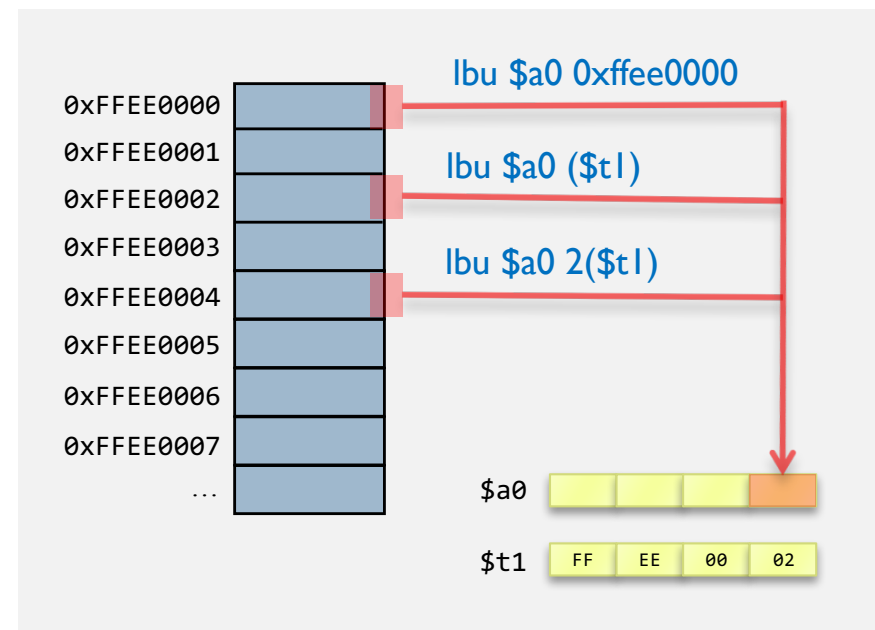
\$t1

00000000	00000000	00000000	10000000
31	24 23	16 15	8 7

Transferencia de datos

Direccionamiento

- ▶ Hay tres posibilidades a la hora de indicar una posición de memoria:
- ▶ A) Directo:
`lbu $a0 0x0FFEE0000`
- ▶ B) Indirecto a registro:
`lbu $a0 ($t1)`
- ▶ C) Relativo a registro:
`lbu $a0 2($t1)`

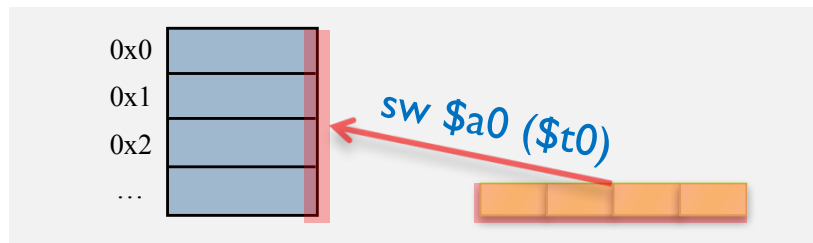
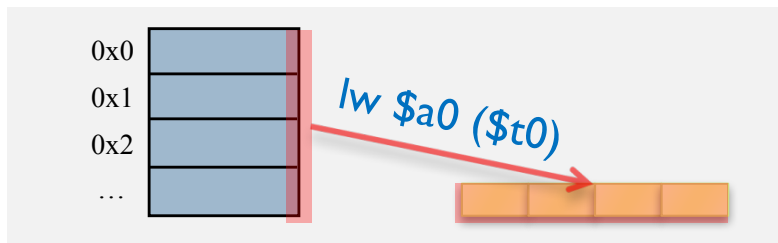
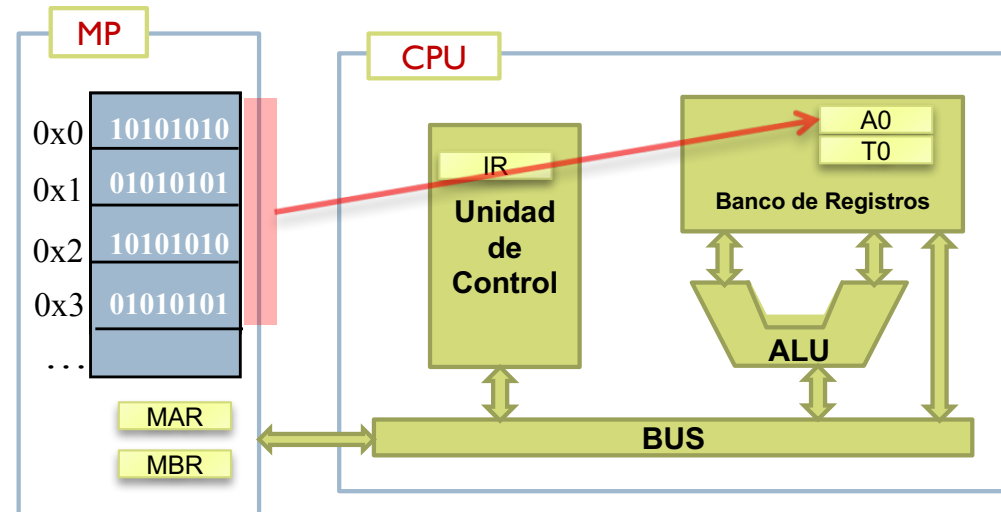


Transferencia de datos palabras

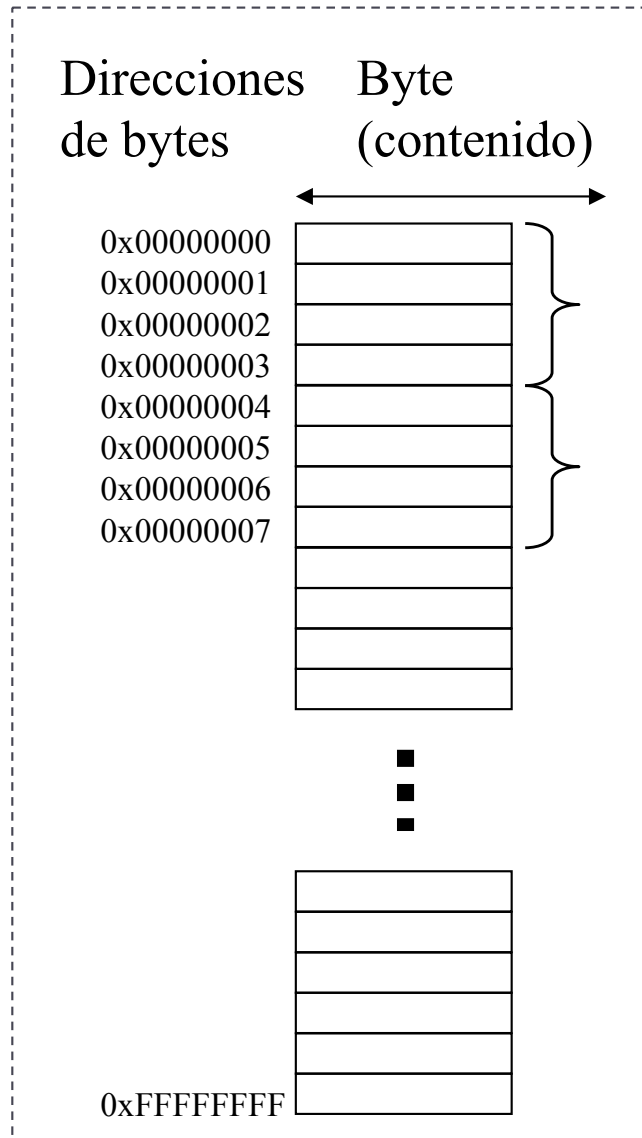
- ▶ Copia una **palabra** de **memoria** a un **registro** o viceversa

- ▶ Ejemplos:

- ▶ Memoria a registro
`lw $a0 ($t0)`
- ▶ Registro a memoria
`sw $a0 ($t0)`



Acceso a palabras



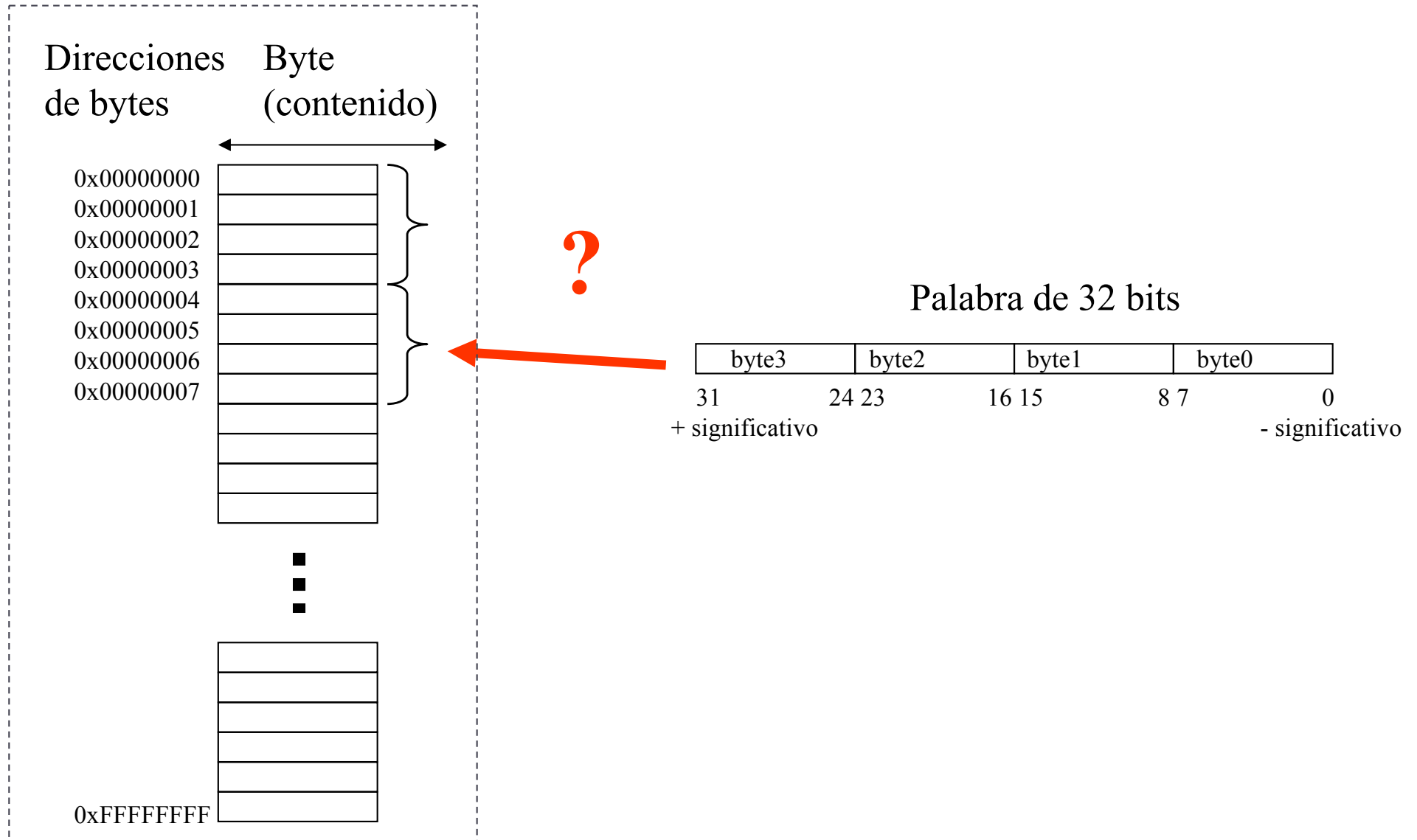
4 bytes forman una palabra

Palabra almacenada a partir del byte 0

Palabra almacenada a partir del byte 4

Las palabras (32 bits, 4 bytes) se almacenan utilizando cuatro posiciones consecutivas de memoria, comenzando la primera posición en una dirección múltiplo de 4

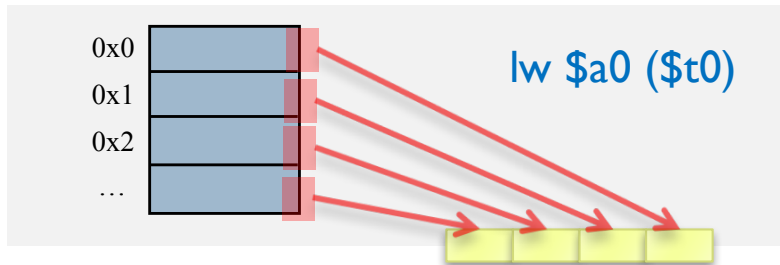
Acceso a palabras



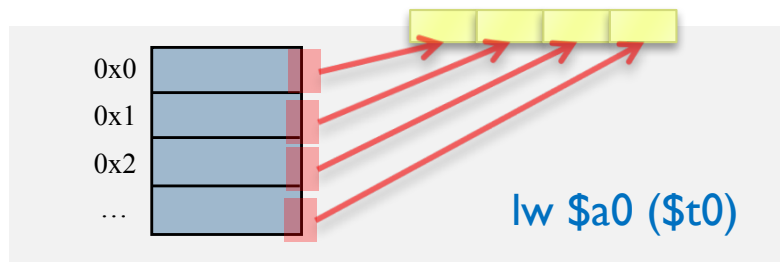
Transferencia de datos ordenamiento de bytes

- ▶ Hay dos tipos de ordenamiento de bytes:

- ▶ Little-endian (Dirección 'pequeña' termina la palabra...)

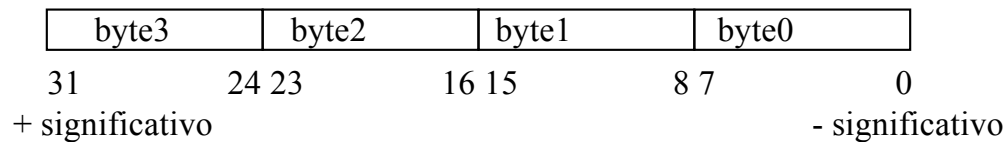


- ▶ Big-endian (Dirección 'grande' termina la palabra...)



Almacenamiento de palabras en la memoria

Palabra de 32 bits



A	byte3
A+1	byte2
A+2	byte1
A+3	byte0

BigEndian

A	byte0
A+1	byte1
A+2	byte2
A+3	byte3

LittleEndian

El número $27_{(10)} = 11011_{(2)} = 00000000000000000000000000011011$

A	00000000
A+1	00000000
A+2	00000000
A+3	00011011


BigEndian

A	00011011
A+1	00000000
A+2	00000000
A+3	00000000

LittleEndian

Problemas en la comunicación entre computadores con arquitectura distinta

El número $27_{(10)} = 11011_{(2)} = 00000000000000000000000000011011$



A	00000000
A+1	00000000
A+2	00000000
A+3	00011011

BigEndian

A	
A+1	
A+2	
A+3	

LittleEndian

Problemas en la comunicación entre computadores con arquitectura distinta

El número $27_{(10)} = 11011_{(2)} = 00000000000000000000000000011011$

A	00000000
A+1	00000000
A+2	00000000
A+3	00011011

BigEndian



Transmisión de
datos por la red

A	
A+1	
A+2	
A+3	

LittleEndian

Problemas en la comunicación entre computadores con arquitectura distinta

El número $27_{(10)} = 11011_{(2)} = 00000000000000000000000000011011$

A	00000000
A+1	00000000
A+2	00000000
A+3	00011011

BigEndian

A	00000000
A+1	00000000
A+2	00000000
A+3	00011011

LittleEndian



El número almacenado es: 00011011000000000000000000000000
que no es el 27

Ejemplo

endian.s

```
.data
```

```
    b1: .byte 0x00, 0x11, 0x22, 0x33
```

```
.text
```

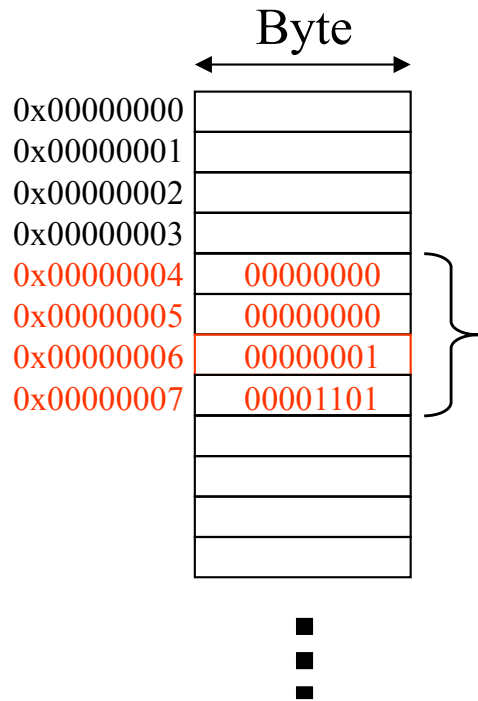
```
.globl main
```

```
main:
```

```
    lw    $t0 b1
```

Acceso a palabras

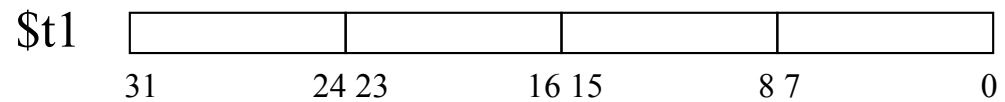
```
lw $t1, 0x4
```



Dirección 0x00000004 (000000.....00100)

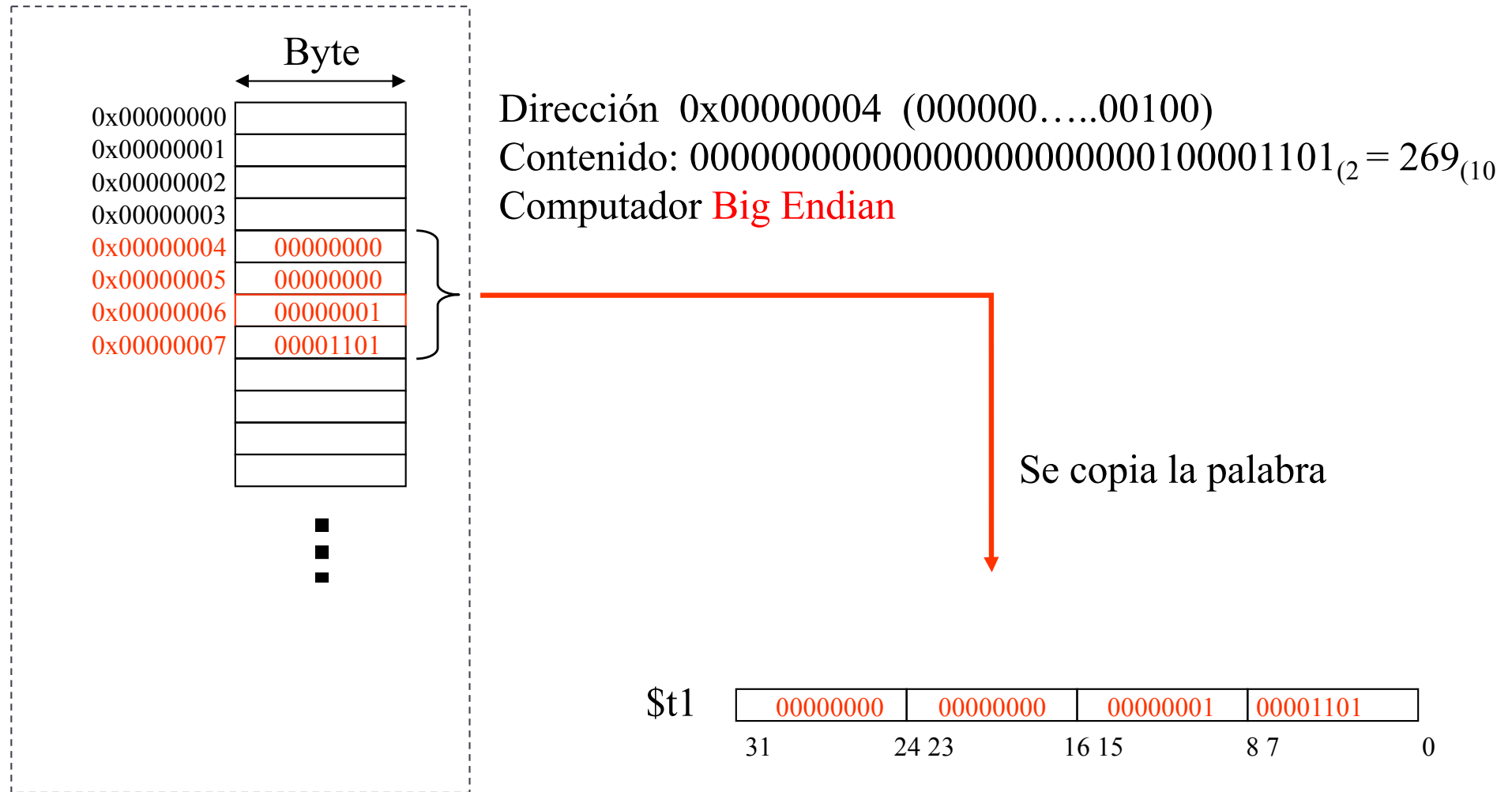
Contenido: $000000000000000000000000100001101_{(2)} = 269_{(10)}$

Computador Big Endian



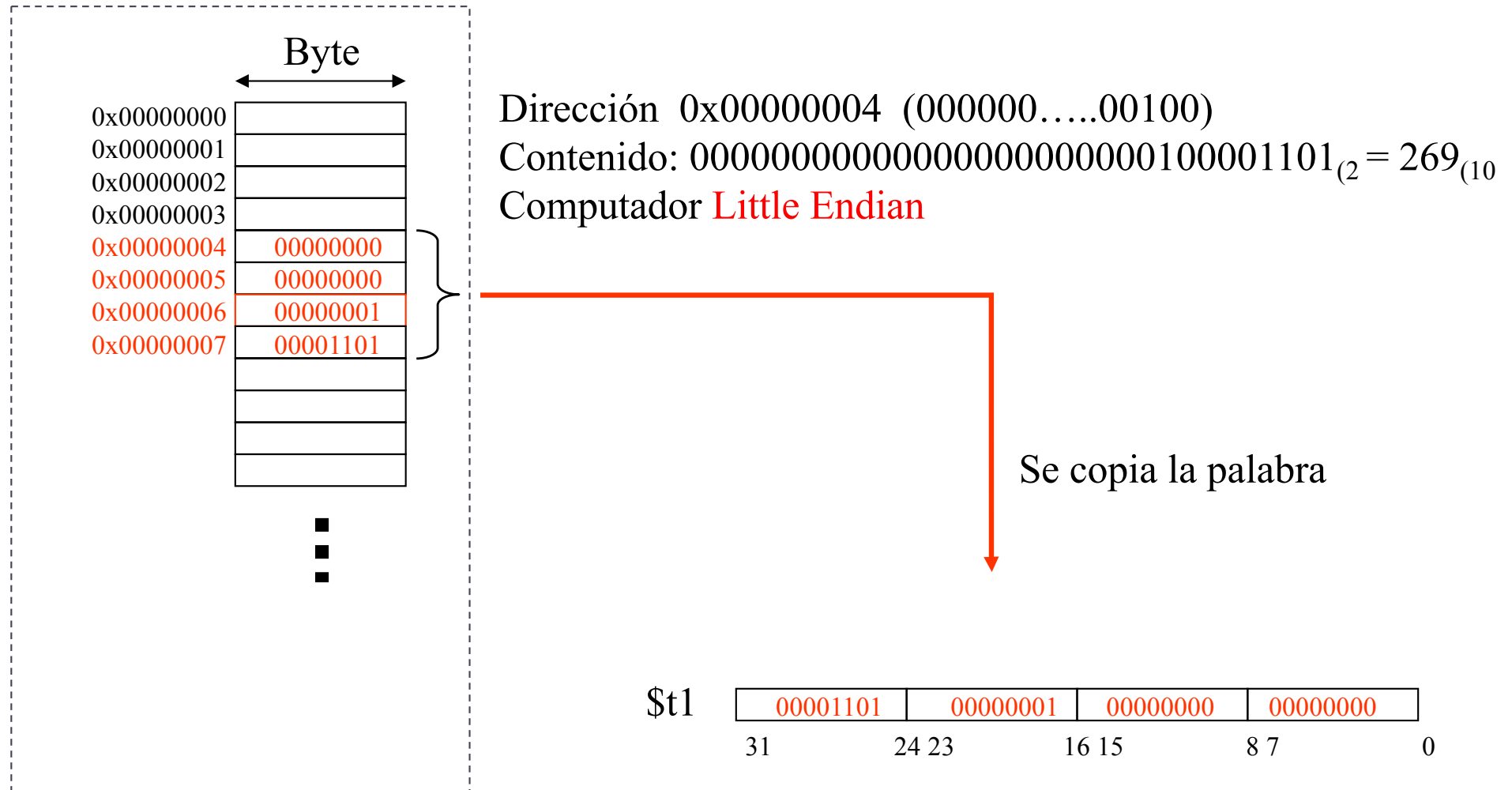
Acceso a palabras

```
lw $t1, 0x4
```



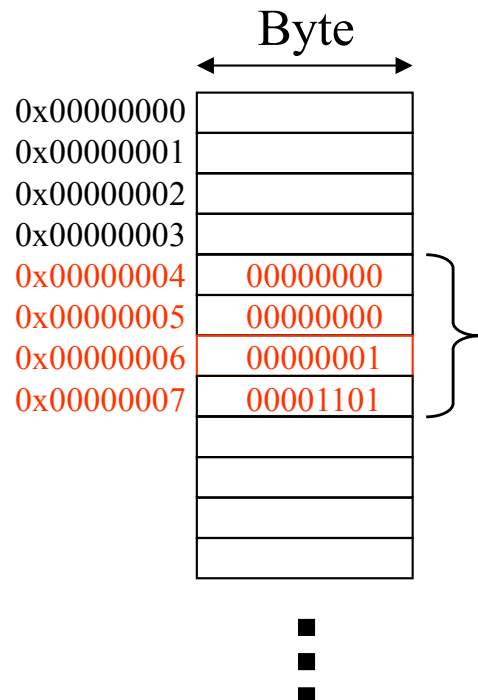
Acceso a palabras

```
lw $t1, 0x4
```



Acceso a palabras

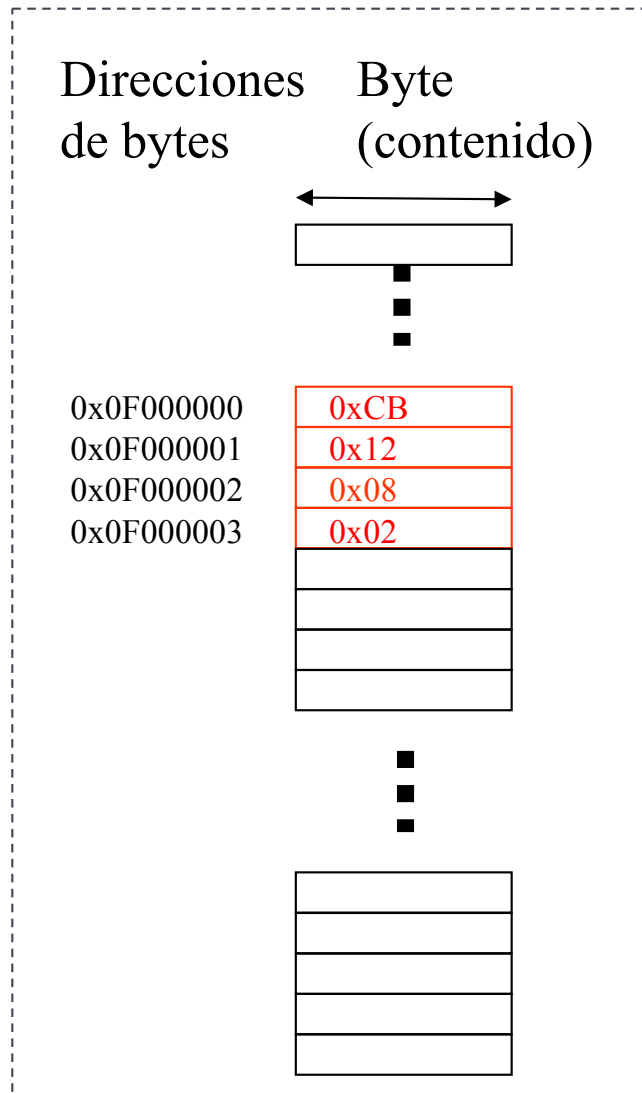
```
lw $t1, 0x4
```



Se especifica la dirección del primer byte
Acceso a la palabra almacenada a partir de la
dirección 0x00000004

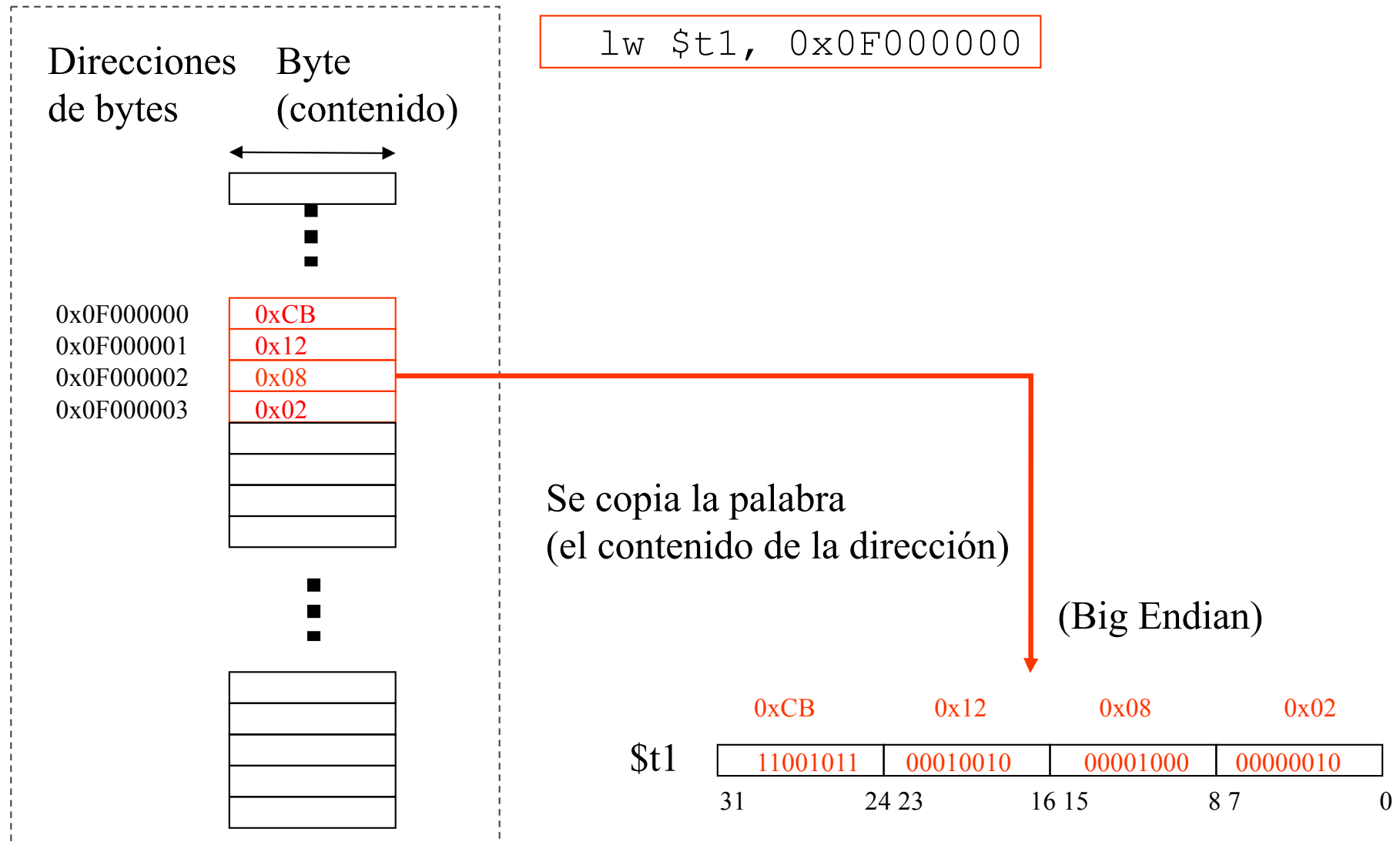


Diferencias entre lw, lb, lbu, la

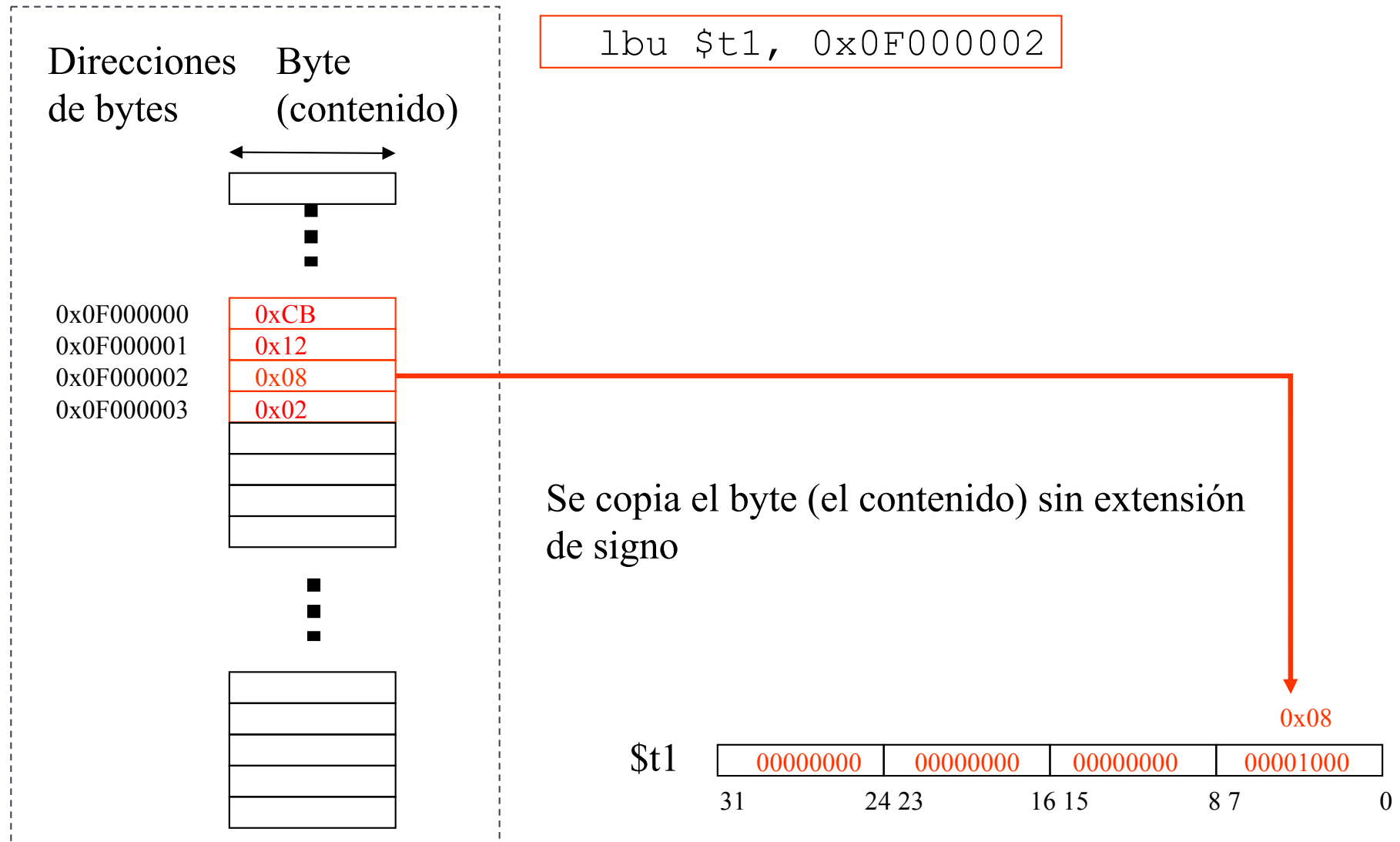


```
lw $t1, 0x0F000000
```

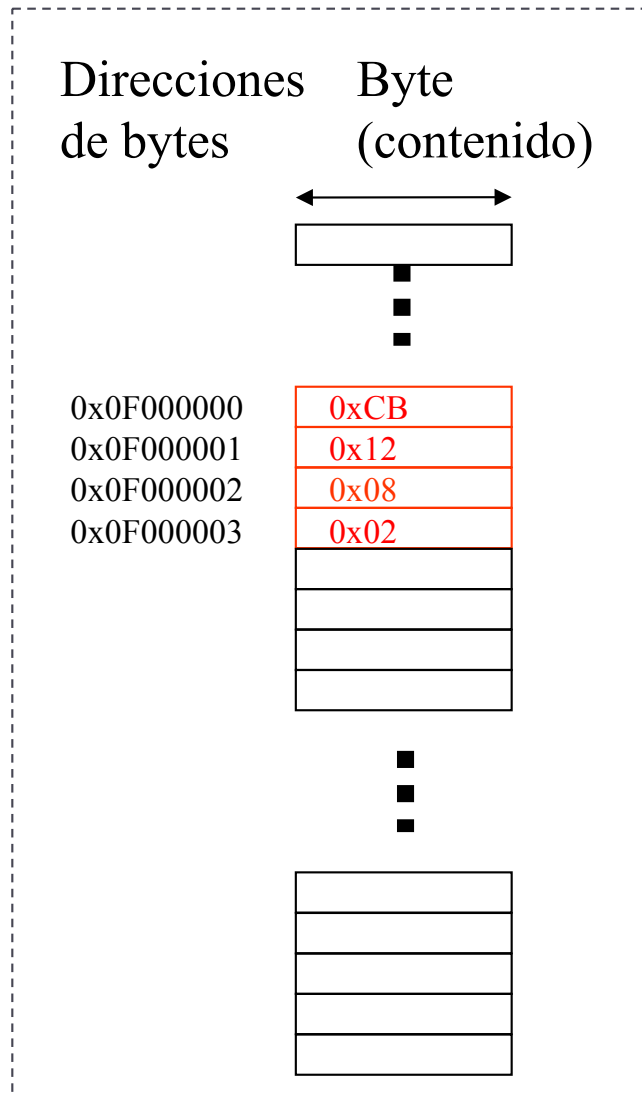
Diferencias entre lw, lb, lbu, la



Diferencias entre lw, lb, lbu, la

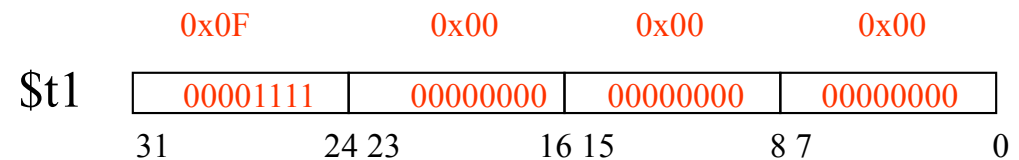


Diferencias entre lw, lb, lbu, la

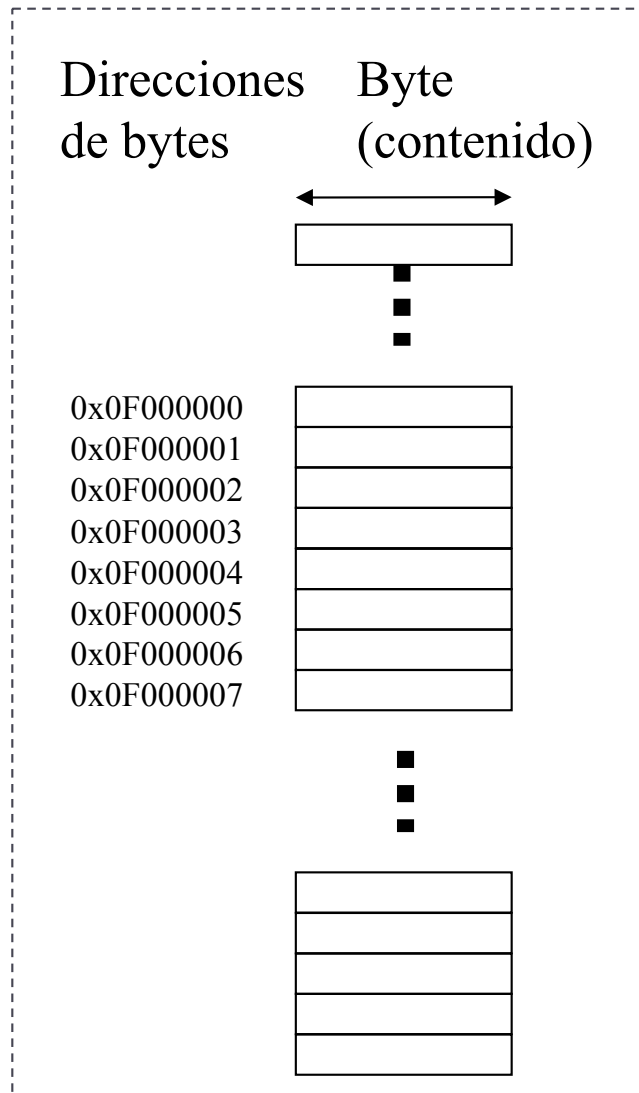


```
la $t1, 0x0F000000
```

Se copia la dirección,
no el contenido

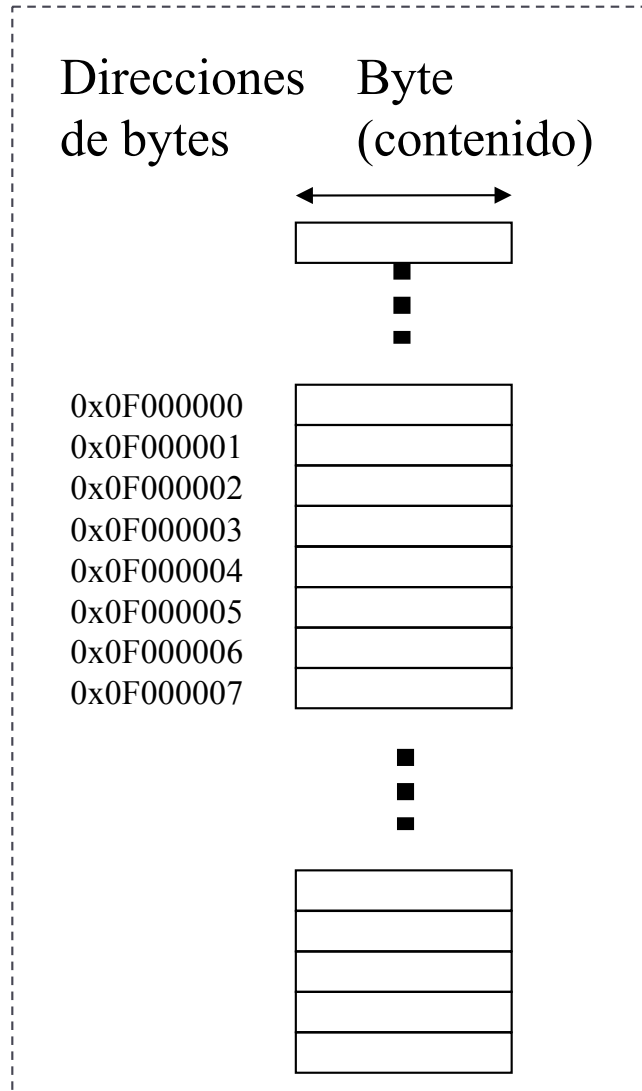


Ejemplo

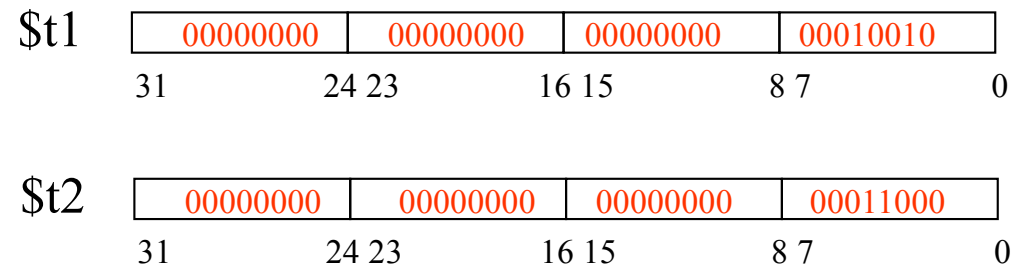


```
li $t1, 18  
li $t2, 24
```

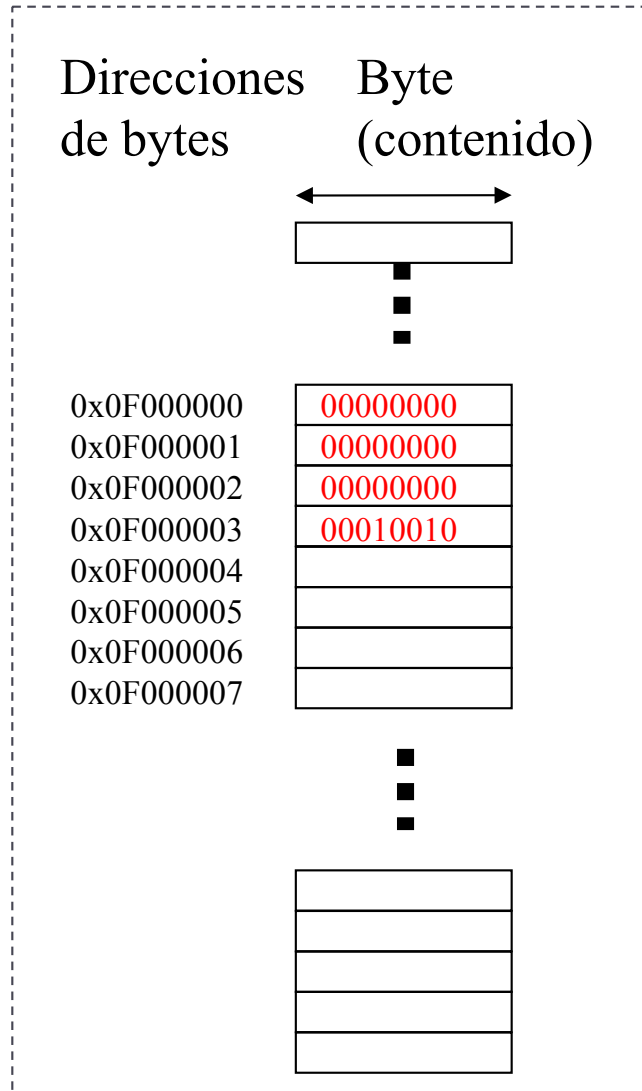
Ejemplo



```
li $t1, 18  
li $t2, 24
```

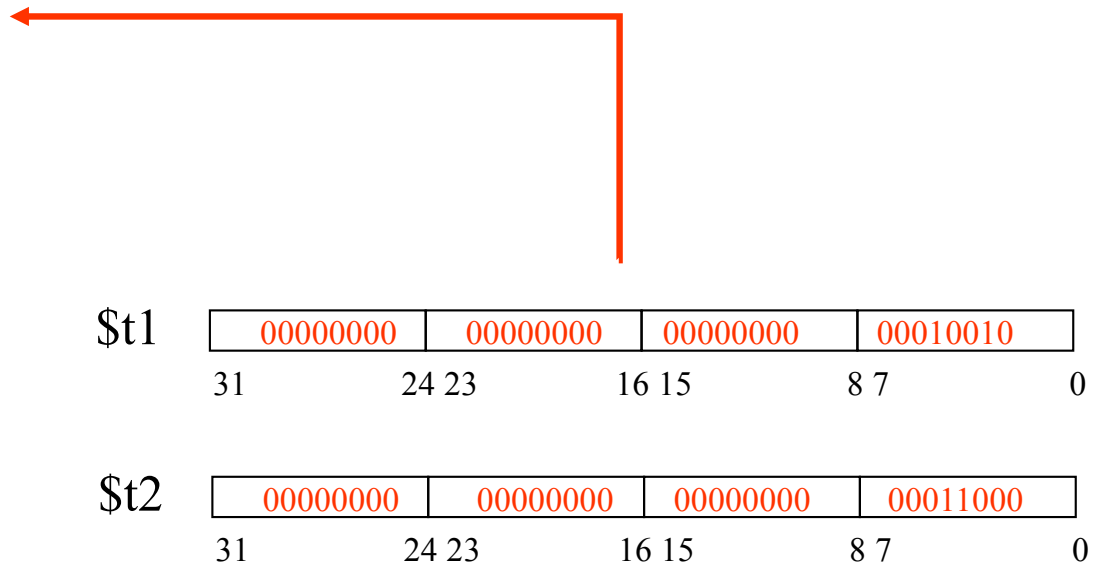


Escritura en memoria de una palabra

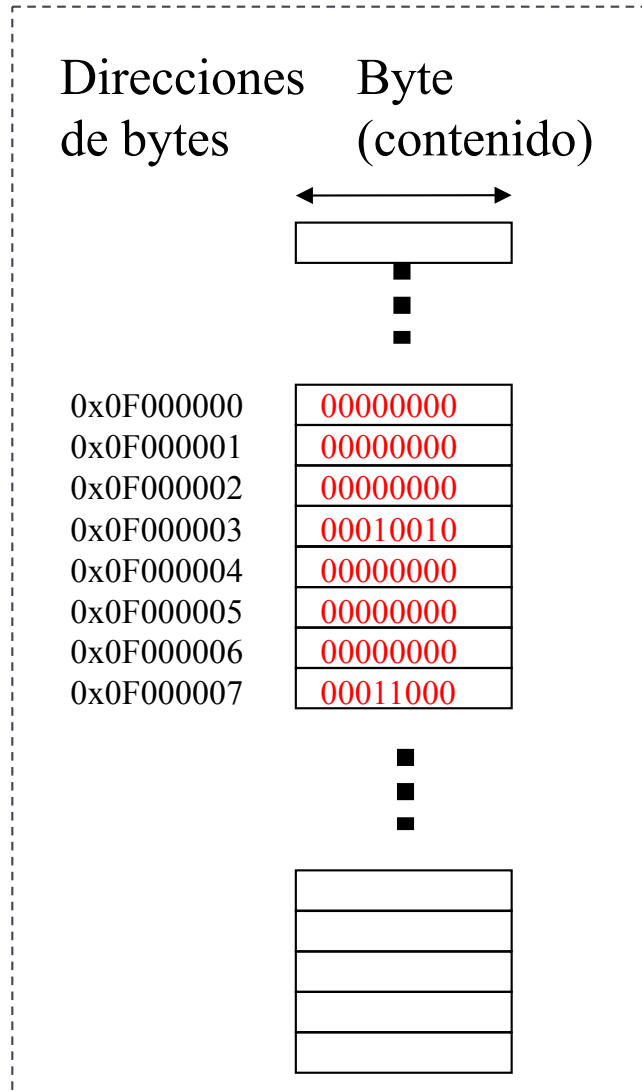


```
sw $t1, 0x0F000000
```

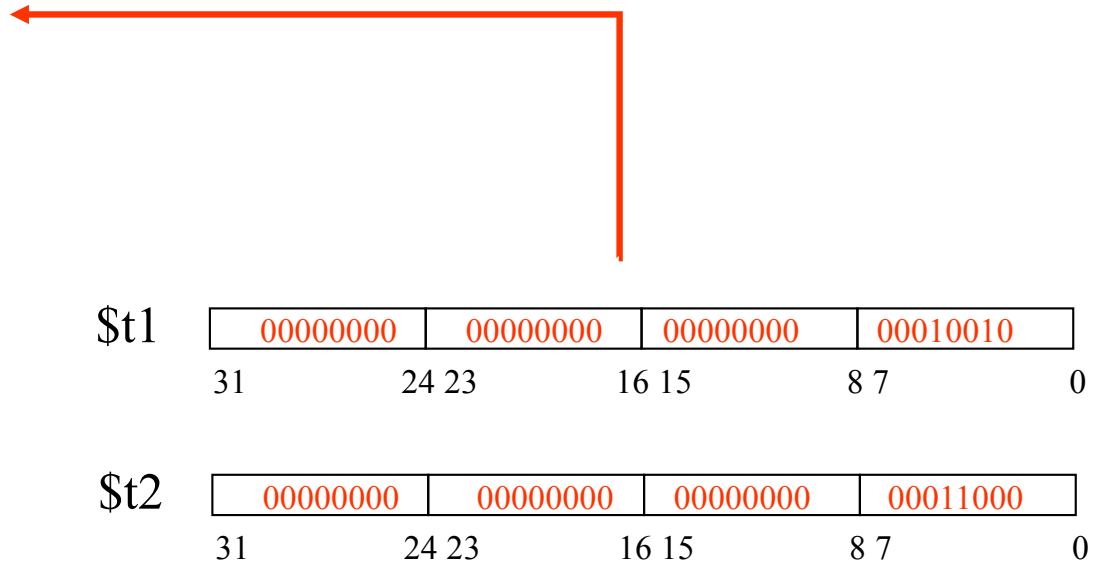
Escribe el contenido de un registro en memoria (la palabra completa)



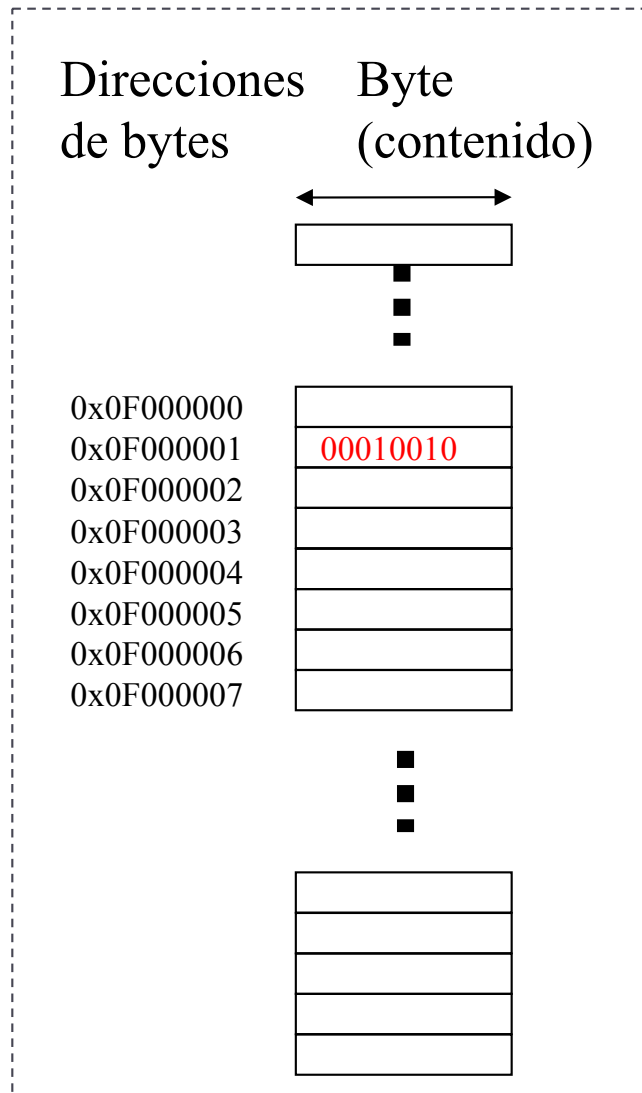
Escritura en memoria de una palabra



```
sw $t1, 0x0F000000  
sw $t2, 0x0F000004
```

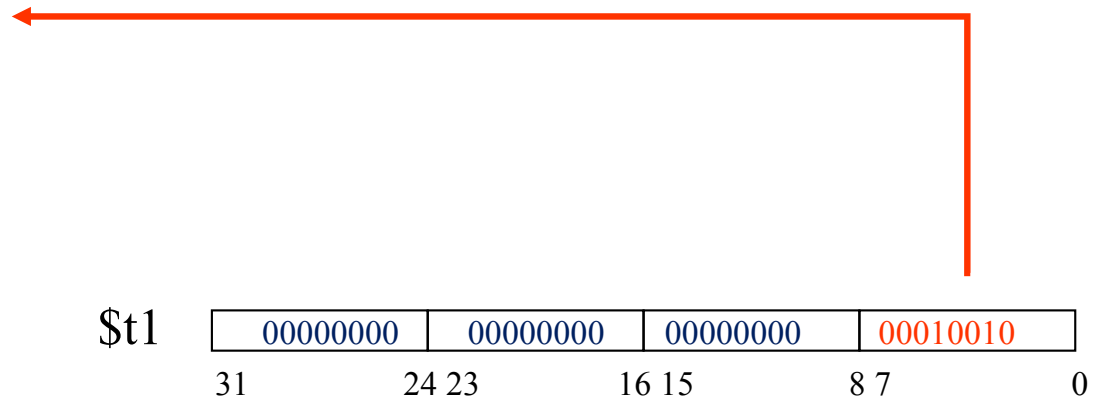


Escritura en memoria de un byte



```
sb $t1, 0x0F000001
```

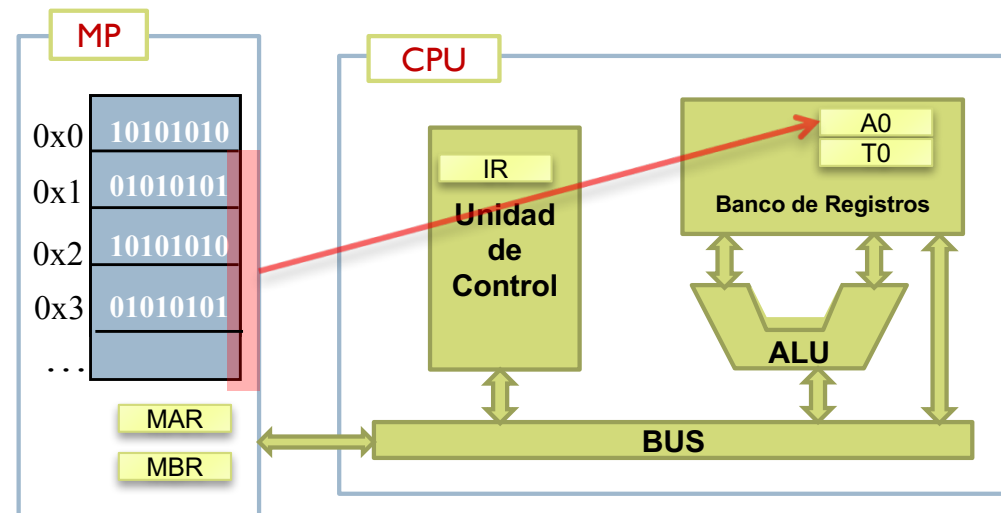
Escribe el contenido del byte **menos significativo** del registro \$t1 en memoria



Transferencia de datos alineamiento y tamaño de acceso

► Peculiaridades:

- Alineamiento de los elementos en memoria
- Tamaño de acceso por defecto

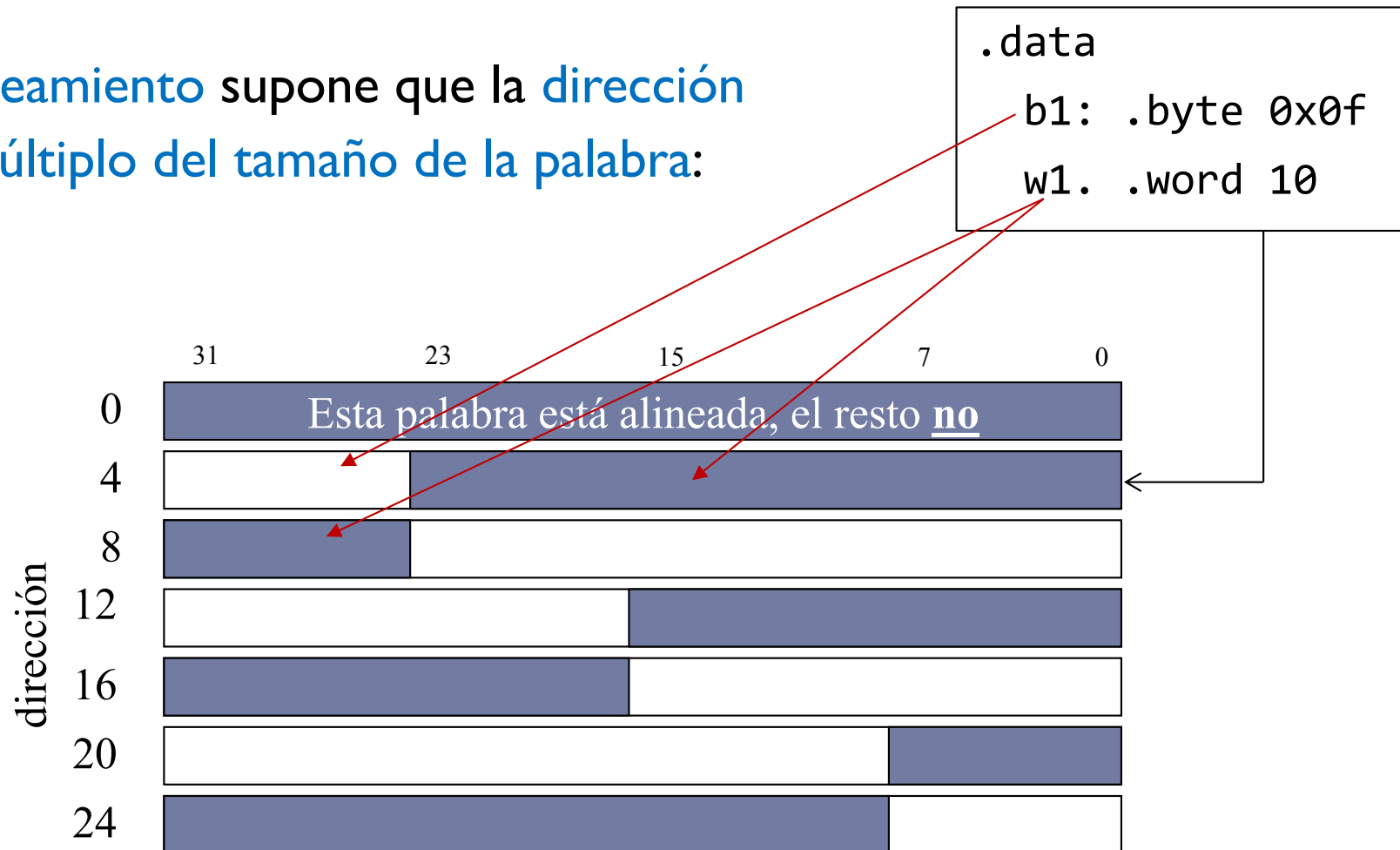


Alineación de datos

- ▶ En general:
 - ▶ Un dato que ocupa K bytes está alineado cuando la dirección D utilizada para accederlo cumple que:
$$D \bmod K = 0$$
- ▶ La alineación supone que:
 - ▶ Los datos que ocupan 2 bytes se encuentran en direcciones pares
 - ▶ Los datos que ocupan 4 bytes se encuentran en direcciones múltiplo de 4
 - ▶ Los datos que ocupan 8 bytes (double) se encuentran en direcciones múltiplo de 8

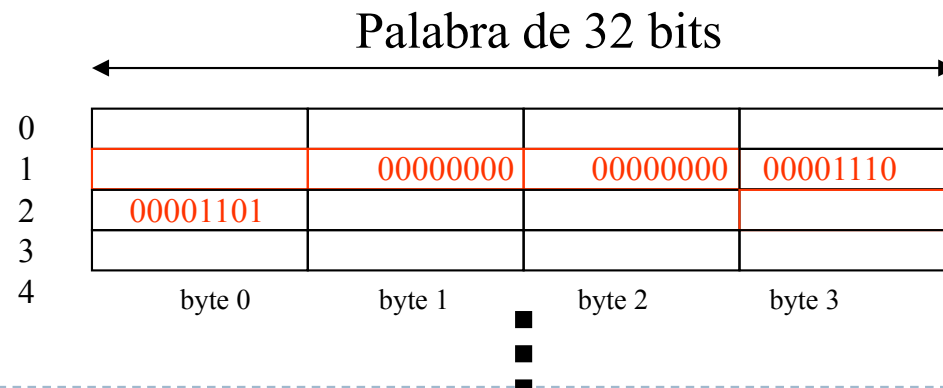
Alineamiento

- El **alineamiento** supone que la **dirección** sea **múltiplo del tamaño de la palabra**:



Alineación de datos

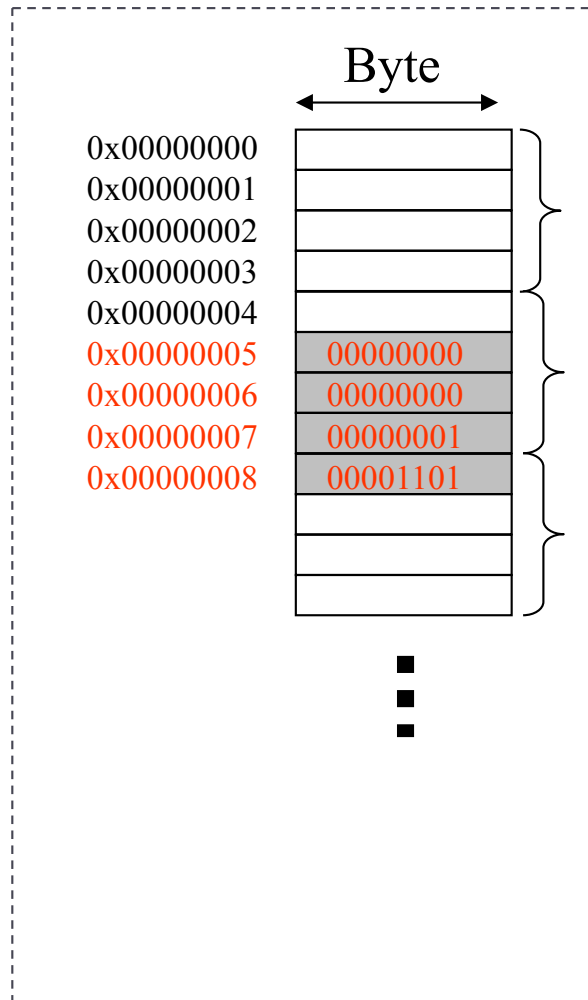
- ▶ En general los computadores no permiten el acceso a datos no alineados
 - ▶ Objetivo: minimizar el número de accesos a memoria
 - ▶ El compilador se encarga de asignar a los datos las direcciones adecuadas
- ▶ Algunas arquitecturas como Intel permiten el acceso a datos no alineados
 - ▶ El acceso a un dato no alineado implica varios accesos a memoria



Datos no alineados

```
lw $t1, 0x05
```

????

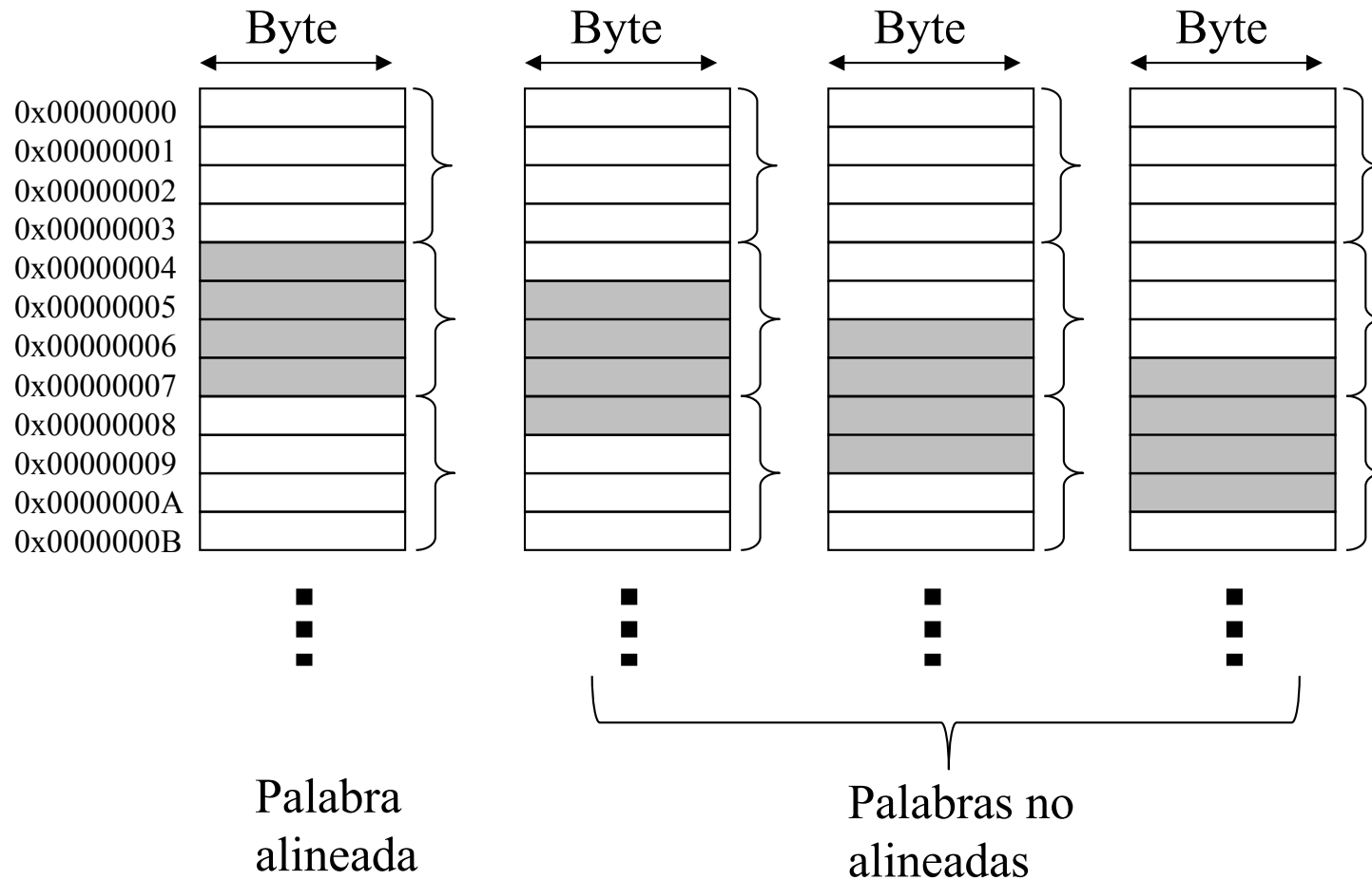


Palabras de memoria

La palabra que está almacenada a partir de la dirección 0x05 **no está alineada** porque se encuentra en dos palabras de memoria distintas

Una palabra tiene que almacenarse a partir de una dirección múltiplo de 4

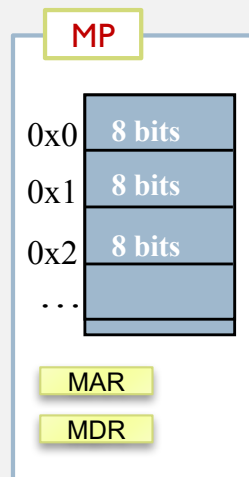
Datos no alineados



Direccionamiento a nivel de palabra o de byte

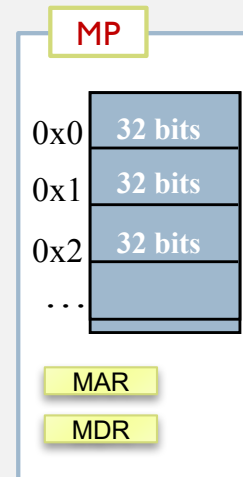
- ▶ La **memoria principal** es similar a un gran vector de una dimensión
- ▶ Una **dirección de memoria** es el índice del vector
- ▶ Hay dos **tipos de direccionamiento**:

- ▶ Direccionamiento por bytes



- ▶ Cada elemento de la memoria es un **byte**
- ▶ Transferir **una palabra** supone transferir **4 bytes**

- ▶ Direccionamiento por palabras



- ▶ Cada elemento de la memoria es una **palabra**
- ▶ **lb** supone **transferir una palabra** y **quedarse con un byte**

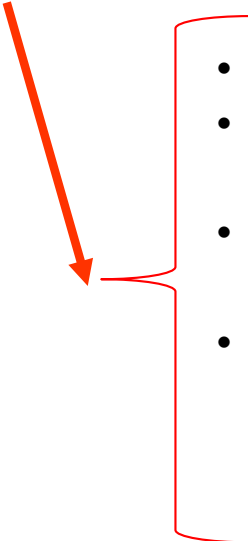
Resumen

- ▶ Un programa para poder ejecutarse debe estar cargado junto con sus datos en memoria
- ▶ Todas las instrucciones y los datos se almacenan en memoria, por tanto todo tiene una dirección de memoria
 - ▶ Las instrucciones y los datos
- ▶ En un computador como el MIPS 32 (de 32 bits)
 - ▶ Los registros son de 32 bits
 - ▶ En la memoria se pueden almacenar bytes (8 bits)
 - ▶ Instrucciones memoria → registro: `lb, lbu, sb`
 - ▶ Instrucciones registro → memoria: `sb`
 - ▶ En la memoria se pueden almacenar palabras (32 bits)
 - ▶ Instrucción memoria → registro: `lw`
 - ▶ Instrucción registro → memoria: `sw`

Formatos de las instrucciones de acceso a memoria

lw
sw
lb
sb
lbu

Registro, dirección de memoria

- 
- Número que representa una dirección
 - Etiqueta simbólica que representa una dirección
 - (registro): representa la dirección almacenada en el registro
 - num(registro): representa la dirección que se obtiene de sumar num con la dirección almacenada en el registro

Formatos de las instrucciones de acceso a memoria

- ▶ `lbu $t0, 0x0F000002`
 - ▶ Direccionamiento directo. Se carga en `$t0` el byte almacenado en la posición de memoria `0x0F000002`
- ▶ `lbu $t0, etiqueta`
 - ▶ Direccionamiento directo. Se carga en `$t0` el byte almacenado en la posición de memoria `etiqueta`
- ▶ `lbu $t0, ($t1)`
 - ▶ Direccionamiento indirecto de registro. Se carga en `$t0` el byte almacenado en la posición de memoria almacenada en `$t1`
- ▶ `lbu $t0, 80($t1)`
 - ▶ Direccionamiento relativo. Se carga en `$t0` el byte almacenado en la posición de memoria que se obtiene de sumar el contenido de `$t1` con 80

Instrucciones de escritura en memoria

- ▶ `sw $t0, 0x0F000000`
 - ▶ Copia la palabra almacenada en \$t0 en la dirección 0x0F000000
- ▶ `sb $t0, 0x0F000000`
 - ▶ Copia el byte almacenado en \$t0 (el menos significativo) en la dirección 0x0F000000

Tipos de datos en ensamblador

- ▶ Booleanos
- ▶ Caracteres
- ▶ Enteros
- ▶ Reales
- ▶ Vectores
- ▶ Cadenas de caracteres
- ▶ Matrices
- ▶ Otras estructuras

Tipos de datos booleanos

```
bool_t b1 = false;  
bool_t b2 = true;  
...
```

```
main ()  
{  
    b1 = true ;  
    ...  
}
```

```
.data
```

```
b1: .byte 0          # 1 byte
```

```
b2: .byte 1
```

```
...
```

```
.text
```

```
.globl main
```

```
main:  la $t0, b1
```

```
       li $t1, 1
```

```
       sb $t1, ($t0)
```

```
...
```

Tipos de datos caracteres

```
char c1 ;  
char c2 = 'a' ;
```

...

```
main ()
```

```
{
```

```
    c1 = c2;
```

...

```
}
```

```
.data
```

```
c1: .space 1      # 1 byte
```

```
c2: .byte 'a'
```

...

```
.text
```

```
.globl main
```

```
main:  la  $t0 c2
```

```
        lbu $t1 c1
```

```
        sb  $t1 ($t0)
```

...

Tipos de datos enteros

```
int  resultado ;  
int  op1 = 100 ;  
int  op2 = -10 ;  
...
```

```
main ()  
{  
    resultado = op1+op2;  
    ...  
}
```

```
.data
```

```
.align 2
```

```
resultado: .space 4 # 4 bytes
```

```
op1:      .word 100
```

```
op2:      .word -10
```

```
...
```

```
.text
```

```
.globl main
```

```
main:  lw $t1 op1  
       lw $t2 op2  
       add $t3 $t1 $t2  
       la $t4 resultado  
       sw $t3 ($t4)
```

```
...
```

Tipos de datos enteros

variable global sin valor inicial

```
int  resultado ;  
int  op1 = 100 ;  
int  op2 = -10 ;  
...
```

variable global con valor inicial

```
main ()  
{  
    resultado = op1+op2;  
    ...  
}
```

```
.data  
.align 2  
resultado: .space 4 # 4 bytes  
op1:      .word 100  
op2:      .word -10  
...
```

```
.text  
.globl main  
main:  lw $t1 op1  
       lw $t2 op2  
       add $t3 $t1 $t2  
       la $t4 resultado  
       sw $t3 ($t4)  
...
```

Ejercicio

- Indique un fragmento de código en ensamblador con la misma funcionalidad que:

```
int  b;  
int  a = 100 ;  
int  c = 5  ;  
int  d;  
main ()  
{  
    d = 80;  
    b = -(a+b*c+a) ;  
}
```

Asumiendo que a, b, c y d son variables que residen en memoria

Tipo de datos básicos

float

```
float  resultado ;  
float  op1 = 100 ;  
float  op2 = 2.5
```

...

```
main ()  
{  
    resultado = op1 + op2 ;  
    ...  
}
```

.data

.align 2

```
resultado:  .space 4 # 4 bytes  
op1:        .float 100  
op2:        .float 2.5
```

...

.text

.globl main

```
main:  l.s      $f0 op1  
       l.s      $f1 op2  
       add.s    $f3 $f1 $f2  
       s.s      $f3 resultado  
       ...
```

Tipo de datos básicos

double

```
double resultado ;  
double op1 = 100 ;  
double op2 = -10.27 ;
```

...

```
main ()  
{  
    resultado = op1 * op2 ;  
    ...  
}
```

.data

.align 2

```
resultado:  .space 4  
op1:        .double 100  
op2:        .double -10.27
```

...

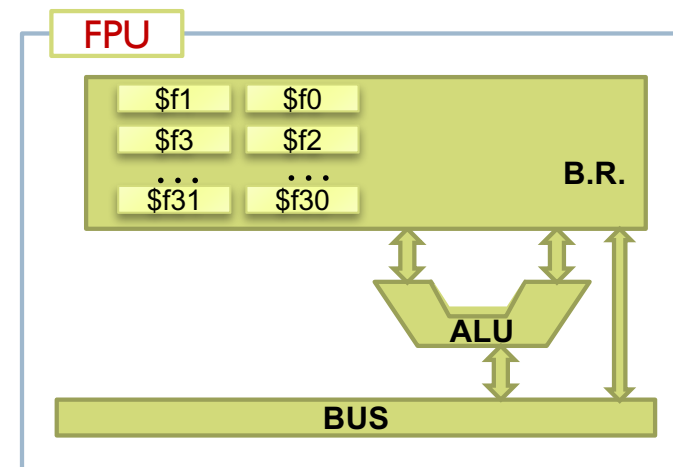
.text

.globl main

```
main:  l.d      $f0 op1      # ($f0,$f1)  
       l.d      $f2 op2      # ($f2,$f3)  
       mul.d    $f6 $f0 $f2  
       s.d      $f6 resultado  
       ...
```

Banco de registros de coma flotante

- ▶ El coprocesador I tiene 32 registros de 32 bits (4 bytes) cada uno
 - ▶ Es posible trabajar con simple o doble precisión
- ▶ Simple precisión (32 bits):
 - ▶ Del \$f0 al \$f31
 - ▶ Ej.: `add.s $f0 $f1 $f5`
 $f0 = f1 + f5$
 - ▶ Otras operaciones:
 - ▶ `add.s, sub.s, mul.s, div.s, abs.s`
- ▶ Doble precisión (64 bits):
 - ▶ Se utilizan por parejas
 - ▶ Ej.: `add.d $f0 $f2 $f8`
 $(f0, f1) = (f2, f3) + (f8, f9)$
 - ▶ Otras operaciones:
 - ▶ `add.d, sub.d, mul.d, div.d, abs.d`



Transferencia de datos

IEEE 754

- ▶ Copia un **número** de **memoria** a un **registro** o viceversa

- ▶ Instrucciones:

- ▶ Memoria a registro

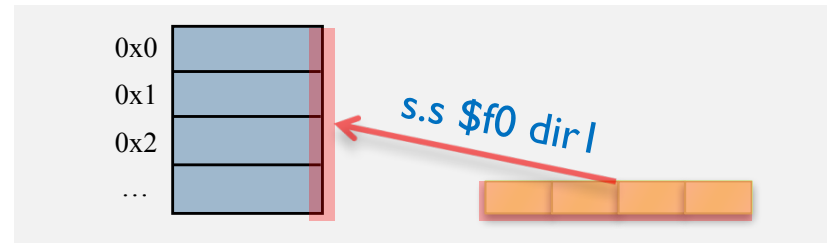
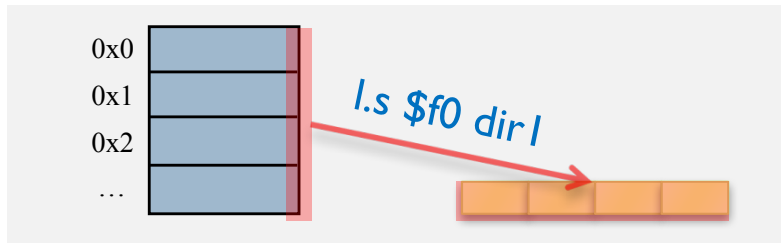
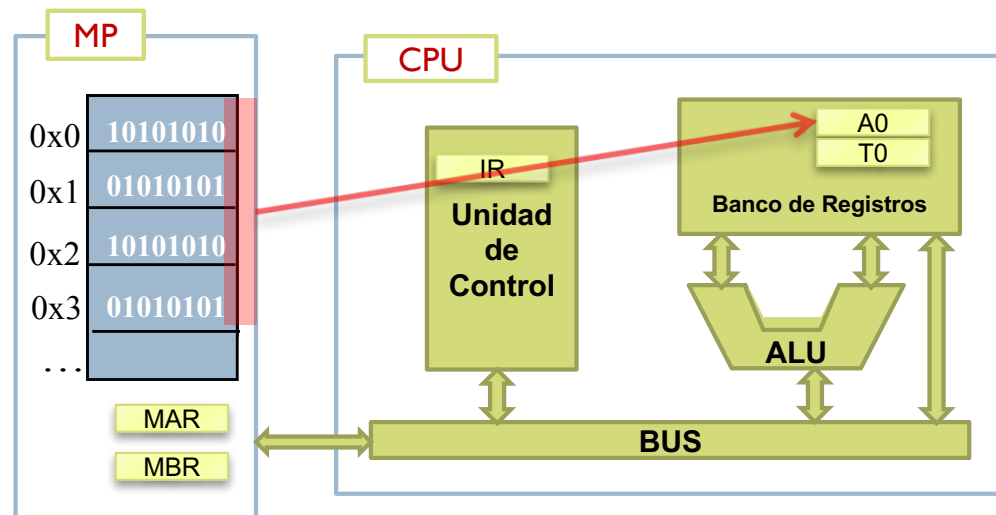
l.s \$f0 dir1

l.d \$f2 dir2

- ▶ Registro a memoria

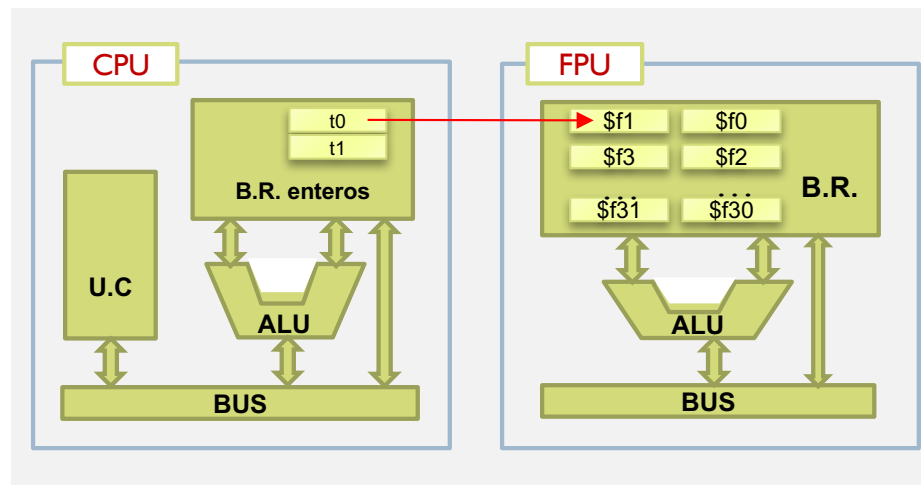
s.s \$f0 dir1

s.d \$f0 dir2



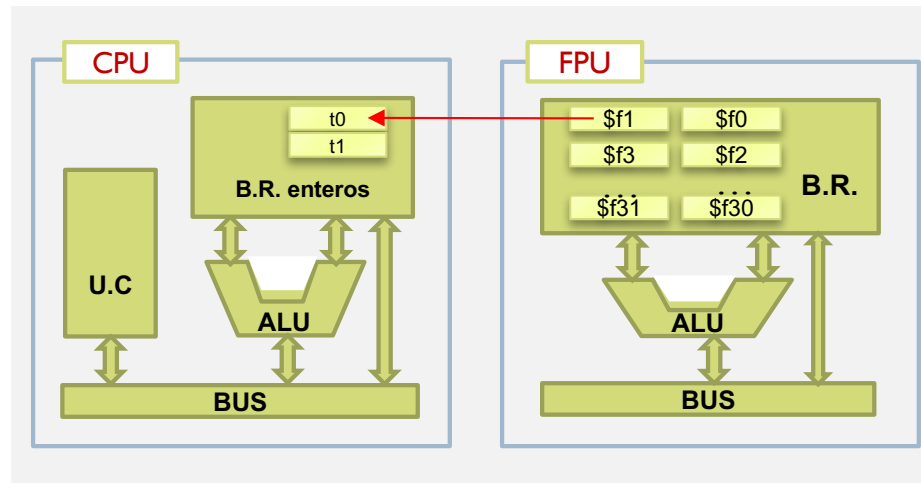
Operaciones con registros (CPU, FPU)

mtc1 \$t0 \$f1



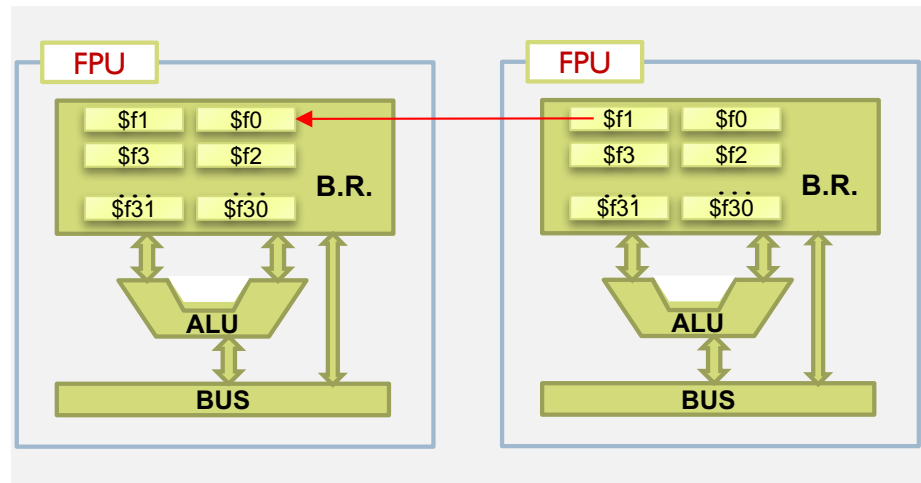
Operaciones con registros (CPU, FPU)

mfc1 \$t0 \$f1



Operaciones con registros (FPU, FPU)

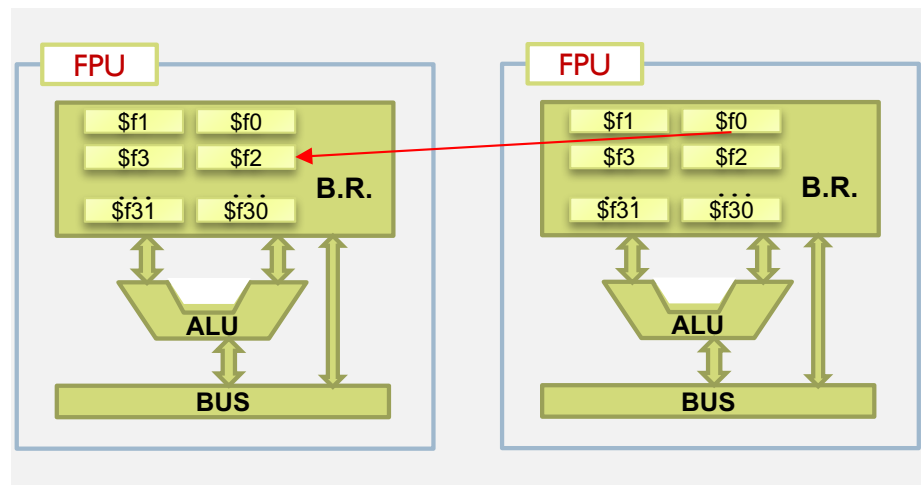
`mov.s $f0 $f1`



`$f0 ← $f1`

Operaciones con registros (FPU, FPU)

`mov.d $f0 $f2`



$(\$f0, \$f1) \leftarrow (\$f2, \$f3)$

Operaciones de conversión

- ▶ `cvt.s.w $f2 $f1`
 - ▶ Convierte un entero (\$f1) a simple precisión (\$f2)
- ▶ `cvt.w.s $f2 $f1`
 - ▶ Convierte de simple precisión (\$f1) a entero (\$f2)
- ▶ `cvt.d.w $f2 $f0`
 - ▶ Convierte un entero (\$f0) a doble precisión (\$f2)
- ▶ `cvt.w.d $f2 $f0`
 - ▶ Convierte de doble precisión (\$f0) a entero (\$f2)
- ▶ `cvt.d.s $f2 $f0`
 - ▶ Convierte de simple precisión (\$f0) a doble (\$f2)
- ▶ `cvt.s.d $f2 $f0`
 - ▶ Convierte de doble precisión (\$f0) a simple (\$f2)

Operaciones de carga

- ▶ `li.s $f4, 8.0`
 - ▶ Carga el valor float 8.0 en el registro \$f4
- ▶ `li.d $f2, 12.4`
 - ▶ Carga el valor double 12.4 en el registro \$f2 , par (\$f2,\$f3)

Ejemplo

```
float PI      = 3,1415;  
int   radio  = 4;  
float longitud;  
  
longitud = PI * radio;
```

```
.text  
    .globl main  
  
main:  
  
    li.s      $f0  3.1415  
    li        $t0  4  
  
    mtc1      $t0 $f1      # 4 en Ca2  
    cvt.s.w   $f2 $f1      # 4 ieee754  
    mul.s     $f3 $f2 $f0
```

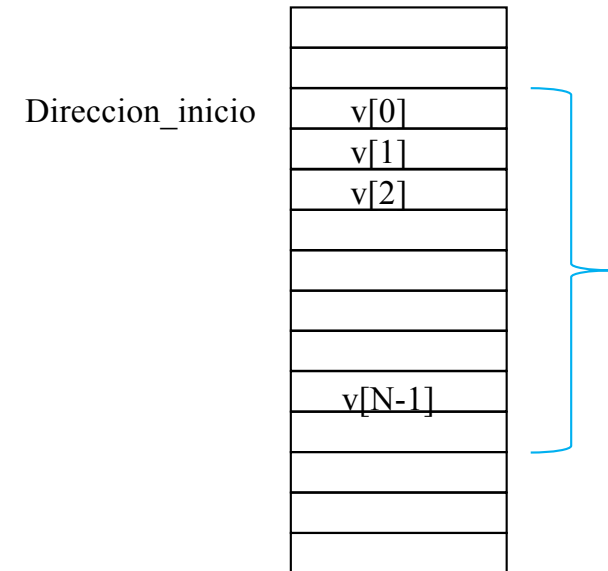
Tipo de datos básicos

vectores

- ▶ Conjunto de elementos ordenados consecutivamente en memoria
- ▶ La dirección del elemento j se obtiene como:

$$\text{Direccion_inicio} + j * p$$

Siendo p el tamaño de cada elemento



Tipo de datos básicos

vectores

```
int vec[5] ;
```

```
...
```

```
main ()
```

```
{
```

```
    vec[4] = 8;
```

```
}
```

```
.data
```

```
    .align    2 #siguiente dato alineado a 4
```

```
vec: .space 20    #5 elem.*4 bytes
```

```
.text
```

```
main:
```

```
    la $t1 vec
```

```
    li $t2 8
```

```
    sw $t2 16($t1)
```

```
    ...
```


Tipo de datos básicos

vectores

```
int vec[5] ;
```

```
...
```

```
main ()
```

```
{
```

```
    vec[4] = 8;
```

```
}
```

```
.data
```

```
.align    2 #siguiente dato alineado a 4  
vec: .space 20    #5 elem.*4 bytes
```

```
.text
```

```
main:
```

```
    li    $t0, 16  
    la    $t1, vec  
    add   $t3, $t1, $t0  
    li    $t2, 8  
    sw    $t2, ($t3)
```

```
...
```

Tipo de datos básicos

vectores

```
int vec[5] ;
```

```
...
```

```
main ()
```

```
{
```

```
    vec[4] = 8;
```

```
}
```

```
.data
```

```
.align    2    #siguiente dato alineado a 4  
vec: .space 20    #5 elem.*4 bytes
```

```
.text
```

```
main:
```

```
    li $t2  8  
    li $t1  16  
    sw $t2  vec($t1)  
    ...
```

Ejercicio

- ▶ Si V es un array de números enteros (int)
 - ▶ V representa la dirección de inicio de vector
- ▶ ¿En qué dirección se encuentra el elemento $V[5]$?
- ▶ ¿Qué instrucción permite cargar en el registro $\$t0$ el valor $v[5]$?

Ejercicio (Solución)

- ▶ Si V es un array de números enteros (int)
 - ▶ V representa la dirección de inicio de vector
- ▶ ¿En qué dirección se encuentra el elemento $V[5]$?
 - ▶ $V + 5*4$
- ▶ ¿Qué instrucción permite cargar en el registro $\$t0$ el valor $v[5]$?
 - ▶ `lw $t1, 20`
 - ▶ `lw $t0, v($t1)`

Ejercicio

- Escriba un programa en ensamblador equivalente a:

```
int vec[100] ;
...

main ()
{
    int i = 0;

    for (i = 0; i < 100; i++)
        vec[i] = 5;

}
```

- Asumiendo que en \$a0 se encuentra almacenada la dirección del vector

Ejercicio

- Escriba un programa en ensamblador equivalente a:

```
int vec[100] ;  
...  
  
main ()  
{  
    int i = 0;  
    suma = 0;  
  
    for (i = 0; i < 100; i++)  
        suma = suma + vec[i];  
  
}
```

- Asumiendo que en \$a0 se encuentra almacenada la dirección del vector y que el resultado ha de almacenarse en \$v0

Tipo de datos básicos

cadenas de caracteres

```
char c1 ;  
char c2='h' ;  
char *ac1 = "hola" ;  
...
```

```
main ()  
{  
    printf("%s",ac1) ;  
    ...  
}
```

.data

```
c1:  .space 1      # 1 byte  
c2:  .byte 'h'  
ac1: .asciiz "hola"  
...
```

.text

```
main:  
    li $v0 4  
    la $a0 ac1  
    syscall  
    ...
```

Representación de cadenas de caracteres

```
// tira de caracteres (strings)
char c1[10] ;
char ac1[] = "hola" ;
```

.data

```
# strings
c1:   .space 10      # 10 byte
ac1:  .asciiz "hola" # 5 bytes (!)
ac2:  .ascii  "hola" # 4 bytes
```

ac1:	...		ac2:	...	
	'h'	0x0108		'h'	0x0108
	'o'	0x0109		'o'	0x0109
	'l'	0x010a		'l'	0x010a
	'a'	0x010b		'a'	0x010b
	0	0x010c		...	0x010c
	...	0x010d		...	0x010d

Ejercicio

```
// variables globales
```

```
char v1;
```

```
int v2 ;
```

```
float v3 = 3.14 ;
```

```
char v4[10] ;
```

```
char v5 = "ec" ;
```

```
int v6[] = { 20, 22 } ;
```

Ejercicio (solución)

```
// variables globales
```

```
char v1;
```

```
int v2 ;
```

```
float v3 = 3.14 ;
```

```
char v4 = "ec" ;
```

```
int v5[] = { 20, 22 } ;
```

```
.data
```

```
v1: .space 1
```

```
.align 2
```

```
v2: .space 4
```

```
v3: .float 3.14
```

```
v4: .asciiz "ec"
```

```
.align 2
```

```
v5: .word 20, 22
```

Ejercicio (solución)

v1:	0	0x0100
	?	0x0101
	?	0x0102
	?	0x0103
	?	
	?	
	?	
	?	
	?	
	?	
	?	
	?	
	?	
	?	
	?	
	?	
	...	

.data

v1: .space 1

.align 2

v2: .space 4

v3: .float 3.14

v4: .ascii "ec"

.align 2

v5: .word 20, 22

Ejercicio (solución)

v1:	0	0x0100
	?	0x0101
	?	0x0102
	?	0x0103
v2:	0	0x0104
	0	0x0105
	0	0x0106
	0	0x0107
v3:	(3.14)	0x0108
	(3.14)	0x0109
	(3.14)	0x010A
	(3.14)	0x010B
v4:	'e'	0x010C
	'c'	0x010D
	0	0x010E
		0x010F
v5:	(20)	0x0110
	(22)	0x0111
	...	0x0112

```
.data

v1: .space 1
.align 2
v2: .space 4
v3: .float 3.14

v4: .ascii "ec"

.align 2
v5: .word 20, 22
```

Tipo de datos básicos

Longitud de una cadena de caracteres

```
char c1 ;
char c2='h' ;
char *ac1 = "hola" ;
char *c;
...

main ()
{
    c = ac1; int l = 0;
    while (c[l] != NULL) {
        l++;
    }
    printf("%d", l);
    ...
}
```

Tipo de datos básicos

Longitud de una cadena de caracteres

```
char c1 ;
char c2='h' ;
char *ac1 = "hola" ;
char *c;
...
```

```
main ()
{
    c = ac1; int l = 0;
    while (c[l] != NULL) {
        l++;
    }
    printf("%d", l);
    ...
}
```

```
.data
c1:  .space 1      # 1 byte
c2:  .byte 'h'
ac1: .asciiz "hola"
.align 2
c:   .space 4      #puntero => dirección
...

.text
        .globl main
main:   la $t0, ac1
        li $a0, 0
        lbu $t1, ($t0)
buc:    beqz $t1, fin
        addi $t0, $t0, 1
        addi $a0, $a0, 1
        lbu $t1, ($t0)
        b buc

fin:    li $v0 1
        syscall
```

Vectores y cadenas

► En general:

► `lw $t0, 4($s3) # $t0 ← M[$s3+4]`

► `sw $t0, 4($s3) # M[$s3+4] ← $t0`

Ejercicio

- ▶ Escriba un programa que:
 - ▶ Indique el número de veces que aparece un carácter en una cadena de caracteres
 - ▶ La dirección de la cadena se encuentra en \$a0
 - ▶ El carácter a buscar se encuentra en \$a1
 - ▶ El resultado se dejará en \$v0

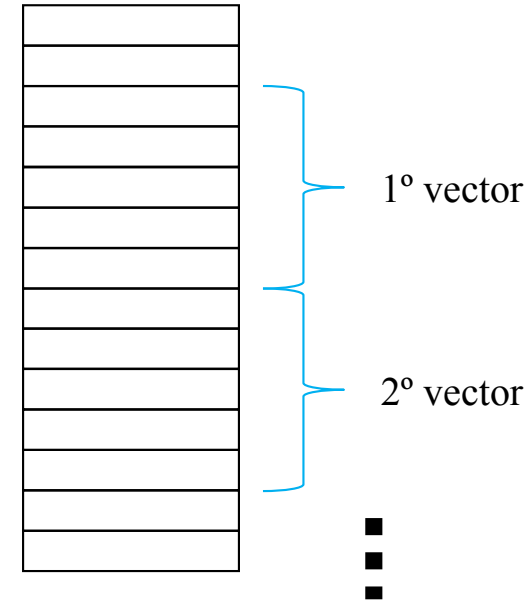
Tipos de datos básicos

matrices

- ▶ Una matriz $m \times n$ se compone de m vectores de longitud n
- ▶ Normalmente se almacenan en memoria por filas
- ▶ El elemento a_{ij} se encuentra en la dirección:

$$\text{direccion_inicio} + (i \cdot n + j) \times p$$

siendo p el tamaño de cada elemento



Tipo de datos básicos

matrices

```
int vec[5] ;  
int mat[2][3] = {{11,12,13},  
                 {21,22,23}};  
...
```

```
main ()
```

```
{
```

```
    m[0][1] = m[0][0] +  
             m[1][0] ;
```

```
    ...
```

```
}
```

```
.data
```

```
    .align 2    #siguiente dato alineado a 4
```

```
vec: .space 20    #5 elem.*4 bytes
```

```
mat: .word 11, 12, 13
```

```
      .word 21, 22, 23
```

```
...
```

```
.text
```

```
main:    lw $t1 mat+0
```

```
         lw $t2 mat+12
```

```
         add $t3 $t1 $t2
```

```
         sw  $t3 mat+4
```

```
...
```

Ejemplo (enteros)

```
int vec[5] ;  
int mat[2][3] = {{11,12,13},  
                 {21,22,23}};  
...
```

```
main ()  
{  
    mat[1][2] = mat[1][1] +  
        mat[2][1];  
    ...  
}
```

```
.data
```

```
align 2
```

```
vec: .space 20    #5 elem.*4 bytes
```

```
mat: .word 11, 12, 13
```

```
      .word 21, 22, 23
```

```
...
```

```
.text
```

```
.globl main
```

```
main:  lw  $t1 mat+0
```

```
      lw  $t2 mat+12
```

```
      add $t3 $t1 $t2
```

```
      sw  $t3 mat+4
```

```
...
```

Punteros en C

```
int a;  
int *b;
```

```
main ()  
{  
    b = &a;  
    *b = 2;  
    ...  
}
```

```
.data
```

```
align 2
```

```
a: .space 4    # int
```

```
b: .space 4    # dirección
```

```
.text
```

```
.globl main
```

```
main:  la $t0,  a
```

```
       sw $t0,  b
```

```
       li $t0,  2
```

```
       lw $t1,  b
```

```
       sw $t0,  ($t1)
```

```
       ... . .
```

Otros tipos de datos (estructuras de C)

```
struct Punto {  
    int x;  
    int y;  
};
```

```
struct Punto p;
```

```
main ()
```

```
{
```

```
    p.x = 80;
```

```
    p.y = 80;
```

```
}
```

```
.data
```

```
align 2
```

```
p:
```

```
p.x: .space 4
```

```
p.y: .space 4
```

```
...
```

```
.text
```

```
main:
```

```
    li $t0, 80
```

```
    sw $t0, p.x
```

```
    li $t1, 70
```

```
    sw $t1, p.y
```

Otros tipos de datos (estructuras de C)

```
struct Punto {  
    int x;  
    int y;  
};
```

```
struct Punto p;  
struct Punto q;
```

```
main ()  
{  
    p.x = 80;  
    p.y = 80;  
    q = p;  
}
```

```
.data
```

```
align 2
```

```
p:
```

```
p.x: .space 4
```

```
p.y: .space 4
```

```
q:
```

```
q.x: .space 4
```

```
q.y: .space 4
```

```
...
```

Otros tipos de datos (estructuras de C)

```
struct Punto {  
    int x;  
    int y;  
};
```

```
struct Punto p;  
struct Punto q;
```

```
main ()  
{  
    p.x = 80;  
    p.y = 80;  
    q = p;  
}
```

.text

```
main:  li $t0, 80  
       sw $t0, p.x  
       li $t1, 70  
       sw $t1, p.y  
  
       lw $t0, p.x  
       sw $t0, q.x  
       lw $t0, p.y  
       sw $t0, q.y
```

Otros tipos de datos (estructuras de C)

```
struct Punto {  
    int x;  
    char y[21];  
    int z;  
};  
  
struct Punto p;
```

```
main ()  
{  
    p.y[2] = 'b';  
}
```

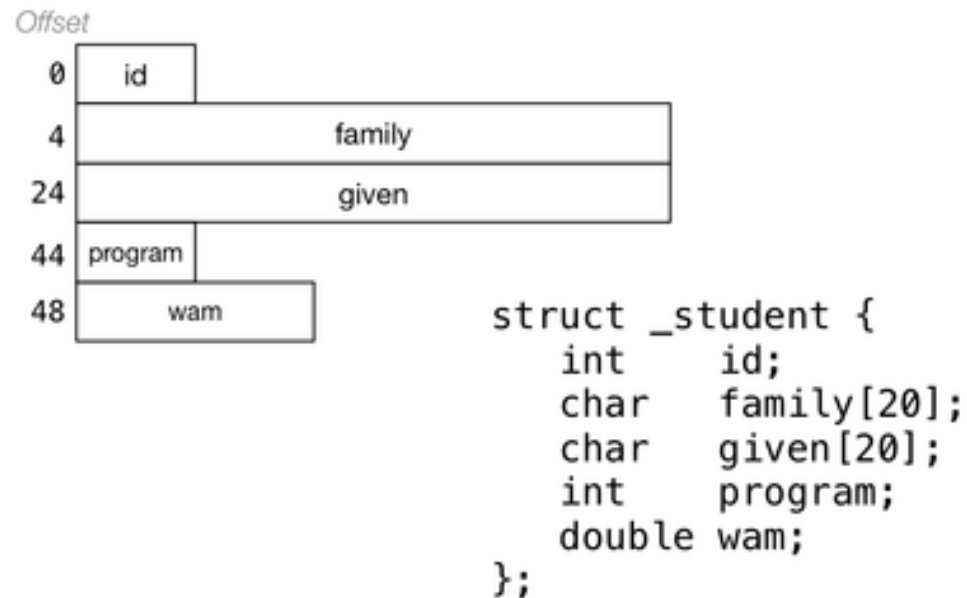
.data

```
align 2  
p:  
p.x: .space 4  
p.y: .space 21  
align 2  
p.z: .space 4
```

.text

```
main:  
    li $t0, 'b'  
    la $t1, p.y  
    sw $t1, 2($t1)
```


Otros tipos de datos (estructuras de C)



Consejos

- ▶ No programar directamente en ensamblador
 - ▶ Mejor **primero hacer diseño** en DFD, Java/C/Pascal...
 - ▶ Ir traduciendo poco a poco el diseño a ensamblador
- ▶ **Comentar** suficientemente el código y datos
 - ▶ Por línea o por grupo de líneas comentar qué parte del diseño implementa.
- ▶ **Probar** con suficientes casos de prueba
 - ▶ Probar que el programa final funciona adecuadamente a las especificaciones dadas

Ejercicio

- ▶ Escriba un programa que:
 - ▶ Cargue el valor -3.141516 en el registro \$f0
 - ▶ Permita obtener el valor del exponente y de la mantisa almacenada en el registro \$f0 (en formato IEEE 754)
 - ▶ Imprima el signo
 - ▶ Imprima el exponente
 - ▶ Imprima la mantisa

Ejercicio (Solución)

```
.data

    saltolinea: .asciiz  "\n"

.text
.globl main
main:

    li.s  $f0, -3.141516

    #se imprime
    mov.s $f12, $f0
    li $v0, 2
    syscall

    la $a0, saltolinea
    li $v0, 4
    syscall

    # se copia al procesador
    mfc1  $t0, $f12
```

```
li  $s0, 0x80000000    #signo
and $a0, $t0, $s0
srl $a0, $a0, 31
li  $v0, 1
syscall

la  $a0, saltolinea
li  $v0, 4
syscall

li  $s0, 0x7F800000    #exponente
and $a0, $t0, $s0
srl $a0, $a0, 23
li  $v0, 1
syscall

la  $a0, saltolinea
li  $v0, 4
syscall

li  $s0, 0x007FFFFF    #mantisa
and $a0, $t0, $s0
li  $v0, 1
syscall

jr  $ra
```