

Grupo ARCOS

uc3m | Universidad **Carlos III** de Madrid

Tema 3 (II)

Fundamentos de la programación en ensamblador

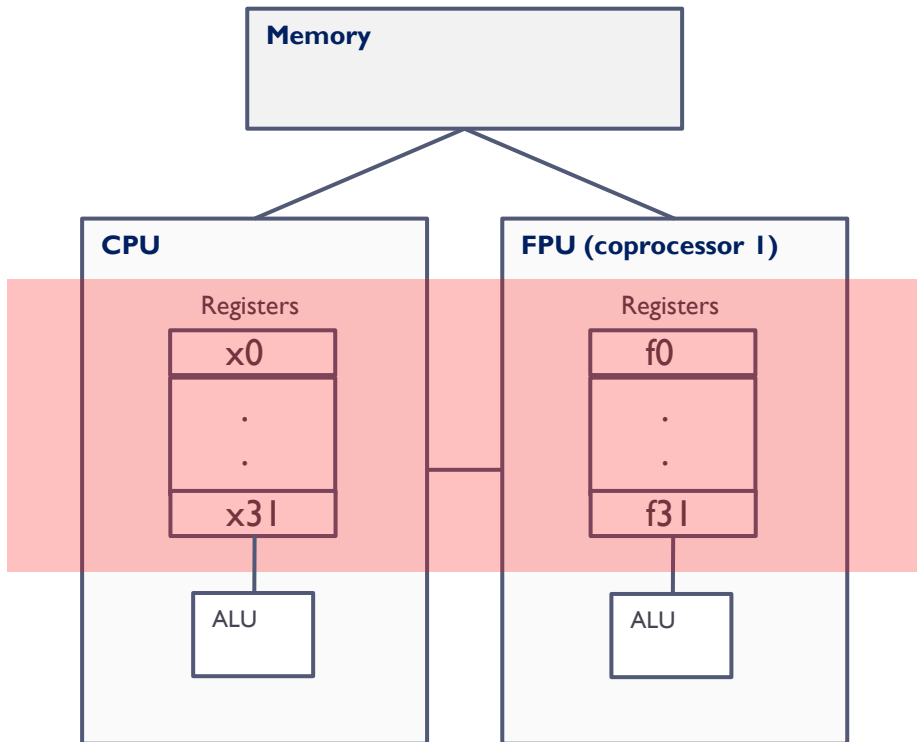
Estructura de Computadores
Grado en Ingeniería Informática



Contenidos

- ▶ Fundamentos básicos de la programación en ensamblador
- ▶ Ensamblador del RISC-V 32, modelo de memoria y representación de datos
 - ▶ RISC-V: registros y memoria
 - ▶ Directivas de ensamblador
 - ▶ Servicios del sistema
 - ▶ Instrucciones para el acceso a memoria
- ▶ Formato de las instrucciones y modos de direccionamiento
- ▶ Llamadas a procedimientos y uso de la pila

RISC-V 32



- ▶ **Registros**
 - ▶ Dentro del procesador
 - ▶ Instrucciones trabajan con valores en registros
 - ▶ 32 registros de 32 bits
- ▶ **Memoria**
 - ▶ Fuera del procesador
 - ▶ Intercambio de datos a/desde registros
 - ▶ Más capacidad pero más tiempo de acceso que los registros
 - ▶ Direcciones de 32 bits
 - ▶ 4 GiB direccionable

Banco de registros del RISC-V 32

| | | | | |
|----|--|------------------------------------|--|----|
| 0 | | ■ Hay 32 registros | | 16 |
| 1 | | □ 4 bytes de tamaño (una palabra) | | 17 |
| 2 | | □ Se nombran con un x al principio | | 18 |
| 3 | | | | 19 |
| 4 | | ■ Convenio de uso | | 20 |
| 5 | | □ Reservados | | 21 |
| 6 | | □ Argumentos | | 22 |
| 7 | | □ Resultados | | 23 |
| 8 | | □ Temporales | | 24 |
| 9 | | □ Punteros | | 25 |
| 10 | | | | 26 |
| 11 | | | | 27 |
| 12 | | | | 28 |
| 13 | | | | 29 |
| 14 | | | | 30 |
| 15 | | | | 31 |

Banco de registros del RISC-V 32

| | | | | |
|----|--|------|--|----|
| 0 | | zero | | 16 |
| 1 | | | | 17 |
| 2 | | | | 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 |

Valor cableado a cero
No puede modificarse

Banco de registros del RISC-V 32

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

Valores temporales

Banco de registros del RISC-V 32

| | | | | |
|----|--|---------|-----|----|
| 0 | | zero | | 16 |
| 1 | | | | 17 |
| 2 | | | | 18 |
| 3 | | | | 19 |
| 4 | | | | 20 |
| 5 | | t0 | s2 | 21 |
| 6 | | t1 | s3 | 22 |
| 7 | | t2 | s4 | 23 |
| 8 | | s0 / fp | s5 | 24 |
| 9 | | | s1 | 25 |
| 10 | | | s6 | 26 |
| 11 | | | s7 | 27 |
| 12 | | | s8 | 28 |
| 13 | | | s9 | 29 |
| 14 | | | s10 | 30 |
| 15 | | | s11 | 31 |
| | | | t3 | |
| | | | t4 | |
| | | | t5 | |
| | | | t6 | |

Valores guardados

Banco de registros del RISC-V 32

| | | | | | |
|----|--|---------|-----|--|----|
| 0 | | zero | | | |
| 1 | | ra | a6 | | 16 |
| 2 | | | a7 | | 17 |
| 3 | | | s2 | | 18 |
| 4 | | | s3 | | 19 |
| 5 | | t0 | s4 | | 20 |
| 6 | | t1 | s5 | | 21 |
| 7 | | t2 | s6 | | 22 |
| 8 | | s0 / fp | s7 | | 23 |
| 9 | | s1 | s8 | | 24 |
| 10 | | a0 | s9 | | 25 |
| 11 | | a1 | s10 | | 26 |
| 12 | | a2 | s11 | | 27 |
| 13 | | a3 | t3 | | 28 |
| 14 | | a4 | t4 | | 29 |
| 15 | | a5 | t5 | | 30 |
| | | | t6 | | 31 |

Paso de parámetros y
Gestión de subrutinas

Banco de registros del RISC-V 32

| | | | | | |
|----|--|---------|-----|--|----|
| 0 | | zero | a6 | | 16 |
| 1 | | ra | a7 | | 17 |
| 2 | | sp | s2 | | 18 |
| 3 | | gp | s3 | | 19 |
| 4 | | tp | s4 | | 20 |
| 5 | | t0 | s5 | | 21 |
| 6 | | t1 | s6 | | 22 |
| 7 | | t2 | s7 | | 23 |
| 8 | | s0 / fp | s8 | | 24 |
| 9 | | s1 | s9 | | 25 |
| 10 | | a0 | s10 | | 26 |
| 11 | | a1 | s11 | | 27 |
| 12 | | a2 | t3 | | 28 |
| 13 | | a3 | t4 | | 29 |
| 14 | | a4 | t5 | | 30 |
| 15 | | a5 | t6 | | 31 |

Puntos de lectura/escritura (punteros)

Banco de registros (enteros)

resumen

| Nombre registro | Número | Uso |
|-----------------|-----------|-----------------------------------------------------------------|
| zero | x0 | Constante 0 |
| ra | x1 | Dirección de retorno (rutinas) |
| sp | x2 | Puntero a pila |
| gp | x3 | Puntero al área global |
| tp | x4 | Puntero al hilo |
| t0...t2 | x5-x7 | Temporal (<u>NO</u> se conserva entre llamadas) |
| s0/fp | x8 | Temporal (se conserva entre llamadas) / Puntero a marco de pila |
| s1 | x9 | Temporal (se conserva entre llamadas) |
| a0...a1 | x10...11 | Argumento de entrada para rutinas/valores de retorno |
| a2...a7 | 12...x17 | Argumento de entrada para rutinas |
| s2... s11 | x18...x27 | Temporal (se conserva entre llamadas) |
| t3...t6 | x28...x31 | Temporal (<u>NO</u> se conserva entre llamadas) |

- ▶ Hay 32 registros
 - ▶ 4 bytes de tamaño (una palabra)
 - ▶ Doble nombrado: lógico y numérico (con **x** al principio)
- ▶ Convenio de uso
 - ▶ Reservados
 - ▶ Argumentos
 - ▶ Resultados
 - ▶ Temporales
 - ▶ Punteros

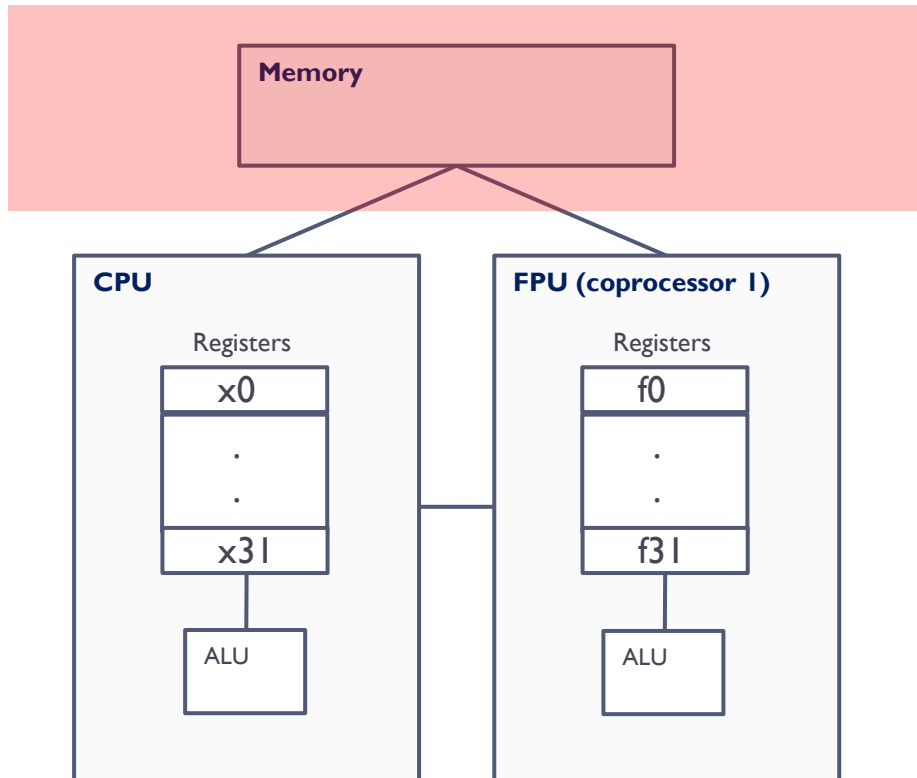
Banco de registros (coma flotante)

resumen

| Nombre registro | Número registro | Uso |
|-----------------|-----------------|------------------------------------|
| ft0-ft7 | f0 ... f7 | Temporales (como los t...) |
| fs0-fs1 | f8 ... f9 | Se guardan (como los s...) |
| fa0-fa1 | f10 ... f11 | Argumentos/retorno (como los a...) |
| fa2-fa7 | f12 ... f17 | Argumentos (como los a...) |
| fs2-fs11 | f18 ... f27 | Se guardan (como los s...) |
| ft8-ft11 | f28 ... f31 | Temporales (como los t...) |

- ▶ Hay 32 registros
- ▶ En R32F (simple precisión) los registros son de 32 bits (4 bytes)
- ▶ En R32D (doble precisión) los registros son de 64 bits (8 bytes) y pueden almacenar:
 - ▶ Valores de simple precisión en los 32 bits inferiores del registro
 - ▶ Valores de doble precisión en todos los 64 bits del registro

RISC-V 32



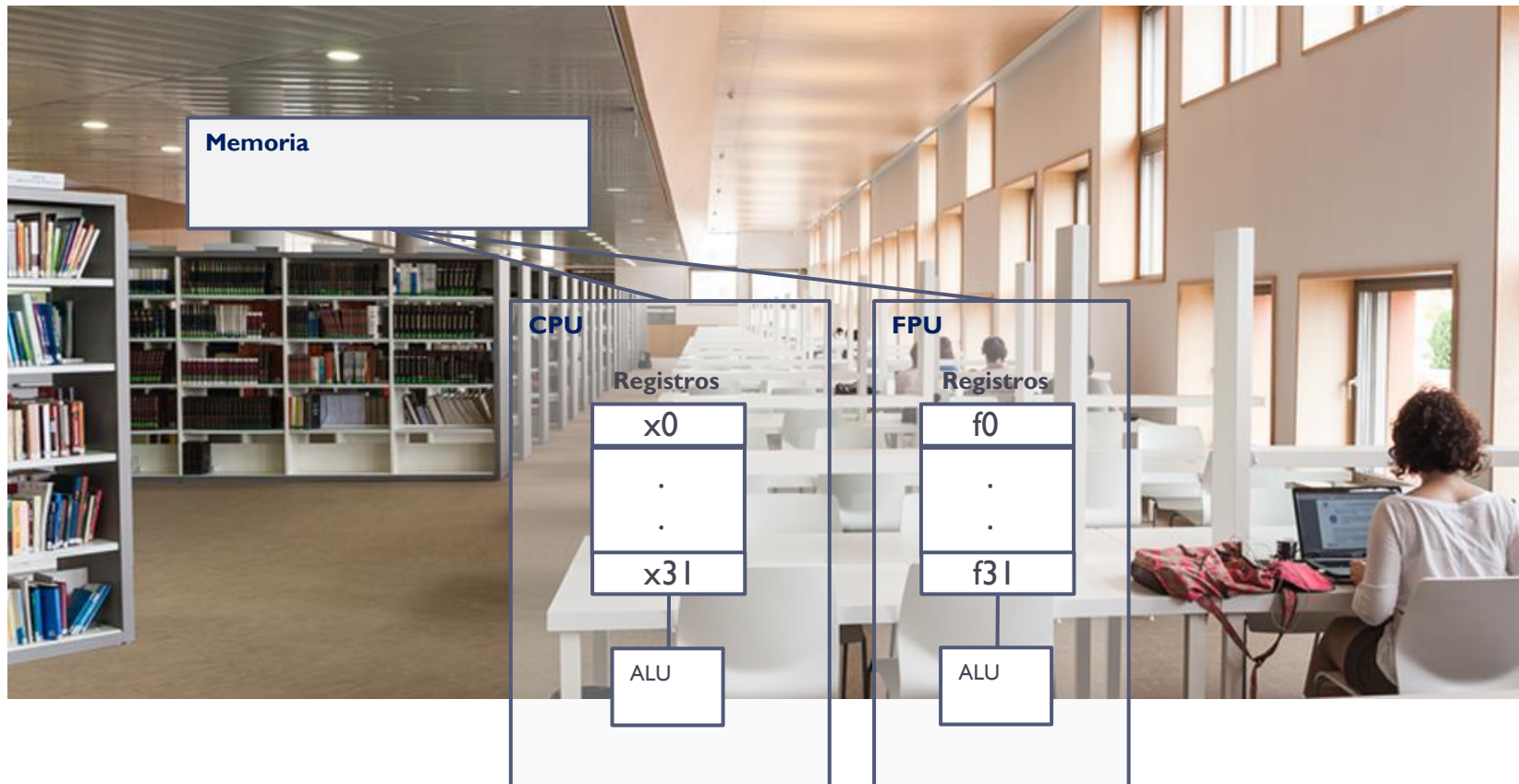
▶ Registros

- ▶ Dentro del procesador
- ▶ Instrucciones trabajan con valores en registros
- ▶ 32 registros de 32 bits

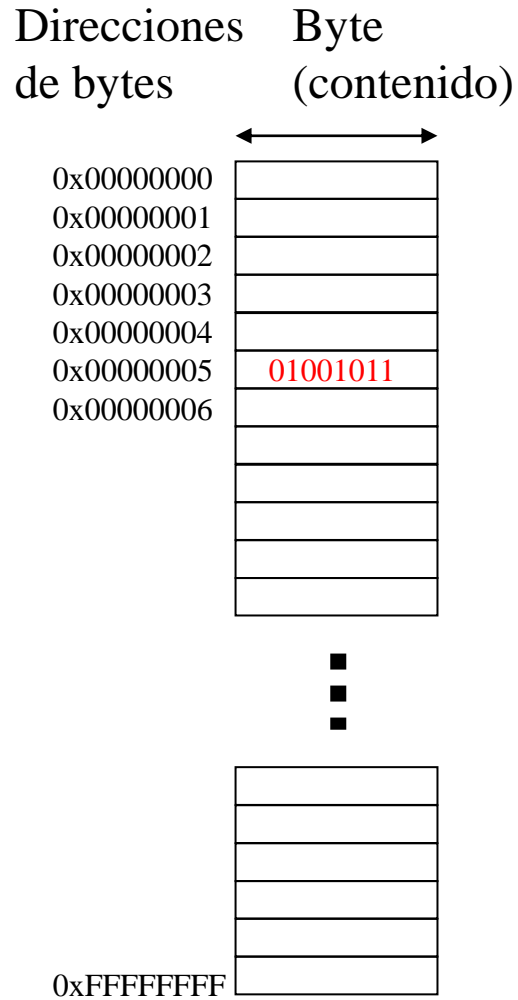
▶ Memoria

- ▶ Fuera del procesador
- ▶ Intercambio de datos a/desde registros
- ▶ Más capacidad pero más tiempo de acceso que los registros
- ▶ Direcciones de 32 bits
 - ▶ 4 GiB direccionable

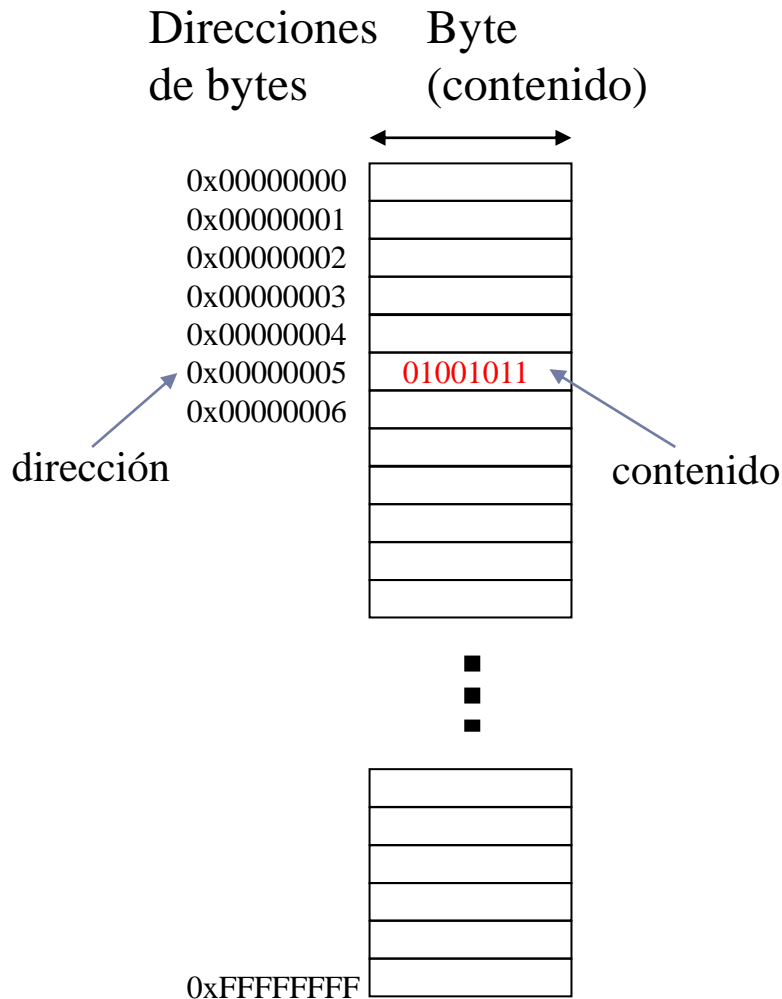
RISC-V 32 (memoria y registros)



Modelo de memoria de RISC-V 32



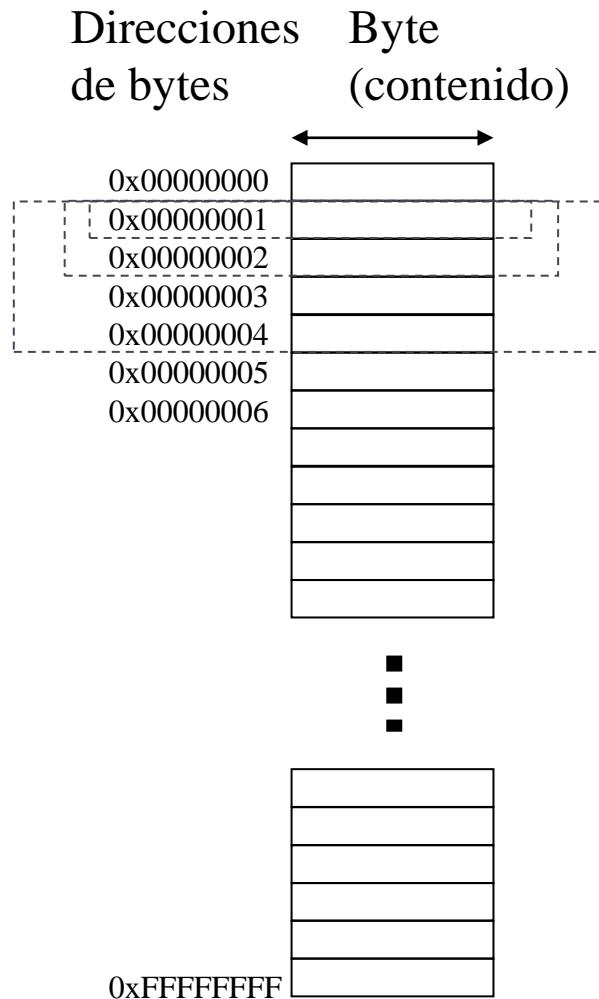
Modelo de memoria de RISC-V 32



La memoria se direcciona por bytes:

- Direcciones de 32 bits
- Contenido de cada dirección: un byte
- Espacio direccionable: 2^{32} bytes = 4GiB

Modelo de memoria de RISC-V 32



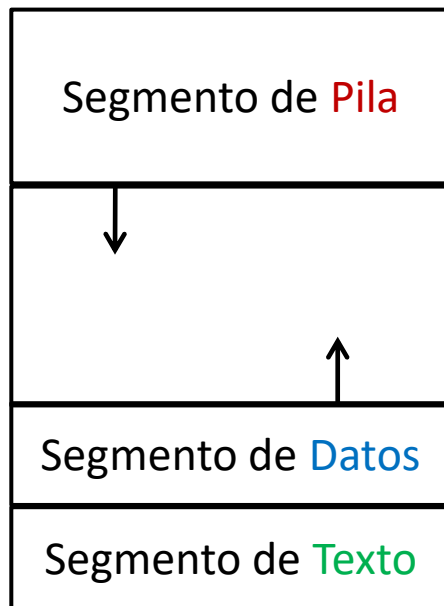
La memoria se direcciona por bytes:

- Direcciones de 32 bits
- Contenido de cada dirección: un byte
- Espacio direccionable: 2^{32} bytes = 4GiB

El acceso puede ser a:

- Bytes individuales
- Palabras (4 bytes consecutivos)
- Medias palabras (2 bytes)

Mapa de memoria de un proceso



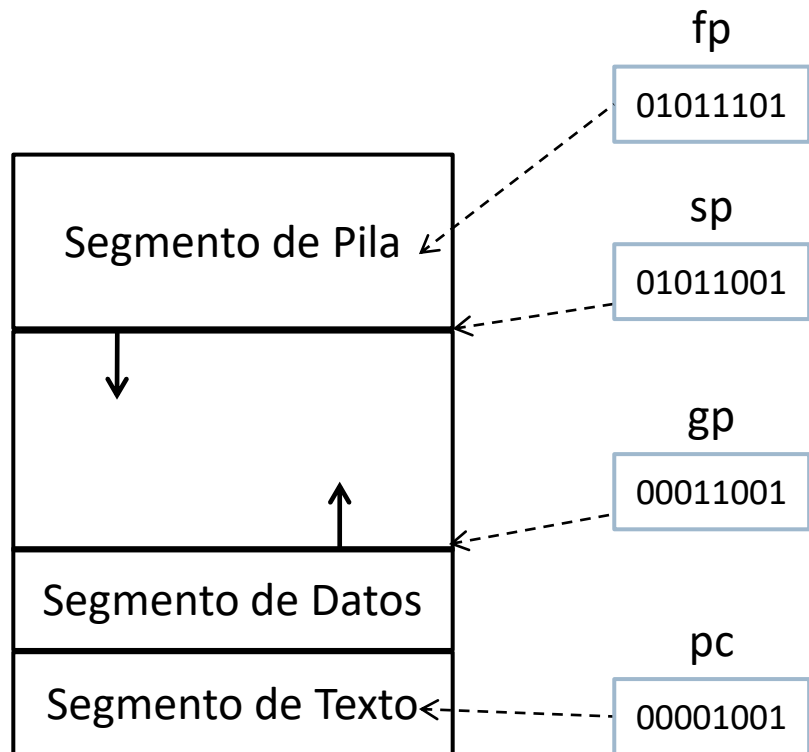
- ▶ Los procesos dividen el espacio de memoria en segmentos lógicos para organizar el contenido:

```
// variables globales
int a;

main ()
{
    // variables locales
    int b;

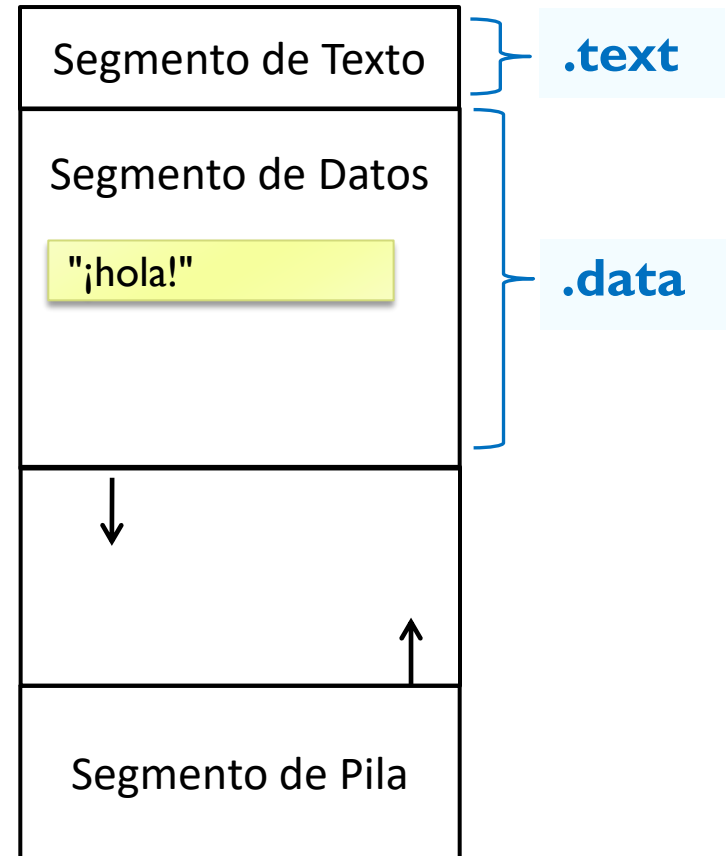
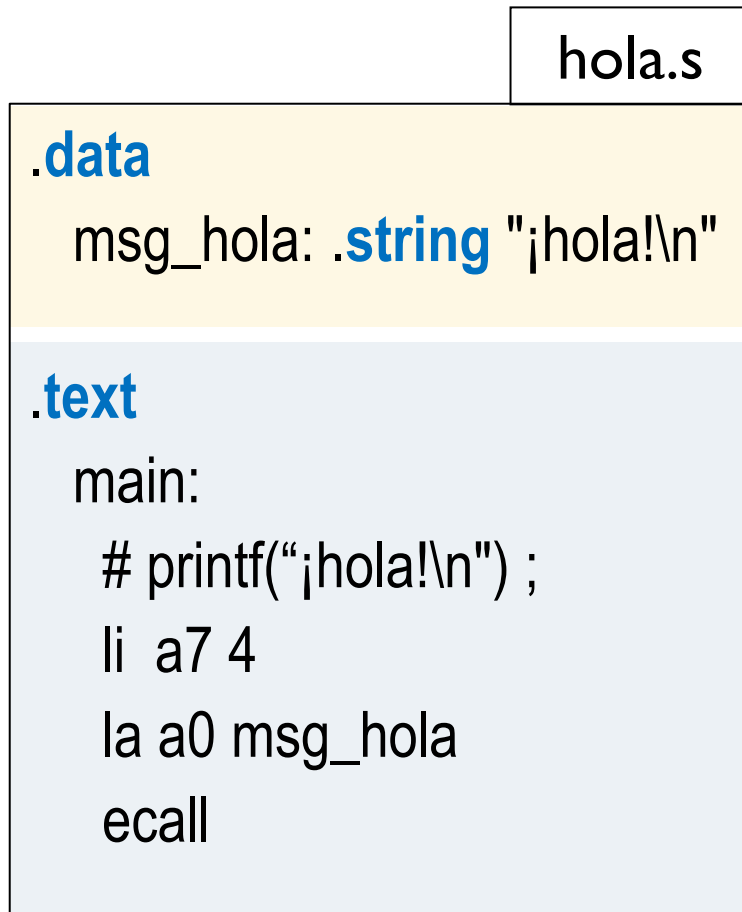
    // código
    return a + b;
}
```

Mapa de memoria de un proceso

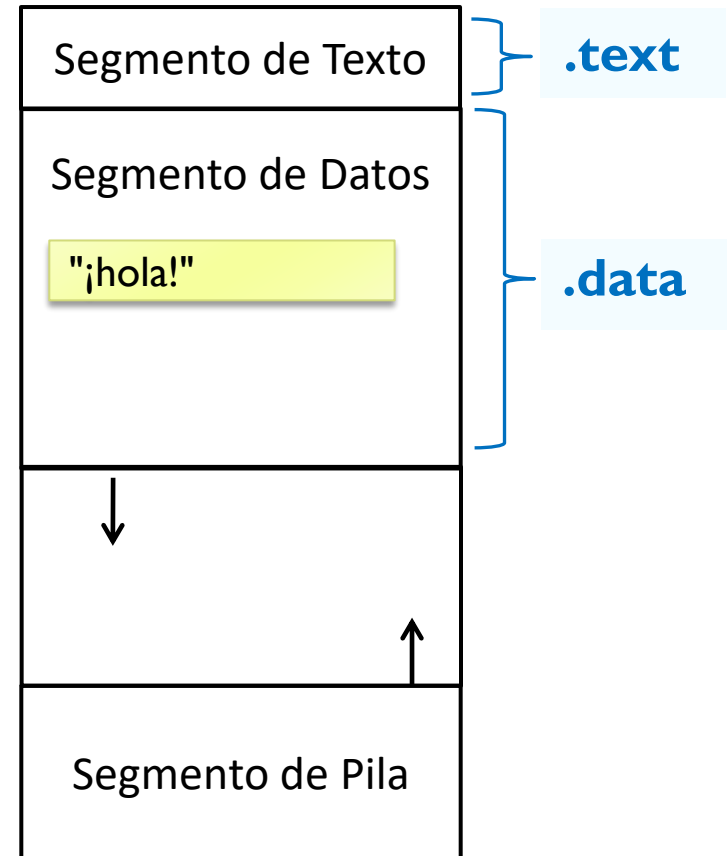
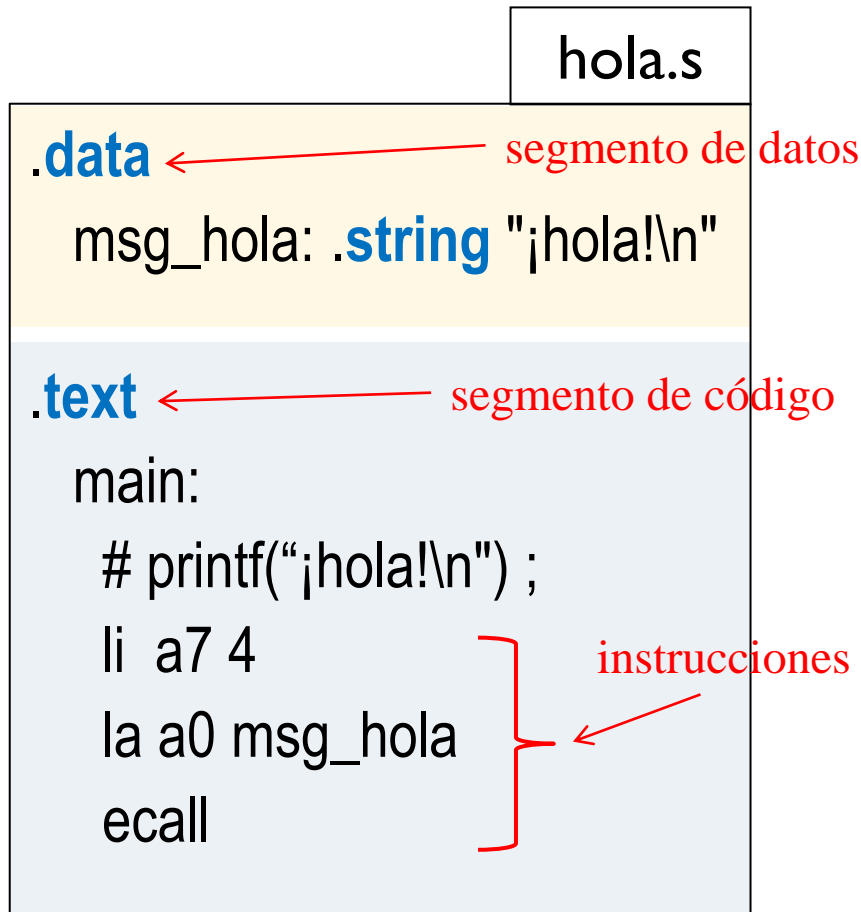


- ▶ Los procesos dividen el espacio de memoria en segmentos lógicos para organizar el contenido:
 - ▶ Segmento de pila
 - ▶ Variables locales
 - ▶ Contexto de funciones
 - ▶ Segmento de datos
 - ▶ Datos estáticos
 - ▶ Segmento de código (texto)
 - ▶ Código

Ejemplo: Hola mundo...



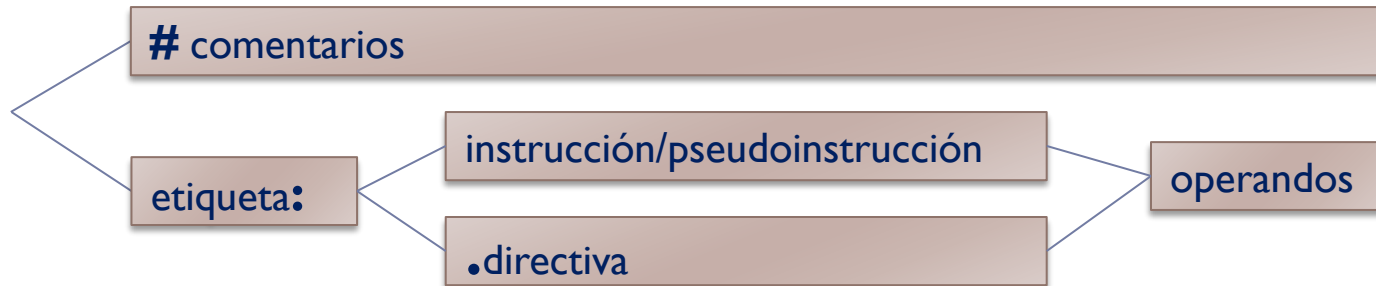
Ejemplo: Hola mundo...



Contenidos

- ▶ Fundamentos básicos de la programación en ensamblador
- ▶ Ensamblador del RISC-V 32, modelo de memoria y representación de datos
 - ▶ RISC-V: registros y memoria
 - ▶ Directivas de ensamblador
 - ▶ Servicios del sistema
 - ▶ Instrucciones para el acceso a memoria
- ▶ Formato de las instrucciones y modos de direccionamiento
- ▶ Llamadas a procedimientos y uso de la pila

Ejemplo: Hola mundo...



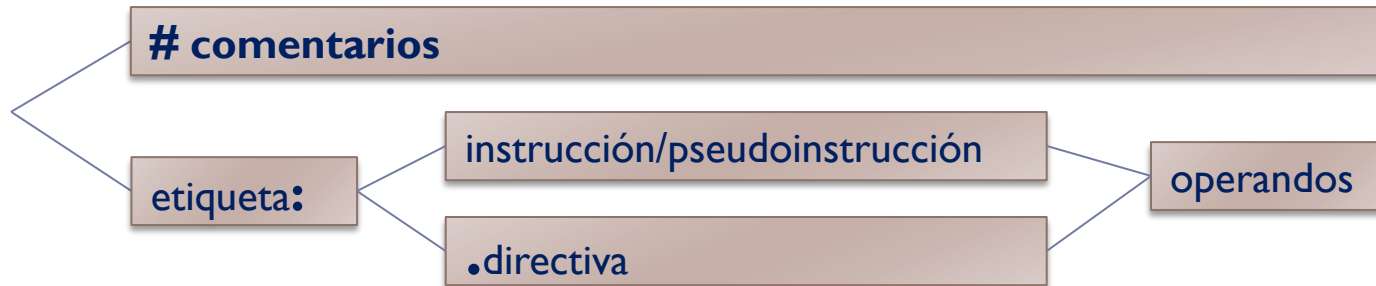
hola.s

```
.data
    msg_hola: .string "Hola mundo\n"

.text
main:
    # printf("Hola mundo\n") ;
    li a7 4
    la a0 msg_hola
    ecall
```

Ensamblador:

Comentarios con



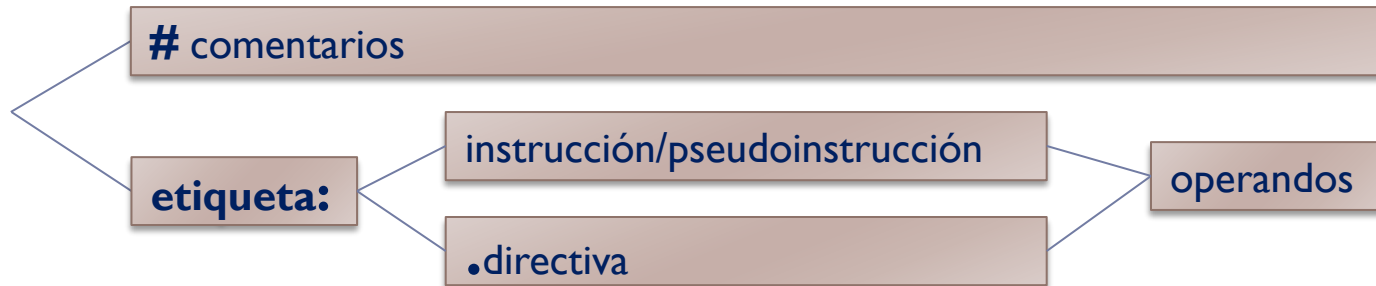
hola.s

```
.data
    msg_hola: .string "Hola mundo\n"

.text
main:
    # printf("Hola mundo\n") ;
    li a7 4
    la a0 msg_hola
    ecall
```

Ensamblador:

etiquetas terminan en :



hola.s

```
.data
msg_hola: .string "Hola mundo\n"
```

```
.text
```

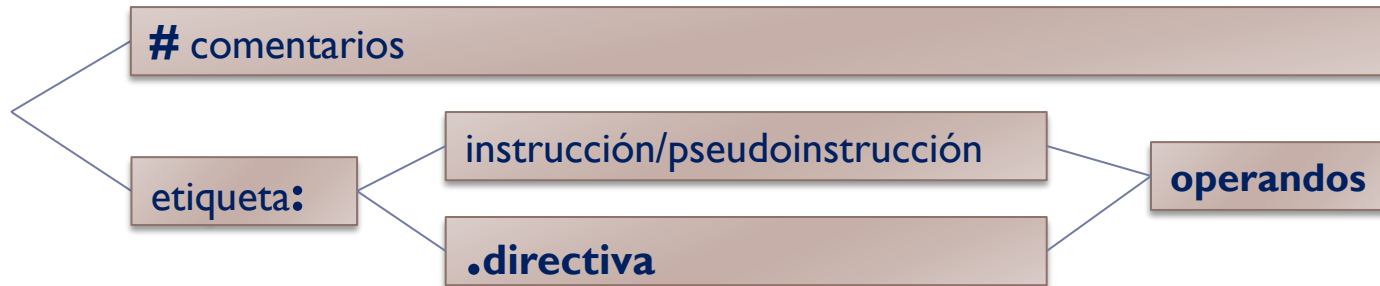
```
main:
    # printf("Hola mundo\n") ;
    li a7 4
    la a0 msg_hola
    ecall
```

msg_hola: representa la dirección de memoria
donde comienza a guardarse la cadena

main: representa la dirección de memoria
donde se guarda la primera instrucción de main

Ensamblador:

Directivas comienzan por .



hola.s

```
.data
    msg_hola: .string "Hola mundo\n"

.text
    main:
        # printf("Hola mundo\n") ;
        li a7 4
        la a0 msg_hola
        ecall
```

Programa en ensamblador: directivas de ensamblador

| Directivas | Uso |
|--------------------------------------------|------------------------------------------------------------|
| .data | Siguientes elementos van al segmento de dato |
| .text | Siguientes elementos van al segmento de código |
| .string <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 |
| .zero 10 | Reserva espacio para 10 bytes a cero en el segmento actual |
| .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: .string "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
```

Representación de tipos básicos (1/3)

```
// booleanos
bool_t b1 ;
bool_t b2 = false ;

// caracteres
char c1 ;
char c2 = 'x' ;

// enteros
int  res1 ;
int  op1 = -10 ;

// coma flotante
float  f0 ;
float  f1 = 1.2 ;
double d2 = 3.0e10 ;
```

```
.data

# boolean
b1:    .zero 1
b2:    .byte 0          # 1 byte

# character
c1:    .byte
c2:    .byte 'x'        # 1 bytes

# integers
res1:  .zero 4
op1:   .word -10        # 4 bytes

# floating point
f0:    .float
f1:    .float  1.2      # 4 bytes
d2:    .double 3.0e10   # 8 bytes
```

Representación de tipos básicos (2/3)

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

.data

```
# strings
c1:  .zero    10      # 10 byte
ac1: .string  "hola"  # 5 bytes (!)
ac2: .byte    'h', 'o', 'l', 'a'
```

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

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

Representación de tipos básicos (3/3)

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

.data

```
# vectores
vec:  .zero 20      # 5 elem.*4 bytes
mat:  .word 11, 12, 13
      .word 21, 22, 23
```

mat:

| | | |
|-----|-----|--|
| | ... | |
| 11 | | |
| 12 | | |
| 13 | | |
| 21 | | |
| 22 | | |
| 23 | | |
| ... | | |

0x0108

0x010c

0x0110

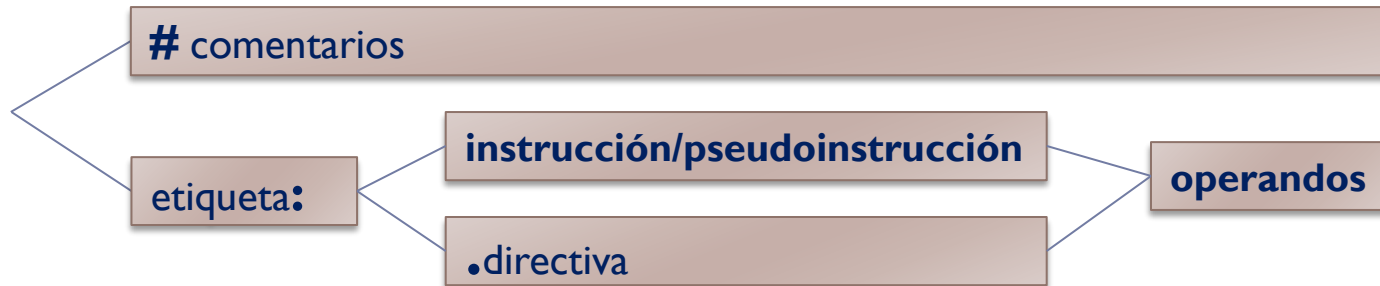
0x0114

0x0118

0x011c

Ensamblador:

Instrucciones vs pseudoinstrucciones



hola.s

```
.data
    msg_hola: .string "Hola mundo\n"

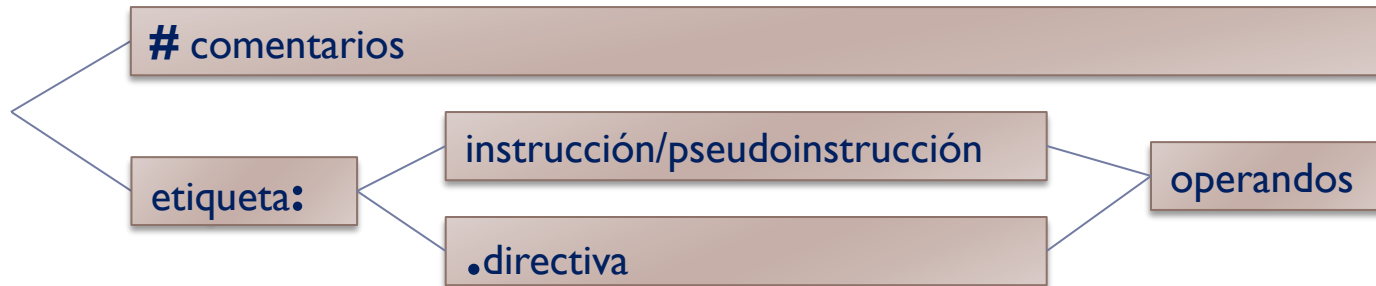
.text
main:
    # printf("Hola mundo\n") ;
    li a7 4
    la a0 msg_hola
    ecall
```

Instrucciones y pseudoinstrucciones

- ▶ Una **instrucción** en ensamblador se corresponde con una única instrucción máquina:
 - ▶ Ocupa 32 bits en RISC-V 32
 - ▶ Ej.: `addi t1, t0, 4`
- ▶ Una **pseudoinstrucción** se puede utilizar en un programa en ensamblador pero no se corresponde con ninguna instrucción máquina:
 - ▶ En el proceso de ensamblado se sustituyen por la secuencia de instrucciones máquina que realizan la misma funcionalidad.
 - Ej.: `mv t1, t2` se sustituye por: `addi t1, x0, t2`
 - `li t1, 0x00800010` se sustituye por: `lui t1, 0x00800` (20 bits)
`addi t1, t1, 0x010` (12 bits)

Ensamblador:

resumen



hola.s

```
.data
    msg_hola: .string "Hola mundo\n"

.text
main:
    # printf("Hola mundo\n") ;
    li a7 4
    la a0 msg_hola
    ecall
```

Contenidos

- ▶ Fundamentos básicos de la programación en ensamblador
- ▶ Ensamblador del RISC-V 32, modelo de memoria y representación de datos
 - ▶ RISC-V: registros y memoria
 - ▶ Directivas de ensamblador
 - ▶ Servicios del sistema
 - ▶ Instrucciones para el acceso a memoria
- ▶ Formato de las instrucciones y modos de direccionamiento
- ▶ Llamadas a procedimientos y uso de la pila

Llamadas al sistema

- ▶ CREATOR incluye un pequeño “sistema operativo”
 - ▶ Ofrece unos 12 servicios.
- ▶ Invocación:
 - ▶ Código de servicio en **a7**
 - ▶ Otros parámetros en registros concretos (ej.: **a0**, **fa0**)
 - ▶ Invocación mediante instrucción máquina **ecall**

```
# printf("hola mundo\n")  
li  a7 4  
la  a0 msg_hola  
ecall
```

Llamadas al sistema

| Servicio | Código de llamada (a7) | Argumentos (a7 y fa0) | Resultado (a7 y fa0) |
|--------------|------------------------|----------------------------|----------------------|
| print_int | 1 | a0 = integer | |
| print_float | 2 | fa0 = float | |
| print_double | 3 | fa0 = double | |
| print_string | 4 | a0 = string | |
| read_int | 5 | | integer en a0 |
| read_float | 6 | | float en fa0 |
| read_double | 7 | | double en fa0 |
| read_string | 8 | a0 = buffer, a1 = longitud | |
| sbrk | 9 | a0 = cantidad | dirección en a0 |
| exit | 10 | | |
| print_char | 11 | a0 (código ASCII) | |
| read_char | 12 | | a0 (código ASCII) |

Ejemplo: Hola mundo...

hola.s

.data

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

.text

```
main:
```

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

```
li a7 4
```

```
la a0 msg_hola
```

```
ecall
```

| Servicio | Código de llamada (a7) | Argumentos (a7 y fa0) |
|--------------|------------------------|-----------------------|
| print_int | 1 | a0 = integer |
| print_float | 2 | fa0 = float |
| print_double | 3 | fa0 = double |
| print_string | 4 | a0 = string |

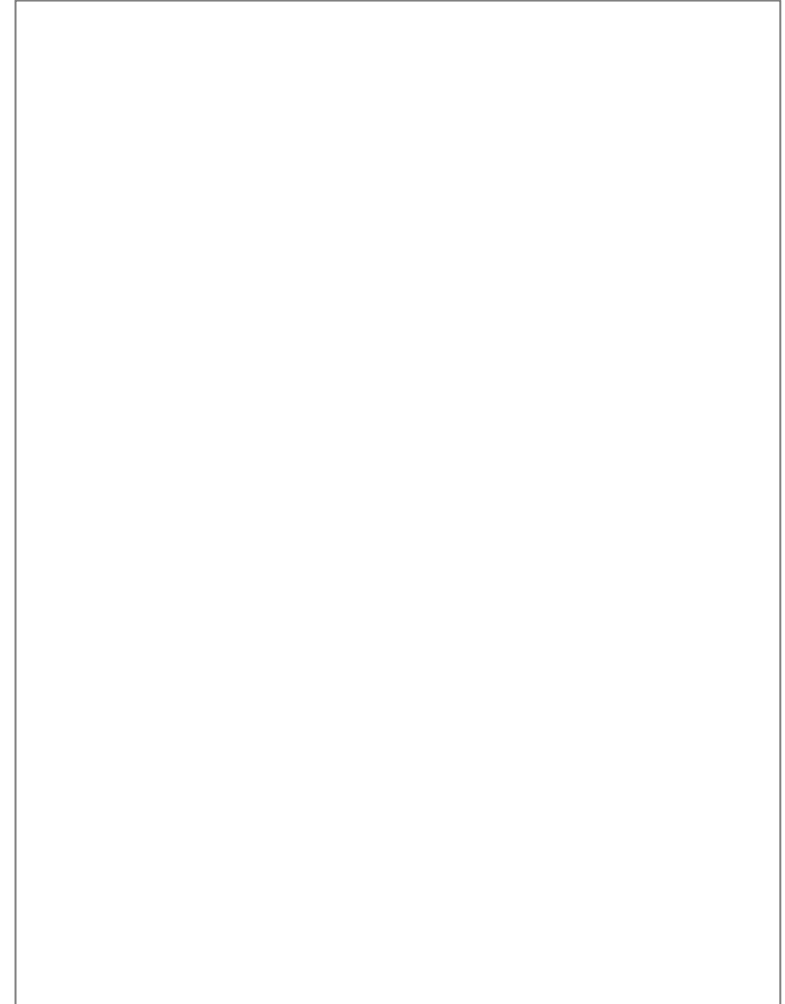
instrucción de llamada al sistema

Ejercicio

. . .

```
readInt(&valor) ;  
valor = valor + 1 ;  
printInt(valor) ;
```

. . .



Ejercicio (solución)

...

```
readInt(&valor) ;
valor = valor + 1 ;
printInt(valor) ;
```

...

| Servicio | Código de llamada (a7) | Argumentos (a7 y fa0) | Resultado (a7 y fa0) |
|--------------|------------------------|-----------------------|----------------------|
| print_int | 1 | a0 = integer | |
| print_float | 2 | fa0 = float | |
| print_double | 3 | fa0 = double | |
| print_string | 4 | a0 = string | |
| read_int | 5 | | integer en a0 |

...

```
# readInt(&valor)
li a7 5
ecall
mv t0 a0 # valor en t0
```

```
# valor = valor + 1
add t0 t0 1
```

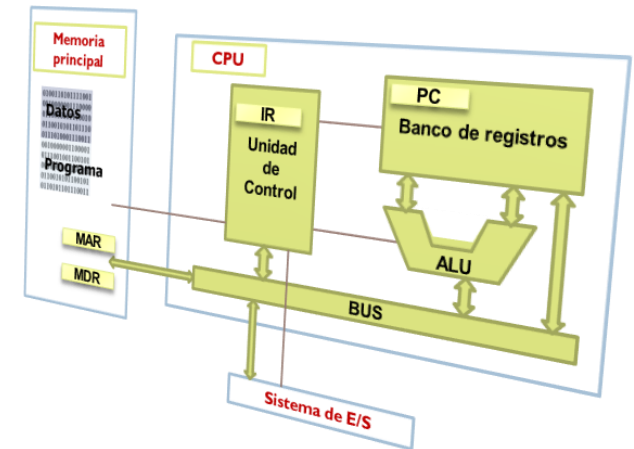
```
# printInt
mv a0 t0
li a7 1
ecall
```

...

Contenidos

- ▶ Fundamentos básicos de la programación en ensamblador
- ▶ Ensamblador del RISC-V 32, modelo de memoria y representación de datos
 - ▶ RISC-V: registros y memoria
 - ▶ Directivas de ensamblador
 - ▶ Servicios del sistema
 - ▶ Instrucciones para el acceso a memoria
- ▶ Formato de las instrucciones y modos de direccionamiento
- ▶ Llamadas a procedimientos y uso de la pila

Uso memoria



► Carga de una dirección en un registro:

► `la reg2, dir32 # reg2 ← dir32`

► Instrucciones de acceso a memoria (datos enteros):

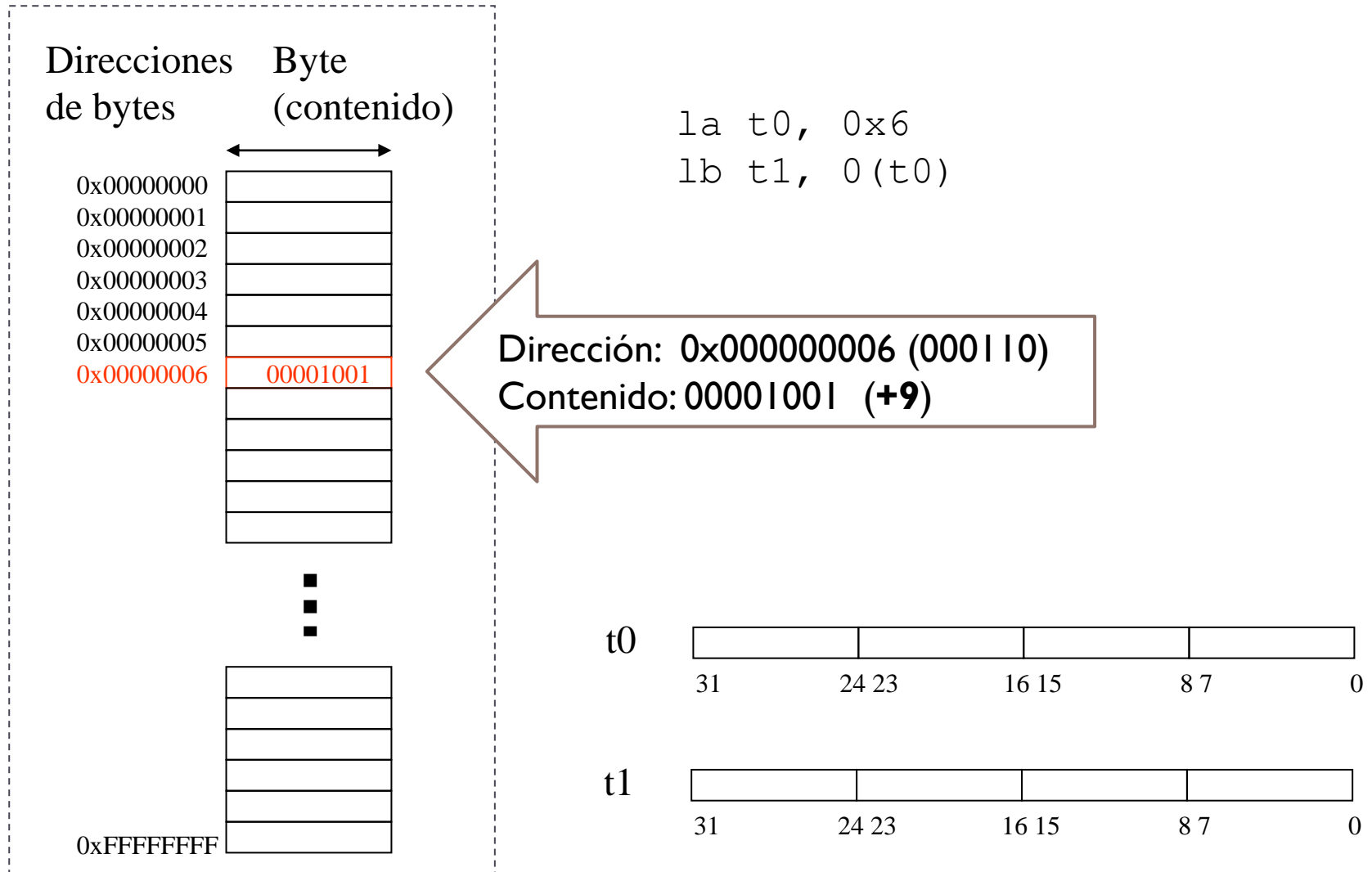
lw
sw
lb
lbu
sb

`reg1,`

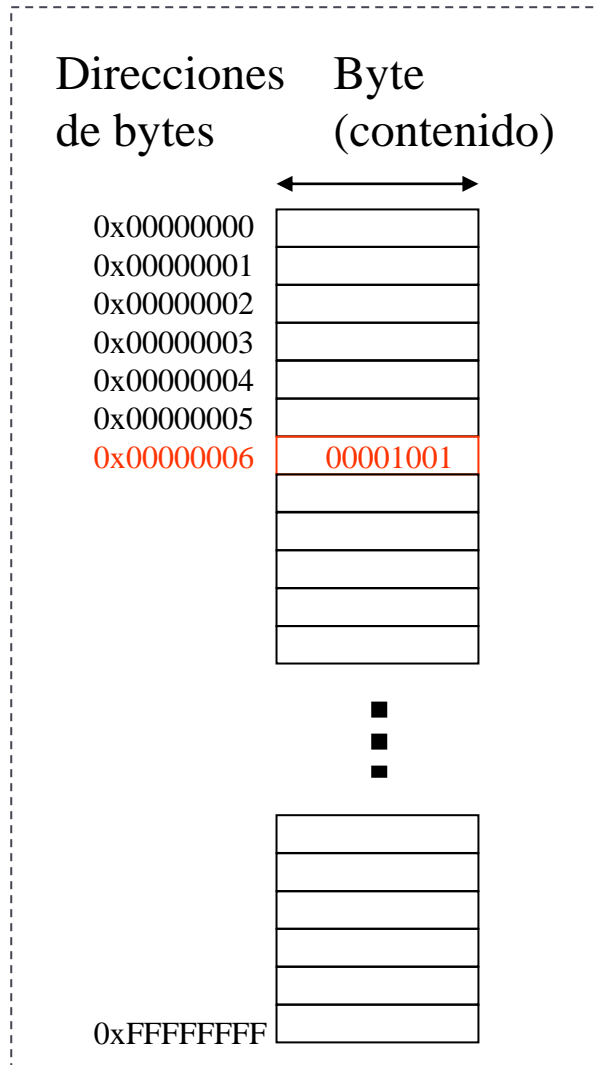
`num (reg2)`

- **num(registro)**: representa la dirección que se obtiene de sumar num con la dirección almacenada en el registro reg2

Accesso a bytes con lb (*load byte*)

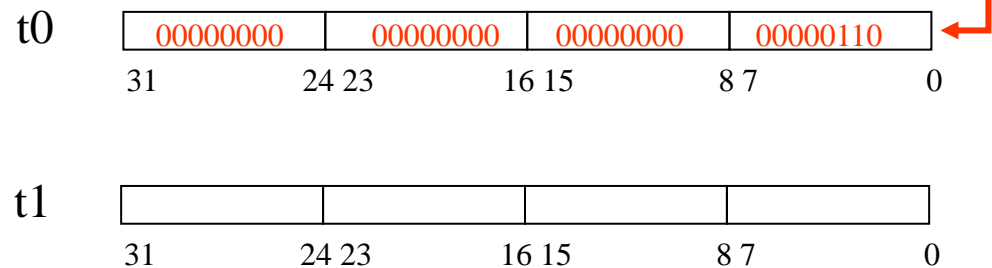


Acceso a bytes con lb (*load byte*)

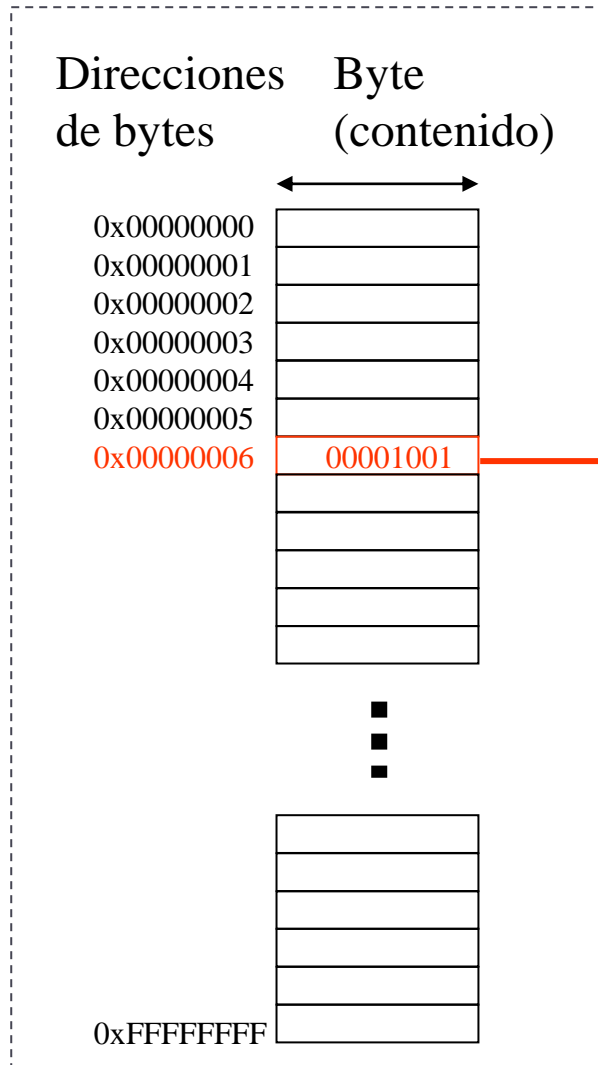


```
la t0, 0x6  
lb t1, 0(t0)
```

Se copia la dirección, no el contenido



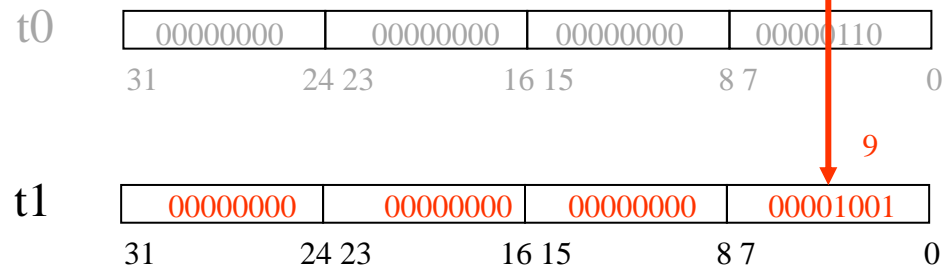
Acceso a bytes con lb (*load byte*)



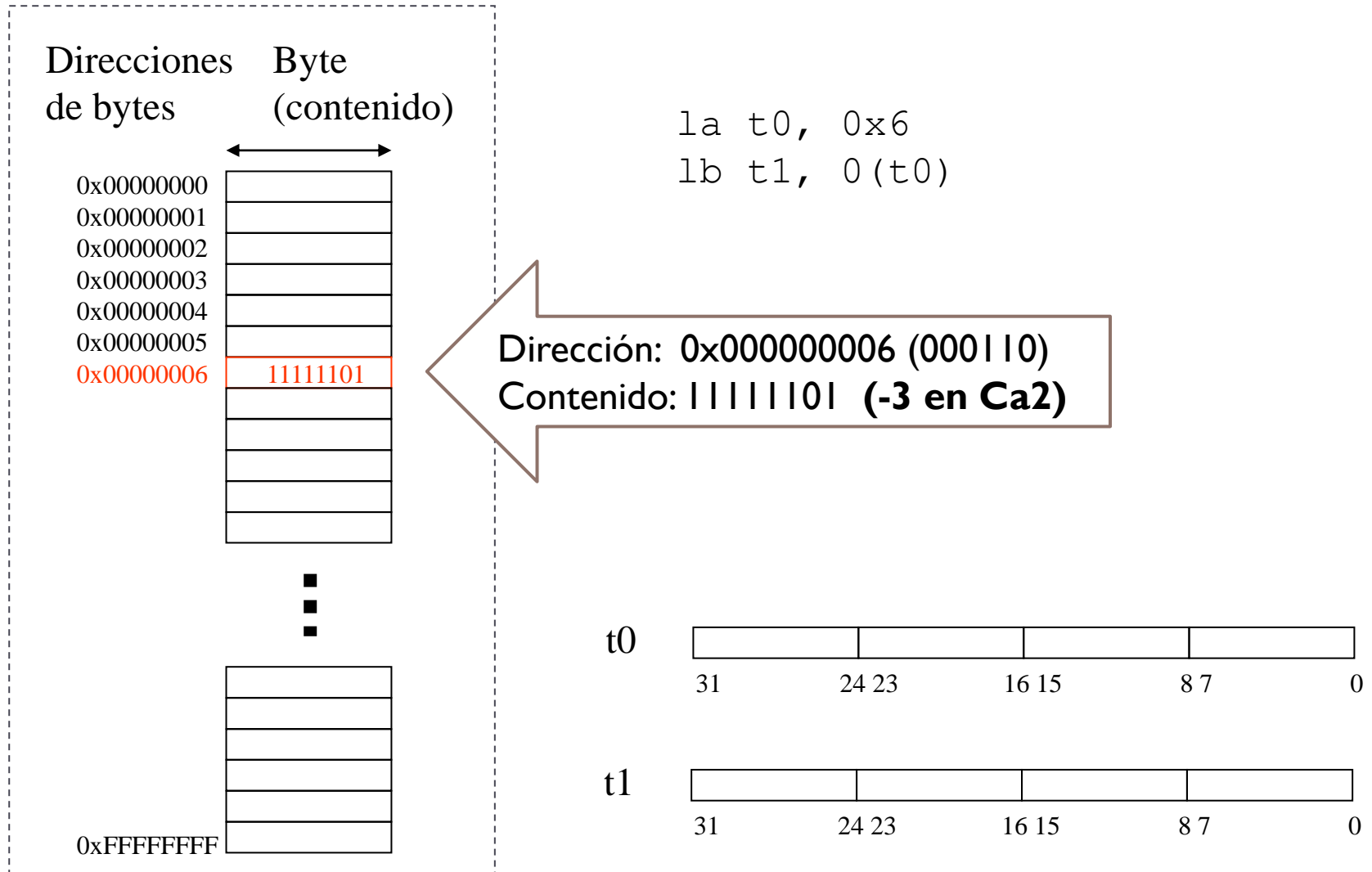
```
la t0, 0x6
```

```
lb t1, 0(t0)
```

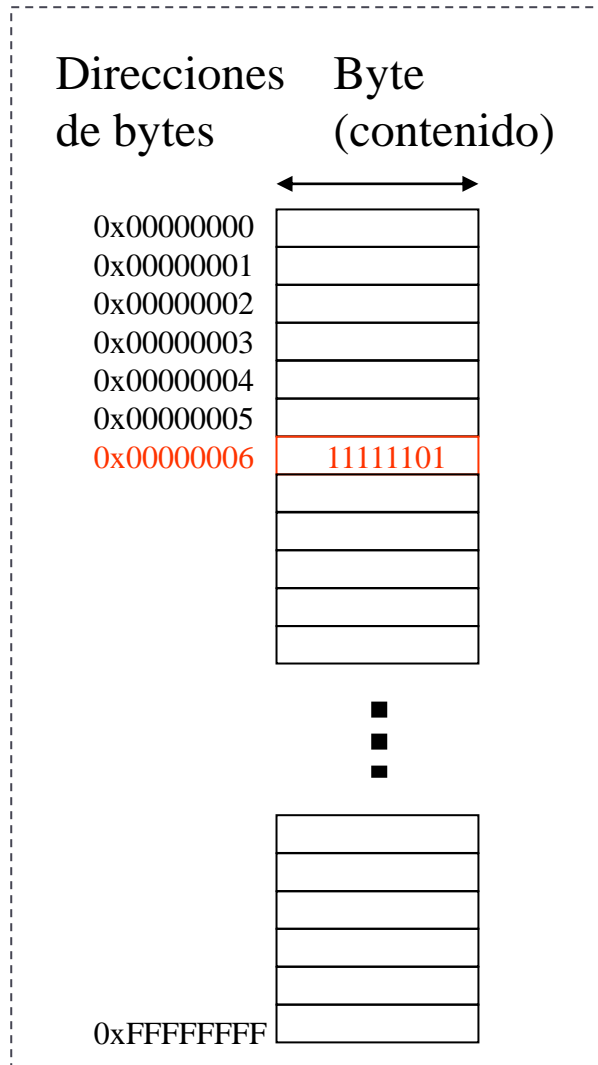
Se copia el contenido de 0x6 (un 9)



Acceso a bytes con lb (*load byte*)

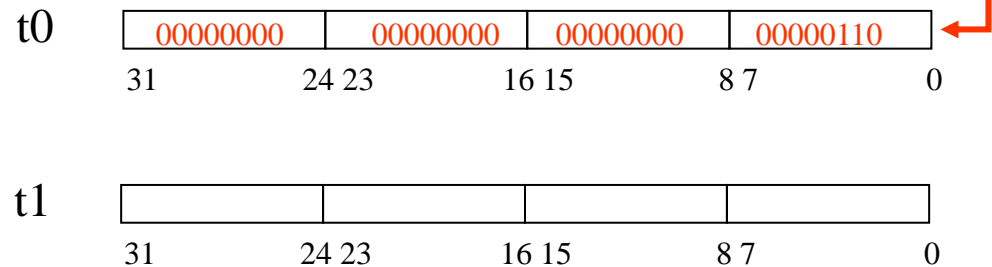


Acceso a bytes con lb (*load byte*)

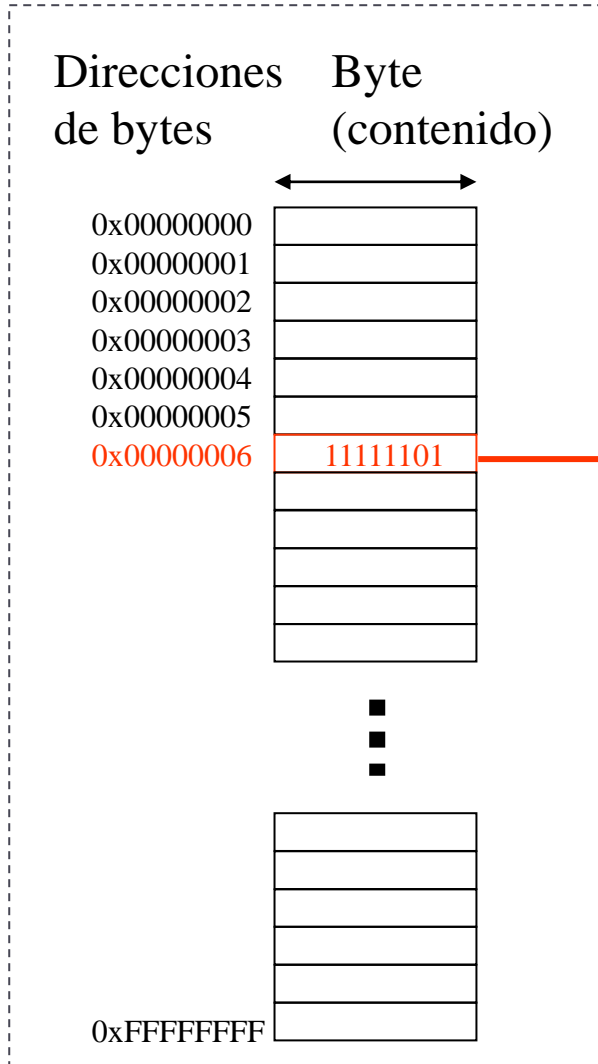


```
la t0, 0x6  
lb t1, 0(t0)
```

Se copia la dirección 0x6 a t0



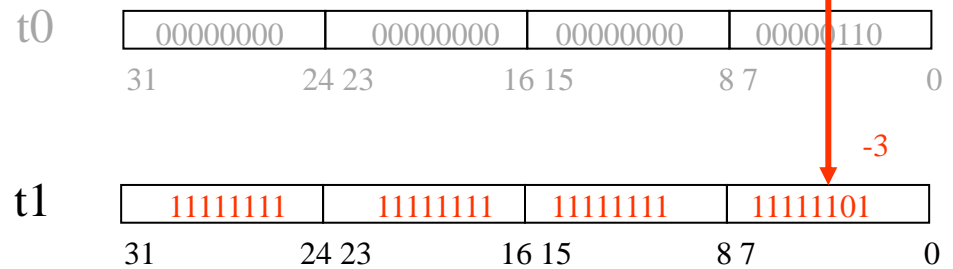
Accesso a bytes con lb (*load byte*)



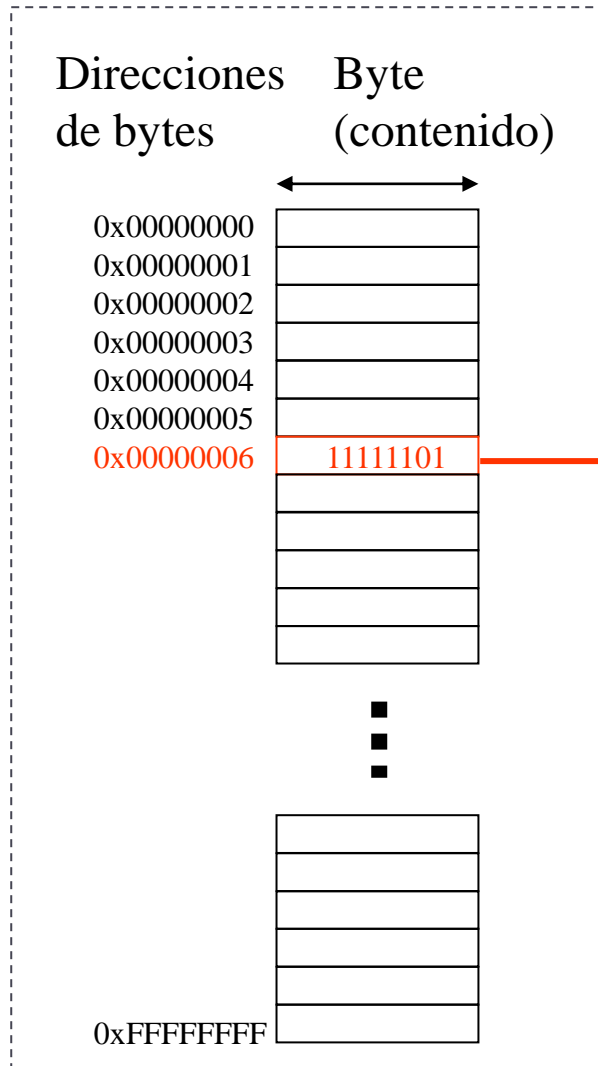
```
la t0, 0x6
```

1b t1, 0 (t0)

Se copia el contenido de 0x6 (un -3)



Acceso a bytes con lb (*load byte*)

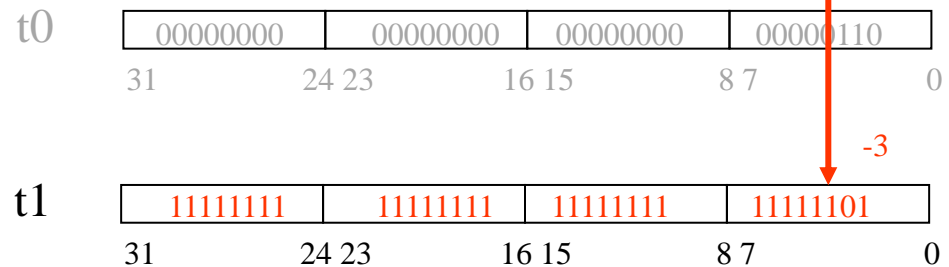


```
la t0, 0x6
```

```
lb t1, 0(t0)
```

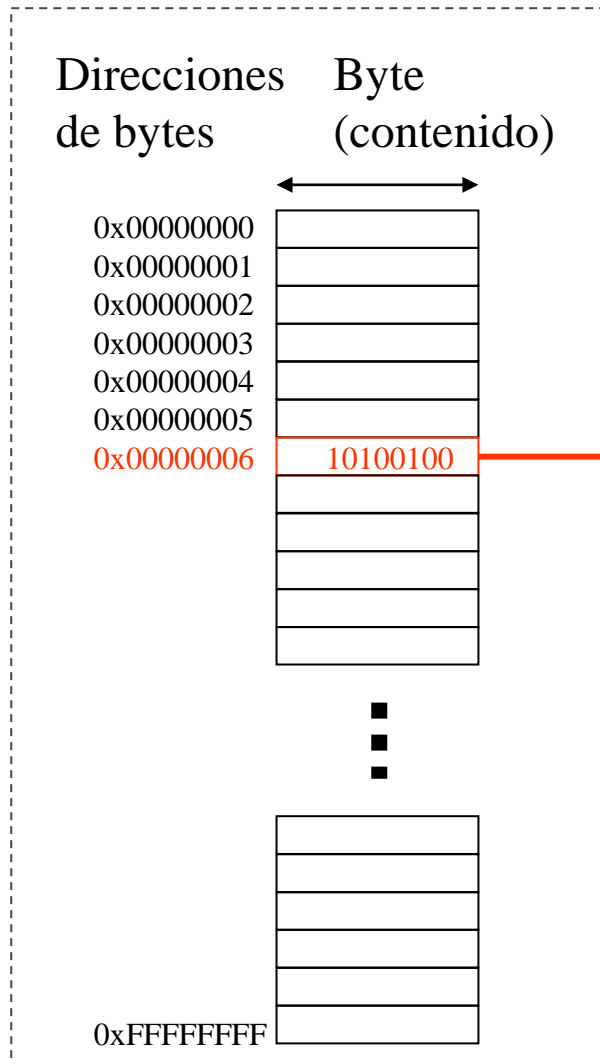
Se copia el contenido de 0x6 (un -3)

La instrucción lb mantiene el signo:
hace extensión de signo



Acceso a bytes con lb (*load byte*)

problemas accediendo a caracteres

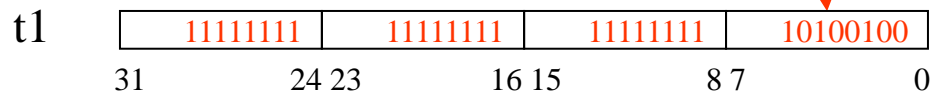


```
la t1, 0x6
lb t1, 0(t1)
```

Dirección: 0x000000006 (000110)

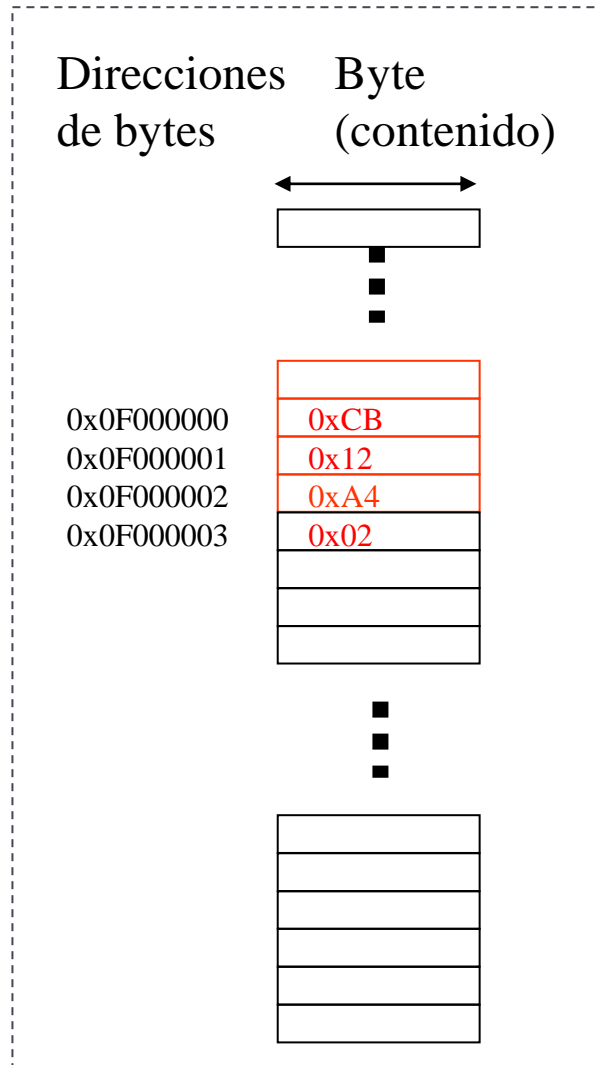
Contenido: 10100100 (código ASCII de la ñ: 164)

```
la t1, 0x6
lb t1, 0(t1)
```



Si se utiliza lb (se mantiene el signo),
el contenido del t1 **no** coincide con el
valor 164 (ñ)

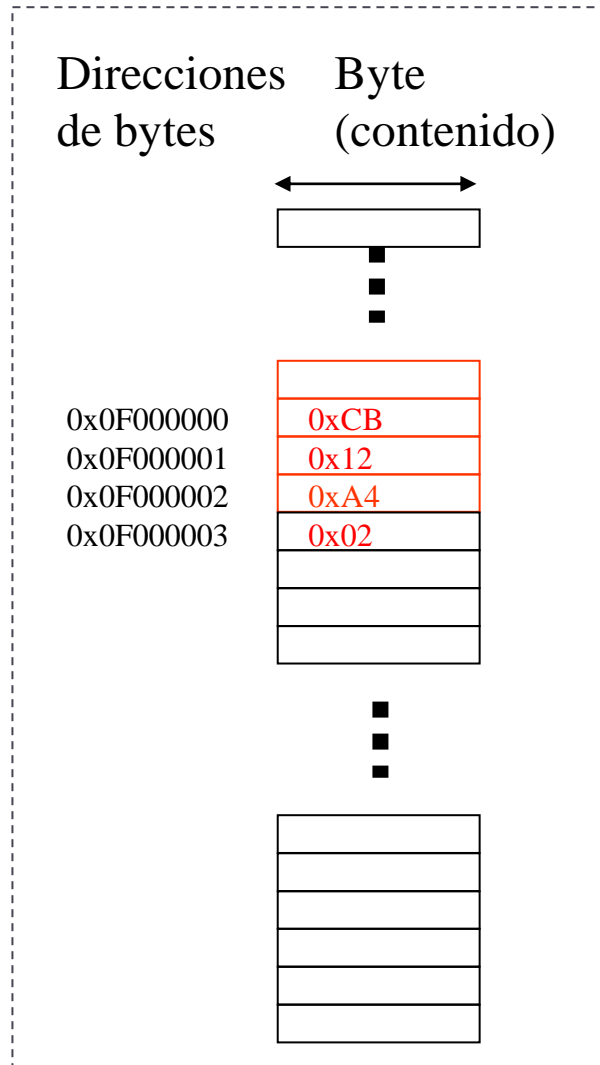
Acceso a bytes con lbu (*load byte unsigned*)



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

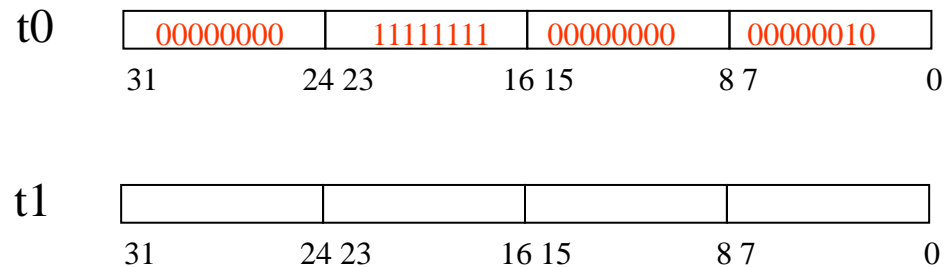


Acceso a bytes con lbu (*load byte unsigned*)

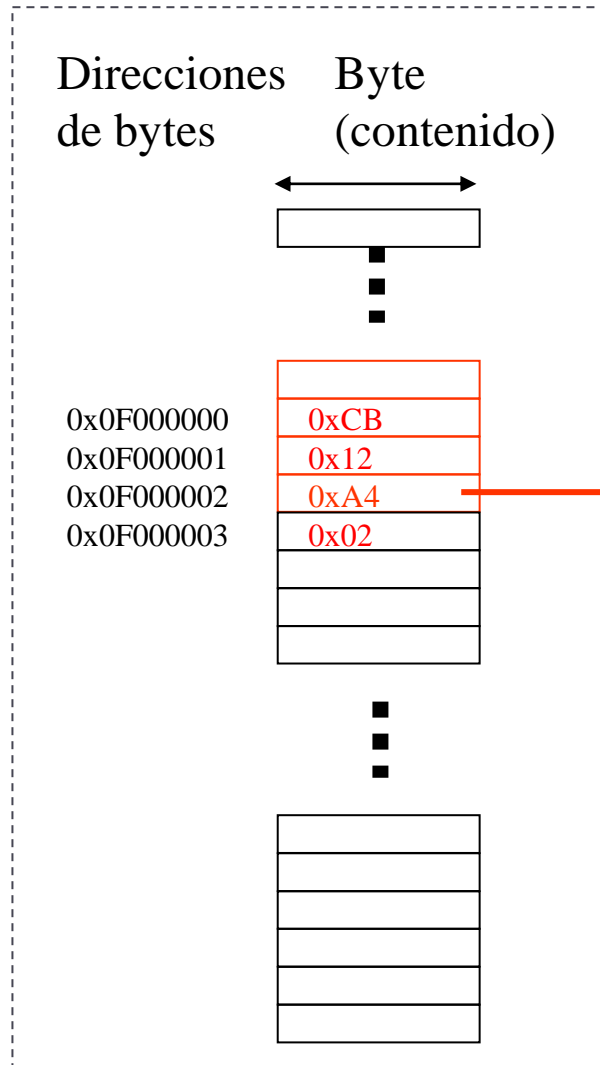


```
la    t0, 0x0F000002
lbu   t1, 0(t0)
```

Se copia la dirección, no el contenido



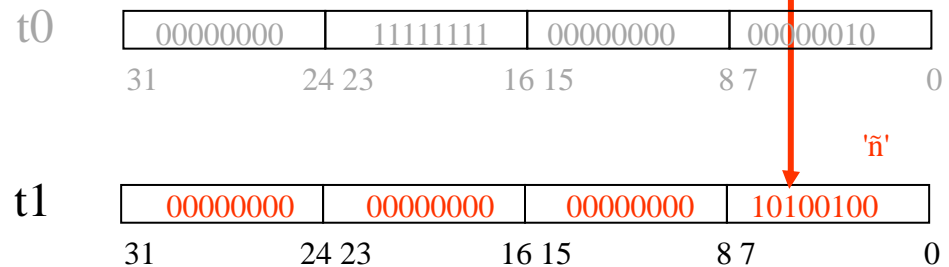
Acceso a bytes con lbu (*load byte unsigned*)



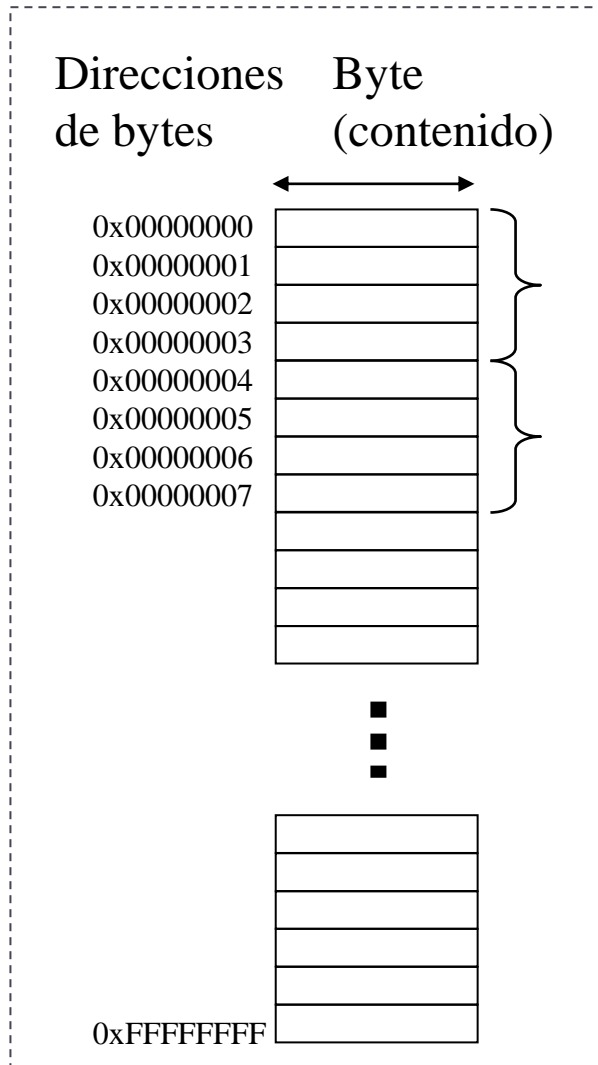
```
la    t0, 0x0F000002
```

```
lbu   t1, 0(t0)
```

Se copia el **byte** almacenado en la posición `0x0F000002` (**sin** extensión de signo)

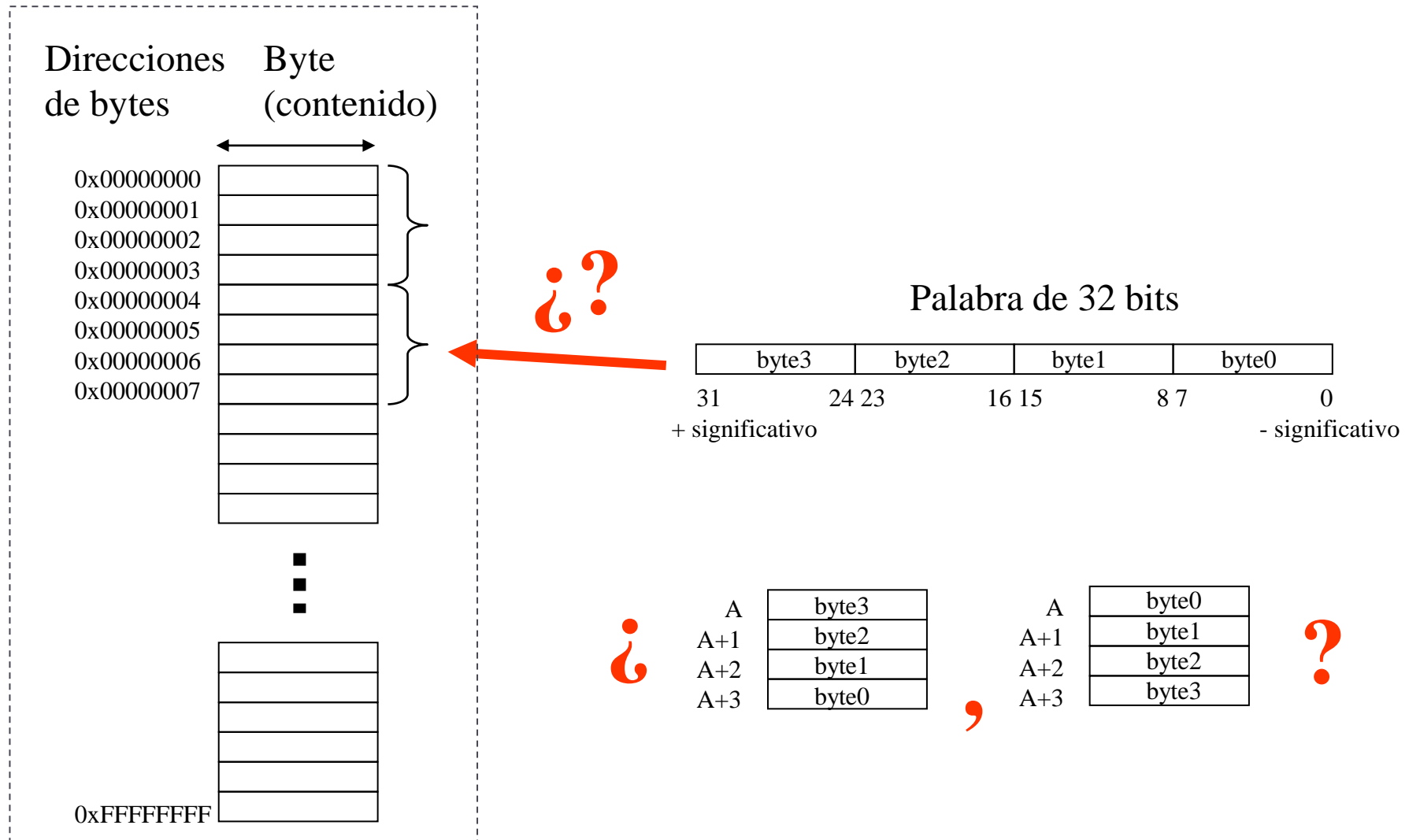


Acceso a palabras



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

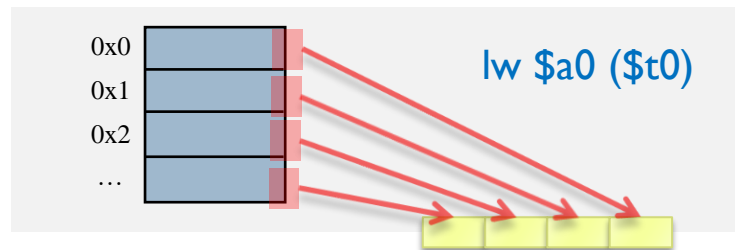


Almacenamiento de palabras en la memoria

Ordenamiento de bytes

- ▶ Hay dos tipos de ordenamiento de bytes:

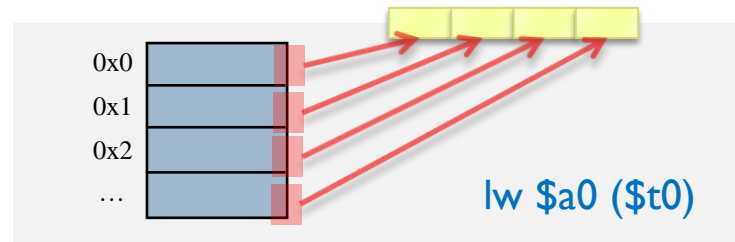
- ▶ Little-endian (Dirección 'pequeña' termina la palabra...)



AMD



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



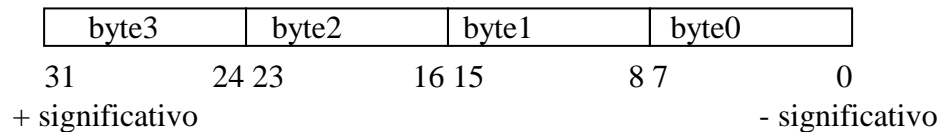
IBM
PowerPC™ (bi-endian)



Almacenamiento de palabras en la memoria

Ejemplo

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 |

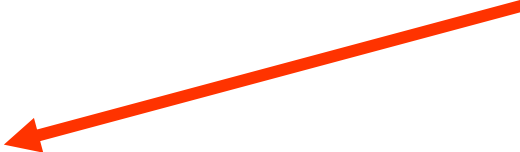
BigEndian

| | |
|-----|----------|
| 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: **00011011**000000000000000000000000
que no es el 27

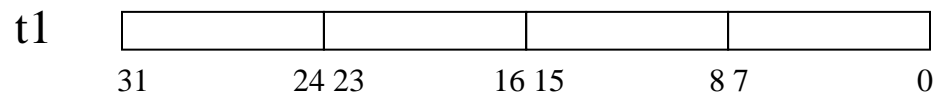
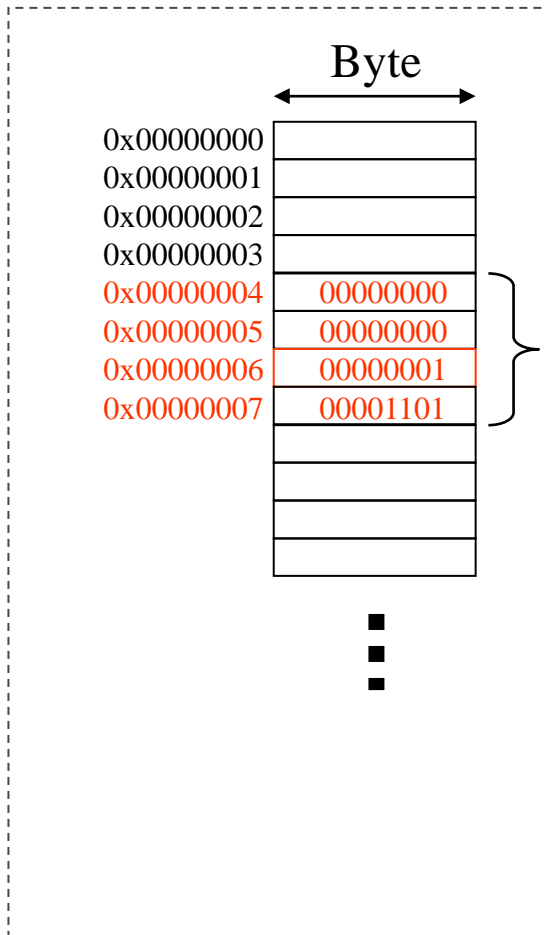
Acceso a palabras

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

```
lw t1, 0x4(x0)
```

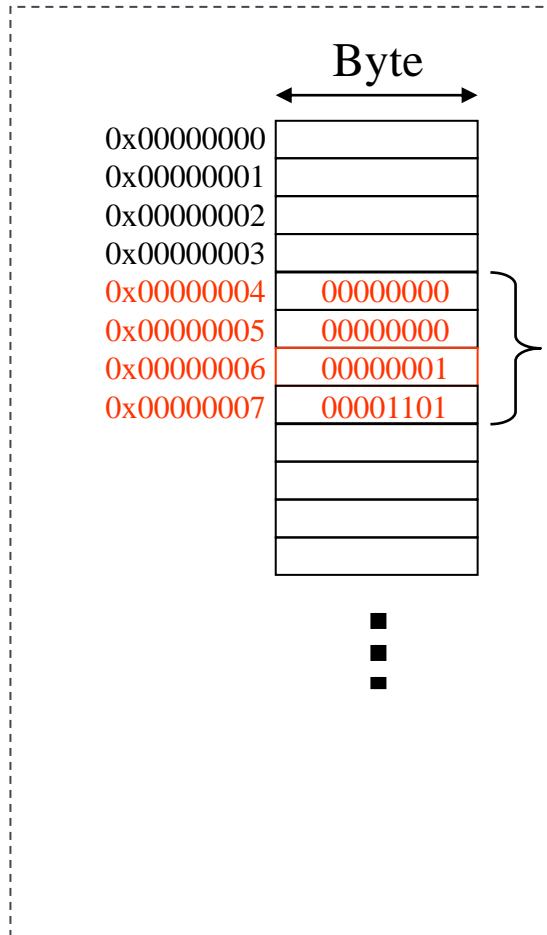
Dirección 0x00000004 (000000.....00100)

Contenido: 00000000000000000000000000000000100001101₍₂₎ = 269₍₁₀₎



Acceso a palabras

```
lw t1, 0x4(x0)
```

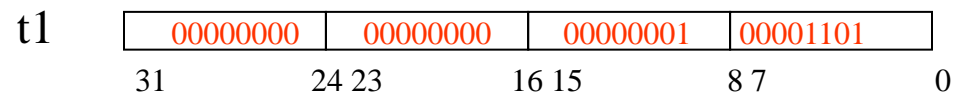


Dirección 0x00000004 (000000....00100)

[illegible]

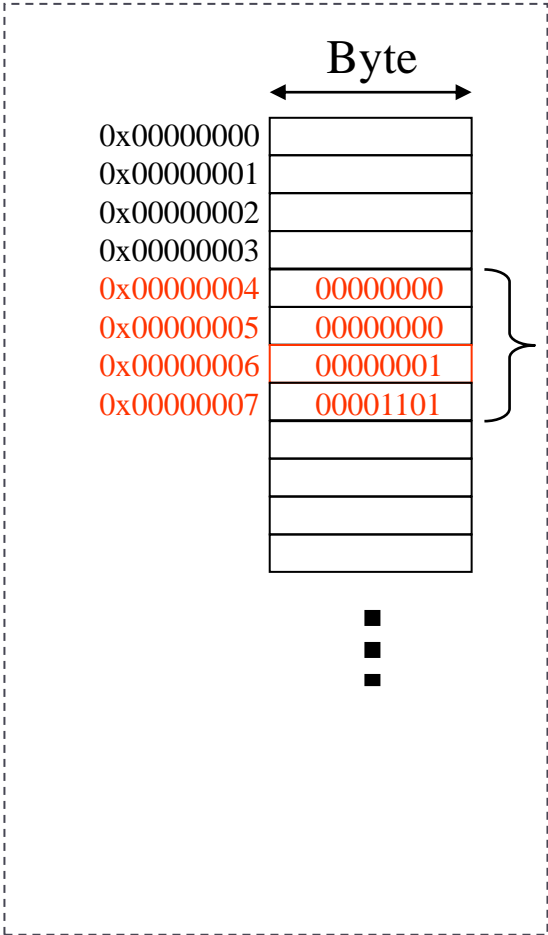
Computador Big Endian

Se copia la palabra



Acceso a palabras

```
lw t1, 0x4(x0)
```



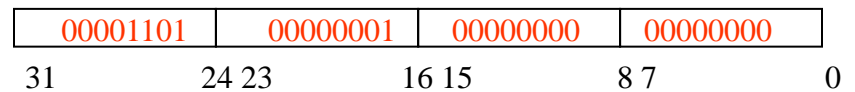
Dirección 0x00000004 (000000....00100)

[illegible]

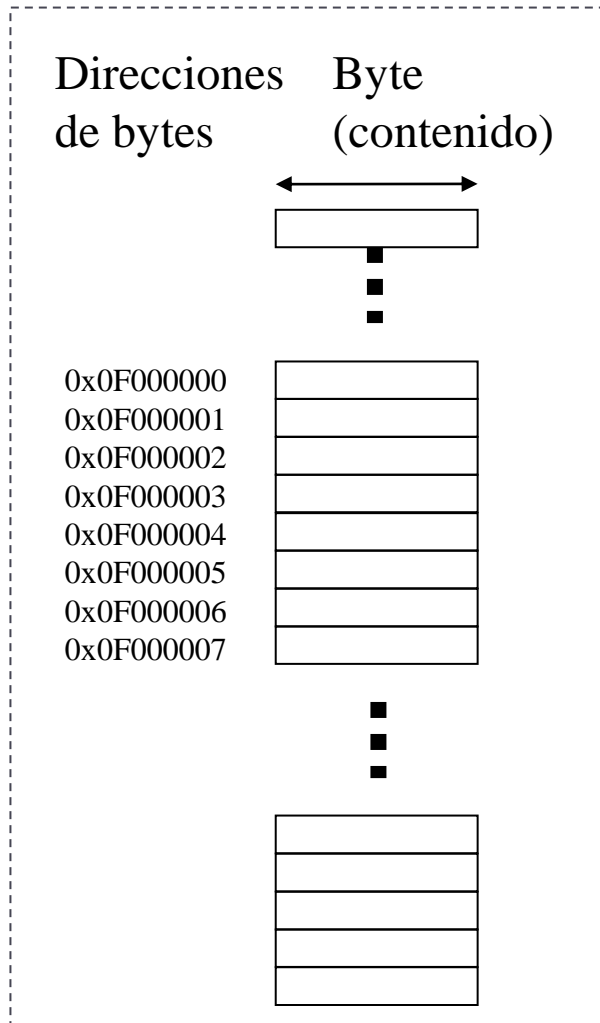
Computador Little Endian

Se copia la palabra

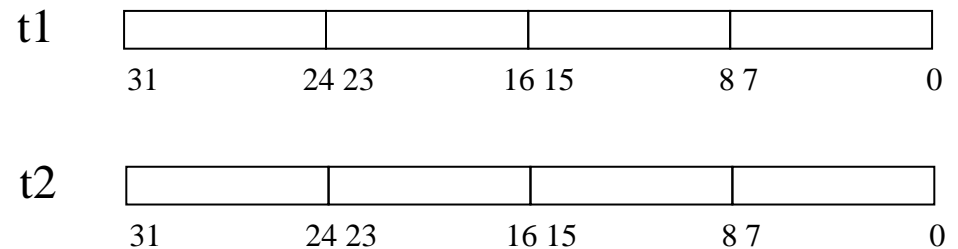
t1



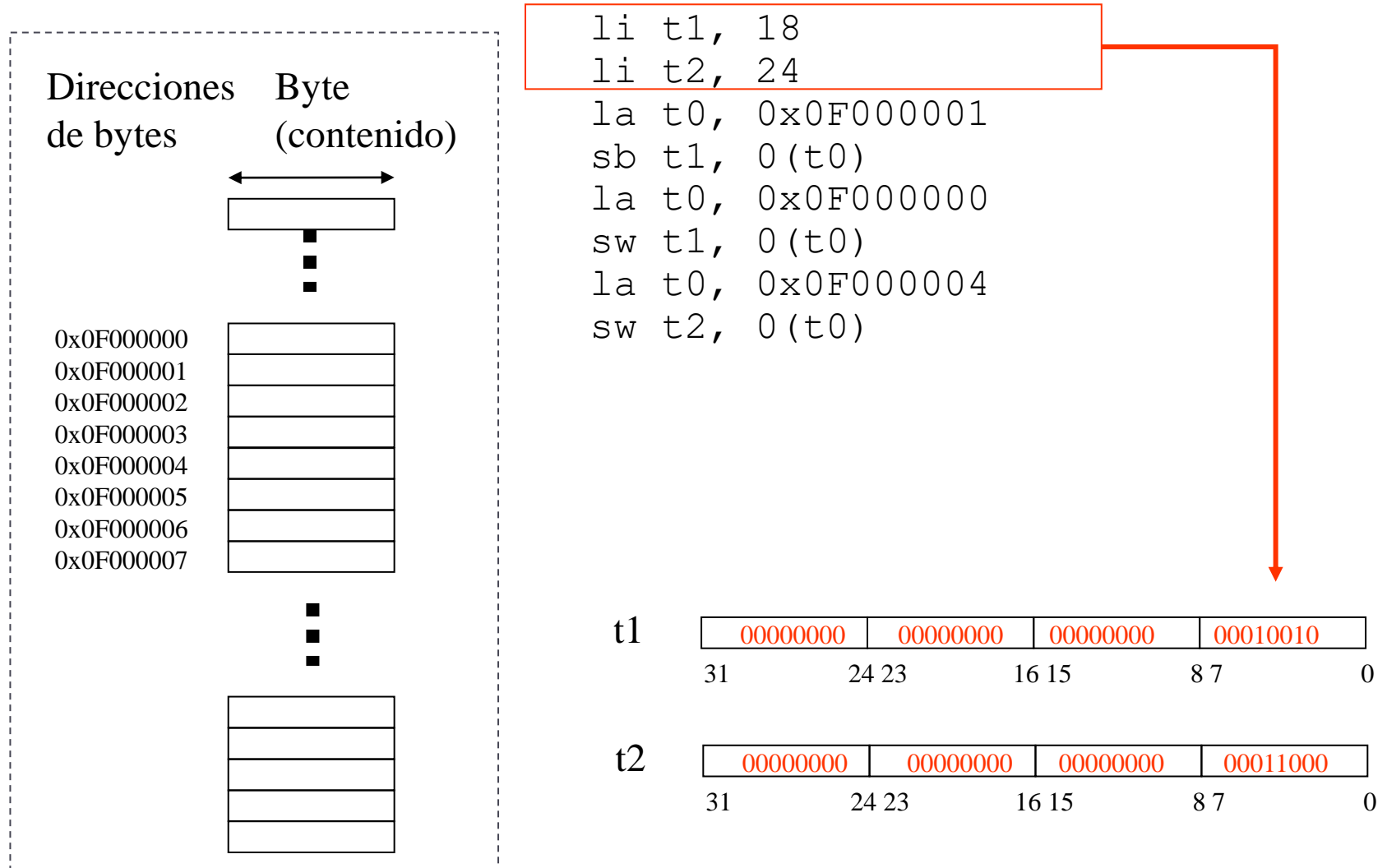
Escritura en memoria



```
li t1, 18
li t2, 24
la t0, 0x0F000001
sb t1, 0(t0)
la t0, 0x0F000000
sw t1, 0(t0)
la t0, 0x0F000004
sw t2, 0(t0)
```



Escritura en memoria



Escritura en memoria

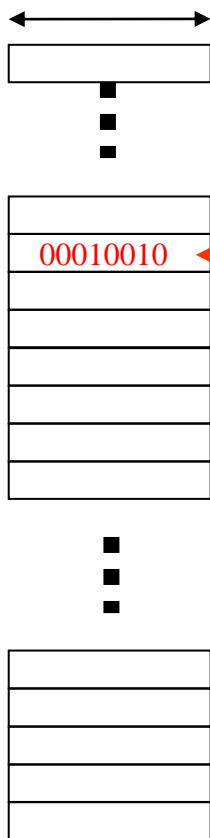
escribir un byte

```
li t1, 18
li t2, 24
la t0, 0x0F000001
sb t1, 0(t0)
la t0, 0x0F000000
sw t1, 0(t0)
la t0, 0x0F000004
sw t2, 0(t0)
```

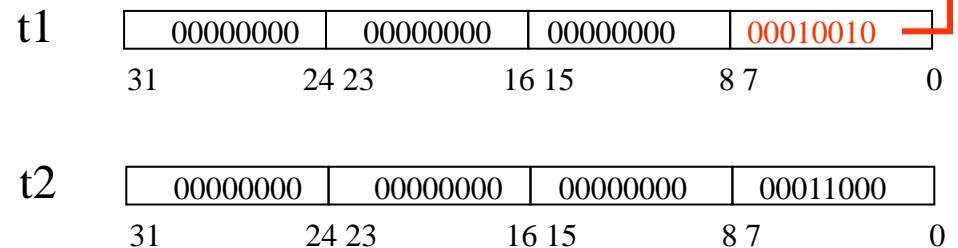
Direcciones
de bytes

Byte
(contenido)

0x0F000000
0x0F000001
0x0F000002
0x0F000003
0x0F000004
0x0F000005
0x0F000006
0x0F000007



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

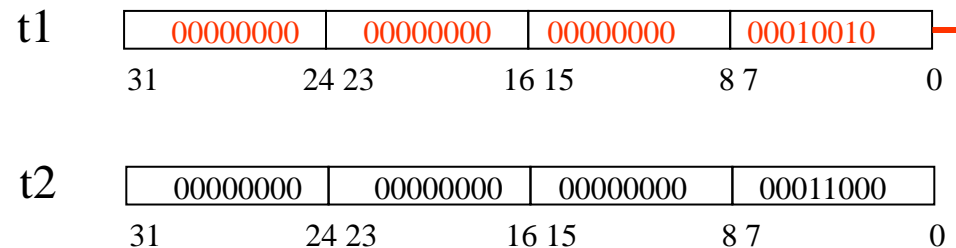


Escritura en memoria

escribir una palabra

```
li t1, 18
li t2, 24
la t0, 0x0F000001
sb t1, 0(t0)
la t0, 0x0F000000
sw t1, 0(t0)
la t0, 0x0F000004
sw t2, 0(t0)
```

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



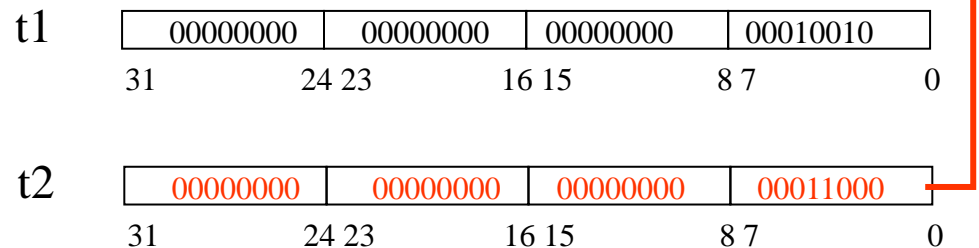
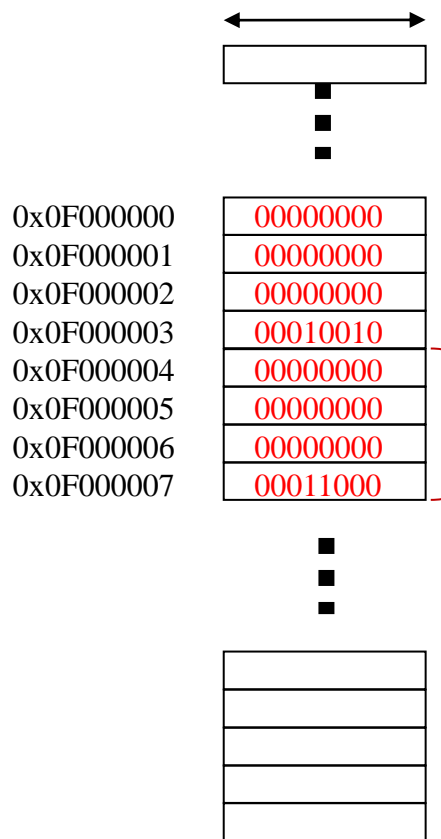
Escritura en memoria

escribir una palabra

```
li t1, 18
li t2, 24
la t0, 0x0F000001
sb t1, 0(t0)
la t0, 0x0F000000
sw t1, 0(t0)
la t0, 0x0F000004
sw t2, 0(t0)
```

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

Direcciones de bytes Byte (contenido)



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 (instrucciones y datos) tiene una dirección de memoria
- ▶ En un computador como el RISC-V 32 (de 32 bits)
 - ▶ Los registros son de 32 bits
 - ▶ En la memoria se pueden almacenar bytes (8 bits)
 - ▶ memoria → registro: `lb/lbu rd num(rs1)`
 - ▶ registro → memoria: `sb rd num(rs1)`
 - ▶ En la memoria se pueden almacenar palabras (32 bits)
 - ▶ memoria → registro: `lw rd num(rs1)`
 - ▶ registro → memoria: `sw rd num(rs1)`

`num(reg)`: representa la dirección que se obtiene de sumar `num` con la dirección almacenada en el registro

Ejemplos de instrucciones

- ▶ `la t0, 0x0F000002`
 - ▶ Direccionamientos directo a registro + inmediato.
Se carga en t0 el valor 0x0F000002
- ▶ `lbu t0, etiqueta(x0)`
 - ▶ Direccionamientos directo a reg. + relativo a registro base.
Se carga en t0 el byte en la dirección de memoria `etiqueta`
- ▶ `lb t0, 0(t1)`
 - ▶ Direccionamientos directo a reg. + relativo a registro base.
Se carga en t0 el byte en la posición de memoria almacenada en `t1+0`
- ▶ `la t0, 0x0F000000`
`sw t1, 0(t0)`
 - ▶ Copia la palabra almacenada en t1 en la dirección 0x0F000000
- ▶ `la t0, 0x0F000000`
`sb t1, 0(t0)`
 - ▶ Copia el byte almacenado en t1 (el menos significativo) en la dirección 0x0F000000

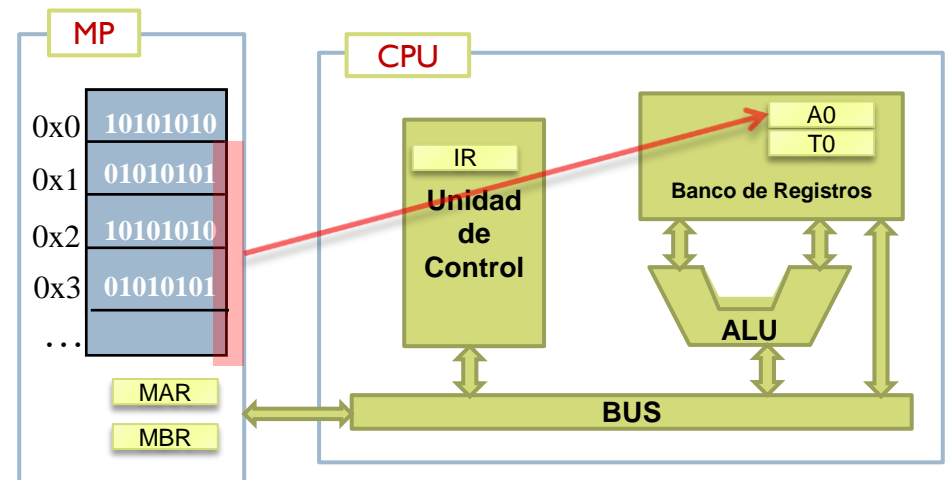
Operaciones de memoria: coma flotante

- ▶ **flw rd, 10(rs1)**
 - ▶ Carga el valor simple precisión almacenado en la dirección (rs1+10) en el registro de coma flotante rd.
- ▶ **fsw rs, 10(rd1)**
 - ▶ Almacena el valor simple precisión del registro rs en la dirección (rd1+10).
- ▶ **fld rd, 10(rs1)**
 - ▶ Carga el valor doble precisión almacenado en la dirección (rs1+10) en el registro rd.
- ▶ **fsd rs, 10(rd1)**
 - ▶ Almacena el valor doble precisión del registro rs en la dirección (rd1+10).

Transferencia de datos: peculiaridades alineamiento y tamaño de acceso

► Peculiaridades:

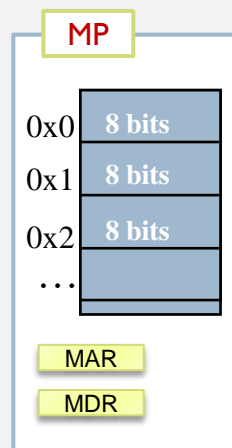
- Tamaño de acceso por defecto
- Alineamiento de los elementos en memoria



Direccionamiento a nivel de palabra o de byte

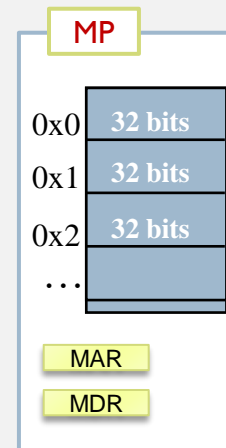
- ▶ La **memoria principal** es similar a un gran vector de una dimensión
- ▶ Una **dirección de memoria** es el índice del vector
- ▶ Hay dos **tipos de direccionamiento**:

- ▶ 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**

Alineación de datos

- ▶ **En general:**

- ▶ Un dato que ocupa K bytes está alineado cuando la dirección D utilizada para accederlo cumple que:

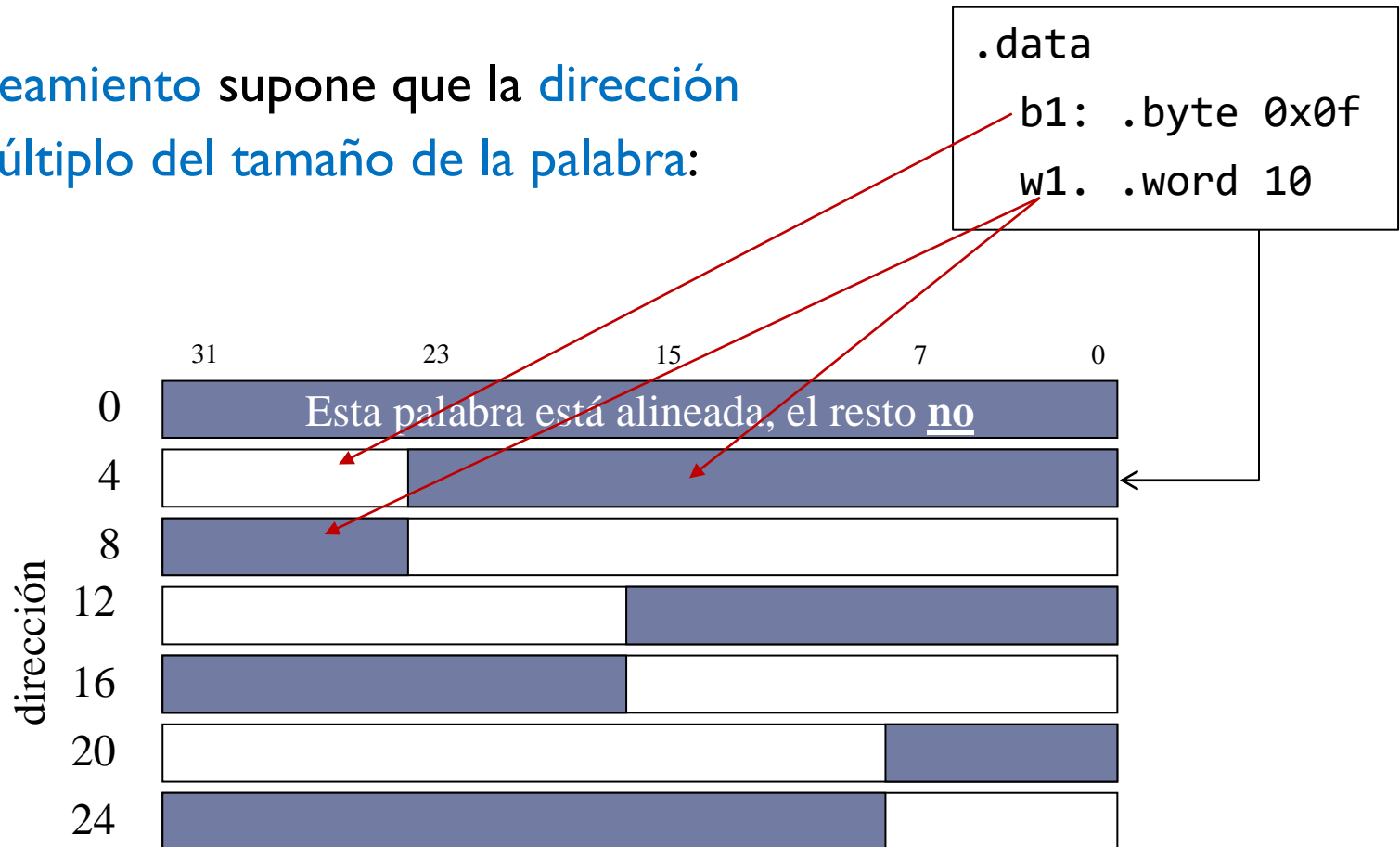
$$D \bmod K = 0$$

- ▶ **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

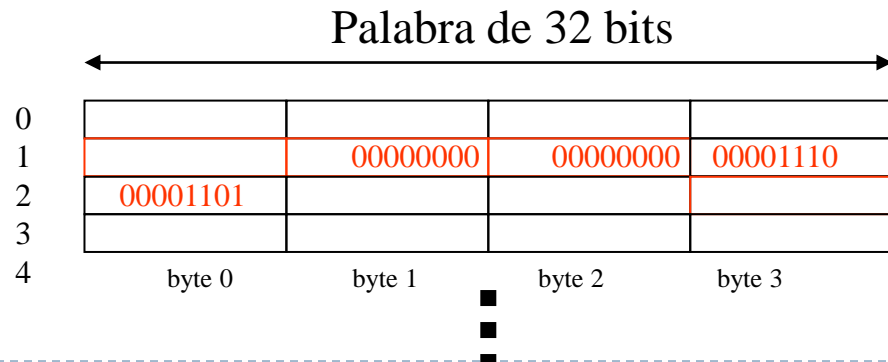
Alineamiento: ejemplo

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

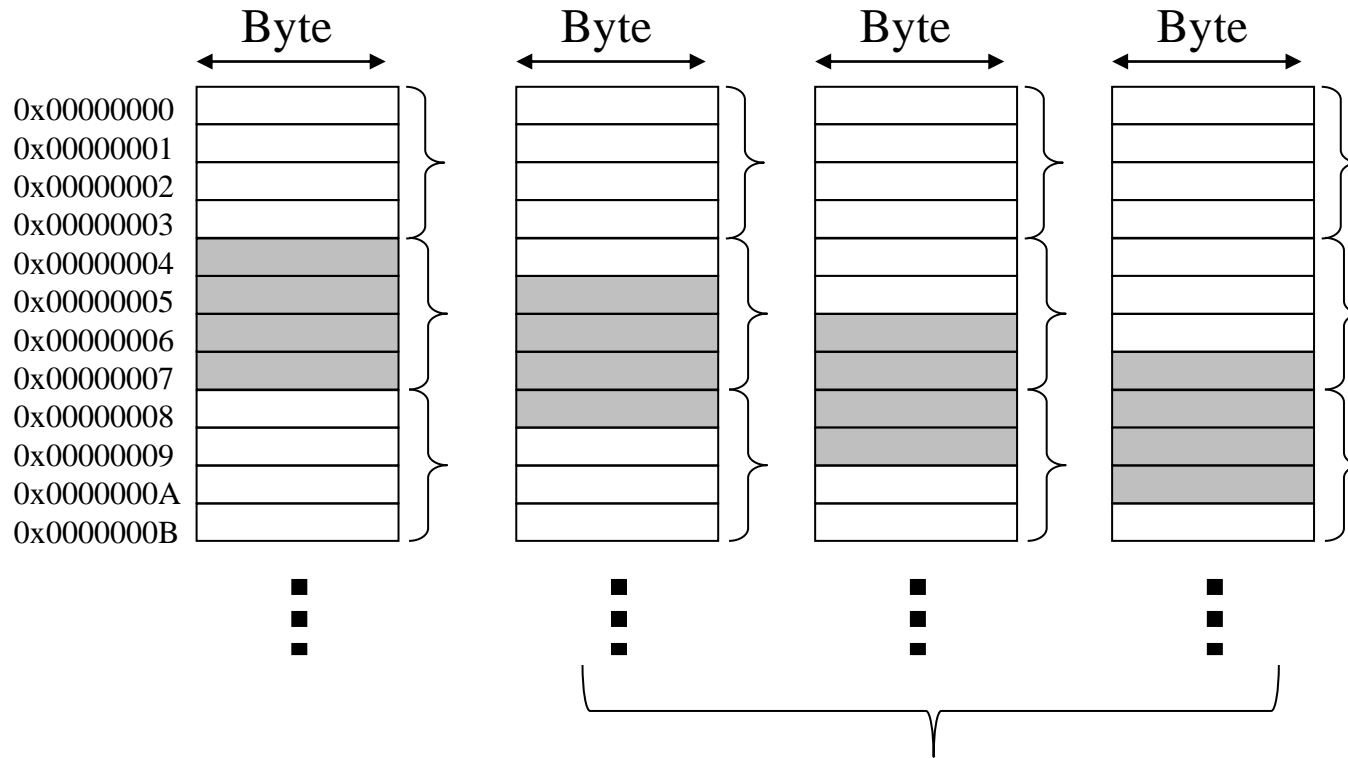


Alineación de datos

- ▶ 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



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

```
main:  la t0, b1  
        li t1, 1  
        sb t1, 0(t0)
```

...

Tipos de datos caracteres

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

```
...
```

```
main ()
```

```
{
```

```
    c1 = c2;
```

```
    ...
```

```
}
```

```
.data
```

```
c1: .zero 1 # 1 byte
```

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

```
...
```

```
.text
```

```
main: la t0 c1
```

```
      la t1 c2
```

```
      lbu t2 0(t1)
```

```
      sb t2 0(t0)
```

```
...
```

Tipos de datos enteros

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

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

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

```
.text

main:
    la t0 op1
    lw t1 0(t0)
    la t0 op2
    lw t2 0(t0)
    add t3 t1 t2
    la t0 resultado
    sw t0 0(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: .zero 4 # 4 bytes  
op1:      .word 100  
op2:      .word -10  
...
```

```
.text
```

```
main:
```

```
    la t0 op1  
    lw t1 0(t0)  
    la t0 op2  
    lw t2 0(t0)  
    add t3 t1 t2  
    la t0 resultado  
    sw t0 0(t4)
```

...

Tipo de datos básicos

float

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

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

.data

.align 2

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

...

.text

```
main: flw      ft0 op1(x0)
      flw      ft1 op2(x0)
      fadd.s   ft3 ft1 ft2
      fsw      ft3 resultado(x0)
      ...
```

Tipo de datos básicos

double

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

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

.data

.align 3

```
resultado:  .zero 8
op1:        .double 100
op2:        .double -10.27
```

...

.text

```
main: fld      ft0 op1(x0)
      fld      ft1 op2(x0)
      fadd.d   ft3 ft1 ft2
      fsd      ft3 resultado(x0)
      ...
```

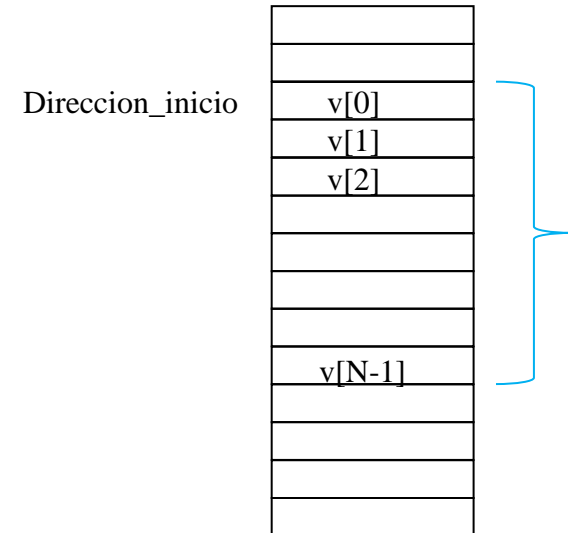
Tipo de datos básicos

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: .zero 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: .zero 20  #5 elem.*4 bytes
```

```
.text
```

```
main:
```

```
    li    t0, 16  
    la    t1, vec  
    add   t3, t1, t0  
    li    t2, 8  
    sw    t2, 0(t3)
```

```
...
```

Tipo de datos básicos

vectores

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

```
...
```

```
main ()
```

```
{
```

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

```
}
```

```
.data
```

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

```
vec: .zero 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 $v[5]$?
 - ▶ `li t1, 20`
 - ▶ `lw t0, v(t1)`

Tipo de datos básicos

cadenas de caracteres

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

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

.data

```
c1:  .zero    1    # 1 byte  
c2:  .byte    'h'  
ac1: .string  "hola"  
...
```

.text

```
main:  
    li a7 4  
    la a0 ac1  
    ecall  
    ...
```

Ejercicio

```
// variables globales
```

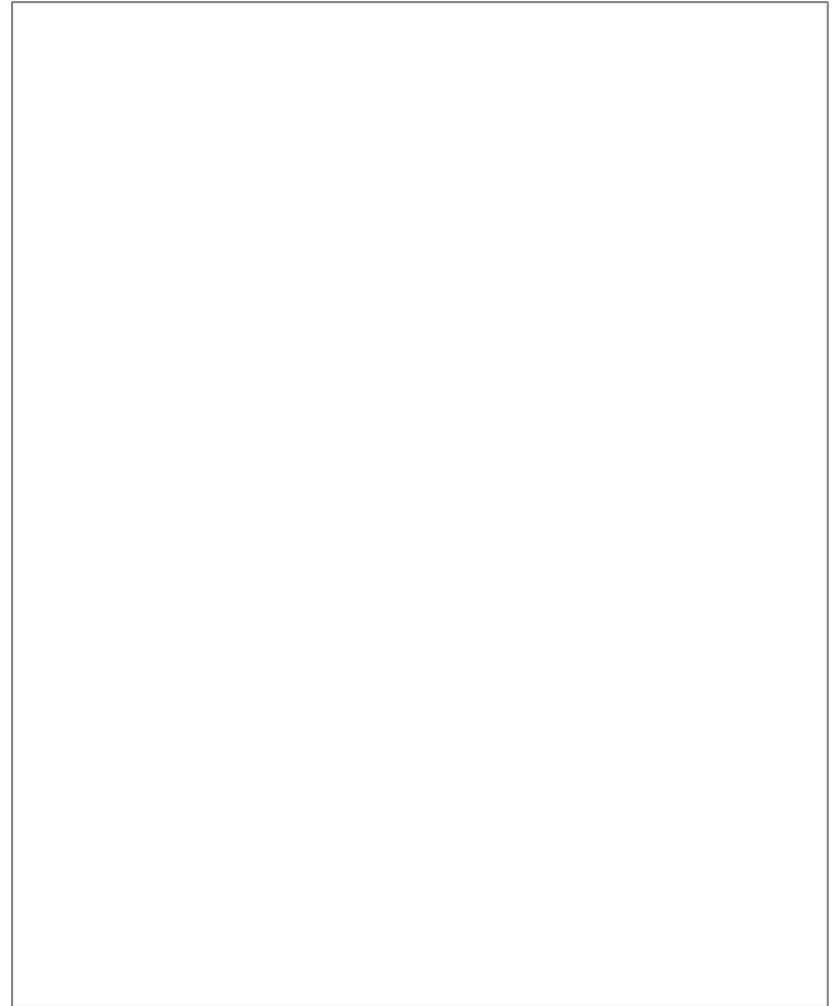
```
char v1;
```

```
int v2 ;
```

```
float v3 = 3.14 ;
```

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

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



Ejercicio (solución)

```
// variables globales
```

```
char v1;
```

```
int v2 ;
```

```
float v3 = 3.14 ;
```

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

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

```
.data
```

```
v1: .byte 0
```

```
.align 2
```

```
v2: .zero 4
```

```
v3: .float 3.14
```

```
v4: .string "ec"
```

```
.align 2
```

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

Ejercicio (solución)

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

```
.data

v1: .byte 0
.align 2
v2: .zero 4
v3: .float 3.14

v4: .string "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 |
| | (20) | 0x0111 |
| | (20) | 0x0112 |
| | (20) | |

```
.data

v1: .byte 0
    .align 2
v2: .zero 4
v3: .float 3.14

v4: .string "ec"

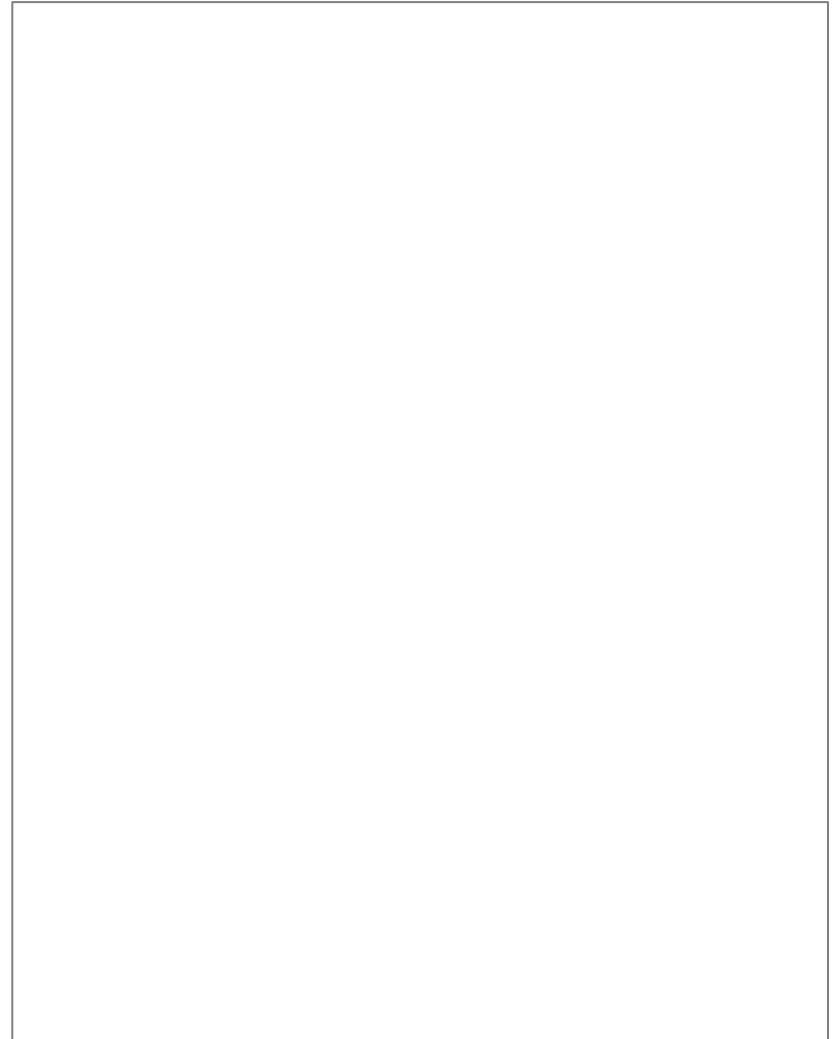
    .align 2
v5: .word 20, 22
```

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: .zero 1 # 1 byte
```

```
c2: .byte 'h'
```

```
ac1: .string "hola"
```

```
.align 2
```

```
c: .zero 4 # puntero => dirección
```

...

```
.text
```

```
main:    la    t0, ac1
```

```
        li    a0, 0
```

```
        lbu   t1, 0(t0)
```

```
    bucl: beq  x0, t1, fin1
```

```
        addi  t0, t0, 1
```

```
        addi  a0, a0, 1
```

```
        lbu   t1, 0(t0)
```

```
        j     bucl
```

```
    fin1: li  a7 1
```

```
        ecall
```

...

Vectores y cadenas

► En general:

► `lw t0, 4(s3) # t0 ← M[s3+4]`

► `sw t0, 4(s3) # M[s3+4] ← t0`

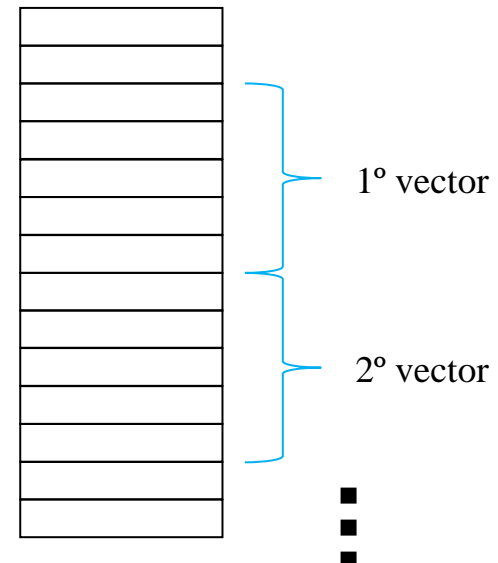
Tipos de datos básicos

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: .zero 20 #5 elem.*4 bytes
mat: .word 11, 12, 13
     .word 21, 22, 23
     ...
```

```
.text
main:
    li    t0    0
    lw    t1    mat(t0)
    li    t0    12
    lw    t2    mat(t0)
    add   t3    t1 t2
    li    t0    4
    sw    t3    mat(t0)
    ...
```

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

Grupo ARCOS

uc3m | Universidad **Carlos III** de Madrid

Tema 3 (II)

Fundamentos de la programación en ensamblador

Estructura de Computadores
Grado en Ingeniería Informática



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: .string  "\n"
    valor:      .float  -3.141516

.text
main:
    flw    ft0, valor(x0)

    #se imprime
    fmv.s  fa0, ft0
    li     a7, 2
    ecall

    la     a0, saltolinea
    li     a7, 4
    ecall

    # se copia al procesador
    fmv.x.w t0, ft0
```

```
li     s0, 0x80000000    #signo
and     a0, t0, s0
srl     a0, a0, 31
li     a7, 1
ecall

la     a0, saltolinea
li     a7, 4
ecall

li     s0, 0x7F800000    #exponente
and     a0, t0, s0
srl     a0, a0, 23
li     a7, 1
ecall

la     a0, saltolinea
li     a7, 4
ecall

li     s0, 0x007FFFFF    #mantisa
and     a0, t0, s0
li     a7, 1
ecall

jr     ra
```

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

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