

ARCOS Group

uc3m | Universidad **Carlos III** de Madrid

Lesson 3 (II)

Fundamentals of assembler programming

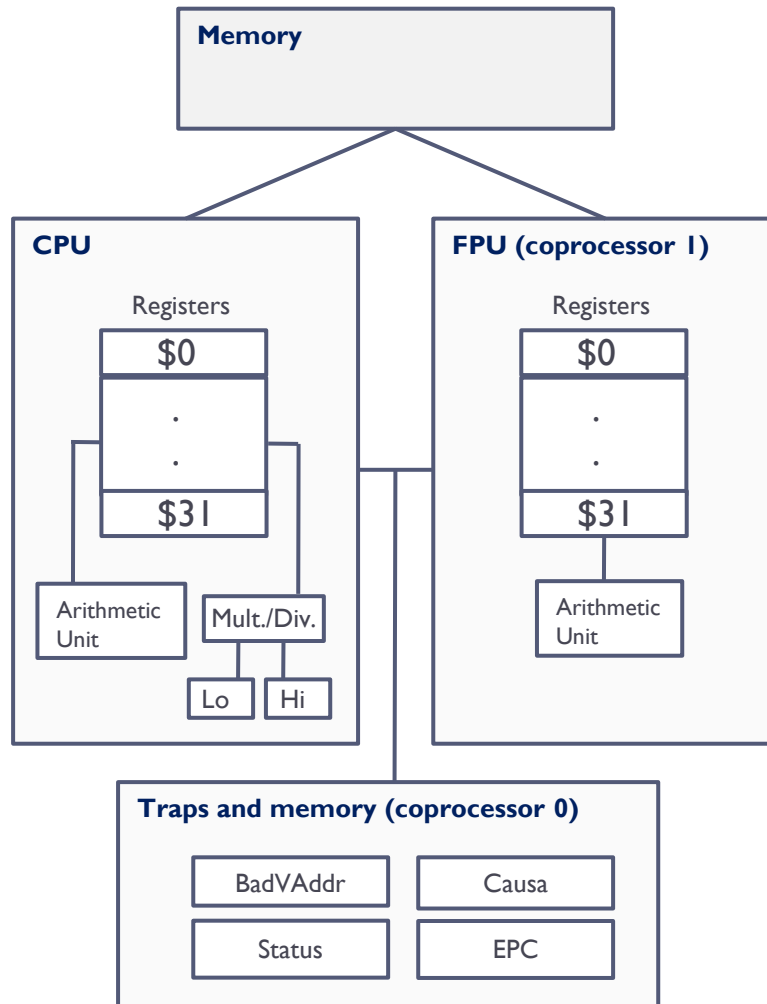
Computer Structure
Bachelor in Computer Science and Engineering



Contents

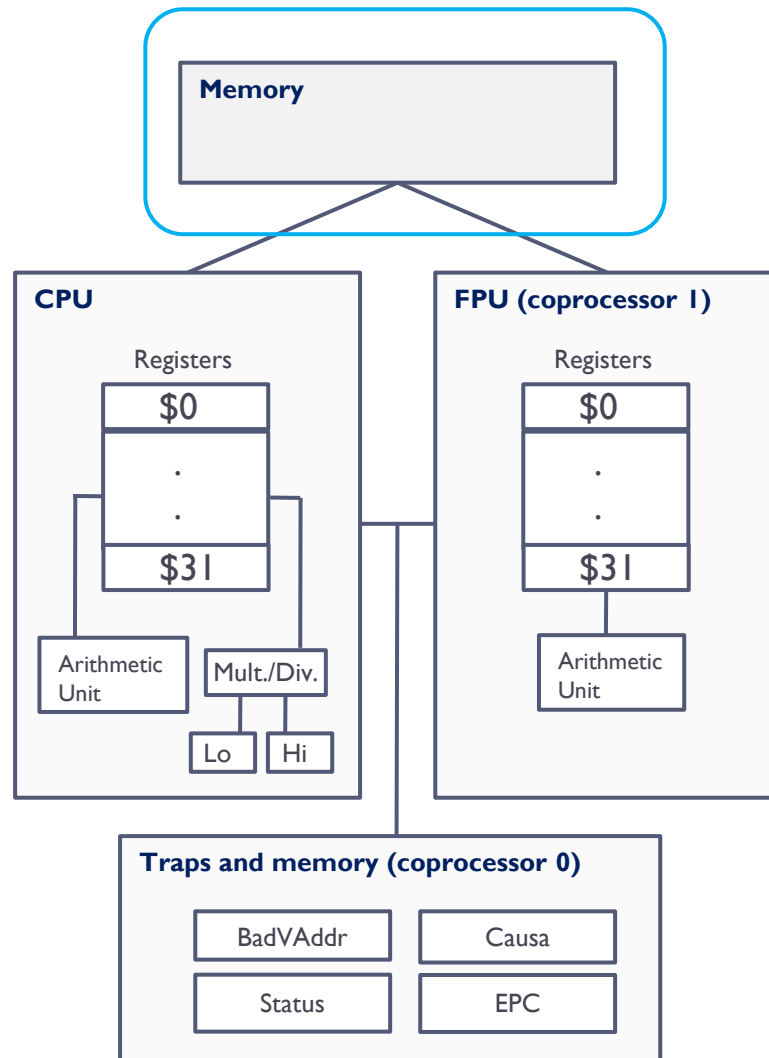
- ▶ Basic concepts on assembly programming
- ▶ MIPS32 assembly language, memory model and data representation
- ▶ Instruction formats and addressing modes
- ▶ Procedure calls and stack convention

MIPS-32 Architecture



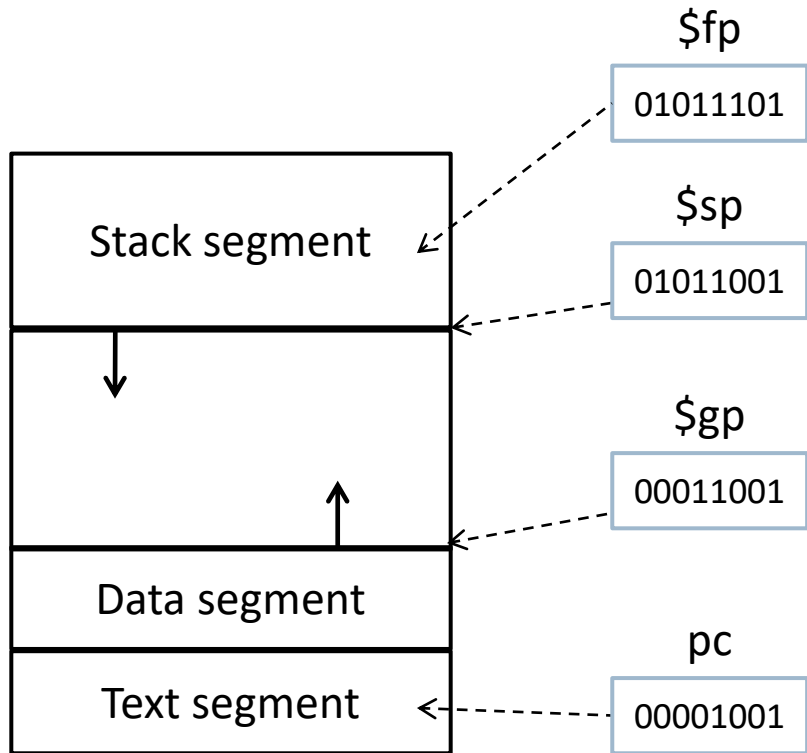
- ▶ **MIPS 32**
 - ▶ 32-bits processor
 - ▶ RISC type
 - ▶ CPU + auxiliar coprocessors
- ▶ **Coprocessor 0**
 - ▶ exceptions, interrupts and virtual memory system
- ▶ **Coprocessor 1**
 - ▶ FPU (Floating Point Unit)

MIPS-32 Architecture



- ▶ 32-bit memory addresses
- ▶ 4 GB addressable memory

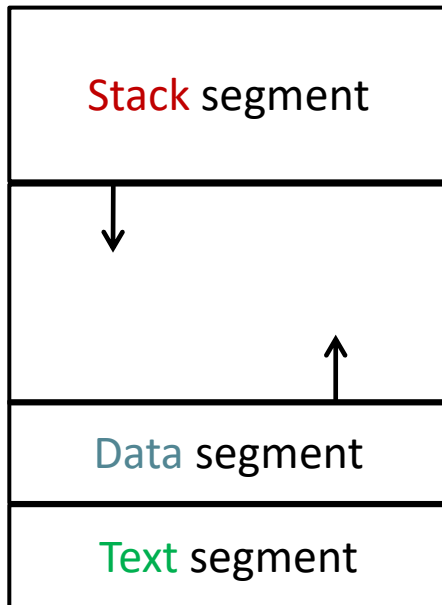
Memory layout for a process



- ▶ The memory space is divided in logic segments in order to organize the content:

- ▶ Stack segment
 - ▶ Local variables
 - ▶ Function contexts
- ▶ Data segments
 - ▶ Static data
- ▶ Code segment (text)
 - ▶ Program code

Storing variables in memory



```
// global variables
int a;

main ()
{
    // local variables
    int b;

    // code (text)
    return a + b;
}
```

Register File (integers)

Symbolic name	Number	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0, v1	2, 3	Results of functions
a0, ..., a3	4, ..., 7	Function arguments
t0, ..., t7	8, ..., 15	Temporary (NO preserved across calls)
s0, ..., s7	16, ..., 23	Saved temporary (preserved across calls)
t8, t9	24, 25	Temporary (NO preserved across calls)
k0, k1	26, 27	Reserved for operating system
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Return address (used by function calls)

- ▶ There are 32 registers
 - ▶ Size: 4 bytes (1 word)
 - ▶ Used a \$ at the beginning
- ▶ Use convention
 - ▶ Reserved
 - ▶ Arguments
 - ▶ Results
 - ▶ Temporary
 - ▶ Pointers

MIPS-32 Register File

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

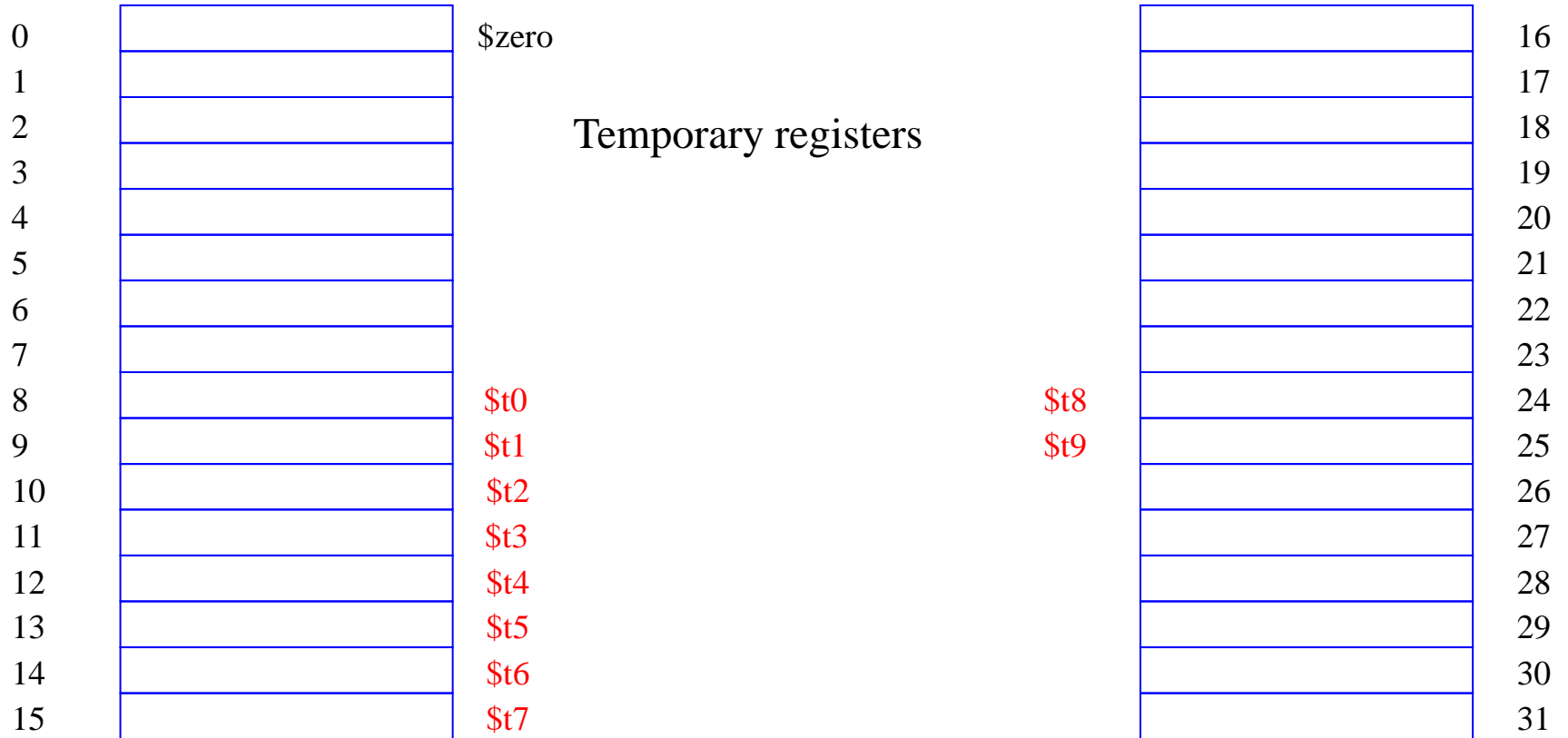
- 32 registers
 - ❑ 4 bytes of size (one word)
 - ❑ Name starts with \$ at the beginning
- Usage Convention
 - ❑ Reserved
 - ❑ Arguments
 - ❑ Results
 - ❑ Temporary
 - ❑ Pointers

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

MIPS-32 Register File

0		<div> <div>\$zero</div> <div>Constant zero</div> <div>Cannot be changed</div> </div>		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

MIPS-32 Register File



MIPS-32 Register File

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

MIPS-32 Register File

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	Arguments and subroutines support	\$sp		29
14		\$t6		\$fp		30
15		\$t7		\$ra		31

MIPS-32 Register File

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

Others

Example: hello world...

hello.s

.data

msg_hola: **.ascii** "hello world\n"

.text

.globl main

main:

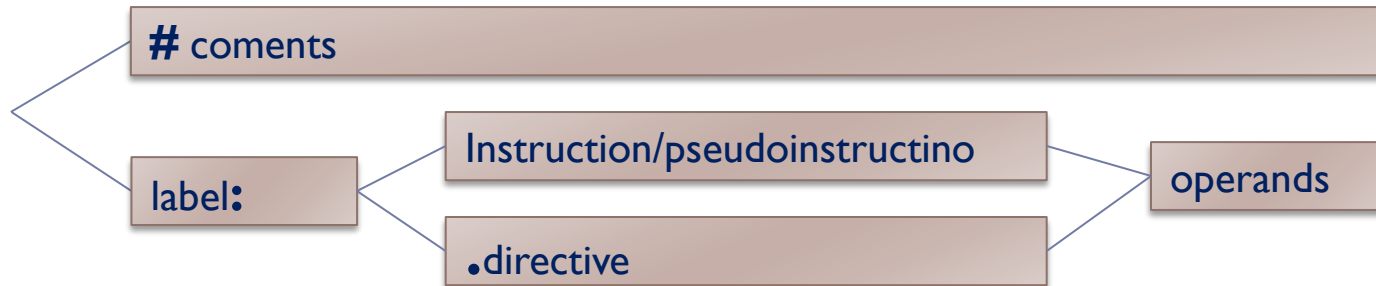
printf("hello world\n") ;

li \$v0 4 # syscall code: 4

la \$a0 msg_hola # address where msg_hola starts

syscall

Example: hello world...



hello.s

```
.data
    msg_hola: .asciiz "Hello world\n"

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

Example: hello world...

hello.s

.data

```
msg_hola: .ascii "hello world\n"
```

.text

```
.globl main
```

```
main:
```

```
    # printf("hello world\n") ;
```

```
    li $v0 4
```

```
    la $a0 msg_hola
```

```
    syscall
```


Example: hello world...

hello.s

.data

msg_hola: **.ascii** "hello world\n"

msg_hola: represents the memory address
where the string starts to be stored.

.text

.globl main

main:

printf("hello world\n") ;

li \$v0 4

la \$a0 msg_hola

syscall

code segment

Example: hello world...

hello.s

.data

```
msg_hola: .asciiz "hello world\n"
```

.text

```
.globl main
```

```
main:
```

```
# printf("hello world\n") ;
```

```
li $v0 4
```

```
la $a0 msg_hola
```

```
syscall
```

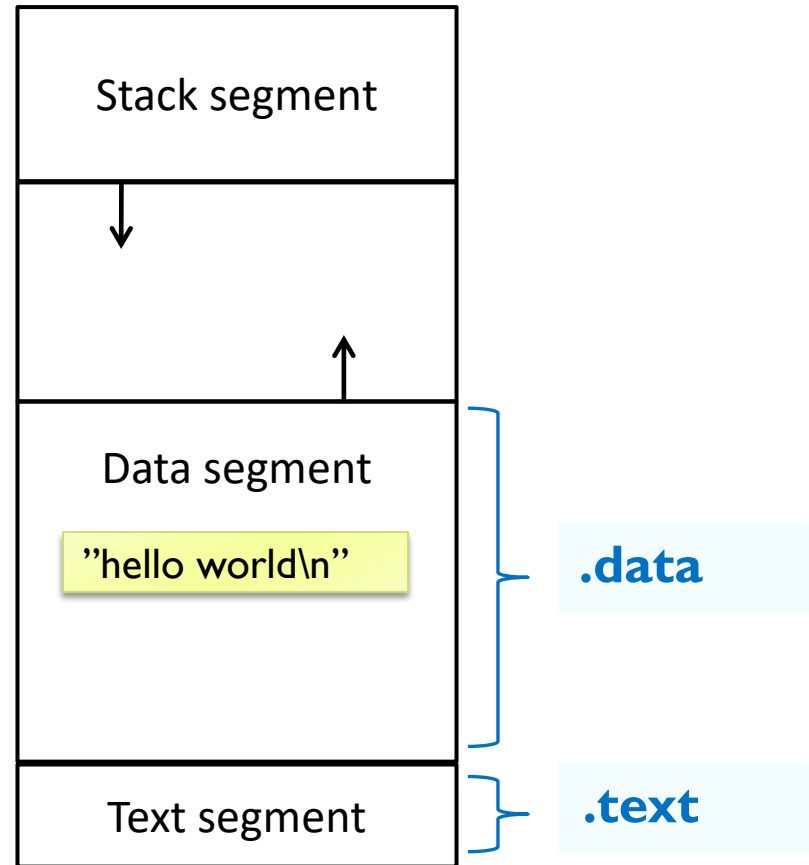
label: represent the memory address
where first instruction of main is stored

comments

instructions

Assembler program: assembler directives (preprocessing)

```
hello.s  
  
.data  
msg_hola: .asciiz "hello  
world\n"  
  
.text  
.globl main  
main:  
    # printf("hello world\n") ;  
    li $v0 4  
    la $a0 msg_hola  
    syscall
```



Assembly: directives

Directives	Description
.data	Next elements will go to the data segment
.text	Next elements will go to the code segment
.ascii "string value"	String definition <u>without</u> '\0' ending terminator
.asciiz "string value"	String definition <u>with</u> '\0' ending terminator ('\0' = 0)
.byte 1, 2, 3	Bytes stored in memory consecutively
.half 300, 301, 302	Half-words stored in memory consecutively
.word 800000, 800001	Words stored in memory consecutively
.float 1.23, 2.13	Floats stored in memory consecutively
.double 3.0e21	Doubles stored in memory consecutively
.space 10	Allocates a space of 10 bytes in the current segment
.extern <i>label n</i>	Declare that <i>label</i> is global of size <i>n</i>
.globl <i>label</i>	Declare <i>label</i> as global
.align <i>n</i>	Align next element to a address multiple of 2^n

Static data definition

label (address) datatype (directive) value

```
.data
cadena : .ascii "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
```

System calls

- ▶ Many assembler simulators include a small "operating system"
 - ▶ The SPIM simulator provides 17 services.
- ▶ How to invoke:
 - ▶ Call code in register \$v0
 - ▶ Other arguments on specific records
 - ▶ Invocation by the **syscall** instruction

System calls

Service	Call code (\$v0)	Arguments	Result
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer in \$v0
read_float	6		float in \$f0
read_double	7		double in \$f0
read_string	8	\$a0 = buffer, \$a1 = length	
sbrk	9	\$a0 = amount	address in \$v0
exit	10		

System calls

Service	Call code (\$v0)	Arguments	Result
print_char	11	\$a0 (ASCII code)	
read_char	12		\$v0 (ASCII code)
open	13	Equivalent to \$v0=open(\$a0, \$a1, \$a2)	file descriptor in \$v0
read	14	Equivalent to \$v0=read(\$a0, \$a1, \$a2)	read bytes in \$v0
write	15	Equivalent to \$v0=write(\$a0, \$a1, \$a2)	written bytes in \$v0
close	16	Equivalent to \$v0=close(\$a0)	0 in \$v0
exit2	17	End the program. Return the value stored in \$a0	

Example: Hello world...

hello.s

.data

msg_hola: **.ascii** "hello world\n"

.text

.globl main

main:

printf("hello world\n") ;

li \$v0 4 # syscall code: 4

la \$a0 msg_hola # address where msg_hola starts

syscall **Operating system invocation instruction**

Service	Call code (\$v0)	Arguments
print_int	1	\$a0 = integer
print_float	2	\$f12 = float
print_double	3	\$f12 = double
print_string	4	\$a0 = string

Exercise

. . .

```
int valor ;
```

. . .

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

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

```
printInt(valor) ;
```

. . .

service	code	arguments	results
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer in \$v0
read_float	6		float in \$f0
read_double	7		double in \$f0
read_string	8	\$a0=buffer, \$a1=long.	
sbrk	9	\$a0=amount	address in \$v0
exit	10		

Exercise (solution)

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

service	code	arguments	results
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 $a0 valor  
  
# printInt  
li $v0 1  
syscall
```

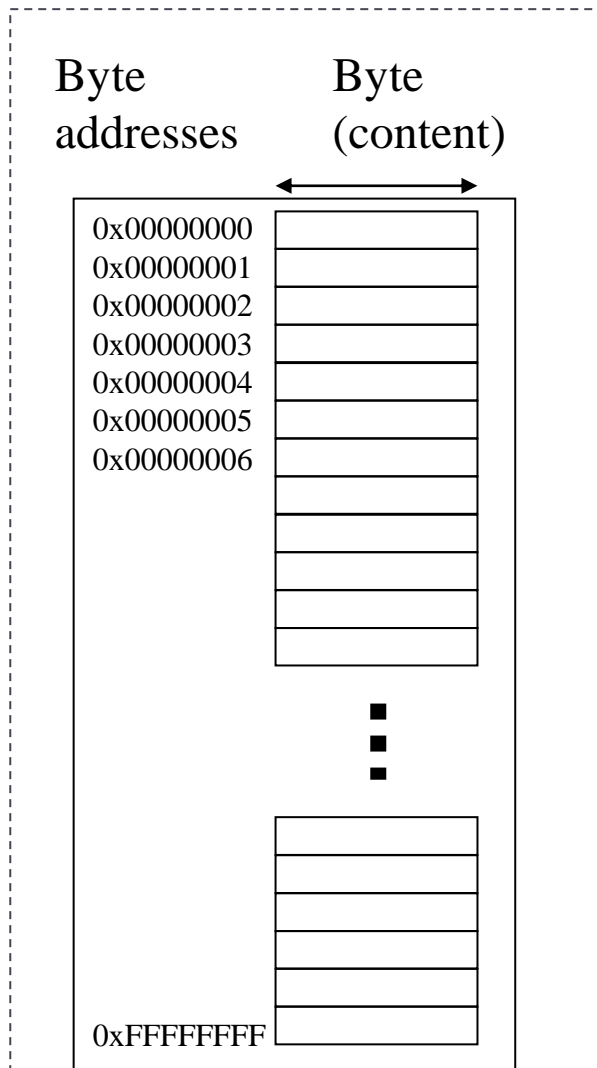
Instructions and pseudo-instructions

- ▶ There is an assembly instruction per machine instruction :
 - ▶ Each machine instruction occupies 32 bits in MIPS32
 - ▶ `addi $t1, $t0, 4`
- ▶ A pseudo-instruction can be used in an assembler program and it corresponds to one or several assembly instructions:
 - ▶ E.g: `li $v0, 4`
`move $t1, $t0`
- ▶ In the assembly process, they are replaced by the sequence of assembly instructions that perform the same functionality.
 - ▶ E.g.: `ori $v0, $0, 4` replaces to: `li $v0, 4`
`addu $t1, $0, $t2` replaces to: `move $t1, $t2`

Other examples of pseudo-instructions

- ▶ An assembler pseudoinstruction can correspond to several machine instructions.
 - ▶ `li $t1, 0x00800010`
 - It does not fit in 32 bits, but can be used as a pseudo-instruction.
 - It is equivalent to:
`lui $t1, 0x0080`
`ori $t1, $t1, 0x0010`

MIPS32 memory model



Memory is addressed by bytes:

- 32-bit addresses
- Content of each address: one byte
- Addressable space: 2^{32} bytes = 4 GB

Access can be to:

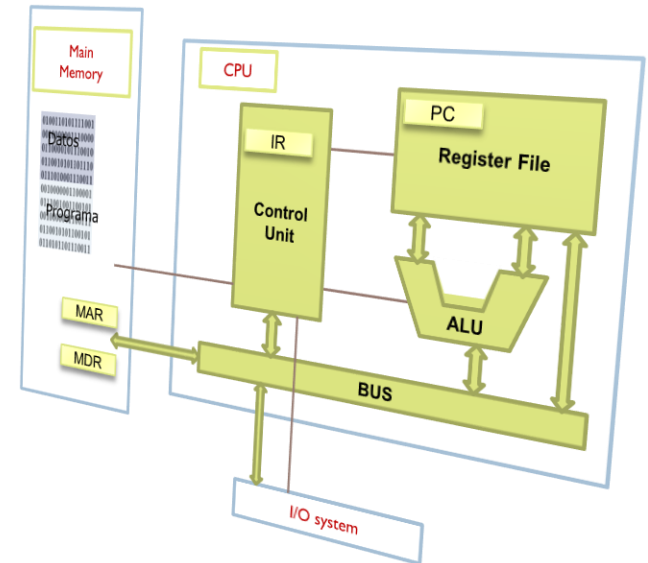
- Individual bytes
- Words (4 consecutive bytes)

Format of memory access instructions

lw
sw
lb
sb
lbu

Register, memory address

- **Number** representing an address
- **Symbolic label** representing an address
- **(register)**: represents the address stored in the register
- **num(register)**: represents the address obtained by summing num with the address stored in the register



Memory access instruction formats

- ▶ `lbu $t0, 0x0F000002`
 - ▶ Direct addressing. The byte stored at memory location `0x0F000002` is loaded into `$t0`.
- ▶ `lbu $t0, labeled`
 - ▶ Direct addressing. The byte stored in the memory location `labeled` is loaded into `$t0`.
- ▶ `lbu $t0, ($t1)`
 - ▶ Indirect register addressing. The byte stored in the memory location stored in `$t1` is loaded in `$t0`.
- ▶ `lbu $t0, 80($t1)`
 - ▶ Relative addressing. The byte stored in the memory location obtained by adding the contents of `$t1` with 80 is loaded in `$t0`.

Data transfer bytes

- Copies a **byte** from **memory** to a **register** or vice versa.

- Examples:

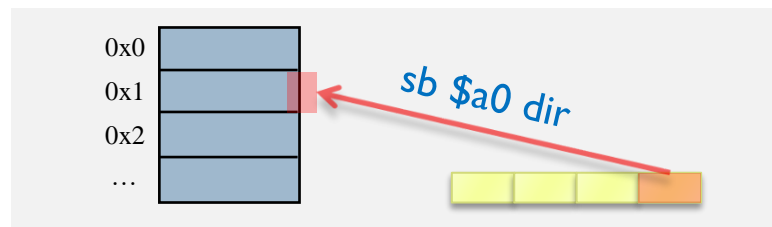
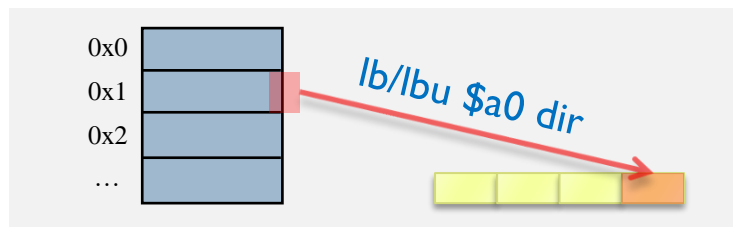
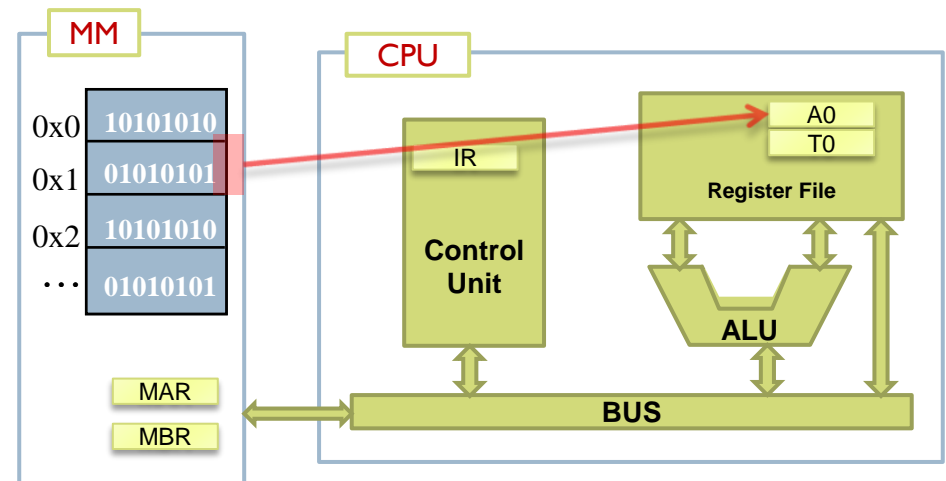
- Memory to register

lb \$a0, dir

lbu \$a0, dir

- Register to memory

sb \$t0, dir

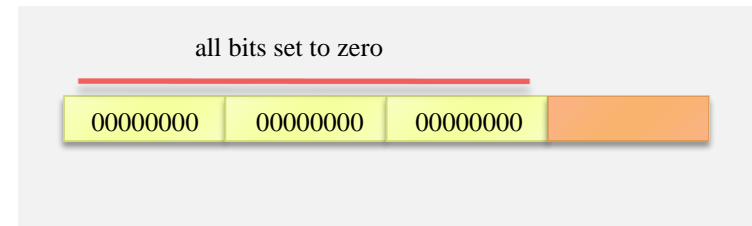
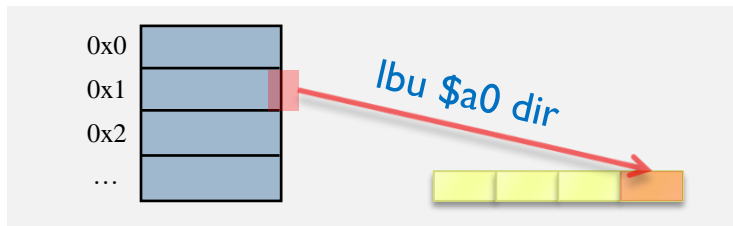


Data transfer

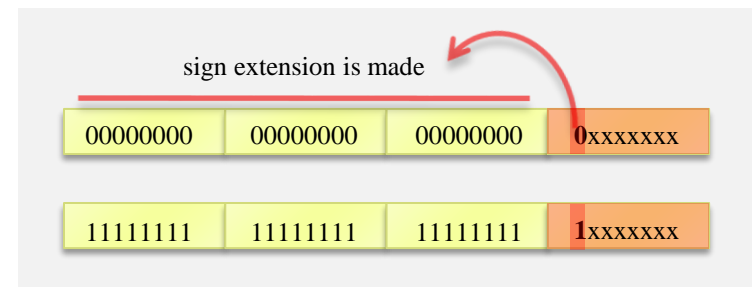
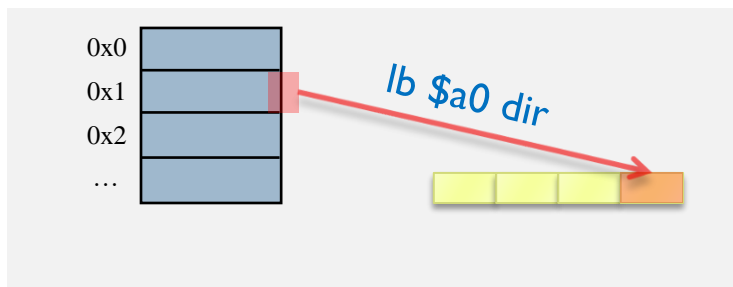
Sign extension

- ▶ There are two possibilities when transferring a byte from memory to register:

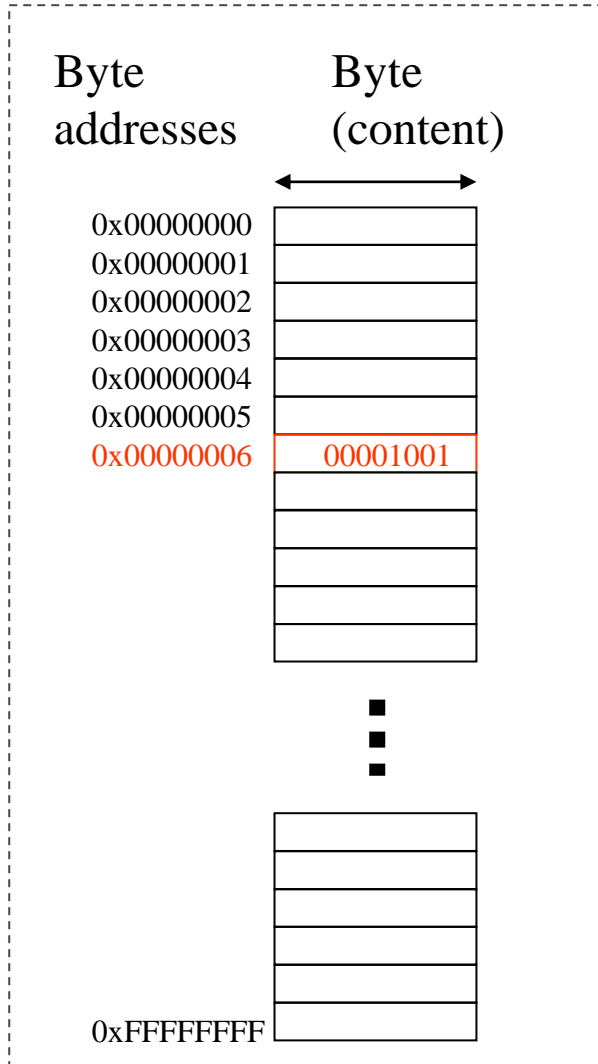
- ▶ A) Transfer **without sign**, for example: `lbu $a0, dir`



- ▶ B) Transfer **with sign**, for example: `lb $a0, dir`



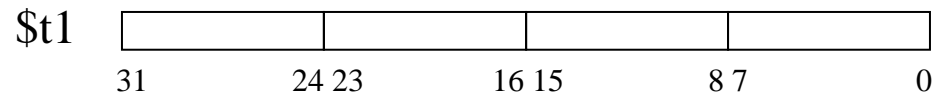
Access to bytes with lb (load byte)



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

Address: 0x000000006 (000110)

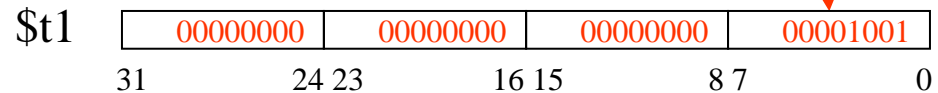
Content: 00001001 (9)



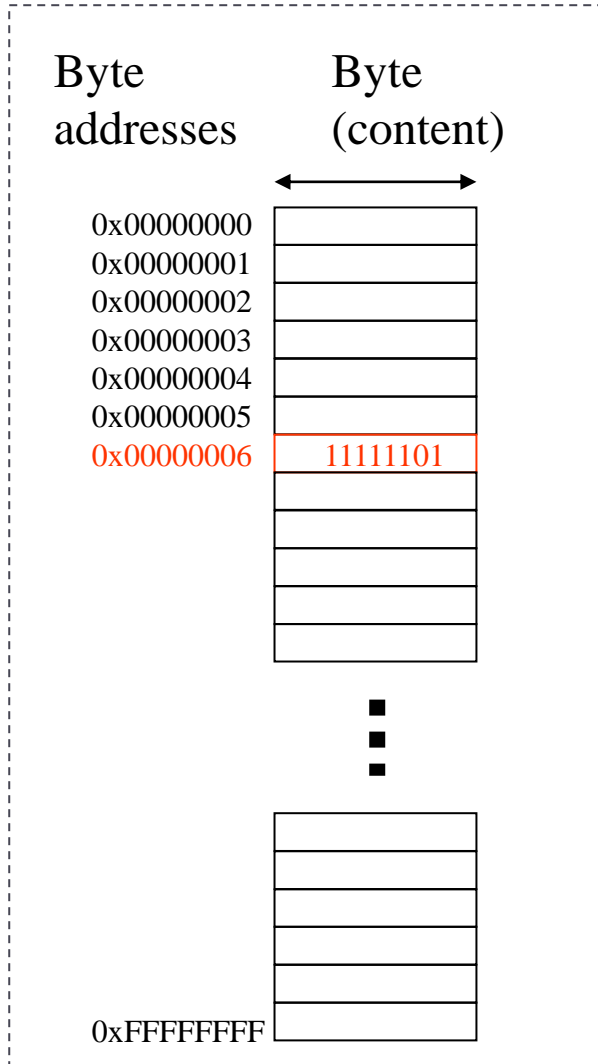
Access to bytes with lb

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

Address: 0x00000006 (000110)
Content: 00001001 (9)



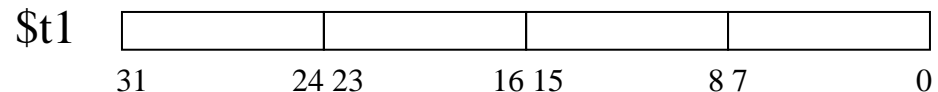
Access to bytes with lb



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

Address: 0x000000006 (000110)

Content: 1111101 (-3 in two's complement)

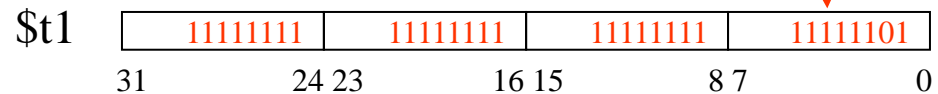
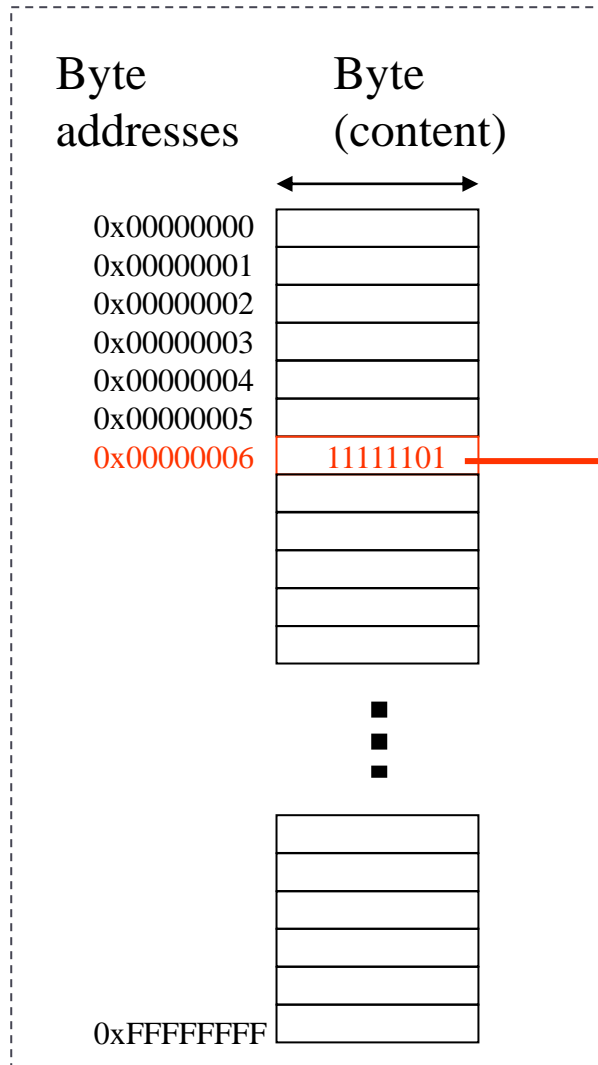


Access to bytes with lb

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

Address: 0x00000006 (000110)

Content: 11111101 (-3 in two's complement)

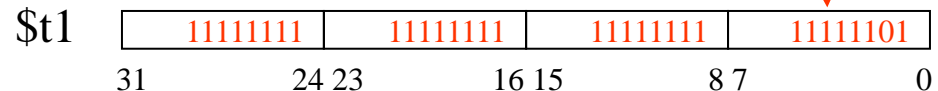


Access to bytes with lb

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

Address: 0x00000006 (000110)

Content: 11111101 (-3 in two's complement)



The “lb” instruction keep the sign
(sign extension)

Access to bytes with lb

problems accessing characters

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

Address: 0x000000006 (000110)

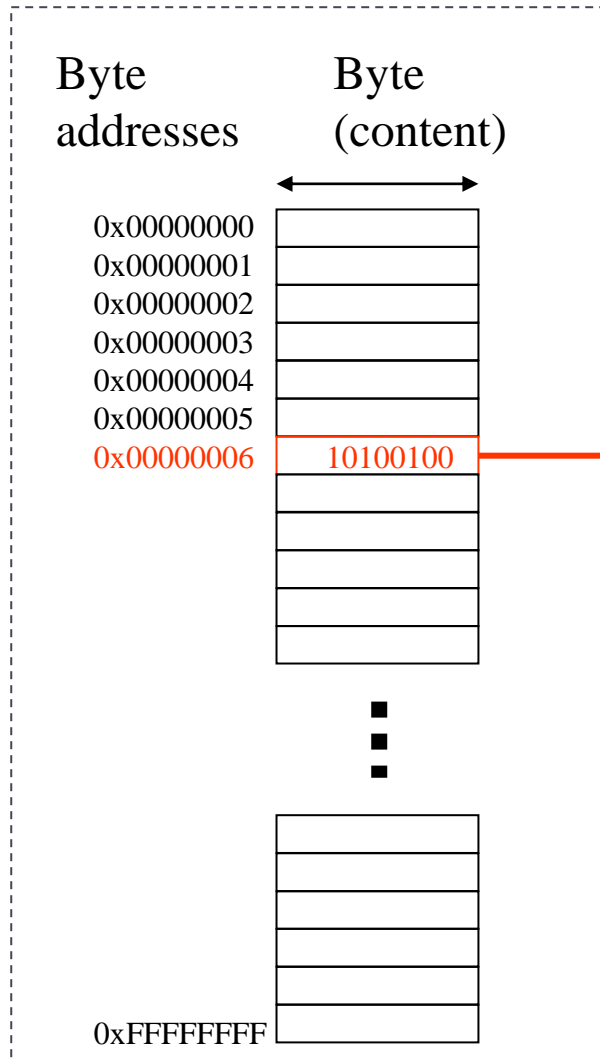
Content: 10100100 (ASCII code for letter ñ is 164)

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

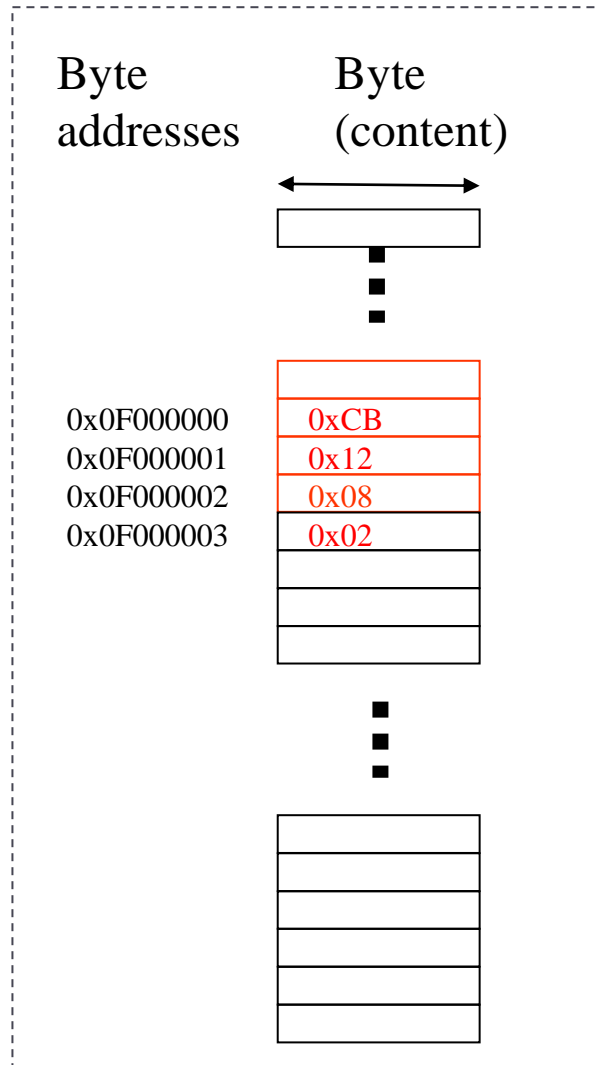
\$t1

11111111	11111111	11111111	10100100
31	24 23	16 15	8 7 0

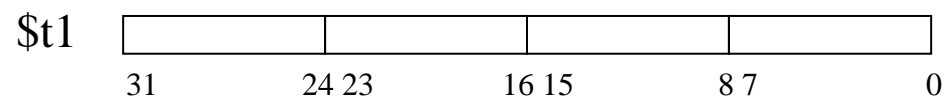
If lb is used (keeps the sign) and the content of \$t1 is not the 164 value (ASCII code for ñ)



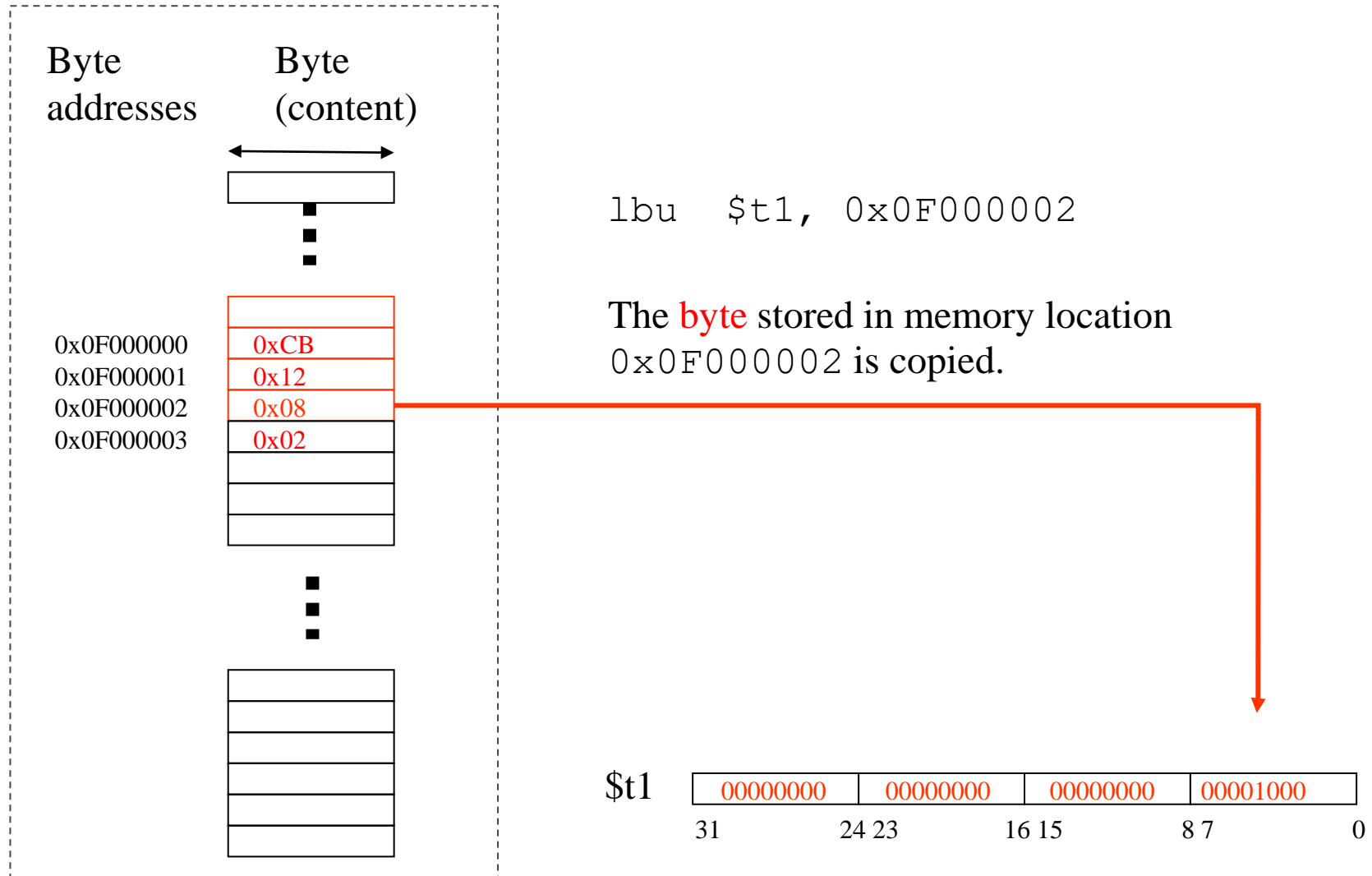
Access to bytes with lbu (load byte unsigned)



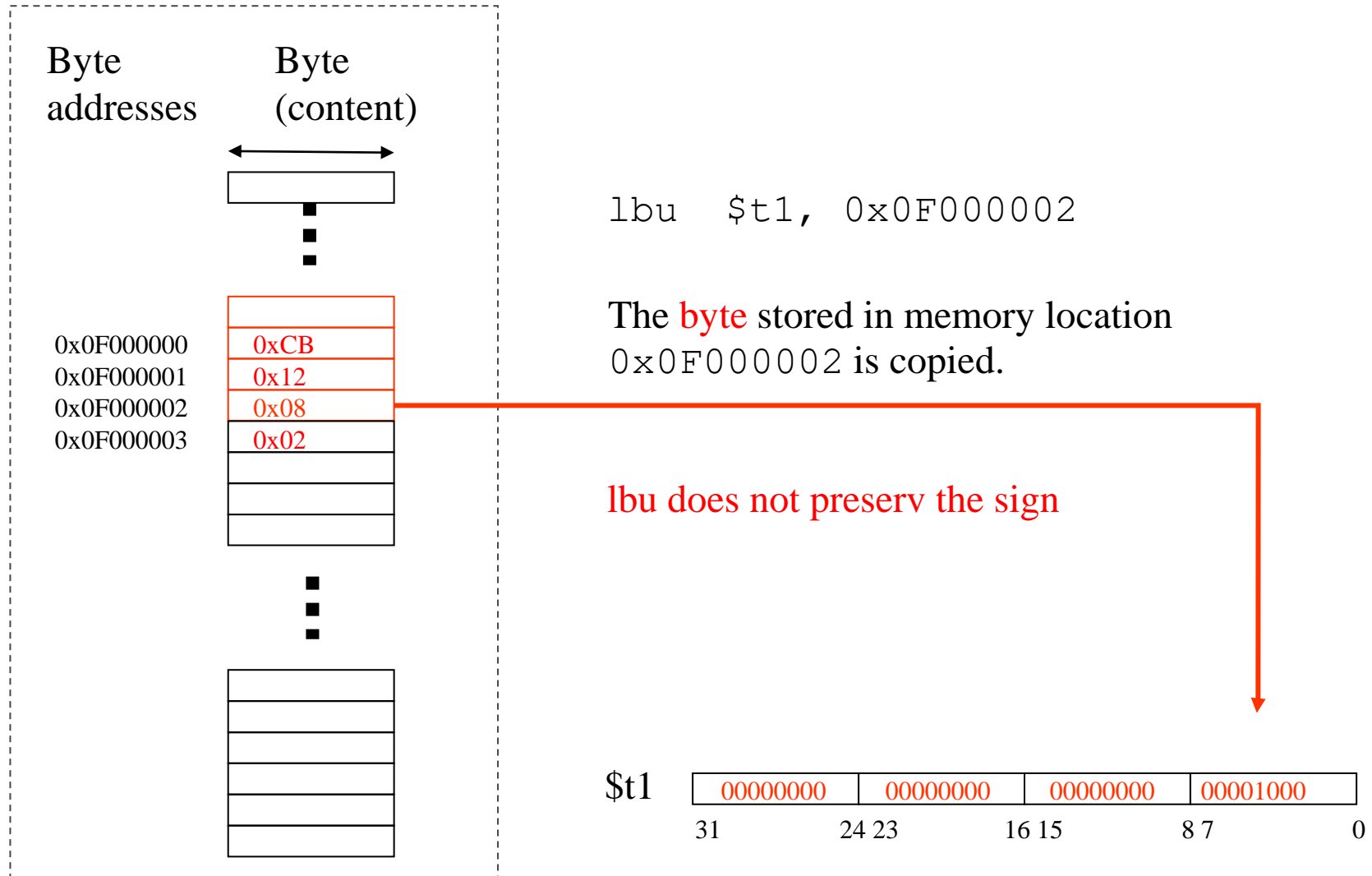
```
lbu    $t1, 0x0F000002
```



Access to bytes with lbu (load byte unsigned)

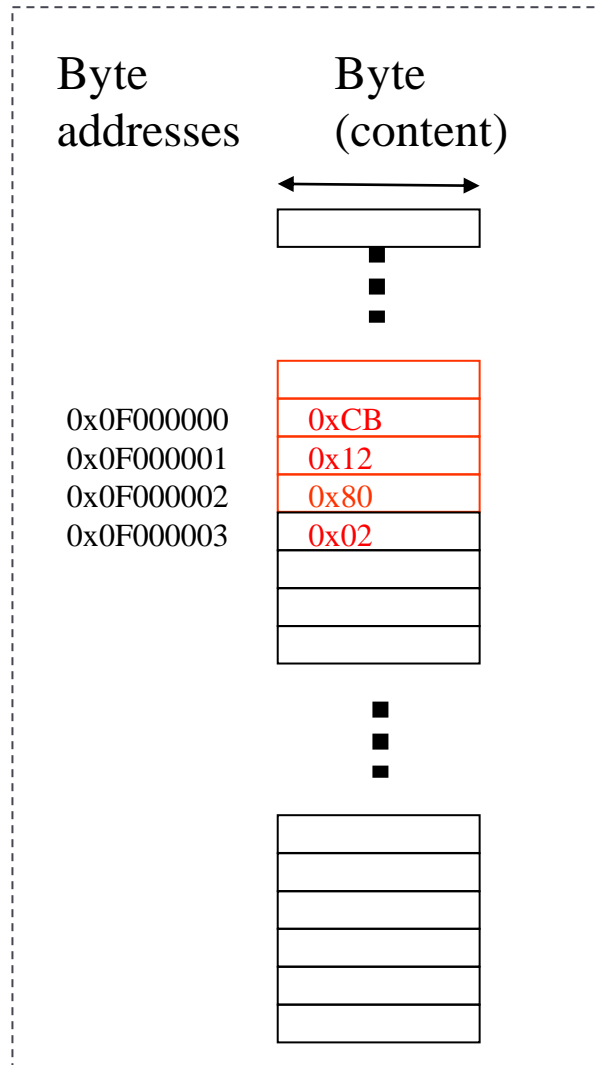


Access to bytes with lbu (load byte unsigned)

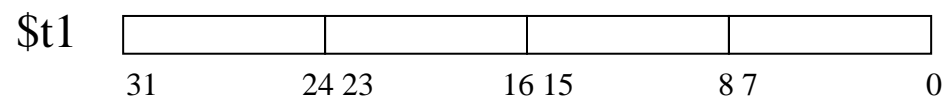


Access to bytes with lbu (load byte unsigned)

No sign-extension

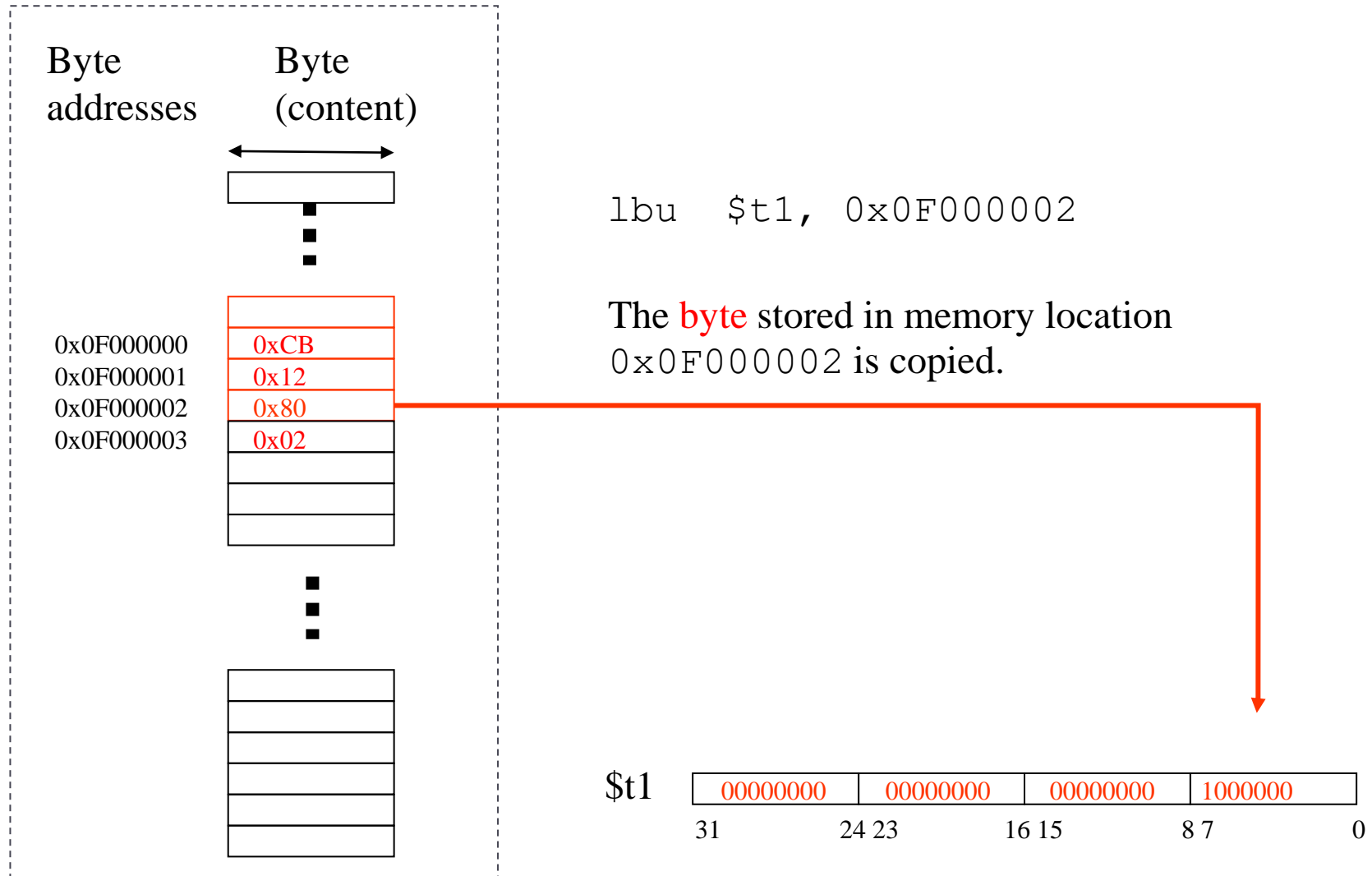


```
lbu    $t1, 0x0F000002
```



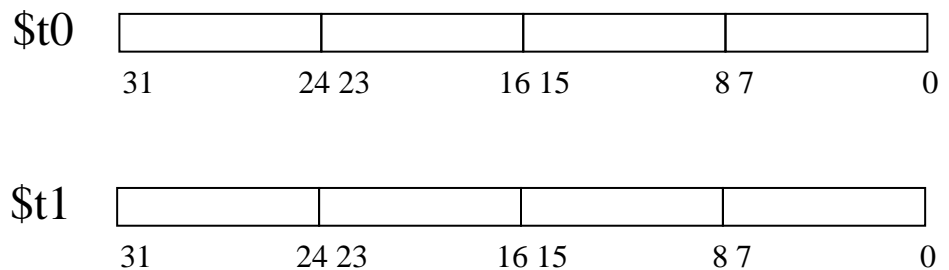
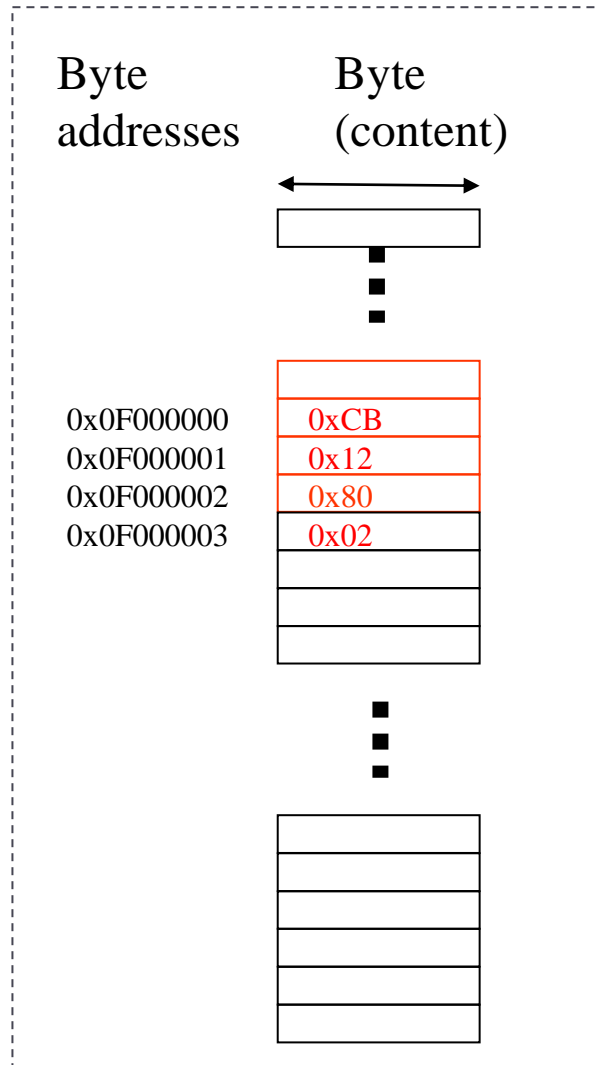
Access to bytes with lbu (load byte unsigned)

No sign-extension



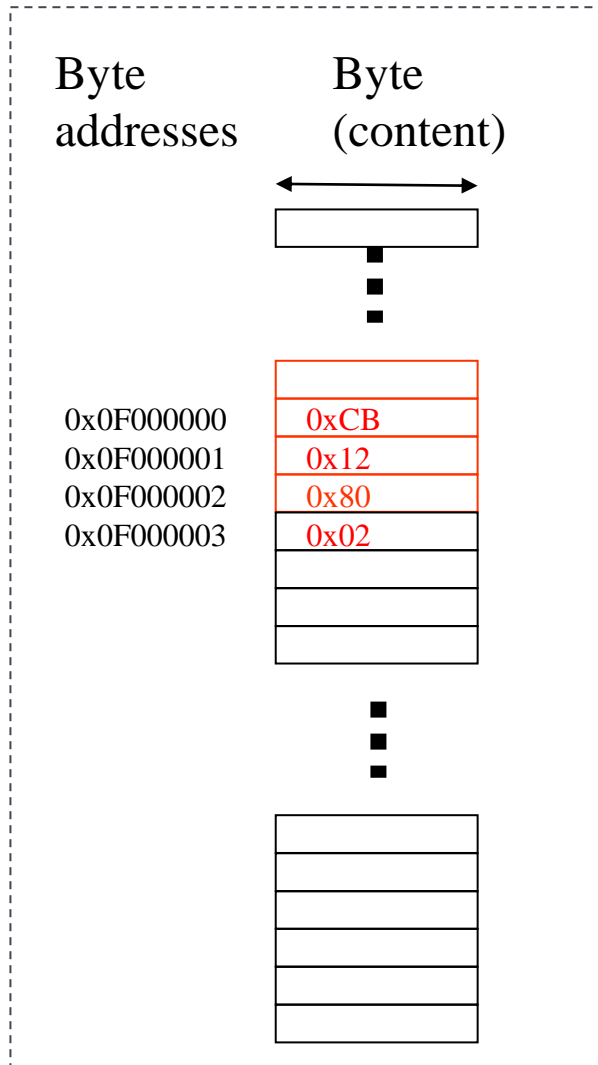
Example of la and lbu

(load address and load byte unsigned)



Example of la and lbu

(load address and load byte unsigned)



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

\$t0

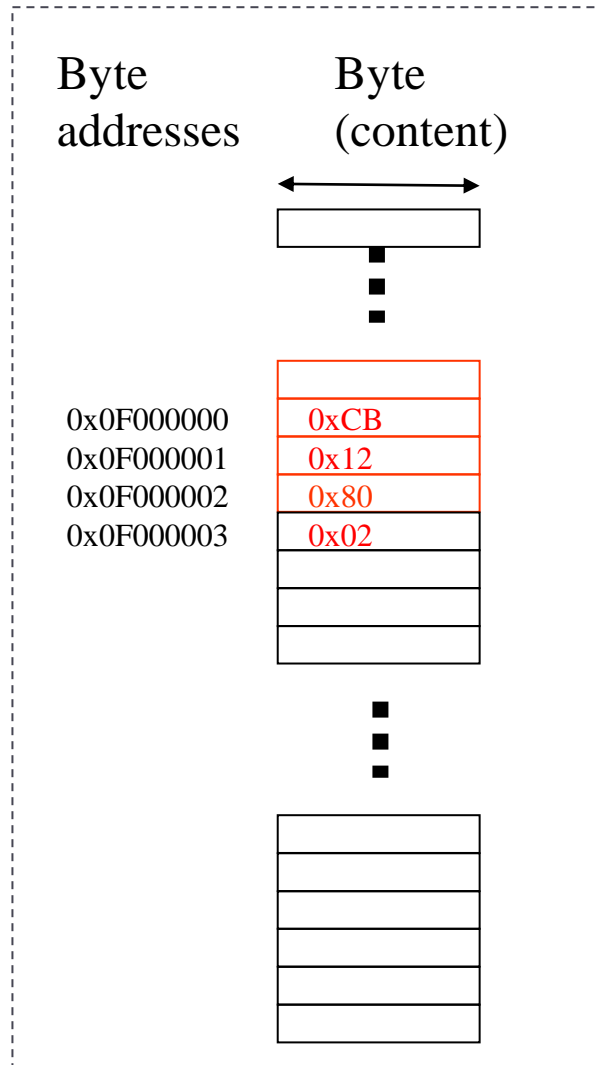
31	24	23	16	15	8	7	0
----	----	----	----	----	---	---	---

\$t1

31	24	23	16	15	8	7	0
----	----	----	----	----	---	---	---

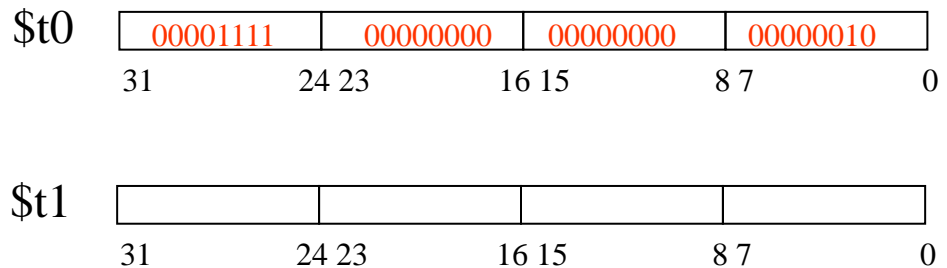
Example of la and lbu

(load address and load byte unsigned)



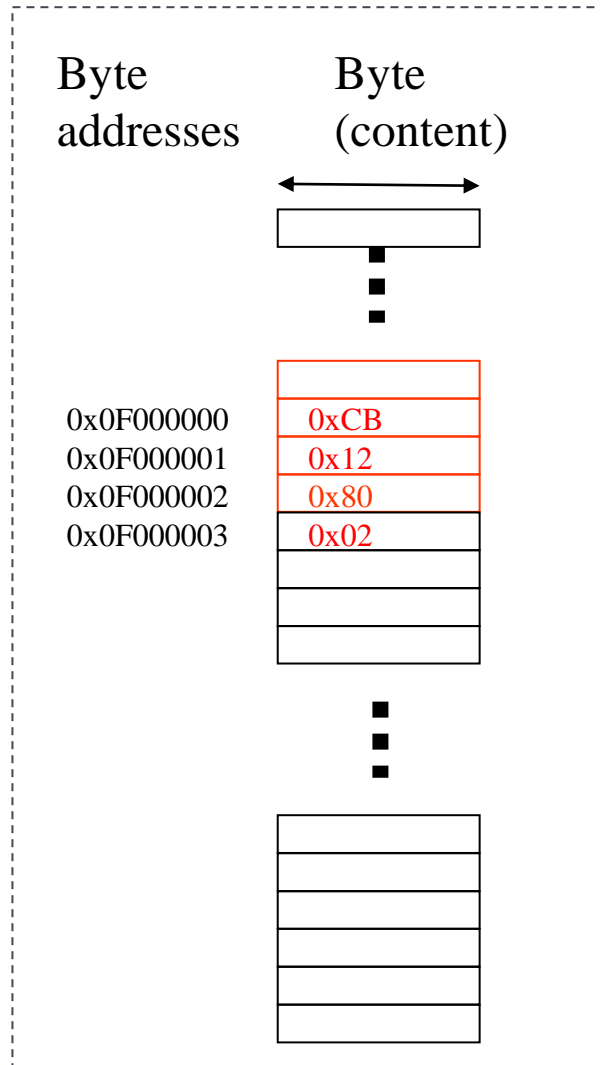
```
la $t0, 0x0F000002
```

The address is copied,
not the content

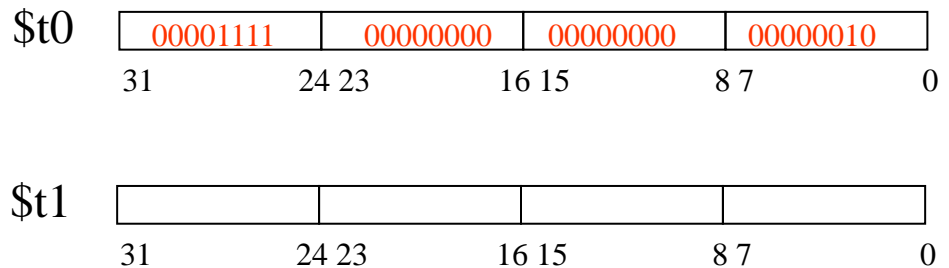


Example of la and lbu

(load address and load byte unsigned)

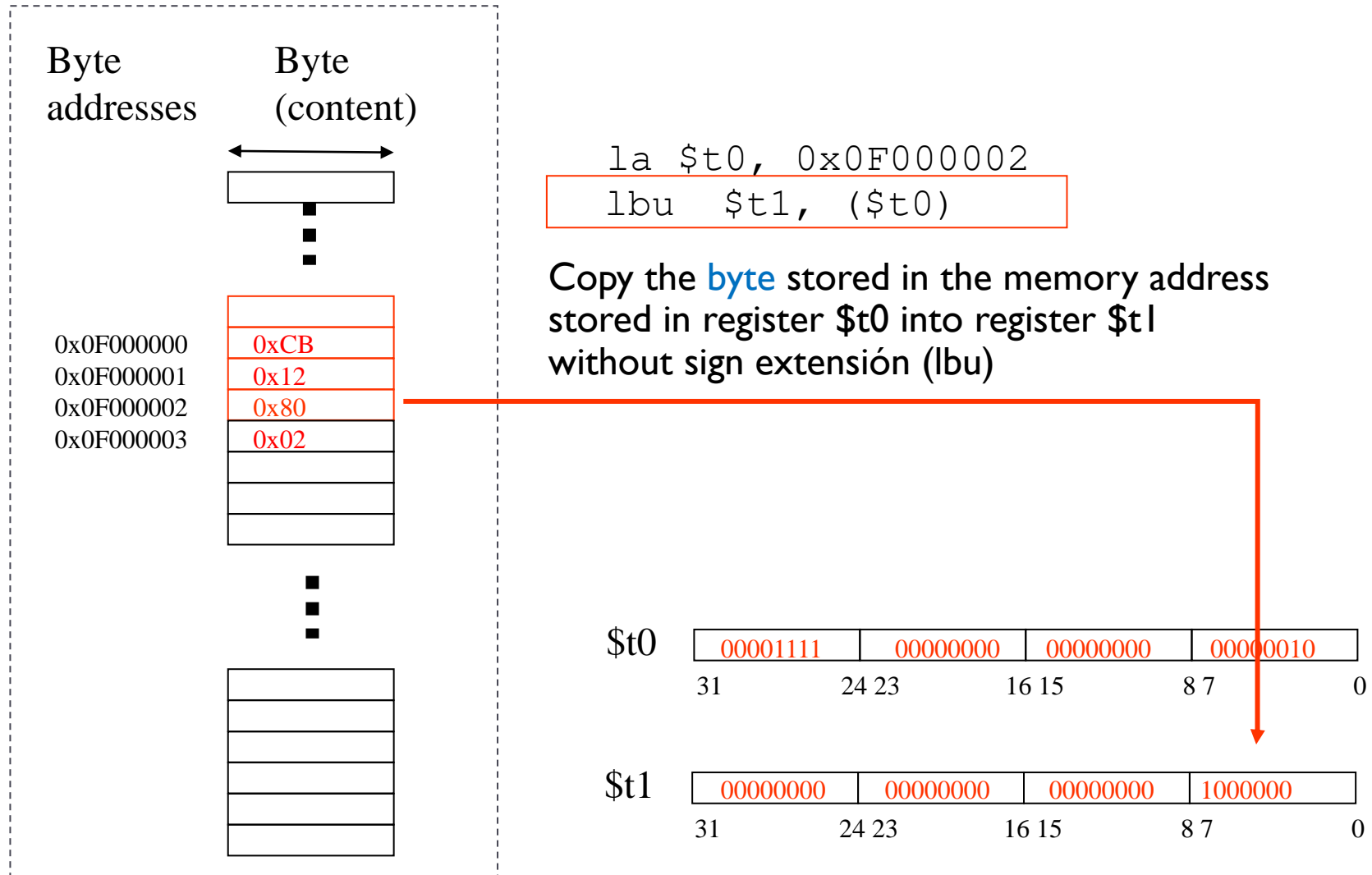


```
la $t0, 0x0F000002  
lbu $t1, ($t0)
```



Example of la and lbu

(load address and load byte unsigned)



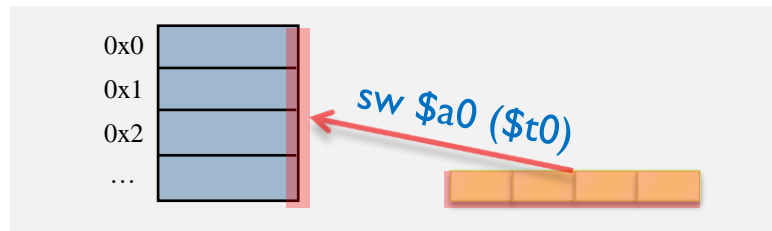
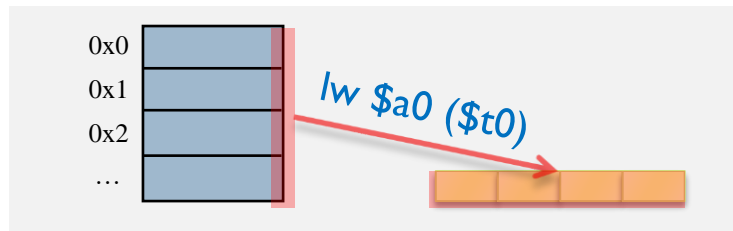
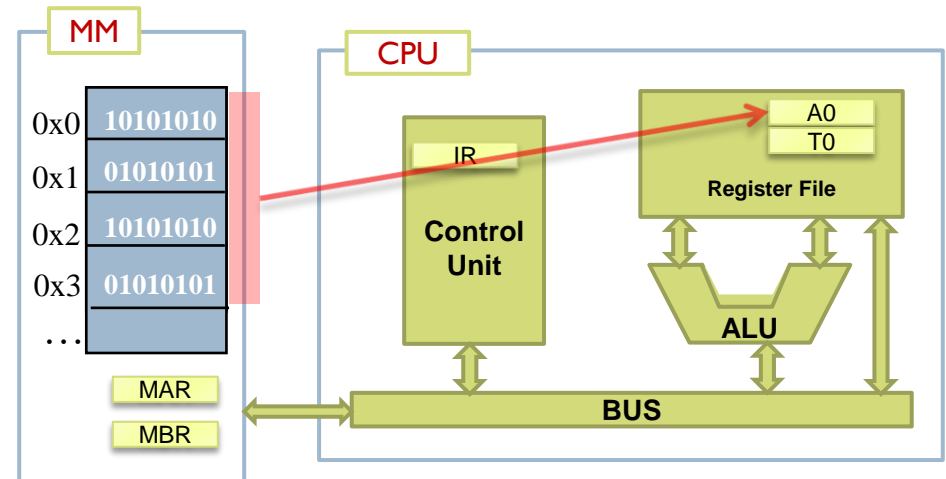
Data transfer words

- Copies a **word** from **memory** to a **register** or vice versa.

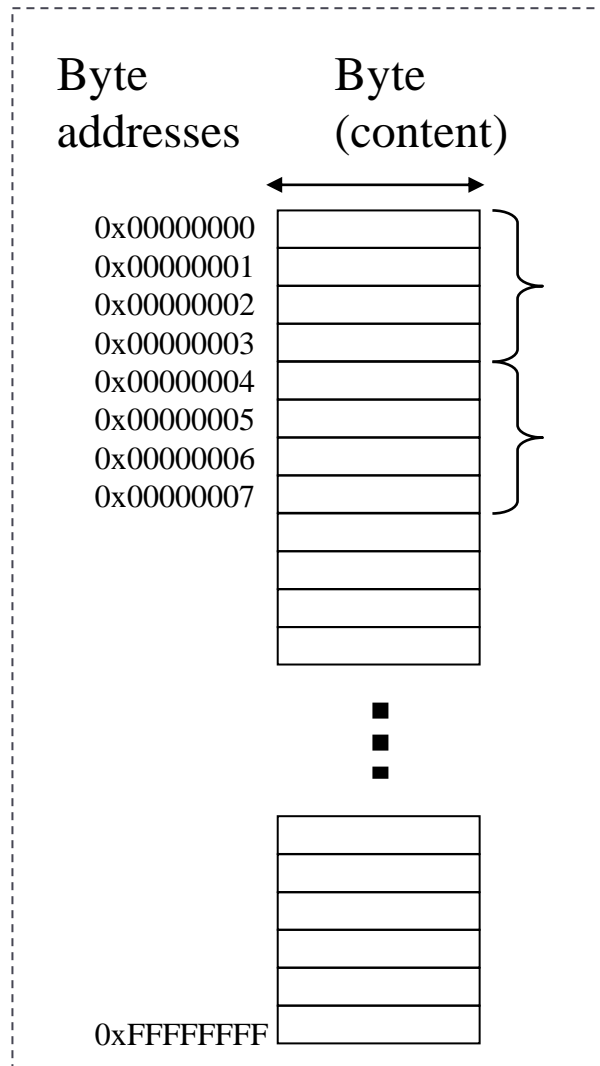
- Examples:

- Memory to register
`lw $a0 ($t0)`

- Register to memory
`sw $a0 ($t0)`



Accessing to words



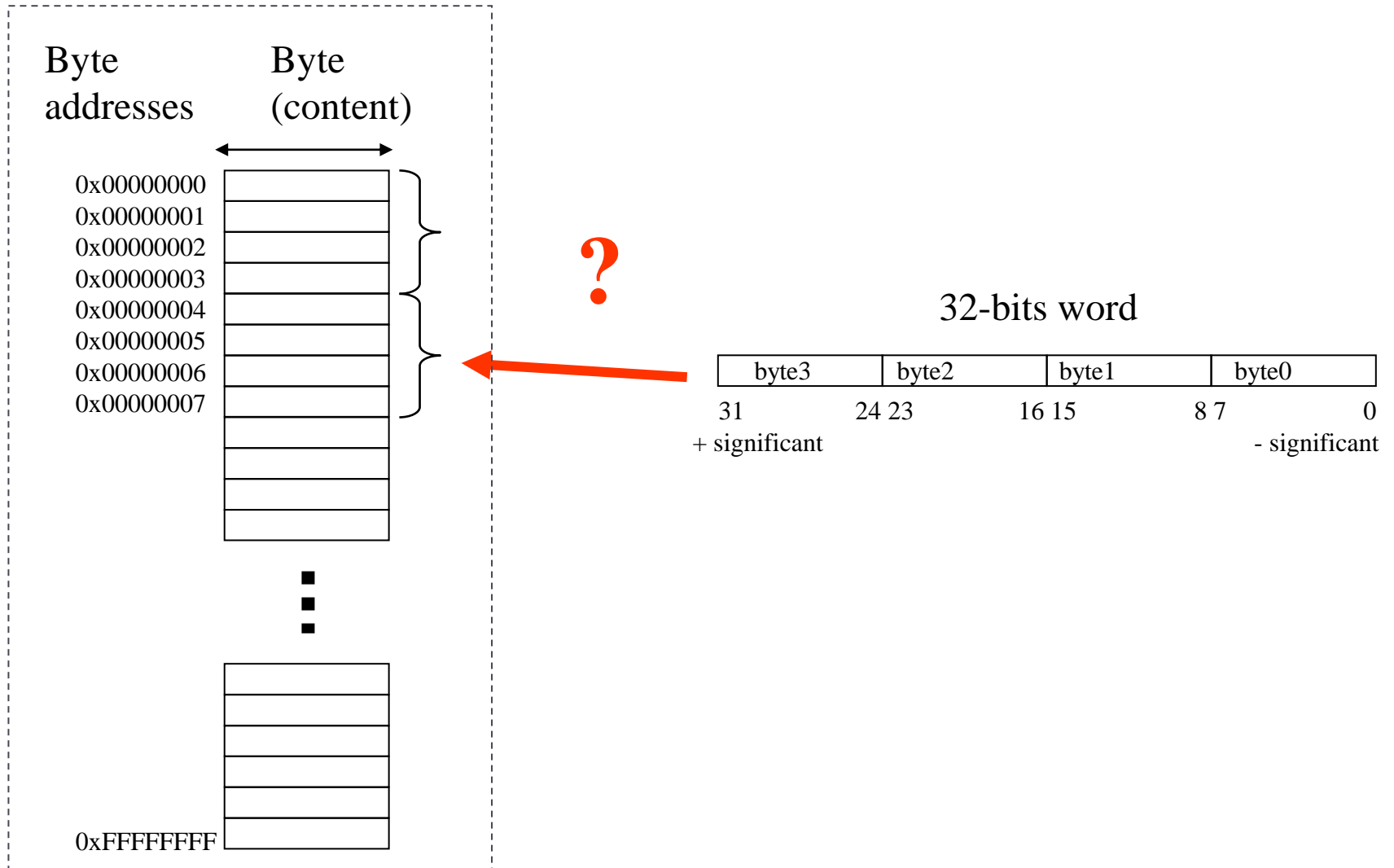
A word: 4 bytes in a 32-bits processor

Word stored starting at byte 0

Word stored starting at 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

Accessing to words

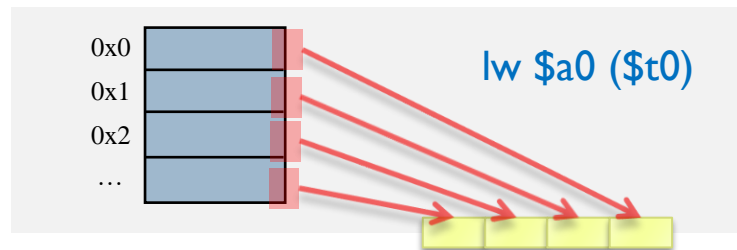


Data transfer

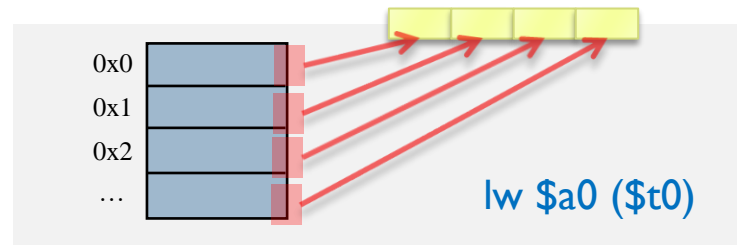
byte order

- ▶ There are 2 types of byte order:

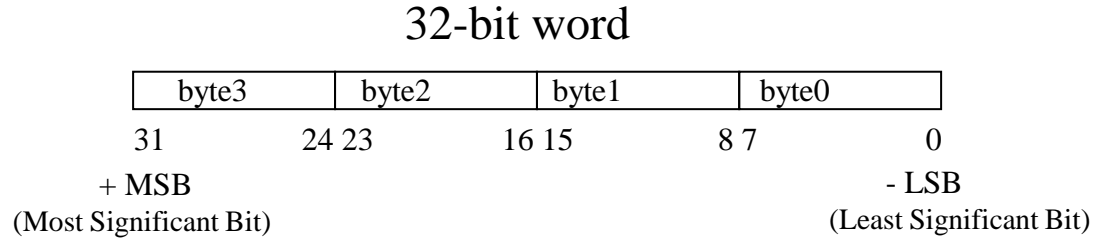
- ▶ Little-endian (‘small’ address ends the word...)



- ▶ Big-endian (‘big’ address ends the word...)



Storing words in memory



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

BigEndian

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

LittleEndian

The number $27_{(10)} = 11011_{(2)} = 00000000000000000000000000011011$

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

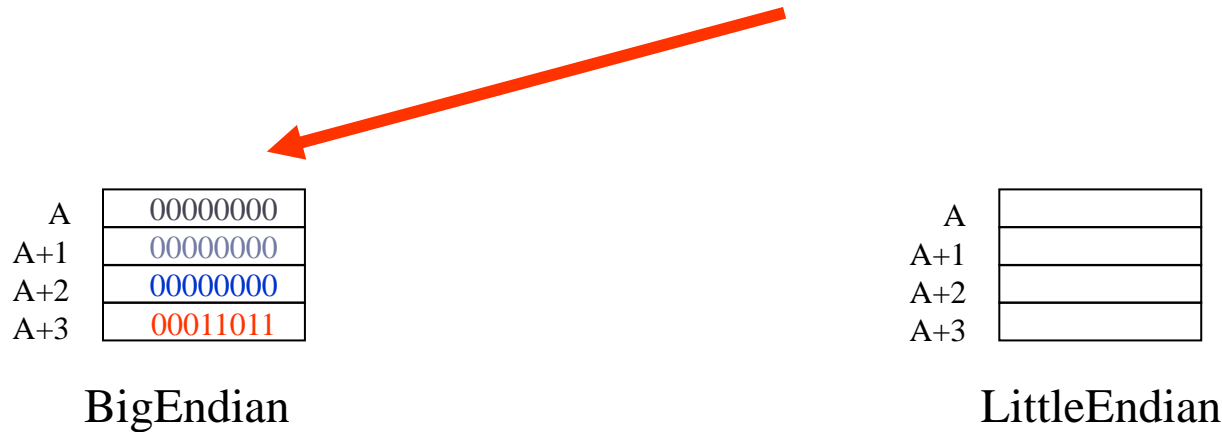
BigEndian

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

LittleEndian

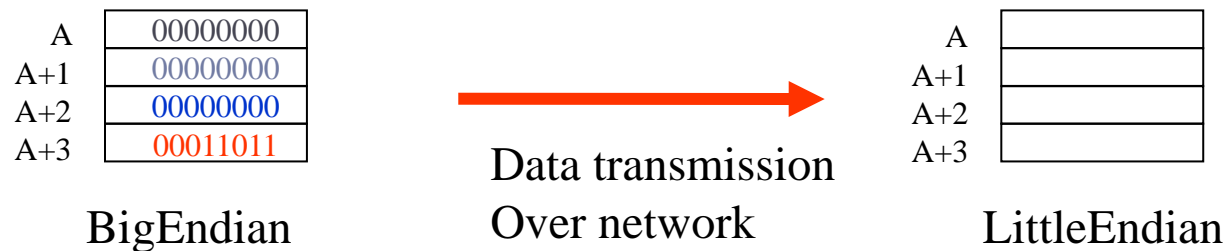
Communication problems in computers with different architectures

The number $27_{(10)} = 11011_{(2)} = 00000000000000000000000000011011$



Communication problems in computers with different architectures

The number $27_{(10)} = 11011_{(2)} = 00000000000000000000000000011011$



Communication problems in computers with different architectures

The number $27_{(10)} = 11011_{(2)} = 00000000000000000000000000011011$

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

BigEndian

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

LittleEndian



The stored number is: **00011011**00000000000000000000000000000000
And is not 27!

Example

endian.s

```
.data
```

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

```
.text
```

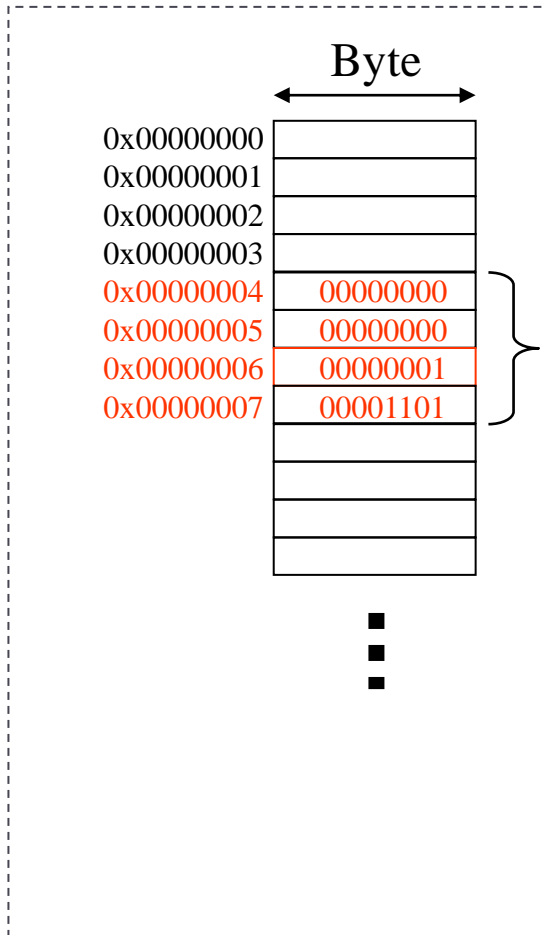
```
.globl main
```

```
main:
```

```
    lw  $t0 b1
```

Read word from memory

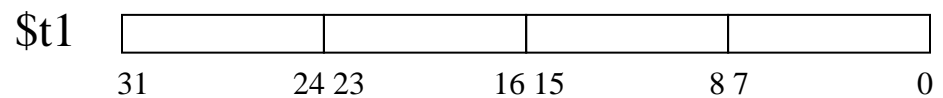
lw \$t1, 0x4



Address: 0x00000004 (000000.....00100)

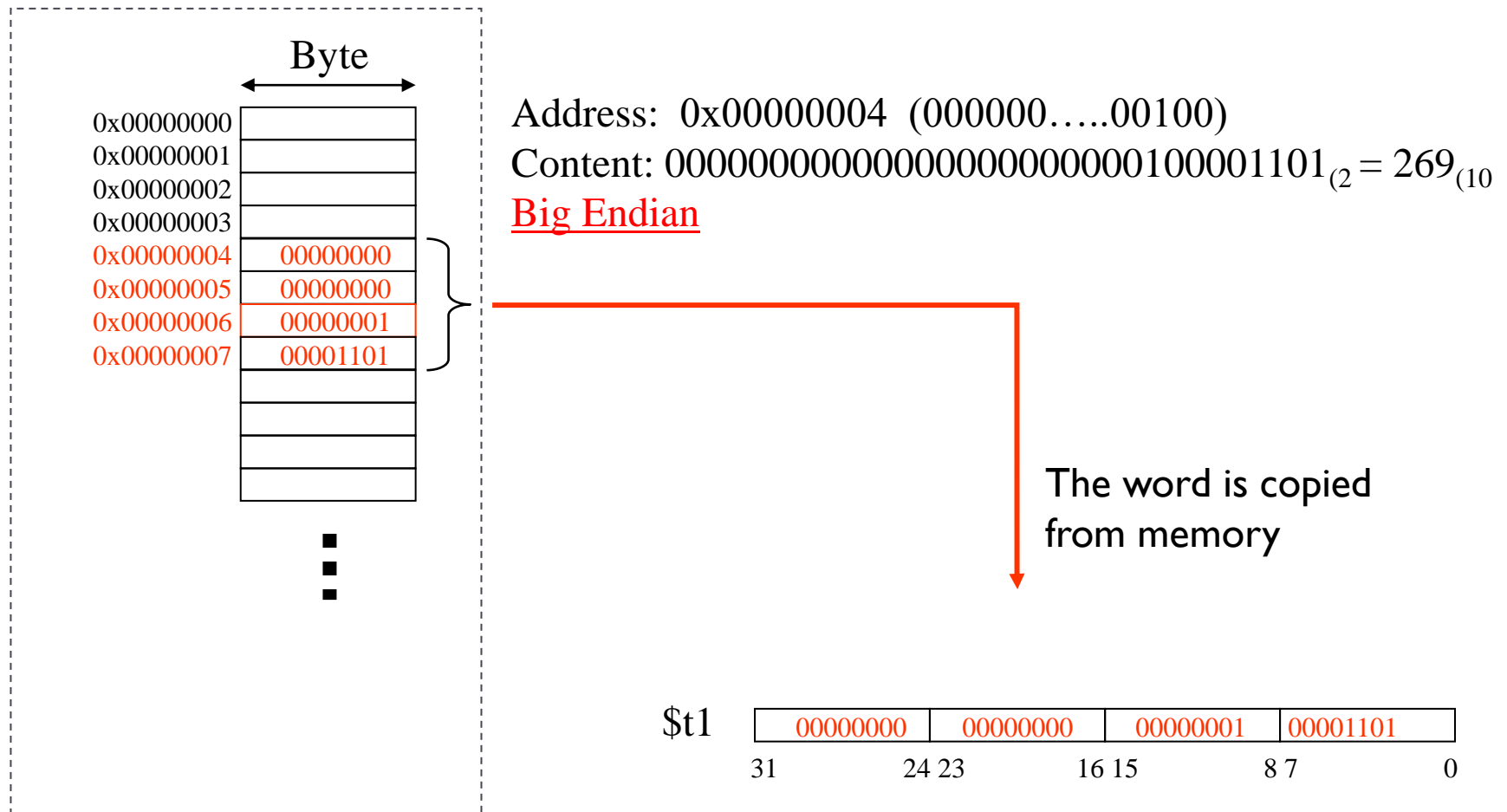
[illegible]

Big Endian



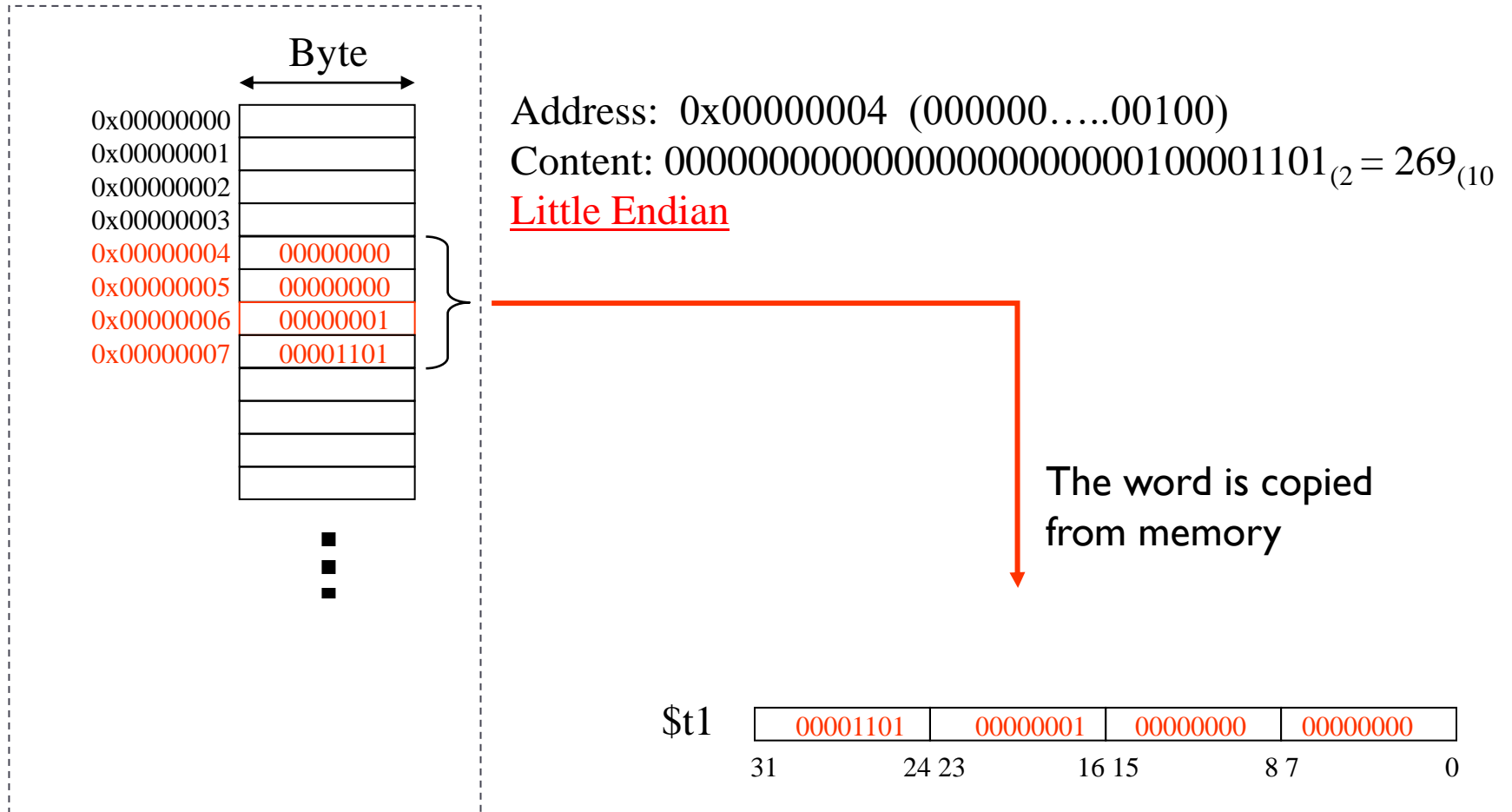
Read word from memory

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



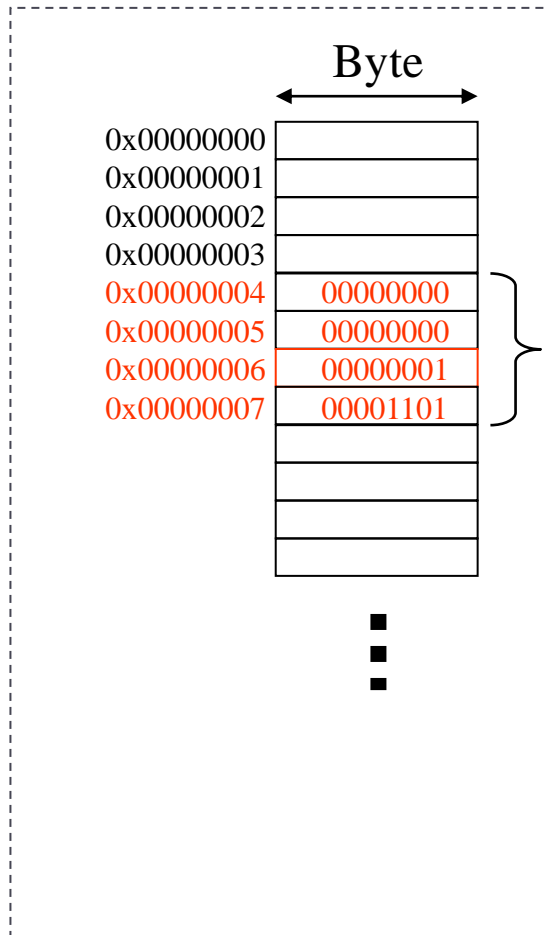
Read word from memory

lw \$t1, 0x4



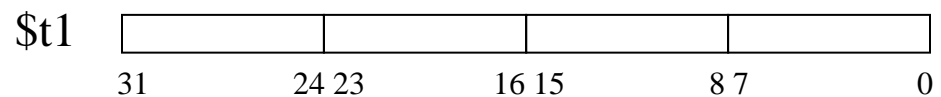
Read word from memory

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



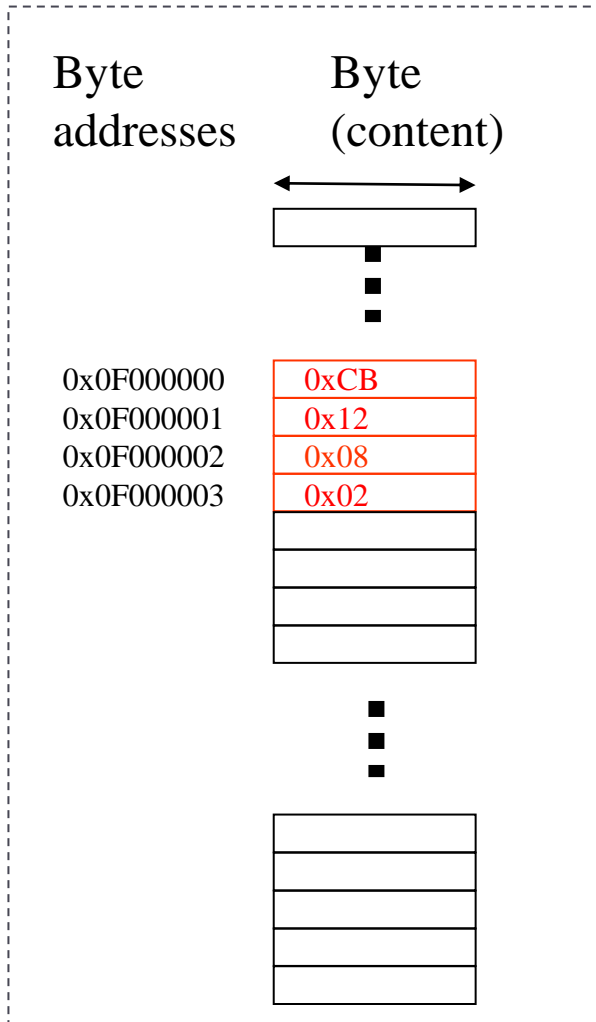
The address of the first byte is specified.

Access to the stored word starting from the address 0x00000004

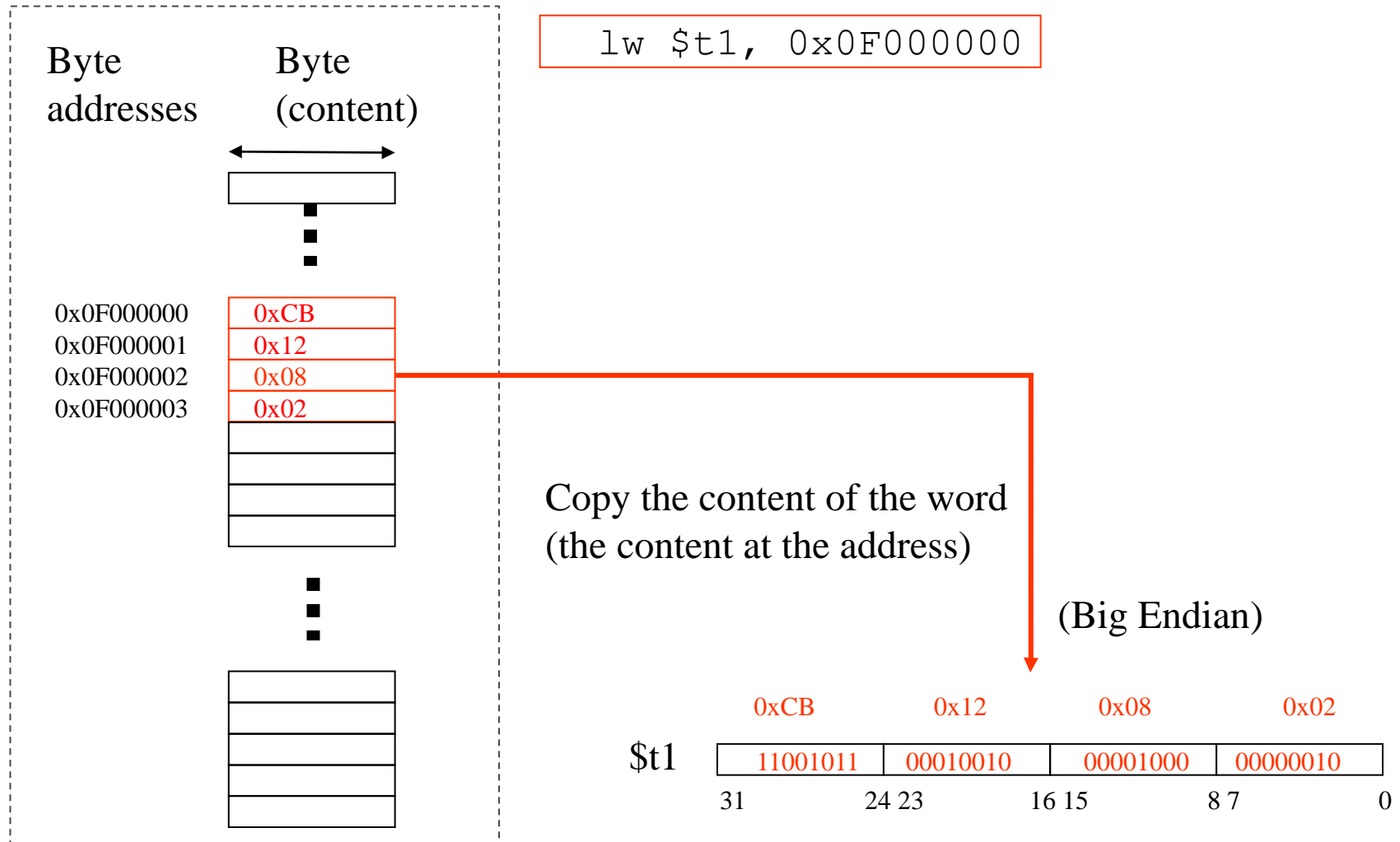


Differences among lw, lb, lbu, la

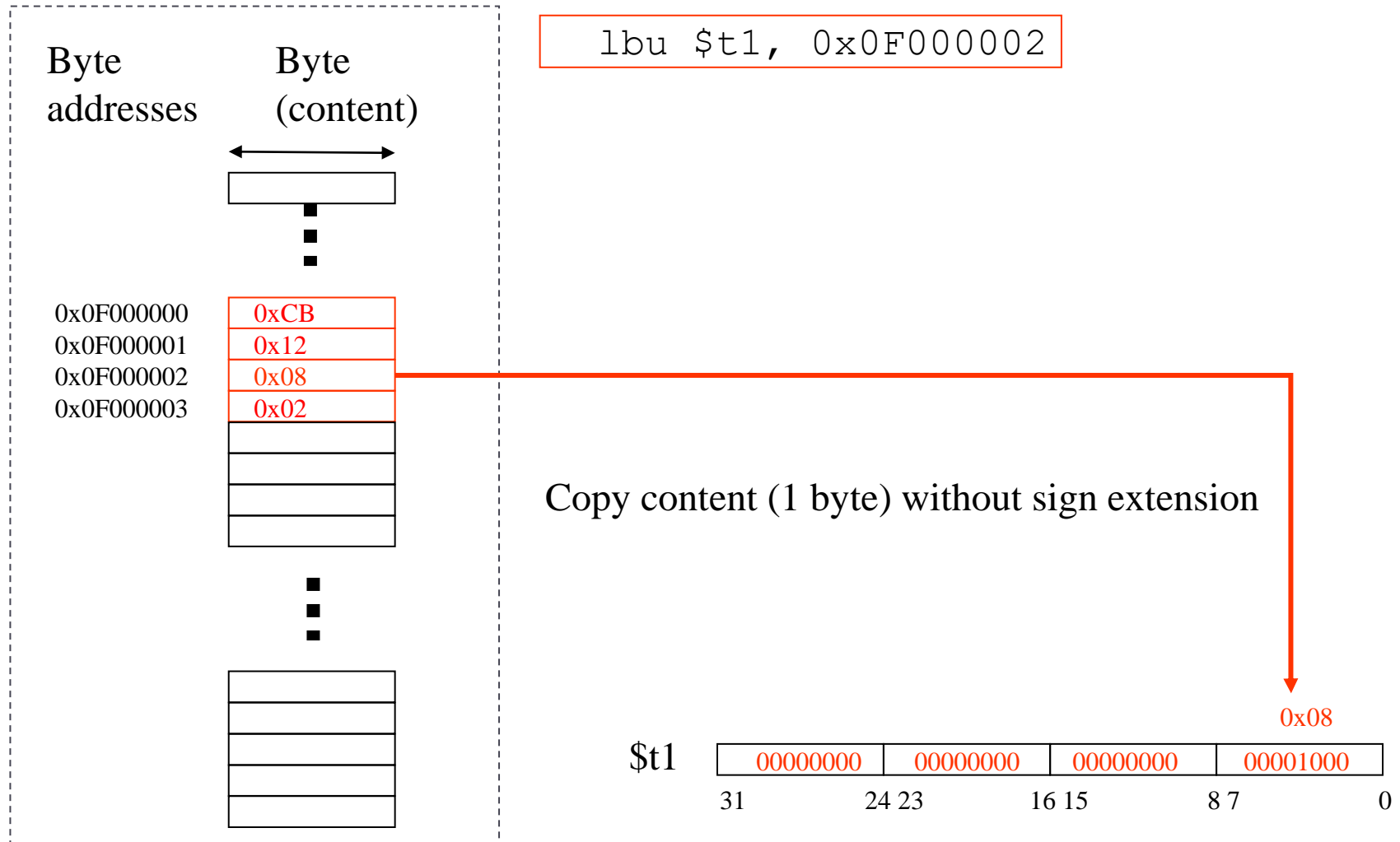
```
lw $t1, 0x0F000000
```



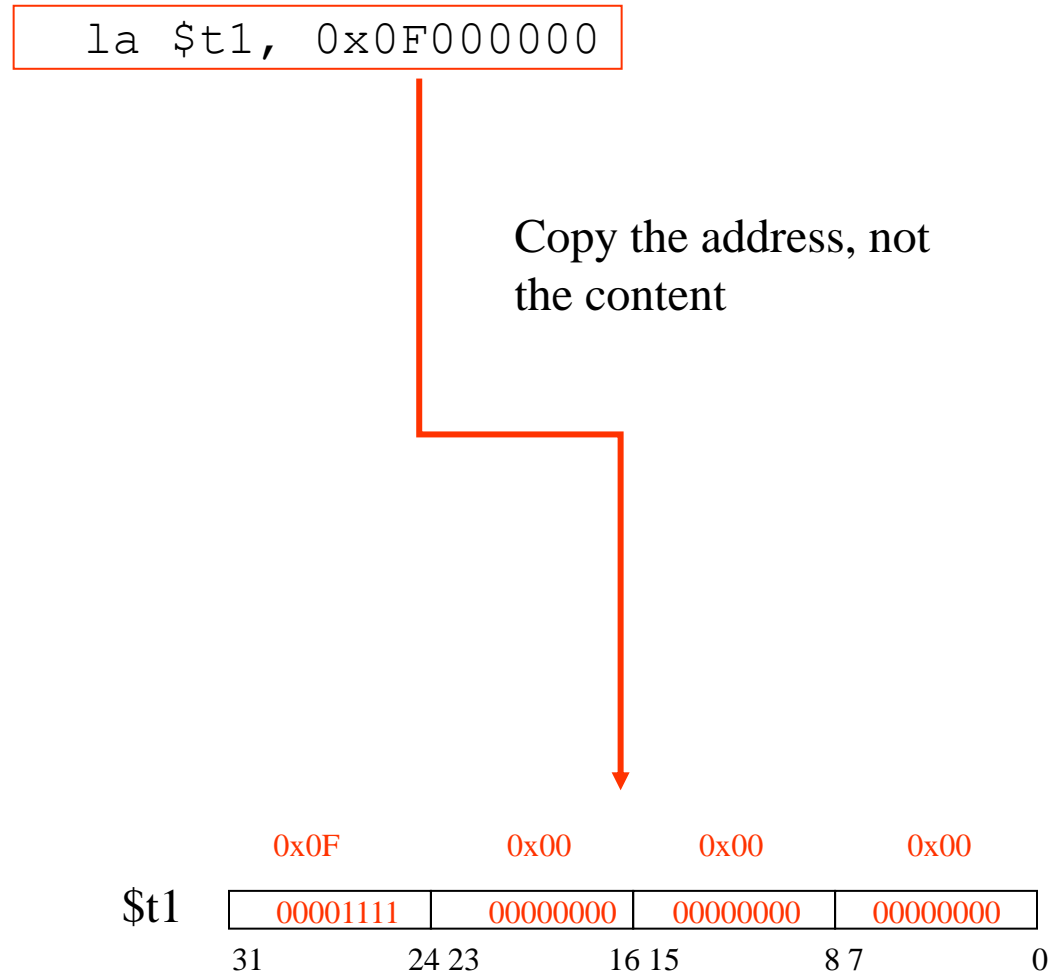
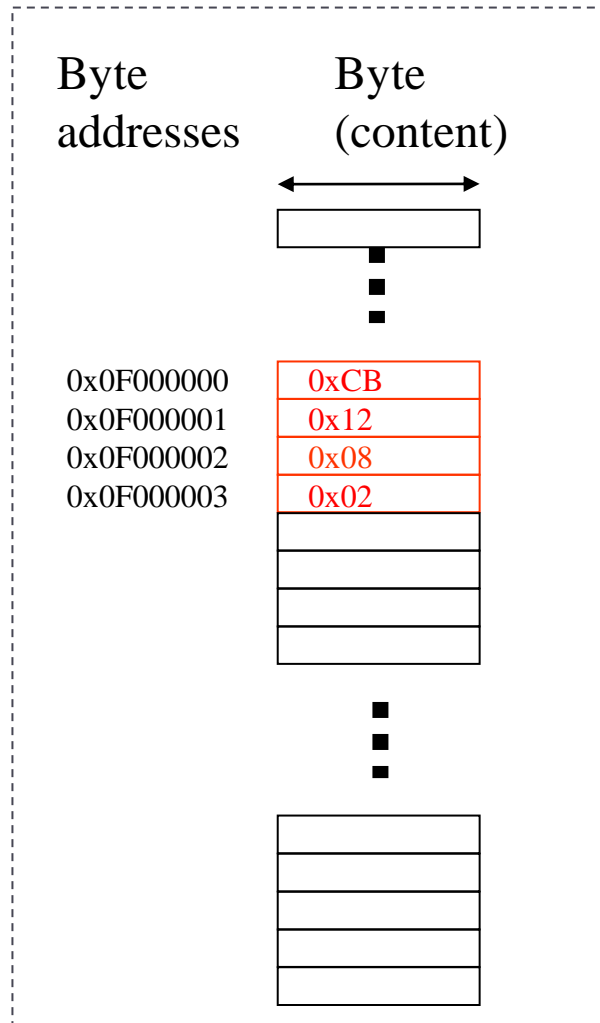
Differences among lw, lb, lbu, la



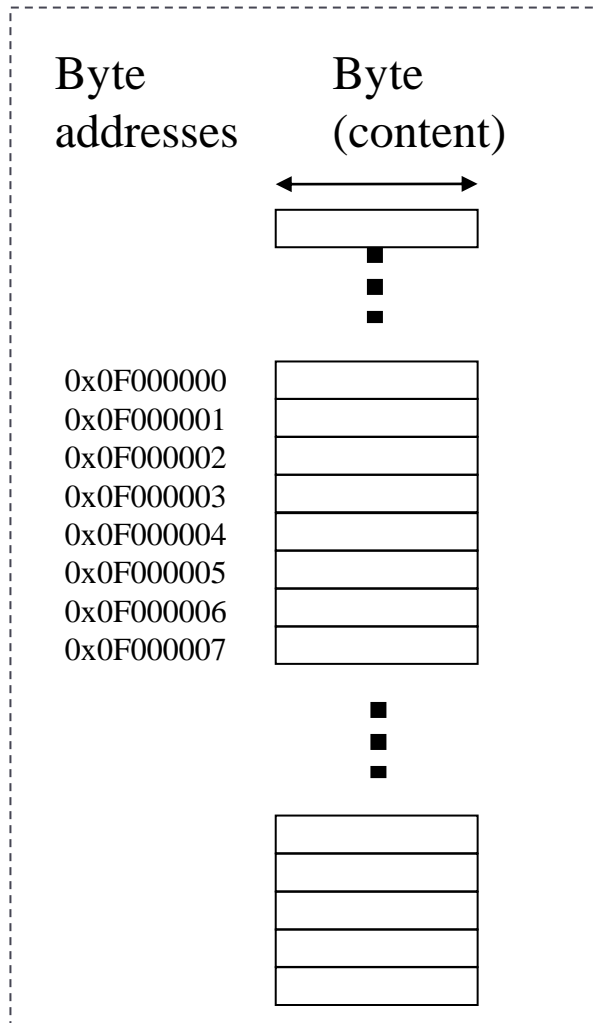
Differences among lw, lb, lbu, la



Differences among lw, lb, lbu, la

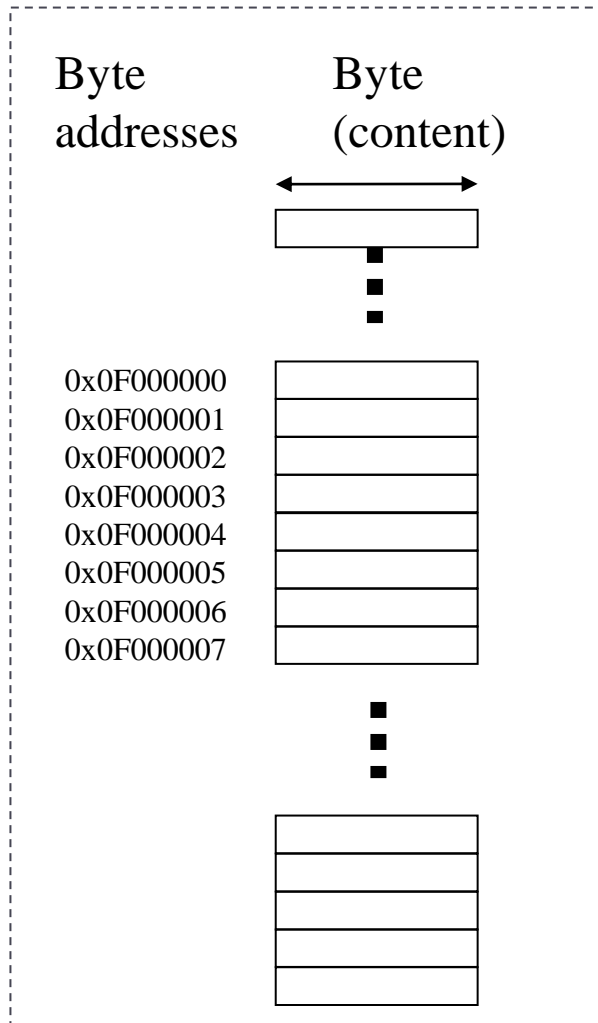


Example

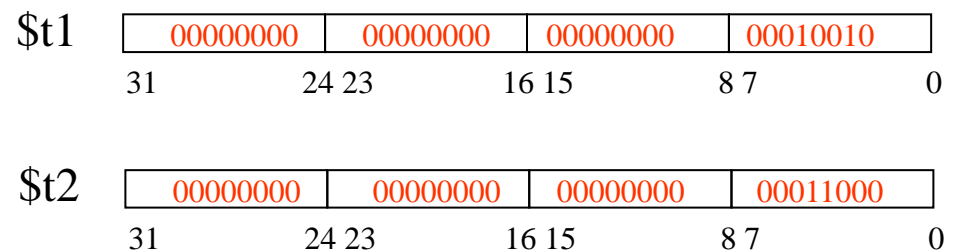


```
li $t1, 18  
li $t2, 24
```

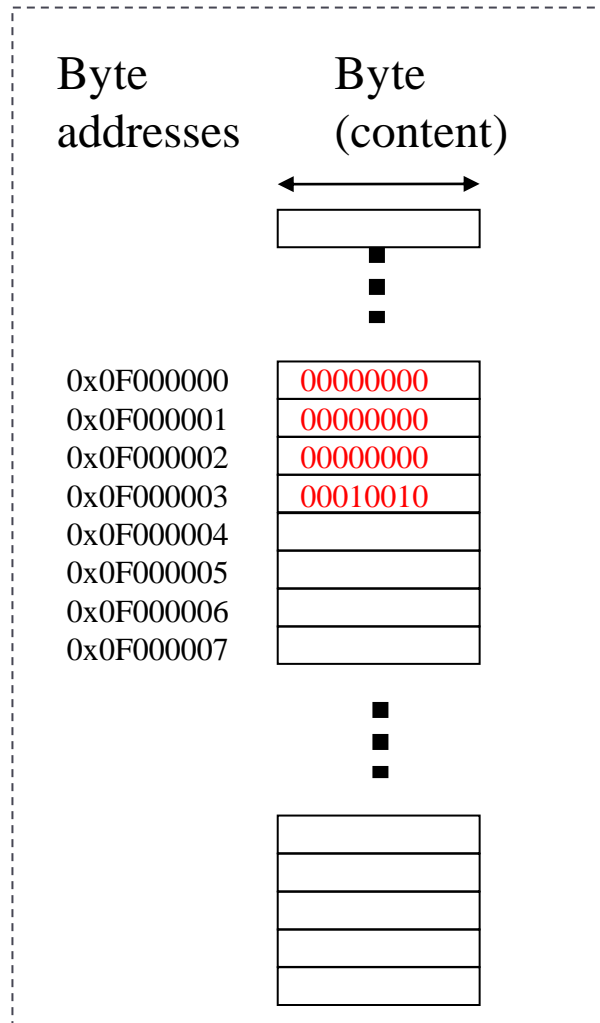
Example



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

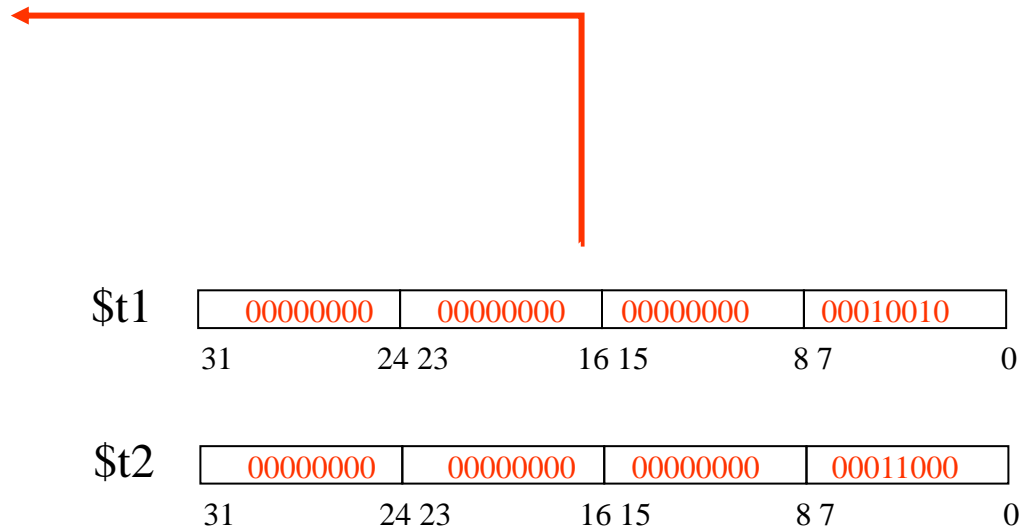


Write word in memory

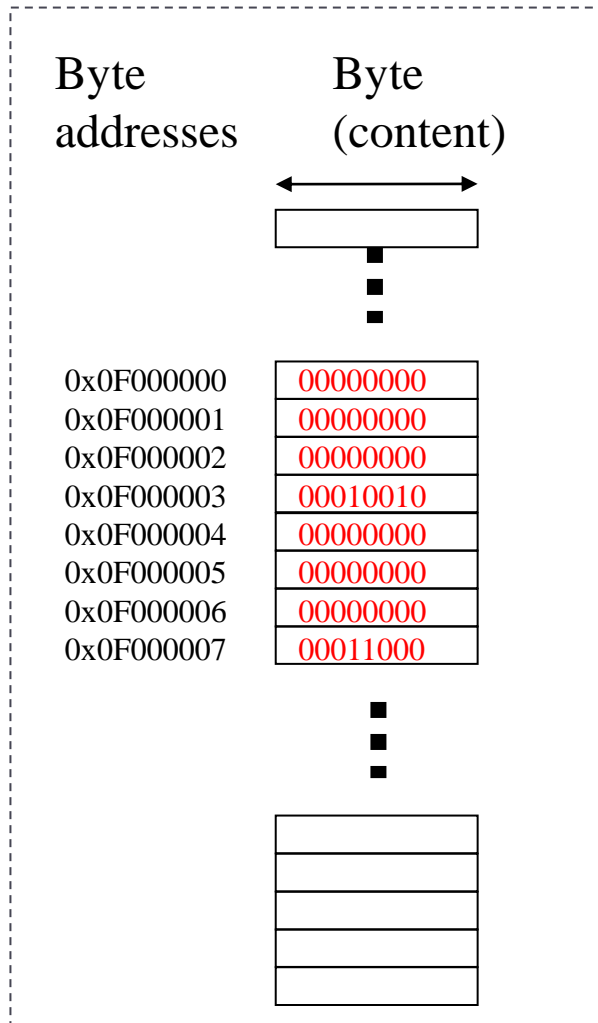


```
sw $t1, 0x0F000000
```

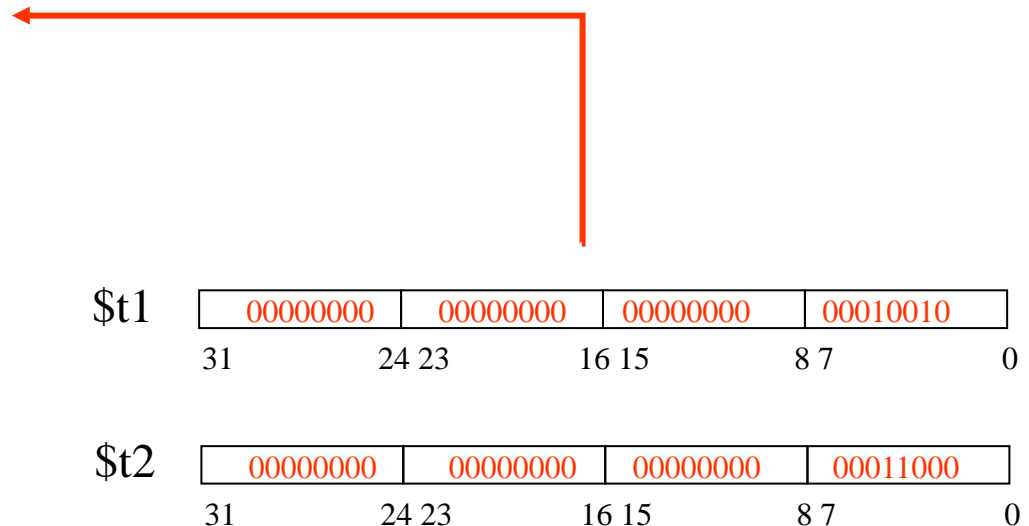
Write the content of a register into memory
(the full word value stored in the register)



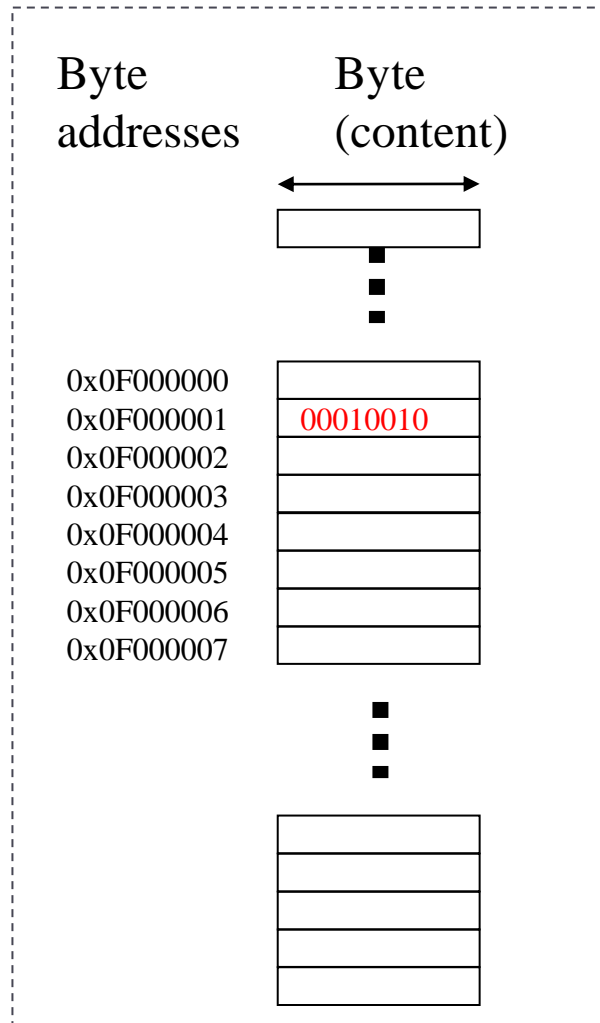
Write word in memory



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

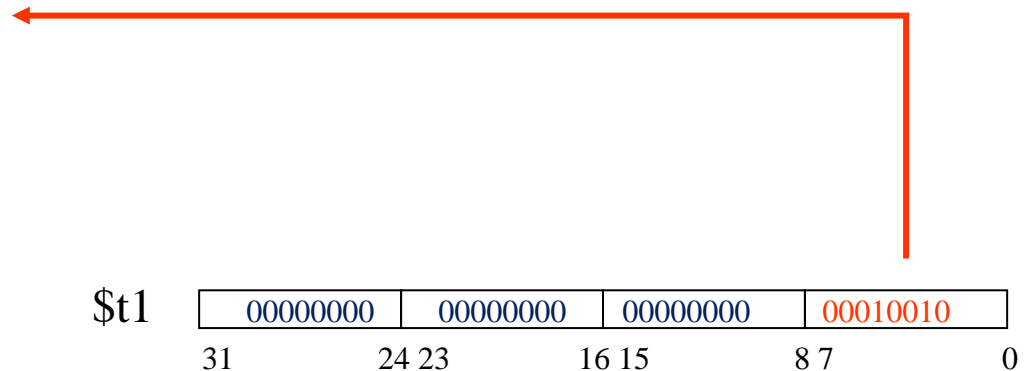


Write byte in memory



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

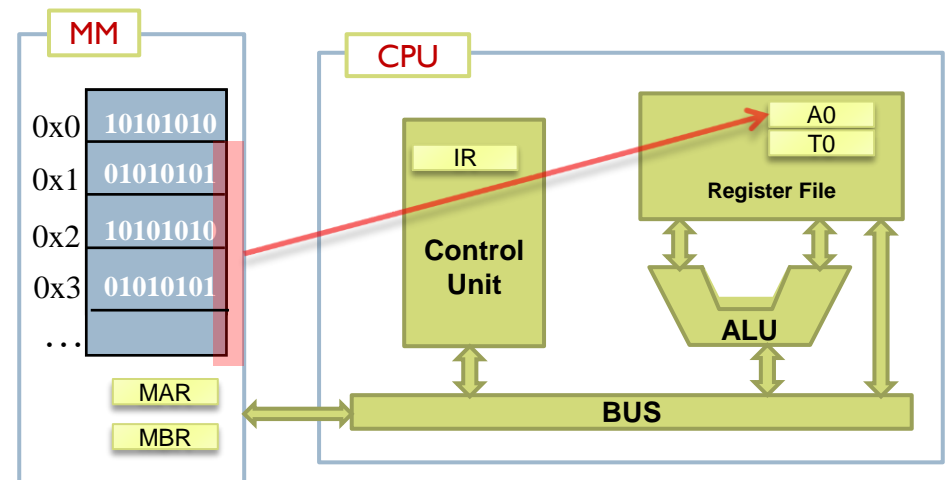
Write the **less significant byte** of register \$t1 in memory



Data transfer alignment and access size

► Peculiarities:

- Alignment of elements in memory
- Default access size



Data alignment

- ▶ In general:

- ▶ A data of K bytes is aligned when the address D used to access this data fulfills the condition:

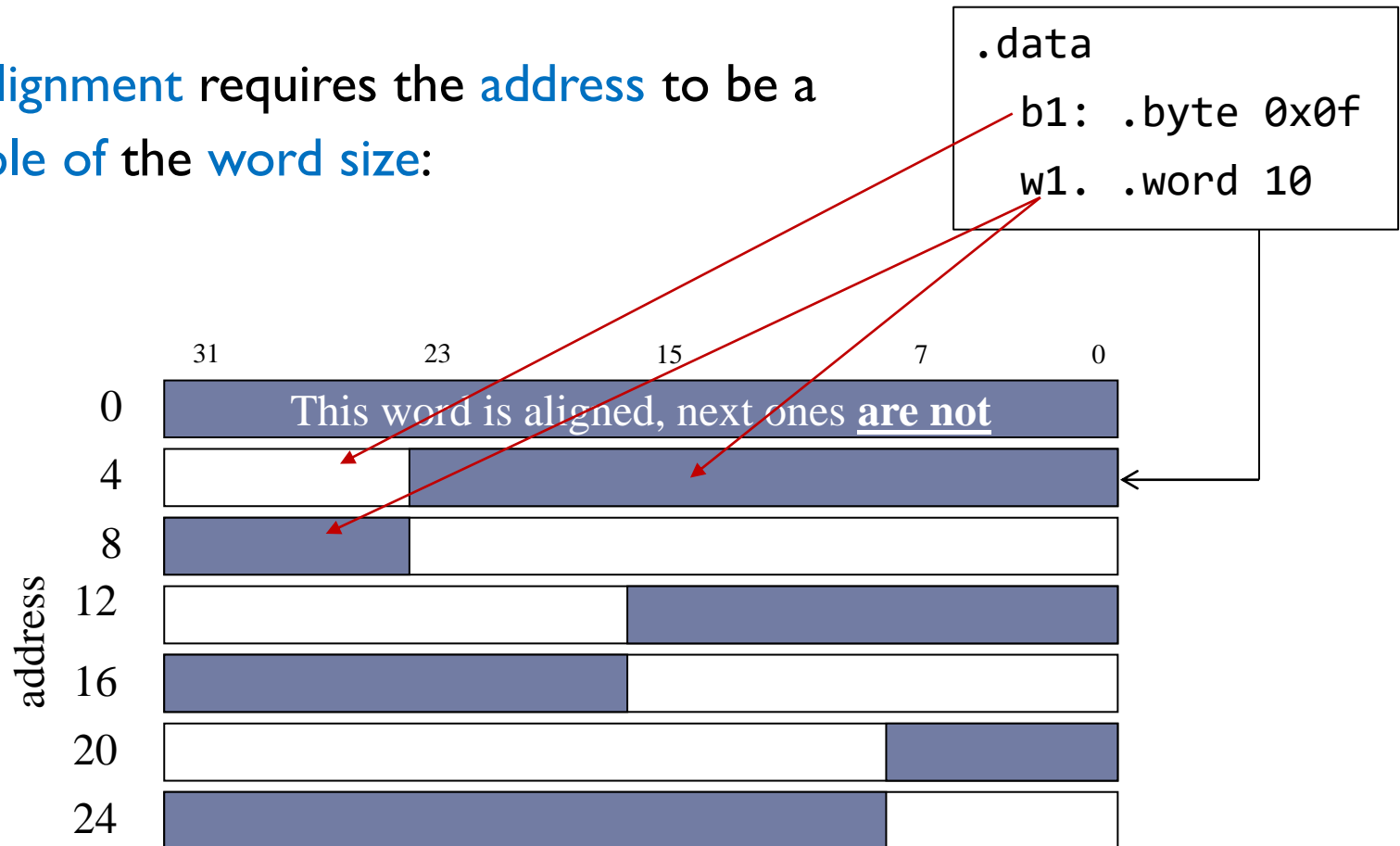
$$D \bmod K = 0$$

- ▶ Data alignment implies:

- ▶ Data of 2 bytes are stored in even addresses
 - ▶ Data of 4 bytes are stored in addresses multiple of 4
 - ▶ Data of 8 bytes (double) are stored in addresses multiple of 8

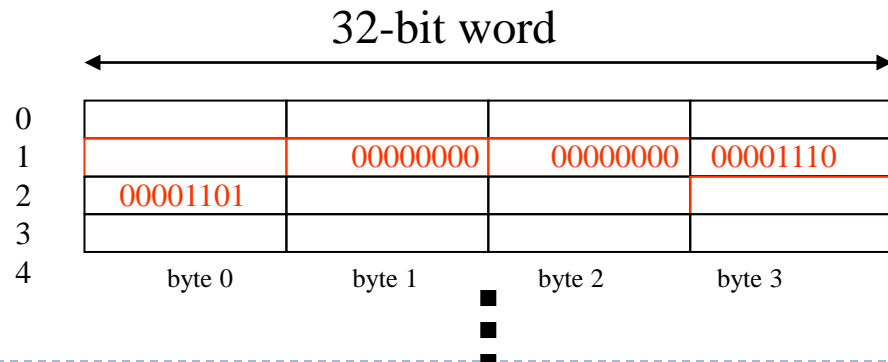
Data alignment

- The **alignment** requires the **address** to be a **multiple of the word size**:



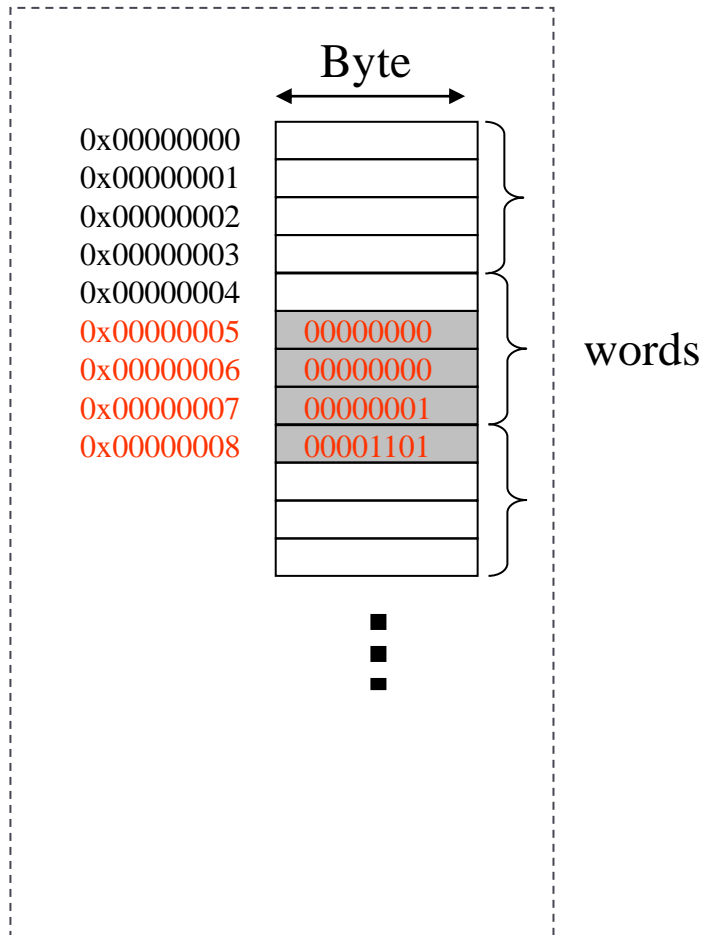
Data alignment

- ▶ Many computers does not allow the access to not aligned data:
 - ▶ Goal: reduce the number of memory accesses
 - ▶ Compilers assign addresses aligned to variables
- ▶ Some processors, such as Intel models, allow the access to not aligned data:
 - ▶ Non-aligned data needs several memory access



Non-aligned data

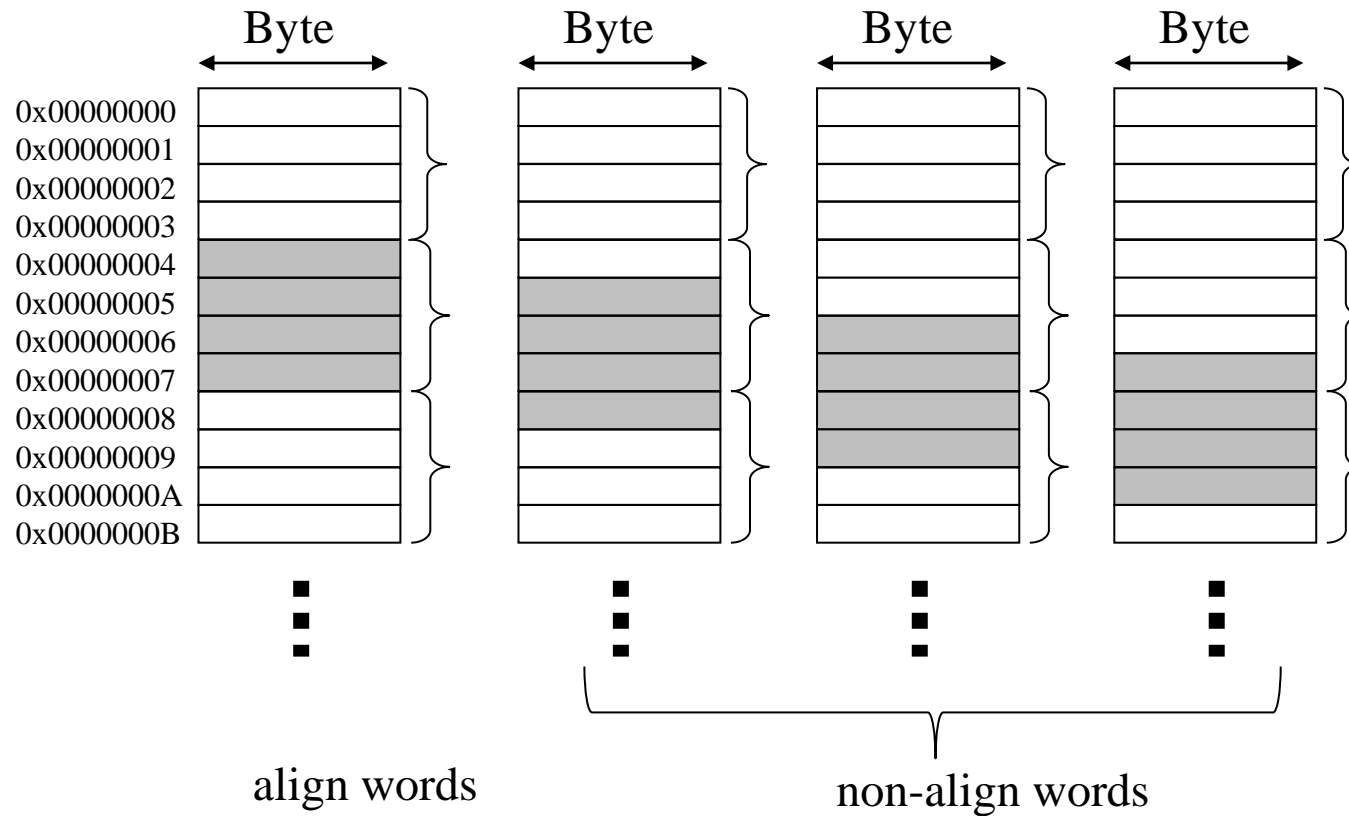
```
lw $t1, 0x05    ????
```



A word stored at address 0x05 is **not aligned** because it is stored in two consecutive aligned memory words.

A aligned word must be stored starting from an address that is a multiple of 4.

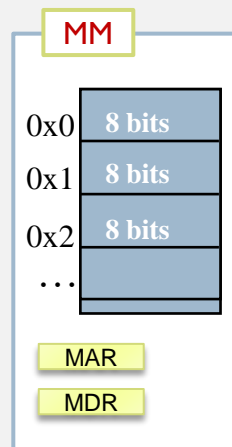
Non-aligned data



Word-level or byte-level addressing

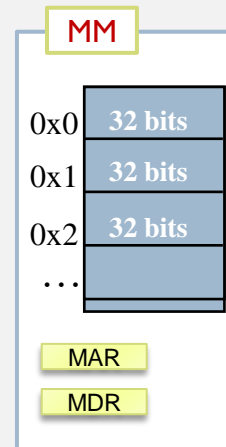
- ▶ The **main memory** is similar to a large one-dimensional vector of items.
- ▶ A **memory address** is the index of one item in the vector.
- ▶ There are two **types of addressing**:

- ▶ **Byte addressing**



- ▶ Each memory element is 1 **byte**
- ▶ Transferring a **word** means transferring 4 **bytes** (in a 32-bit CPU)

- ▶ **Word addressing**



- ▶ Each memory element is a **word**
- ▶ **1b** means transferring one **word** and keeping one **byte**.

Summary


- ▶ The instructions and data of a program must be loaded in memory for the execution (process)
- ▶ All data and instructions are stored in memory so all have an associated memory address where is stored
- ▶ In a 32-bit computer such as MIPS 32:
 - ▶ Registers have 32 bits
 - ▶ Memory can store bytes (8 bits)
 - ▶ Instructions: memory → register: `lb, lbu`
 - ▶ Instructions: register → memory: `sb`
 - ▶ Memory can store words (32 bits)
 - ▶ Instructions: memory → register: `lw`
 - ▶ Instructions: register → memory: `sw`

Format of the memory access instructions

summary

lw
sw
lb
sb
lbu

Register, memory address

- 
- Number that represent an address
 - Symbolic label that represents the associated address
 - (register): address is stored in the register
 - num(register): represent the address that is obtained by adding num with the address stored in the register

Memory access instruction formats

summary

- ▶ `lbu $t0, 0x0F000002`
 - ▶ Direct addressing. The byte stored at memory location `0x0F000002` is loaded into `$t0`.
- ▶ `lbu $t0, labeled`
 - ▶ Direct addressing. The byte stored in the memory location `labeled` is loaded into `$t0`.
- ▶ `lbu $t0, ($t1)`
 - ▶ Indirect register addressing. The byte stored in the memory location stored in `$t1` is loaded in `$t0`.
- ▶ `lbu $t0, 80($t1)`
 - ▶ Relative addressing. The byte stored in the memory location obtained by adding the contents of `$t1` with 80 is loaded in `$t0`.

Instructions to write in memory

summary

- ▶ `sw $t0, 0x0F000000`

- ▶ **Copy the word stored in \$t0 in the address**
`0x0F000000`

- ▶ `sb $t0, 0x0F000000`

- ▶ **Copy the (least significant) byte stored in \$t0 in the address**
`0x0F000000`

Assembly data types

▶ Basic

- ▶ Booleans
- ▶ Characters
- ▶ Integers
- ▶ Decimals (float/double)

▶ Compound

- ▶ Vector
- ▶ String
- ▶ Matrix
- ▶ Others... (struct)

Basic data types

booleans

```
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)  
      ...
```

Basic data types

characters

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

```
...
```

```
main ()
```

```
{
```

```
    c1 = c2;
```

```
    ...
```

```
}
```

```
.data
```

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

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

```
...
```

```
.text
```

```
.globl main
```

```
main:  la  $t0 c1
```

```
        lbu $t1 c2
```

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

```
...
```

Basic data types

Integers

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

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

```
.data  
.align 2  
result:  .word    0 # 4 bytes  
op1:     .word   100  
op2:     .word  -10
```

...

```
.text  
.globl main
```

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

...

Basic data types

Integers

global variable without initial value

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

global variable with initial value

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

```
.data  
.align 2  
result: .word 0 # 4 bytes  
op1:     .word 100  
op2:     .word -10  
...
```

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


Exercise

- Write in MIPS-32 assembly a fragment of code with the same functionality that:

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

Assuming that a, b, c and d are variables stored in memory

Basic data types

float

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

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

```
.data
```

```
.align 2
```

```
    result:  .word  0 # 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 result
```

```
    ...
```

Basic data types

double

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

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

.data

.align 3

```
result:  .space      8  
op1:     .double    100  
op2:     .double   -10.27
```

...

.text

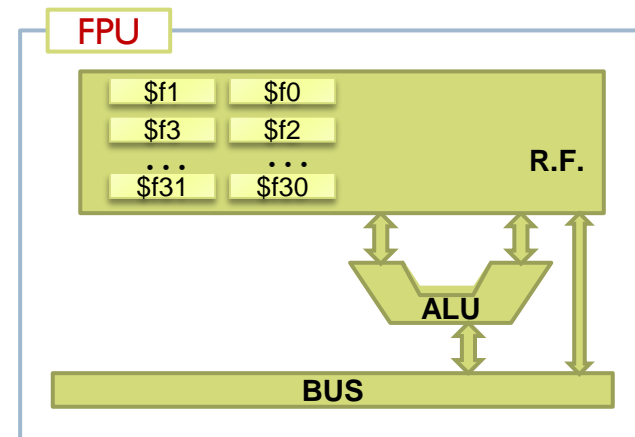
.globl main

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

...

Floating point. IEEE 754

- ▶ The coprocessor 1 has 32 registers of 32 bits (4 bytes) each.
 - ▶ It is possible to work with single or double precision
- ▶ Simple precision (32 bits):
 - ▶ From \$f0 to \$f31
 - ▶ E.g.: `add.s $f0 $f1 $f5`
 $f0 = f1 + f5$
 - ▶ Other operations:
 - ▶ `add.s, sub.s, mul.s, div.s, abs.s`
- ▶ Double precision (64 bits):
 - ▶ Registers used in pairs
 - ▶ E.g.: `add.d $f0 $f2 $f8`
 $(f0, f1) = (f2, f3) + (f8, f9)$
 - ▶ Other operations:
 - ▶ `add.d, sub.d, mul.d, div.d, abs.d`



Data transfer

IEEE 754

- ▶ Copies a **number** from **memory** to a **register** or vice versa.

- ▶ Examples:

- ▶ Memory to register

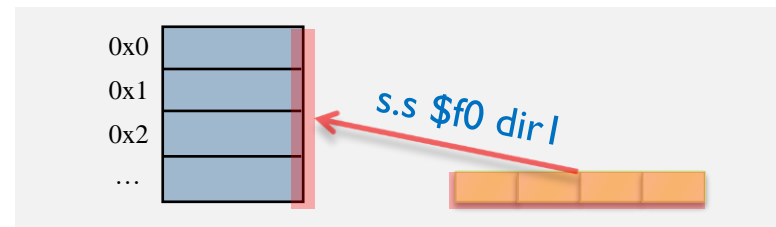
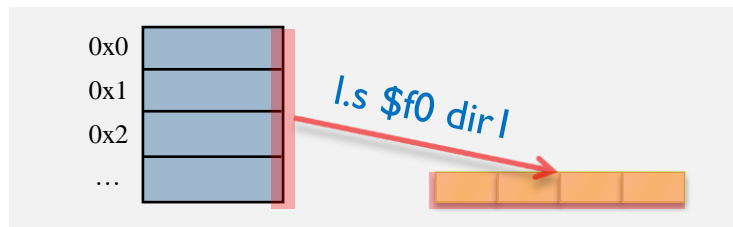
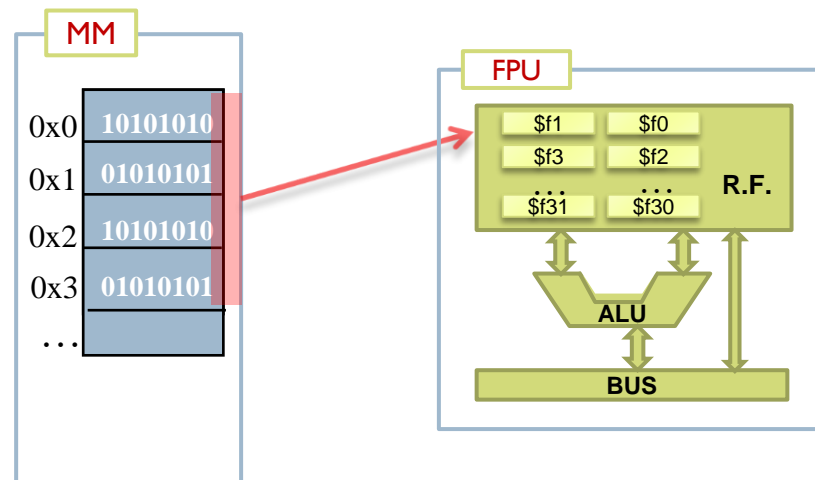
l.s \$f0 dir1

l.d \$f2 dir2

- ▶ Register to memory

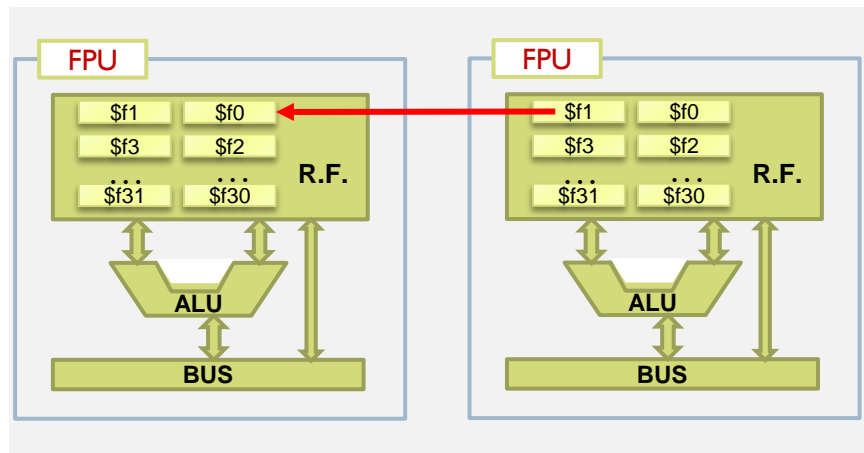
s.s \$f0 dir1

s.d \$f0 dir2



Operations with registers (FPU, FPU)

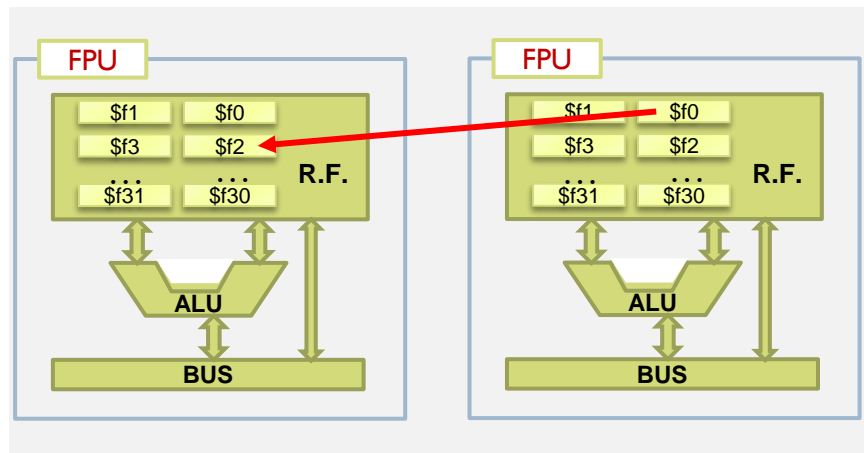
`mov.s $f0 $f1`



`$f0 ← $f1`

Operations with registers (FPU, FPU)

`mov.d $f0 $f2`



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

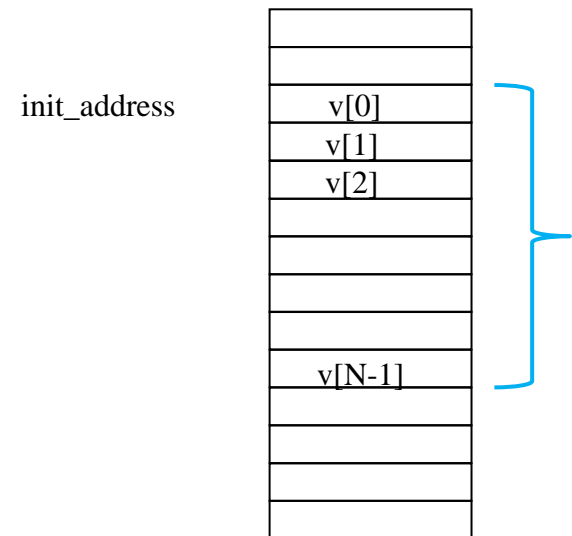
Compound data types

Arrays

- ▶ Collection of data items stored consecutively in memory
- ▶ The address of the j element can be computed as:

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

Where p is the size of each item



Compound data types

Arrays

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

```
...
```

```
main ()
```

```
{
```

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

```
}
```

```
.data
```

```
.align      2 # next item aligned to 4
```

```
vec: .space 20 # 5 items * 4 bytes/item
```

```
.text
```

```
.globl main
```

```
main:
```

```
    la $t1, vec
```

```
    li $t2, 8
```

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

```
    ...
```

Compound data types

Arrays

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

```
...
```

```
main ()
```

```
{
```

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

```
}
```

```
.data
```

```
.align      2 # next item aligned to 4
```

```
vec: .space 20 # 5 items * 4 bytes/item
```

```
.text
```

```
.globl main
```

```
main:
```

```
    li    $t0, 16
```

```
    la    $t1, vec
```

```
    add   $t3, $t1, $t0
```

```
    li    $t2, 8
```

```
    sw    $t2, ($t3)
```

```
...
```

Compound data types

Arrays

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

```
...
```

```
main ()
```

```
{
```

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

```
}
```

```
.data
```

```
.align 2          # next item align to 4
```

```
vec: .space 20    # 5 items * 4 bytes/item
```

```
.text
```

```
main:
```

```
    li $t2 8
```

```
    li $t1 16
```

```
    sw $t2 vec($t1)
```

```
...
```

Exercise

- ▶ Let V be an array of integer elements
 - ▶ V represents the initial address of the array
- ▶ What is the address of the $V[5]$ item?
- ▶ Which are the instructions to load in register $\$t0$ the value of $v[5]$?

Exercise (solution)

- ▶ Let V be an array of integer elements
 - ▶ V represents the initial address of the array
- ▶ What is the address of the $V[5]$ item?
 - ▶ $V + 5 * 4$
- ▶ Which are the instructions to load in register $\$t0$ the value of $v[5]$?
 - ▶ `li $t1, 20`
 - ▶ `lw $t0, v($t1)`

Compound data types


String

- Array of bytes
- '\0' ends string

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



String layout in memory

```
// 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:	...	
	'h'	0x0108
	'o'	0x0109
	'l'	0x010a
	'a'	0x010b
	0	0x010c
	...	0x010d

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

Exercise

```
// global variables
```

```
char v1;
```

```
int v2 ;
```

```
float v3 = 3.14 ;
```

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

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

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


Exercise (solution)

```
// variables globales
```

```
char v1;
```

```
int v2 ;
```

```
float v3 = 3.14 ;
```

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

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

```
.data
```

```
v1: .byte 0
```

```
.align 2
```

```
v2: .space 4
```

```
v3: .float 3.14
```

```
v4: .ascii "ec"
```

```
.align 2
```

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

Exercise (solution)

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

```
.data

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

v4: .ascii "ec"

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

Exercise (solution)

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: .space 4
v3: .float 3.14

v4: .ascii "ec"

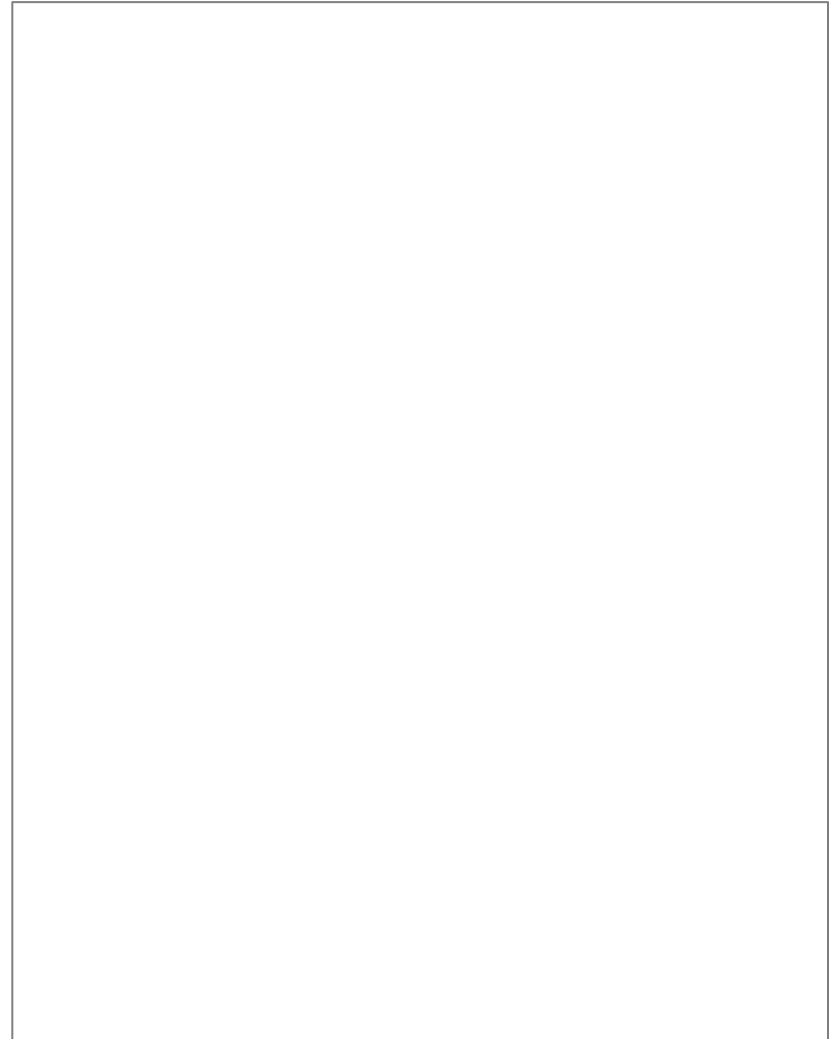
.align 2
v5: .word 20, 22
```

Compound data types

String length

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

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



Compound data types

String length

```
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:   .word 0      # pointer => address
```

...

```
.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
```

...

Arrays and strings

► Review (in general) :

► `lw $t0, 4($s3) # $t0 ← M[$s3+4]`

► `sw $t0, 4($s3) # M[$s3+4] ← $t0`

Exercise

- ▶ Write a program that:
 - ▶ Calculate the number of occurrences of a char in a string
 - ▶ String address stored in \$a0
 - ▶ Char to look for in \$a1
 - ▶ Result must be stored in \$v0

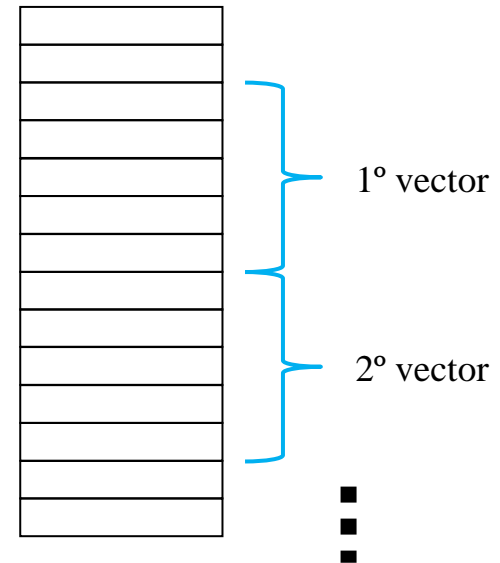
Compound data types

Matrix

- ▶ A matrix $m \times n$ consists of m vectors (m rows) of length n
- ▶ Usually stored by rows
- ▶ The element a_{ij} is stored in the address:

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

where p is the size of each item



Compound data types

Matrix

```
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           # next item align to 4  
vec: .space 20     # 5 item * 4 bytes/item  
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
```

```
...
```

Example: integer matrix

```
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 items*4 bytes/item  
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
```

...

Tips

- ▶ Do not program directly in assembler
 - ▶ Better to **first do the design** in DFD, Java/C/Pascal...
 - ▶ Gradually translate the design to assembler.
- ▶ Sufficiently **comment** the code and data
 - ▶ By line or by group of lines
comment which part of the design implements.
- ▶ **Test** with enough test cases
 - ▶ Test that the final program works properly to the given specifications.

Exercise

- ▶ Write an assembly program that:
 - ▶ Load the value -3.141516 in register \$f0
 - ▶ Obtain the exponent and mantissa values stored in the register \$f0 (IEEE 754 format)
 - ▶ Display the sign
 - ▶ Display the exponent
 - ▶ Display the mantissa

Exercise (solution)

```
.data
    newline: .asciiz "\n"

.text
.globl main
main:

    li.s    $f0, -3.141516

    # print value
    mov.s   $f12, $f0
    li $v0, 2
    syscall

    la $a0, newline
    li $v0, 4
    syscall

    # copy to processor
    mfc1    $t0, $f12
```

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

la    $a0, newline
li    $v0, 4
syscall

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

la    $a0, newline
li    $v0, 4
syscall

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

jr    $ra
```