ARCOS Group

uc3m | Universidad Carlos III de Madrid

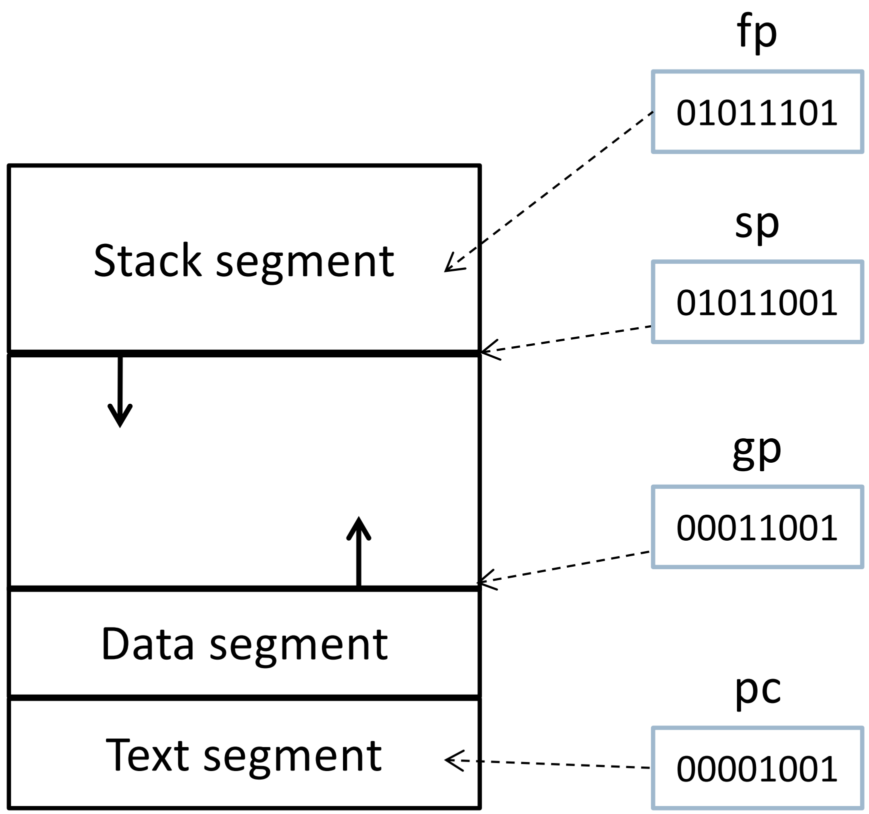
Lesson 3 (II)
Fundamentals of assembler programming

Computer Structure
Bachelor in Computer Science and Engineering

Contents

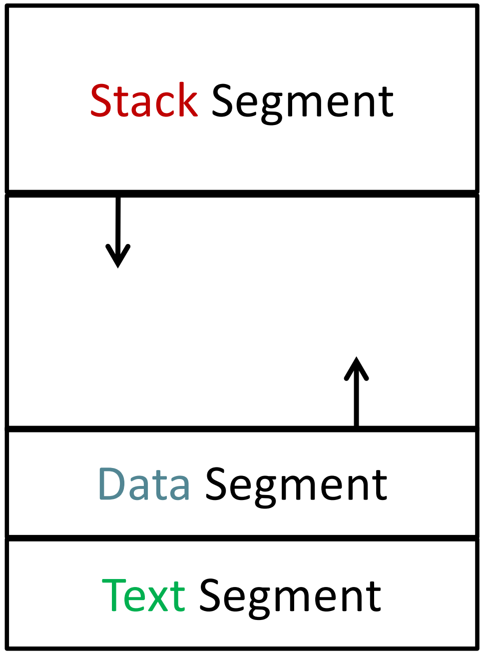
- Basic concepts on assembly programming
- RISC-V_{32} assembly language, memory model and data representation
- Instruction formats and addressing modes
- Procedure calls and stack convention

Memory layout for a process



- The memory space is divided in logic segments in order to organize the content:
  - Code segment (text)
    - Program code
  - Data segments
    - Global variables
    - Static variables
  - Stack segment
    - Local variables
    - Function contexts

Storing variables in memory




```
// global variables
int a;

main ()
{
    // local variables
    int b;

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

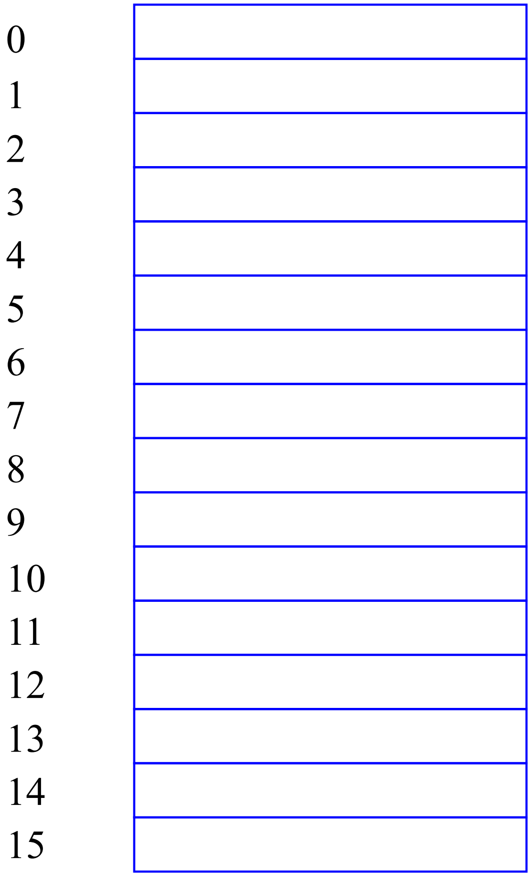

ARCOS @ UC3M
Félix García Carballeira, Alejandro Calderón Mateos

Register File (integers)
summary

Symbolic name	Number	Usage
zero	x0	Constant 0
ra	x1	Return address (routines/functions)
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0...t2	x5-x7	Temporary (NO preserved across calls)
s0/fp	x8	Saved temporary (preserved across calls) / Frame pointer
s1	x9	Saved temporary (preserved across calls)
a0...a1	x10…11	Arguments for routines/return value
a2...a7	12…x17	Arguments for routines
s2… s11	x18…x27	Saved temporary (preserved across calls)
t3...t6	x28…x31	Temporary (NO preserved across calls)

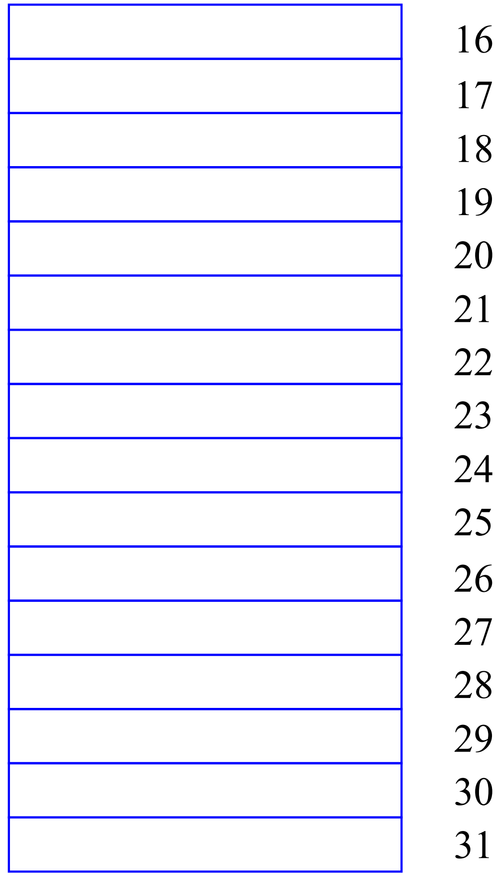
- There are 32 registers
  - Size: 4 bytes (1 word)
  - Used a x at the beginning
- Use convention
  - Reserved
  - Arguments
  - Results
  - Temporary
  - Pointers

RISC-V$_{32}$ Register File

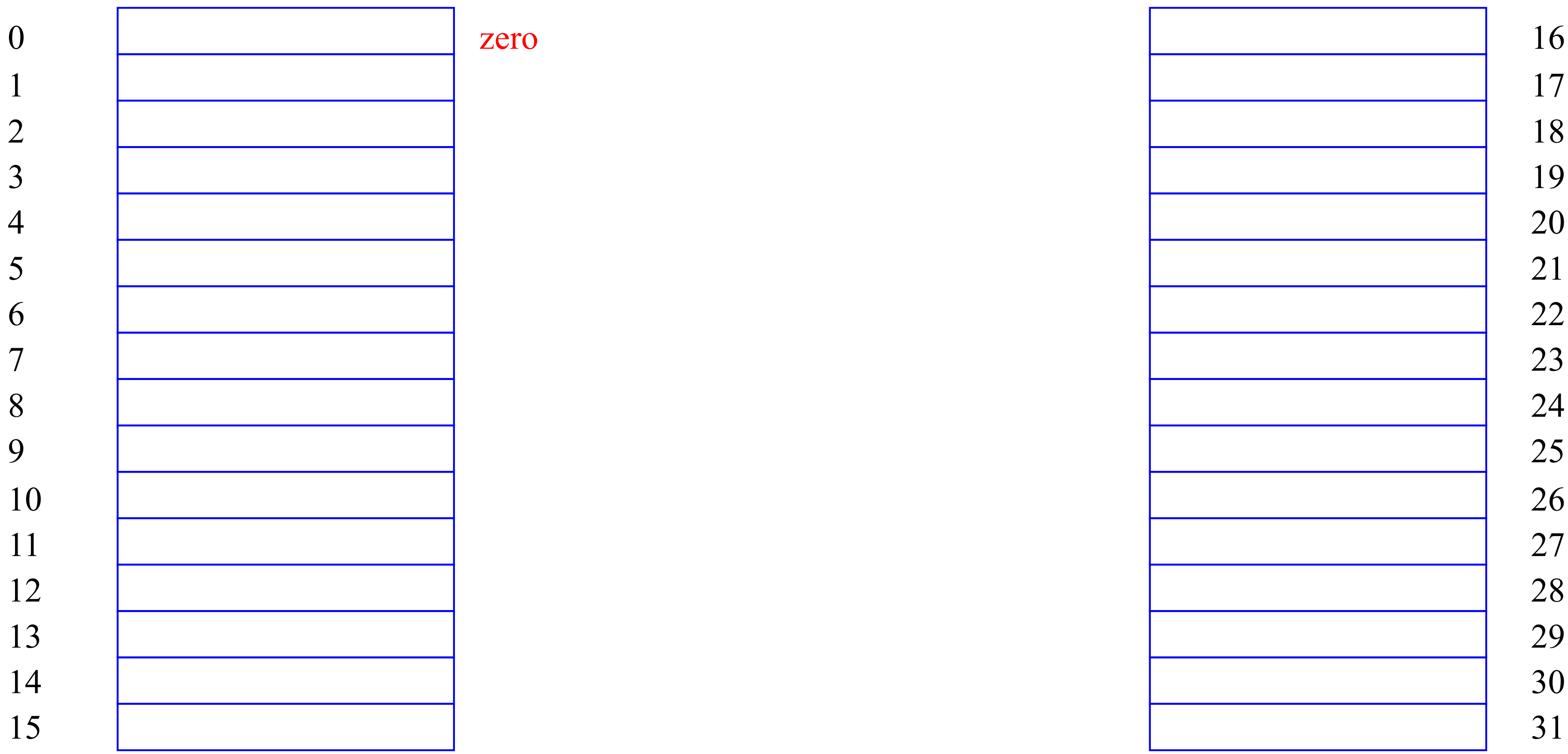


- 32 registers
  - 4 bytes of size (one word)
  - Name starts with x at the beginning

- Usage Convention
  - Reserved
  - Arguments
  - Results
  - Temporary
  - Pointers

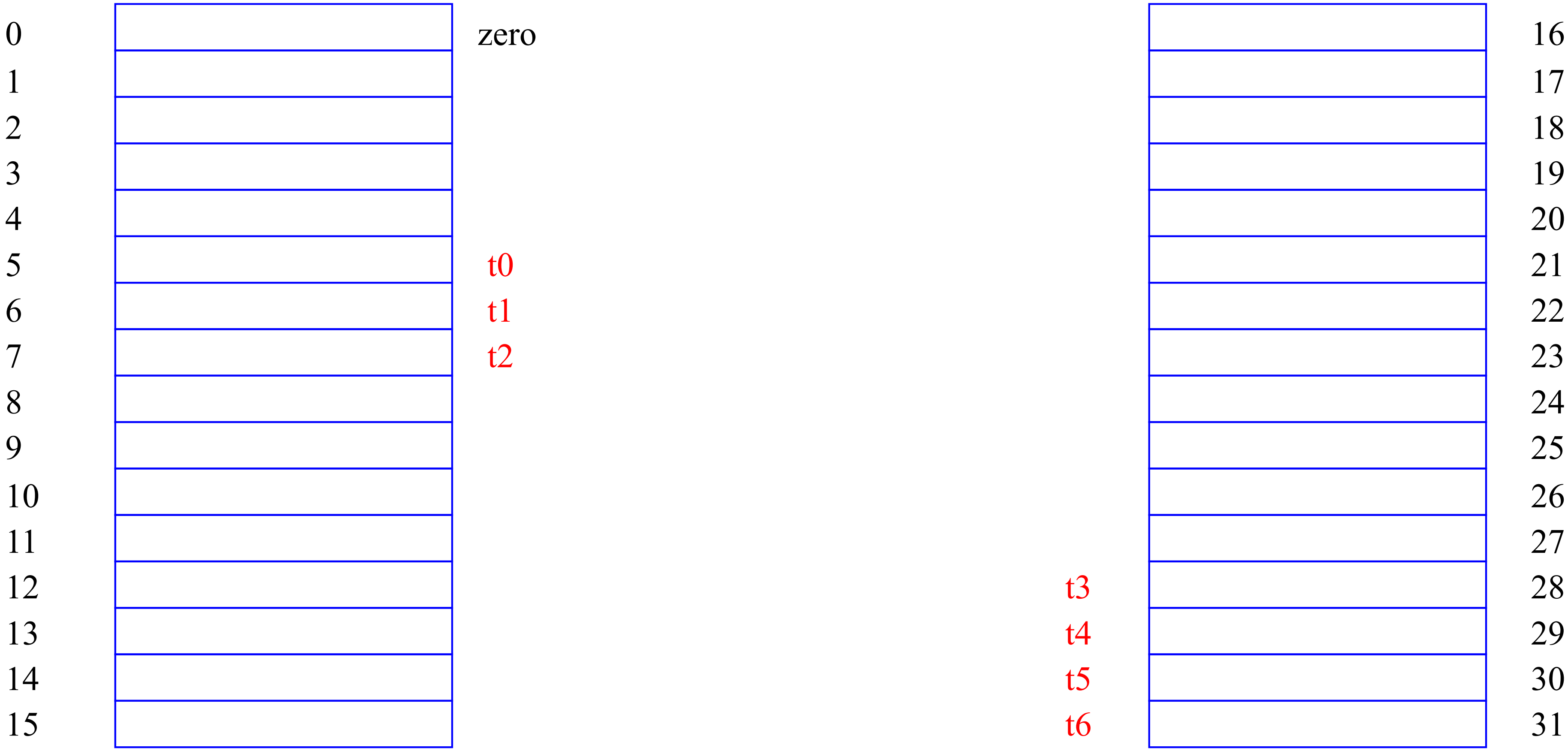


RISC-V_{32} Register File



Constant zero
Cannot be changed

RISC-V$_{32}$ Register File



Temporary registers

RISC-V_{32} Register File

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

Preserved values

RISC-V_{32} Register File

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

Arguments and
subroutines support

RISC-V$_{32}$ Register File

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

Pointers

Example: hello world…

hello.s

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

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

Example: hello world…

hello.s

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

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

label: it represents the associated memory address where main function starts.

comments

instructions

Example: hello world…

hello.s

data segment

```
.data
     msg_hola: .asciiz "hello world\n"
.text
     main:
      # printf("hello world\n") ;
      li a7 4
      la a0 msg_hola
      ecall
```

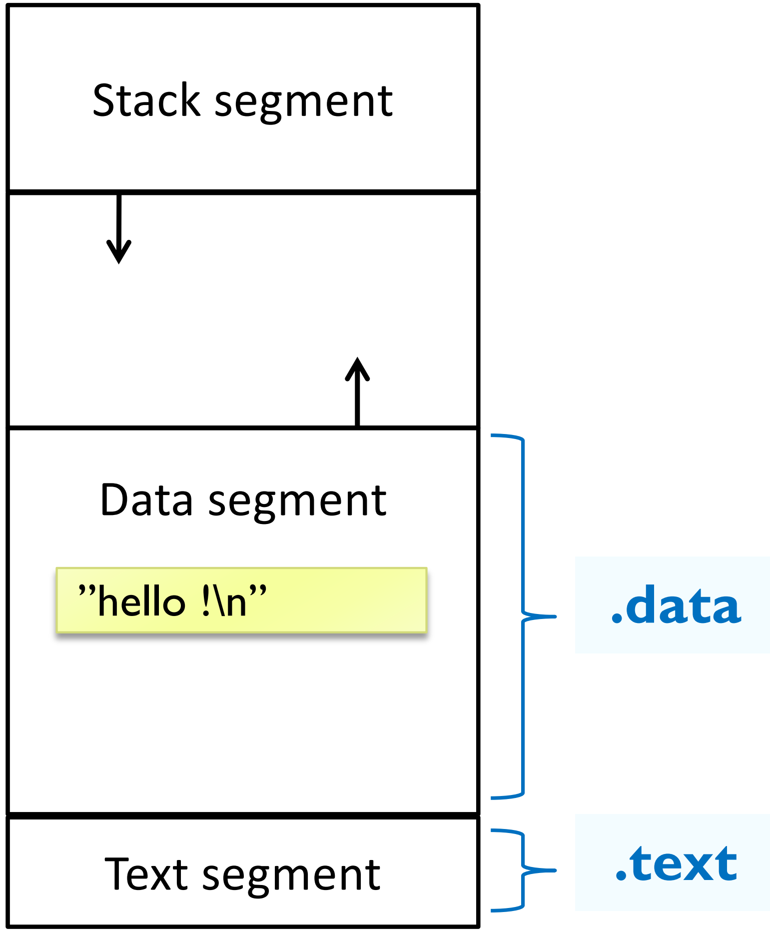
msg_hola: represents the memory address where the string begins to be stored

code segment

Assembly: directives

hello.s

```
.data                                   <- data segment
   msg_hola: .asciiz "hello !\n"

.text                                   <- text/code segment
    .globl main
   main:
    # printf("hello !\n") ;
    li $v0 4
    la $a0 msg_hola                     } instructions
    syscall
```



ARCOS @ UC3M
Félix García Carballeira, Alejandro Calderón Mateos

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
.align *n*	Align next element to a address multiple of 2^n

Static data definition

label (address)

datatype (directive)

value

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

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

Register File (floating point)

Symbolic name	Numbered name	Uso
ft0-ft7	f0 … f7	Temporals (like t…)
fs0-fs1	f8 … f9	Saved (like s…)
fa0-fa1	f10 … f11	Arguments/return (like a0/a1)
fa2-fa7	f12 … f17	Arguments (like a…)
fs2-fs11	f18 … f27	Saved (like s…)
ft8-ft11	f28 … f31	Temporals (like t…)

- There are 32 registers
- For simple precision register are 4 bytes
- For double precision registers are 8 bytes
  - For simple precision, values are stored in the less significant bits
  - For double precision are stored in all bits of the register

System calls

- Many assembler simulators include a small "operating system"
  - The simulators provides ~17 services.

- How to invoke:
  - Call code in register **a7**
  - Other arguments on specific registers
  - Invocation by the **ecall** instruction

```
# printf("hello world\n")
li a7 4
la a0 msg_hola
ecall
```

System calls

Service	Call code (a7)	Arguments	Result
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)

Example: Hello world…

hola.s

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

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

Service	Call code (a7)	Arguments
print_int	1	a0 = integer
print_float	2	fa0 = float
print_double	3	fa0 = double

Operating system invocation instruction

Exercise

```
. . .

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

. . .
```

Exercise (solution)

```
. . .
int valor ;

. . .

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

. . .
```

Service	Call code (a7)	Arguments
print_int	1	a0 = integer
print_float	2	fa0 = float
print_double	3	fa0 = double

```
. . .

# 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

 . . .
```

Instructions and pseudo-instructions

- There is an assembly instruction per machine instruction :
  - Each machine instruction occupies 32 bits in RISC-V_{32}
  - 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 a0, 4
    mv t1, t0

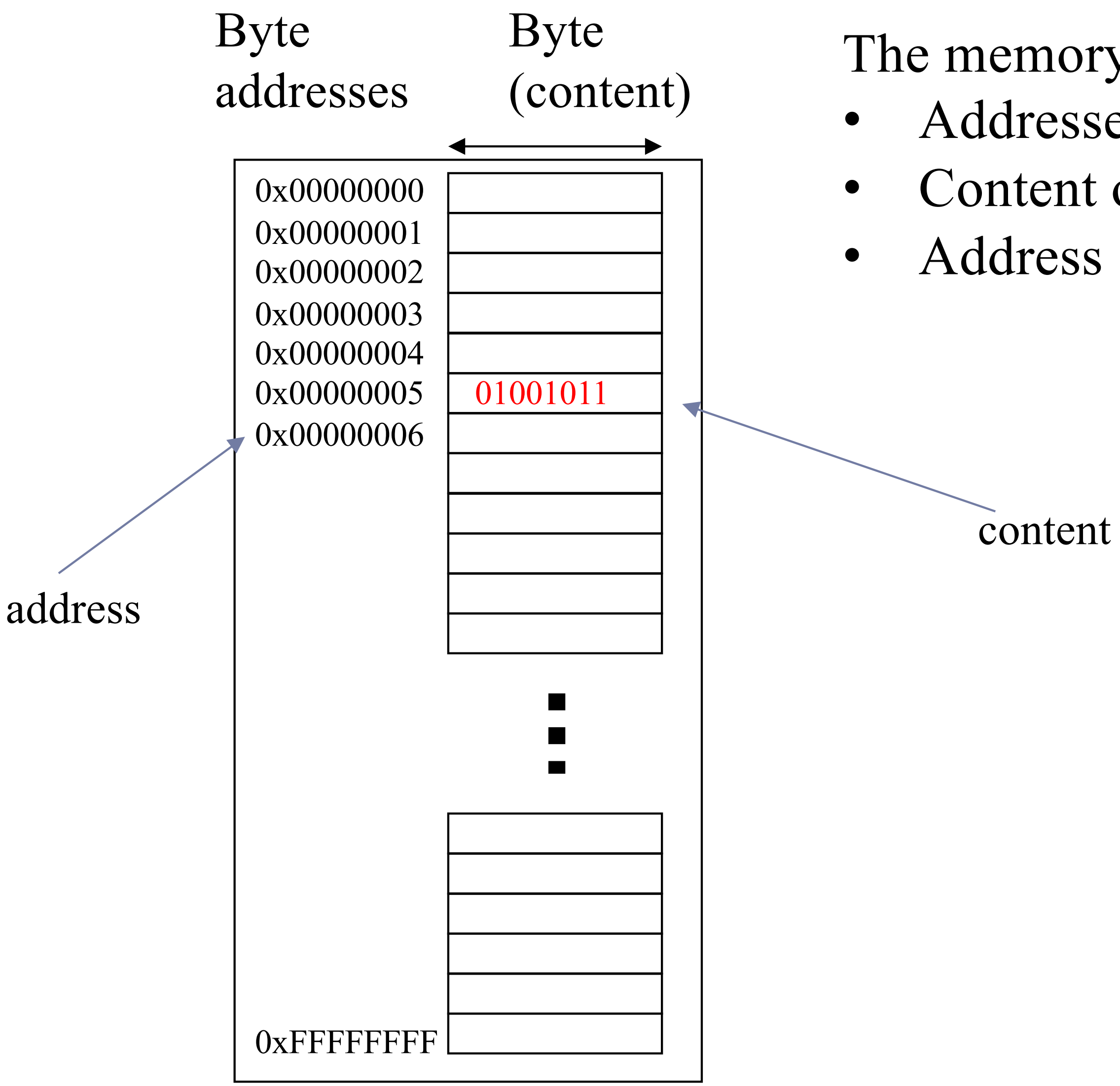
- In the assembly process, they are replaced by the sequence of assembly instructions that perform the same functionality.
  - E.g.: addi a0, x0, 4 replaces to: li a0, 4
    addi t1, x0, t2 replaces to: mv t1, t2

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, 0x00800      (20 bits)
ori t1, t1, 0x010     (12 bits)
```

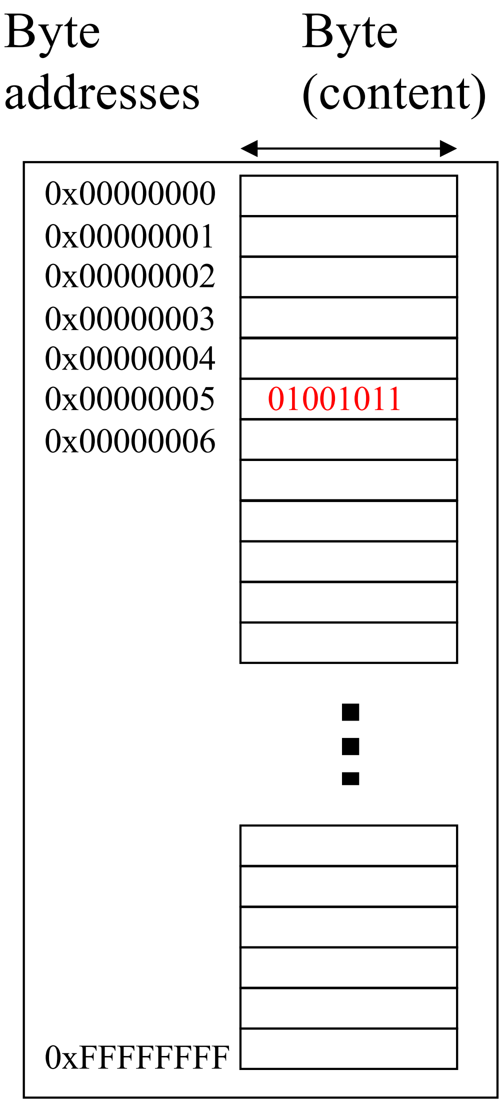
Memory model of a computer (32-bits)



The memory is addressed by bytes:

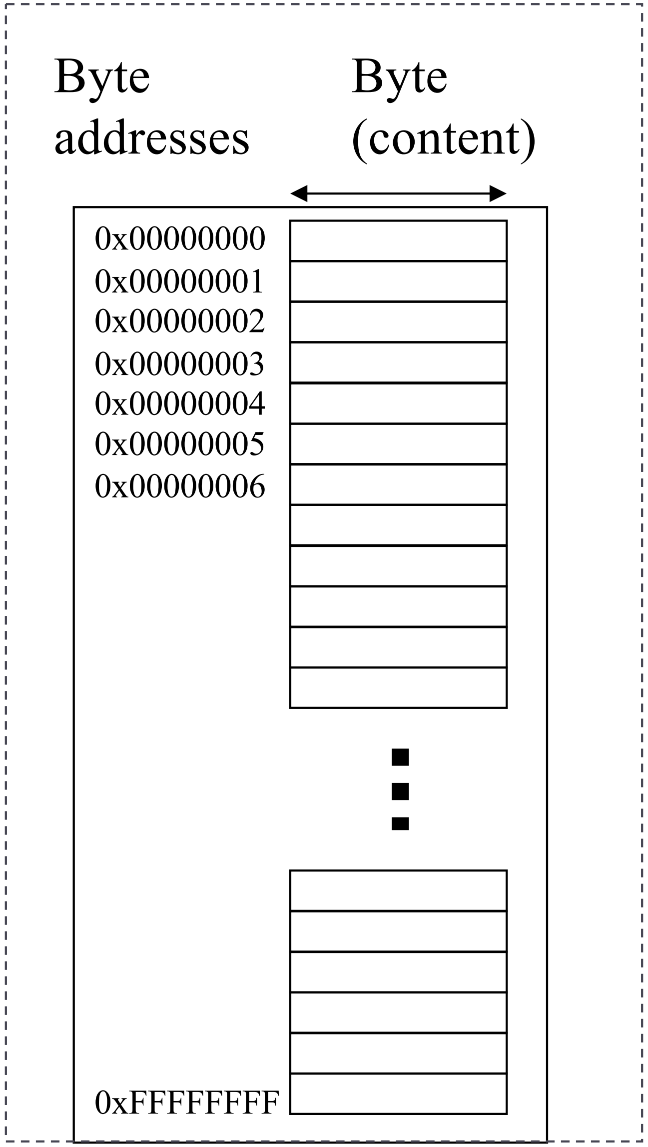
- Addresses of 32 bits
- Content of each address: one byte
- Address space: 2^{32} bytes = 4GiB

Memory model of a computer (32-bits)



Félix García Carballeira, Alejandro Calderón Mateos

RISC-V_{32} memory model



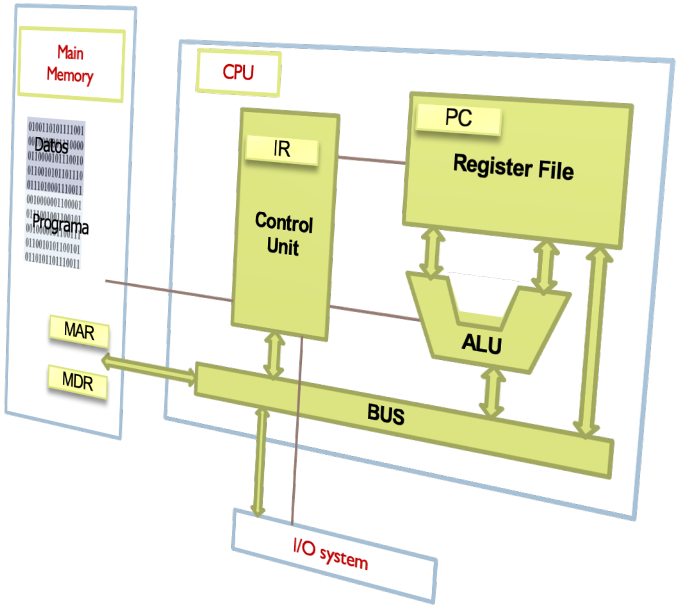
Memory is addressed by bytes:

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

Access can be to:

- Individual bytes
- Words (4 consecutive bytes)

Format of memory access instructions




```
lw
sw
lb
sb
lbu
```

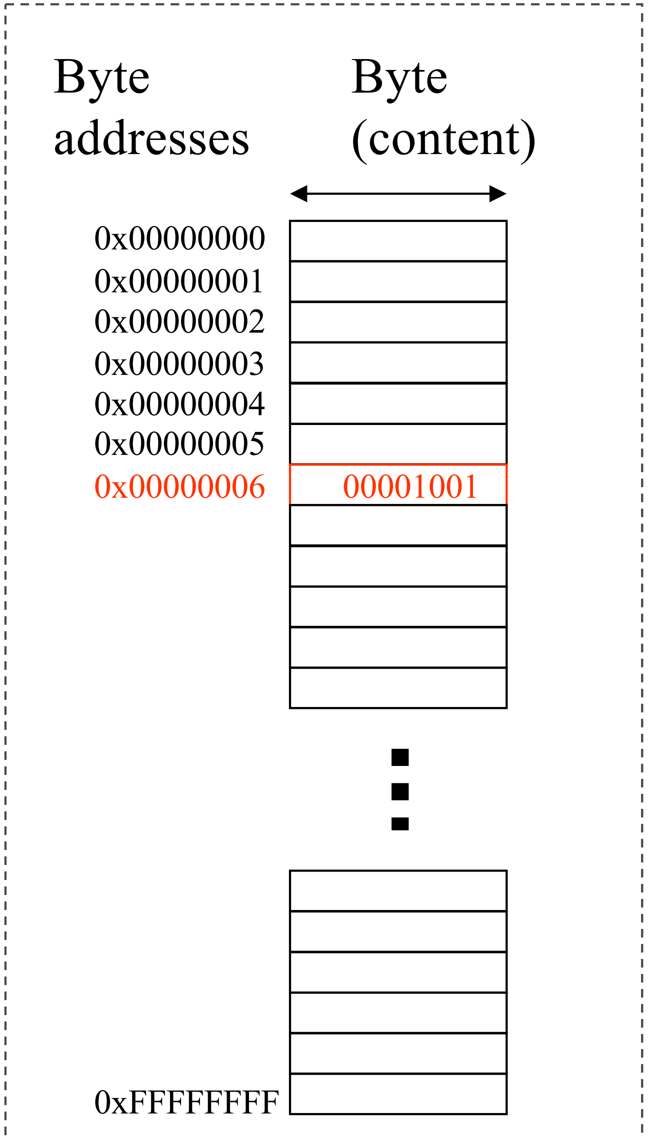
`Register, num(reg2)`

- num(register): represents the address obtained by summing num with the address stored in the register

Loading an address in a register

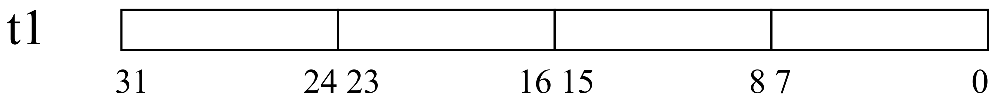
- Pseudoinstrucción `la`:
  - `la  rd,  addresss        rd ← address`

Access to bytes with lb (load byte)

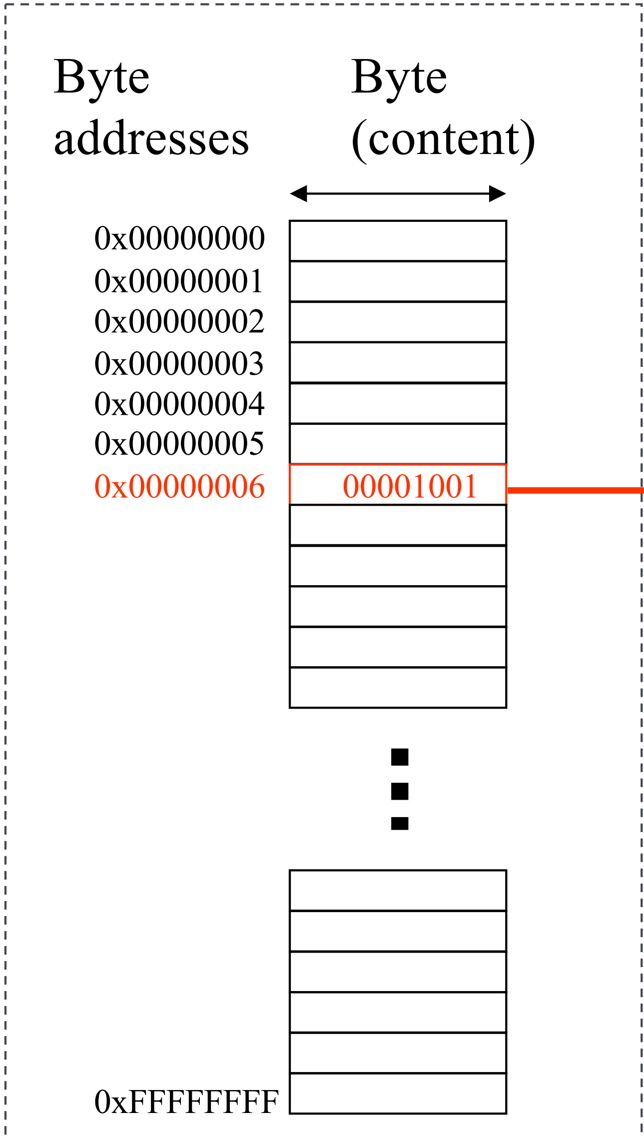



```
la t1, 0x6
lb t1, 0(t1)
```

Address: 0x000000006 (000110)
Content: 00001001 (9)

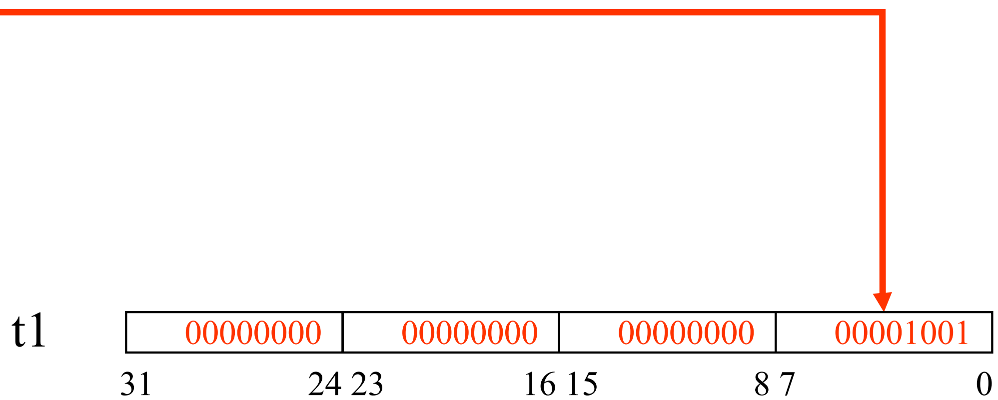


Access to bytes with lb

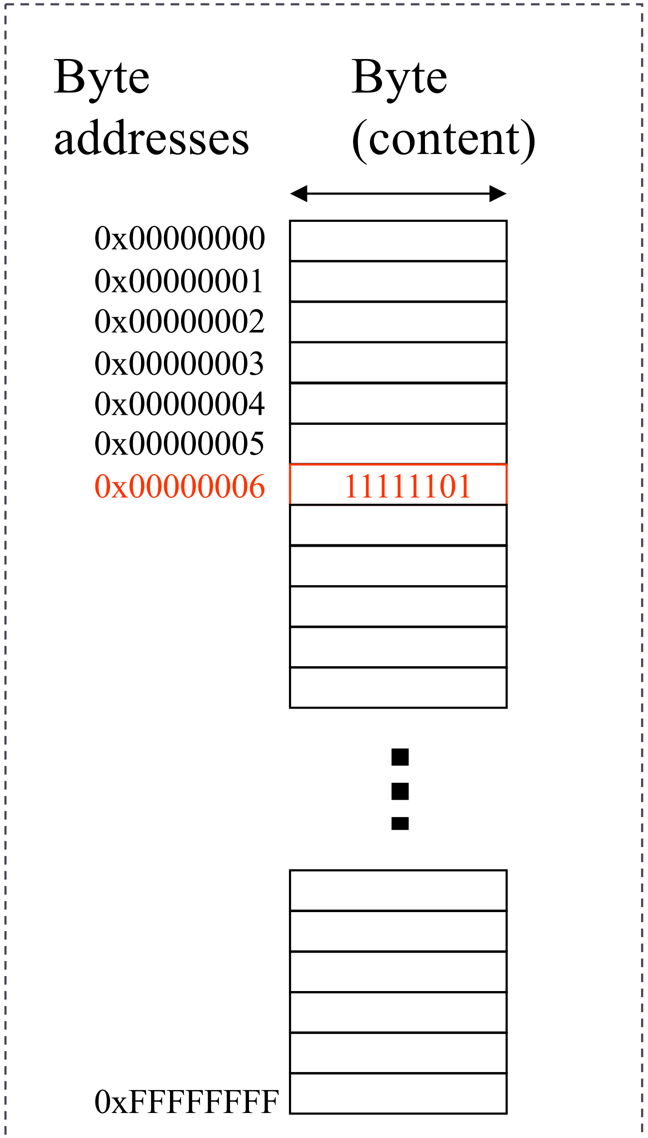



```
la t1, 0x6
lb t1, 0(t1)
```

Address: 0x000000006 (000110)
Content: 00001001 (9)

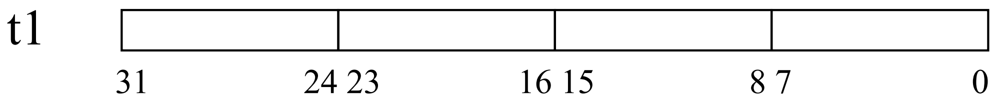


Access to bytes with lb

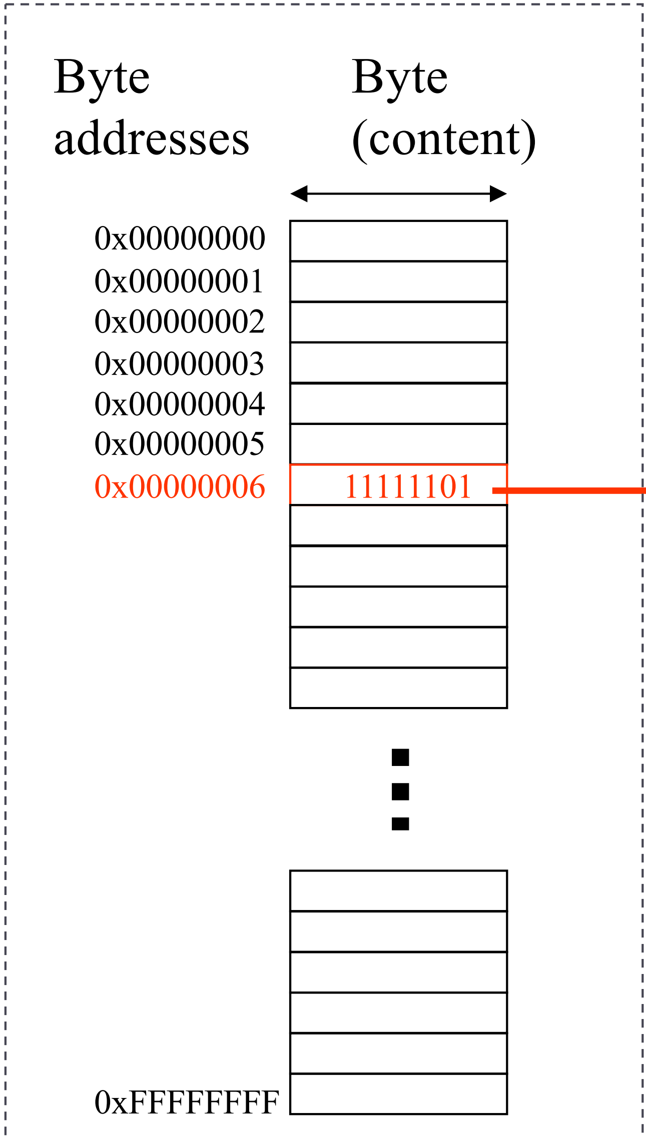



```
la t1, 0x6
lb t1, 0(t1)
```

Address: 0x000000006 (000110)
Content: 11111101 (-3 in two's complement)

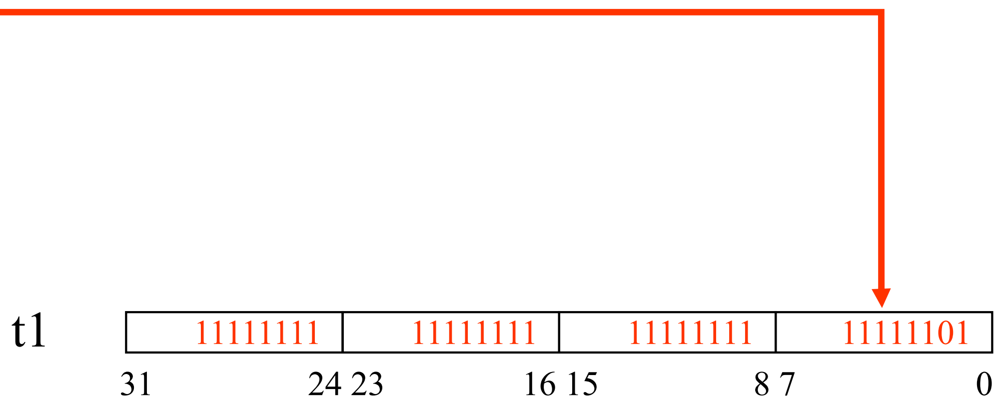


Access to bytes with lb

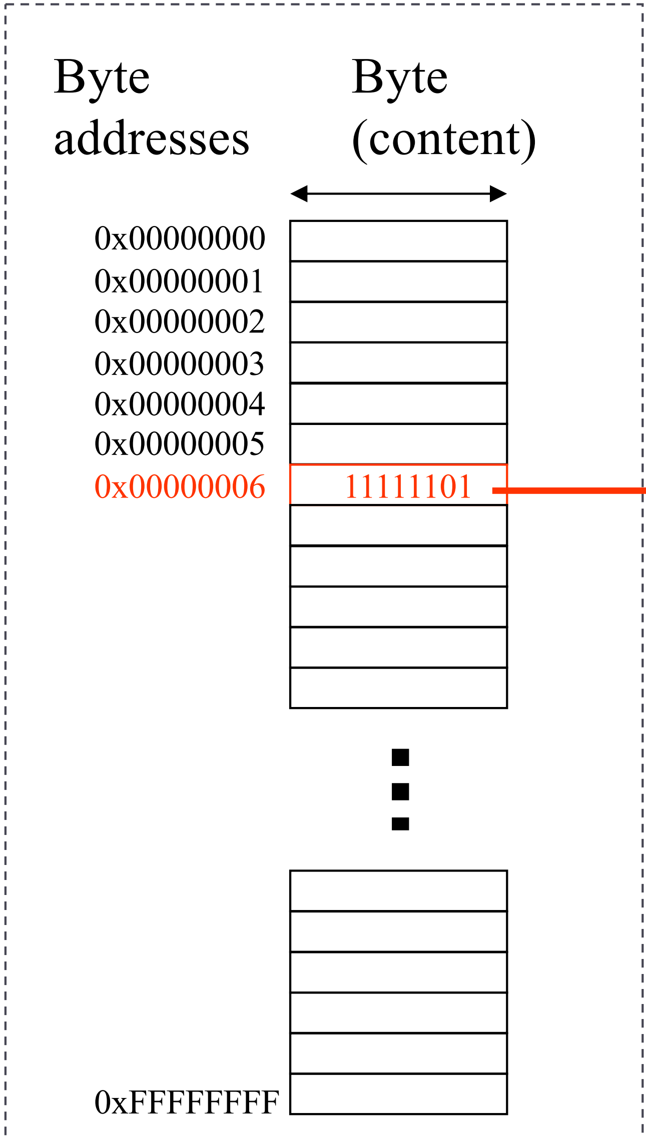



```
la t1, 0x6
lb t1, 0(t1)
```

Address: 0x000000006 (000110)
Content: 11111101 (-3 in two's complement)

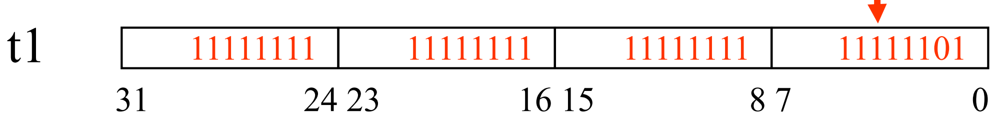


Access to bytes with lb




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

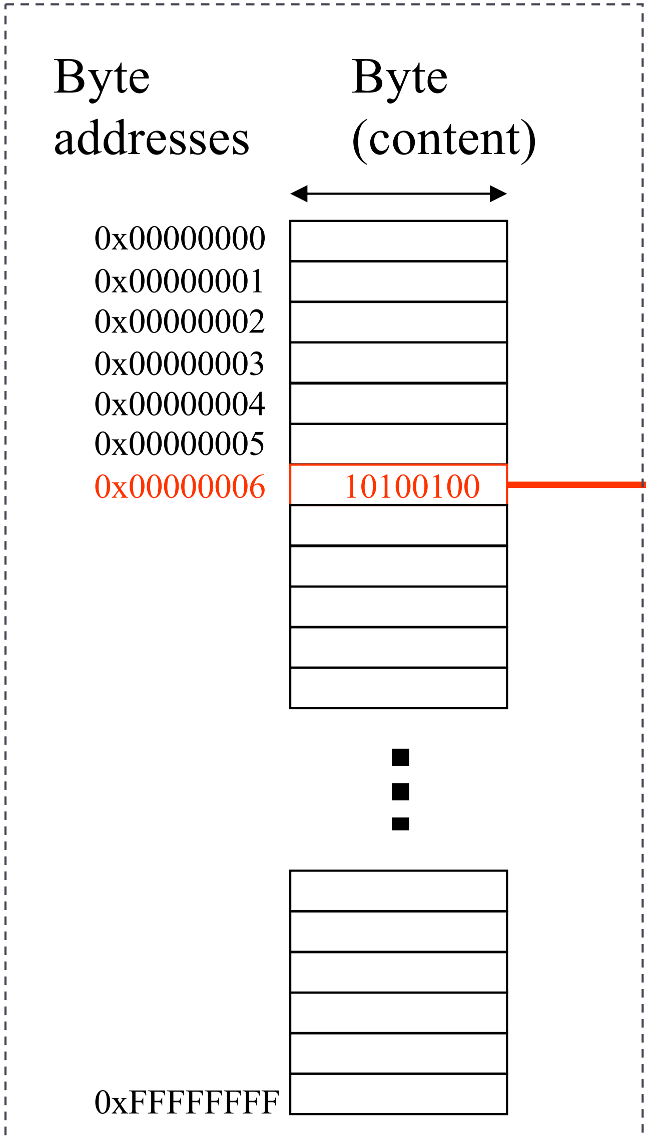
Address: 0x000000006 (000110)
Content: 11111101 (-3 in two's complement)



The "lb" instruction keep the sign
(sign extension)

Access to bytes with lb

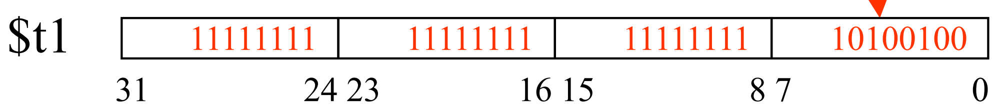
problems accessing characters




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

Address: 0x000000006 (000110)
Content: 10100100 (ASCII code for letter ñ is 164)

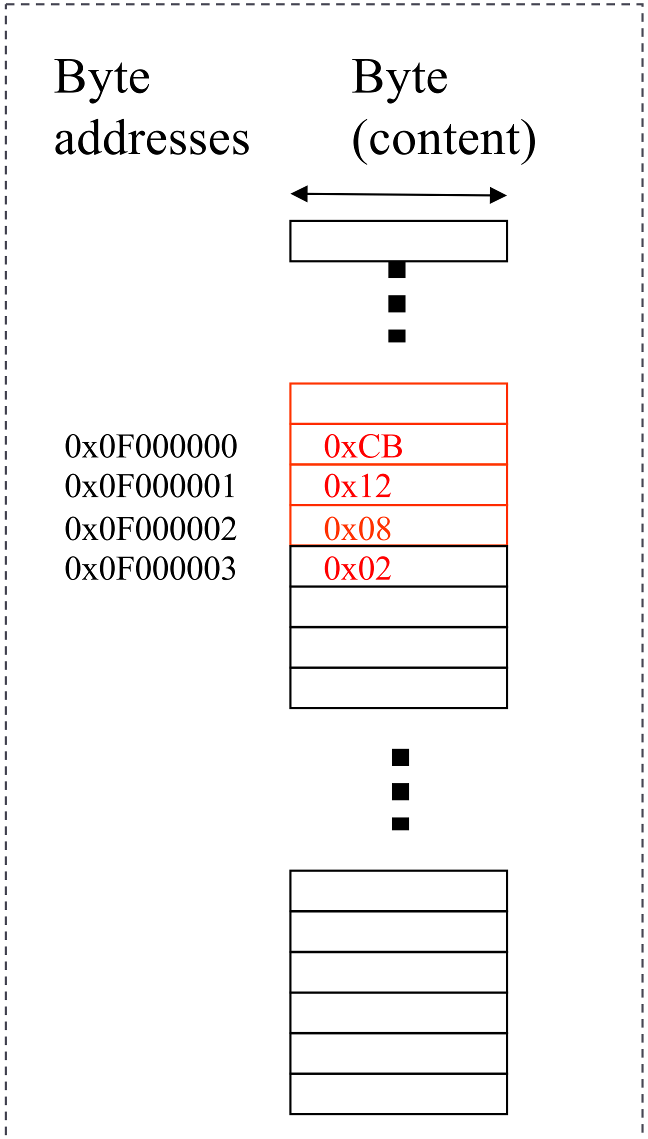
```
lb $t1, 0x6
```



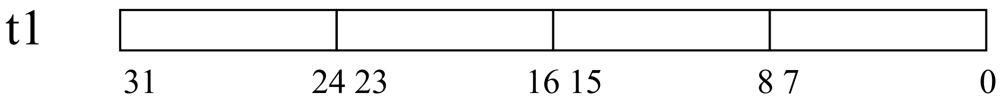
If lb is used (keeps the sign) and the content of t1 is not the 164 value (ASCII code for ñ)

ARCOS @ UC3M
Félix García Carballeira, Alejandro Calderón Mateos

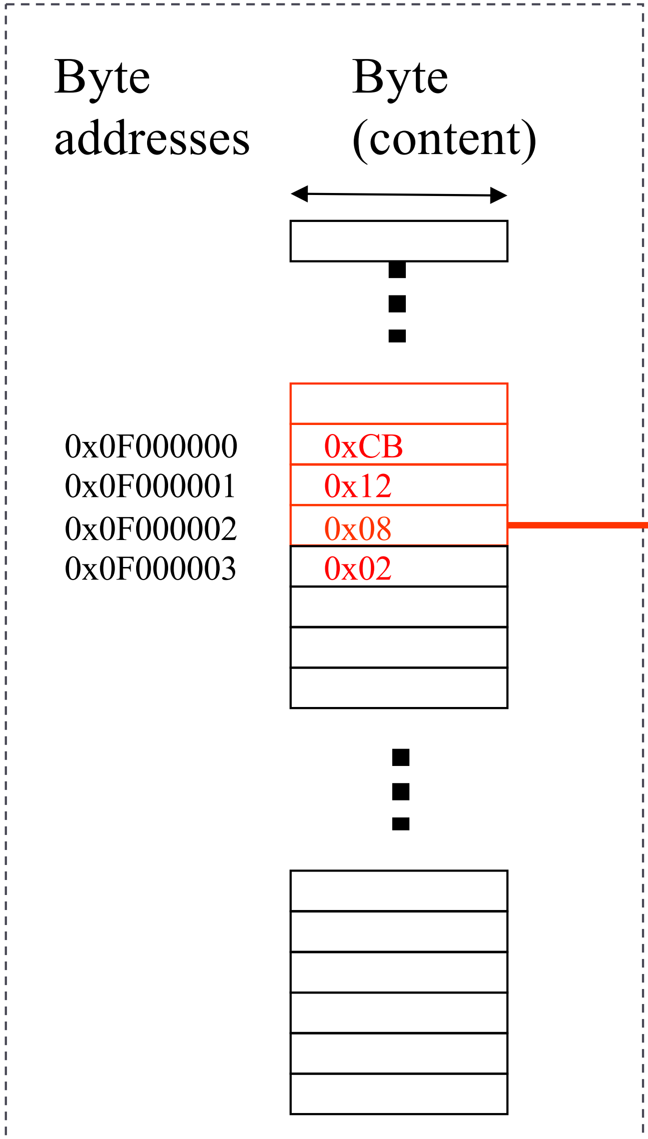
Access to bytes with lbu (load byte unsigned)




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



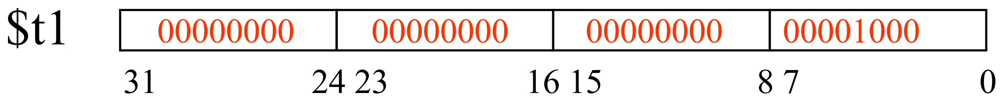
Access to bytes with lbu (load byte unsigned)



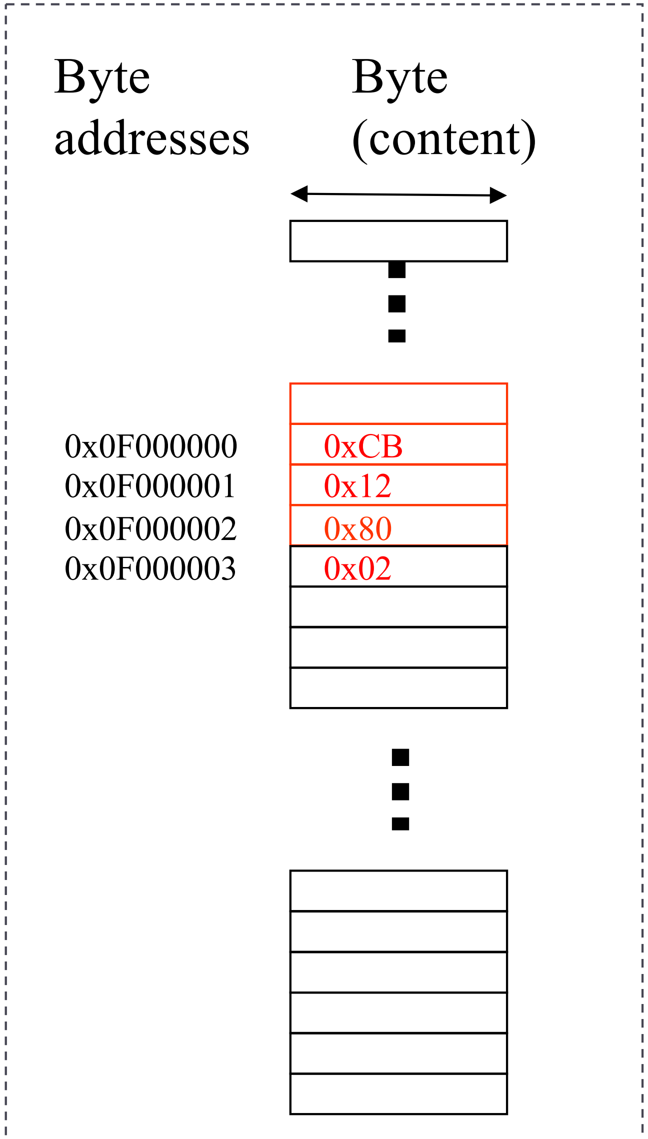

```
la   t1, 0x0F000002
lbu  t1, 0(t1)
```

The byte stored in memory location `0x0F000002` is copied.

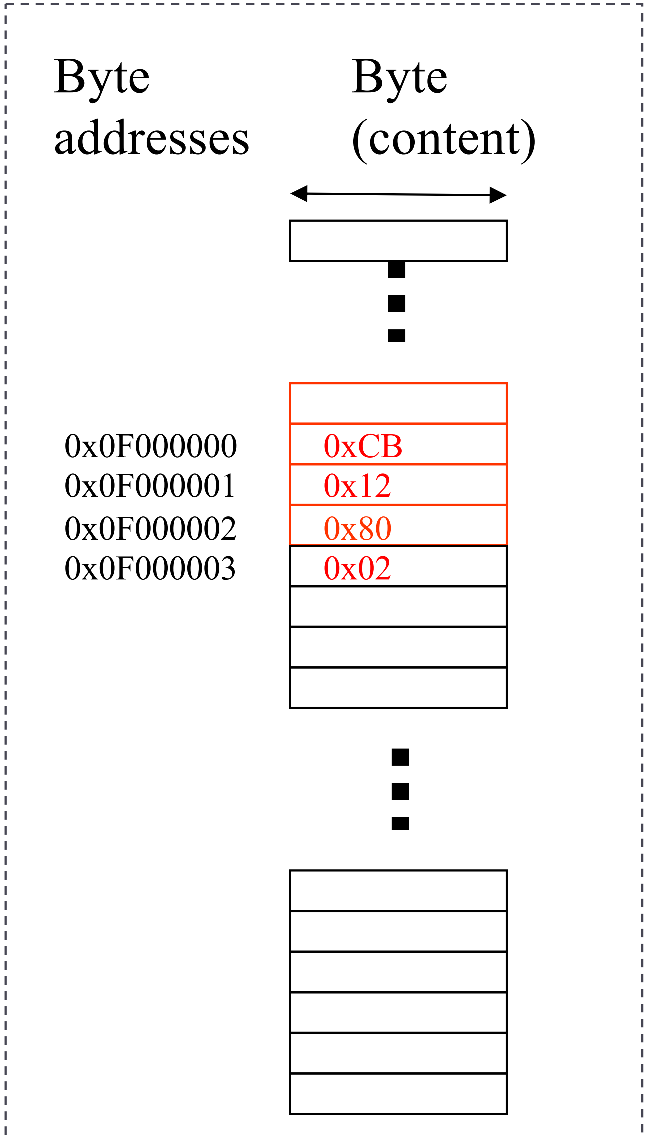
lbu does not preserv the sign (no sign-extensión)



Example of la and lbu
(load address and load byte unsigned)



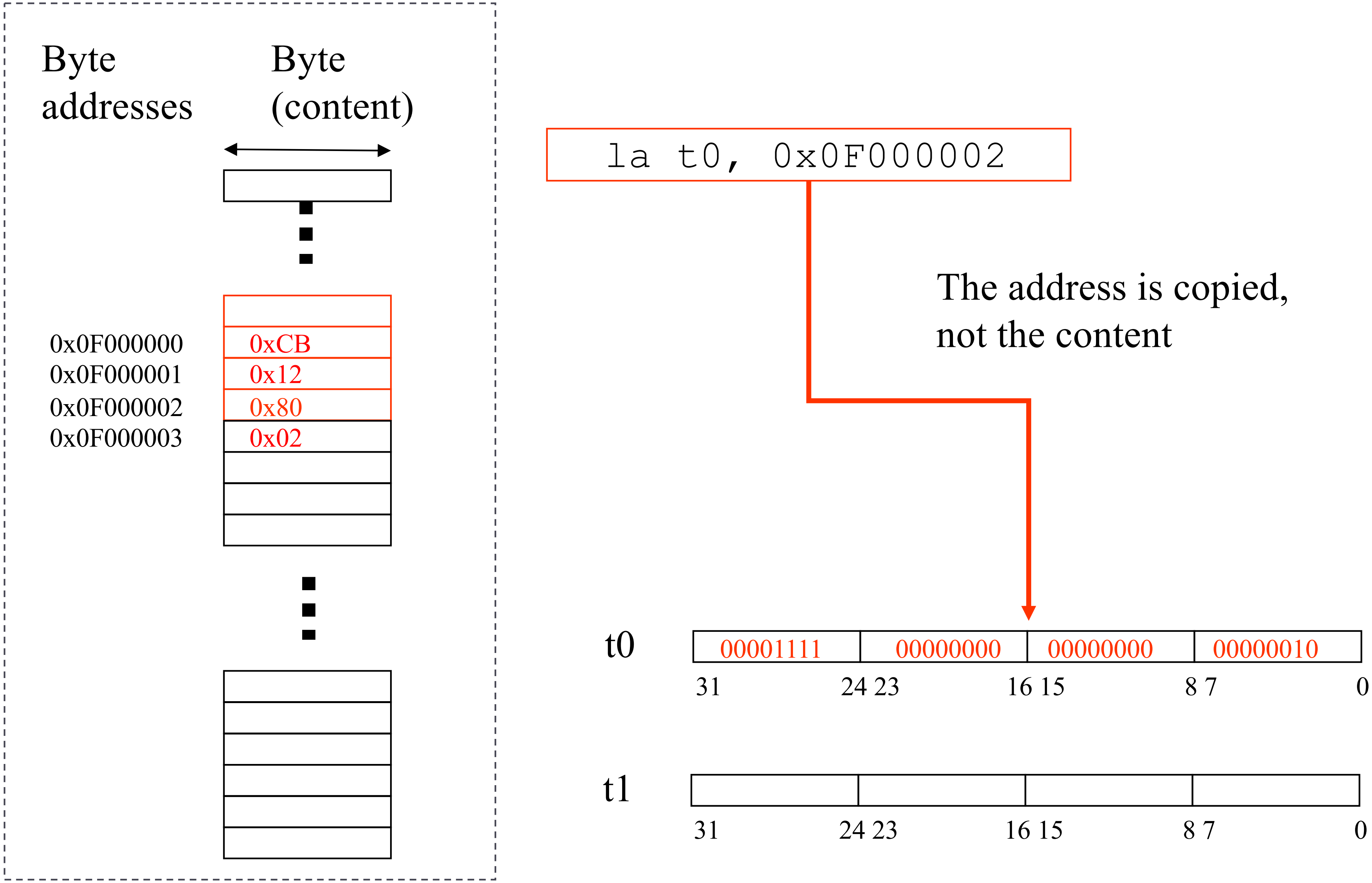
Example of la and lbu
(load address and load byte unsigned)



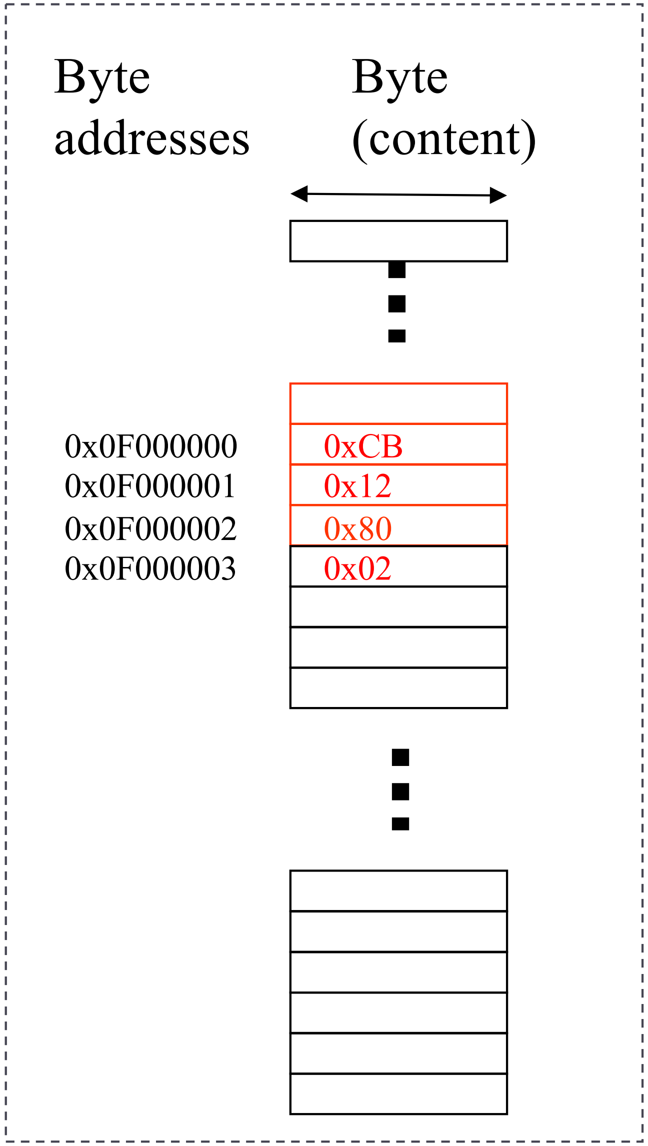

```
la t0, 0x0F000002
```

ARCOS @ UC3M
Félix García Carballeira, Alejandro Calderón Mateos

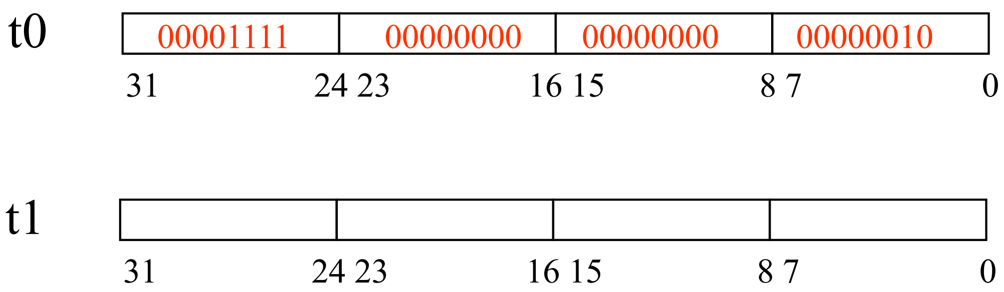
Example of la and lbu
(load address and load byte unsigned)



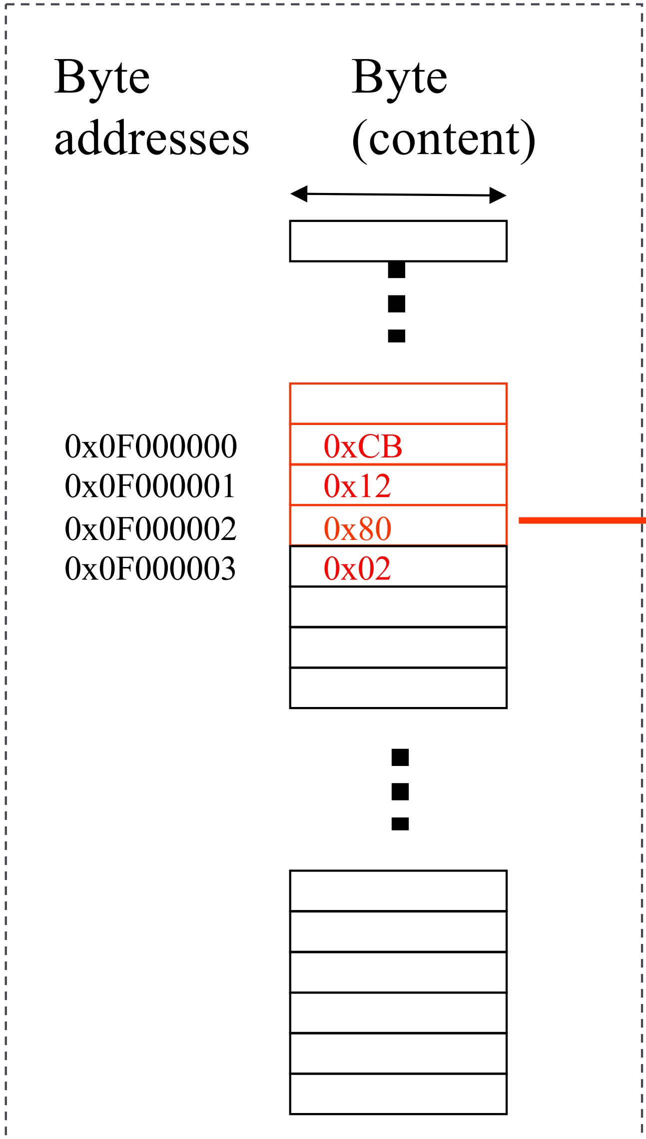
Example of la and lbu
(load address and load byte unsigned)




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

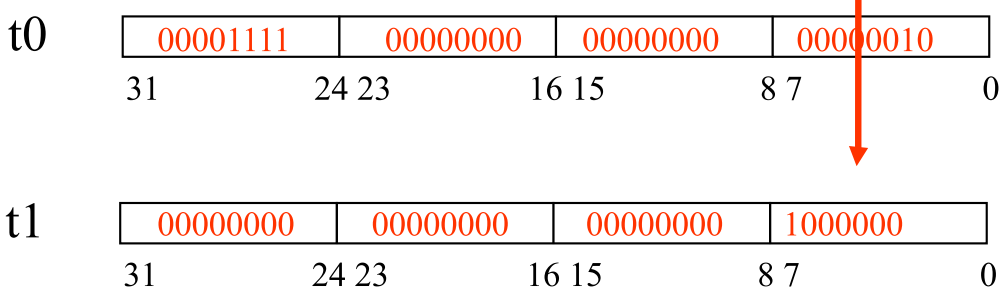


Example of la and lbu
(load address and load byte unsigned)

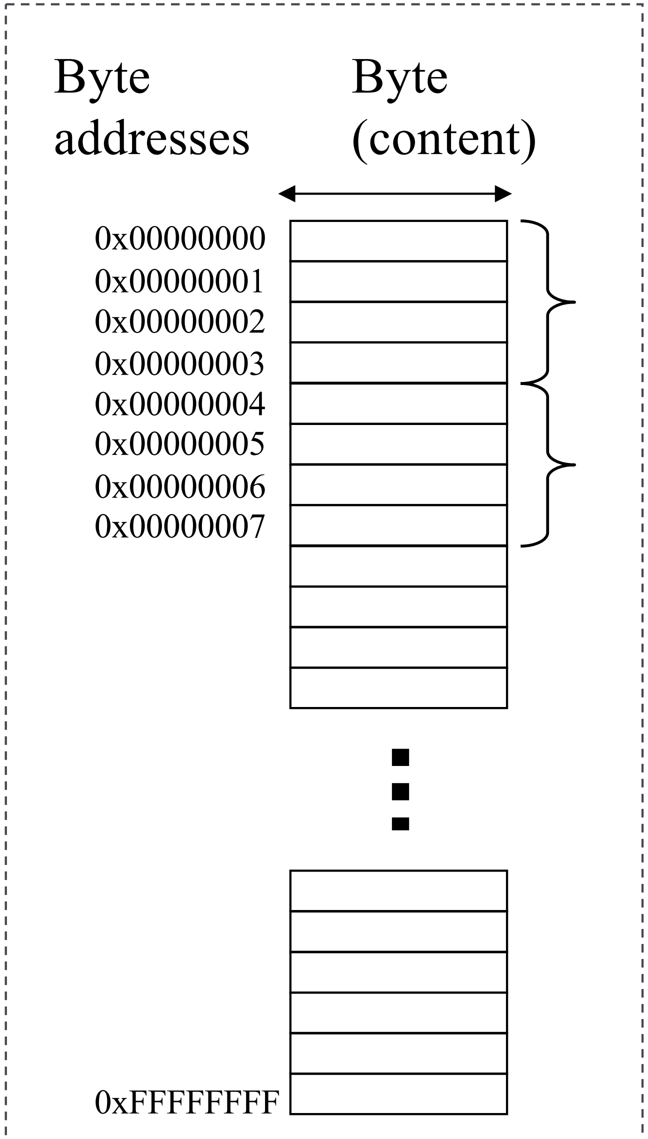



```
la    t0, 0x0F000002
lbu   t1, 0(t0)
```

Copy the byte stored in the memory address stored in register t0 into register t1 without sign extensión (lbu)



Accessing to words



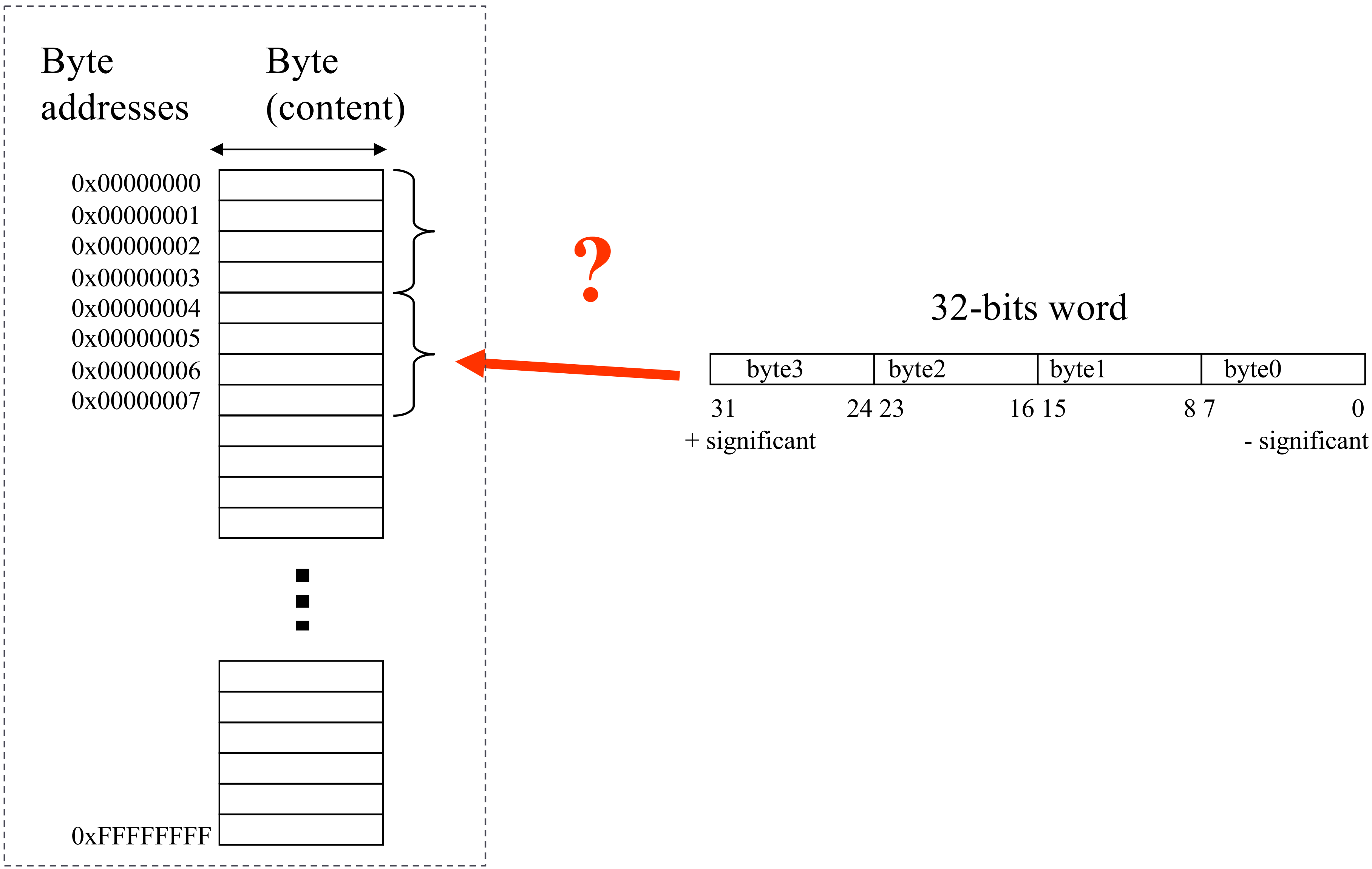
A word: 4 bytes in a 32-bits processor

Word stored starting at byte 0

Word stored starting at byte 4

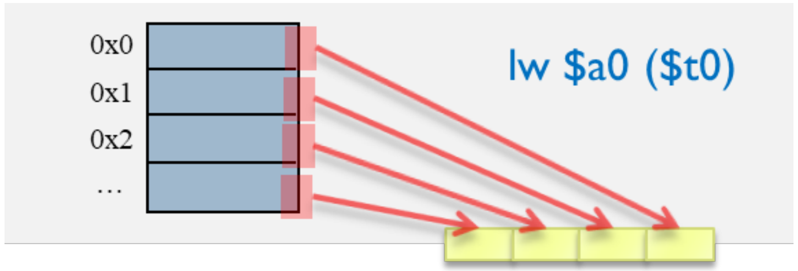
Words (32 bits, 4 bytes) are stored using
4 consecutive memory locations,
starting with the first position at an address
multiple of 4

Accessing to words

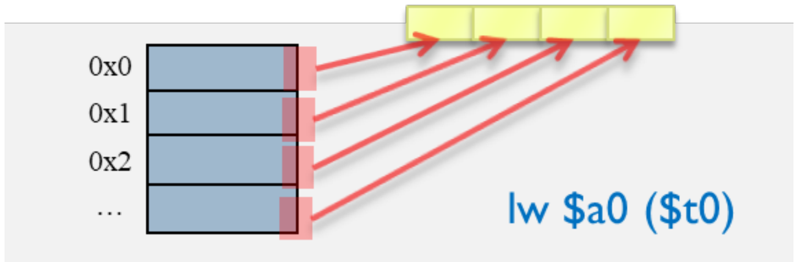


Data transfer
byte order

- There are 2 types of byte order:
  - Little-endian ('small' address ends the word…)



  - Big-endian ('big' address ends the word…)



(bi-endian)

Storing words in memory

32-bit word

byte3	byte2	byte1	byte0
31 24	23 16	15 8	7 0

+ MSB
(Most Significant Bit)

- LSB
(Least Significant Bit)

BigEndian

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

LittleEndian

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

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

BigEndian

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

LittleEndian

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

Communication problems in computers with different architectures

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

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

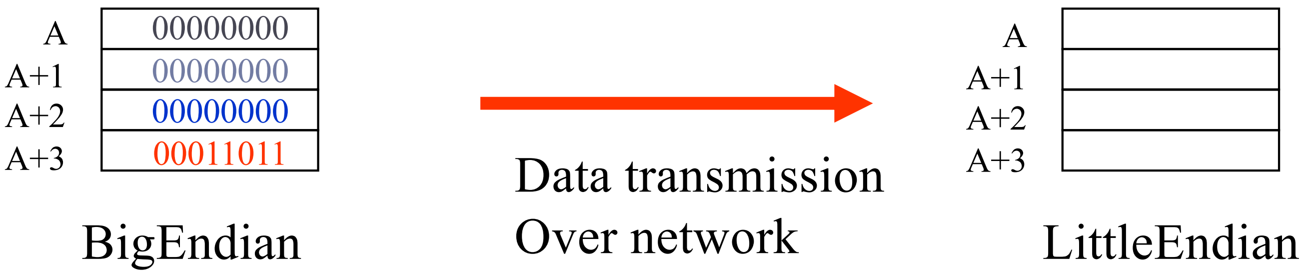
BigEndian

LittleEndian	
A	
A+1	
A+2	
A+3	

LittleEndian

Communication problems in computers with different architectures

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



Communication problems in computers with different architectures

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

BigEndian

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

LittleEndian

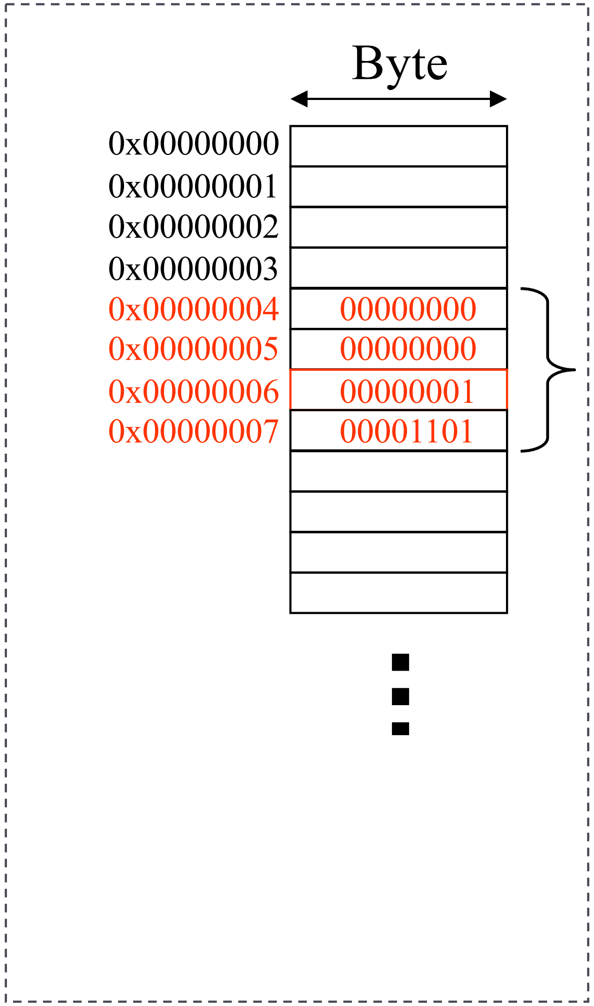
A	00000000
A+1	00000000
A+2	00000000
A+3	00011011

The stored number is: 00011011000000000000000000000000

And is not 27!

Read word from memory

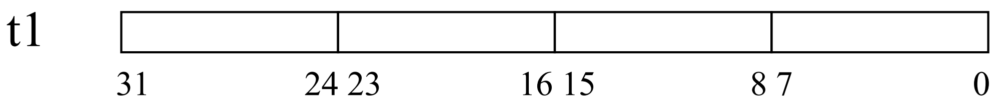
```
lw t1, 0x4(x0)
```



Address: 0x00000004 (000000…..00100)

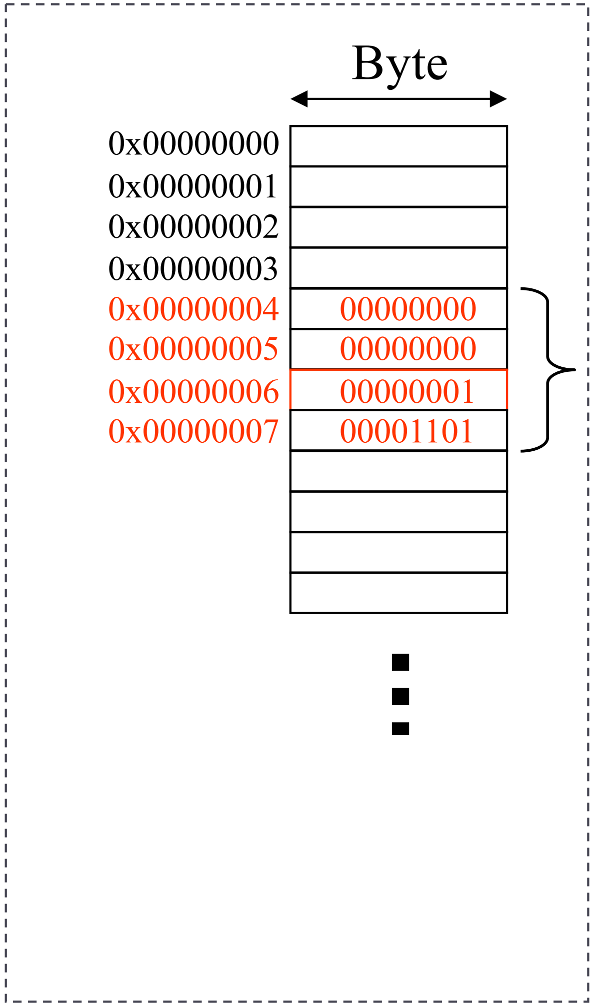
Content: $00000000000000000000000100001101_{(2} = 269_{(10}$

<u>Big Endian</u>



Read word from memory

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

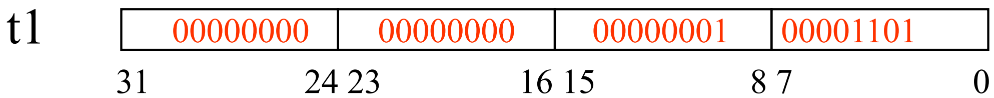


Address: 0x00000004 (000000…..00100)

Content: $00000000000000000000000100001101_{(2} = 269_{(10}$

Big Endian

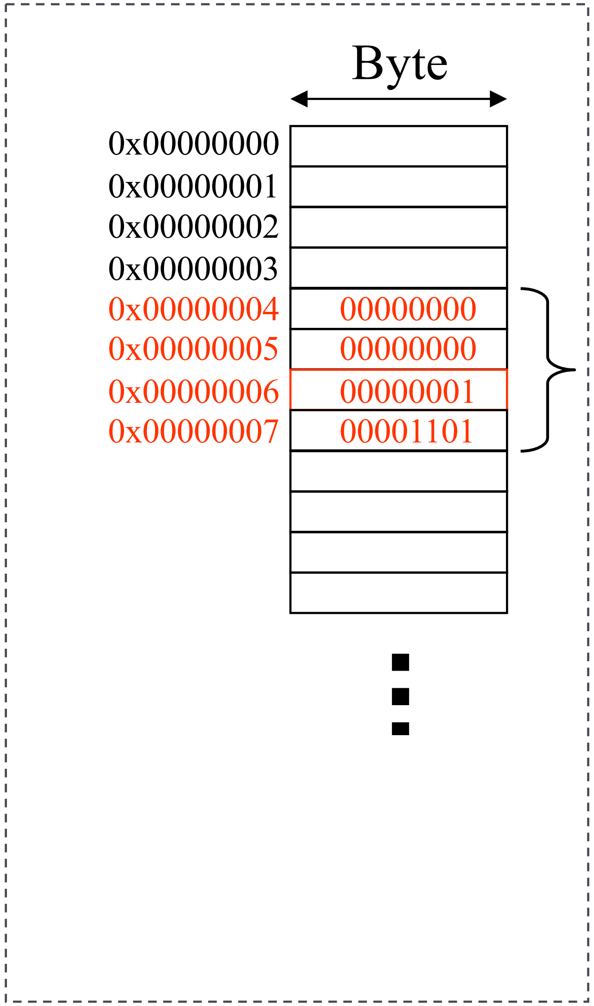
The word is copied from memory



ARCOS @ UC3M
Félix García Carballeira, Alejandro Calderón Mateos

Read word from memory

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



Address: 0x00000004 (000000…..00100)

Content: $00000000000000000000000100001101_{(2} = 269_{(10}$

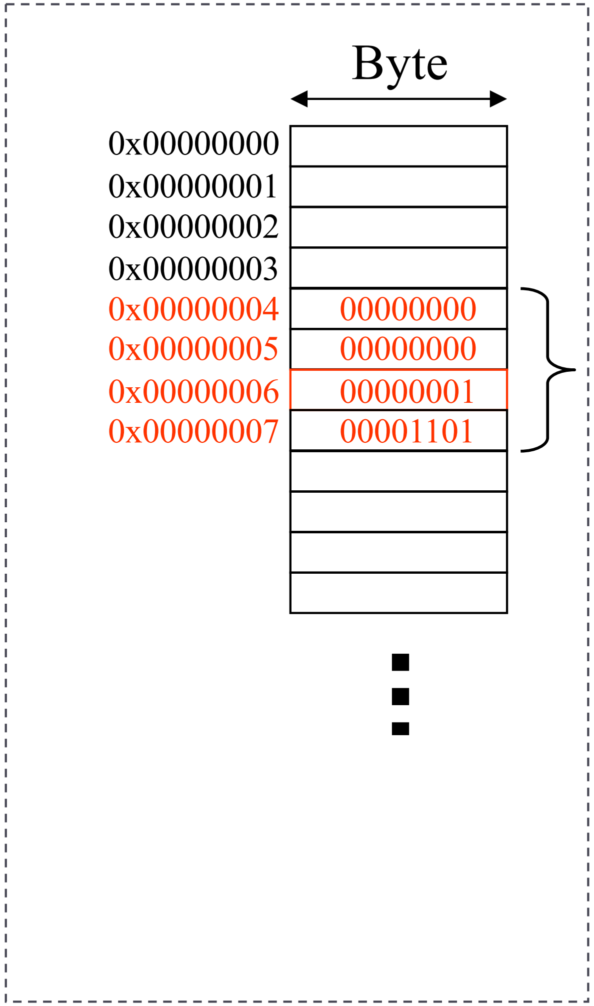
Little Endian

The word is copied from memory

t1

31 ... 24	23 ... 16	15 ... 8	7 ... 0
00001101	00000001	00000000	00000000

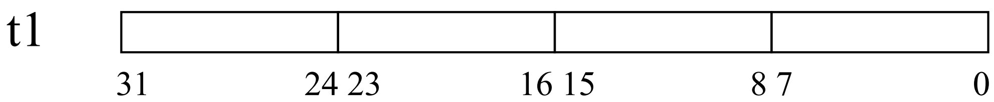
Read word from memory



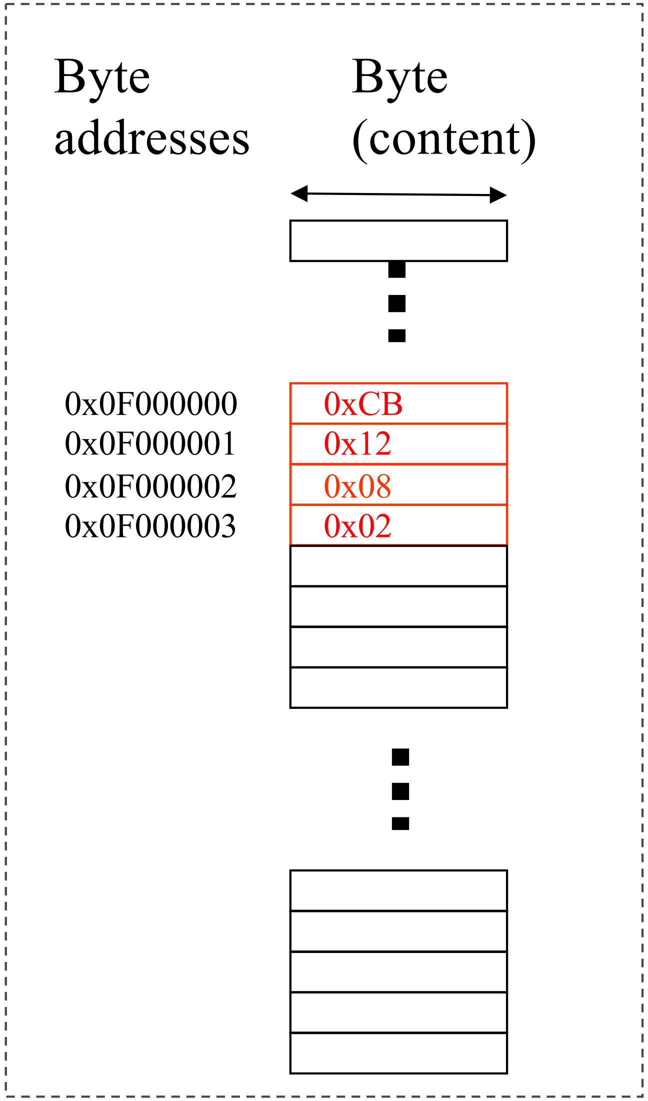

```
lw t1, 0x4(x0)
```

The address of the first byte is specified.

Access to the stored word starting from the address 0x00000004



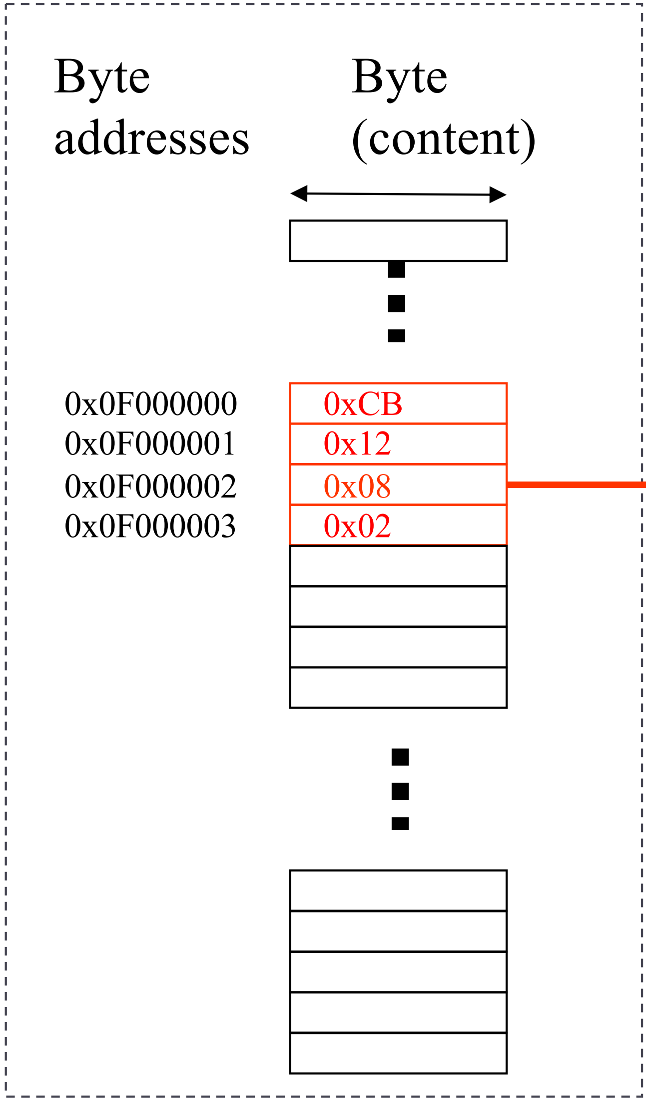
Differences among lw, lb, lbu, la




```
la   t1, 0x0F000000
lw   t1, 0(t1)
```

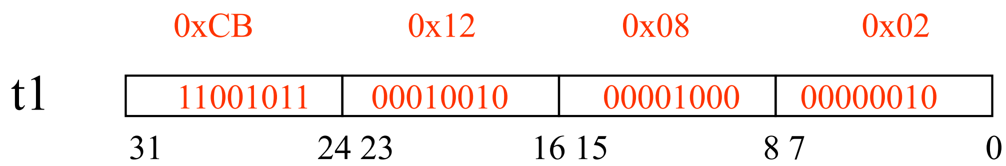
Differences among lw, lb, lbu, la

```
la   t1, 0x0F000000
lw   t1, 0(t1)
```



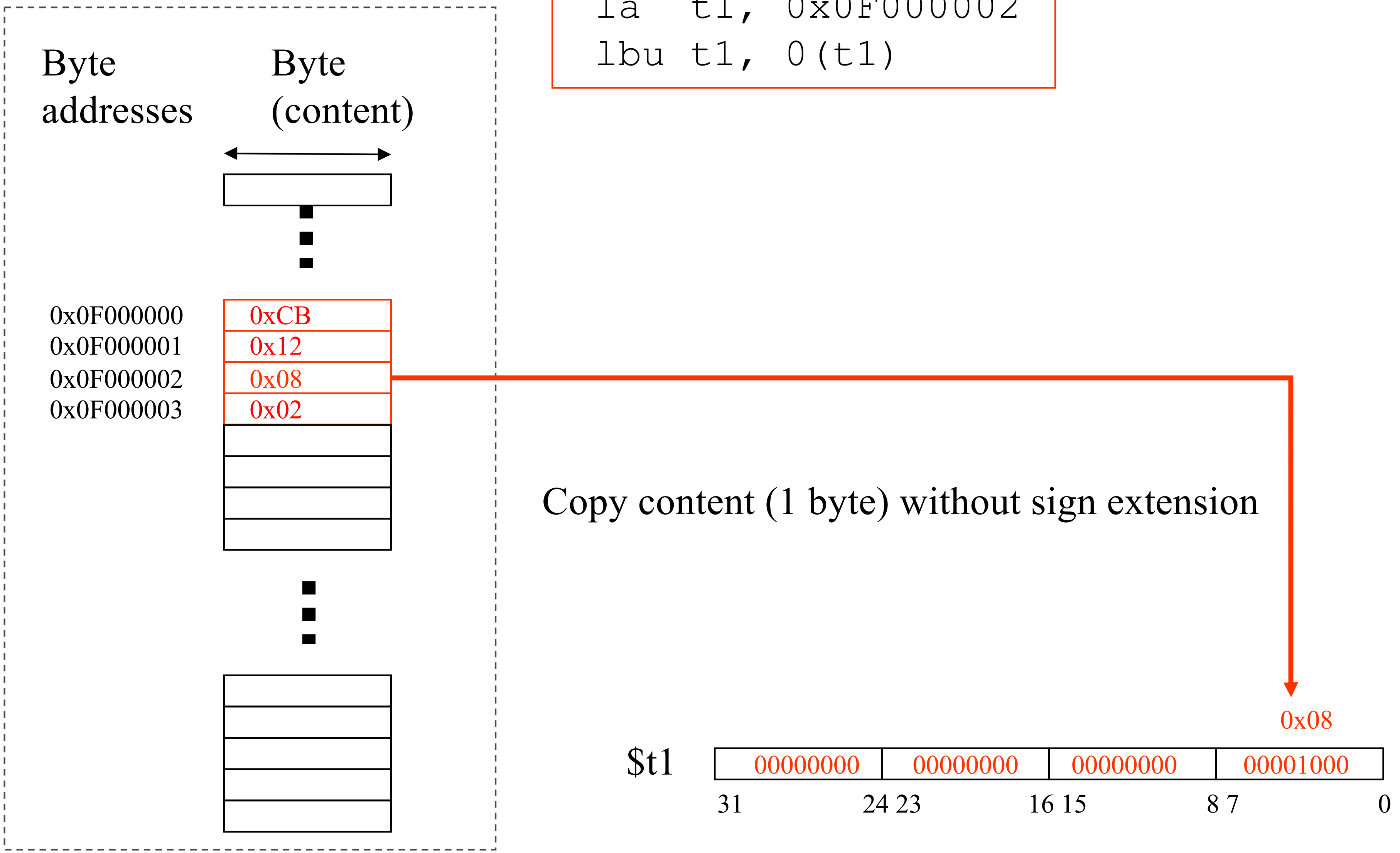
Copy the content of the word (the content at the address)

(Big Endian)

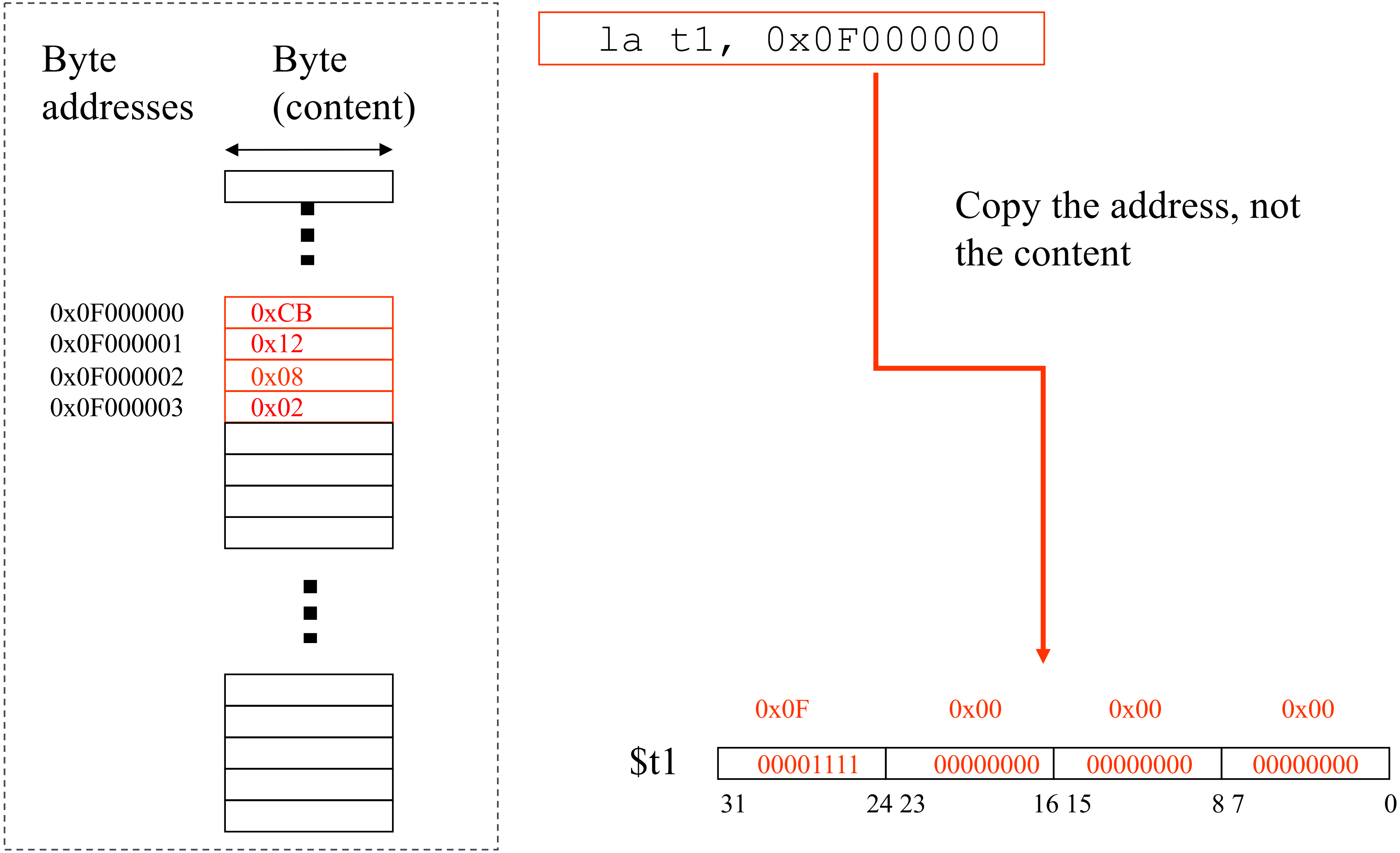


Differences among lw, lb, lbu, la

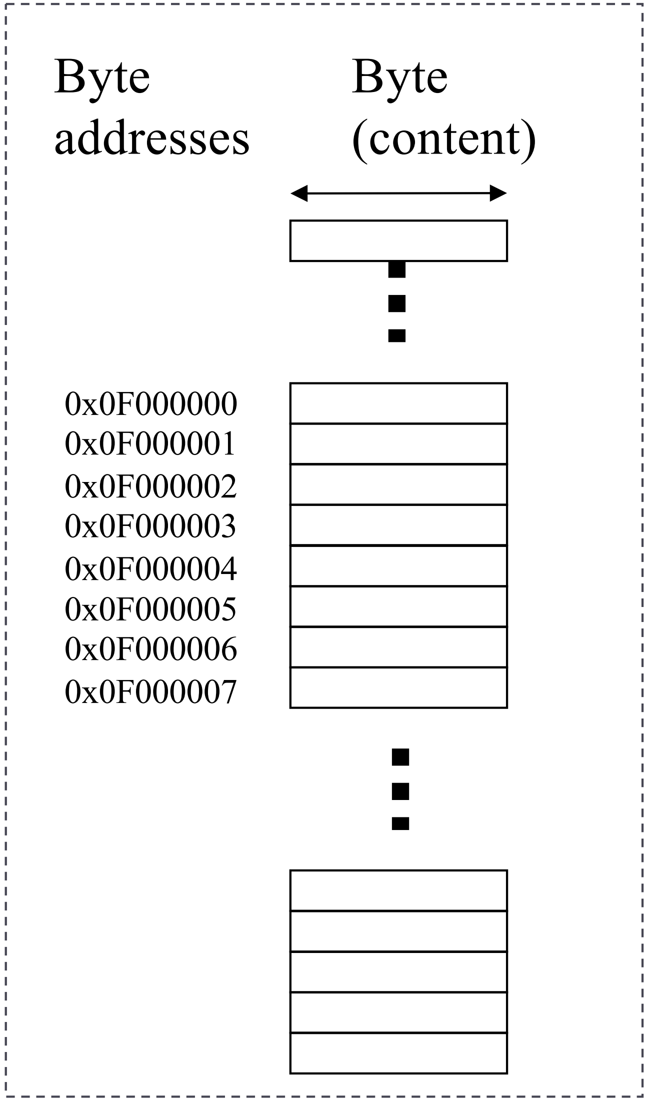
```
la  t1, 0x0F000002
lbu t1, 0(t1)
```



Differences among lw, lb, lbu, la

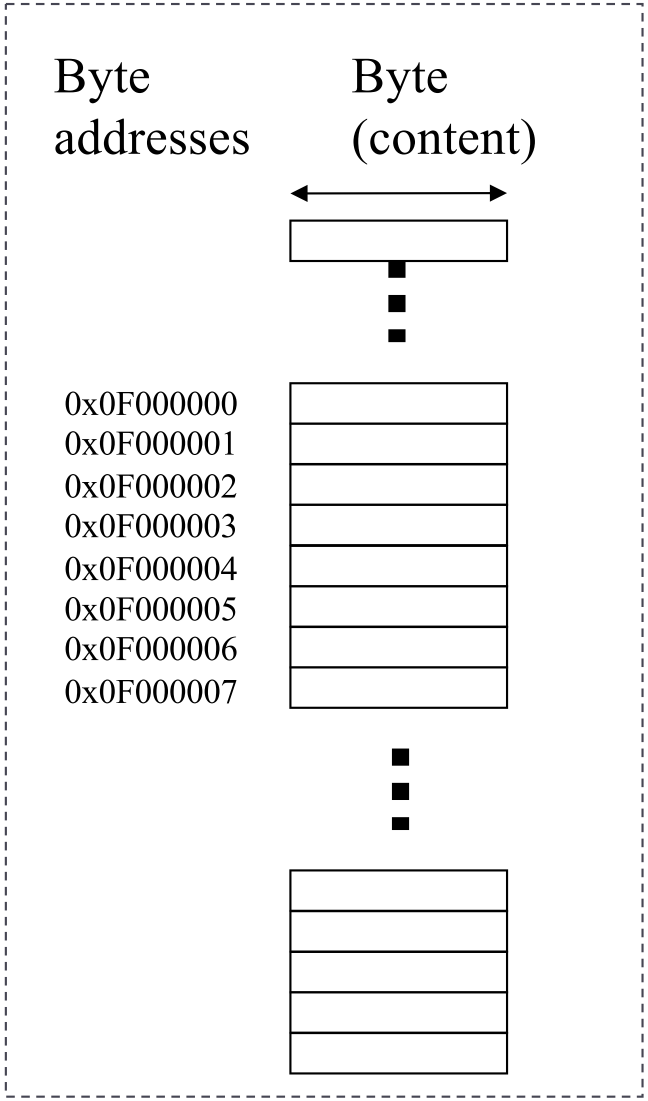


Example

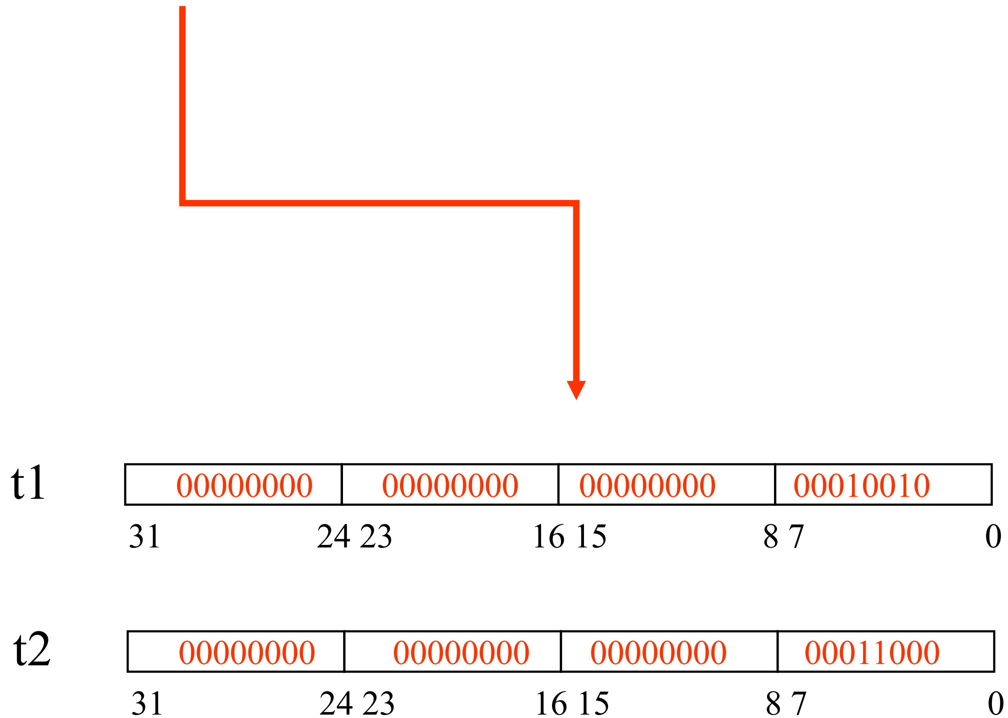



```
li t1, 18
li t2, 24
```

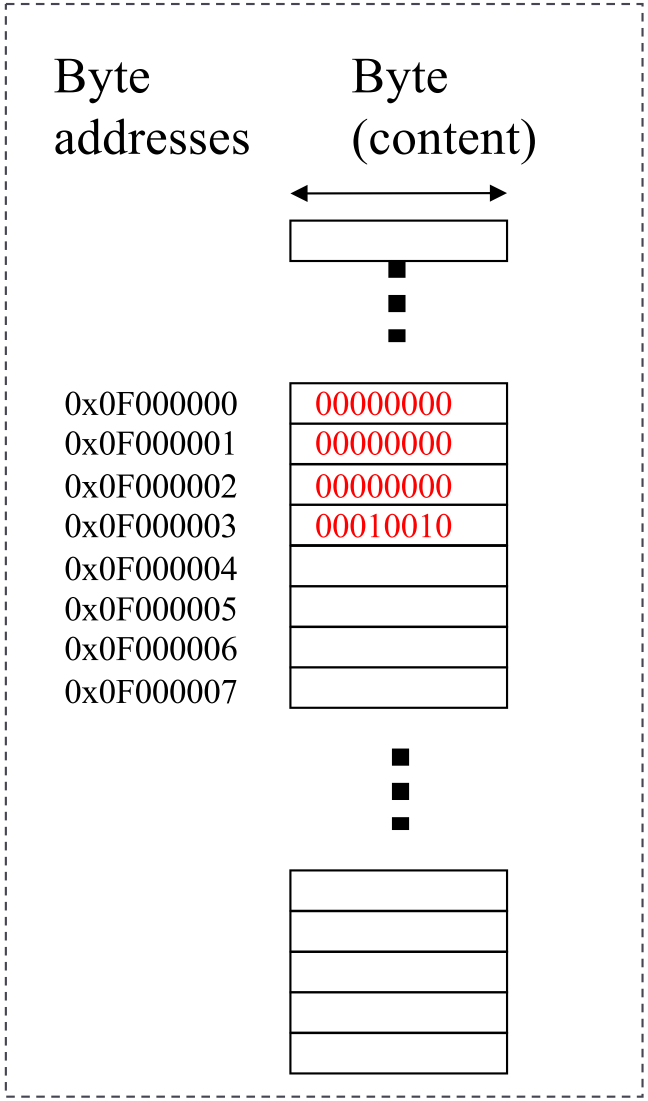
Example




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

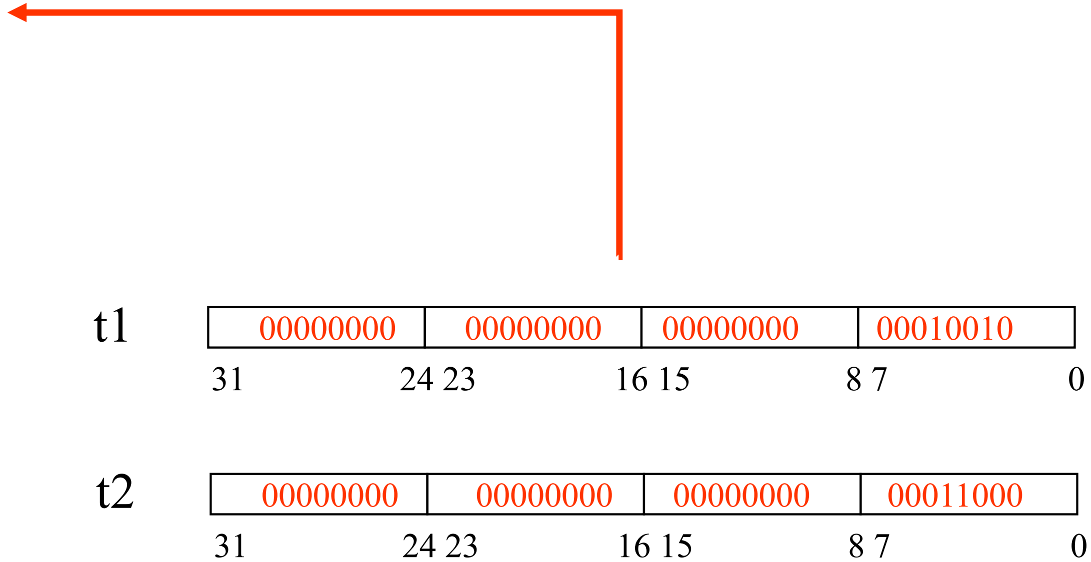


Write word in memory



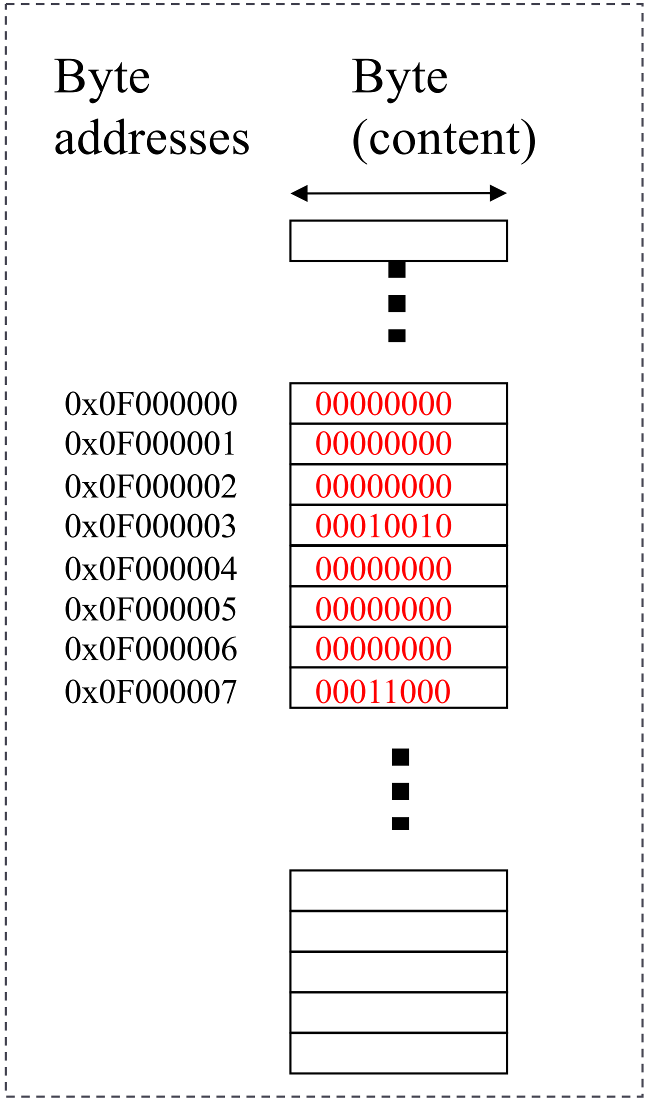

```
la t0, 0x0F000000
sw t1, 0(t0)
```

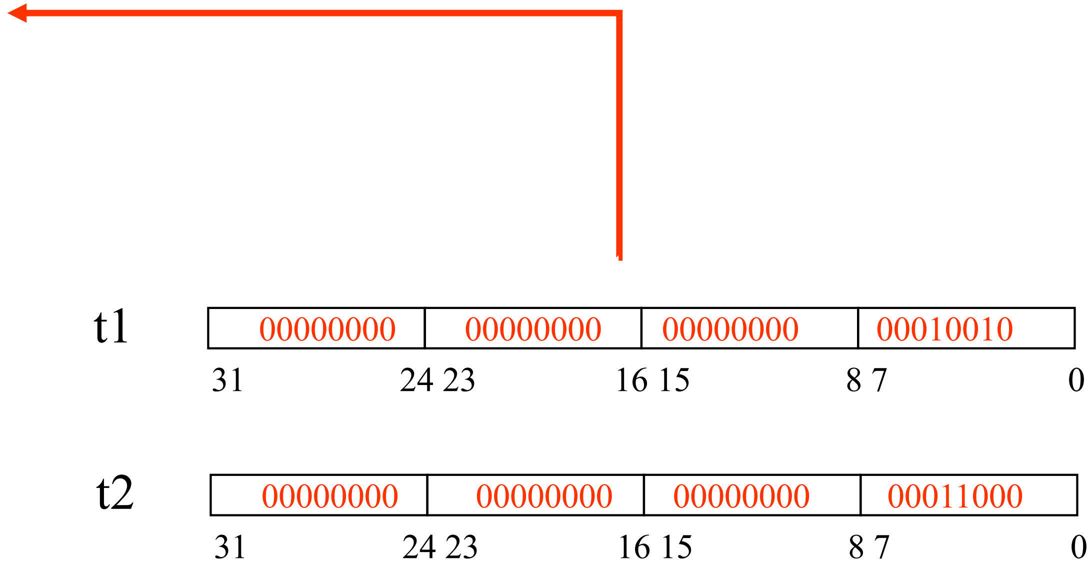
Write the content of a register into memory
(the full word value stored in the register)



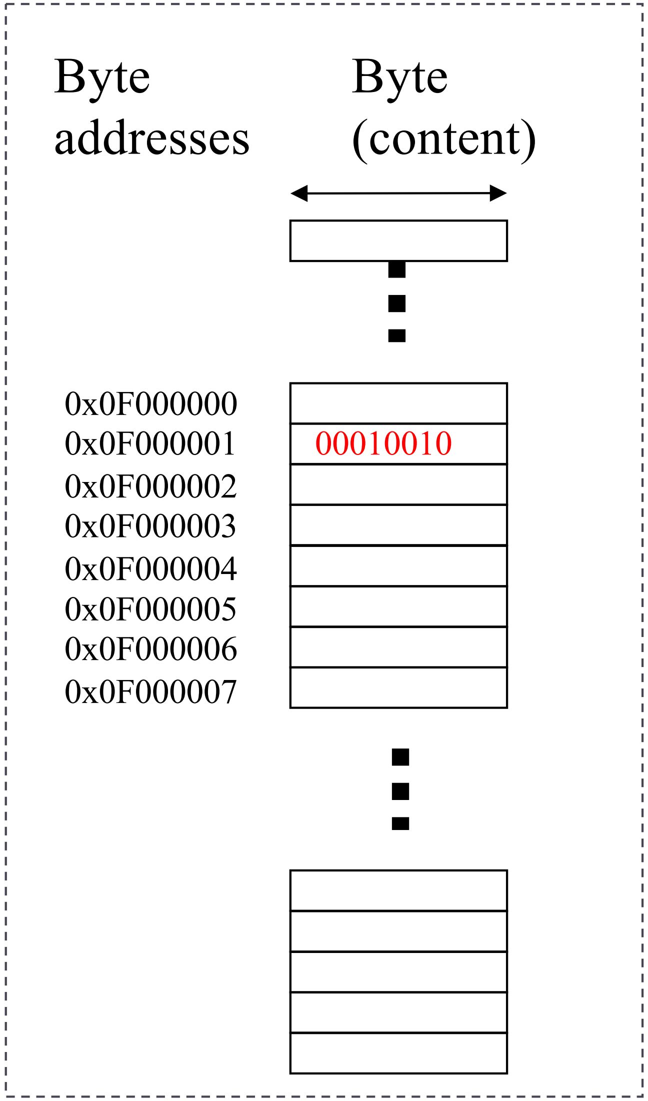
Write word in memory

```
la t0, 0x0F000000
sw t1, 0(t0)
la t0, 0x0F000004
sw t2, 0(t0)
```



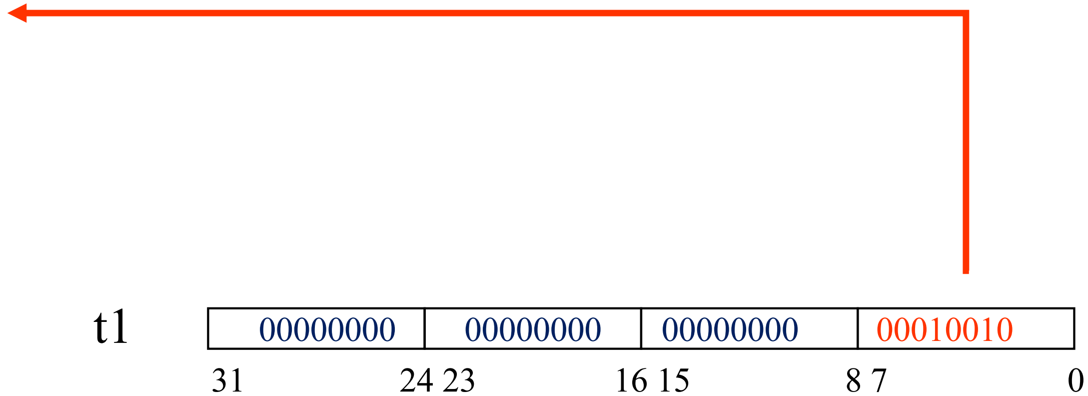


Write byte in memory



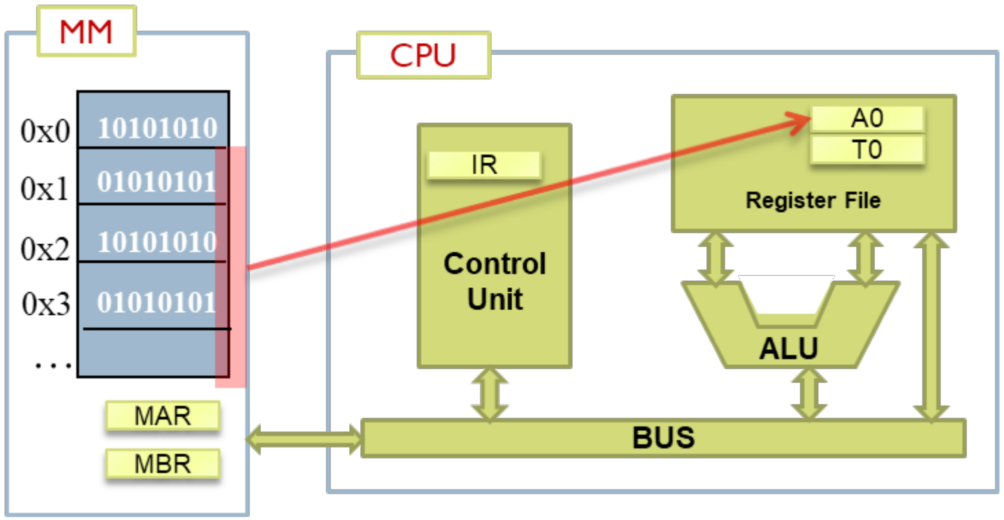

```
la t0, 0x0F000001
sb t1, 0(t0)
```

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:

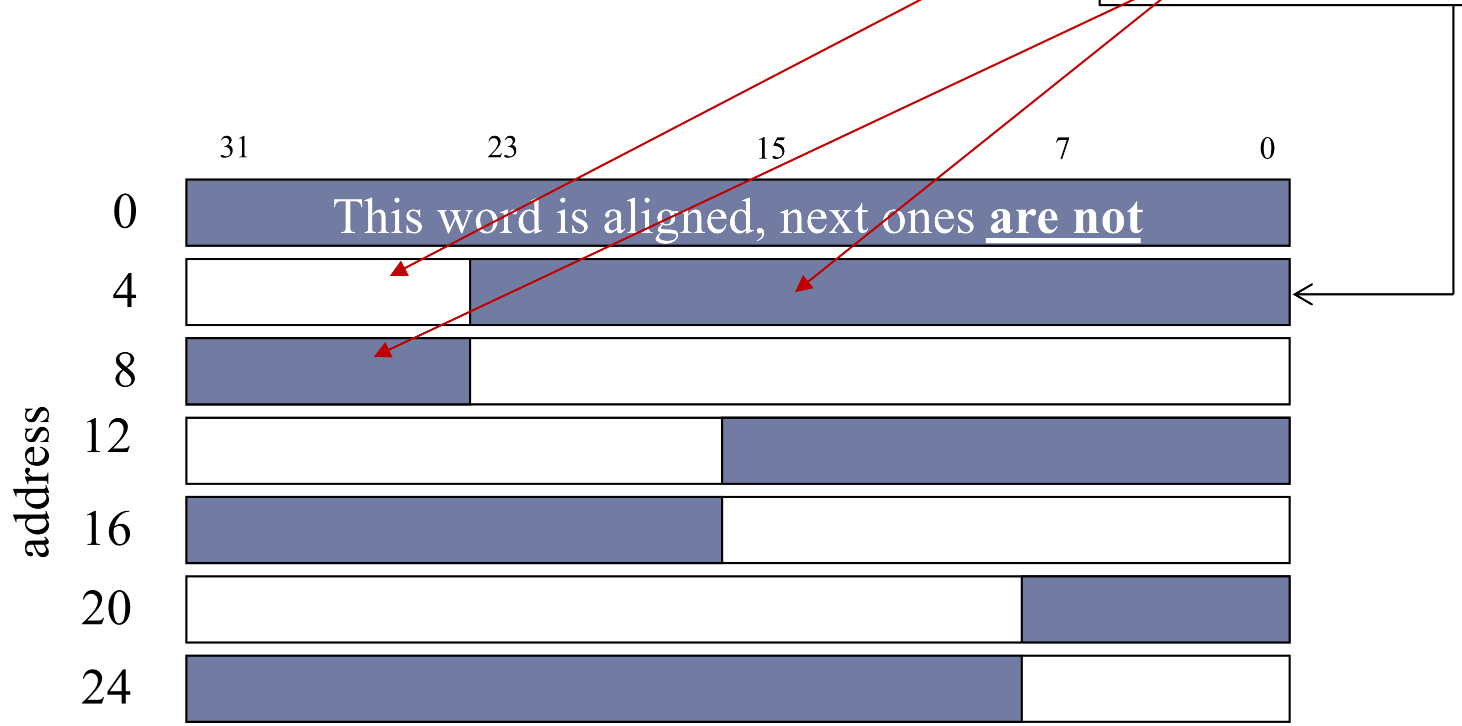
    D mod K = 0

- 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

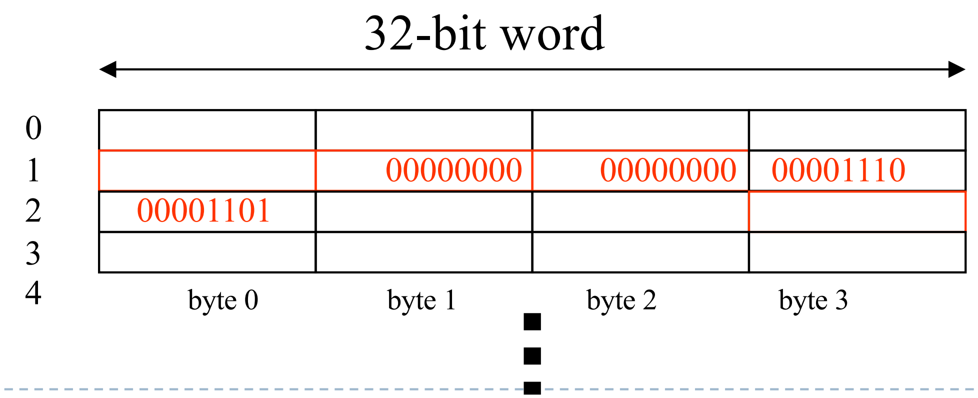
- The alignment requires the address to be a multiple of the word size:

```
.data
  b1:  .byte 0x0f
  w1.  .word 10
```



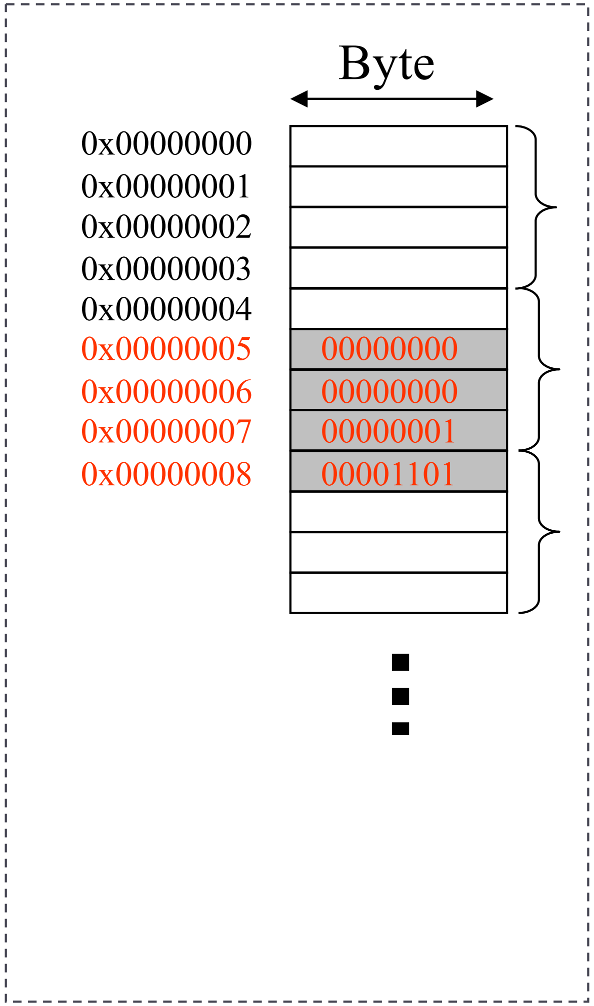
Data alignment

- 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(x0)` ????

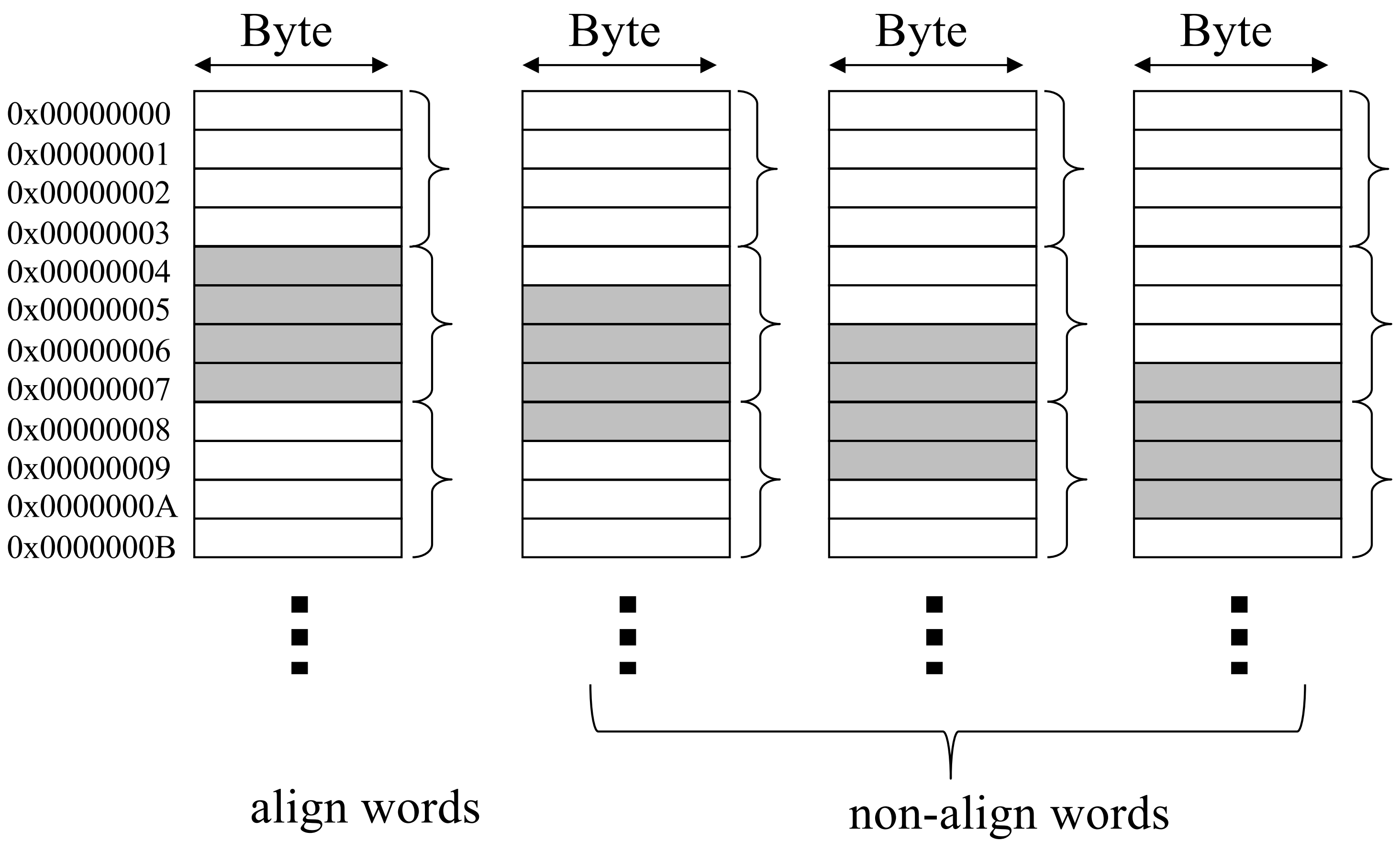


words

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

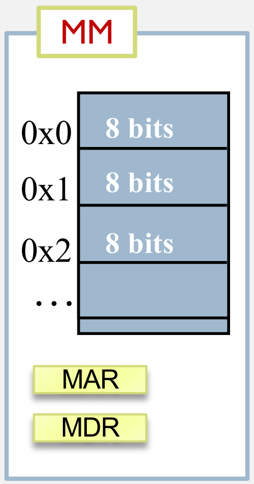
An aligned word must be stored starting from an address that is multiple of 4.

Non-aligned data



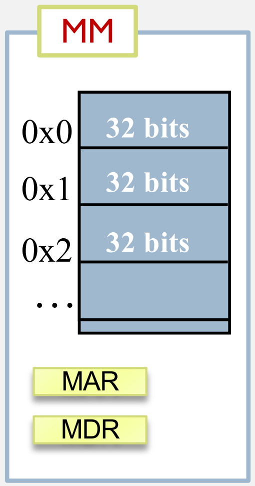
Word-level or byte-level addressing

- 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
- lb 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 RISC-V_{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, num(Reg2)`

- +num(register): represent the address that is obtained by adding num with the value stored in the register

Formatos de las instrucciones de acceso a memoria
resumen

- `la t0, 0x0F000002`
  - Immediate addressing. The memory address `0x0F000002` is loaded into t0
- `lbu t0, label(x0)`
  - Relative (to index) addressing. The byte stored in the memory location stored in label is loaded in t0
- `lbu t0, 0(t1)`
  - Indirect register addressing. The byte stored in the memory location stored in t1 is loaded in t0
- `lb t0, 80(t1)`
  - Relative (to base) 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

- ```
  la   t0, 0x0F000000
  sw   t0, 0(t0)
  ```
  - Copy the word stored in t0 in the address `0x0F000000`

- ```
  la t0, 0x0F000000
  sb t0, 0(t0)
  ```
  - Copy the (least significant) byte stored in t0 in the address `0x0F000000`

Memory Access. Floating point

- `flw rd, 10(rs1)`
  - Copy the single precisión value stored in (rs1+10) in register rd
- `fsw rs, 10(rd1)`
  - Store the single precision value of reister rs in the address (rd1+10).
- `fld rd, 10(rs1)`
  - Copy the double
  - Copy the double precisión value stored in (rs1+10) in register rd
- `fsd rs, 10(rd1)`
  - Store the double precision value of reister rs in the address (rd1+10).

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

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

Basic data types
characters

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


main ()
{
   c1 = c2;
   ...
}
```

```
 .data
c1: .space  1      # 1 byte
c2: .byte   'a'
...


.text

 main:   la   t1  c2
         lbu  t1  0(t1)
         la   t0  c1
         sb   t1  0(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
 main:  la   t1 op1
        lw   t1 0(t1)
        la   t2 op2
        lw   t2 0(t2)
        add  t3 t1 t2
        la   t4 result
        sw   t3 0(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
 main:   la   t1 op1
         lw   t1 0(t1)
         la   t2 op2
         lw   t2 0(t2)
         add  t3 t1 t2
         la   t4 result
         sw   t3 0(t4)
         ...
```

Exercise

- Write in RISC-V_{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

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

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

```
 .data
.align 2
     result    :   .space 4  # 4 bytes
     op1:          .float 100
     op2:          .float 2.5
...

.text

 main:  flw     f0  op1(x0)
        flw     f1  op2(x0)
        fadd.s  f3  f1  f2
        fsw     f3  result(x0)
        ...
```

Basic data types
double

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


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

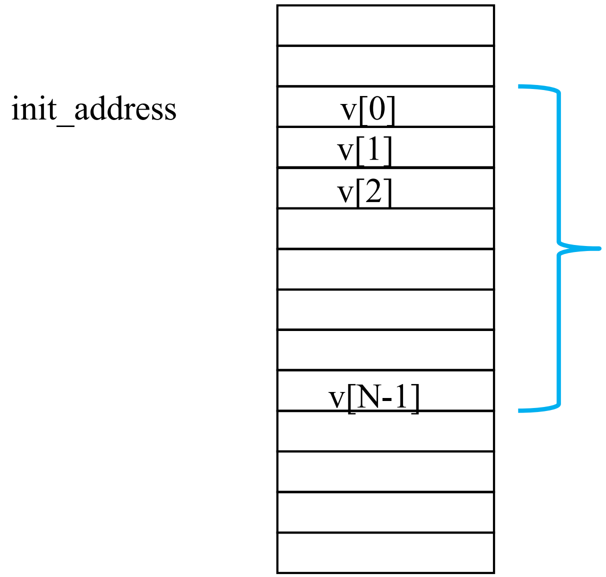
```
.data
.align 3
    result:   .space 8
    op1:          .double 100
    op2:          .double -10.27
...

.text

main:  fld     f0  op1(x0)
       fld     f1  op2(x0)
       fadd.d  f3  f1  f2
       fsd     f3  result(x0)
       ...
```

Compound data types
Arrays

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

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


.text
 main:
     la t1 vec
     li t2  8
     sw t2 16(t1)
     ...
```

Compound data types

Arrays

```c
int vec[5] ;
...

main ()
{
    vec[4] = 8;

}
```

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


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

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
 .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 instruction 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 instruction 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

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

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

```
// variables globales

char v1;
int v2 ;
float v3 = 3.14 ;

char v4 = "ec" ;

int v5[] = { 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: .asciiz "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: .asciiz "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: .asciiz "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
main:       la   t0, ac1
            li   a0, 0
            lbu  t1, 0(t0)
   buc1:    beqz t1, fin1
            addi t0, t0, 1
            addi a0, a0, 1
            lbu  t1, 0(t0)
            j    buc1

   fin1:    li a7 1
            ecall
 ...
```

Arrays and strings

- Review (in general) :
  - `lw t0, 4(s3) # t0 ← M[s3+4]`
  - `sw t0, 4(s3) # M[s3+4] ← t0`

Exercise

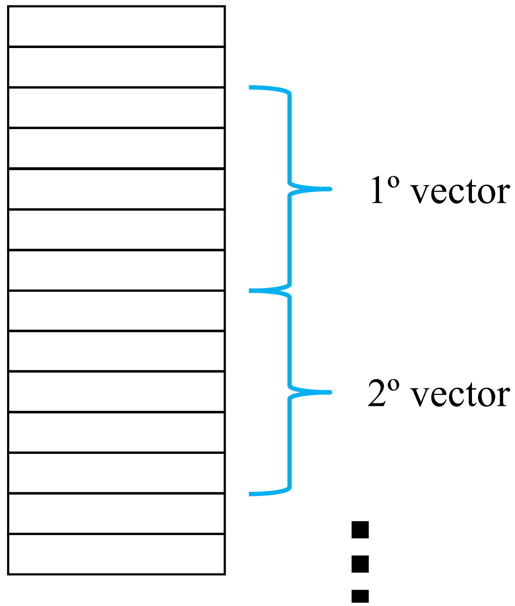
- Write a program that:
  - Calculate the number of occurrences of a char in a string
    - String address stored in a0
    - Char to look for in a1
    - Result must be stored in v0

Compound data types
Matrix

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

$$\textbf{init\_address} + (i \cdot n + j) \times p$$

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 ()
{
 mat[0][1] = mat[0][0] +
             mat[1][0] ;
 ...
}
```

```
.data
  .align 2   #siguiente dato alineado a 4
vec: .space 20    #5 elem.*4 bytes
mat: .word 11, 12, 13
     .word 21, 22, 23
...

.text
main:
          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)
        ...
```

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"
   value:         .float -3.141516

.text
main:
         flw  ft0, value(x0)

         #print
         fmv.s fa0, ft0
         li  a7, 2
         ecall

         la a0, newline
         li a7, 4
         ecall

         # copy
         fmv.x.w   t0,  ft0
```

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

    la  a0, saltolinea
    li  a7, 4
    ecall

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

    la a0,  saltolinea
    li a7,  4
    ecall

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

    jr ra
```