ARCOS Group

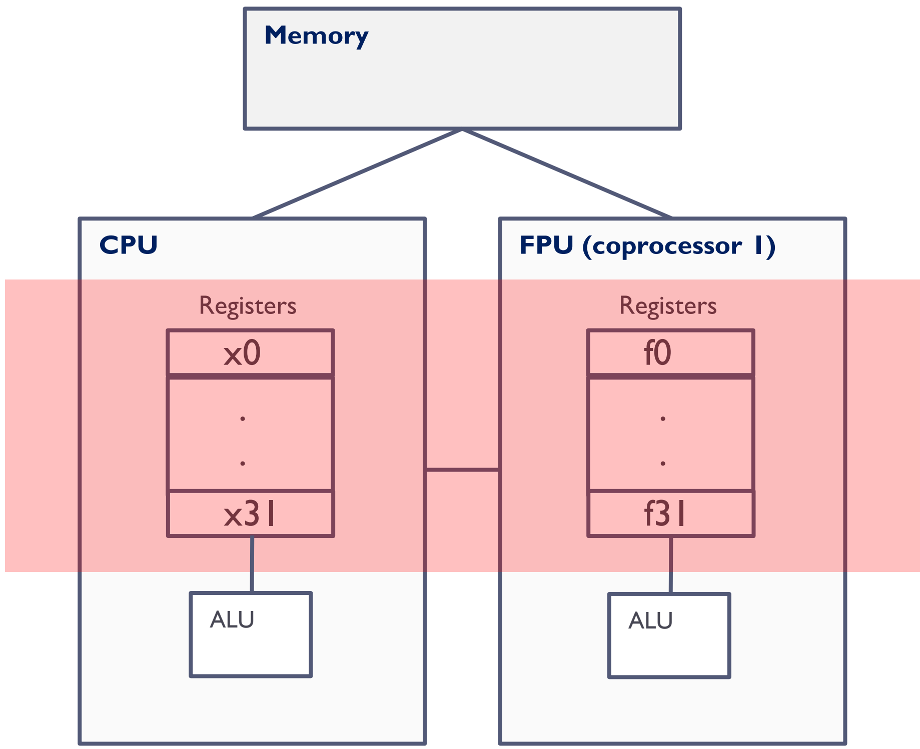
uc3m | Universidad Carlos III de Madrid

L3: Fundamentals of assembler programming (2)
Computer Structure

Bachelor in Computer Science and Engineering
Bachelor in Applied Mathematics and Computing
Dual Bachelor in Computer Science and Engineering and Business Administration

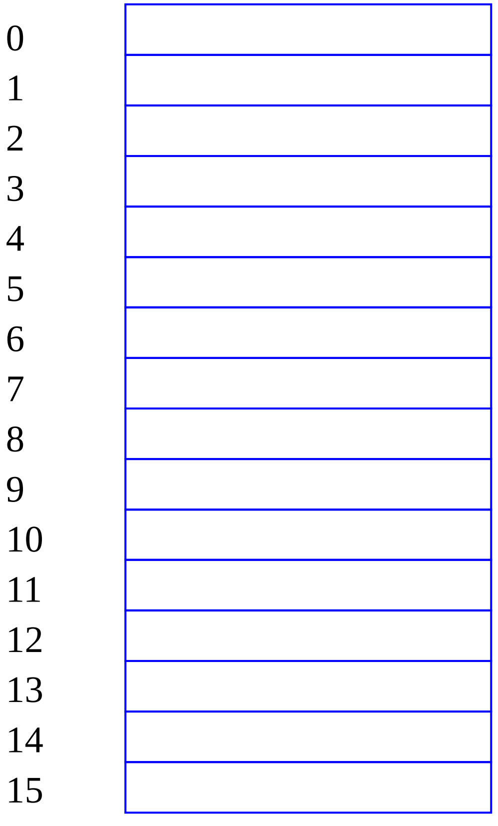
Contents

RISC-V 32

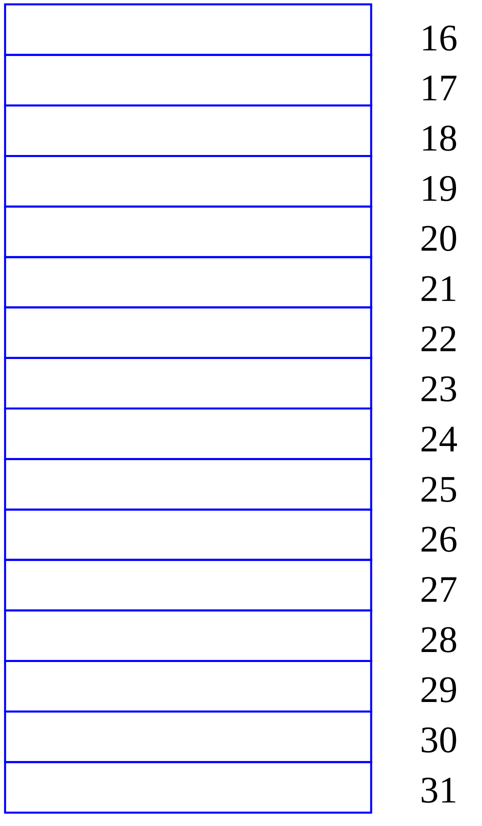


- Registers
  - Inside the processor
  - Instructions work with values in registers
  - 32 registers of 32 bits
- Memory
  - Outside the processor
  - Data exchange to/from registers
  - More capacity but longer access time than registers
  - 32-bit addresses
    - 4 GiB addressable

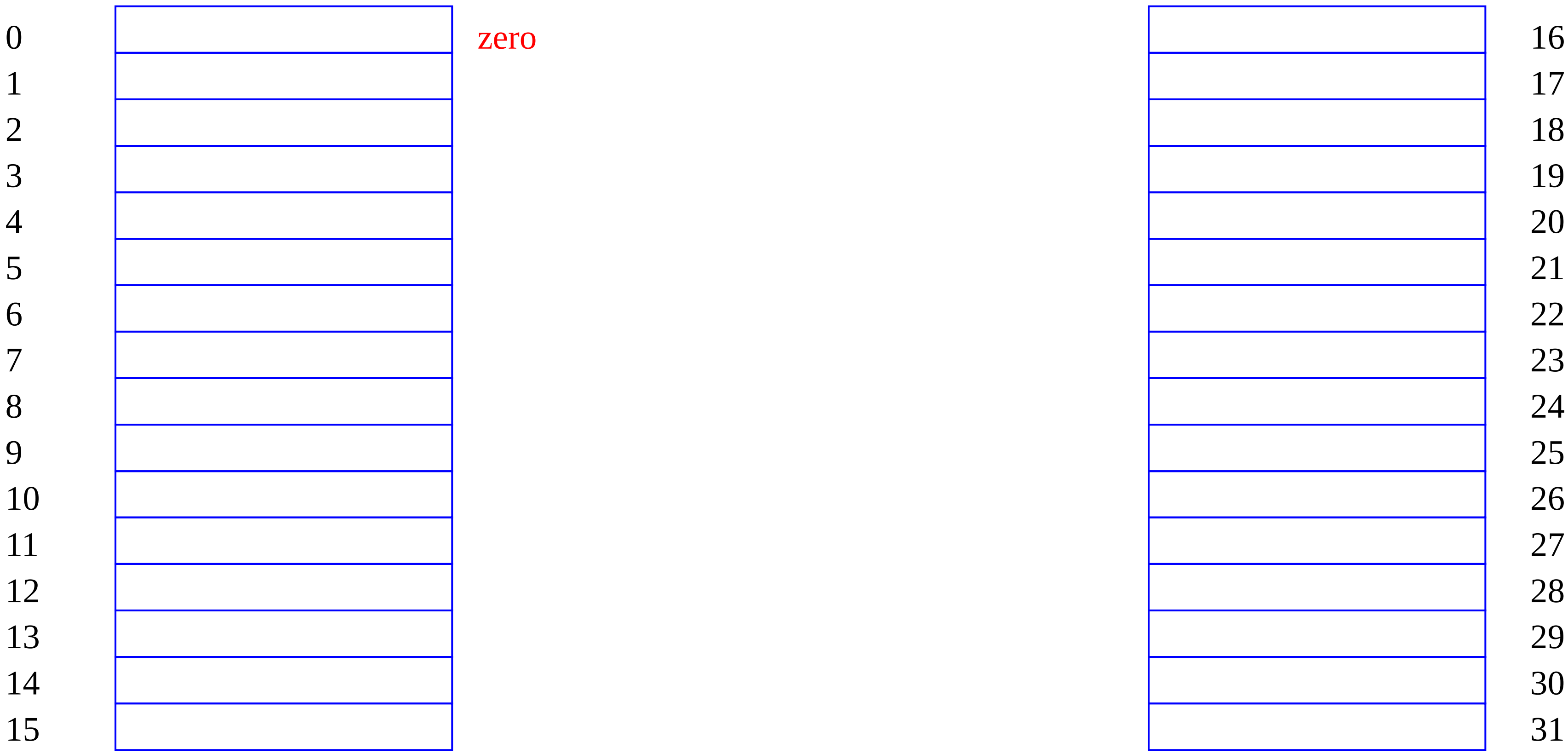
RISC-V 32 register file



- 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



Hardwired to zero
Cannot be changed

RISC-V 32 register file

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

Temporary registers

RISC-V 32 register file

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

No.	Name	Name	No.
0	zero	a6	16
1	ra	a7	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	a0	s10	26
11	a1	s11	27
12	a2	t3	28
13	a3	t4	29
14	a4	t5	30
15	a5	t6	31

Arguments and functions support

Félix García Carballeira, Alejandro Calderón Mateos

RISC-V 32 register file

Register	Name	Name	Register
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 to memory “areas” (segments)

Register file (integer)
summary

Symbolic name	Number	Usage
zero	x0	Constant 0
ra	x1	Return address (used by function calls)
sp	x2	Stack pointer
gp	x3	Pointer to global area
tp	x4	Thread pointer
t0...t2	x5-x7	Temporary (NO preserved across calls)
s0/fp	x8	Temporary (preserved across calls) / Frame pointer
s1	x9	Temporary (preserved across calls)
a0...a1	x10…11	Input function arguments / return values
a2...a7	12…x17	Input function arguments
s2… s11	x18…x27	Temporary (preserved across calls)
t3...t6	x28…x31	Temporary (NO preserved across calls)

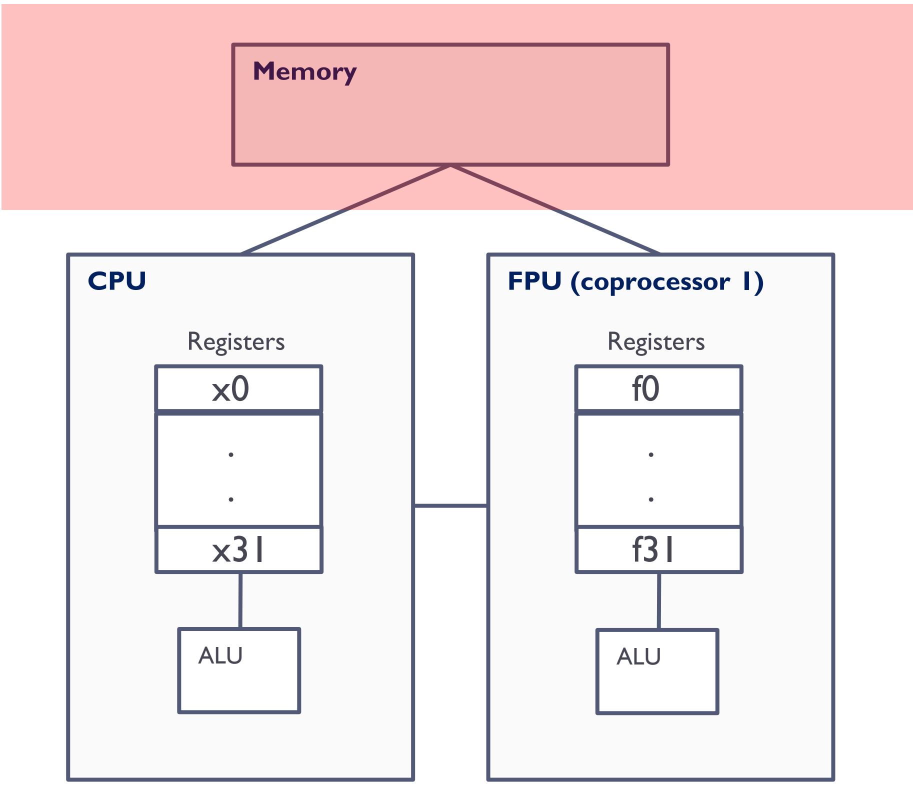
- There are 32 registers
  - Size: 4 bytes (1 word)
  - Dual name: logical and numerical (starts with x at the beginning)
- Use convention
  - Reserved
  - Arguments
  - Results
  - Temporary
  - Pointers

Register file (floating point)
summary

Logical name	Numeric name	Description
ft0-ft7	f0 … f7	Temporals (like t…)
fs0-fs1	f8 … f9	Temporals preserved (like s…)
fa0-fa1	f10 … f11	Arguments/return (like a…)
fa2-fa7	f12 … f17	Arguments (like a…)
fs2-fs11	f18 … f27	Temporals preserved (like s…)
ft8-ft11	f28 … f31	Temporals (like t…)

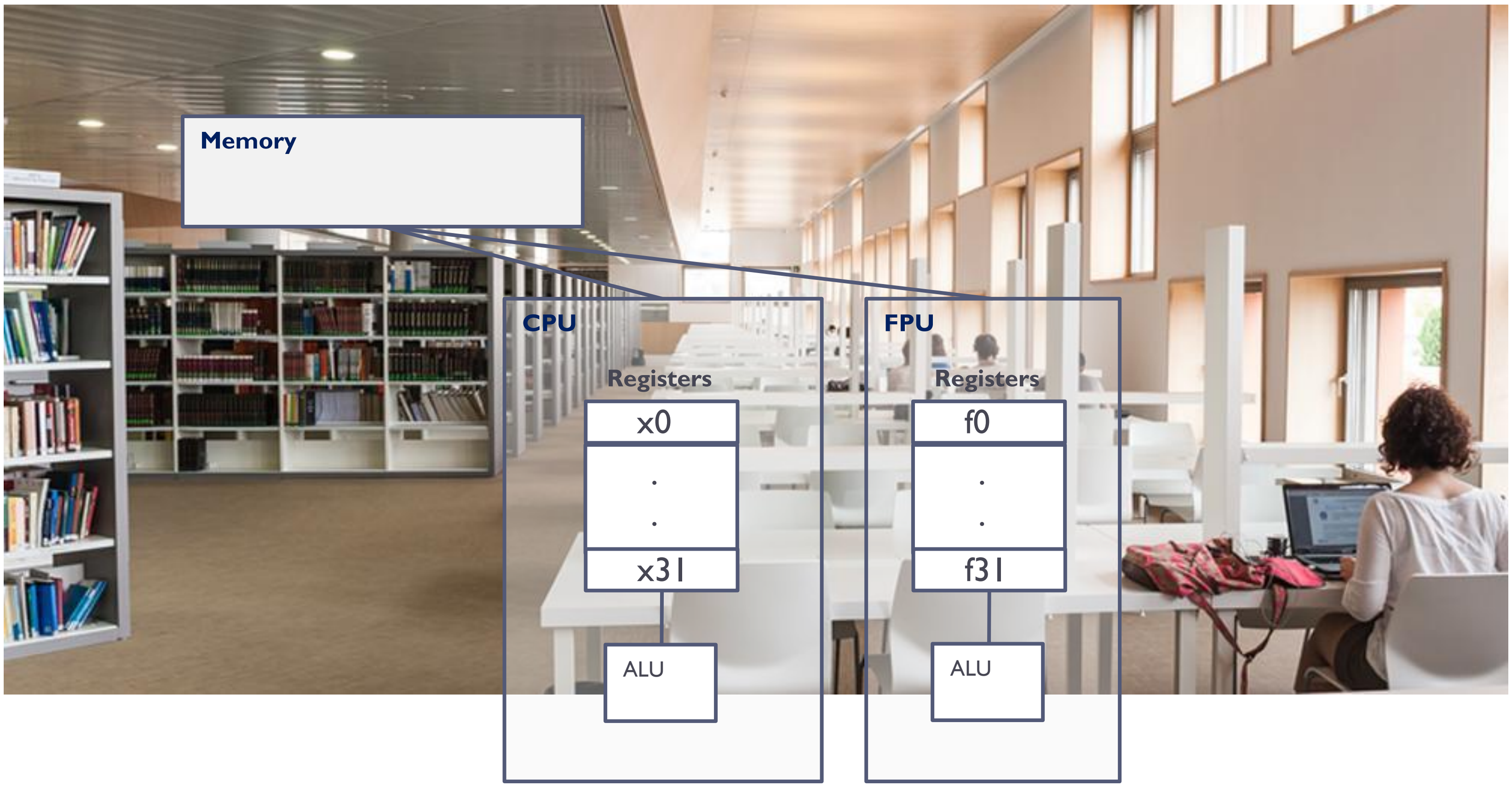
- There are 32 registers
- In R32F (simple precision) registers are 32-bits (4 bytes)
- In R32D (doble precision) registers are 64-bits (8 bytes) and can store:
  - Single-precision values in the lower 32 bits of the register
  - Double precision values in all 64 bits of the register

RISC-V 32



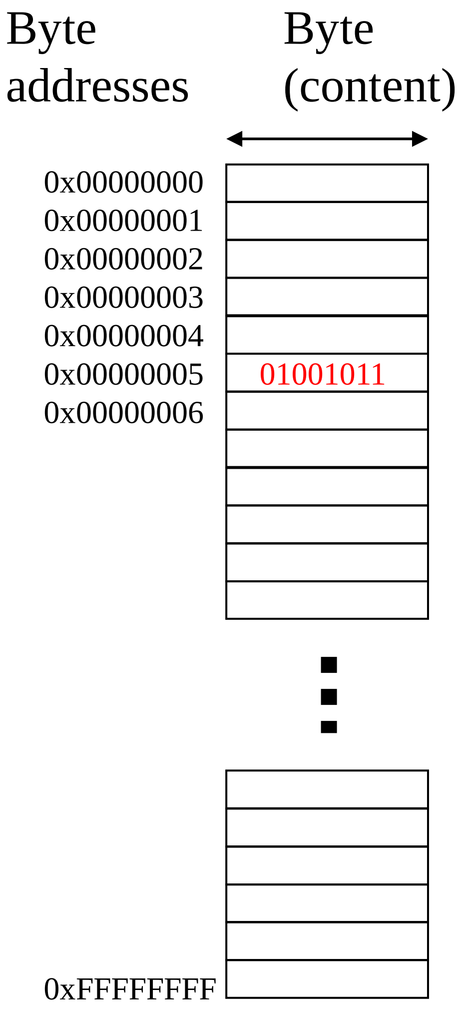
- Registers
  - Inside the processor
  - Instructions work with values in registers
  - 32 registers of 32 bits
- Memory
  - Outside the processor
  - Data exchange to/from registers
  - More capacity but longer access time than registers
  - 32-bit addresses
    - 4 GiB addressable

RISC-V 32 (memory and registers)

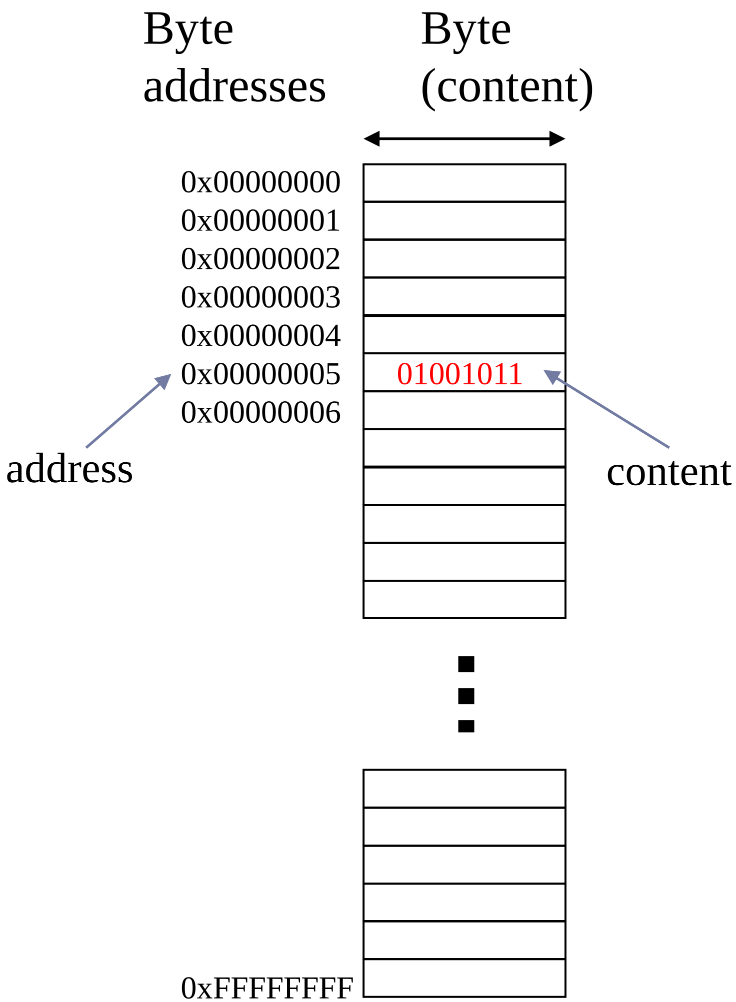


https://www.uc3m.es/ss/Satellite/UC3MInstitucional/es/TextoMixta/1371208194039/Nueva_biblioteca_de_Humanidades,_Comunicacion_y_Documentacion_%E2%80%9CCarmen_Martin_Gaite%E2%80%9D

RISC-V 32 memory model



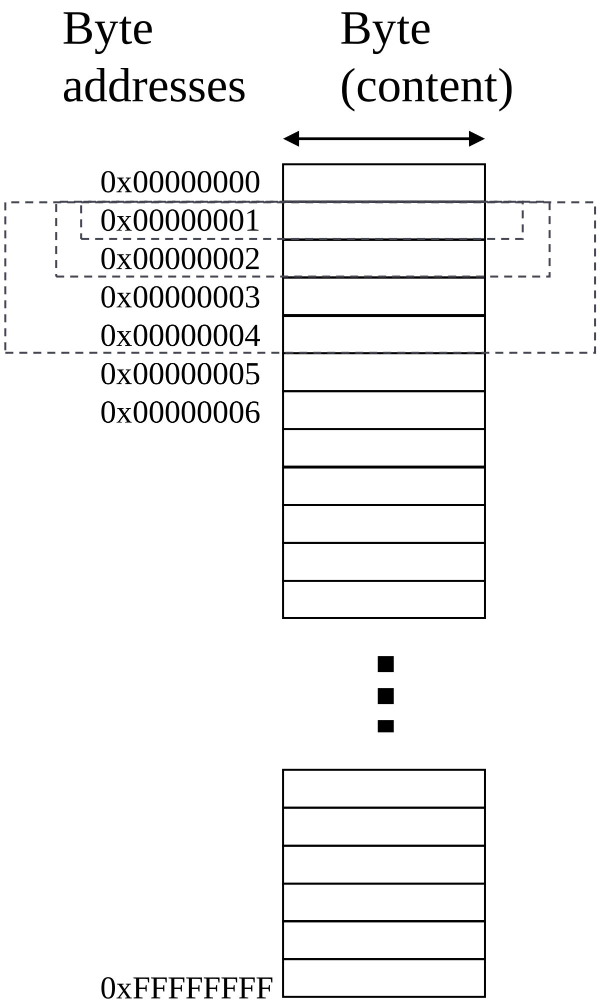
RISC-V 32 memory model



Memory is **addressed** at **byte level**:

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

RISC-V 32 memory model



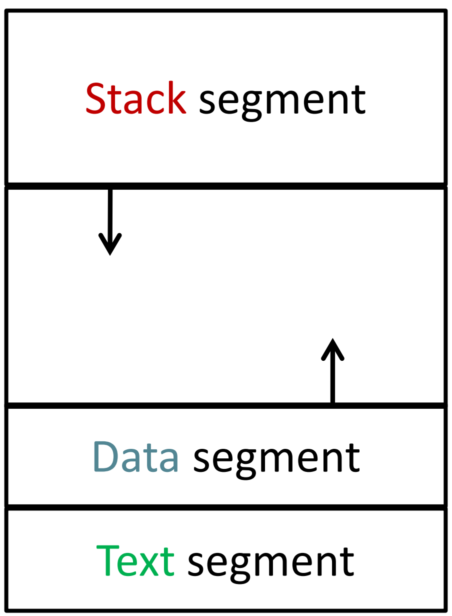
Memory is **addressed** at **byte level**:

- **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)

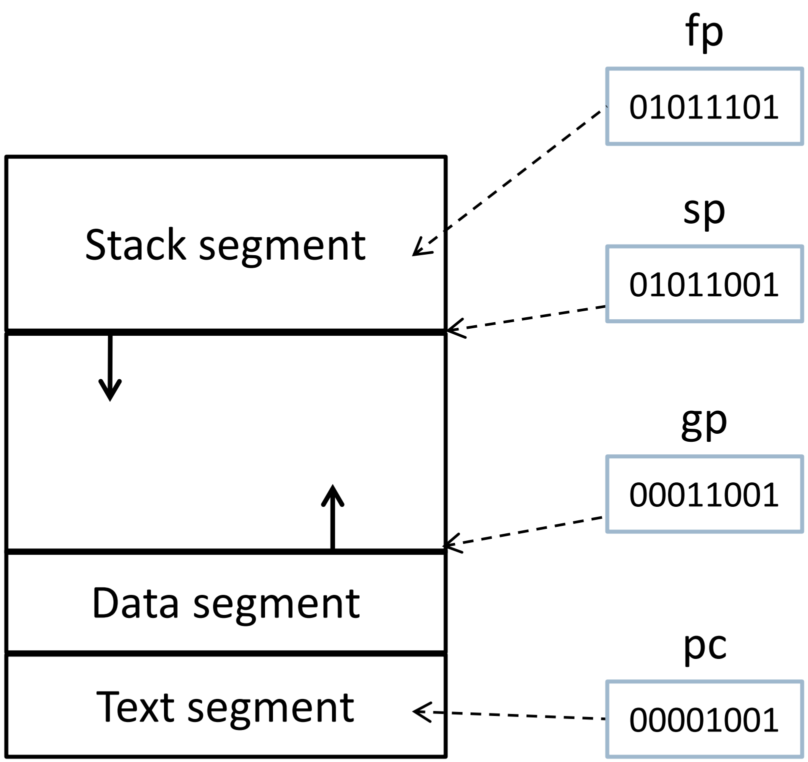
Memory layout for a process



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

```
// global variables
int a;

main ()
{
    // local variables
    int b;

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

Memory layout for a process



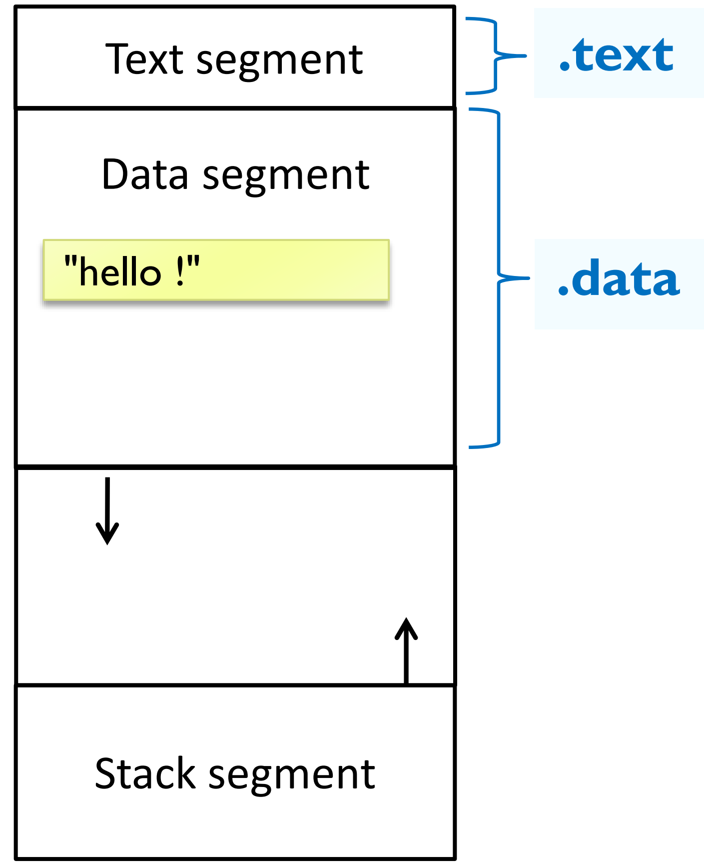
- The memory space is divided in logic segments in order to organize the content:
  - Stack segment
    - Local variables
    - Function contexts
  - Data segments
    - Global variables
    - Static variables
  - Code segment (text)
    - Program code

Example: hello world…

CREATOR
WepSIM

hello.s

```
.data
   msg_hola: .string "hello !\n"

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



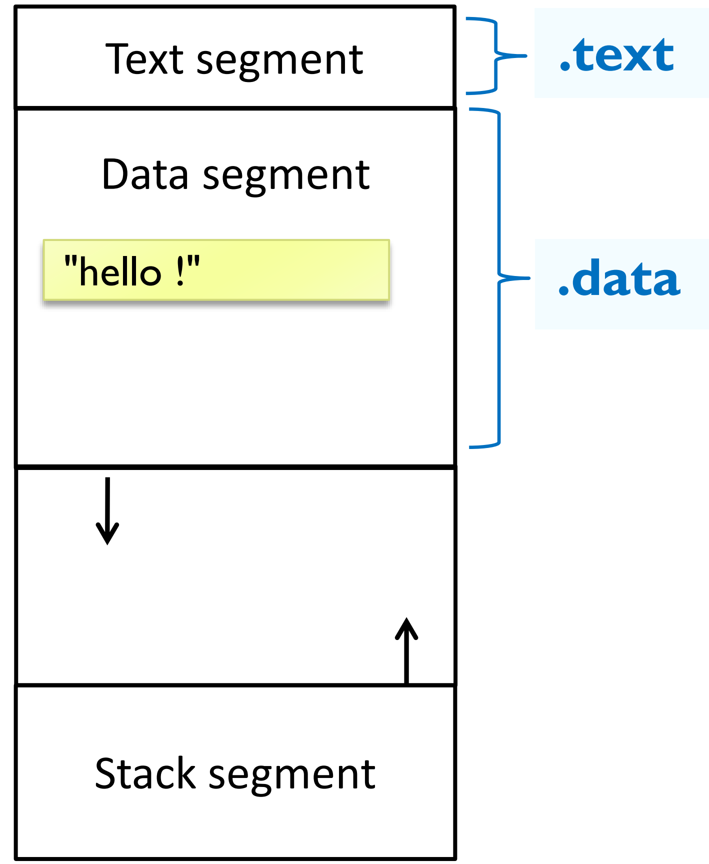
CREATOR

WepSIM

Example: hello world…

hello.s

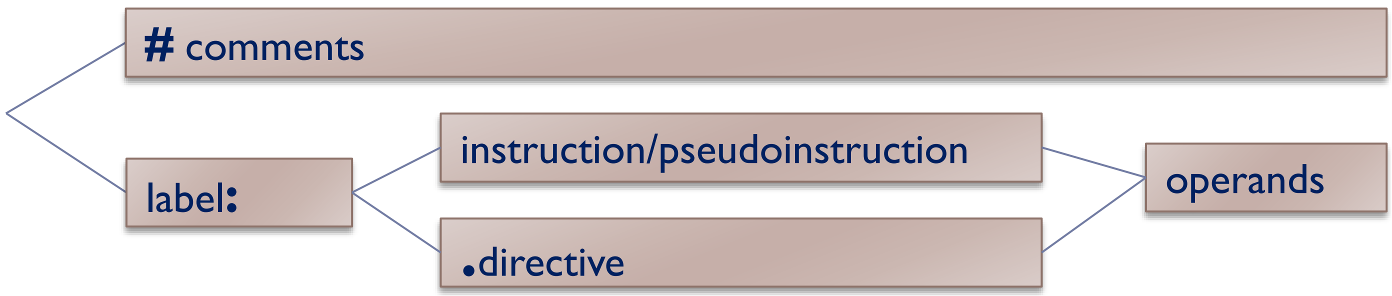
```
.data                                 Data segment
   msg_hola: .string "hello !\n"

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



Contents

Example: hello world…

CREATOR
WepSIM



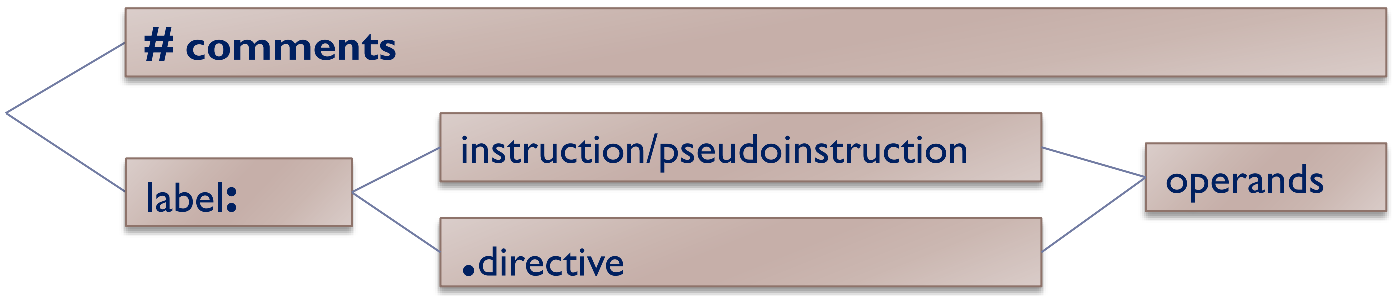
hello.s

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

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

CREATOR

WepSIM

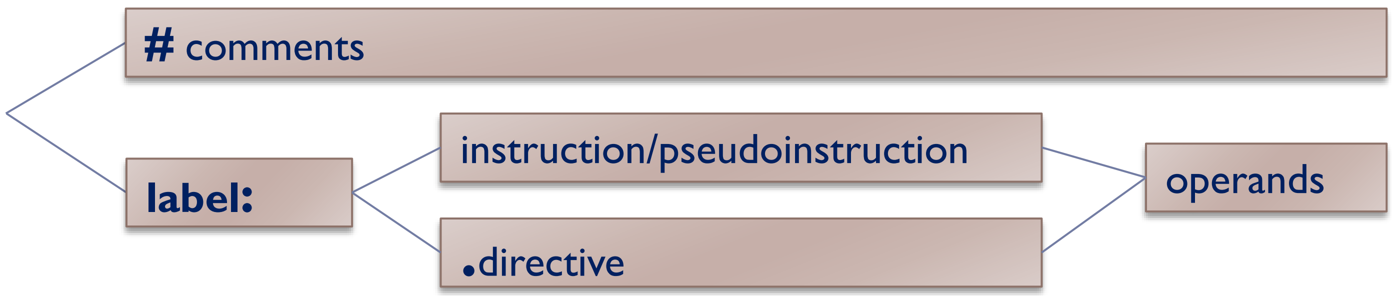
Assembly:
Comments with



hello.s

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

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

Assembly:
labels ends with :



hello.s

```
.data
   msg_hola: .string "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 starts to be stored

main: represents the memory address where the first main instruction is stored

CREATOR

WepSIM

Assembly:

Directives begins with .

hello.s

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

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

Assembly: directives

Directives	Description
.data	Next elements will go to the data segment
.text	Next elements will go to the code segment
.string "*tira de caracteres*"	String definition with '\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
.zero 10	Allocates a space of 10 bytes in the current segment
.align *n*	Align next element to a address multiple of 2^n

Definition of static data

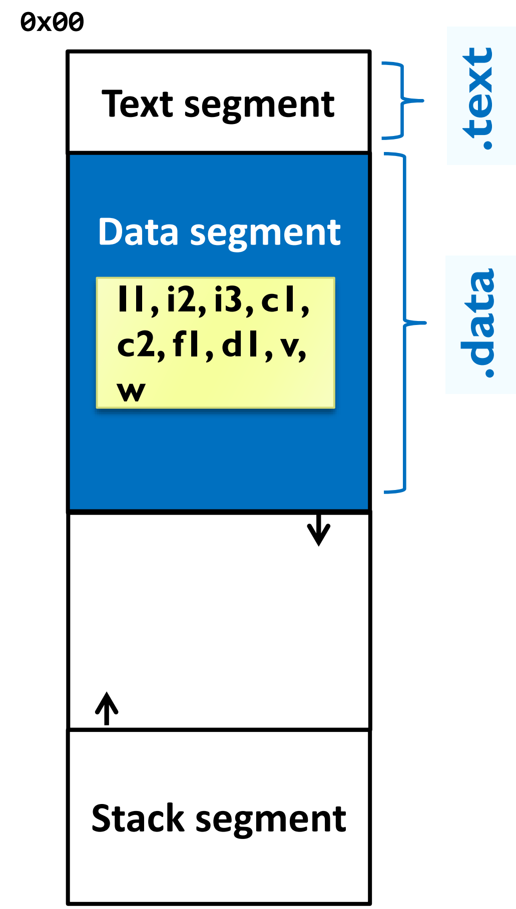
(at compile time)

label (address)

Data type (directive)

value

```
.data
 cadena: .string "Hello world\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: .zero 400
```



Representation of basic data types (1/3)

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


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


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


// floating point
float  f0 ;
float  f1 = 1.2 ;
double d2 = 3.0e10 ;
```

```
.data

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


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


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


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

Representation of basic data types (2/3)

```
// strings
char c1[10] ;
char ac1[] = "hola" ;
char ac2[] = ['h','o','l','a'] ;
```

```
.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

Félix García Carballeira, Alejandro Calderón Mateos

Representation of basic data types (3/3)

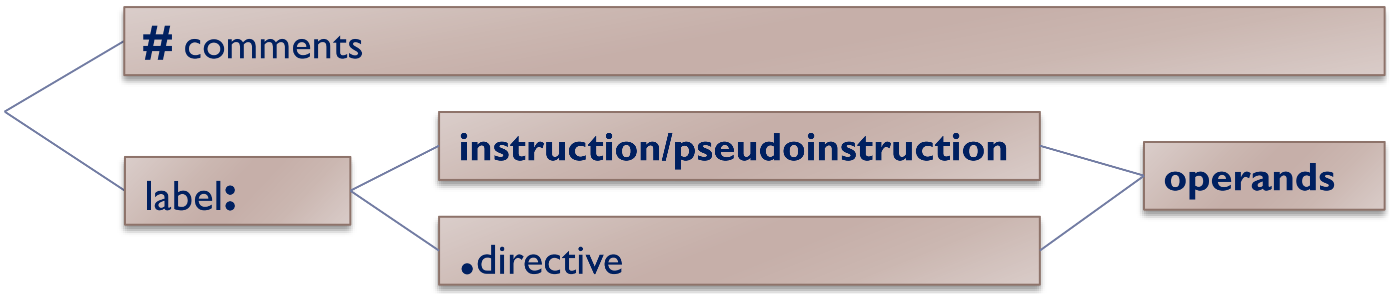
```
// vectors
int vec[5] ;
int mat[2][3] = {{11,12,13},
                 {21,22,23}};
```

```
.data

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

	...	
mat:	11	0x0108
	12	0x010c
	13	0x0110
	21	0x0114
	22	0x0118
	23	0x011c
	...	

ARCOS @ UC3M
Félix García Carballeira, Alejandro Calderón Mateos

Assembly:
Instructions vs pseudoinstructions



hello.s

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

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

Instructions and pseudoinstructions

- An assembly **instruction** corresponds to a single machine instruction.:
  - Occupies 32 bits in RISC-V 32
  - E.g.: addi t1, t0, 4

- A pseudo-instruction can be used in an assembler program but does not correspond to any machine instruction:
  - In the assembly process, they are replaced by the sequence of machine instructions that perform the same functionality.

E.g.: mv t1, t2 is replaced by: add t1, x0, t2

li t1, 0x00800010 is replaced by: lui t1, 0x00800 (20 bits)

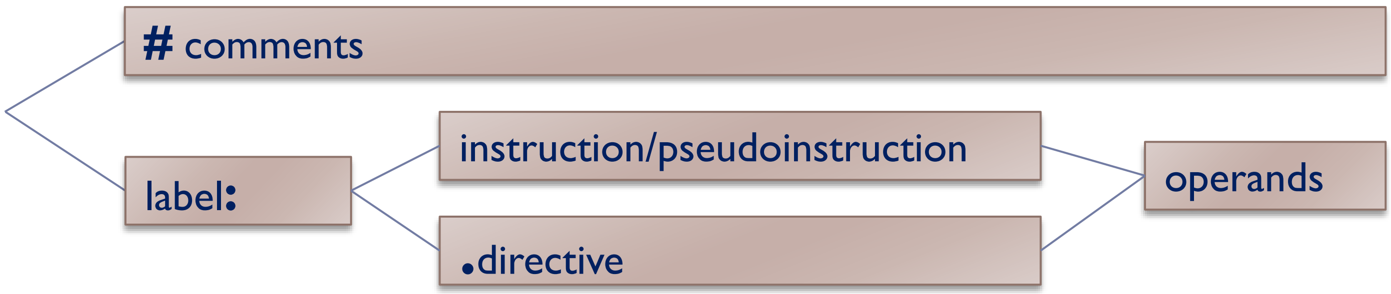
addi t1, t1, 0x010 (12 bits)

Félix García Carballeira, Alejandro Calderón Mateos

CREATOR

WepSIM

Assembly:
summary



hello.s

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

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

Contents

- Basic concepts on assembly programming
- RISC-V 32 assembly, memory model and data representation
  - RISC-V: registers and memory
  - Assembler directives
  - System services
  - Memory access instructions
- Instruction formats and addressing modes
- Procedure calls and stack convention

System calls

- Many simulators include a small "operating system"
  - CREATOR provides 12 services.

- How to invoke:
  - Call code in register **a7**
  - Other arguments on specific registers (ej.: **a0**, **fa0**)
  - Invocation by the **ecall** instruction

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

System calls

Service	Call code (a7)	Arguments (a7 and fa0)	Results (a7 and 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 in a0
read_float	6		float in fa0
read_double	7		double in fa0
read_string	8	a0 = buffer, a1 = length	
sbrk	9	a0 = amount	Address in a0
exit	10		
print_char	11	a0 (ASCII code)	
read_char	12		a0 (ASCII code)

Example: print “hello world”…

hello.s

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

.text
    main:
      # printf("hello world\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

Operating system invocation instruction

Exercise: read integer, print integer

```
. . .

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

. . .
```

Exercise: read integer, print integer
solution

CREATOR
WepSIM

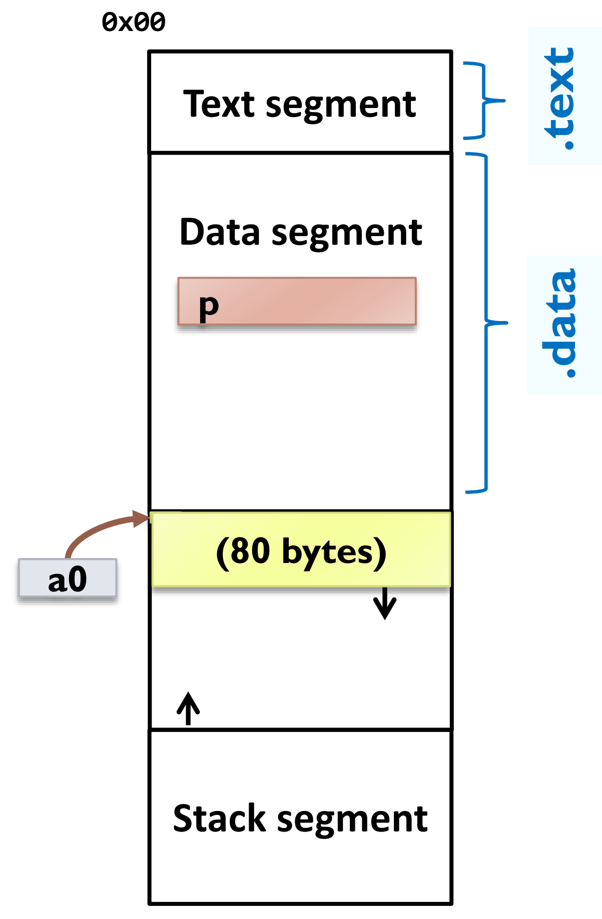
```
. . .

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

. . .
```

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(&value)
li a7 5
ecall
mv t0 a0 # value in t0

# value = value + 1
addi t0  t0  1

# printInt
mv  a0 t0
li  a7 1
ecall

. . .
```

sbrk: dynamic data allocation
(at runtime)

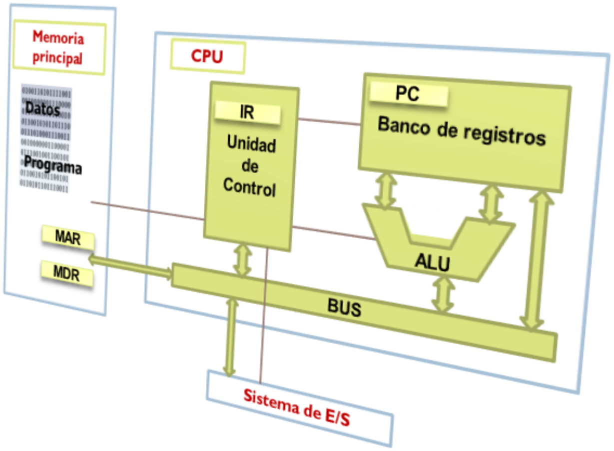
```
.data
  # char *p; // pointer to a byte
  p: .word 0x0

.text

  # p = malloc(80); // 80 bytes
  li a0, 80
  li a7, 9 # system call
  ecall

  # the address is in a0
  ...

  # free(p); // free 80 bytes
  li a0, -80
  li a7, 9 # system call
  ecall
```



Contents

- Basic concepts on assembly programming
- RISC-V 32 assembly, memory model and data representation
  - RISC-V: registers and memory
  - Assembler directives
  - System services
  - Memory access instructions
- Instruction formats and addressing modes
- Procedure calls and stack convention

Access to memory



- Loading an address into a register:

```
la reg2, dir32
```

  - reg2 ← dir32

- Memory access instructions (integers):

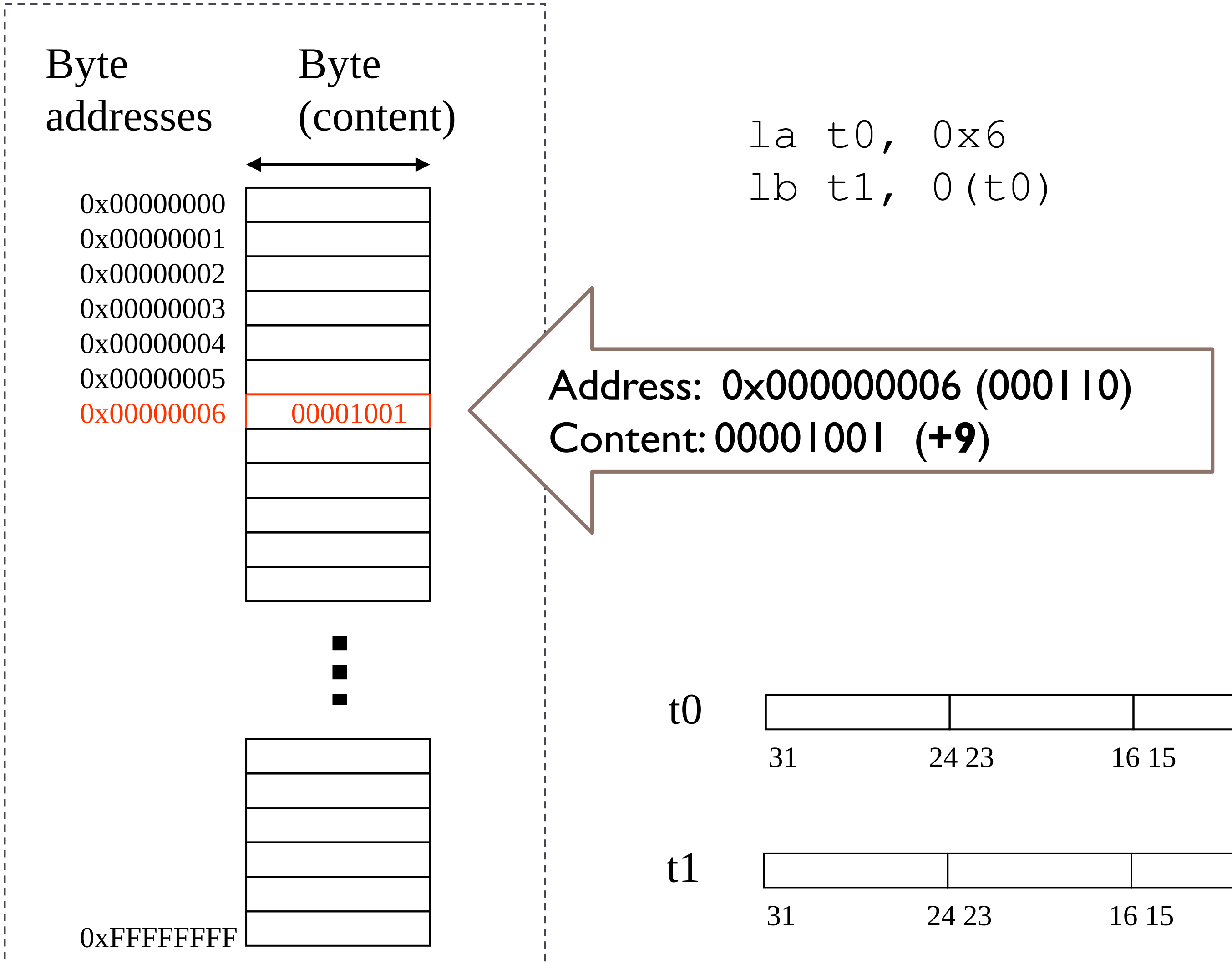
```
lw
sw
lb      reg1,  num(reg2)
lbu
sb
```

  - **num(registro)**: represents the address obtained by adding num to the address stored in the reg2 register

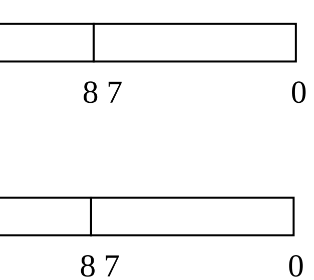
Memory operations

Memory access		
`la  reg2, dir32`	`reg2 = dir32`	Load the 32-bit memory address dir32 into the reg2 register (pseudoinstruction)
`lw  r1, num(r2)`	`r1 = Memory[r2+num]`	Accesses a stored word from memory location r2+num and saves it to r1
`sw  r1, num(r2)`	`Memory[r2+num] = r1`	Stores from memory location r2+num the word contained in r1
`lb  r1, num(r2)`	`r1 = Memory [r2+num]`	Accesses a byte stored in memory location r2+num and saves it in the least significant byte of r1
`lbu r1, num(r2)`	`r1 = Memory [r2+num]`	Accesses a byte stored in memory location r2+num and saves it in the least significant byte of r1
`sb  r1, num(r2)`	`Memory[r2+num] = r1`	Saves from memory location r2+num the least significant byte of r1

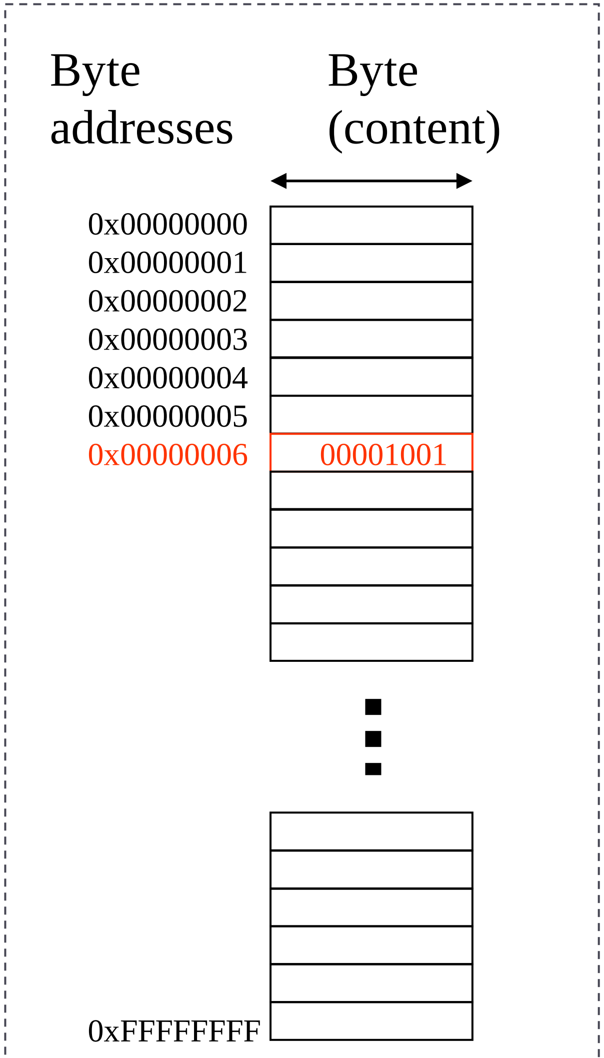
Access to bytes with lb (*load byte*)




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

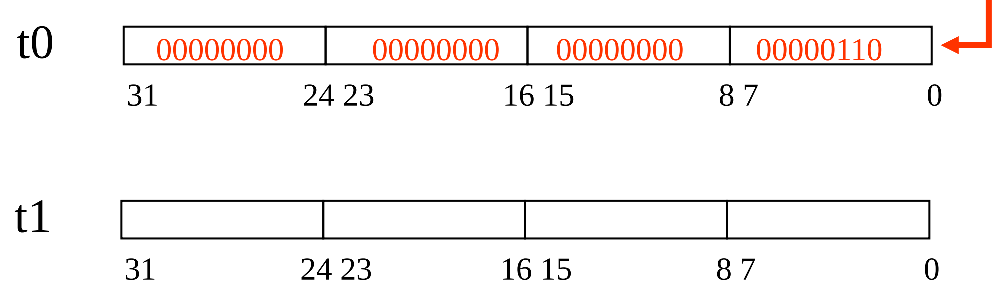


Access to bytes with lb (*load byte*)




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

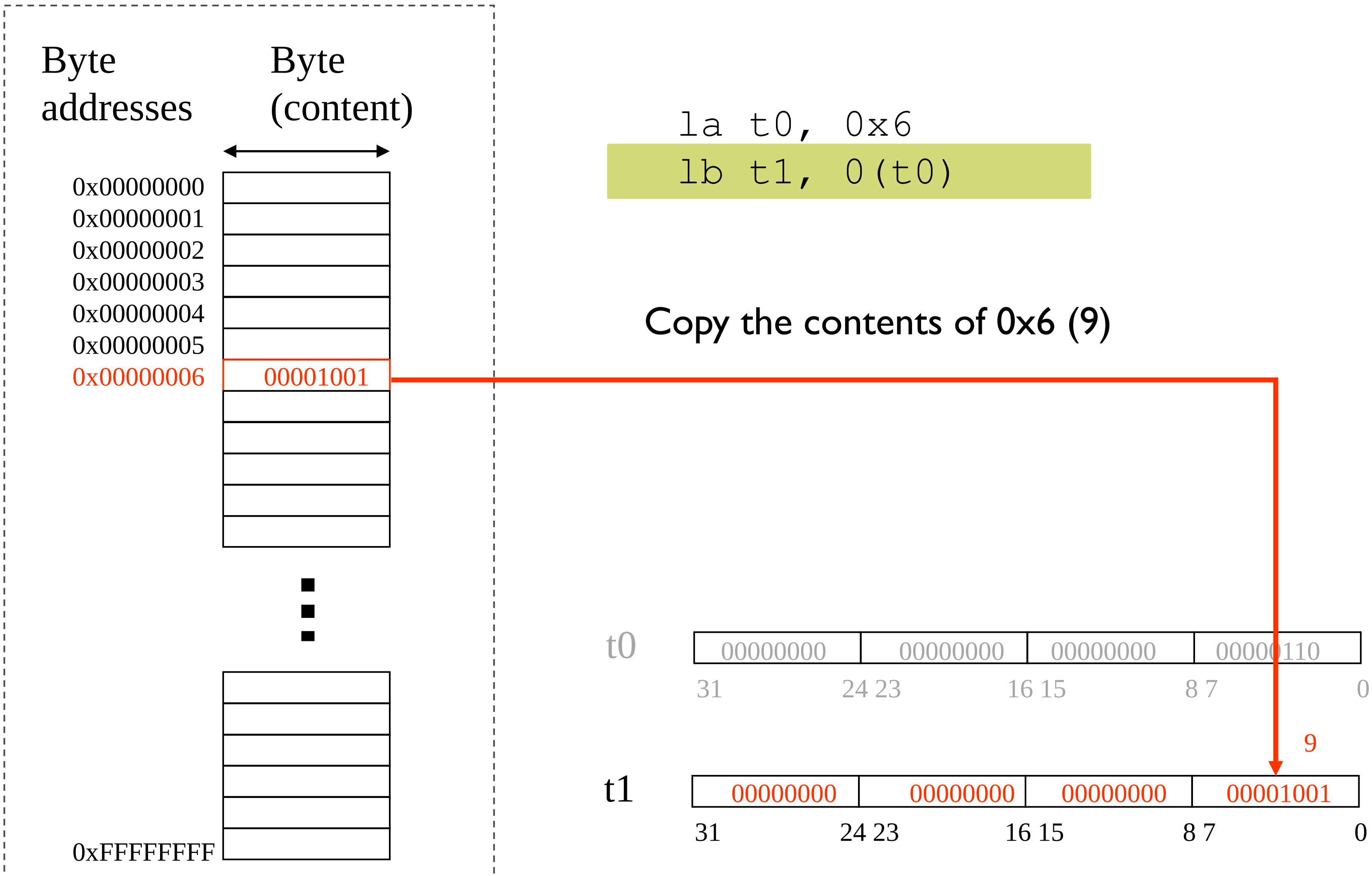
The address is copied, not the content



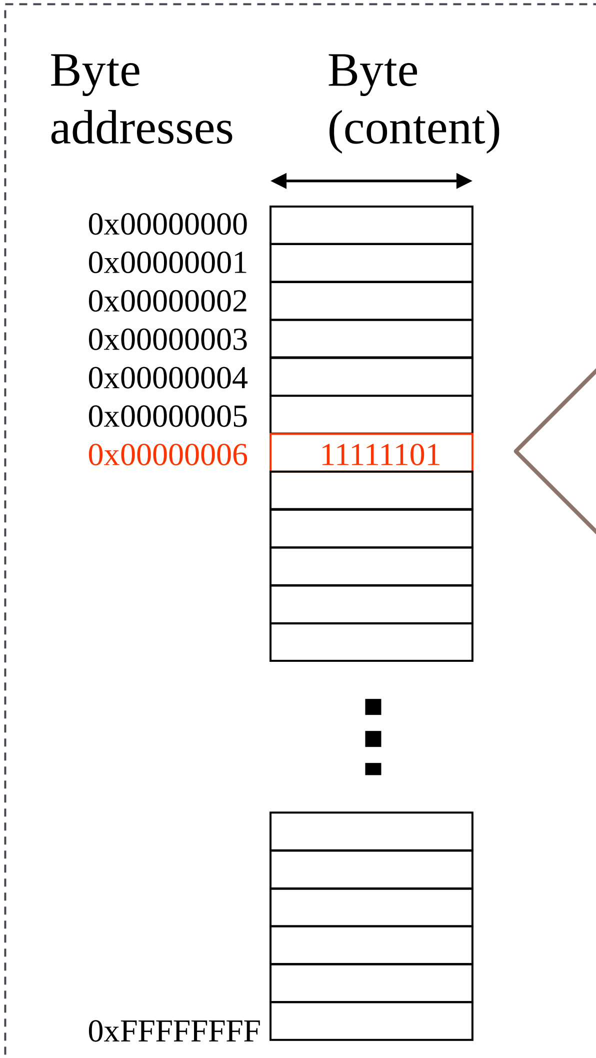
Access to bytes with lb (*load byte*)

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

Copy the contents of 0x6 (9)

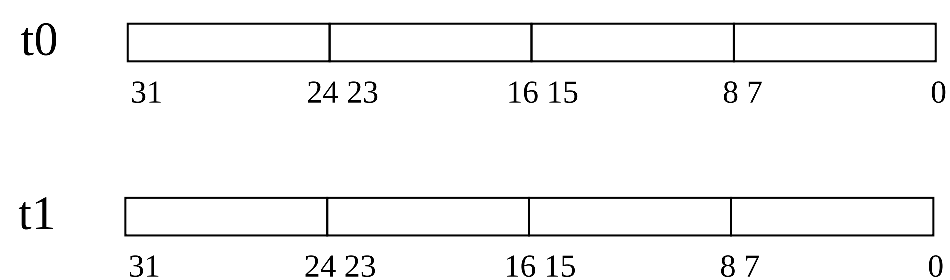


Access to bytes with lb (*load byte*)

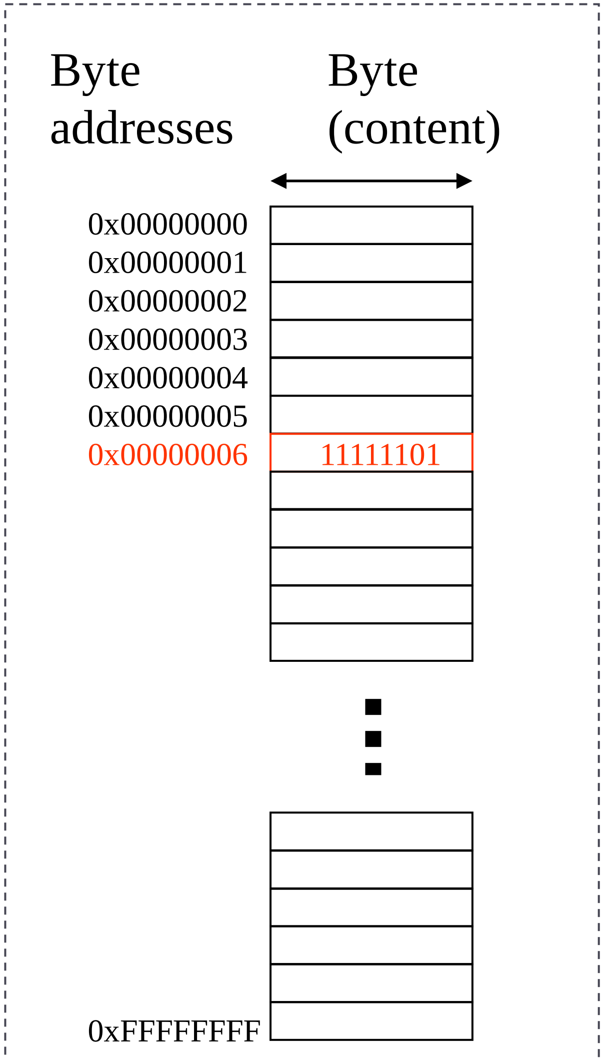



```
la t0, 0x6
lb t1, 0(t0)
```

Address: 0x000000006 (000110)
Content: 11111101 **(-3 in 2'C)**

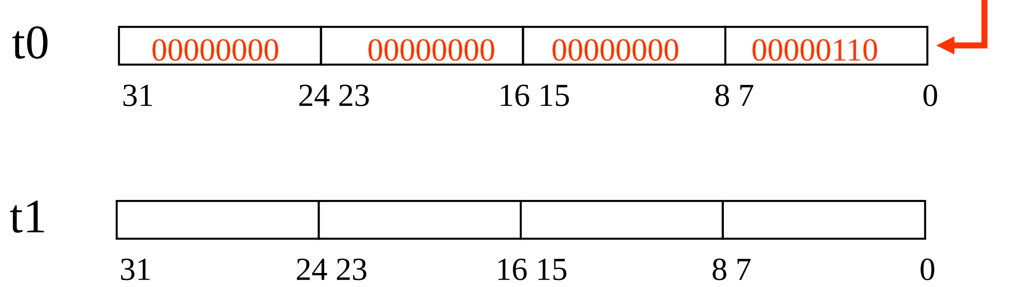


Access to bytes with lb (*load byte*)




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

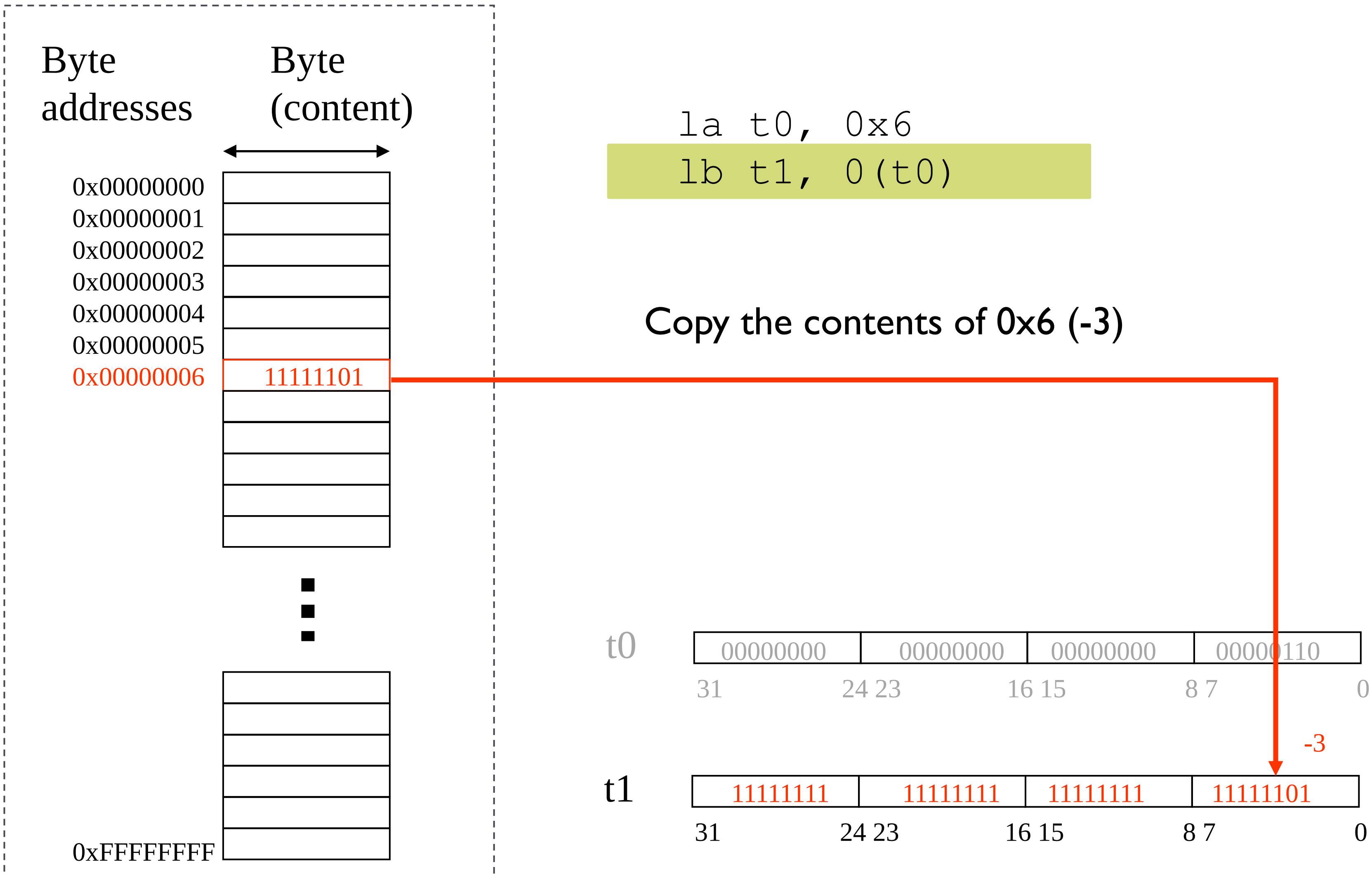
Copy 0x6 to t0



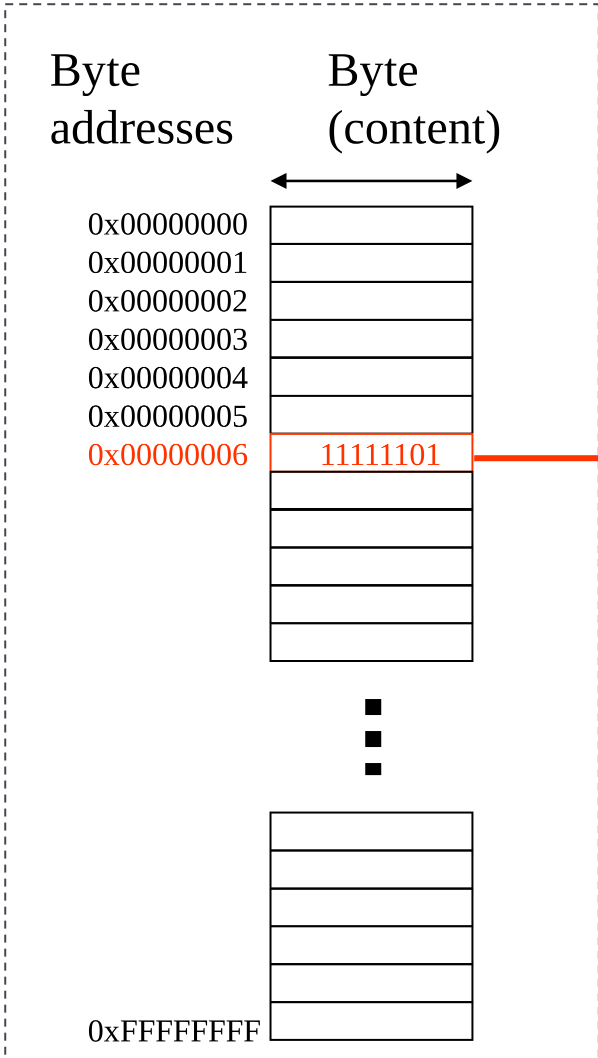
Access to bytes with lb (*load byte*)

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

Copy the contents of 0x6 (-3)



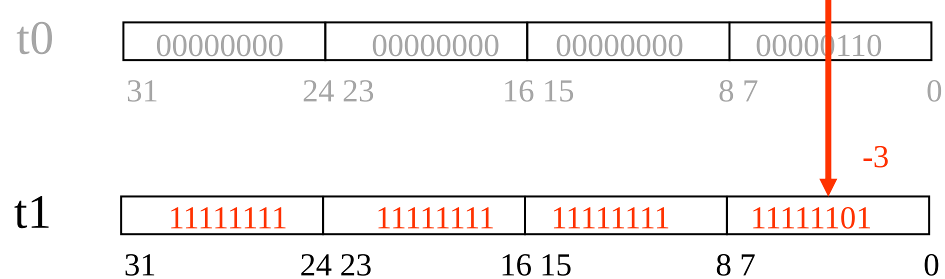
Access to bytes with lb (*load byte*)



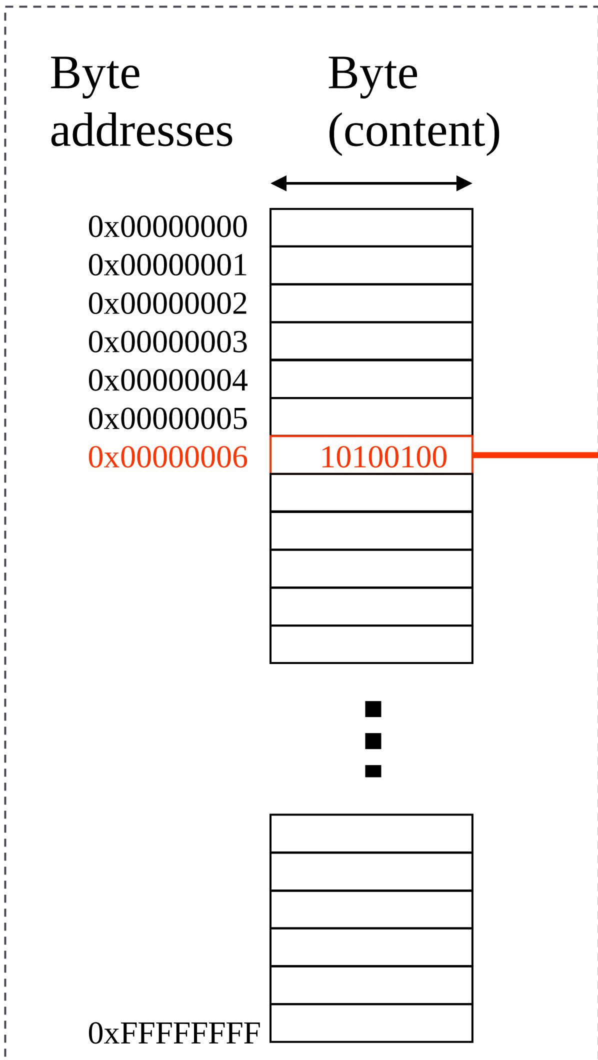

```
la t0, 0x6
lb t1, 0(t0)
```

Copy the contents of 0x6 (-3)

The lb instruction keeps the sign:
does sign extension



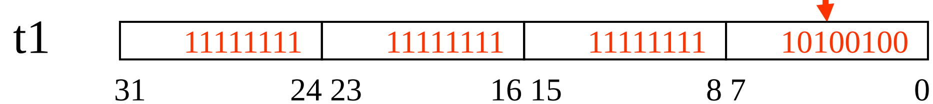
Access to bytes with lb (*load byte*)
problems accessing characters




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

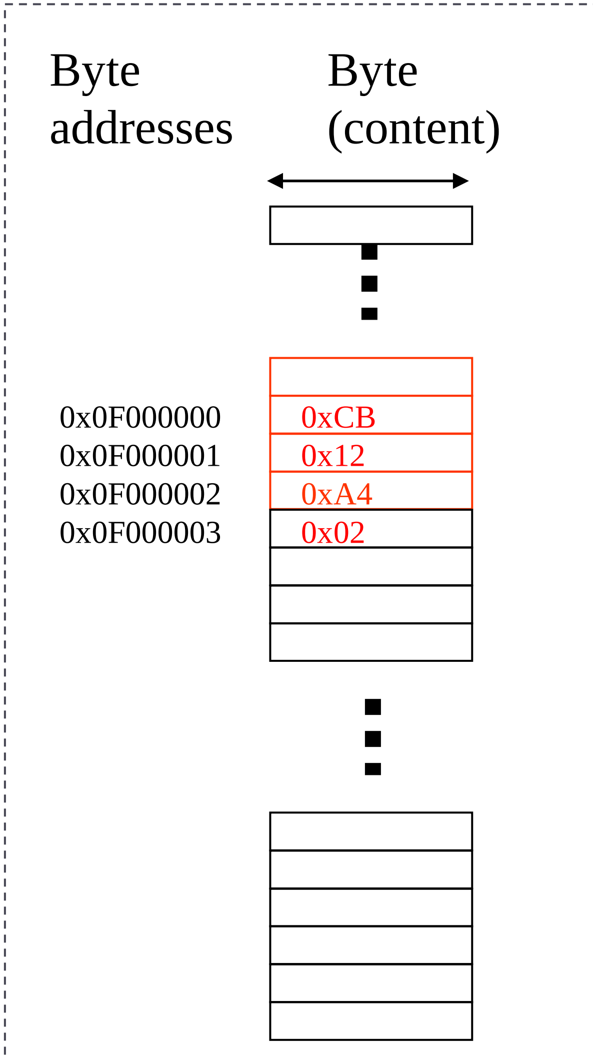
Address: 0x000000006 (000110)
Content: 10100100 (ASCII code for ñ: 164)

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

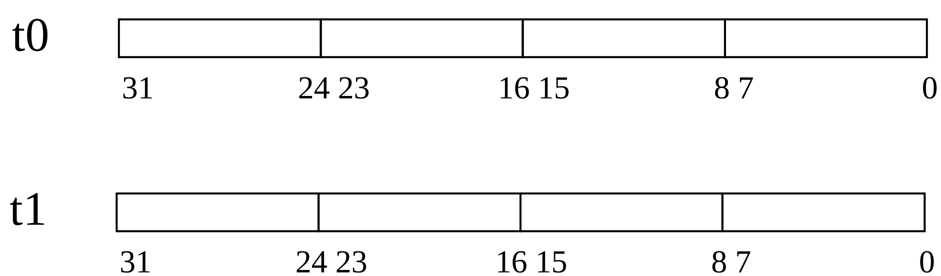


If lb is used (the sign is kept),
the content of t1 **does not** coincide with
the value 164 (ñ)

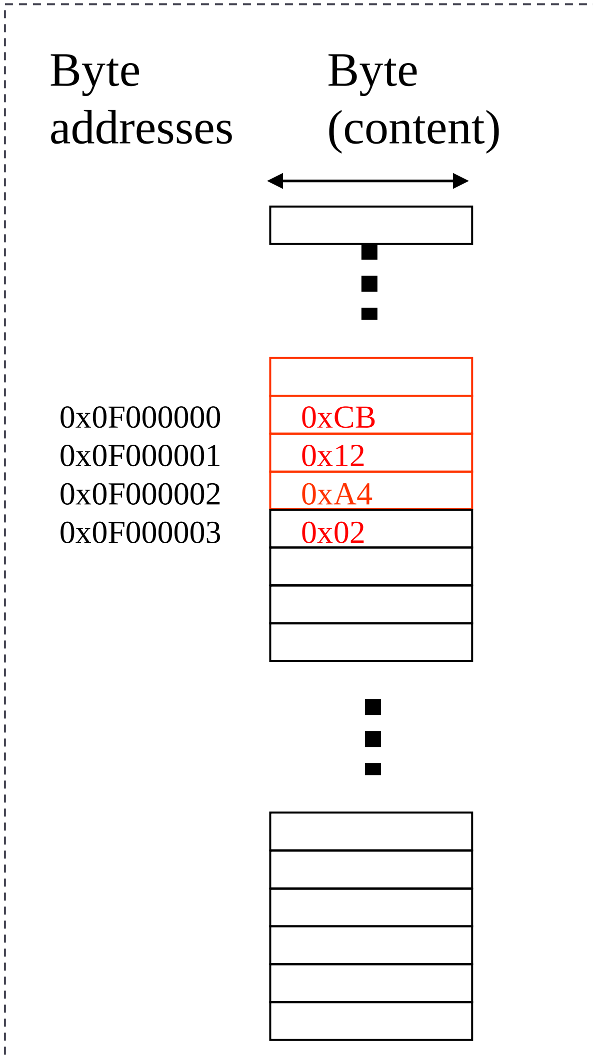
Access to bytes with lbu (*load byte unsigned*)




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

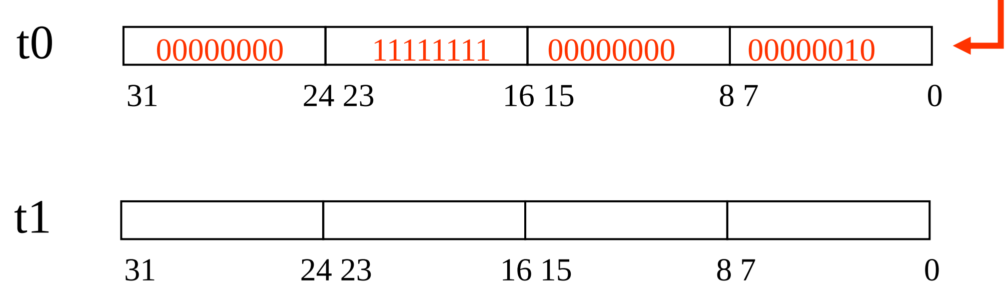


Access to bytes with lbu (*load byte unsigned*)

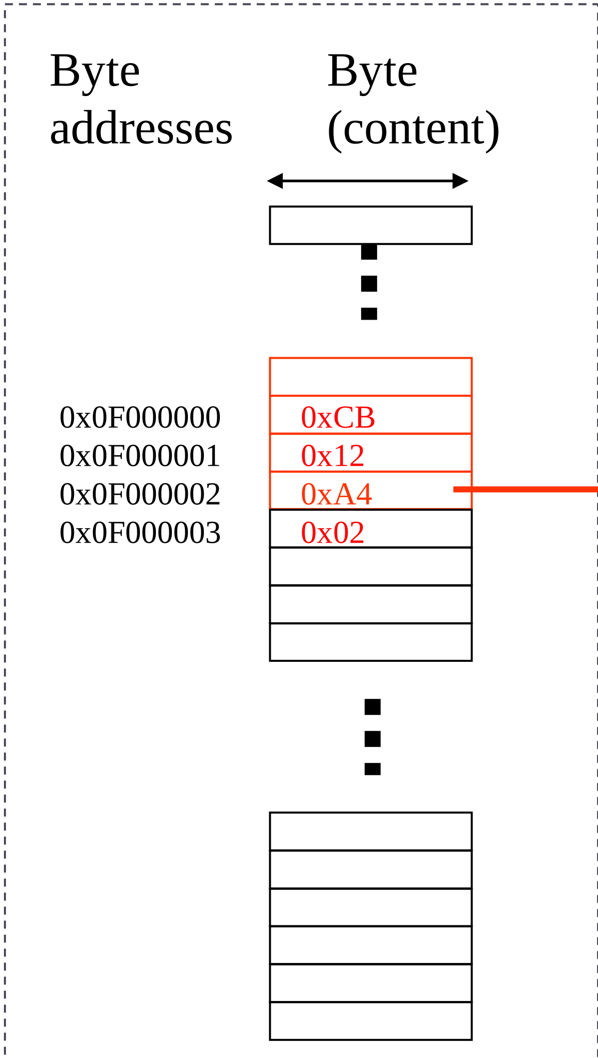



```
la   t0, 0x0F000002
lbu  t1, 0(t0)
```

The address is copied, not the content

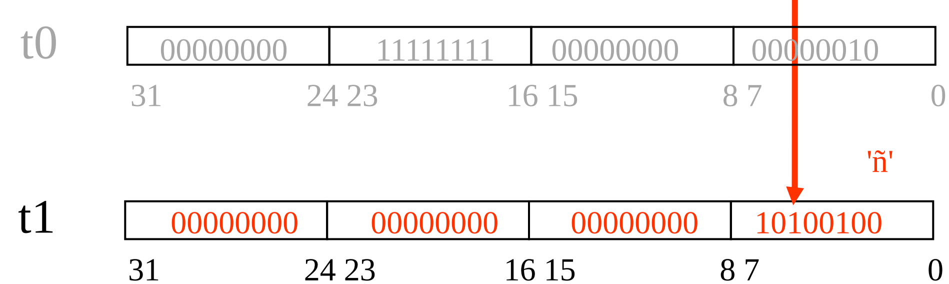


Access to bytes with lbu (*load byte unsigned*)

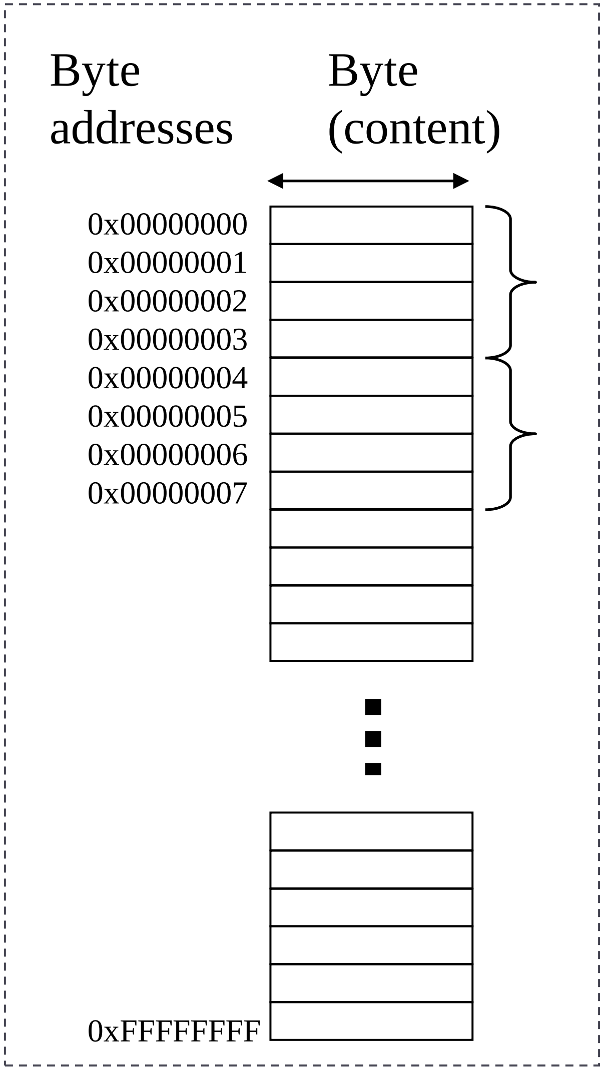



```
la   t0, 0x0F000002
lbu  t1, 0(t0)
```

The byte stored at position `0x0F000002` is copied (**without** sign extension)



Access to words



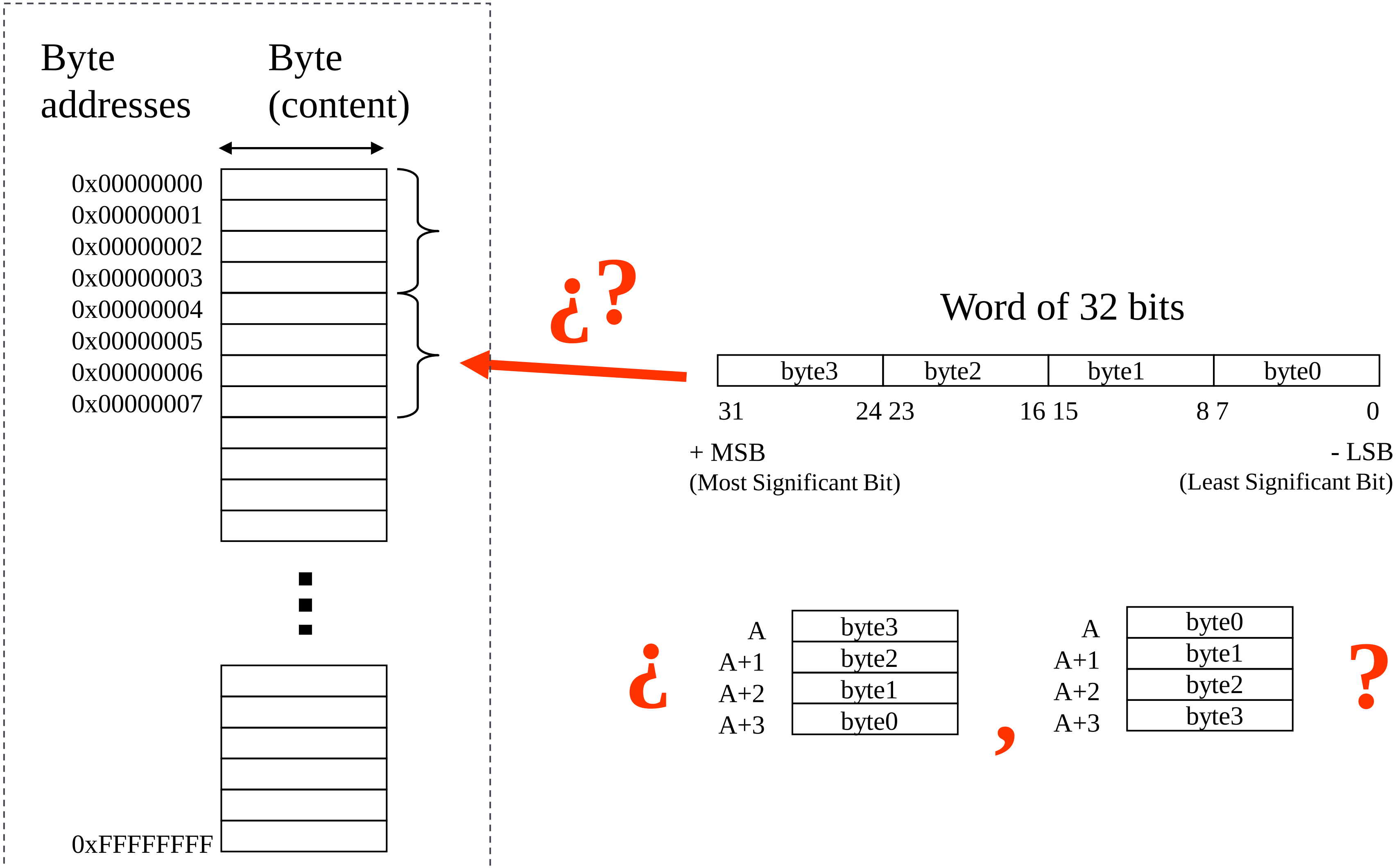
4 bytes make a word

Word stored from byte 0

Word stored from byte 4

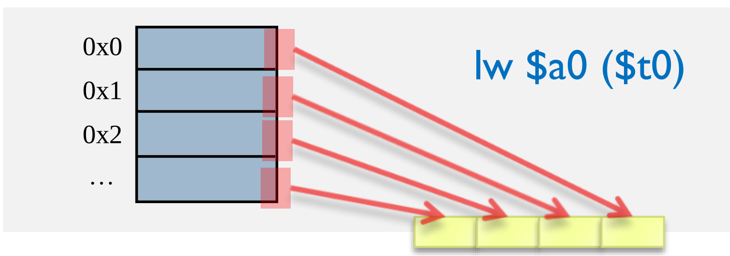
The words (32 bits, 4 bytes) are stored using four consecutive memory locations, with the first location starting at an address multiple of 4

Access to words



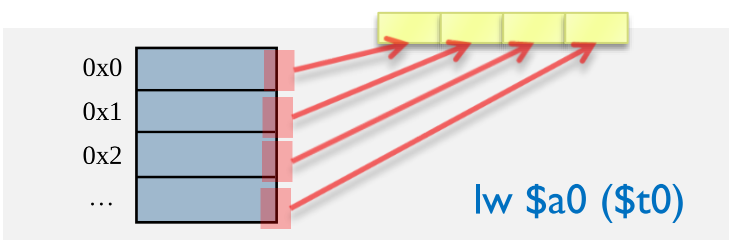
Storage of words in memory
byte order

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



(bi-endian)

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



Storage of words in memory
example

32-bit word

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

+ MSB
(Most Significant Bit)

- LSB
(Least Significant Bit)

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 $27_{(10}$ is represented

00000000000000000000000000011011

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

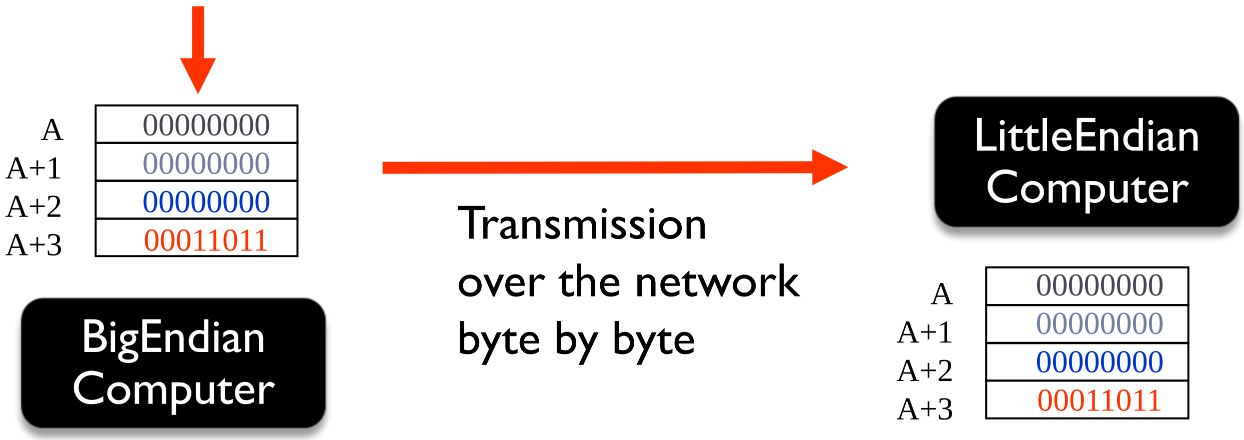
BigEndian Computer

LittleEndian Computer

A	
A+1	
A+2	
A+3	

Communication problems in computers with different architectures

The number $27_{(10}$ is represented
00000000000000000000000000011011



Communication problems in computers with different architectures

The number $27_{(10}$ is represented
00000000000000000000000000011011

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

BigEndian Computer

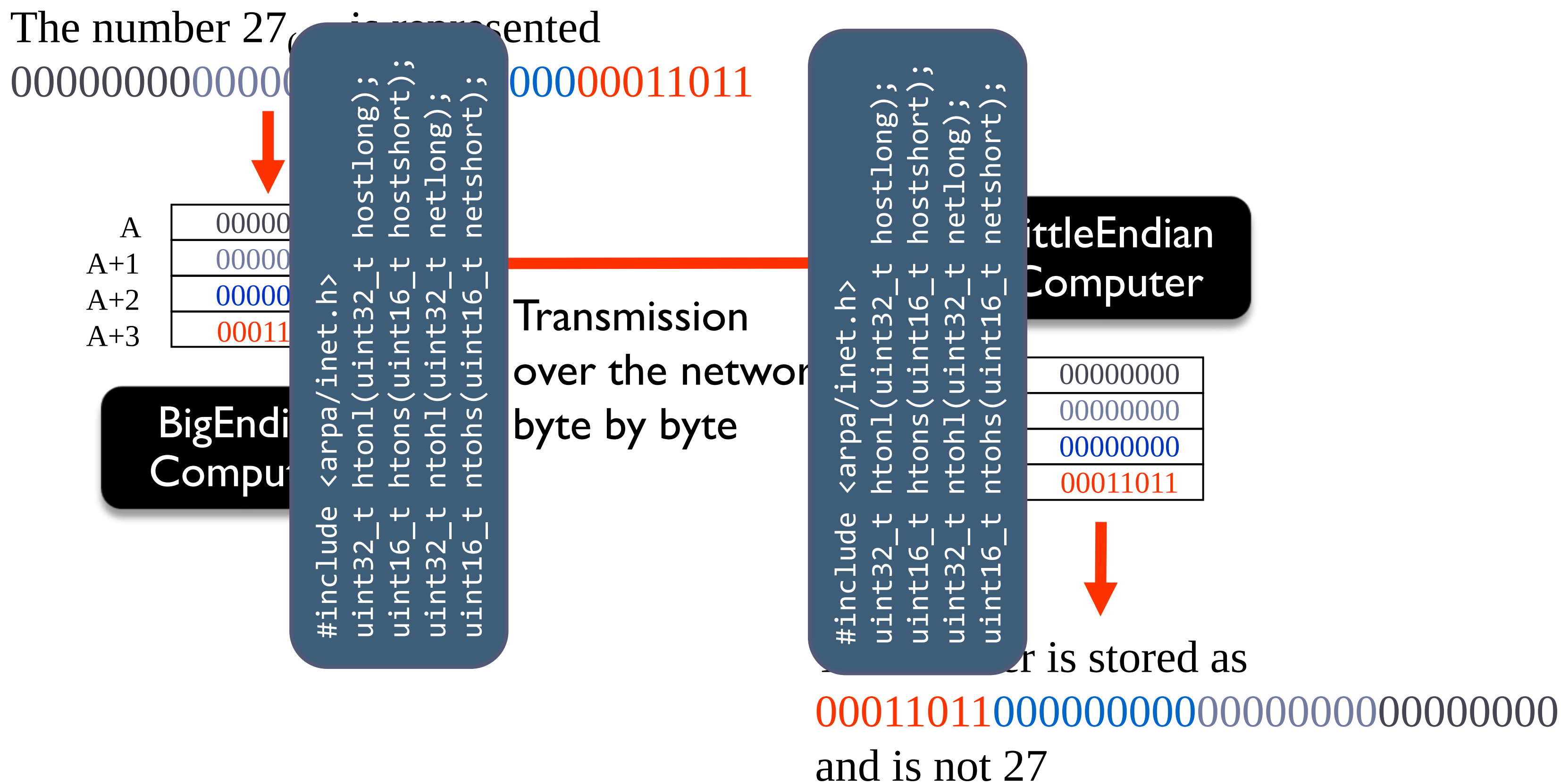
Transmission over the network byte by byte

LittleEndian Computer

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

The number is stored as
00011011000000000000000000000000
<u>and is not 27</u>

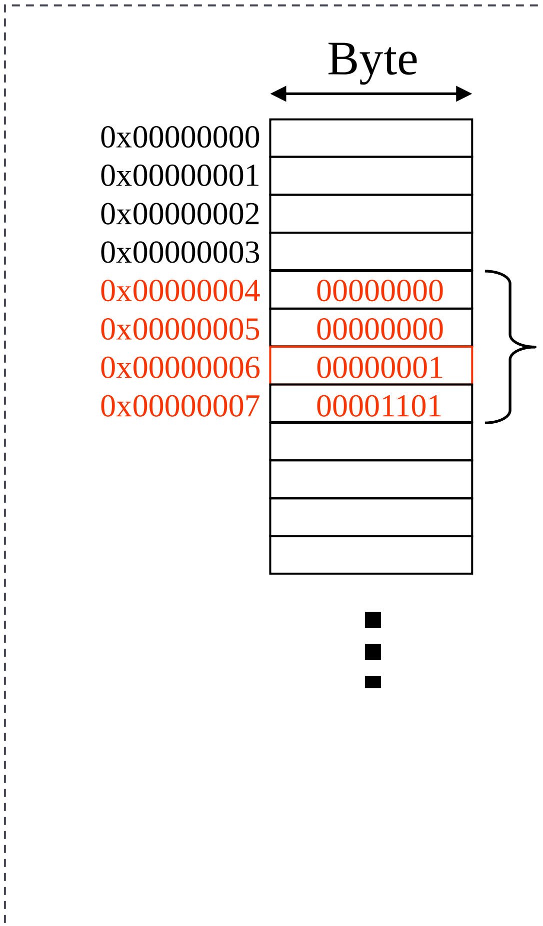
Communication problems in computers with different architectures



Access to words

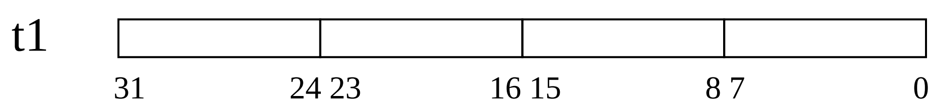
First byte address is specified: access to the stored word from address 0x00000004

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



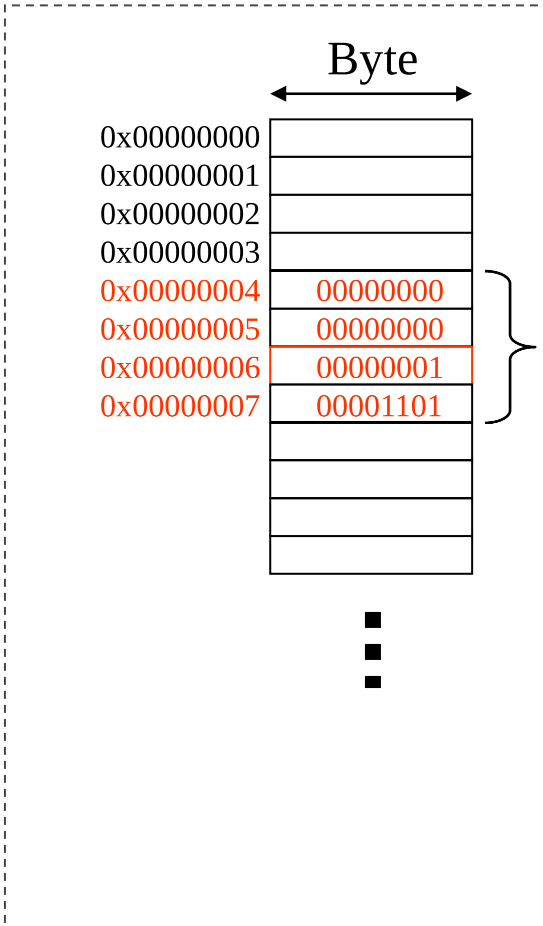
Address: 0x00000004 (000000…..00100)

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



Access to words

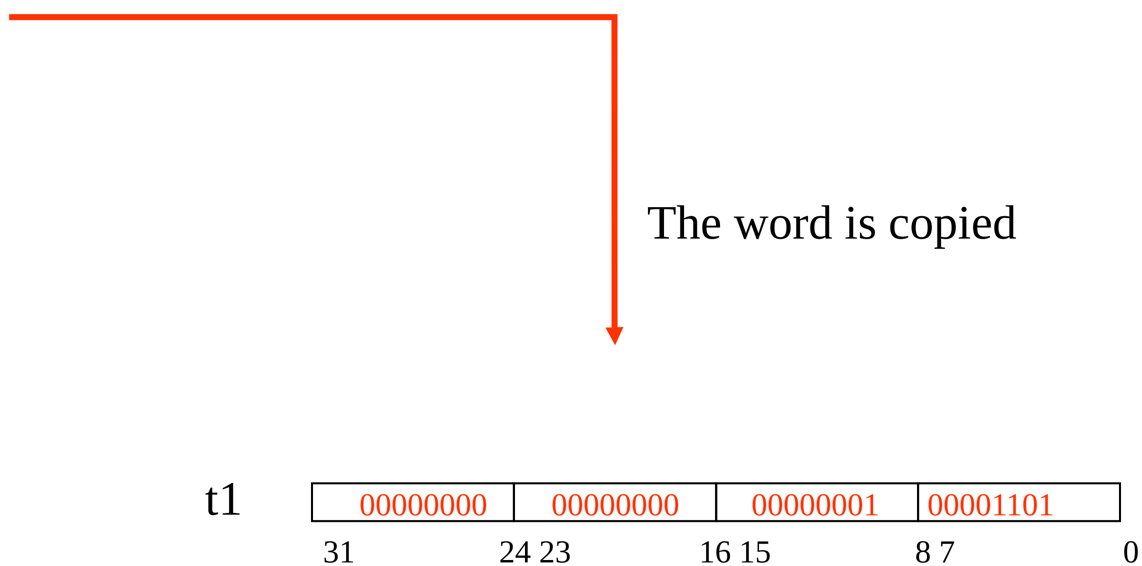
```
lw t1, 0x4(x0)
```



Address: 0x00000004 (000000…..00100)

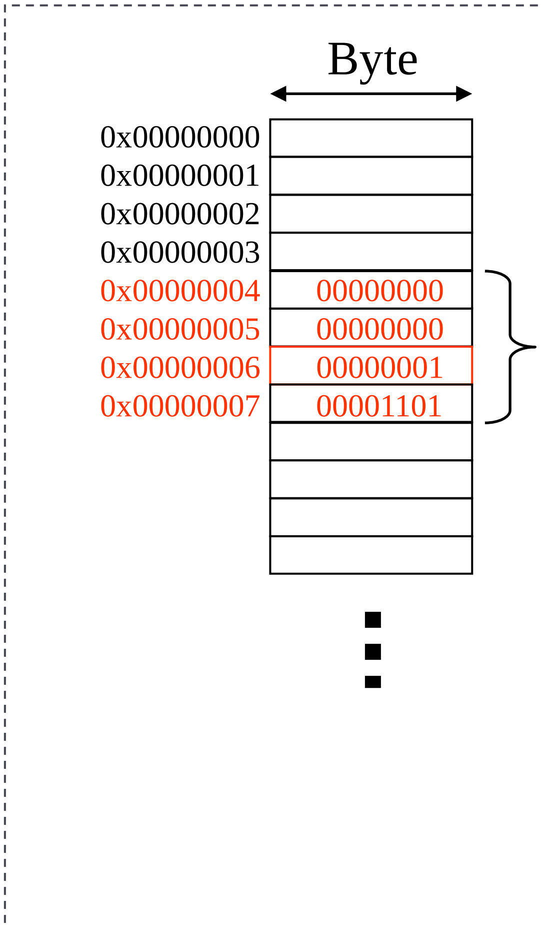
Content: $00000000000000000000000100001101_{(2} = 269_{(10}$

Computador Big Endian



Access to words

`lw t1, 0x4(x0)`

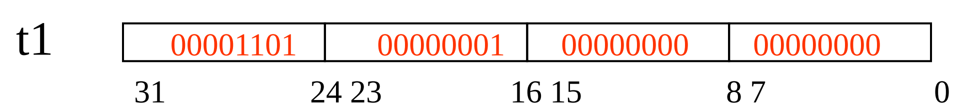


Address: 0x00000004 (000000…..00100)

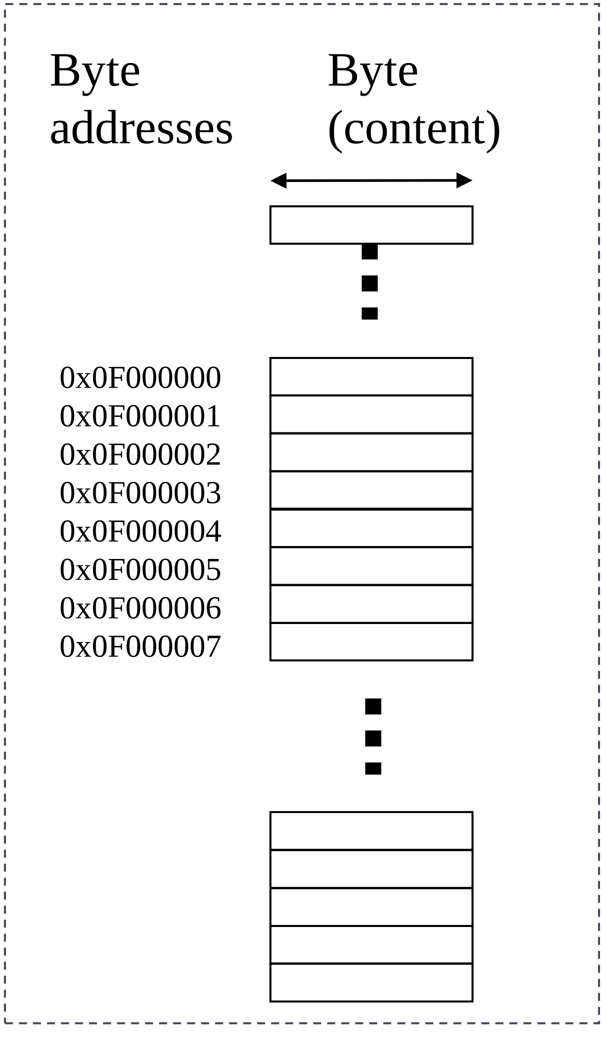
Content: $00000000000000000000000100001101_{(2} = 269_{(10}$

Computador Little Endian

The word is copied

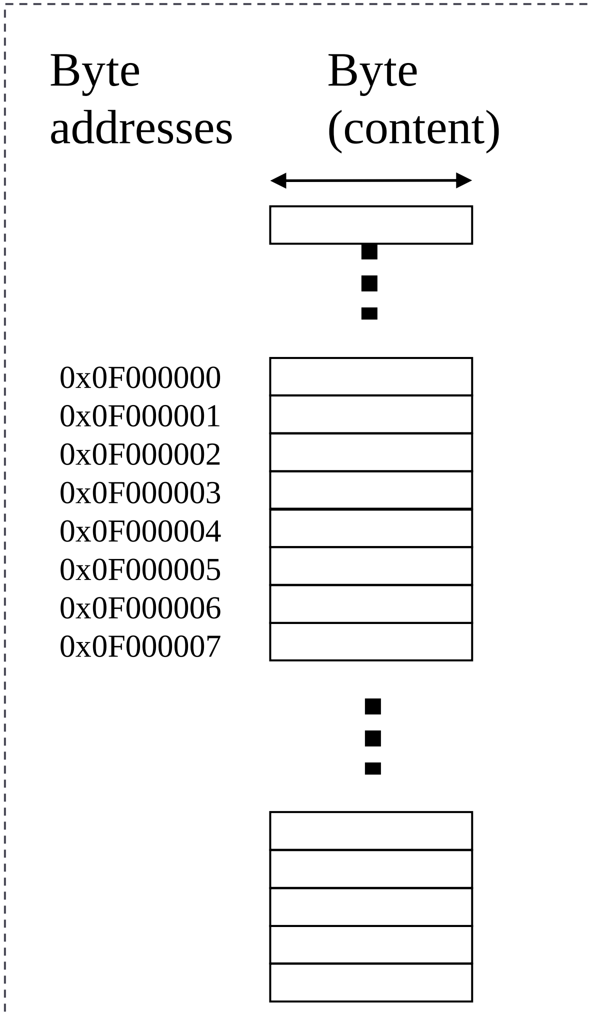


Writing to memory

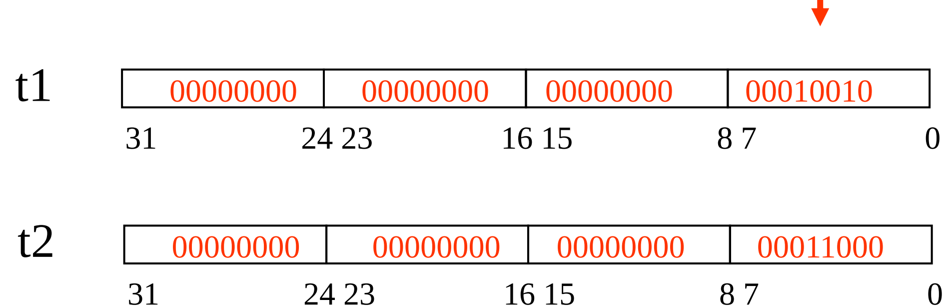



```
li t1, 18
li t2, 24
la t0, 0x0F000001
sb t1, 0(t0)
la t0, 0x0F000000
sw t1, 0(t0)
la t0, 0x0F000004
sw t2, 0(t0)
```

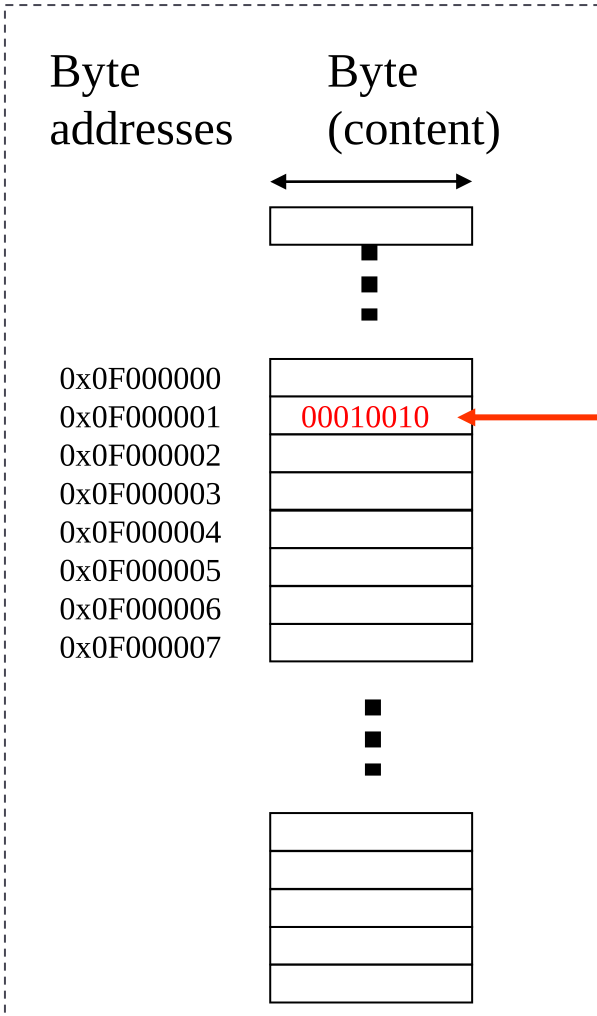
Writing to memory




```
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)
```

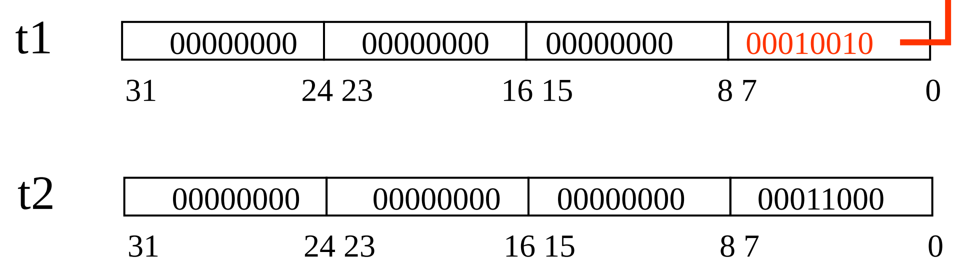


Writing to memory
write one byte

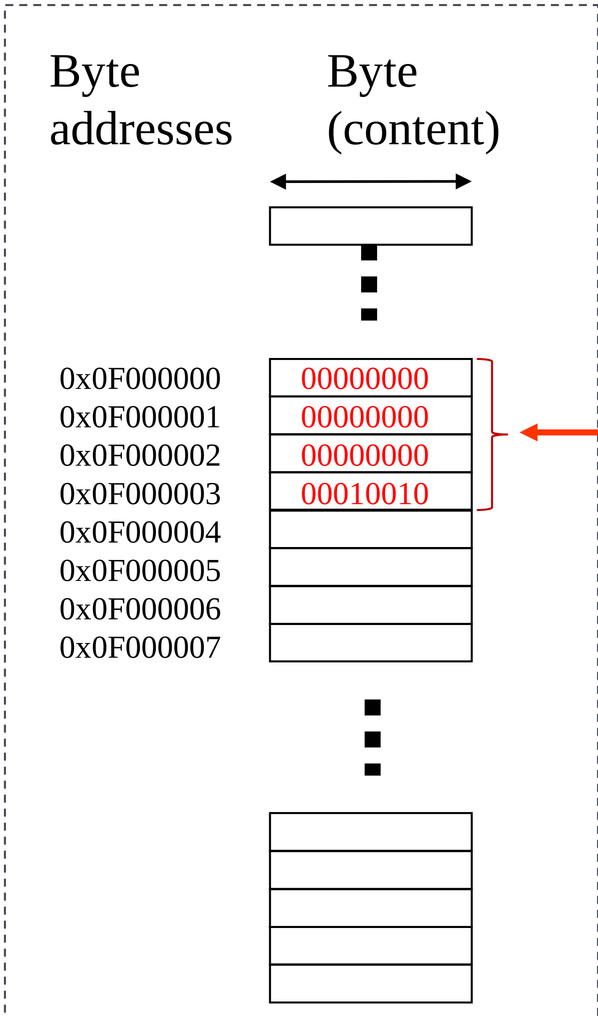



```
li t1, 18
li t2, 24
la t0, 0x0F000001
sb t1, 0(t0)
la t0, 0x0F000000
sw t1, 0(t0)
la t0, 0x0F000004
sw t2, 0(t0)
```

Writes the contents of the least significant byte of register t1 to memory

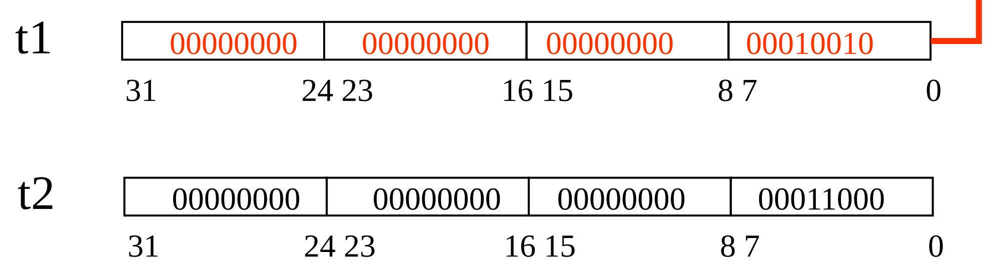


Writing to memory
write one word

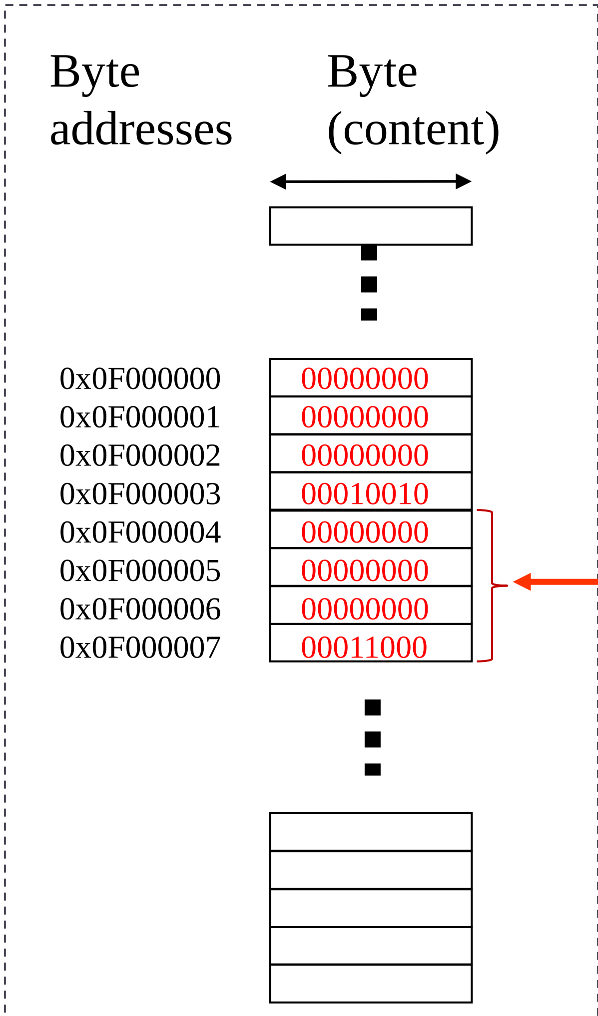



```
li t1, 18
li t2, 24
la t0, 0x0F000001
sb t1, 0(t0)
la t0, 0x0F000000
sw t1, 0(t0)
la t0, 0x0F000004
sw t2, 0(t0)
```

Writes the content of a record to memory (the complete word)

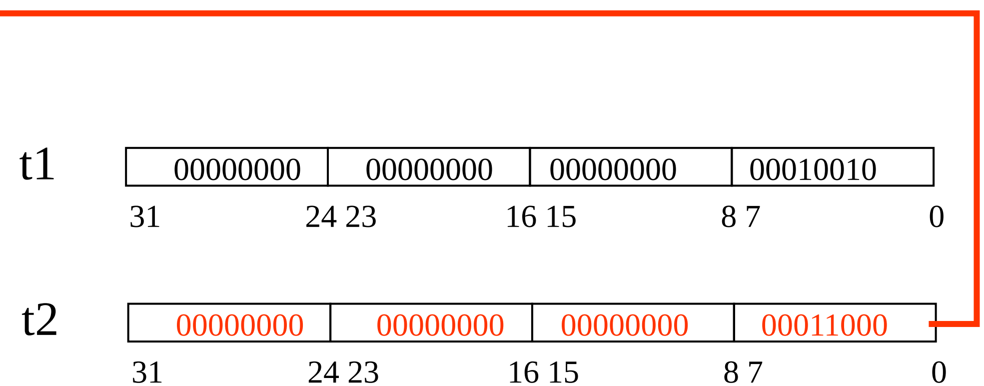


Writing to memory
write one word




```
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)
```

Writes the content of a record to memory (the complete word)



Review: memory access

- In order to be executed, a program must be loaded together with its data in memory.
  - All instructions and data are stored in memory.
  - Therefore, everything (instructions and data) has a memory address.
- In a 32-bits computer such as RISC-V 32:
  - Registers have 32 bits
  - Memory can store bytes (8 bits)
    - memory → register: `lb/lbu rd num(rs1)`
    - register → memory: `sb rd num(rs1)`
  - Memory can store words (32 bits)
    - memory → register: `lw rd num(rs1)`
    - Register → memory: `sw rd num(rs1)`

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

Examples of instructions

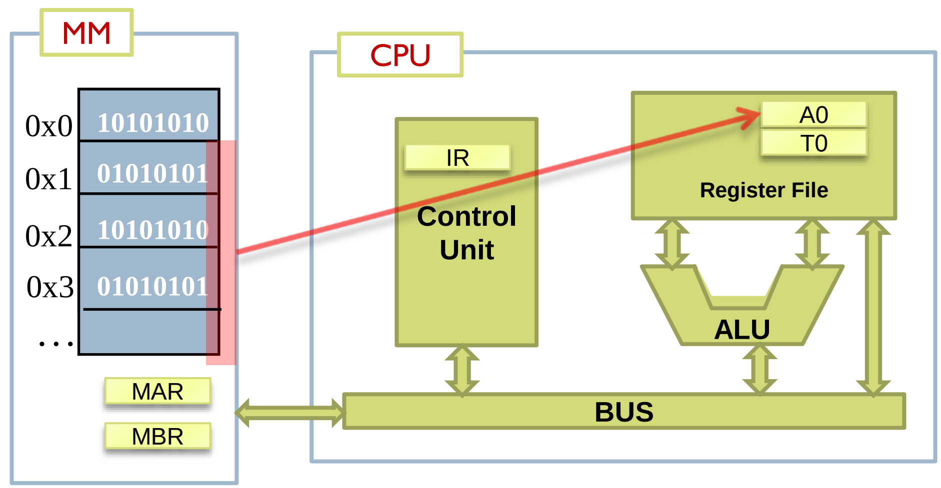
- `la t0, 0x0F000002`
  - The value `0x0F000002` is loaded at t0
- `lbu t0, label(x0)`
  - The byte at memory address `label` is loaded at t0.
- `lb  t0, 0(t1)`
  - The byte in the memory location stored in t1+0 is loaded in t0
- ```
  la  t0, 0x0F000000
  sw  t1, 0(t0)
  ```
  - Copies the word stored in t1 at address `0x0F000000`
- ```
  la t0, 0x0F000000
  sb t1, 0(t0)
  ```
  - Copies the byte stored in t1 (the least significant byte) to address `0x0F000000`

Memory operations: floating point

Memory access (floating point)		
`flw rs, 10(rs1)`	`rs = Memory[rs1+10]`	Loads the single precision value stored at address (rs1+10) into the floating-point register rd.
`fsw rs, 10(rs1)`	`Memory[rs1+10] = rs`	Stores the single value precision of the rs register at address (rd1+10).
`fld rd, 10(rs1)`	`rd = Memory[rs1+10]`	Loads the double precision value stored at address (rs1+10) into the rd register.
`fsd rd, 10(rs1)`	`Memory[rs1+10] = rd`	Stores the double precision value of the rs register at address (rd1+10).

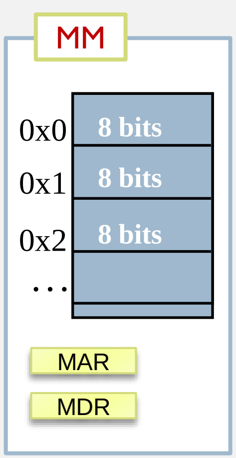
Data transfer
alignment and access size

- Peculiarities:
  - Default access size
  - Alignment of elements in memory



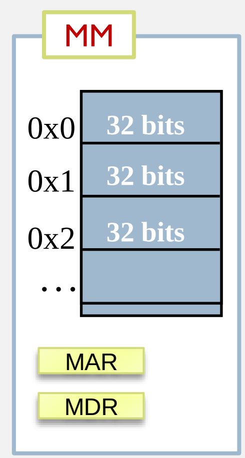
Word-level or byte-level addressing

- The main memory is like 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.

Data alignment

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

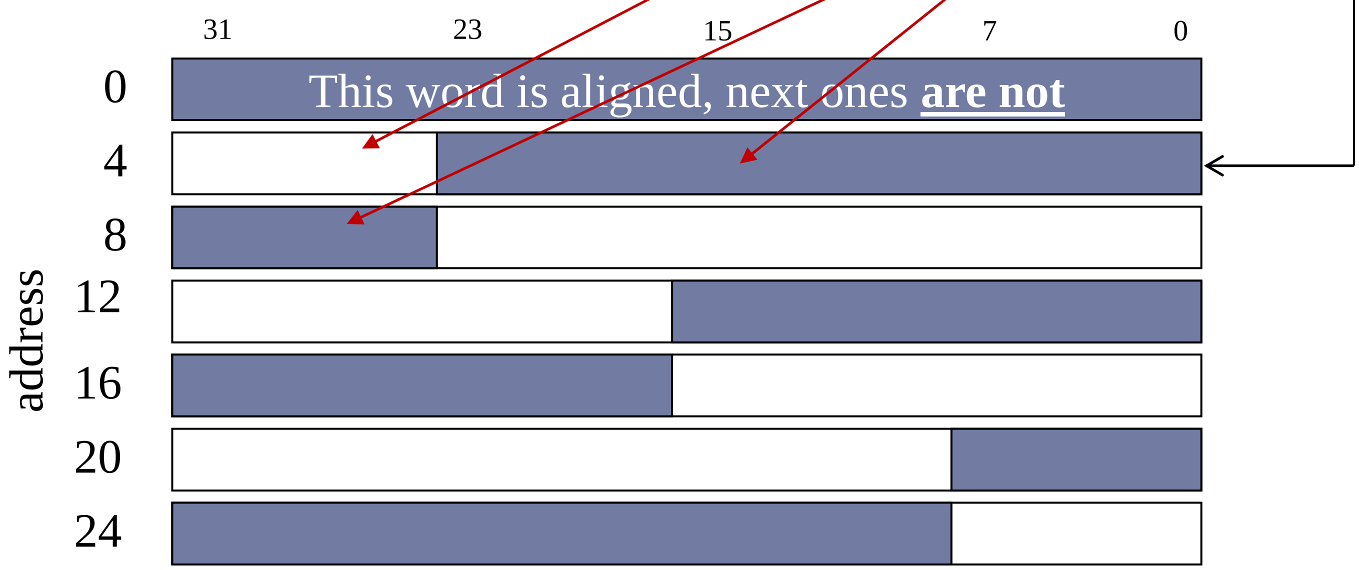
    D mod K = 0

- 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: example

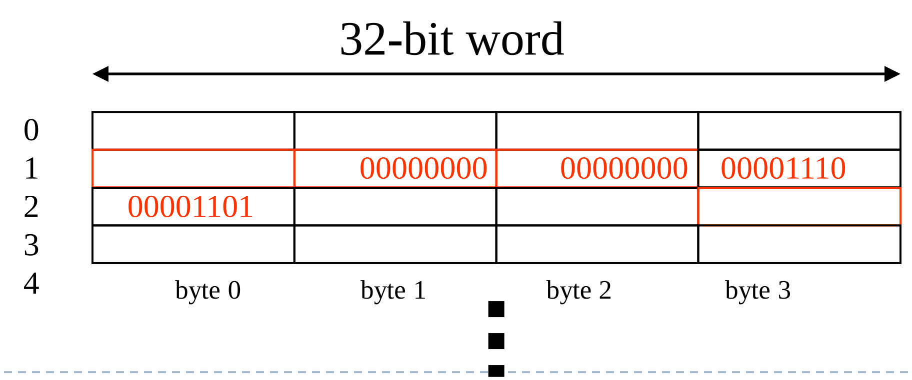
- The alignment requires the address to be a multiple of the word size:

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

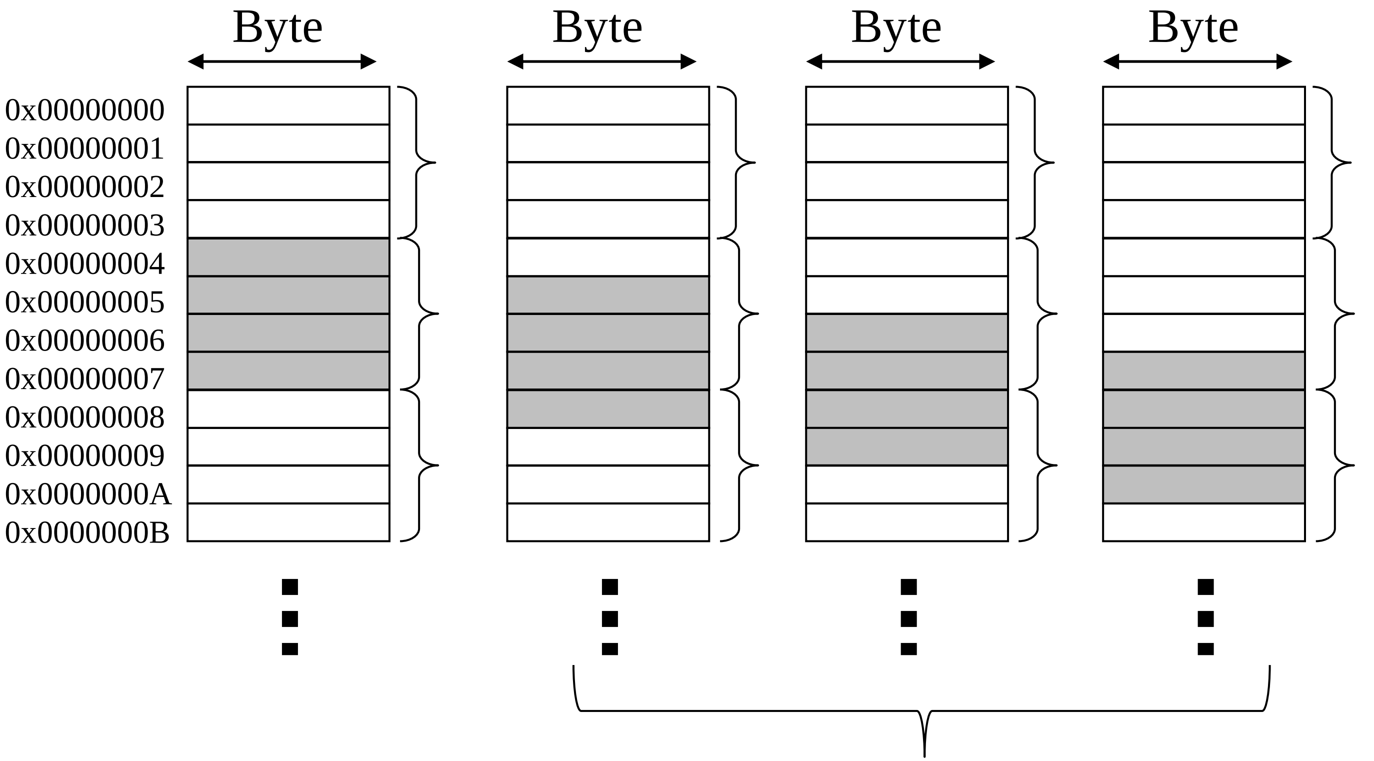


Data alignment

- 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



align words

non-align words

Must be stored from an address multiple of 4

Data types in assembly

- Elemental data types
  - Booleans
  - Character
  - Integer
  - Float and double
- Compound data types
  - Vector: consecutives elements of the same type indexed by position
    - String
  - Matrix: two dimensional indexed elements of the same type
  - Structs: consecutive elements of same/different types indexed by name

Elemental data types

Boolean

```
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)
       ...
```

Elemental data types
Character


```
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)
       ...
```

Elemental data types
Integer

```c
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 t3 0(t0)
           ...
```

Elemental data types

Integer

global variable without initial value

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

global variable with initial value

```
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 t3 0(t0)
          ...
```

Elemental data types

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)
       ...
```

CREATOR

Elemental data types

Double

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



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

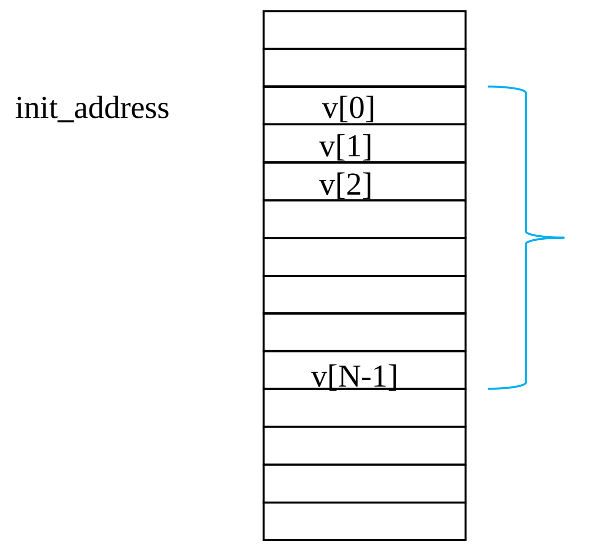
```
.data
.align 3
    resultado:   .zero 8
    op1:         .double 100
    op2:         .double -10.27
    ...

.text

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

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      # next item aligned to 4
  vec: .zero 20 # 5 elem.*4 bytes

.text
 main:
      la t1 vec
      li t2  8
      sw t2 12(t1)
      ...
```

Compound data types
Arrays

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

main ()
{
    vec[4] = 8;

}
```

```
.data
  .align   2      # next item aligned to 4
  vec: .zero 20 # 5 elem.*4 bytes

.text
 main:
          li   t0 12
          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: .zero 20  #5 elem.*4 bytes

.text
 main:
         li t2  8
         li t1  12
         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

CREATOR

WepSIM

```
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
           ...
```

Exercise

```
// global variables

char v1;
int v2 ;
float v3 = 3.14 ;

char v4 = "ec" ;

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

Exercise (solution)

CREATOR
WepSIM

```
// global variables

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
```

Exercise (solution)

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
```

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: .zero 4
v3: .float 3.14

v4: .string "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

CREATOR WepSIM

```
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   # pointer => address
...
.text
main:       la   t0, ac1
            li   a0, 0
            lbu  t1, 0(t0)
   buc1:    beq  x0, 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`

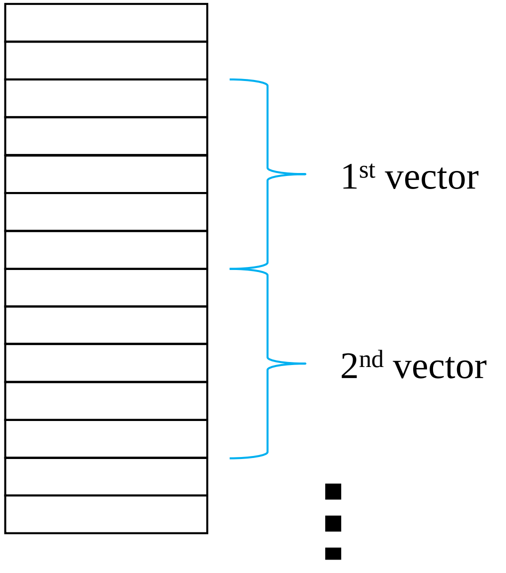
Félix García Carballeira, Alejandro Calderón Mateos

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

CREATOR

WepSIM

```
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 aligned to 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)
       ...
```

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.

ARCOS Group

uc3m | Universidad Carlos III de Madrid

L3: Fundamentals of assembler programming (2)
Computer Structure

Bachelor in Computer Science and Engineering
Bachelor in Applied Mathematics and Computing
Dual Bachelor in Computer Science and Engineering and Business Administration