

# 2 - Instruccions i tipus de dades

Estructura de Computadores - FIB



UNIVERSITAT POLITÈCNICA  
DE CATALUNYA  
BARCELONATECH

David Álvarez

# Introducció

- Per comandar (donar ordres) a una CPU, necessitem parlar el “seu llenguatge”.
- Aquest llenguatge es diu Joc d’Instruccions o ISA (Instruction Set Architecture). Descriu:
  - Les instruccions que una CPU pot entendre i executar
  - Els tipus de dades
  - Els registres
  - La organització de memòria
- Històricament han anat evolucionant des dels anys 50, on eren molt senzilles, fins ara.
- A EC utilitzarem la **MIPS ISA**, que és un RISC

# RISC i CISC

Un RISC correspon a un **Reduced Instruction Set Computer**. Totes les arquitectures RISC tenen unes característiques en comú:

- Load-store MIPS, ARM, RISC-V
- Mida fixa d'instruccions
- Poques instruccions diferents

Això ve en contraposició a CISC (Complex ISC), que solen ser diferents:

- Operands en memòria Intel x86, AMD64
- Mida variable d'instruccions
- Més instruccions i més complexes

# MIPS ISA

- MIPS és una ISA que va aparèixer al 1985 (com el Intel 80386).
- L'arquitectura és RISC de **32 bits**
- 32 registres
- Memòria unificada instruccions i dades (von Neumann)
- Actualment s'usa a embedded systems, encara que poc a poc perd terreny a RISC-V.

# Instruccions bàsiques

Quan tenim un programa que vol sumar les “variables” b i c i guardar el resultat en a.

```
addu a, b, c
```

Les operacions bàsiques de MIPS sempre tenen 3 operands: un de sortida i dos d’entrada. Per tant, si el que volem és sumar  $a = b + c + d + e$

```
addu a, b, c # a = b + c  
addu a, a, d # a = b + c + d  
addu a, a, e # a = b + c + d + e
```

El requisit de tenir sempre 3 operands és part del disseny de MIPS, i un dels principis: *Simplicity Favors Regularity*.

# Exemple

// Codi C	# Ensamblador
a = b + c;	addu a, b, c
d = a - e;	subu d, a, e
f = (g + h) - (i + j);	addu t0, g, h addu t1, i, j subu f, t0, t1

# Registres

- Fins ara hem parlat de “variables”, però en realitat a MIPS els operands de les operacions anteriors són registres.
- Els registres són ubicacions físiques al hardware, on cada un pot guardar 32 bits de dades.
- A MIPS tenim 32 registres (part del disseny, *Smaller is Faster*).
- Ens podem referir a cada registre amb el número o el nom, encara que usarem sempre els noms.

# Registres

Número	Nom	Descripció
\$0	\$zero	Valor 0, read-only
\$1	\$at	Reservat
\$2-\$3	\$v0-\$v1	Retorns de subrutines
\$4-\$7	\$a0-\$a3	Arguments de subrutines
<b>\$8-\$15</b>	<b>\$t0-\$t7</b>	<b>Valors temporals</b>
<b>\$16-\$23</b>	<b>\$s0-\$s7</b>	<b>Variables locals</b>
<b>\$24-\$25</b>	<b>\$t8-\$t9</b>	<b>Valors temporals</b>
\$26-\$27	\$k0-\$k1	Reservats
\$28	\$gp	Global Pointer
\$29	\$sp	Stack Pointer
\$30	\$fp	Frame Pointer
\$31	\$ra	Return Address

# Exemple

```
// Codi C (f=j-a $t0-$t4)          # Ensamblador  
  
f = (g + h) - (i + j);           addu $t0, $t1, $t2  
                                  addu $t1, $t3, $t4  
                                  subu $t0, $t0, $t1  
  
a = b;                          move  $t0, $t1 # o  
                                  addiu $t0, $t1, 0 # o  
                                  addu  $t0, $t1, $zero
```

# Instruccions fins ara

```
addu rd, rs, rt  
    rd = rs + rt
```

```
subu rd, rs, rt  
    rd = rs - rt
```

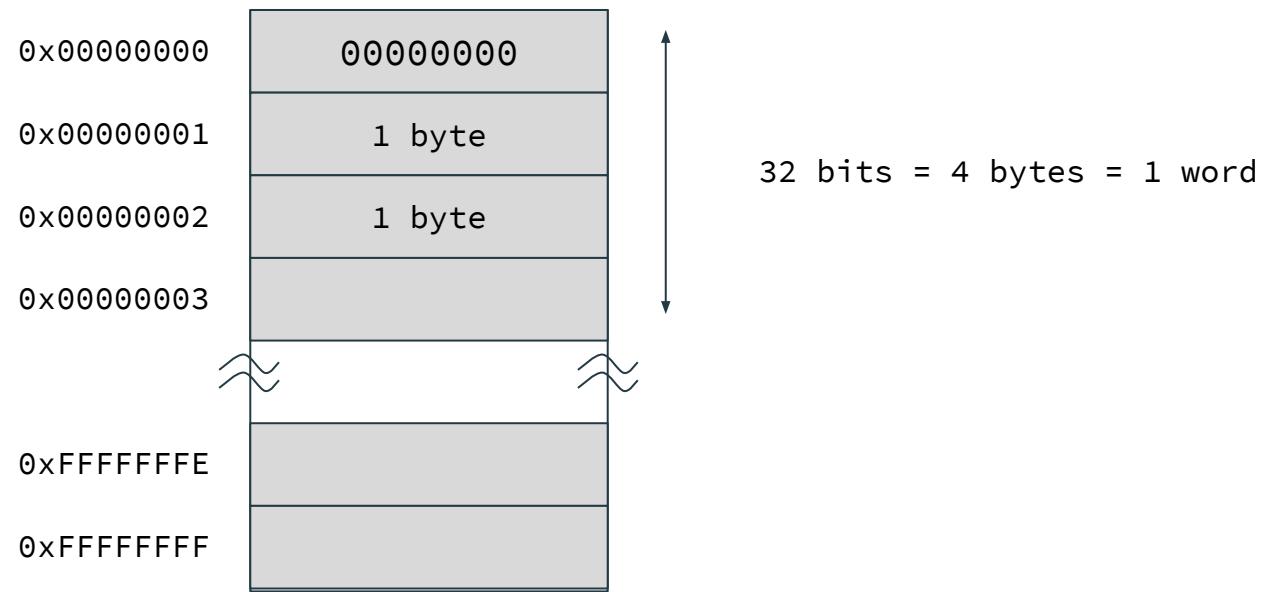
```
addiu rd, rs, imm16  
    rd = rs + imm16 (sign-extended)
```

```
movepseudo rd, rs  
    rd = rs
```

# Memòria

- Què fem quan no en tenim prou amb els 32 registres?
- Ens farem un “model” de memòria, i aprendrem a accedir-hi desde MIPS.

# Memòria



# Little endian vs Big endian

Volem guardar el word 0x12345678 a l'adreça 0x00010000

Little-endian

0x00010000	0x78
0x00010001	0x56
0x00010002	0x34
0x00010003	0x12

Big-endian

0x00010000	0x12
0x00010001	0x34
0x00010002	0x56
0x00010003	0x78

# Instruccions d'accés a memòria

Load Word

```
lw rd, offset(rs)  
rd = M[rs + offset]
```

Store Word

```
sw rt, offset(rs)  
M[rs + offset] = rt
```

# Exemple

// Codi C	# Ensamblador
f = A[0];	lw \$t0, 0(\$t1)
f = A[8];	lw \$t0, 32(\$t1)
A[1] = A[0] + f;	lw \$t2, 0(\$t1) addu \$t2, \$t2, \$t0 sw \$t2, 4(\$t1)

# Instruccions d'accés a memòria

lb rd, offset(rs)	rd = SignExtend(Mb[rs + offset])	Load byte
sb rt, offset(rs)	Mb[rs + offset] = rt<0..7>	Store byte
lbu rd, offset(rs)	rd = Mb[rs + offset]	Load byte unsigned

# Exemple

0x00010000	0x11
0x00010001	0x22
0x00010002	0xCC
0x00010003	0xDD

# \$t0 = 0x00010000

lb \$t1, 0(\$t0) # 0x00000011  
lb \$t1, 2(\$t0) # 0xFFFFFFFCC

lbu \$t1, 0(\$t0) # 0x00000011  
lbu \$t1, 2(\$t0) # 0x0000000CC

# Instruccions d'accés a memòria

lh rd, offset(rs)  
rd = SignExtend(Mh[rs + offset]) Load half-word

sh rt, offset(rs)  
Mh[rs + offset] = rt<0..15> Store half-word

lhu rd, offset(rs)  
rd = Mh[rs + offset] Load half-word unsigned

# Visibilitat de variables

A C, trobem dos tipus de variables:

- Locals
  - Declarades a dins d'una funció
  - Només duren mentre que s'executa la funció (fins al return)
  - Es guarden o bé a registres o bé a una part de la memòria que s'anomena **stack**
- Globals
  - Declarades fora de qualsevol funció, o a dins d'una funció amb el keyword *static*.
  - Duren fins al final del programa
  - Es guarden a la memòria, en la secció *.data*

# Variables globals (ensamblador)

```
unsigned char a;                                .data
short x = 13;
char b = -1, c = 10;
int y = 0x10AA00FF;
long long d = 0x7766554433221100;           a:    .byte 0
                                                x:    .half 13
                                                b:    .byte -1
                                                c:    .byte 10
                                                y:    .word 0x10AA00FF
                                                d:    .dword 0x7766554433221100
```

Tipus C	MIPS	Amplada
char	.byte	1 byte
short	.half	2 bytes
int / long	.word	4 bytes (1 word)
long long	.dword	8 bytes (2 words)

# Alineació

- Les instruccions `lw` i `sw` només poden carregar adreces múltiples de 4
- Les instruccions `lh`, `lhu` i `sw` només poden carregar adreces múltiples de 2
- Les instruccions `lb`, `lbu` i `sb` no tenen restriccions

# Alineació

```
.data  
x:    .word 0xDDCCAABB  
  
.text  
lw    $t1, 1($t0) # Excepció
```

# Alineació automàtica

```
.data  
a: .byte 0  
x: .half 13  
b: .byte -1  
c: .byte 10  
y: .word 0x10AA00FF  
d: .dword 0x7766554433221100
```

a:	00	0
x:	0D 00	2
b:	FF	4
c:	0A	5
y:	FF 00 AA 10	8
d:	00 11 22 33 44 55 66 77	16

# Eliminar alineació automàtica

```
.data  
.align 0 # Deshabilitar alineació automàtica  
x: .byte 0xFF  
y: .word 0xDDCCBBAA  
  
.text  
lw    $t1, 0($t0) # $t0 = @y
```

# Alineació explícita

```
# .align n = alinear següent directiva a 2^n

    .data
    .align 0
a:   .byte 0
      .align 1 # Múltiple de 2^1 = 2
x:   .half 13
b:   .byte -1
c:   .byte 10
      .align 2 # Múltiple de 2^2 = 4
y:   .word 0x10AA00FF
      .align 3 # Múltiple de 2^3 = 8
d:   .dword 0x7766554433221100
```

# Constants i immediats

- Constant 0: registre \$zero

```
addu $t1, $zero, $zero # $t1 = $zero
```

- Constant de 16 bits: addiu

```
addiu $t1, $zero, 4 # $t1 = 4
```

- Immediats de 32 bits: com els carreguem?

# Constants i immediats

```
lui rd, imm16  
    rd <31..16> = imm16  
    rd <15..0> = 0x0000
```

```
# 0x003D0900 -> $t0
```

```
lui $t0, 0x003D  
ori $t0, 0x0900
```

# Constants i immediats

```
lipseudo rd, imm32  
rd = imm32
```

# I les adreces?

```
lapseudo rd, addr  
rd = addr
```

# Exemple

```
// Codi C                                     # Ensamblador

int y = 0x10AA00FF; // Global                 .data
y = 0;                                         y:    .word 0x10AA00FF

                                                .text
                                                la    $t0, y
                                                sw    $zero, 0($t0)
```

# Representació de dades

# Representació de naturals

$x$  = valor natural

$X = X_{n-1}..N_1N_0$  = representació binària

$$N^n \rightarrow N$$

$$f : X \rightarrow x$$

s.t.

$$x = \sum_{i=0}^{n-1} X_i \cdot 2^i$$

Rang representable:  $0 \leq x \leq 2^n - 1$

# Representació de naturals

- Interpretar un número binari com un natural implica seguir la fòrmula anterior.
- Representar un número  $x$  com un natural:  $f^{-1}$   
Algorisme de divisions successives
- Extensió del nombre de bits d'un natural: afegir zeros a l'esquerra
- Declarar en C variables naturals:

Bits	C	MIPS
8	unsigned char	.byte
16	unsigned short	.half
32	unsigned int	.word
64	unsigned long long	.dword

# Representació enters

$x = \text{valor enter}$

$X = X_{n-1}..N_1N_0 = \text{representació binària}$

$$\mathbb{N}^n \rightarrow \mathbb{Z}$$

$$f : X \rightarrow x$$

*s.t.*

$$x = -X_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} X_i \cdot 2^i$$

Rang representable:  $-2^{n-1} \leq x \leq 2^{n-1} - 1$

# Representació enters

- Interpretar un número binari com un enter implica seguir la fòrmula anterior.
- El bit  $X_{n-1}$  determina el signe (si és 1,  $x < 0$ ).
- Declarar en C variables enteres:

Bits	C	MIPS
8	char	.byte
16	short	.half
32	int	.word
64	long long	.dword

# Representació enters

Pot ser útil calcular el valor explícit de un enter, que és el nombre natural que té la mateixa representació binària.

$x_e$  = valor explícit de  $x$

$$x_e = x + X_{n-1} \cdot 2^n$$

$$x = x_e - X_{n-1} \cdot 2^n$$

$$x_e = \begin{cases} x & x \geq 0 \\ x + 2^n & \text{otherwise.} \end{cases}$$

# Representació enters

- Calcular la representació de  $x$  en binari implica calcular el valor explícit i aplicar les divisions successives.
- Com canviem el signe d'un enter en binari?
- Com ampliem la quantitat de bits d'un enter?

# Canvi de signe

Regla útil: invertir tots els bits i sumar-li 1

$X = X_{n-1} \dots X_0$  representa  $x$

$\overline{X} = \overline{X_{n-1}} \dots \overline{X_0}$  representa  $\overline{x}$

$X + \overline{X} = x + \overline{x}$

$X + \overline{X} = 111 \dots 1_2 \rightarrow x + \overline{x} = -1$

$\overline{x} + 1 = -x$

# Extensió rang

Regla útil: afegir bits a l'esquerra iguals al bit de signe

$$x = -X_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} X_i \cdot 2^i$$

$$x' = -X_n \cdot 2^n + X_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} X_i \cdot 2^i$$

Si  $X_n = X_{n-1} = s$  llavors  $x = x'$

$$x' - x = -s \cdot 2^n + s \cdot 2^{n-1} + s \cdot 2^{n-1} = s \cdot (2^{n-1} + 2^{n-1} - 2^n) = 0$$

# Altres representacions

- Complement a 1

La representació dels negatius és exactament la mateixa que els positius amb tots els bits invertits. Té dues representacions per 0 (0 i “-0”), i calen més passos per calcular.

- Signe i Magnitud

Separar en un bit de signe i el valor absolut. S’usa només a coma flotant pel significand.

- En excés

Representem el negatiu més petit a  $000\dots0_2$  i el positiu més gran a  $111\dots1_2$ . Fa més fàcil la comparació, s’usa per l’exponent a coma flotant.

# ASCII

- Forma de codificar **caràcters** (lletres, números, símbols)
- American Standard Code for Information Interchange, 1963
- Cada caràcter té 7 bits (el 8è a 0).
- Existeixen altres codificacions: UTF-8, Unicode, UTF-16, etc.

# ASCII

# ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	'
1	1	1	1	[START OF HEADING]	49	31	1100001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	1100010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	1100011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	\$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(	88	58	1011000	130	X					
41	29	101001	51	)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[					
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135	I					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

# ASCII

## 1. Ordre alfabètic

- $'A' + 1 = 'B'$
- $'a' + 1 = 'b'$
- $'0' + 1 = '1'$

## 2. Conversió majúscules / minúscules

- $'a' - 'A' = 32$
- $'e' = 'E' + 32$

## 3. Número a caràcter

- $'3' = '0' + 3$

// Codi C

```
char variable = 'R';
```

# MIPS

```
variable: .byte 'R'
```

# Format de les instruccions

- **Stored-program:** igual que representem les dades, guardem les instruccions a memòria, amb una codificació especial
- El format de la instrucció determina la seva representació
- Idealment volem tenir la menor quantitat de formats diferents
- *Good design demands good compromises*

# Format de les instruccions

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

Instr	T	6 bits	5 bits	5 bits	5bits	5bits	6bits
addu rd, rs, rt	R	0x0	rs	rt	rd	0	0x21
sra rd, rt, shamt	R	0x0	rs	rt	rd	shamt	0x03
addiu rt, rs, imm	I	0x8	rs	rt	imm		
lui rt, imm	I	0xF	0	rt	imm		
lw rt, offset(rs)	I	0x23	rs	rt	offset		
j target	J	0x2	target				

# Punters

Un punter és una variable que conté una adreça de memòria (32 bits en MIPS).

Si un punter  $p$  conté l'adreça d'una variable  $v$ ,  $p$  apunta a  $v$ .

```
// Codi C                                # Ensamblador
int *p1;                               .data
char *p2;                               p1: .word 0
                                         p2: .word 0

p1 = p2 // Warning
```

# Punters: Inicialització

Inicialització global

```
// Codi C (variables globals)          # Ensamblador  
  
char a = 'E';                      .data  
char *p = &a;                      a:   .char 'E'  
                                         p:   .word a
```

# Punters: Inicialització

Inicialització en un programa

```
// Codi C (variables globals)          # Ensamblador  
  
int *p;                                la  $t0, v+32  
int v[100];                             la  $t1, p  
...  
p = &v[8];                            sw  $t0, 0($t1)
```

# Punters: desreferència

Escriure o llegir la variable apuntada per un punter és fer una **desreferència**. A C es fa amb l'operador \*

// Codi C	# Ensamblador
int i = 1;	la \$t1, p # \$t1 = &p
int *p = &i;	lw \$t1, 0(\$t1) # \$t1 = p
...	lw \$t0, 0(\$t1) # \$t0 = *p = i = 1
tmp = *p;	

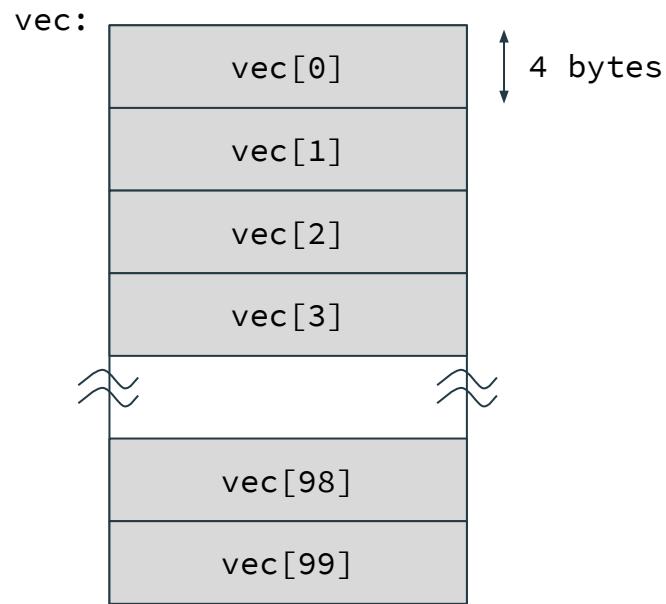
# Vectors

Un vector és una col·lecció d'elements del mateix tipus:

- Cada element es pot identificar per un índex
- S'emmagatzema de forma contigua a memòria

# Vectors: Declaració

```
// C  
int v1[100];  
int v2[5] = { 0, 1, 2, 3, 4 };  
  
# MIPS  
    .data  
v1: .space 400  
v2: .word 0, 1, 2, 3, 4
```



# Vectors: Accés aleatori

```
v[i] = &v[0] + i * MIDA_ELEMENT
```

```
// Codi C (int v[])
v[i] = 10;                                # Ensamblador ($t0 = i)
                                              la    $t2, v
                                              sll   $t1, $t0, 2    # $t1 = i * 4
                                              addu $t1, $t1, $t2 # $t1 = @v[i]
                                              li    $t2, 10
                                              sw    $t2, 0($t1)
```

# Vectors vs punters

A C un vector és un punter al primer element, per tant, podem combinar punters i vectors

```
int v[100];
int *p;
...
p = v;
p[8] = 10;
```

# Aritmètica de punters

A C, sumar un enter  $n$  a un punter implica “moure” el punter  $n$  elements. Per tant, s’ha de multiplicar l’enter per la mida d’un element.

// Codi C	# Ensamblador
char *p1;	
int *p2;	
long long *p3;	
p1 = p1 + 3;	addiu            \$t1, \$t1, 3
p2 = p2 + 3;	addiu            \$t2, \$t2, 12
p3 = p3 + 3;	addiu            \$t3, \$t3, 24

# Strings

Un string és un vector de caràcters

- En general, són de mida variable, per tant necessitem saber on acaben. Tenim 2 opcions:
  - Guardar la longitud del vector a la primera posició (Java)
  - Acabar l'string en un caràcter reservat (C)
- El caràcter per acabar un string a C és 0x00, que s'escriu '\0'

```
cadena = "UNO";
```

cadena:

‘U’
‘N’
‘O’
0x00

# Strings

En C, es declara una string com un vector de chars, amb inicialització opcional:

```
char cadena[20] = "Hola";
```

Equival a:

```
char cadena[20] = { 'H', 'o', 'l', 'a', '\0' };
```

I a MIPS:

```
.data  
cadena: .asciiz "Hola"  
.space 15
```

```
.data  
cadena: .ascii "Hola"  
.space 16
```

# Strings: accés

```
// Codi C                                     # Ensamblador

i = 0;
while ((x[i] = y[i]) != '\0')
    i++;

move $t0, $zero
la   $t1, x
la   $t2, y
while:
    addu $t3, $t0, $t2    # $t3 = @y[i]
    lb   $t3, 0($t3)      # $t3 = y[i]
    addu $t4, $t0, $t1    # $t4 = @x[i]
    sb   $t3, 0($t4)      # x[i] = y[i]
    beq  $t3, $zero, out  # Condició
    addiu $t0, $t0, 1      # i++;
    b    while              # Bucle
out:
```

## 2 - Instruccions i tipus de dades