

Tema 2. Instruccions i tipus de dades bàsics

Estructura de Computadors (EC)

Rubèn Tous

rtous@ac.upc.edu
Computer Architecture Department
Universitat Politècnica de Catalunya



UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Índex

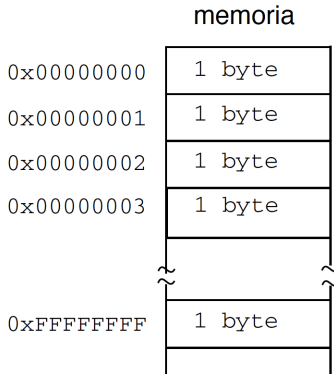
1 2.2 La memòria

2 2.3 Variables

3 2.4 Operands

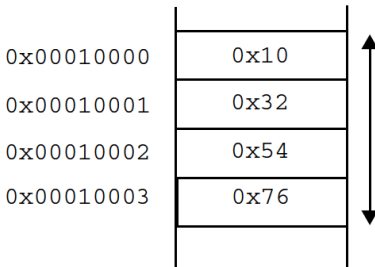
2.2.1 Adreçament a nivell de byte

- Memòria = vector en què cada element s'identifica per una adreça i conté 8 bits (un byte).
- 2^{32} bytes adreçables.



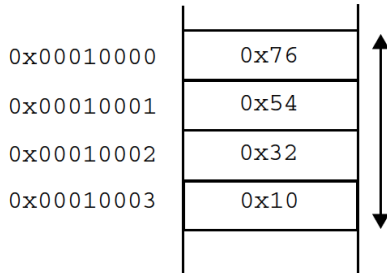
2.2.2 Ordenació de bytes (“byte ordering” o “endianness”)

- Little-endian: Coloquem primer (en l'adreça més baixa) el byte de MENYS pes.
- Exemple: guardem el número 0x76543210 (4 bytes) a l'adreça 0x00010000:



2.2.2 Ordenació de bytes (“byte ordering” o “endianness”)

- Big-endian: Coloquem primer (en l'adreça més baixa) el byte de MÉS pes.
- Exemple: guardem el número 0x76543210 (4 bytes) a l'adreça 0x00010000:



2.2.2 Ordenació de bytes (“byte ordering” o “endianness”)

- La nomenclatura prové de Els viatges de Gulliver de Jonathan Swift.
- Dos regnes s'enfronten per quina és la manera correcta de trencar un ou bullit.



BIG ENDIAN



LITTLE ENDIAN

2.2.2 Ordenació de bytes (“byte ordering” o “endianness”)

- MIPS no defineix una ordenació específica (little o big endian).
- A EC utilitzarem little-endian.
- Compte que el llibre de Patterson Hennesy utilitza big-endian!

2.3 Variables

Exemple C:

```
1  int g1, g2;  
2  void main() {  
3      int l1, l2;  
4      l1 = g1;  
5      ...  
6  }
```


2.3.1 Variables locals i globals

En llenguatge C:

- Tipus d'emmagatzematge: automatic, extern, static o register.
- Per defecte les declarades dins una funció són automatic (auto).
- Per defecte les declarades fora d'una funció són extern.

2.3.1 Variables locals i globals

En llenguatge C:

- Cada tipus d'emmagatzematge té una **visibilitat** i una **durada**.
- Visibilitats: local (dins la funció), global multi-file scope i global single-file scope.
- Durada: Fins al final subrutina, fins al final del programa.

2.3.1 Variables locals i globals

En llenguatge C:

- Les auto tenen visibilitat local i durada fins al final de la subrutina.
- Les extern tenen visibilitat global multi-file scope i durada fins al final del programa.

2.3.1 Variables locals i globals

- El modificador **static** aplicat a una variable local amplia la seva durada fins al final del programa.
- El modificador **static** aplicat a una variable global restringeix la seva visibilitat a global single-file.
- NO FAREM SERVIR CAP MODIFICADOR.

2.3.1 Variables locals i globals

- A les auto les anomenarem **locals** i quan sigui possible les emmagatzemarem en registres.
- A les extern les anomenarem **globals** i les emmagatzemarem a la memòria.

2.3.1 Variables locals i globals

Exemple C:

```
1  int g1, g2;  
2  void main() {  
3      int l1, l2;  
4      l1 = g1;  
5      ...  
6  }
```

Exemple MIPS:

```
1  .data  
2  g1: .word 0  
3  g2: .word 0  
4  .text  
5  .globl main  
6  main:  
7      la $t0, g1  
8      lw $t1, 0($t0) #l1 a $t1  
9      ...
```

2.3.2 Mida de les variables

- En C, la mida en bytes que ocupa una variable depèn de cada ISA on s'implementa.
- A EC farem les següents suposicions per als tipus enters:
 - char \rightarrow 1 byte
 - short \rightarrow 2 bytes
 - int \rightarrow 4 bytes
 - long long \rightarrow 8 bytes
- El mateix per als naturals (precedits per unsigned).

2.3.3 Declaració i alineació de variables emmagatzemades en memòria

Exemple C:

```
1  int a = 1;  
2  unsigned int b;  
3  char c, d;  
4  
5  void main() {  
6      ...  
7  }
```


2.3.3 Declaració i alineació de variables emmagatzemades en memòria

- Les directives `.byte`, `.half`, `.word` i `.dword` reserven 1, 2, 4 o 8 bytes:
 - `char` → `.byte`
 - `short` → `.half`
 - `int` → `.word`
 - `long long` → `.dword`
- A continuació s'escriu el valor inicial de la variable (o 0).

2.3.3 Declaració i alineació de variables emmagatzemades en memòria

```
1 char c = 0x11;  
2 short s = 0x2211;  
3 int i = 0x44332211;  
4 unsigned int ui;  
5 long long l = 0x8877665544332211
```

```
1 .data  
2 c: .byte 0x11  
3 s: .half 0x2211  
4 i: .word 0x44332211  
5 ui: .word 0  
6 l: .dword 0x8877665544332211
```

2.3.3 Declaració i alineació de variables emmagatzemades en memòria

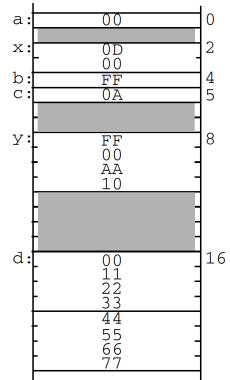
- **Alineació:** L'adreça inicial de la variable ha de ser múltiple de la grandària de la variable:
 - .byte → -
 - .half → múltiple de 2.
 - .word → múltiple de 4.
 - .dword → múltiple de 8.

2.3.3 Declaració i alineació de variables emmagatzemades en memòria

- **Alineació automàtica:** per defecte les variables globals s'ubiquen en adreces múltiples de la seva mida.
- Les directives `.half`, `.word` i `.dword` produeixen l'alineació correcta.

2.3.3 Declaració i alineació de variables emmagatzemades en memòria

```
1 a: .byte 0
2 x: .half 13
3 b: .byte -1
4 c: .byte 10
5 y: .word 0x10AA00FF
6 d: .dword 0x7766554433221100
```



2.3.3 Declaració i alineació de variables emmagatzemades en memòria

.byte/ .half/ .word/ .dword directives	
.byte n	Reserva 1 byte i l'inicialitza a n
.half n	Reserva 1 halfword alineat en una adreça múltiple de 2 i l'inicialitza a n
.word n	Reserva 1 word alineat en una adreça múltiple de 4 i l'inicialitza a n
.dword n	Reserva 1 doubleword alineat en una adreça múltiple de 8 i l'inicialitza a n

2.3.3 Declaració i alineació de variables emmagatzemades en memòria

Vectors?

```
1 int v = {1, 2, 3, 4, 5};
```

```
1 v: .word 1, 2, 3, 4, 5
```

2.3.3 Declaració i alineació de variables emmagatzemades en memòria

En cas que...

```
1 char v[100];
```

- La directiva **.space n** reserva n bytes de memòria i els inicialitza a zero.

```
1 v: .space 100
```


2.3.3 Declaració i alineació de variables emmagatzemades en memòria

```
1  int v[100];
```

```
1  v: .space 100*4
```

2.3.3 Declaració i alineació de variables emmagatzemades en memòria

```
1 char c;  
2 int v[100];
```

- Alineació? .space no pot alinear!

2.3.3 Declaració i alineació de variables emmagatzemades en memòria

Alineacions explícites amb **.align n** (propera adreça múltiple de 2^n).

.align (assembler directive)	
.align n	Ubica la propera dada a una adreça múltiple de 2^n

2.3.3 Declaració i alineació de variables emmagatzemades en memòria

```
1 char c;  
2 int v[100];
```

```
1 c: .byte 0  
2   .align 2  
3 v: .space 100*4
```

2.3.3 Declaració i alineació de variables emmagatzemades en memòria

.align		directiva
.align	n	Ubica la següent dada en una adreça múltiple de 2^n (ha de ser $n > 0$)
.space	m	Reserva m bytes i els inicialitza a zero

2.4 Operands

- Majoria d'instruccions amb 3 operands.
- Principi de Disseny 1: Simplicity favors regularity.
- La manera com s'especifica un operand és el **mode d'adreçament**.
- 5 modes d'adreçament: registre, immediat, memòria, pseudodirecte i relatiu al PC.

```
1 addu $t1 , $t2 , $t3 # $t1 = $t2 + $t3
```

2.4 Operands

Amb immediat:

```
1 addiu $t1, $t2, 10 # $t1 = $t2 + 10
```

2.4.1 Operands en mode registre i immediat

- Instruccions de suma i resta:

addu/addiu/subu		
addu rd, rs, rt	$rd = rs + rt$	
addiu rt, rs, imm16	$rt = rs + \text{SignExt}(\text{imm16})$	Immediat de 16 bits en Ca2
subu rd, rs, rt	$rd = rs - rt$	

2.4.1 Operands en mode registre i immediat

- 32 registres de 32 bits cadascun per a propòsit general.
- Dues maneres d'usarlos: pel número o pel nom.

Número	Nom	Utilització
\$0	\$zero	Sempre val zero, no modificable
\$1	\$at	Reservat a l'expansió de macros
\$2-\$3	\$v0-\$v1	Resultat de subrutines (sols usarem \$v0)
\$4-\$7	\$a0-\$a3	Arguments de subrutines
\$8-\$15	\$t0-\$t7	Temporals no preservats per les crides a subrutines
\$16-\$23	\$s0-\$s7	Temporals preservats per les crides a subrutines
\$24-\$25	\$t8-\$t9	Temporals no preservats per les crides a subrutines
\$26-\$27	\$k0-\$k1	Reservats per al SO (convé no usar-los)
\$28	\$gp	Global pointer, no l'estudiem
\$29	\$sp	Stack pointer, conté l'adreça del cim de la pila
\$30	\$fp	Frame pointer, no l'estudiem
\$31	\$ra	Return address, adreça de retorn de subrutina

2.4.1 Operands en mode registre i immediat

Exemple en C:

```
1 void main() {  
2     int f, g, h, i, j;  
3     ...  
4     f = (g + h) - (i + j);  
5     ...
```

En MIPS, suposant que f, g, h, i, j ocupen \$t0, \$t1, \$t2, \$t3, \$t4:

```
1 addu $t0, $t1, $t2 # $t0 = g + h  
2 addu $t1, $t3, $t4 # $t1 = i + j  
3 subu $t0, $t0, $t1 # f = (g + h) - (i + j)
```

2.4.1 Operands en mode registre i immediat

Còpia entre registres:

```
1 addiu $t1, $t2, 0 # $t1 = $t2
```

Alternativa:

```
1 addu $t1, $zero, $t2 # $t1 = $t2
```

Alternativa (amb pseudoinstrucció):

```
1 move $t1, $t2 # Equival a addu $t1, $zero, $t2
```

2.4.1 Operands en mode registre i immediat

- Macros o pseudoinstruccions.
- S'expandeixen en una o més instruccions MIPS.
- Faciliten la lectura i depuració.

move (pseudoinstrucció o macro)		
move rdest, rsrc	rdest = rsrc	addu rdest, rsrc, \$zero

2.4.2 Operands literals i constants simbòliques

- Algunes instruccions admeten operands en mode immediat (literal de 16 bits).
- Això ens permet assignar un literal a un registre fent per exemple:

```
1 addiu $t1, $zero, 100
```

2.4.2 Operands literals i constants simbòliques

Però si l'operand és de 32 bits (per exemple una adreça) podem fer servir la macro **li**.

```
1 li $t1, 0x44332211 # lui $at, 0x4433  
2                      # ori $t1, $at, 0x2211
```

També serveix per operands petits:

```
1 li $t1, 0x2211 # addiu $t1, $zero, $0x2211
```

2.4.2 Operands literals i constants simbòliques

Compte amb estalviar dígit, el Mars treballa amb literals de 32 bits:

```
1 li $t1 , 0xFFFF #El Mars llegeix 0x0000FFFF
```

2.4.2 Operands literals i constants simbòliques

li (pseudoinstrucció o macro)		
li rdest, imm16	rdest = SignExt(imm16)	addiu rdest, \$zero, imm16
li rdest, imm32	rdest = imm32	lui \$at, hi(imm32) ori rdest, \$at, lo(imm32)

2.4.2 Operands literals i constants simbòliques

Adreces. La macro **la** (load address):

```
1  .data
2  a:.word 0
3  .text
4  .globl main
5  main:
6  la    $t0, a # $t0 = &a = 0x10010000
7  ...
```

2.4.2 Operands literals i constants simbòliques

la (pseudoinstrucció o macro)		
la rdest, address	rdest = address	lui \$at, hi(address) ori rdest, \$at, lo(address)

2.4.2 Operands literals i constants simbòliques

Constants simbòliques:

```
1 #define N 10
```

```
1    .eqv N 10  
2    .data  
3 a: .word 0  
4    .text  
5    .globl main  
6 main:  
7    li $t0, N  
8    ...
```

2.4.3 Operands en memòria. Load i store

Exemple:

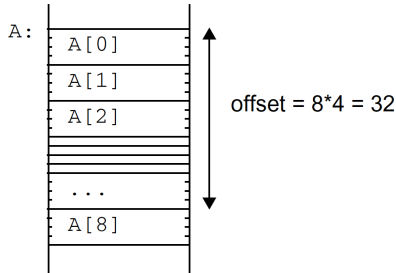
```
1  int a;  
2  void main() {  
3      int b; //b a $t1  
4      b = a;  
5      a = b;  
6  }
```

```
1  .data  
2  a:.word 0  
3  .text  
4  .globl main  
5  main:  
6      la $t0, a  
7      lw $t1, 0($t0)  
8      sw $t1, 0($t0)  
9      ...
```

2.4.3 Operands en memòria. Load i store

Exemple lectura de memòria:

```
1  int h[100];  
2  void main() {  
3      int g, h; //a $t1 i $t2.  
4      g = h + A[8];  
5  }
```



2.4.3 Operands en memòria. Load i store

Exemple lectura de memòria:

```
1  int A[100];  
2  void main() {  
3      int g, h; //a $t1 i $t2.  
4      g = h + A[8];  
5  }
```

```
1  la    $t3, A           # $t3 = adr. base del vector  
2  lw    $t0, 32($t3)     # $t0 = A[8]. Offset 32  
    bytes = 8*4  
3  addu  $t1, $t2, $t0    # g = h + A[8];
```

2.4.3 Operands en memòria. Load i store

Exemple escriptura de memòria:

```
1  int A[100];  
2  void main() {  
3      int h; //a $t2.  
4      A[12] = h + A[8];  
5  }
```

```
1  la    $t3, A           # $t3 = adr. base del vector  
2  lw    $t0, 32($t3)     # $t0 = A[8] (offset = 32 = 8*4)  
3  addu  $t0, $t2, $t0    # $t0 = h + A[8]  
4  sw    $t0, 48($t3)     # A[12] = h + A[8] (offset = 48  
    = 12*4)
```

2.4.3 Operands en memòria. Load i store

Instruccions Load Byte, Load Byte Unsigned i Store Byte: lb, lbu, sb.

```
1 lb    $t1, 0($t2) # $t1 = 0x00000011
2 lb    $t1, 2($t2) # $t1 = 0xFFFFFCC
3 lbu   $t1, 0($t2) # $t1 = 0x00000011
4 lbu   $t1, 2($t2) # $t1 = 0x000000CC
```

Existeixen instruccions anàlogues per accedir a dades “half” de 16 bits: lh, lhu, sh.

2.4.3 Operands en memòria. Load i store

WARNING

La instrucció lb extén signe. La instrucció lbu extén zeros.

El mateix passarà amb lh i lhu.

2.4.3 Operands en memòria. Load i store

lw/lh/lhu/lb/lbu/sw/sh/sb			
lw	rt, off16(rs)	$rt = M_w[rs + \text{SignExt}(\text{off16})]$	load word
lh	rt, off16(rs)	$rt = \text{SignExt}(M_h[rs + \text{SignExt}(\text{off16})])$	load half (extén signe)
lhu	rt, off16(rs)	$rt = M_h[rs + \text{SignExt}(\text{off16})]$	load half (extén zeros)
lb	rt, off16(rs)	$rt = \text{SignExt}(M_b[rs + \text{SignExt}(\text{off16})])$	load byte (extén signe)
lbu	rt, off16(rs)	$rt = M_b[rs + \text{SignExt}(\text{off16})]$	load byte (extén zeros)
sw	rt, off16(rs)	$M_w[rs + \text{SignExt}(\text{off16})] = rt$	store word
sh	rt, off16(rs)	$M_h[rs + \text{SignExt}(\text{off16})] = rt_{15:0}$	store half
sb	rt, off16(rs)	$M_b[rs + \text{SignExt}(\text{off16})] = rt_{7:0}$	store byte

2.4.3 Operands en memòria. Load i store

- **Alineació:** les adreces utilitzades en les instruccions de lectura/escriptura de memòria han de ser múltiple del número de bytes que es vulgui llegir/escriure.
- Per exemple, a les instruccions lw i sw han de ser múltiples de quatre. A lh i sh múltiples de 2. A lb i sb no hi ha restricció.

2.4.3 Operands en memòria. Load i store

Exemple:

```
1  .data  
2  x : .word 0xDDCCAABB  
3  .text  
4  lw $t1, 1($t0) # $t1 = M[$t0 + 1]
```

Provoca excepció d'accés no alineat!