

Tema 2. Instruccions i tipus de dades bàsics

Joan Manuel Parcerisa



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona



Instruccions i tipus de dades bàsics

- Introducció al MIPS
- Operands: en registre, immediats, o en memòria
- Sistemes de representació binària
 - Naturals i Enters (repàs)
 - Caràcters
 - Instruccions MIPS
- Vectors i punters

Introducció: RISC vs CISC

- CISC (Complex Instruction Set Computer)
 - Joc d'instruccions gran i complex
 - Instruccions de mida i nombre d'operands variable
 - Múltiples formats d'instrucció i modes d'adreçament
 - Difícil d'optimitzar les operacions més simples i freqüents
 - Exemple: x86
- RISC (Reduced Instruction Set Computer)
 - Poques instruccions
 - Mida fixa i pocs operands
 - Formats d'instrucció i modes d'adreçament simples
 - Lema: "make the common case fast"
 - Exemple: MIPS, ARM, RISC-V, PowerPC...
- Actualment s'han difuminat les diferències
 - Les CPU x86 (PentiumPro, 1995) descomponen internament les instruccions complexes en microoperacions *tipus-RISC*
 - Els repertoris dels RISC han crescut igual que els CISC

El processador MIPS

- **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages
 - Dissenyat per J.L. Hennessy a Stanford, 1981
 - Tipus RISC, usat en múltiples dispositius
 - Routers, Impressores, Playstation (PS2, PSP)
 - Àmpliament usat en docència
- Successives versions del ISA de MIPS
 - En EC estudiarem la versió MIPS32

El ISA MIPS32

- MIPS32
 - 32 registres de 32 bits cadascun
 - Paraules (words) de 32 bits (4 bytes)
 - Normalment *word* = "dada de la mida dels registres"
 - Instruccions i adreces de 32 bits
 - Memòria: 2^{32} bytes adreçables (1 byte per adreça)
- Com emmagatzema en memòria números multi-byte?
 - Little-endian o big-endian, MIPS ho decideix durant l'arrancada
 - En EC assumirem sempre **little-endian**. Per exemple: guardar el word **0x76543210** a l'adreça 0x10010000

adreça	
0x10010000	0x10
0x10010001	0x32
0x10010002	0x54
0x10010003	0x76

Declaració de variables

- En assemblador MIPS
 - NO hi ha declaracions de variables
 - Hi ha *etiquetes* (que identifiquen adreces de memòria)
 - Hi ha *directives* que *reserven* X bytes, i els *inicialitzen*
- En C
 - Hi ha *declaracions* de variables, on s'especifica el seu *tipus*
 - La *mida* en bytes de cada *tipus* depèn de l'ISA on s'implementa
 - En EC assumirem les següents mides:

Tipus de variable en C	Directiva de reserva d'espai en MIPS	Mida en bytes
char / unsigned char	.byte	1
short / unsigned short	.half	2
int / unsigned int	.word	4
long long / unsigned long long	.dword	8

Quan les operem, assumim que són *naturals*

Quan les operem, assumim que són *enters* en Ca2

Variables globals i locals

- Exemple en C

```
int g1, g2;

void main() {
    int loc1, loc2;
    loc1 = g1;
    ...
}
```

- Tipus de variables en C: *auto*, *extern*, *static*, *register*
- Per defecte, les declarades FORA d'una funció són *extern*
 - Accessibles des de qualsevol funció
 - Ocupen una adreça fixa de memòria durant tot el programa
 - En EC les anomenarem ***globals***
- Per defecte, les declarades DINS una funció són *auto*
 - Accessibles sols dins el bloc on es declaren
 - Ocupen memòria (o registres) sols mentre s'executa la funció
 - En EC les anomenarem ***locals***

Declaració de variables. Exemple

- Codi en C

```
int g = 0x76543210;
void func() {
    int loc1;

    loc1 = 13;
    ...
}
```

GLOBAL: **g** ocuparà l'adreça fixa 0x10010000 (sinònim de l'etiqueta g) durant tot el programa

- Codi assembleador MIPS

```
.data
g: .word 0x76543210

.text
func:
    addiu    $t0,$zero,13    #loc1=13
    ...
```

LOCAL: **loc1** ocupa el registre \$t0 mentre s'executa *func*. Després, \$t0 pot allotjar altres dades

Declaració de variables. Més exemples

- Declaracions globals en C...

```
char c    = 0x11;
short s   = 0x2211;
int i     = 0x44332211;
int iext  = 0xFF;           // 0xFF equival a 0x000000FF
int iz;           // les globals no inicialitzades valen 0
unsigned int ui;
long long l = 0x8877665544332211;
```

Declaració de variables. Més exemples

- Declaracions globals en C...

```
char c    = 0x11;
short s   = 0x2211;
int i     = 0x44332211;
int iext  = 0xFF;           // 0xFF equival a 0x000000FF
int iz;           // les globals no inicialitzades valen 0
unsigned int ui;
long long l = 0x8877665544332211;
```

- ... i la seva traducció a ensamblador

```
        .data
c:       .byte    0x11
s:       .half    0x2211
i:       .word    0x44332211
iext:    .word    0xFF
iz:      .word    0
ui:      .word    0
l:       .dword   0x8877665544332211
```

Alineació automàtica

- Les variables en memòria han d'estar *alineades*:

L'adreça de la variable ha de ser múltiple de la seva mida

- Altrament es poden produir *excepcions* durant l'execució

Tipus de variable en C	Directiva de reserva d'espai en MIPS	Mida en bytes	Adreça múltiple de
char / unsigned char	<code>.byte</code>	1	-
short / unsigned short	<code>.half</code>	2	2
int / unsigned int	<code>.word</code>	4	4
long long / unsigned long long	<code>.dword</code>	8	8

- Les directives `.half`, `.word`, `.dword` ja alineen automàticament

Alineació automàtica. Exemple

- En C

```
unsigned char a;  
short b = 13;  
char c = -1, d = 10;  
int e = 0x10AA00FF;  
long long f = 0x7766554433221100;
```

- Assemblador

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

Alineació automàtica. Exemple

- En C

```
unsigned char a;  
short b = 13;  
char c = -1, d = 10;  
int e = 0x10AA00FF;  
long long f = 0x7766554433221100;
```

- Assemblador

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

Memòria

etiqs.		adrees
a:	00	0
b:	0D 00	2
c:	FF	4
d:	0A	5
e:	FF 00 AA 10	8
f:	00 11 22 33 44 55 66 77	16

Alineació explícita amb .align

- Vector inicialitzat

```
short vec[5] = {2, -1, 3, 5, 0};
```

	.data
vec:	.half 2, -1, 3, 5, 0

- La directiva .half assegura l'alineació correcta de vec ✓

Alineació explícita amb .align

- Vector inicialitzat

```
short vec[5] = {2, -1, 3, 5, 0};
```

```
                .data  
vec:            .half 2, -1, 3, 5, 0
```

- La directiva .half assegura l'alineació correcta de vec ✓

- Vector global sense inicialitzar (zeros, per defecte)

```
char a;
```

```
int vec[100];
```

```
                .data  
a:              .byte 0  
  
vec:            .space 400 # 100 enters int de 4 bytes
```

- La directiva **.space n** reserva **n** bytes i els posa a 0
- Però no fa cap alineació: vec pot quedar mal alineat! ✗

Alineació explícita amb .align

- Vector inicialitzat

```
short vec[5] = {2, -1, 3, 5, 0};
```

```
      .data  
vec:   .half 2, -1, 3, 5, 0
```

- La directiva .half assegura l'alineació correcta de vec ✓

- Vector global sense inicialitzar (zeros, per defecte)

```
char a;
```

```
int vec[100];
```

```
      .data  
a:     .byte 0  
      .align 2      # força @vec múltiple de 4 = 22  
vec:   .space 400    # 100 enters int de 4 bytes
```

- La directiva **.space n** reserva **n** bytes i els posa a 0
- Però no fa cap alineació: vec pot quedar mal alineat! ✗
- Cal forçar l'alineació amb **.align n**, essent $n \in [1, 2, 3]$
 - Ubica l'etiqueta vec a la primera adreça disponible múltiple de 2^n

Constants simbòliques

- Constants que apareixen en múltiples llocs del codi
 - Modificant la definició queden redefinides totes les instàncies

- En C

```
#define N 100
int vec[N];
...
for (i=0; i<N; i++) vec[i]=vec[N-1-i];
```

- En MIPS

```
        .eqv N 100
        .data
vec:     .space N*4
        .text
        ...
bucle:   ...
        slti    $t0, N, fibucle
```

Operands

- En mode registre
- En mode immediat
- En memòria

Operands

- *Mode d'adreçament és la manera d'especificar l'operand*
 - MIPS suporta 5 modes
 - registre, immediat, memòria, pseudodirecte i relatiu al PC
- Operands en mode registre
 - L'operand resideix en un registre
 - La instrucció especifica l'identificador del registre (5 bits)
 - Suma i resta:

addu rd, rs, rt

rd \leftarrow rs + rt

subu rd, rs, rt

rd \leftarrow rs - rt

Registres

Número	Nom	Utilització
\$0	\$zero	Sempre val zero, no modificable
\$1	\$at	Reservat a l'expansió de macros (convé no usar-lo)
\$2-\$3	\$v0-\$v1	Resultat de subrutines (sols usarem \$v0)
\$4-\$7	\$a0-\$a3	<i>Arguments</i> o paràmetres de subrutines
\$8-\$15	\$t0-\$t7	<i>Temporals</i>
\$16-\$23	\$s0-\$s7	<i>Saved</i> ("segurs"), es preserven en cridar a una subrutina
\$24-\$25	\$t8-\$t9	<i>Temporals</i>
\$26-\$27	\$k0-\$k1	Reservats per al nucli (<i>Kernel</i>) del SO (convé no usar-los)
\$28	\$gp	<i>Global pointer</i> , explicat al Tema 3 (no l'usarem)
\$29	\$sp	<i>Stack pointer</i> , conté l'adreça del cim de la pila
\$30	\$fp	<i>Frame pointer</i> (no l'estudiem)
\$31	\$ra	<i>Return address</i> , adreça de retorn de subrutina

Operands en mode immediat

- L'operand de 16 bits es codifica en la pròpia instrucció
- Quan s'executa, converteix l'operand de 16 a 32 bits
 - Per extensió de signe (enters en Ca2)
 - Per extensió de zeros (operands sense signe)
- Alguns exemples:

```
addiu    rt, rs, imm16    # rt ← rs + SignExt(imm16)
ori      rt, rs, imm16    # rt ← rs OR ZeroExt(imm16)
lui      rt, imm16        # rt31..16 ← imm16
                                # rt15..0 ← 0x0000
```

Exercici

- Donat el codi en C:

```
int f, g, h, i;
```

```
...
```

```
f = (g + h) - (i - 100);
```

- Suposem que f , g , h , i estan als registres \$t0, \$t1, \$t2, \$t3
- Traduir la sentència a ensamblador MIPS

```
addu    $t4, $t1, $t2          # g+h
addiu    $t5, $t3, -100        # i-100
subu     $t0, $t4, $t5         # f = (...) - (...)
```

Operands en mode memòria

- Sols s'admeten en instruccions de tipus *load* i *store*
→ Cal carregar les dades de memòria en registres per usar-les en operacions aritmètico-lògiques

- Lectura d'un word en memòria: Load Word

```
lw    rt, off16(rs)    # rt ← Memword[rs + SignExt(off16)]
```

- Escriptura d'un word en memòria: Store Word

```
sw    rt, off16(rs)    # Memword[rs + SignExt(off16)] ← rt
```

Exercici

- Donat el codi en C:

```
int A[100], h;
```

```
...
```

```
A[12] = h + A[8];
```

- Suposem que h està en \$t2, i l'adreça base de A en \$t3
- Traduir la sentència a assembleador MIPS

```
lw      $t0, 32($t3)          # $t0 ← A[8]
addu    $t0, $t2, $t0         # sumar h + A[8]
sw      $t0, 48($t3)          # A[12] ← $t0
```


Operands en memòria *halfword* amb signe

- Load Halfword amb signe

```
lh      rt, off16(rs)      #  $rt \leftarrow \text{SignExt}[\text{Mem}_{\text{half}}[rs + \text{SignExt}(\text{off16})]]$ 
```

- Copia *un halfword* de memòria als 2 bytes de menor pes de rt
- El converteix a 4 bytes **extenent el signe** (replica el bit 15)

- Store Halfword

```
sh      rt, off16(rs)      #  $\text{Mem}_{\text{half}}[rs + \text{SignExt}(\text{off16})] \leftarrow rt_{15..0}$ 
```

- Copia a memòria els 2 bytes de menor pes de rt

Operands en memòria *halfword* amb signe

- Load Halfword amb signe

`lh rt, off16(rs) # $rt \leftarrow \text{SignExt}[\text{Mem}_{\text{half}}[rs + \text{SignExt}(\text{off16})]]$`

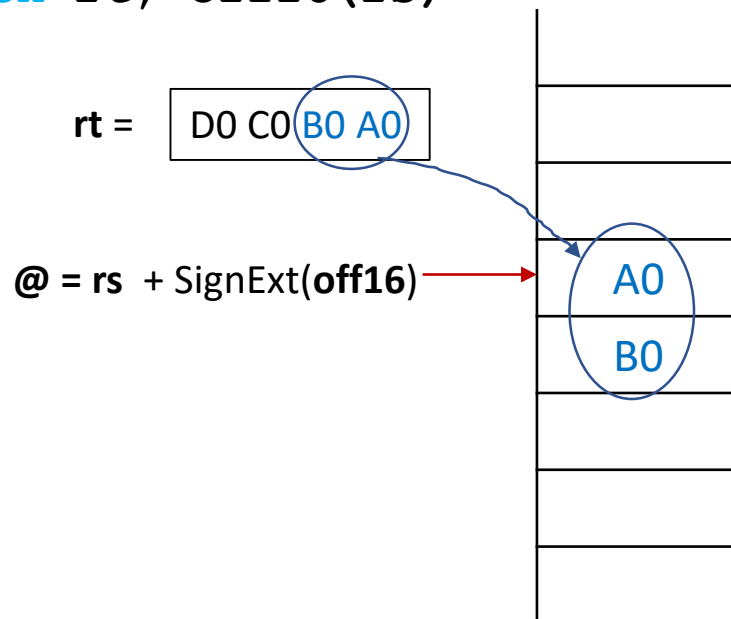
- Copia *un halfword* de memòria als 2 bytes de menor pes de `rt`
- El converteix a 4 bytes **extenent el signe** (replica el bit 15)

- Store Halfword

`sh rt, off16(rs) # $\text{Mem}_{\text{half}}[rs + \text{SignExt}(\text{off16})] \leftarrow rt_{15..0}$`

- Copia a memòria els 2 bytes de menor pes de `rt`

`sh rt, off16(rs)`



Operands en memòria *halfword* amb signe

- Load Halfword amb signe

`lh rt, off16(rs)` # $rt \leftarrow \text{SignExt}[\text{Mem}_{\text{half}}[rs + \text{SignExt}(\text{off16})]]$

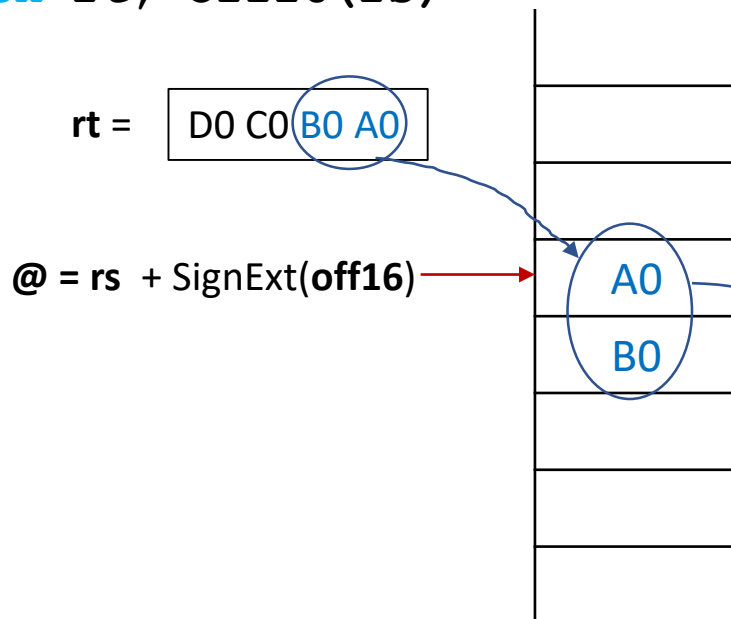
- Copia *un halfword* de memòria als 2 bytes de menor pes de `rt`
- El converteix a 4 bytes **extenent el signe** (replica el bit 15)

- Store Halfword

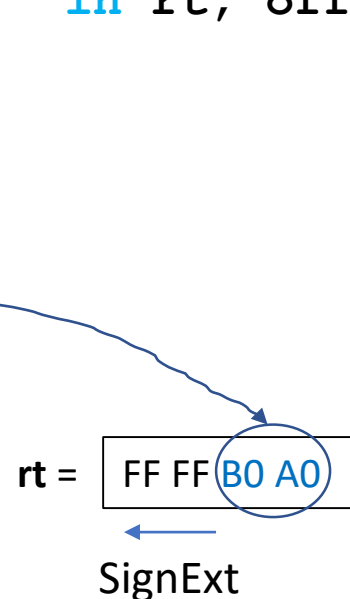
`sh rt, off16(rs)` # $\text{Mem}_{\text{half}}[rs + \text{SignExt}(\text{off16})] \leftarrow rt_{15..0}$

- Copia a memòria els 2 bytes de menor pes de `rt`

`sh rt, off16(rs)`



`lh rt, off16(rs)`



Operands en memòria *byte* amb signe

- Load Byte, amb signe

`lb rt, off16(rs) # rt ← SignExt[Membyte[rs + SignExt(off16)]]`

- Copia *un byte* de memòria al byte de menor pes de rt
- El converteix a 4 bytes **extenent el signe** (replica el bit 7)

- Store Byte

`sb rt, off16(rs) # Membyte[rs + SignExt(off16)] ← rt7..0`

- Copia a memòria el byte de menor pes de rt

Operands en memòria *byte* amb signe

- Load Byte, amb signe

`lb rt, off16(rs) # $rt \leftarrow \text{SignExt}[\text{Mem}_{\text{byte}}[rs + \text{SignExt}(\text{off16})]]$`

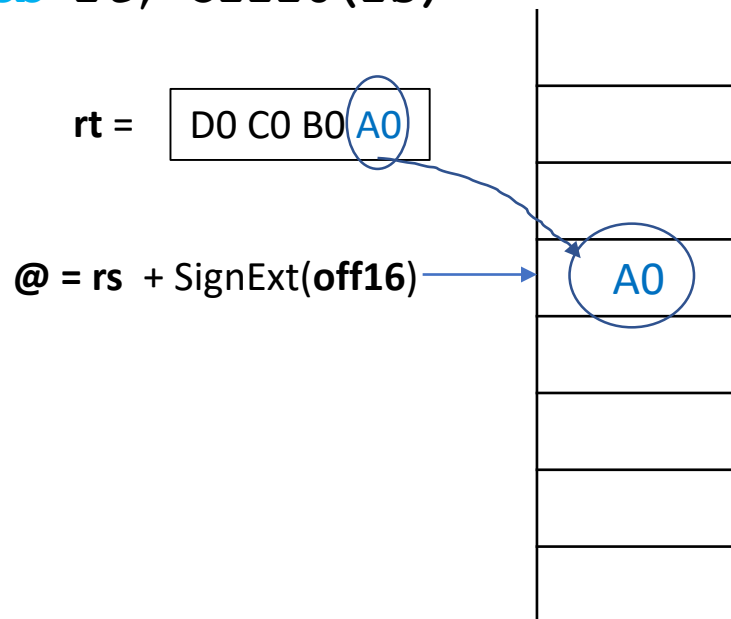
- Copia *un byte* de memòria al byte de menor pes de `rt`
- El converteix a 4 bytes **extenent el signe** (replica el bit 7)

- Store Byte

`sb rt, off16(rs) # $\text{Mem}_{\text{byte}}[rs + \text{SignExt}(\text{off16})] \leftarrow rt_{7..0}$`

- Copia a memòria el byte de menor pes de `rt`

`sb rt, off16(rs)`



Operands en memòria *byte* amb signe

- Load Byte, amb signe

`lb rt, off16(rs)` # $rt \leftarrow \text{SignExt}[\text{Mem}_{\text{byte}}[rs + \text{SignExt}(\text{off16})]]$

- Copia *un byte* de memòria al byte de menor pes de `rt`
- El converteix a 4 bytes **extenent el signe** (replica el bit 7)

- Store Byte

`sb rt, off16(rs)` # $\text{Mem}_{\text{byte}}[rs + \text{SignExt}(\text{off16})] \leftarrow rt_{7..0}$

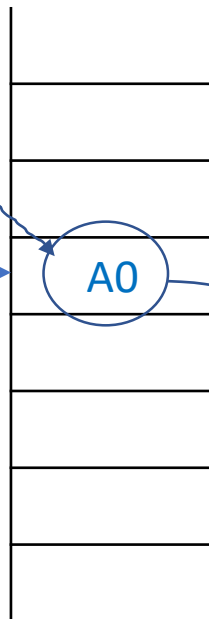
- Copia a memòria el byte de menor pes de `rt`

`sb rt, off16(rs)`

`rt =`

D0	C0	B0	A0
----	----	----	----

`@ = rs + SignExt(off16)`



`lb rt, off16(rs)`

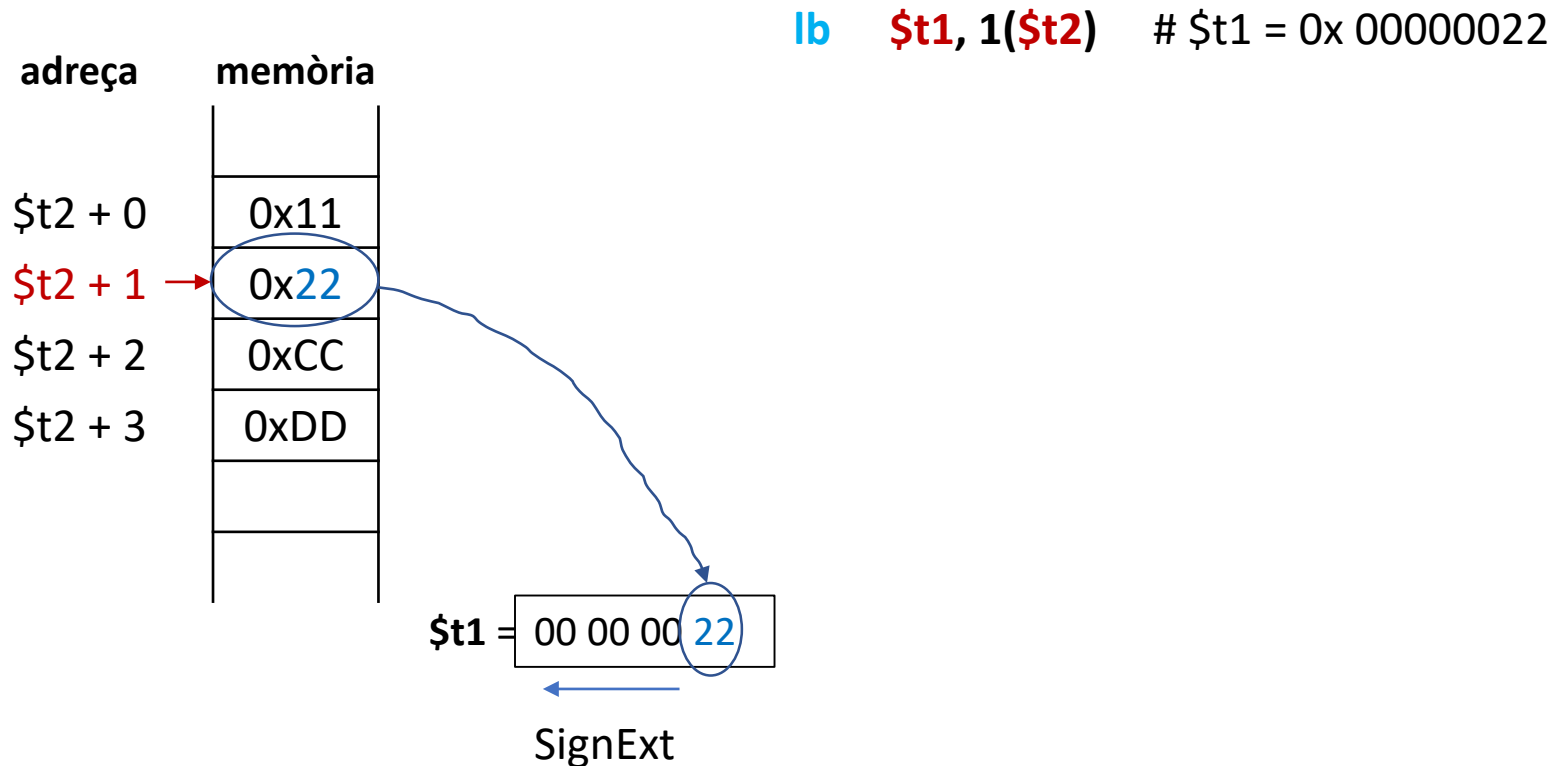
`rt =`

FF	FF	FF	A0
----	----	----	----

←
SignExt

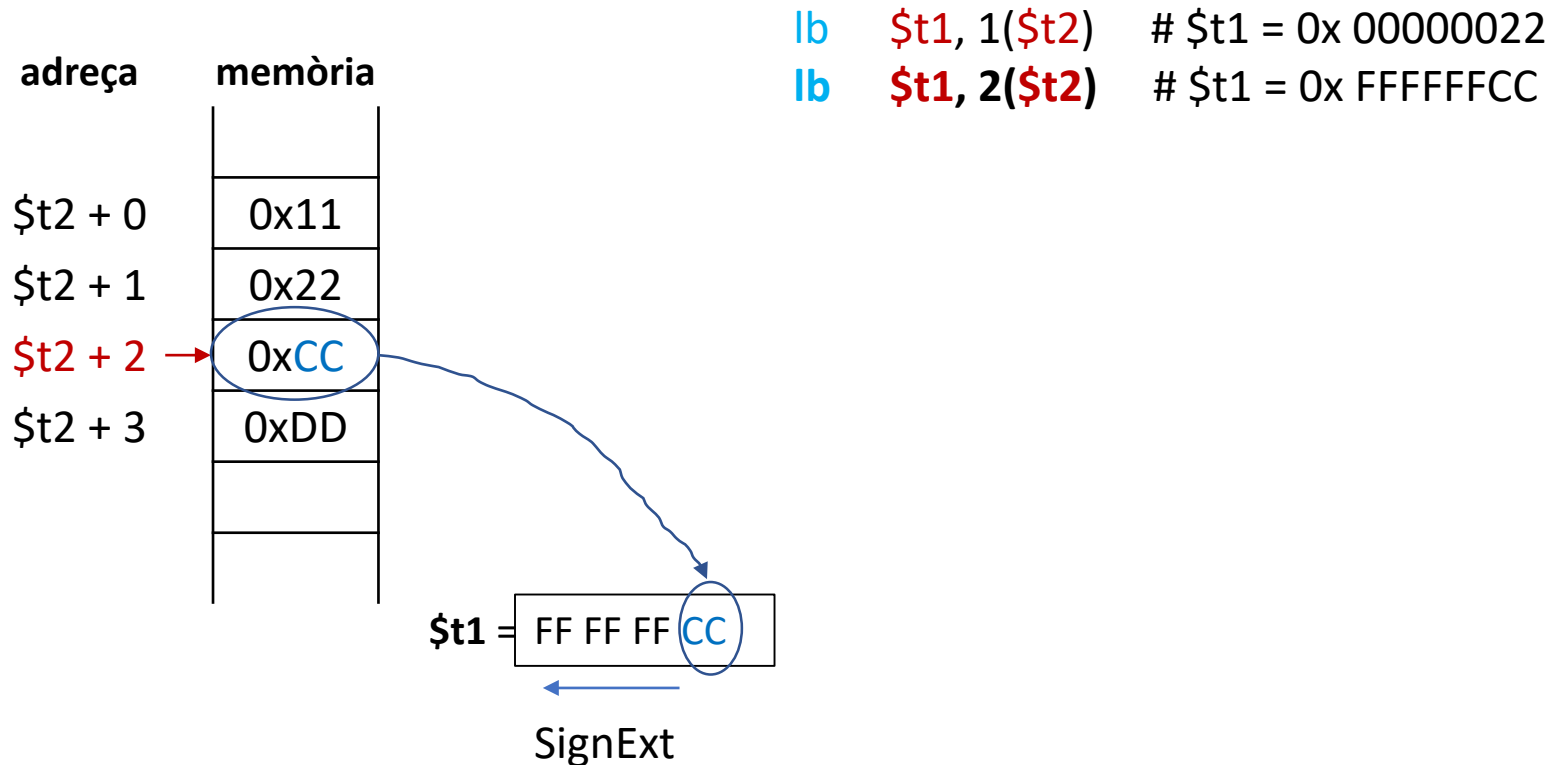
Exercici

- Indica el resultat de cada instrucció i l'estat final de la memòria



Exercici

- Indica el resultat de cada instrucció i l'estat final de la memòria



Exercici

- Indica el resultat de cada instrucció i l'estat final de la memòria

adreça	memòria
$\$t2 + 0$	0x11
$\$t2 + 1$	0x22
$\$t2 + 2$	0xCC
$\$t2 + 3$	0xDD

$\$t1 =$ 00 00 22 11
← SignExt

lb $\$t1, 1(\$t2)$ # $\$t1 = 0x\ 00000022$
lb $\$t1, 2(\$t2)$ # $\$t1 = 0x\ FFFFFFFC$
lh $\$t1, 0(\$t2)$ # $\$t1 = 0x\ 00002211$

Exercici

- Indica el resultat de cada instrucció i l'estat final de la memòria

adreça	memòria
\$t2 + 0	0x11
\$t2 + 1	0x22
\$t2 + 2	0xCC
\$t2 + 3	0xDD

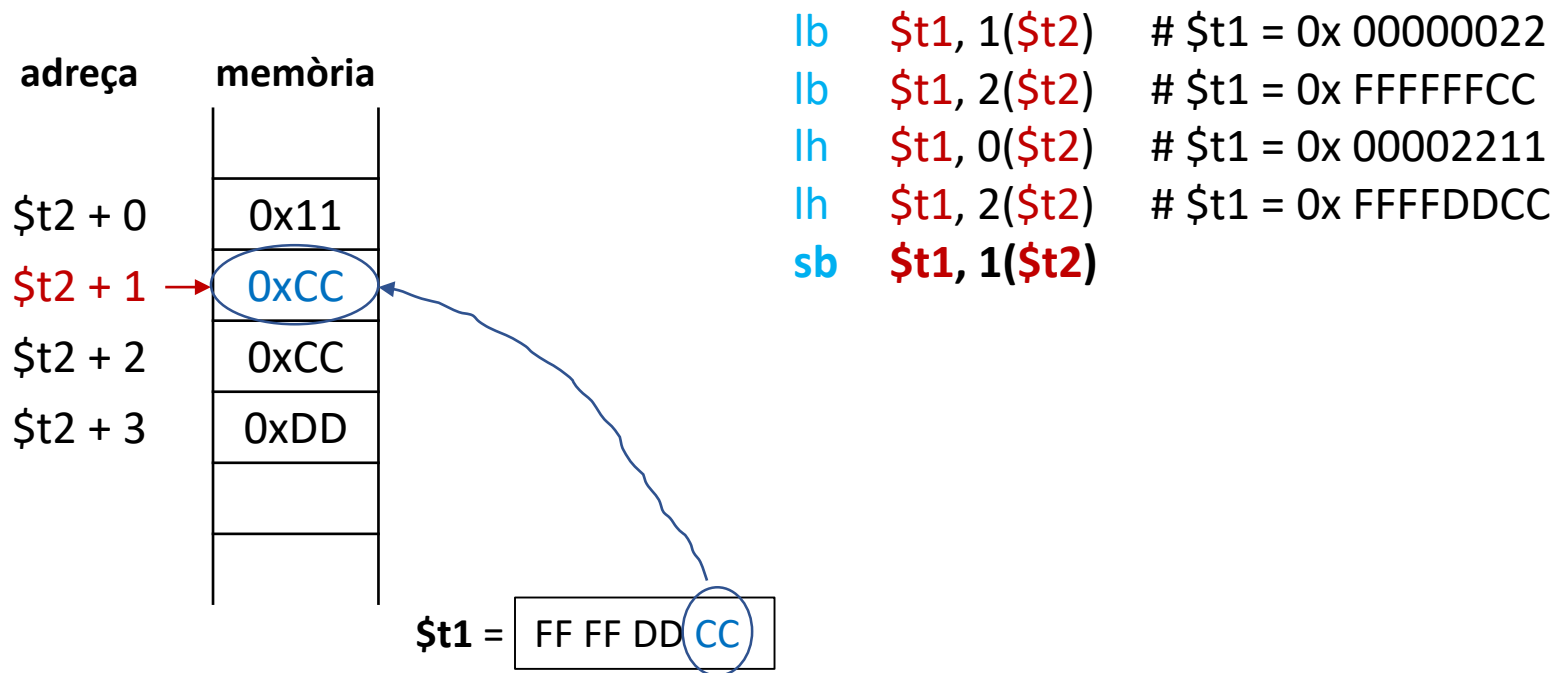
lb \$t1, 1(\$t2) # \$t1 = 0x 00000022
lb \$t1, 2(\$t2) # \$t1 = 0x FFFFFFFC
lh \$t1, 0(\$t2) # \$t1 = 0x 00002211
lh **\$t1, 2(\$t2)** # \$t1 = 0x FFFDCC

\$t1 = FF FF **DD CC**

←
SignExt

Exercici

- Indica el resultat de cada instrucció i l'estat final de la memòria



Operands en memòria *halfword* o *byte* naturals

- Load Halfword Unsigned

```
lhu    rt, off16(rs)    # rt ← ZeroExt[Memhalf[rs + SignExt(off16)]]
```

- Copia *un halfword* de memòria als 2 bytes de menor pes de rt
- El converteix a 4 bytes **extenent zeros**

- Load Byte Unsigned

```
lbu    rt, off16(rs)    # rt ← ZeroExt[Membyte[rs + SignExt(off16)]]
```

- Copia *un byte* de memòria al byte de menor pes de rt
- El converteix a 4 bytes **extenent zeros**

Operands en memòria *halfword* o *byte* naturals

- Load Halfword Unsigned

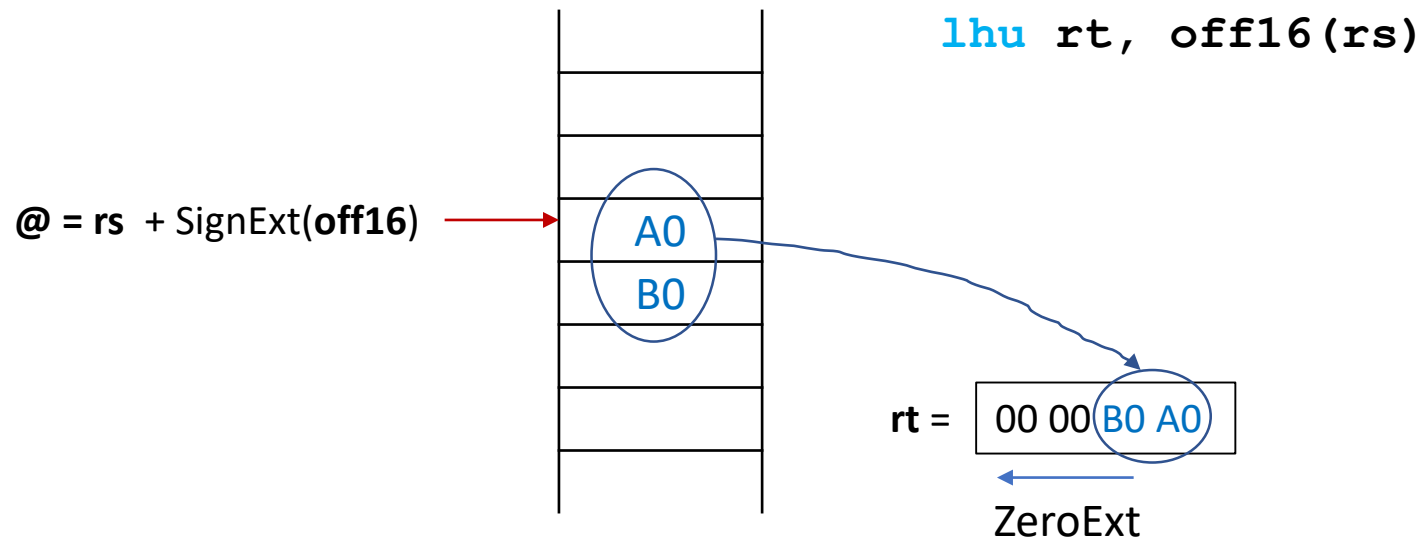
```
lhu    rt, off16(rs)    # rt ← ZeroExt[Memhalf[rs + SignExt(off16)]]
```

- Copia *un halfword* de memòria als 2 bytes de menor pes de rt
- El converteix a 4 bytes **extenent zeros**

- Load Byte Unsigned

```
lbu    rt, off16(rs)    # rt ← ZeroExt[Membyte[rs + SignExt(off16)]]
```

- Copia *un byte* de memòria al byte de menor pes de rt
- El converteix a 4 bytes **extenent zeros**



Operands en memòria *doubleword*

- Com s'accedeix a una paraula de 64 bits (long long)?

- Declaració en C:

```
long long x = 0x7766554433221100;
```

- En ensamblador:

```
.data
```

```
x: .dword 0x7766554433221100
```

- Codi ensamblador per carregar x en \$t0-\$t1:

```
.text
```

```
...
```

```
# Suposem que $t2 conté l'adreça de memòria de x
```

```
lw $t0, 0($t2)
```

```
lw $t1, 4($t2)
```

Restricció d'alineació

- L' adreça efectiva de **lw** i **sw** ha de ser múltiple de 4
- L' adreça efectiva de **lh**, **lhu** i **sh** ha de ser múltiple de 2
- En cas contrari
 - Es produeix una **excepció per adreça no-alineada**
 - El programa acaba

- Exemple:

```
.data
x:  .word 0xDDCCBBAA
    .text
    ...

# Suposem que $t2 conté l'adreça de memòria de x
lw  $t0, 1($t2)
```

- Adreça no-alineada! → excepció

Pseudoinstruccions o macros: `move`, `li`, `la`

- Simplifiquen operacions freqüents i faciliten l'escriptura i depuració del codi
- L'assemblador l'*expandeix* a una o vàries instruccions

```
.data
x: .word 0
y: .word 69, 70
```

```
.text
move    $t1, $t2
```



```
.data
x: .word 0
y: .word 69, 70
```

```
.text
addu    $t1, $t2, $zero
```


Pseudoinstruccions o macros: `move`, `li`, `la`

- Simplifiquen operacions freqüents i faciliten l'escriptura i depuració del codi
- L'assemblador l'*expandeix* a una o vàries instruccions

```
.data
x: .word 0
y: .word 69, 70
```

```
.text
move    $t1, $t2

li      $t1, 100
```



```
.data
x: .word 0
y: .word 69, 70

.text
addu    $t1, $t2, $zero

addiu   $t1, $zero, 100
```

Pseudoinstruccions o macros: `move`, `li`, `la`

- Simplifiquen operacions freqüents i faciliten l'escriptura i depuració del codi
- L'assemblador l'*expandeix* a una o vàries instruccions

```
.data
x: .word 0
y: .word 69, 70
```

```
.text
move    $t1, $t2
```

```
li      $t1, 100
```

```
li      $t1, 0x075080AA
```

```
.data
x: .word 0
y: .word 69, 70
```

```
.text
addu     $t1, $t2, $zero
```

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

```
lui      $at, 0x0750
ori      $t1, $at, 0x80AA
```

Pseudoinstruccions o macros: `move`, `li`, `la`

- Simplifiquen operacions freqüents i faciliten l'escriptura i depuració del codi
- L'assemblador l'*expandeix* a una o vàries instruccions

```
.data
x: .word 0
y: .word 69, 70
```

```
.text
move    $t1, $t2
```

```
li      $t1, 100
```

```
li      $t1, 0x075080AA
```

```
la      $t1, y      # $t1 ← 0x10010004 →
```

```
.data
x: .word 0
y: .word 69, 70
```

```
.text
addu    $t1, $t2, $zero
```

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

```
lui     $at, 0x0750
ori     $t1, $at, 0x80AA
```

```
lui     $at, 0x1001
ori     $t1, $at, 0x0004
```

Pseudoinstruccions o macros: `move`, `li`, `la`

- Simplifiquen operacions freqüents i faciliten l'escriptura i depuració del codi
- L'assemblador l'*expandeix* a una o vàries instruccions

```
.data
x: .word 0
y: .word 69, 70
```

```
.text
move    $t1, $t2

li      $t1, 100

li      $t1, 0x075080AA

la      $t1, y      # $t1 ← 0x10010004

la      $t1, y+4    # $t1 ← 0x10010008 →
```

```
.data
x: .word 0
y: .word 69, 70

.text
addu    $t1, $t2, $zero

addiu   $t1, $zero, 100

lui     $at, 0x0750
ori     $t1, $at, 0x80AA

lui     $at, 0x1001
ori     $t1, $at, 0x0004

lui     $at, 0x1001
ori     $t1, $at, 0x0008
```

Exercici

- Tradueix a MIPS el programa en C

```
short v[3] = {-1, 32, 15}, sum;           // v, sum són globals
void main() {
    sum = v[0] + v[1] + v[2] + 91;
}
```

Exercici

- Tradueix a MIPS el programa en C

```
short v[3] = {-1, 32, 15}, sum;           // v, sum són globals
void main() {
    sum = v[0] + v[1] + v[2] + 91;
}
```

- Traducció

```
.data
v:  .half -1, 32, 15
sum: .half 0
.text
main:
    la    $t0, v
    lh    $t1, 0($t0)    # v[0]
    lh    $t2, 2($t0)    # v[1]
    addu   $t1, $t1, $t2
    lh    $t3, 4($t0)    # v[2]
    addu   $t1, $t1, $t3
    addiu  $t1, $t1, 91
    la    $t0, sum
    sh    $t1, 0($t0)
```

Sistemes de representació binària

- Naturals i Enters (repàs)
- Caràcters
- Instruccions MIPS

Representació de naturals (repàs)

- Denotem:

- x_u = valor natural a representar
- $X = X_{n-1} \dots X_1 X_0$ = vector de n bits que representa x_u

- Trobar quin número natural x_u està representat per X

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

- Exemple: per a $X = 00011001_2$

$$x_u = 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^0 = \mathbf{25}$$

Representació de naturals (repàs)

- Denotem:

- x_u = valor natural a representar
- $X = X_{n-1} \dots X_1 X_0$ = vector de n bits que representa x_u

- Trobar quin número natural x_u està representat per X

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

- Exemple: per a $X = 00011001_2$

$$x_u = 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^0 = \mathbf{25}$$

- Rang de representació: $x_u \in [0, 2^n - 1]$

- Per exemple, per a $n=8$ és: $x_u \in [0, 255]$

- Extensió del rang: afegir zeros a l'esquerra

- P.ex. estendre $X=1001_2$ a 8 bits $\rightarrow X=00001001_2$

Representació de naturals (repàs)

- Trobar la representació X per al natural x_u

$$X_0 = x_u \bmod 2$$

$$X_1 = Q_0 \bmod 2$$

...

$$X_{n-2} = Q_{n-3} \bmod 2$$

$$X_{n-1} = Q_{n-2} \bmod 2$$

$$Q_0 = x_u \operatorname{div} 2$$

$$Q_1 = Q_0 \operatorname{div} 2$$

...

$$Q_{n-2} = Q_{n-3} \operatorname{div} 2$$

Representació de naturals (repàs)

- Trobar la representació X per al natural x_u

$$X_0 = x_u \bmod 2$$

$$Q_0 = x_u \operatorname{div} 2$$

$$X_1 = Q_0 \bmod 2$$

$$Q_1 = Q_0 \operatorname{div} 2$$

...

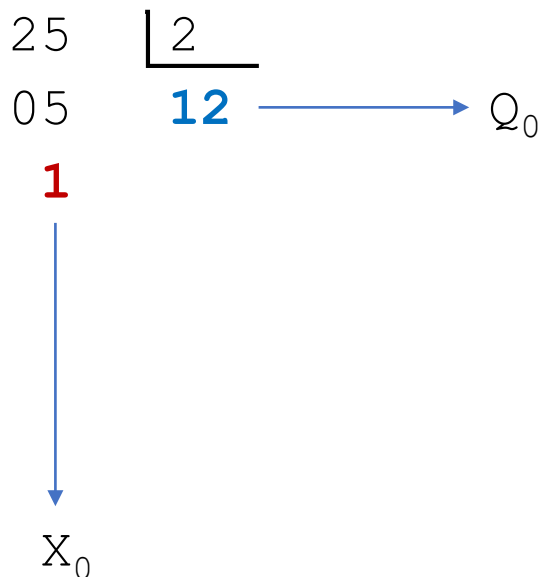
...

$$X_{n-2} = Q_{n-3} \bmod 2$$

$$Q_{n-2} = Q_{n-3} \operatorname{div} 2$$

$$X_{n-1} = Q_{n-2} \bmod 2$$

- Exemple: $x_u = 25$



Representació de naturals (repàs)

- Trobar la representació X per al natural x_u

$$X_0 = x_u \bmod 2$$

$$Q_0 = x_u \operatorname{div} 2$$

$$X_1 = Q_0 \bmod 2$$

$$Q_1 = Q_0 \operatorname{div} 2$$

...

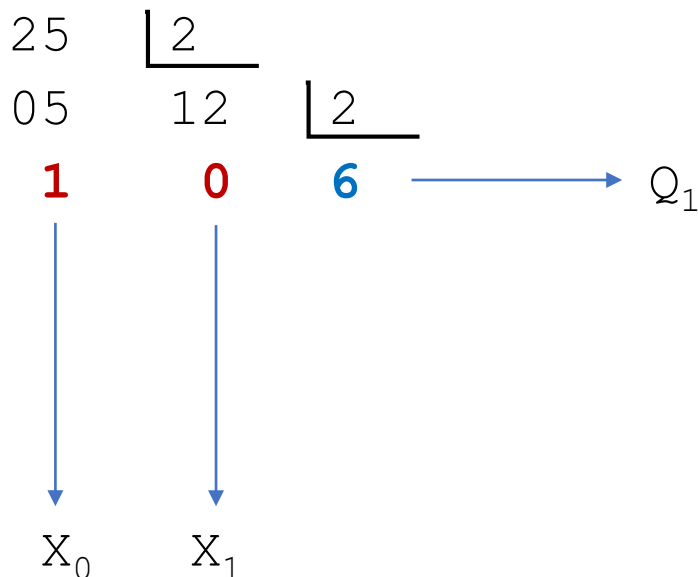
...

$$X_{n-2} = Q_{n-3} \bmod 2$$

$$Q_{n-2} = Q_{n-3} \operatorname{div} 2$$

$$X_{n-1} = Q_{n-2} \bmod 2$$

- Exemple: $x_u = 25$



Representació de naturals (repàs)

- Trobar la representació X per al natural x_u

$$X_0 = x_u \bmod 2$$

$$Q_0 = x_u \operatorname{div} 2$$

$$X_1 = Q_0 \bmod 2$$

$$Q_1 = Q_0 \operatorname{div} 2$$

...

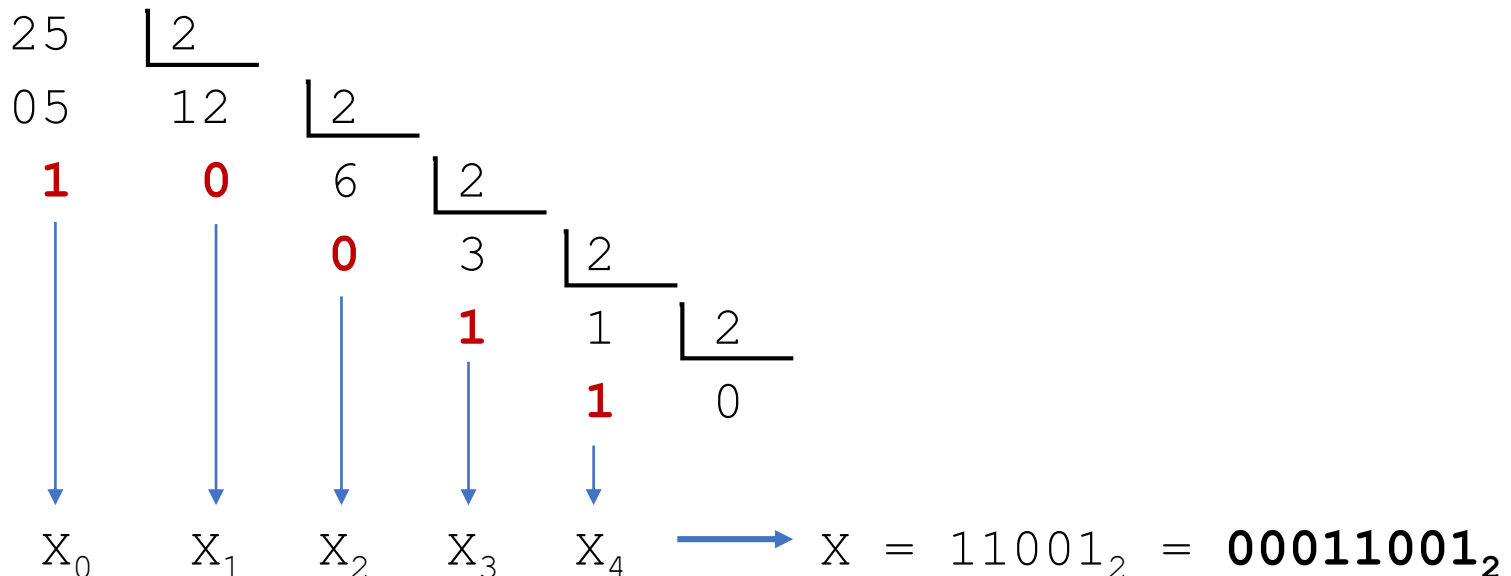
...

$$X_{n-2} = Q_{n-3} \bmod 2$$

$$Q_{n-2} = Q_{n-3} \operatorname{div} 2$$

$$X_{n-1} = Q_{n-2} \bmod 2$$

- Exemple: $x_u = 25$



Representació de naturals (repàs)

- Variables de tipus natural (*unsigned integers*) en C

```
unsigned char var1;           // 1 byte
unsigned short var2;          // 2 bytes
unsigned int var3;            // 4 bytes
unsigned long long var4;      // 8 bytes
```

- En Assemblador MIPS

```
                .data
var1:           .byte 0
var2:           .half  0
var3:           .word  0
var4:           .dword 0
```

Representació d'enters en Ca2 (repàs)

- Denotem:

- x_s = valor enter a representar
- $X = X_{n-1} \dots X_1 X_0$ = vector de n bits que representa x_s
- x_u = natural representat per X (valor explícit de x_s)

- Trobar quin número enter x_s està representat per X

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

- O bé: interpretar-lo com a natural x_u , i restar-li 2^n si $X_{n-1}=1$

$$x_s = x_u - X_{n-1} \cdot 2^n$$

Representació d'enters en Ca2 (repàs)

- Denotem:

- x_s = valor enter a representar
- $X = X_{n-1} \dots X_1 X_0$ = vector de n bits que representa x_s
- x_u = natural representat per X (valor explícit de x_s)

- Trobar quin número enter x_s està representat per X

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

- O bé: interpretar-lo com a natural x_u , i restar-li 2^n si $X_{n-1}=1$

$$x_s = x_u - X_{n-1} \cdot 2^n$$

- Exemple: per a $X = 10011001_2$

$$x_s = -1 \cdot 2^7 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^0 = -128 + 25 = -103$$

- O bé, interpretar-lo com a natural x_u , i restar-li 2^8

$$x_u = 1 \cdot 2^7 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^0 = 153$$

$$x_s = x_u - 1 \cdot 2^8 = 153 - 256 = -103$$

Representació d'enters en Ca2 (repàs)

- Trobar la representació X per a l'enter x_s (en 2 passos)
 1. Trobar el valor explícit: $x_u = \begin{cases} x_s, & \text{si } x_s \geq 0 \\ x_s + 2^n, & \text{altrament} \end{cases}$
 2. Representar x_u com a natural (divisions successives)
- Exemple: representar amb 8 bits $x_s = -124$
 1. $x_u = x_s + 2^8 = -124 + 256 = 132$
 2. Divisions succ. \rightarrow el natural 132 es representa $X = 10000100_2$

Representació d'enters en Ca2 (repàs)

- Trobar la representació X per a l'enter x_s (en 2 passos)
 1. Trobar el valor explícit: $x_u = \begin{cases} x_s, & \text{si } x_s \geq 0 \\ x_s + 2^n, & \text{altrament} \end{cases}$
 2. Representar x_u com a natural (divisions successives)
- Exemple: representar amb 8 bits $x_s = -124$
 1. $x_u = x_s + 2^8 = -124 + 256 = 132$
 2. Divisions succ. \rightarrow el natural 132 es representa $X = 10000100_2$
- Rang de representació: $x_s \in [-2^{n-1}, 2^{n-1} - 1]$
 - Per exemple, per a $n=8$ és: $x_s \in [-128, 127]$
- Extensió del rang: replicar el bit de signe X_{n-1} a l'esquerra
 - Per exemple, estendre $X=1001_2$ a 8 bits: $X=11111001_2$

Representació d'enters en Ca2 (repàs)

- **Regla del canvi de signe:** Complementar bits i sumar 1
 - Exemple: canviar signe de $X = 01100011_2$ (és l'enter $x_s = 99$)
$$X' = \bar{X} + 1$$
$$= 10011100_2 + 1$$
$$= 10011101_2 \quad (\text{representa l'enter } x'_s = -99)$$

- Variables de tipus enter en Ca2 (*signed integers*) en C

```
char var1;           // 1 byte
short var2;          // 2 bytes
int var3;             // 4 bytes
long long var4;       // 8 bytes
```

- En Assemblador MIPS (amb signe o sense, no canvia)

```
        .data
var1:   .byte 0
var2:   .half 0
var3:   .word 0
var4:   .dword 0
```

Representació d'enters en Ca1 (repàs)

- Trobar quin número enter x_s està representat per X

- Interpretar X com a natural x_u , i restar-li $2^n - 1$ si $X_{n-1} = 1$

$$x_s = x_u - X_{n-1} \cdot (2^n - 1)$$

- Trobar la representació X per a l'enter x_s (en 2 passos)

1. Trobar el valor explícit: $x_u = \begin{cases} x_s, & \text{si } x_s \geq 0 \\ x_s + (2^n - 1), & \text{si } x_s \leq 0 \end{cases}$
2. Representar x_u com a natural (divisions successives)

Representació d'enters en Ca1 (repàs)

- Trobar quin número enter x_s està representat per X

- Interpretar X com a natural x_u , i restar-li $2^n - 1$ si $X_{n-1} = 1$

$$x_s = x_u - X_{n-1} \cdot (2^n - 1)$$

- Trobar la representació X per a l'enter x_s (en 2 passos)

1. Trobar el valor explícit: $x_u = \begin{cases} x_s, & \text{si } x_s \geq 0 \\ x_s + (2^n - 1), & \text{si } x_s \leq 0 \end{cases}$

2. Representar x_u com a natural (divisions successives)

- Rang de representació simètric: $x_s \in [-(2^{n-1}-1), 2^{n-1}-1]$

- Per exemple, per a $n=8$ és: $x_s \in [-127, +127]$

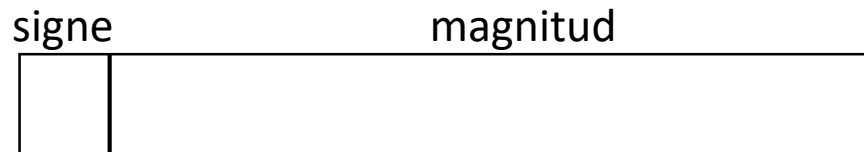
- 2 representacions per al zero! $00000000_2 = 11111111_2$

- Canvi de signe: complementar bits

- Exemple: canviar de signe $X = 01100011_2$ (és l'enter $x_s = 99$)

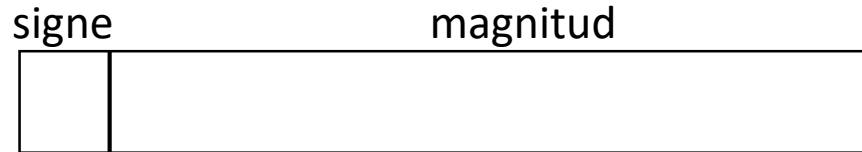
$$X' = \bar{X} = 10011100_2 \quad (\text{representa l'enter } x'_s = -99)$$

Enters en Signe i Magnitud (repàs)



- Trobar quin número enter x_s està representat per X
 - El bit de major pes indica el signe (0 positiu, 1 negatiu)
 - Els $n-1$ bits restants codifiquen el valor absolut (natural)
- Trobar la representació X per a l'enter x_s
 - Codificar el signe en el bit de major pes
 - Codificar el valor absolut (natural) en els $n-1$ bits restants
 - 2 representacions per al zero! $00000000_2 = 10000000_2$

Enters en Signe i Magnitud (repàs)



- Trobar quin número enter x_s està representat per X
 - El bit de major pes indica el signe (0 positiu, 1 negatiu)
 - Els $n-1$ bits restants codifiquen el valor absolut (natural)
- Trobar la representació X per a l'enter x_s
 - Codificar el signe en el bit de major pes
 - Codificar el valor absolut (natural) en els $n-1$ bits restants
 - 2 representacions per al zero! $00000000_2 = 10000000_2$
- Rang de representació simètric: $x_s \in [-(2^{n-1}-1), 2^{n-1}-1]$
- Canvi de signe: complementar el bit de major pes

Enters en excés a E

- Trobar quin número enter x_s està representat per X

1. Interpretar X com a natural x_u
2. Restar-li l'excés E (normalment, $E = 2^{n-1} - 1$)

$$x_s = x_u - (2^{n-1} - 1)$$

- Trobar la representació X per a l'enter x_s

1. Trobar el valor explícit, sumant-li l'excés E
$$x_u = x_s + (2^{n-1} - 1)$$
2. Representar-lo com a natural (divisions successives)

Enters en excés a E

- Trobar quin número enter x_s està representat per X

1. Interpretar X com a natural x_u
2. Restar-li l'excés E (normalment, $E = 2^{n-1} - 1$)

$$x_s = x_u - (2^{n-1} - 1)$$

- Trobar la representació X per a l'enter x_s

1. Trobar el valor explícit, sumant-li l'excés E
$$x_u = x_s + (2^{n-1} - 1)$$
2. Representar-lo com a natural (divisions successives)

- Rang de representació: $x_s \in [-(2^{n-1}-1), 2^{n-1}]$

- Per exemple, per a $n=8$ és $x_s \in [-127, 128]$

- Propietat

- Es poden comparar amb el mateix circuit que els naturals
- S'usen per a l'exponent en números de coma flotant (Tema 5)

Caràcters

- Necessitat de representar símbols tipogràfics
 - Ja des dels temps de la telegrafia (s.XIX, codi Morse)
- Codificació = correspondència símbols ↔ números
 - Unicode, EBCDIC, ASCII, etc.
- Codi ASCII de 7 bits (1963)
 - Els codis 0 a 31 són *de control* (no imprimibles)

codi	símbol	en C i MIPS
0x00	null	'\0'
...		
0x09	TAB	'\t'
0x0A	LF	'\n'
...		
0x0D	CR	'\r'
...		
0x20	space	' '

codi	símbol	en C i MIPS
0x30	0	'0'
0x31	1	'1'
...		
0x41	A	'A'
0x42	B	'B'
...		
0x61	a	'a'
0x62	b	'b'

Caràcters

- Alguns codis i propietats a recordar
 - **Dígits decimals** (del '0' al '9')
 - a partir del codi 48 (0x30)
 - **Majúscules** ordenades (de 'A' a 'Z')
 - a partir del codi 65 (0x41)
 - **Minúscules** ordenades (de 'a' a 'z')
 - a partir del codi 97 (0x61)
 - Observar que de majúscules a minúscules sols canvia el bit 5
`cmin = cmaj + 32;`
 - Representar com a caràcter un dígit decimal n ($0 \leq n < 10$)
`cdigit = '0' + n;`
- Alguns caràcters més
 - Caràcter **null** ('\0'), té codi 0 (0x00)
 - Caràcter **espai** (' '), té codi 32 (0x20)
 - Caràcter **salt de línia** ('\n'), té codi 10 (0x0A)

Caràcters

- Variables de tipus caràcter, en C

```
char lletra = 'R';           // 1 byte
```

(si es declara *unsigned* no canvia res, ja que el bit 8 val 0)

- En Assemblador MIPS

```
        .data
lletra: .byte 'R'
```

- Programa en C

```
char cars[2] = {'A', '1'};
char n = 7;           // enter de 8 bits

void main() {
    // convert cars[0] a minúscules
    cars[0] = cars[0] + 32;

    // codificar n com caràcter
    cars[1] = n + '0';
}
```

- Programa en MIPS

```
        .data
cars:   .byte 'A', '1'
n:      .byte 7

        .text
main:
    la    $t0, cars
    lb    $t1, 0($t0)
    addiu $t1, $t1, 32
    sb    $t1, 0($t0)

    la    $t2, n
    lb    $t1, 0($t2)
    addiu $t1, $t1, '0'
    sb    $t1, 1($t0)
```

Format de les instruccions MIPS

- Sols 3 formats: R (registre), I (immediate), J (jumps)

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	imm16		
J	opcode	target				

- Alguns exemples

			6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
addu	rd, rs, rt	R	0x00	rs	rt	rd	0x00	0x21
sra	rd, rt, shamt	R	0x00	rs	rt	rd	shamt	0x03
addiu	rt, rs, imm16	I	0x08	rs	rt	imm16		
lui	rt, imm16	I	0x0F	0x00	rt	imm16		
lw	rt, offset16(rs)	I	0x23	rs	rt	offset16		
jal	target	J	0x03	target				

Exercici

- Codificar en binari la instrucció⁽¹⁾:

`addu $t0, $s2, $zero`

(1) Consulteu sintaxi, opcodes i números de registre al document [*Instruccions MIPS i macros MARS*](#) penjat a la web

Exercici

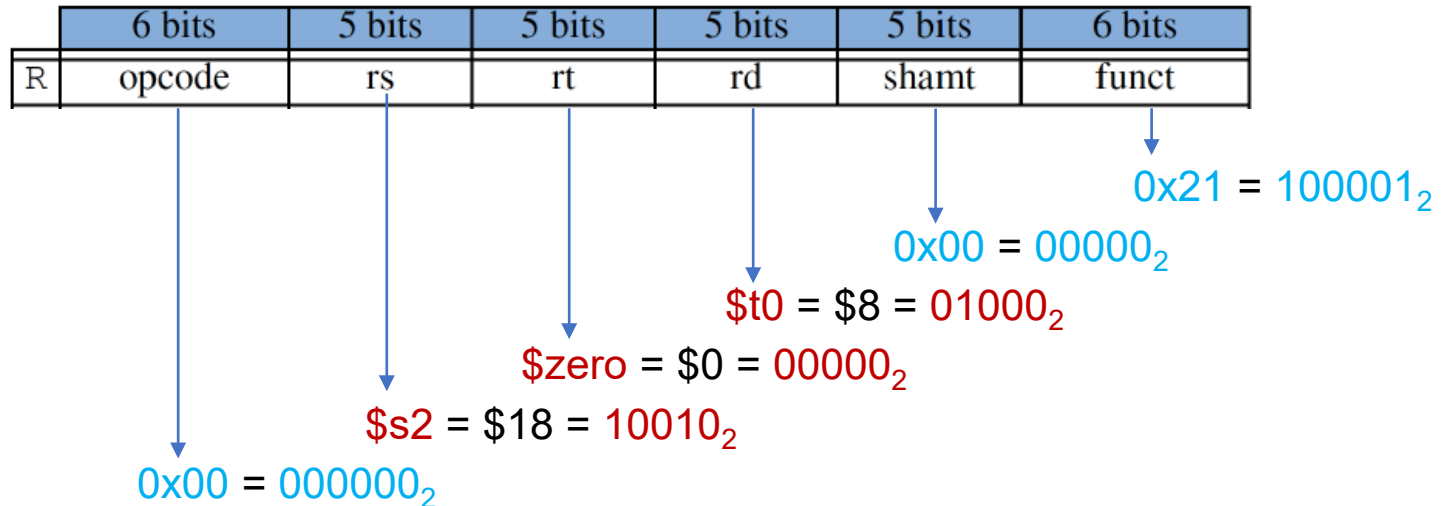
- Codificar en binari la instrucció⁽¹⁾:

`addu $t0, $s2, $zero`

- Solució

- Sintaxi: `addu rd, rs, rt`

- Format:



- Codificació:

`000000 10010 00000 01000 00000 1000012 = 0x02404021`

(1) Consulteu sintaxi, opcodes i números de registre al document [Instruccions MIPS i macros MARS](#) penjat a la web

Vectors i punters

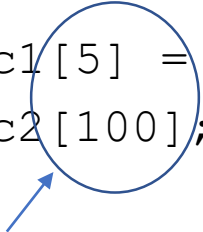
Vectors

- Agrupació unidimensional de N elements del mateix tipus
 - Elements identificats per un índex $\epsilon [0, N-1]$
 - S'emmagatzemen en posicions consecutives de memòria
 - En MIPS, han de respectar les regles d'alineació
 - Si la mida és potència de 2, alineant el primer s'alineen tots

Declaració de vectors

- Declaracions en C (variables globals)

```
short  vec1[5] = {0, -1, 2, -3, 4};  
int    vec2[100];
```



La *dimensió*: ha de ser una constant

Declaració de vectors

- Declaracions en C (variables globals)

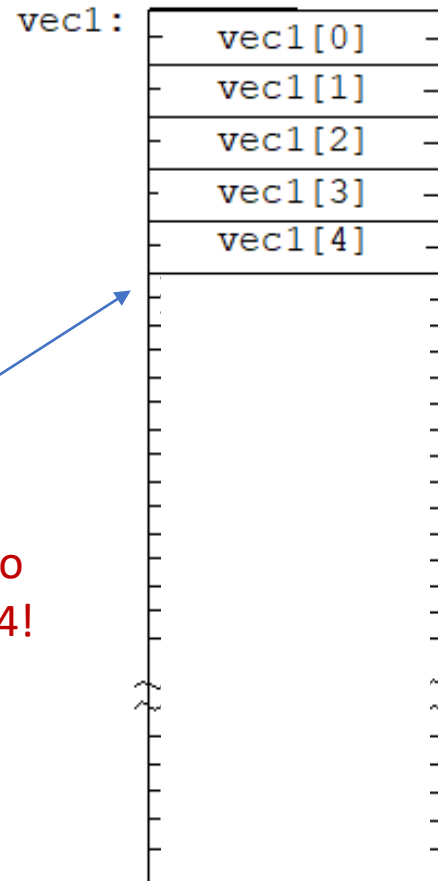
```
short  vec1[5] = {0, -1, 2, -3, 4};  
int    vec2[100];
```

La *dimensió*: ha de ser una constant

- Codi en MIPS

```
.data  
vec1:  .half 0, -1, 2, -3, 4  
  
vec2:  .space 400
```

(@vec1 = 0x10010000)



L'adreça
0x1001000A no
és múltiple de 4!

Declaració de vectors

- Declaracions en C (variables globals)

```
short vec1[5] = {0, -1, 2, -3, 4};  
int vec2[100];
```

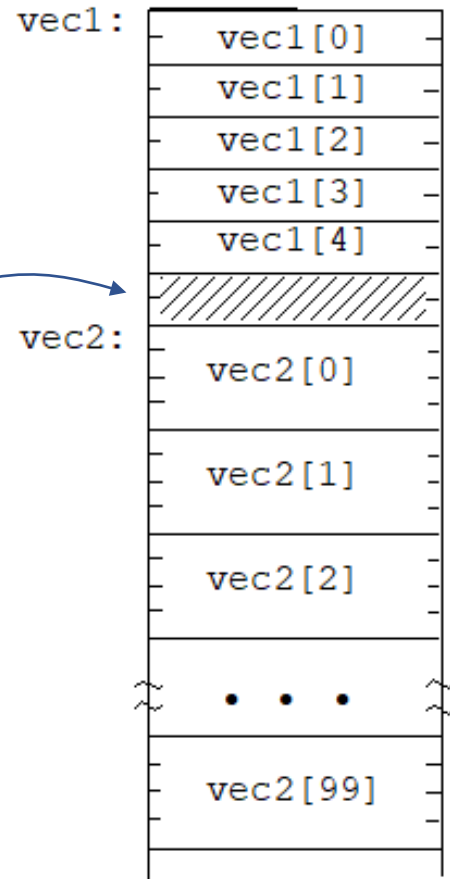
La *dimensió*: ha de ser una constant

- Codi en MIPS

```
.data  
vec1: .half 0, -1, 2, -3, 4  
.align 2  
vec2: .space 400
```

Alinea `vec2` a
l'adreça
0x1001000C,
múltiple de 4

(@vec1 = 0x10010000)

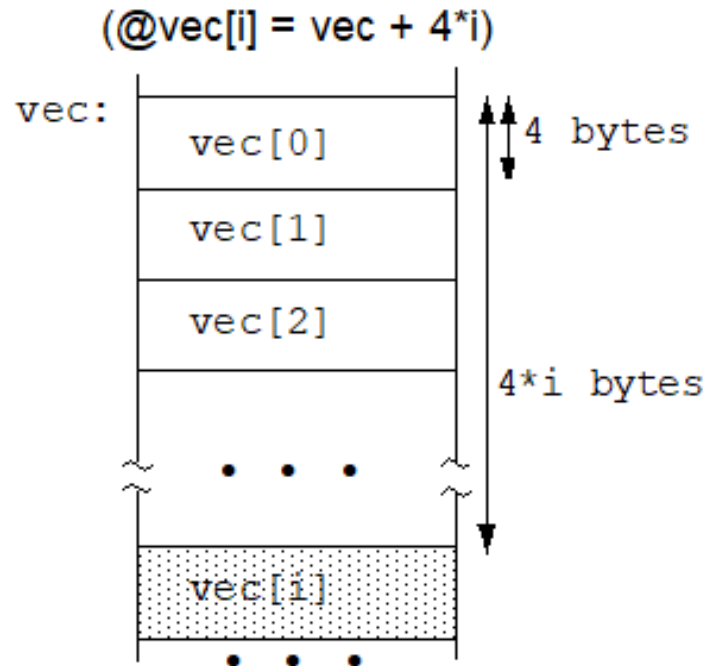


Accés a un element d'un vector

- Per accedir a l'element i -èssim
 - N'hem de calcular l'adreça
 - Suposant elements de mida = T bytes

$$\text{@vec}[i] = \text{@vec}[0] + i \cdot T$$

- Exemple (suposem que vec és global i $T=4$)



Accés a un element d'un vector

- Exemple, amb un índex **constant**: `vec[3]`

```
int vec[100];  
main() {  
    int x; // suposem que x es guarda en $t1  
    x = vec[3];  
}
```

- Traducció de la sentència a MIPS

```
la    $t0, vec  
lw    $t1, 12($t0)
```

o bé...

```
la    $t0, vec+12  
lw    $t1, 0($t0)
```

Accés a un element d'un vector

- Exemple, amb un índex **variable**: `vec[i]`

```
int vec[100];  
main() {  
    int x, i; // suposem x, i guardats en $t1, $t2  
    x = vec[i];  
}
```

- Traducció de la sentència a MIPS

<code>la</code>	<code>\$t0, vec</code>	<code># \$t0 = @vec[0]</code>
<code>sll</code>	<code>\$t3, \$t2, 2</code>	<code># \$t3 = i*4</code>
<code>addu</code>	<code>\$t0, \$t0, \$t3</code>	<code># \$t0 = @vec[0] + i*4</code>
<code>lw</code>	<code>\$t1, 0(\$t0)</code>	<code># x = vec[i]</code>


Strings o cadenes de caràcters

- Són vectors amb un nombre variable de caràcters
- Emmagatzemament
 - En Java: com una tupla (un enter i un vector de caràcters)
 - En C: vector de caràcters amb sentinella (caràcter null = '\0')
- Declaracions en C (formes equivalents)

```
char cadena[20] = "Una frase";
```

```
char cadena[20] =
```

```
    {'U','n','a',' ','f','r','a','s','e','\0'};
```



- Traducció a MIPS (formes equivalents)

```
.data
```

```
cadena: .ascii "Una frase"      # 9 caràcters
```

```
        .space 11                # sentinella + 10 zeros
```

```
cadena: .asciiz "Una frase"     # 9 caràcters + sentinella
```

```
        .space 10                # 10 zeros
```


Strings o cadenes de caràcters

- Exemple en C

```
char nom[80];  
main() {  
    int num = 0;  
    ...  
    while (nom[num] != '\0') num +=1;  
}
```

- Traducció a MIPS

```
main:    ...  
         move    $t0, $zero           # num = 0  
         la      $t1, nom  
while:   addu     $t3, $t1, $t0        # $t3 = @nom[0] + num*1  
         lb      $t2, 0($t3)          # $t2 = nom[num]  
         beq     $t2, $zero, fiwhile  
         addiu   $t0, $t0, 1          # num +=1  
         b       while  
fiwhile:  
         ...
```

Punters

- **Punter:** Variable que conté una adreça de memòria
 - 32 bits en MIPS
 - Si el punter p conté l'adreça de la variable v ...
... diem que p *apunta a* v

Declaració de punters

- Declaració dels punters *p1*, *p2* i *p3* en C:

```
int *p1, *p2;      // p1, p2 apunten a variables de tipus int
char *p3;          // p3 apunta a variables de tipus char
```

Alerta amb els tipus!

```
p1 = p2;           // correcte: p1 i p2 del mateix tipus
p1 = p3;           // incorrecte! p1 i p3 de tipus diferents
```

- Declaracions de *p1*, *p2* i *p3* en MIPS

- Si són globals:

```
.data
p1: .word 0
p2: .word 0
p3: .word 0
```

- Si són locals:

- no cal reservar, sols decidir en quin registre es guardaran

Inicialització d'un punter

- Assignant-li un altre punter del mateix tipus
- Assignant-li l'adreça d'una variable (operador &, en C)
 - Si suposem que el punter és global (*pglob*):

```
char a = 'E';  
char b = 'K';  
char *pglob = &a;
```

```
                .data  
a:               .byte 'E'  
b:               .byte 'K'  
pglob:          .word a
```

Inicialització d'un punter

- Assignant-li un altre punter del mateix tipus
- Assignant-li l'adreça d'una variable (operador &, en C)
 - Si suposem que el punter és global (*pglob*):

```
char a = 'E';  
char b = 'K';  
char *pglob = &a;
```

```
void f() {  
    pglob = &b;  
    ...  
}
```

```
.data  
a:      .byte 'E'  
b:      .byte 'K'  
pglob:  .word a  
  
.text  
f:  
  
la      $t0, b  
la      $t1, pglob  
sw      $t0, 0($t1)
```

Inicialització d'un punter

- Assignant-li un altre punter del mateix tipus
- Assignant-li l'adreça d'una variable (operador &, en C)
 - Si suposem que el punter és global (*pglob*):

```
char a = 'E';  
char b = 'K';  
char *pglob = &a;
```

```
void f() {  
    pglob = &b;  
    ...  
}
```

```
.data  
a:      .byte 'E'  
b:      .byte 'K'  
pglob:  .word a  
  
.text  
f:  
  
la      $t0, b  
la      $t1, pglob  
sw      $t0, 0($t1)
```

- Si suposem que el punter és local (*ploc*) i el guardem en **\$t5**:

```
void g() {  
char *ploc; // guardat en $t5  
    ploc = &b;  
    ...  
}
```

```
.text  
f:  
  
la      $t5, b
```

Operació *desreferència* (indirecció) de punters

- Desreferència:
 - Consisteix a accedir a l'adreça de memòria apuntada pel punter
 - En C, usant l'**operador** * precedint el punter
 - *p significa "*contingut de l'adreça de memòria apuntada per p*"
 - No confondre amb el símbol * usat per **declarar** punters!

Operació *desreferència* (indirecció) de punters

- Desreferència:

- Consisteix a accedir a l'adreça de memòria apuntada pel punter
- En C, usant l'**operador** `*` precedint el punter
 - `*p` significa "*contingut de l'adreça de memòria apuntada per p*"
 - No confondre amb el símbol `*` usat per **declarar** punters!
- Exemple: suposem que el punter és global (*pglob*):

```
char a = 'E';  
char *pglob = &a;
```

```
void ff() {  
    char tmp; // guardat en $t0  
    tmp = *pglob;  
    ...  
}
```

```
                .data  
a:              .byte 'E'  
pglob:         .word a  
  
                .text  
ff:  
                la    $t1, pglob  
                lw    $t5, 0($t1)  
                lb    $t0, 0($t5)
```


Operació *desreferència* (indirecció) de punters

- Desreferència:

- Consisteix a accedir a l'adreça de memòria apuntada pel punter
- En C, usant l'**operador** `*` precedint el punter
 - `*p` significa "*contingut de l'adreça de memòria apuntada per p*"
 - No confondre amb el símbol `*` usat per **declarar** punters!
- Exemple: suposem que el punter és global (*pglob*):

```
char a = 'E';  
char *pglob = &a;
```

```
void ff() {  
    char tmp; // guardat en $t0  
    tmp = *pglob;  
    ...  
}
```

```
        .data  
a:      .byte 'E'  
pglob:  .word a  
  
        .text  
ff:  
        la    $t1, pglob  
        lw    $t5, 0($t1)  
        lb    $t0, 0($t5)
```

- Exemple: suposem que el punter és local (*ploc*), guardat en **\$t5**:

```
void gg() {  
    char tmp, *ploc; //en $t0,$t5  
    tmp = *ploc;  
}
```

```
        .text  
gg:  
        lb    $t0, 0($t5)
```

Operació aritmètica de punters

- Aritmètica de punters

- Suma d'un punter p més un enter N
- Dóna com a resultat un altre punter q del mateix tipus: $q = p + N$
- q apunta a una adreça situada **N elements més endavant**
- Si els elements tenen mida T , cal sumar **$N * T$ bytes** a l'adreça

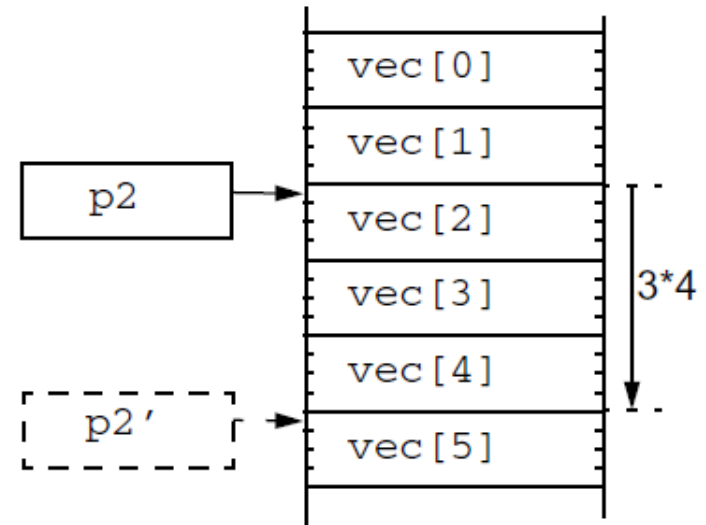
- Exemple

- Suposem que $p2$ s'ha inicialitzat així

```
int *p2 = &vec[2];
```

- i executem la següent operació

```
p2 = p2 + 3;
```



Operació aritmètica de punters

- Exemple

- Suposem les següents declaracions en C

```
char *p1;           // guardat en $t1
int *p2;            // guardat en $t2
long long *p3;      // guardat en $t3
```

- I les sentències...

```
p1 = p1 + 3;
p2 = p2 + 3;
p3 = p3 + 3;
```

- La traducció a MIPS seria...

```
addiu $t1, $t1, 3
addiu $t2, $t2, 12
addiu $t3, $t3, 24
```

Relació entre punters i vectors en C

En C, un vector és en realitat un punter que apunta al seu primer element

- Per exemple

```
int vec[100];  
int *p;
```

- Els tipus de p i vec són equivalents!
 - Però vec és **constant**, sempre apunta al mateix element
 - En canvi, p és **variable**, pot apuntar a elements diferents

Relació entre punters i vectors en C

En C, un vector és en realitat un punter que apunta al seu primer element

- En conseqüència

1. Podem escriure

```
p = vec;           // Fem que p apunti a vec[0]
```

Relació entre punters i vectors en C

En C, un vector és en realitat un punter que apunta al seu primer element

- En conseqüència

1. Podem escriure

```
p = vec;           // Fem que p apunti a vec[0]
```

2. Als punters també els podem aplicar l'operador []

```
p[8] = 3;          // assignem un 3 a vec[8]
```

Relació entre punters i vectors en C

En C, un vector és en realitat un punter que apunta al seu primer element

- En conseqüència

1. Podem escriure

```
p = vec;           // Fem que p apunti a vec[0]
```

2. Als punters també els podem aplicar l'operador []

```
p[8] = 3;          // assignem un 3 a vec[8]
```

3. Als vectors també els podem aplicar l'operador *

```
*vec = 5;           // assignem un 5 a vec[0]
```

Relació entre punters i vectors en C

En C, un vector és en realitat un punter que apunta al seu primer element

- En conseqüència

1. Podem escriure

```
p = vec;           // Fem que p apunti a vec[0]
```

2. Als punters també els podem aplicar l'operador []

```
p[8] = 3;          // assignem un 3 a vec[8]
```

3. Als vectors també els podem aplicar l'operador *

```
*vec = 5;           // assignem un 5 a vec[0]
```

4. Si p apunta a vec[0], p+i apunta a vec[i]

- per tant, l'expressió

```
*(p+i) = 0;         // assignem un 0 a vec[i]
```

- és equivalent a

```
p[i] = 0;           // assignem un 0 a vec[i]
```