Apunts del Tema 2
Instruccions i tipus de dades bàsics

Joan Manuel Parcerisa

Departament d'Arquitectura de Computadors
Facultat d'Informàtica de Barcelona
Febrer 2022

Licencia Creative Commons

Tema 2. Instruccions i tipus de dades bàsics

2.1, 2.20

1. Introducció a la MIPS ISA

En el curs anterior d'Introducció als Computadors (IC), s'ha estudiat en detall el disseny d'un computador simple (SISP), estudiant la relació entre el llenguatge màquina i el disseny a nivell de circuits lògics digitals. En aquest curs d'Estructura de Computadors (EC) s'aprofundirà en el llenguatge màquina (i assemblador) i s'estudiarà la relació entre aquest i un llenguatge d'alt nivell. Quant al llenguatge màquina, s'utilitzarà a mode d'exemple el del processador MIPS de 32 bits, i quant al llenguatge d'alt nivell, s'utilitzarà el llenguatge C.

1.1 Què és una arquitectura del joc d'instruccions (ISA)?

Anomenem "arquitectura del joc d'instruccions" d'un processador, en anglès Instruction Set Architecture (ISA), a l'especificació de la interfície entre el hardware i el software. La ISA descriu el format de les instruccions que el processador pot entendre i executar i detalla de manera precisa la manera com la seva execució afecta a l'estat del processador. En realitat, el terme té un abast més ampli, ja que descriu tots els aspectes del processador visibles a un programador, incloent els tipus de dades suportades, els registres i l'estat del processador, l'arquitectura i gestió de la memòria, el model d'excepcions i interrupcions, etc. Tots aquests aspectes seran la matèria d'estudi en aquest curs d'EC.

La ISA, en tant que interfície, constitueix un contracte entre els fabricants del hardware i els fabricants del software. Al llarg dels anys, el fabricant d'un processador pot introduir en el seu disseny successives millores, a fi d'augmentar-ne el rendiment. Però s'ha demostrat que comercialment resulta crític mantenir la compatibilitat dels nous processadors amb els programes existents, i per a això cal que el nou disseny respecti punt per punt totes les especificacions del ISA.

1.2 Què són les arquitectures de tipus RISC?

El terme Reduced Instruction Set Computer (RISC[1]) apareix a principis dels anys 80 per designar l'arquitectura de certs processadors dissenyats amb criteris innovadors respecte a la tendència dels processadors existents fins llavors, els quals van passar a anomenar-se per contraposició Complex Instruction Set Computers (CISC).

1. Terme introduït el 1981 per John L. Hennessy a Stanford.

Abans que els processadors RISC fossin populars, els ràpids avenços en la tecnologia VLSI havien permès als arquitectes integrar circuits cada cop més complexos, i adoptar llenguatges assemblador cada cop més potents, amb l'objectiu de reduir la diferència semàntica amb els llenguatges d'alt nivell. Així per exemple, en alguns processadors es podien realitzar amb una sola instrucció de llenguatge màquina operacions tan complexes com avaluar un polinomi en coma flotant, o copiar una cadena de caràcters traduint-los prèviament per mitjà d'una taula de conversió. Per abastar tal varietat i sofisticació de tasques calia un repertori amplíssim d'instruccions, amb un nombre variable d'operands, fent necessari codificar-les amb un format de mida variable. Per especificar la ubicació dels operands es necessitaven molts modes d'adreçament i complexes indireccions de memòria. Algunes instruccions involucraven tantes operacions que per interpretar-les es necessitava executar cada cop un petit programa escrit en microcodi, en una ROM interna. Aquestes arquitectures van ser més tard conegudes com CISC.

Però aquesta tendència va començar a canviar quan alguns estudis estadístics van mostrar que les instruccions més complexes a penes es feien servir en la pràctica, ja que responien a necessitats tan particulars que els compiladors trobaven moltes dificultats per aplicar-les. Aviat es va fer evident que tot i que les instruccions complexes podien resoldre problemes particulars de manera molt eficient, la seva implementació augmentava la complexitat del processador de tal manera que comportava sovint que l'execució de les instruccions més simples es ralentitzés. Si tenim en compte que estadísticament les instruccions complexes constitueixen un percentatge molt petit del temps d'execució, els beneficis que aporten són molt menors que el perjudicis. La filosofia RISC va tendir doncs a suprimir les instruccions complexes i lentes, seguint el principi que la mateixa tasca es pot realitzar amb una seqüència d'instruccions simples, i que suprimir-les del repertori permet dissenyar un hardware més simple, capaç d'executar més ràpidament les altres instruccions ("simplicity favours regularity"). La principal característica de les arquitectures RISC (ARM, PowerPC, MIPS, Alpha, SPARC, etc.), i que les diferenciava de les CISC (x86, PDP-11, VAX, System/360, 68000, etc.), consistia en un joc d'instruccions dissenyat amb el criteri bàsic del rendiment. A pesar del nom RISC, el repertori d'instruccions curt no era la seva característica principal sinó més aviat una conseqüència. El més característic dels RISC era el fet de tenir instruccions de mida fixa, pocs modes d'adreçament (solament operands en registre o bé immediats), i que l'accés a memòria es feia solament amb instruccions específiques anomenades load i store.

Originàriament, els nous microprocessadors de tipus RISC eren estructuralment més simples que els de tipus CISC i estaven formats per un nombre molt més reduït de transistors, característica que els feia més barats, ràpids i energèticament eficients. Moltes de les arquitectures CISC van desaparèixer, però algunes que estaven fermament instal·lades en el mercat (com la x86 d'Intel i AMD) simplement van evolucionar, i amb els anys, les diferències entre CISC i RISC s'han anat difuminant. Per un costat, els processadors x86 actuals converteixen internament, en l'etapa de decodificació, cada una de les instrucions en una o vàries "micro-operacions" del tipus RISC. Per traduir les instruccions més complexes i que produeixen seqüències més llargues de micro-operacions (herència del seu origen CISC, i que no poden eliminar del repertori per no trencar la retrocompatibilitat), el decodificador executa unes rutines de microcodi guardades en una ROM dins el xip. Per una altra banda, a mesura que la dissipació de calor ha frenat l'augment de les freqüències de rellotge, que era un important avantatge de la simplicitat dels RISC, aquestes arquitectures han anat incorporant als seus repertoris noves instruccions per augmentar-ne la funcionalitat. Amb aquestes ampliacions, el repertori ha deixat de ser "reduït" però no per

això s'ha desvirtuat el criteri RISC, que no és pas la llargada del repertori sinó la millora de rendiment global.

1.3 Què és l'arquitectura MIPS?

En aquest curs estudiarem els aspectes bàsics de l'arquitectura MIPS[2]. Va ser proposada el 1981 per l'equip liderat pel Dr. John L. Hennessy a la Universitat de Stanford. Ell mateix va ser un dels fundadors de la companyia MIPS Computer Systems[3] el 1984. El gener de 1986, aquesta companyia va introduir el primer processador basat en aquesta arquitectura, el MIPS R2000. Els processadors MIPS han format part de consoles de jocs tan conegudes com Nintendo 64, Playstation (R3000), PSP (R4000) i PS2 (R5900), i en processadors empotrats (embedded processors) de multitud d'aparells domèstics.

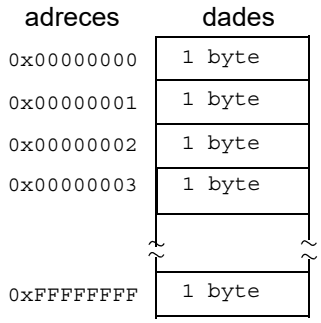
Al llarg dels anys, l'arquitectura MIPS ha sofert successives ampliacions, preservant sempre la compatibilitat amb els codis ja existents: MIPS I, II, III, IV, V, 32, 64, etc. Les versions MIPS-I, MIPS-II, i MIPS32 són de 32 bits, és a dir que els operands són de 32 bits, mentre que la resta són de 64 bits. En EC estudiarem la versió MIPS32.

MIPS32 té 32 registres de 32 bits, així com adreces i dades de memòria de 32 bits. L'espai de memòria és unificat (s'hi emmagatzemen tant les instruccions com les dades) i consta de 2^{32} adreces, cada una identifica una posició de memòria que conté 1 byte. A continuació veurem aquestes característiques amb més detall.

2. La memòria

2.1 Adreçament a nivell de byte

Des del punt de vista del programador la memòria és un vector en què cada element conté 1 byte (8 bits) i s'identifica per una adreça:



Nota: En endavant, en tots els diagrames que representen la memòria, pressuposarem que les adreces apareixen en ordre creixent de dalt cap a baix[4].

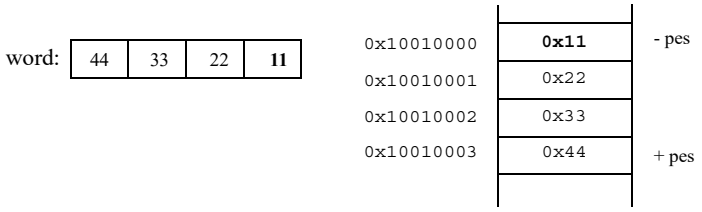
2. Microprocessor without Interlocked Pipeline Stages.
3. Convertida en MIPS Technologies des de 1988 (subsidiària de SGI), i comprada en 2013 per Imagination Technologies (coneguda per la GPU PowerVR que equipa alguns telèfons mòbils de Apple).
4. Naturalment, aquest és un conveni gràfic arbitrari i altres textos, com el llibre recomanat (D.A.Patterson & J.L.Hennessy, *Computer Organization and Design*), poden adoptar just el conveni contrari: les adreces dels diagrames creixen de baix cap a dalt.

2.2 Ordenació de bytes ("byte ordering" o "endianness")

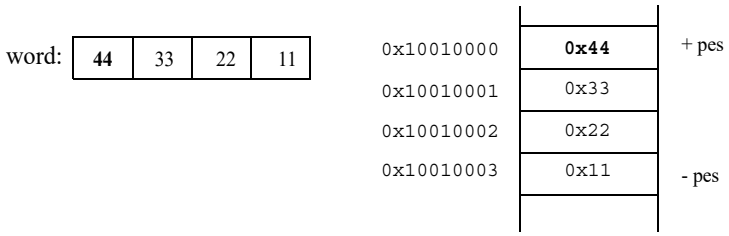
En un processador s'anomena *paraula* o *word* a la dada que té la mida nativa de l'arquitectura, i que normalment coincideix amb la llargada dels registres (p.ex. en MIPS 1 word equival a 4 bytes o 32 bits). Quan guardem en memòria un número format per múltiples bytes (com per exemple un word), aquest ocuparà diverses posicions de memòria consecutives a partir d'una adreça inicial, i el processador accedirà al word complet per mitjà d'una sola instrucció de *load* o *store* que especificarà tan sols l'adreça inicial. Però cal decidir en quin ordre es guarden els bytes que componen el word. De les múltiples permutacions possibles, n'hi ha dues que històricament s'han usat de manera més freqüent: Little-endian i Big-endian.

EXEMPLE 1. Suposem que guardem el número `0x44332211` (4 bytes) en les quatre posicions ubicades a partir de l'adreça `0x10010000`:

a) Little-endian: coloquem en l'adreça més baixa el byte de menys pes (el de més a la dreta quan l'escrivim), i després, consecutivament, la resta de bytes.



b) Big-endian: coloquem en l'adreça més baixa el byte de més pes (el de més a l'esquerra quan l'escrivim), i després, consecutivament, la resta de bytes.



Històricament es va produir una certa controvèrsia teòrica sobre quin dels dos mètodes era més eficient. En les èpoques en què la informació es transmetia de byte en byte, accedir en primer lloc al de menor pes permetia iniciar tempranament una suma, mentre que accedir en primer lloc al de major pes permetia iniciar tempranament una comparació. A mesura que els computadors han evolucionat, aquesta discussió ha esdevingut tan estèril com la discussió que Johnathan Swift relata en *Els viatges de Gulliver*, entre els habitants de Lilliput, partidaris d'obrir un ou passat per aigua pel seu costat més prim (little-end) i els de Blefuscu, partidaris de fer-ho pel seu costat més gros (big-end), d'aquí els noms.

Si bé antigament la competència comercial entre fabricants induïa a crear barreres d'incompatibilitat entre ells adoptant formats de dades diferents, en l'era actual de la comunicació es considera una dificultat a superar. No és doncs estrany que en la ISA del MIPS no s'hi definís una ordenació específica (little o big endian). La majoria de fabricants que adopten l'arquitectura MIPS implementen mecanismes en el processador per permetre configurar-lo amb una o altra ordenació durant l'arrencada. En aquest curs, utilitzarem l'ordenació little-endian[5] en tots els exemples i exercicis.

3. Variables

3.1 Visibilitat i durada de variables locals i globals

En C, les variables s'han de declarar explícitament abans de poder-les usar en el codi. La declaració ha d'incloure el tipus (*int*, *char*, etc.) i el nom, i opcionalment un valor inicial. Una variable que es declara fora de qualsevol funció és *global* (*extern*, en la nomenclatura de C) i una que es declara dins d'un bloc o funció és *local* al bloc o funció. Aquestes dues classes de variables tenen assignades per defecte algunes propietats de *visibilitat* i també de *durada* (o *categoria d'emmagatzemament*, en la nomenclatura de C):

Per defecte, les variables **globals** en C tenen visibilitat global, és a dir que poden ser usades per qualsevol funció en qualsevol fitxer pertanyent al programa[6]; i tenen durada permanent (tenen categoria *static*, en la nomenclatura de C), és a dir que mantenen la seva existència i poden ser accedides durant tot el temps que dura l'execució del programa. Per garantir aquesta permanència, aquestes variables s'emmagatzemen en posicions de memòria fixes, dedicades a elles durant tot el programa. Si la declaració omet un valor inicial, aquest és zero, per defecte. En assemblador, la seva declaració ha d'aparèixer a la secció `.data`, i normalment la situarem al principi del programa.

Per defecte, les variables **locals** en C tenen visibilitat local, és a dir que sols poden ser accedides dins de la funció o bloc on es declaren, i durada temporal (*automatic*, en la nomenclatura de C), és a dir que es creen cada cop que el bloc o la funció s'invoca, i deixen d'existir així que finalitza[7]. Donada aquesta volatilitat, no es reserva un espai d'emmagatzematge dedicat a aquestes variables, sinó que cada cop que es creen cal reservar-los un espai lliure, ja sigui en memòria o bé en un registre del processador. Si la declaració omet un valor inicial, aquest resta indeterminat fins que se li assigni algun valor. En assemblador no requereixen cap declaració explícita. S'estudiaran en detall al Tema 3, no obstant apareixeran ja en els exemples d'aquest mateix tema, en forma de variables escalars que s'emmagatzemen en registres.

3.2 Mida de les variables

En C, la mida en bytes que ocupa una variable depèn del seu tipus però no està prefixada pel llenguatge, sinó que depèn de cada ISA on s'implementa. En aquest curs, per a l'arquitectura MIPS farem les següents suposicions per als tipus enters: *char* ocupa 1 byte, *short* 2 bytes, *int* 4 bytes i *long long* 8 bytes. I farem anàlogues suposicions sobre la mida dels naturals, que es declaren precedint cada un dels anteriors tipus enters de la paraula *unsigned*.

5. S'ha triat així per coherència amb el simulador MARS utilitzat al laboratori. Cal parar atenció però, ja que el llibre de text recomanat (D.A.Patterson & J.L.Hennessy, *Computer Organization and Design*) pressuposa l'ordenació big-endian. No obstant, són pocs els exemples on això té alguna repercussió.
6. La visibilitat de les variables globals es pot restringir al fitxer on estan declarades precedint la declaració amb el modificador *static*, però no ho estudiarem.
7. La durada de les variables locals (però no la seva visibilitat) es pot convertir en permanent precedint la declaració amb el modificador *static*, però no ho estudiarem.

3.3 Declaració i alineació de variables emmagatzemades en memòria

En general, les variables que s'emmagatzemen en memòria (com per exemple les variables globals), ocupen posicions de memòria consecutives seguint l'ordre textual de les declaracions, aprofitant al màxim l'espai disponible. Però en MIPS, en el cas de variables escalars de mida major que 1 byte (o vectors d'elements de més de 1 byte), l'adreça inicial de la variable ha de respectar certes restriccions d'alineació: un *short* en una adreça múltiple de 2, un *int* en una múltiple de 4, i un *long long* en una múltiple de 8. Si convé, es deixen posicions de memòria sense usar, a fi de respectar aquesta regla.

En MIPS, les directives `.byte`, `.half`, `.word` i `.dword` reserven 1, 2, 4 o 8 bytes de memòria, respectivament. A continuació de cada directiva s'escriu el valor inicial de la variable (0 en cas de variables globals sense inicialitzar). Opcionalment, la directiva pot anar precedida d'una etiqueta amb el nom de la variable, i aquesta passa a representar de forma simbòlica l'adreça inicial de la dada definida.

Les directives `.half`, `.word` i `.dword` apliquen automàticament la restricció d'alineació. Cada directiva intenta reservar posicions de memòria a continuació de la dada anterior, seguint adreces correlatives. Però si l'adreça que li pertoca no és múltiple de 2, 4 o 8 respectivament llavors descartarà tantes posicions com faci falta i començarà la reserva d'espai a partir de la següent adreça alineada disponible. Si la directiva va precedida d'una etiqueta, a aquesta se li assignarà la nova adreça alineada.

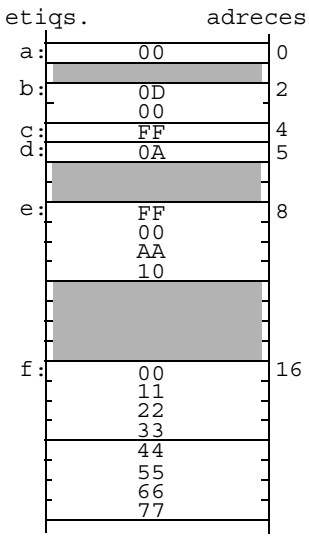
`.byte n`	Reserva 1 byte i l'inicialitza a n
`.half n`	Reserva 1 halfword alineat en una adreça múltiple de 2 i l'inicialitza a n
`.word n`	Reserva 1 word alineat en una adreça múltiple de 4 i l'inicialitza a n
`.dword n`	Reserva 1 doubleword alineat en una adreça múltiple de 8 i l'inicialitza a n

EXEMPLE 2. Suposem les següents declaracions de variables globals en C:

```
unsigned char a;           /* 1 byte    */
short b = 13,              /* 2 bytes   */
char c = -1, d = 10;       /* 1+1 bytes */
int e = 0x10AA00FF;        /* 4 bytes   */
long long f = 0x7766554433221100; /* 8  */
```

En MIPS, les declaracions es tradueixen així

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



Al diagrama de la dreta es pot veure com queden definides les etiquetes i el contingut de la memòria (en sombrejat, les posicions descartades).

Quan declarem vectors, ens caldrà reservar un espai de memòria d'una mida arbitrària (la del vector). Si la declaració del vector inclou uns valors inicials dels elements, usarem una de les directives anteriors, segons la mida dels elements. En aquest cas, en comptes d'un sol valor inicial, la directiva anirà seguida de tants valors inicials com elements tingui el vector, separats per comes; i reservarà espai en

posicions estrictament consecutives de memòria per a tants elements com valors inicials tingui la directiva.

EXEMPLE 3. Suposem la declaració global en C:

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

La traducció en MIPS seria:

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

Però si el vector és molt llarg i no està inicialitzat explícitament (per defecte tots els elements són zero), usar les anteriors directives pot resultar en un codi font exageradament llarg. En aquest cas usarem la directiva `.space n`, la qual reserva `n` bytes de memòria i els inicialitza a zero. Per altra banda, recordem que si el vector consta d'elements de mida multibyte (halfword, word o doubleword), cal assegurar que l'adreça inicial del vector estigui alineada, és a dir que sigui múltiple de 2, 4 o 8 respectivament. Però la directiva `.space` no força cap alineació concreta ja que no s'hi especifica la mida dels elements. Així doncs, si ens cal assegurar una determinada alineació, caldrà que precedim la declaració `.space` amb la directiva `.align n`, on n=1, 2, 3... (atenció, n no ha de ser 0!). Aquesta directiva assegura que la propera adreça usada estigui alineada (sigui múltiple de 2^n), forçant a descartar els bytes de memòria que calguin en cas que no ho estigui.

EXEMPLE 4. Suposem les declaracions globals en C:

```
char a;                    /* 1 byte */
int v[100];                /* 100 words = 400 bytes */
```

EN MIPS, la declaració del vector `v` requereix una alineació explícita:

```
        .data
a:      .byte  0
        .align 2            # Alinear a múltiple de 4 (salta 3 bytes)
v:      .space 400          # Vector de 100 enters
```

`.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

2.2, 2.3, 2.10

4. Operands

La majoria d'instruccions en MIPS, al igual que en la majoria de processadors RISC, tenen 3 operands explícits (principi de disseny *simplicity favors regularity*: la regularitat en el disseny afavoreix implementacions simples i que s'executen en menys temps), encara que algunes instruccions poden tenir menys operands o fins i tot cap.

La manera com s'especifica un operand en una instrucció rep el nom de *mode d'adreçament*. Els operands en MIPS poden tenir fins a 5 modes d'adreçament: mode registre, mode immediat, mode memòria, mode pseudodirecte i mode relatiu al PC. Els tres primers s'explicaran en els següents apartats, els altres dos en el Tema 3.

4.1 Operands en mode registre

En mode registre, l'operand resideix en un registre, i la instrucció sols especifica el número de registre. Vegem, per exemple, les dues instruccions bàsiques de suma i resta:

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

Les instruccions `addu` i `subu` només tenen operands en mode registre. En instruccions amb 3 operands, el primer acostuma a ser l'operand-destinació i els altres dos els operands-font.

MIPS té 32 registres de 32 bits cadascun per a propòsit general. En el llenguatge assemblador es poden fer constar de dues maneres: pel número o pel nom (però als programes de l'assignatura usarem exclusivament el nom):

	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`	*Arguments* de subrutines
`$8-$15`	`$t0-$t7`	*Temporals*, no preservats per les crides a subrutines
`$16-$23`	`$s0-$s7`	*Saved* ("segurs"), 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 nucli (*Kernel*) del SO (convé no usar-los)
`$28`	`$gp`	*Global pointer*, explicat al Tema 3 (no l'usarem)
`$29`	`$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

4.2 Operands en mode immediat

En mode immediat, l'operand es codifica en la pròpia instrucció. Algunes instruccions admeten un operand immediat, on generalment s'hi especifica un literal de 16 bits.

Vegem, per exemple, les tres següents instruccions[8]:

`addiu/subu`		
`addiu rt, rs, imm16`	rt = rs + SignExt(imm16)	Immediat de 16 bits en Ca2
`lui rt, imm16`	$rt_{31..16}$ = imm16 $rt_{15..0}$ = 0x0000	Posa a zero la part baixa!
`ori rt, rs, imm16`	rt = rs OR ZeroExt(imm16)	L'immediat s'extén amb zeros

EXEMPLE 5. Sigui la següent sentència en C:

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

Suposant que *f*, *g*, *h*, *i*, són variables locals enteres emmagatzemades als registres $t0, $t1, $t2, $t3, respectivament, la traducció a assemblador, serà:

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

4.3 Ús de pseudoinstruccions o macros

Sovint en un programa ens cal copiar una dada d'un registre a un altre. No hi ha una instrucció dedicada a això ja que es pot fer amb algunes de les que hem vist anteriorment, per exemple amb `addu` (amb un operand a zero). No obstant, fer servir una suma per copiar un registre dificulta la lectura del programa. A fi de facilitar la lectura i depuració, el programa assemblador disposa de la capacitat de definir *macros* o *pseudoinstruccions*, les quals amplien el repertori. Una macro té la forma d'una instrucció i una sintaxi precisa. En el moment de ser assemblada, es converteix (diem que *s'expandeix*) en una o més instruccions del repertori original MIPS.

a) La macro **move.** Per a la còpia entre registres s'ha definit la macro `move`, la qual s'expandeix en una instrucció `addu` que té un operand a zero:

`move rdest, rsrc`	`rdest = rsrc`	`addu rdest, rsrc, $zero`

EXEMPLE 6. Còpia del registre $t2 a $t1:

```
move   $t1, $t2                   # Expansió:    addu $t1,$t2,$zero
```

b) La macro **li** (load immediate). Serveix per inicialitzar un registre amb un valor literal. S'expandeix de manera diferent segons si el literal és de 16 o 32 bits:

`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)`

EXEMPLE 7. Copiar en $t1 un literal de 16 bits. S'expandeix en 1 instrucció

```
li     $t1, 100                   # Expansió:    addiu $t1,$zero,100
```

EXEMPLE 8. Copiar en $t1 un literal de 32 bits. S'expandeix en 2 instruccions

```
li     $t1, 0x0030D900            # Expansió:    lui $at,0x0030
                                  #              ori $t1,$at,0xD900
```

8. En aquelles instruccions que MIPS executa extenent el rang de l'immediat de 16 a 32 bits fent extensió de signe (`addiu`, `slti`, `sltiu`, `lw`, `sw`, i restants instruccions de memòria), els immediats han de ser enters representables en Ca2. En aquelles instruccions on no s'extén el rang (`lui`) o bé s'extén amb zeros (`ori`, `andi`, `xori`), els immediats han de ser números naturals de 16 bits.

   Si escrivim un immediat en hexadecimal amb menys de 8 dígits, l'assemblador incorporat al Mars el considera sempre positiu. Així doncs, 0xFFF és positiu i equival a 0x0FFF (per extensió de zeros). Però 0xFFFF i 0x8000 equivalen a 65535 i 32768 respectivament: com a naturals són representables en 16 bits, però com a enters són números positius fora del rang representable. Si es vol expressar un immediat negatiu, cal afegir-li el signe de forma explícita. Així doncs, -0x8000 és un enter de 16 bits vàlid però -0x8001 està fora del rang.

   Per contra, si escrivim l'immediat amb 8 dígits hexadecimals, Mars el considera ja codificat (com a enter o com a natural, segons la instrucció), i simplement el redueix a un valor equivalent representable amb 16 bits, si existeix. Així doncs, si escrivim 0xFFFF8000 en un `addiu`, Mars l'interpreta com el número enter -32768, el qual és vàlid ja que es pot representar amb 16 bits com 0x8000.

La instrucció `lui` (load upper immediate) copia els 16 bits de l'immediat a la part alta de $at i posa els 16 bits baixos a zero (`$at=0x00300000`). La instrucció `ori` (logical or immediate) extén l'immediat `0xD900` de 16 a 32 bits afegint zeros a la part alta (`0x0000D900`), i després fa la o lògica d'aquest resultat amb el registre $at (`$t0=0x0030D900`). El resultat final de la macro és que s'ha copiat el literal en $t1.

c) La macro **la** (load address). Serveix per inicialitzar un registre amb una adreça de memòria constant, estant aquesta expressada com una etiqueta.

Hem vist que les variables globals es declaren en assemblador amb una directiva precedida d'una etiqueta. Aquesta etiqueta equival a l'adreça de la variable. Sovint necessitem copiar aquesta adreça, que és un número natural de 32 bits, en un registre. La macro `la` s'expandeix igual que la macro `li`, però es diferencia en la sintaxi. L'operand ha de ser una etiqueta del programa o bé una etiqueta sumada a un literal:

EXEMPLE 9. Carregar l'adreça de *y* en el registre $t0, i l'adreça *y*+4 en $t1.

```
        .data
y:      .word  14, -1        # Suposem que y està a l'adreça 0x10010024

        .text
la      $t0, y               # Expansió:          lui $at, 0x1001
                             #                    ori $t0, $at, 0x0024
la      $t1, y+4             # Expansió:          lui $at, 0x1001
                             #                    ori $t0, $at, 0x0028
```

4.4 Constants simbòliques

En un programa podem declarar constants simbòliques. Serveixen per facilitar l'escriptura de programes, ja que llavors es pot modificar una constant que apareix en múltiples llocs fent la modificació una sola vegada. A més a més, facilita la lectura del programa. En C les constants simbòliques es declaren mitjançant la directiva `#define`:

```
#define N 10
```

En assemblador MIPS[9] les constants simbòliques es declaren mitjançant una simple assignació al principi del programa amb la directiva `.eqv`, abans de la secció `.data`:

```
.eqv   N, 10
```

Un cop definida, es pot usar en qualsevol context on usaríem el literal equivalent

```
li      $t0, N                    # equival a li $t0, 10
```

4.5 Operands en mode memòria (instruccions del tipus *load* i *store*)

En processadors RISC com el MIPS, solament les instruccions del tipus *load* i *store* admeten un operand resident en memòria. Diem llavors que aquest operand està codificat en un mode d'adreçament de memòria. El repertori de MIPS té diverses variants d'instruccions de *load* i *store*, per accedir a dades de mida word (4 bytes), halfword (2 bytes) o byte, ja siguin enters amb signe o sense.

9. El simulador MARS admet la directiva `.eqv simbol, valor` a partir de la versió 4.3. L'assemblador as de GNU reconeix les directives anàlogues `.equ` o `.equiv`, i l'assemblador MIPSPro de Silicon Graphics usa l'assignació `simbol=valor`.

La següent taula mostra totes les variants:

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

La sintaxi és la mateixa en totes les instruccions d'aquest tipus. Per exemple:

```
lw     rt, offset16 (rs)          # rt = Mw[rs + SignExt(offset16)]
sw     rt, offset16 (rs)          # Mw[rs + SignExt(offset16)] = rt
```

En tots els casos, l'adreça efectiva es calcula sumant el registre-base *rs* amb el desplaçament *offset16*. El desplaçament és un immediat codificat en Ca2 amb 16 bits (abasta el rang de valors [-2^{15}, $+2^{15}-1$]). MIPS l'extén a 32 bits abans de sumar-lo. Recordem que en Ca2, l'extensió del rang rep el nom d'*extensió de signe* ja que es fa replicant el bit de signe (bit 15) en els bits restants de major pes (veure l'apartat 6.1(e) per a més detalls).

a) Accessos a word: *lw*, *sw*

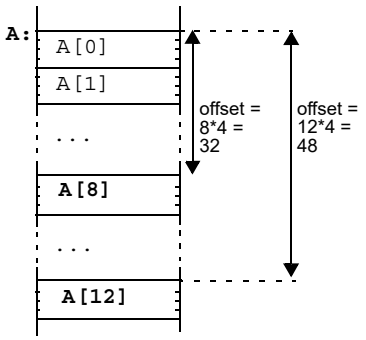
La instrucció *lw* copia 4 bytes consecutius de la memòria, a partir de l'adreça efectiva, al registre destinació. I *sw* copia els 4 bytes del registre font a altres tantes posicions consecutives de memòria a partir de l'adreça efectiva.

EXEMPLE 10. Traduir a MIPS la següent assignació de valors enters (words) en C:

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

En MIPS, suposant que *h* ocupa $t2, i que A és una variable global, l'adreça d'A[8] és la suma de l'adreça base A i l'offset 8*4=32. L'adreça d'A[12] és la suma de A més l'offset 12*4=48:

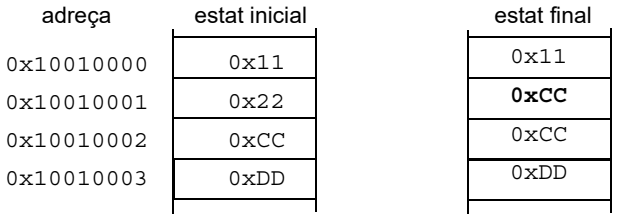
```
la     $t3, A
lw     $t0, 32($t3)      # $t0 = A[8]
addu   $t0, $t2, $t0     # $t0 = h + $t0
sw     $t0, 48($t3)      # A[12] = $t0
```



b) Accessos a halfword o byte (enters amb signe): *lh*, *lb*, *sh*, *sb*

La instrucció *lh* copia un halfword (2 bytes) de la memòria als 16 bits de menor pes del registre destinació, i realitza una extensió de signe (extén el bit 15) de la dada transferida. Anàlogament, la instrucció *lb* copia 1 byte de la memòria als 8 bits de menor pes del registre destinació i realitza una extensió de signe (extén el bit 7). La instrucció *sh* copia a la memòria els 2 bytes de menor pes del registre font, i la instrucció *sb* copia a la memòria només el byte de menor pes del registre font.

EXEMPLE 11. Suposem que $t2 conté l'adreça `0x10010000`, i que en aquesta adreça i les següents el contingut inicial de la memòria és el que es mostra al dibuix (esquerra).



Determina el resultat de cada instrucció i l'estat final de memòria (dibuix a la dreta)

```
lb     $t1, 1($t2)          # load byte 0x22.    $t1 = 0x00000022
lb     $t1, 2($t2)          # load byte 0xCC.    $t1 = 0xFFFFFFCC
lh     $t1, 0($t2)          # load half 0x2211.  $t1 = 0x00002211
lh     $t1, 2($t2)          # load half 0xDDCC.  $t1 = 0xFFFFDDCC
sb     $t1, 1($t2)          # store byte 0xCC a l'adreça 0x10010001
```

c) Loads de halfword o byte (naturals): *lhu*, *lbu*

Les instruccions *lhu* (load halfword unsigned) i *lbu* (load byte unsigned) serveixen per llegir de memòria dades de mida halfword o byte que no són enters sinó naturals. L'extensió del rang de representació es fa omplint els bits de més pes amb zeros.

EXEMPLE 12. Amb el mateix contingut inicial de la memòria de l'exemple anterior

```
lbu    $t1, 2($t2)          # load byte 0xCC.     $t1 = 0x000000CC
lhu    $t1, 2($t2)          # load half 0xDDCC.   $t1 = 0x0000DDCC
```

d) Restricció d'alineació

Les adreces utilitzades en les instruccions `lw` i `sw` han de ser múltiples de 4, i les adreces usades en `lh` i `sh` han de ser múltiples de 2. Altrament, es produeix una excepció per adreça no alineada i el programa finalitza (veurem les excepcions en detall al tema 8).

EXEMPLE 13. El codi següent en MIPS executa un `lw` a una adreça no-alineada

```
     .data
x:   .word   0xDDCCAABB

     .text
     la     $t0, x+3            # $t0 = 0x10010003
     lw     $t1, 0($t0)         # excepció d'accés no alineat!
```

2.4

5. Repàs de representació de naturals en base 2

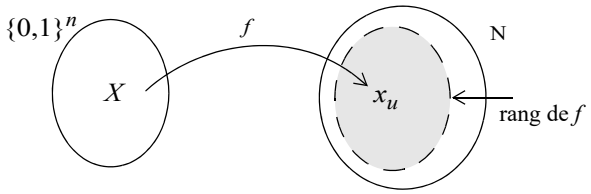
Denotem: $X = X_{n-1}...X_1X_0$ vector de n bits, $X \in \{0, 1\}^n$

x_u = nombre natural (unsigned) representat per X, $x_u \in \mathrm{N}$

I definim la funció $x_u = f(X)$ que assigna un nombre natural a cada cadena de bits

$$\{0,1\}^n \to N$$
$$f: \mathrm{X} \to x_u$$

Podem il·lustrar-ho gràficament així:



a) Interpretar *X* com a natural

Consisteix en calcular la seva imatge per la funció f, és a dir $x_u = f(X)$:

$$x_u = \sum_{i=0}^{n-1} X_i \cdot 2^i \qquad \textbf{(eq1)}$$

b) Representar un natural x_u en binari

Consisteix en calcular la seva imatge per la funció inversa f^{-1}, és a dir $X=f^{-1}(x_u)$. Els dígits $X_{n-1}...X_0$ s'obtenen dels residus de dividir successivament x_u entre 2 (n vegades, o fins que el quocient sigui zero):

Q_0	$= x_u/2,$	$X_0 = x_u \bmod 2$
Q_1	$= Q_0/2,$	$X_1 = Q_0 \bmod 2$
...		...
Q_{n-2}	$= Q_{n-3}/2,$	$X_{n-2} = Q_{n-3} \bmod 2$
Q_{n-1}	$= Q_{n-2}/2 = 0,$	$X_{n-1} = Q_{n-2} \bmod 2$

(eq2)

Demostració

Si traiem el 2 factor comú del sumatori

$$x_u = X_{n-1}\cdot 2^{n-1} + ... + X_1\cdot 2^1 + X_0\cdot 2^0 = 2\cdot(X_{n-1}\cdot 2^{n-2} + ... + X_1\cdot 2^0) + X_0$$

es fa evident que si dividm x_u per 2 obtenim de residu (R_0) i quocient (Q_0)

$$R_0 = X_0, \qquad Q_0 = X_{n-1}\cdot 2^{n-2} + ... + X_1\cdot 2^0$$

Dividint novament Q_0 per 2 obtenim de residu (R_1) i quocient (Q_1)

$$R_1 = X_1, \qquad Q_1 = X_{n-1}\cdot 2^{n-3} + ... + X_2\cdot 2^0$$

I així successivament...

$$R_{n-2} = X_{n-2}, \quad Q_{n-2} = X_{n-1}\cdot 2^0$$
$$R_{n-1} = X_{n-1}, \quad Q_{n-1} = 0$$

És a dir que els residus de les n divisions successives de x_u entre 2 donen, per aquest ordre, els bits X_0 ... X_{n-1} de la seva representació X.

EXEMPLE 14. Representar $x_u = 25$ en binari, amb 8 bits:

```
25/2 = 12     residu = 1  (bit de menys pes)
12/2 = 6      residu = 0
6/2 = 3       residu = 0
3/2 = 1       residu = 1
1/2 = 0       residu = 1
0/2 = 0       residu = 0
0/2 = 0       residu = 0
0/2 = 0       residu = 0  (bit de més pes)
```

Resultat: $X = 00011001_2$

c) Rang de representació

El rang de *f* és: $x_u \in [0, 2^n\text{-}1]$. Per exemple, per a n=8 el rang és: $x_u \in [0, 255]$

d) Extensió de zeros

Donada la representació X d'un natural x_u amb n bits, es demana obtenir la seva representació equivalent amb n+1 bits (amb un rang major). La regla l'anomenem *extensió de zeros* i consisteix a afegir-li un bit X_n=0. Afegir un zero a l'esquerra no altera el valor representat, ja que no altera el valor final del sumatori en eq1.

Demostració

Si expressem x_u en funció de X per a n i per a n+1 bits segons eq1, veurem que són equivalents, ja que X_n=0:

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

Com que els zeros inicials no alteren el valor representat, l'algorisme de divisions successives (eq2) no necessita fer sempre n divisions, tan sols fins a obtenir el primer quocient a zero. Com que les divisions restants donaran zeros, els podem afegir directament. En l'exemple 14, després de les 5 primeres divisions obtenim quocient zero i el resultat és 11001_2. Les divisions següents tan sols serveixen per fer l'extensió de zeros: 00011001_2.

e) Declaració de variables de tipus natural (enters "unsigned", en C)

En C:

```
unsigned char var1 = 0;                 // 1-byte
unsigned short var2 = 0;                // 2-bytes
unsigned int var4 = 0;                  // 4-bytes
unsigned long long var8 = 0;            // 8-bytes
```

En MIPS, en el cas de variables globals:

```
.data
var1: .byte 0
var2: .half 0
var4: .word 0
var8: .dword 0
```

2.4

6. Repàs de representacions d'enters en base 2

Denotem:

$X = X_{n-1}...X_1X_0$ vector de n bits, $X_i \in \{0, 1\}$

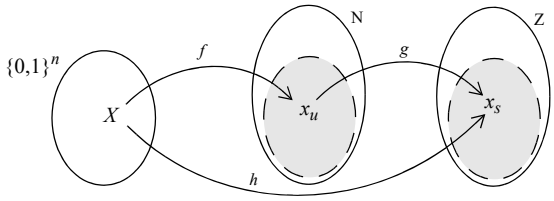
x_s = nombre enter representat per X, $x_s \in \mathrm{Z}$

x_u = nombre natural representat per X, $x_u \in \mathrm{N}$

I definim la funció $x_s = h(X)$ que assigna un nombre enter a cada cadena de bits

$$\{0,1\}^n \rightarrow Z$$
$$h: \mathrm{X} \rightarrow x_s$$

Una manera simple i convenient de definir h és descomponent-la com $h = g \circ f$ essent f la funció que assigna a cada vector de bits X un nombre natural x_u (segons eq1), i essent g la funció que assigna a cada natural x_u l'enter x_s que té la mateixa representació segons h. Al nombre natural x_u que té la mateixa representació que x_s se l'anomena també "valor explícit de x_s"

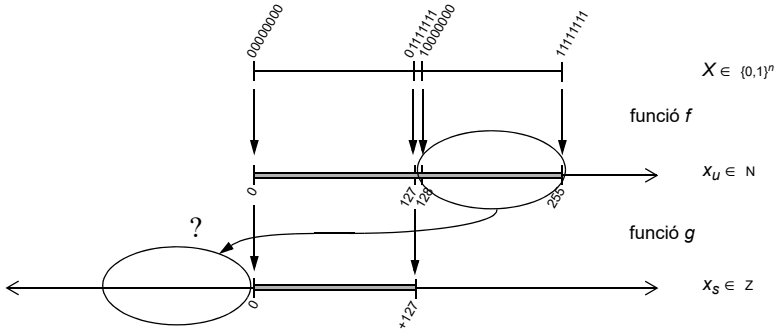


Segons això $x_s = h(X)$ equival a $x_s = g(x_u) = g(f(X))$. Com que f és una funció ja coneguda, en comptes de donar una definició de h directament, ho podem fer indirectament, simplement definint la funció g entre els naturals i els enters.

Es desitja que la funció h tingui entre altres les següents propietats:

- La representació dels enters positius ha de ser igual que la dels naturals
- El bit de més pes de la representació X ha d'indicar el signe: 0 si és positiu, 1 si és negatiu. L'anomenem *bit de signe*.

També podem representar f i g sobre un diagrama amb els 3 conjunts representats per rectes. Així per exemple, per a n=8 bits, les anteriors condicions impliquen que les cadenes de bits del 00000000_2 al 01111111_2 (on el bit de més pes és 0), les quals s'usen per representar els naturals del 0 al 127, representin també als enters positius del 0 al 127:

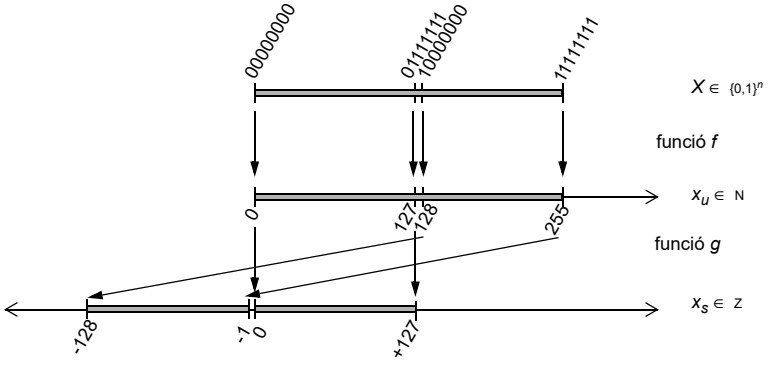


Queda per decidir com assignarem l'altra meitat de les representacions, les que tenen el bit de més pes a 1, als valors enters negatius. Estudiarem tres possibilitats, començant per la més comú i que rep el nom de "Complement a 2", però també revisarem les

codificacions "Complement a 1" i "Signe i Magnitud", que també compleixen les dues propietats abans enunciades. I finalment estudiarem la codificació "En excés", que no requereix satisfer aquestes dues propietats.

6.1 Enters en Complement a 2 (Ca2)

La codificació en Ca2 consisteix a assignar les representacions que tenen el bit de més pes a 1 als enters del -2^{n-1} al -1. Així per exemple, per a n=8 bits, assignaríem les representacions de la 10000000_2 a la 11111111_2 (que corresponen als naturals del 128 al 255), per aquest ordre, als enters del -128 al -1:



Com es pot veure gràficament, la relació entre naturals i enters negatius consisteix en un simple desplaçament. Per als enters negatius de 8 bits es compleix que $x_s = x_u - 256$.

a) Regla d'interpretació: En general, per a *n* bits, la representació en Ca2 es caracteritza per la seva funció *g* que ens permet calcular un nombre enter x_s, donat el seu valor explícit x_u. És a dir, $x_s = g(x_u)$ segons la següent regla:

$$x_s = \begin{cases} x_u & , \text{ si } \; x_u < 2^{n-1} \quad (X_{n-1} = 0) \\ x_u - 2^n & , \text{ si } \; x_u \geq 2^{n-1} \quad (X_{n-1} = 1) \end{cases} \qquad \textbf{(eq3)}$$

Que es pot expressar també en un format més compacte així:

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

Tanmateix, també podem expressar x_s directament en funció dels bits de la seva representació, és a dir, $x_s = \underset{n-2}{h}(X)$ segons la següent regla:

$$x_s = -X_{n-1} \cdot 2^{n-1} + \sum_{i=0} X_i \cdot 2^i \qquad \textbf{(eq4)}$$

Demostració

Segons el format compacte de eq3 podem escriure x_s en funció de x_u:

$$\begin{aligned} x_s &= x_u - X_{n-1} \cdot 2^n \\ &= (X_{n-1} \cdot 2^{n-1} + \ldots + X_0 \cdot 2^0) - X_{n-1} \cdot 2^n \\ &= X_{n-1} \cdot (2^{n-1} - 2^n) + \sum_{i=0}^{n-2} X_i \cdot 2^i \\ &= -X_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} X_i \cdot 2^i \end{aligned}$$

Cas pràctic: interpretar X com a enter en Ca2. És a dir, donada una cadena de *n* bits $X=X_{n-1}...X_1X_0$, es demana calcular quin és l'enter x_s representat per X en Ca2. El mètode més directe és aplicant la regla $x_s = h(X)$ segons eq4.

EXEMPLE 15: Interpretar $X = 11001101_2$ (8 bits) com a enter en Ca2. Aplicant eq4:

$$x_s = -1\cdot2^7 + 1\cdot2^6 + 0\cdot2^5 + 0\cdot2^4 + 1\cdot2^3 + 1\cdot2^2 + 0\cdot2^1 + 1*2^0$$
$$= -128 + 64 + 8 + 4 + 1 = -128+77 = -51$$

No obstant, també ho podem fer indirectament, calculant primer el valor explícit $x_u = f(X)$ segons eq1, i calculant després l'enter $x_s = g(x_u)$ segons eq3.

EXEMPLE 16: Ídem, però ara calculant primer el valor explícit x_u (segons eq1):

$$x_u = 1\cdot2^7 + 1\cdot2^6 + 0\cdot2^5 + 0\cdot2^4 + 1\cdot2^3 + 1\cdot2^2 + 0\cdot2^1 + 1*2^0$$
$$= 128 + 64 + 8 + 4 + 1 = 205$$

I després, tenint en compte que $X_7 = 1$, calculem l'enter x_s (segons eq3):

$$x_s = x_u - 256 = 205 - 256 = -51$$

b) Regla inversa, de representació: També podem deduir la regla per a la funció inversa g^{-1}, per a calcular el valor explícit de x_s. És a dir, $x_u=g^{-1}(x_s)$

$$x_u = \begin{cases} x_s & , \text{ si } x_s \geq 0 \\ x_s + 2^n & , \text{ si } x_s < 0 \end{cases} \qquad \textbf{(eq5)}$$

Cas pràctic: representar un enter x_s en Ca2. És a dir, donat un enter x_s, es demana calcular la cadena de bits $X=X_{n-1}...X_1X_0$ que el representa en Ca2, és a dir $X = h^{-1}(x_s)$. És el procés invers al de l'apartat anterior. Com que hem definit $h = g \circ f$, això equival a calcular $X= f^{-1}(g^{-1}(x_s))$. Calculem primer el valor explícit $x_u = g^{-1}(x_s)$ segons eq5, i després calculem la seva representació $X = f^{-1}(x_u)$ per divisions successives segons eq2.

EXEMPLE 17: Representar amb 8 bits l'enter positiu $x_s = +29$.

Solució: El valor explícit (segons eq5) és $x_u = 29$. Després, per divisions successives (eq2), obtenim els següents parells (quocient, residu):

(14,1), (7,0), (3,1), (1,1), (0,1), (0,0), (0,0), (0,0)

Resultat: $X= 00011101_2$

EXEMPLE 18: Representar amb 8 bits l'enter negatiu $x_s = -124$.

Solució: El valor explícit (segons eq5) és $x_u = - 124 + 2^8 = 256 - 124 = 132$. Després, per divisions successives (eq2), obtenim els següents parells (quocient, residu):

(66,0), (33, 0), (16,1), (8,0), (4,0), (2,0), (1,0), (0,1)

Resultat: $X=10000100_2$

c) Propietat: Un gran avantatge de la codificació en Ca2 és que l'algorisme de suma de dues cadenes de bits és idèntic tant si les considerem com a naturals o com enters, de manera que el mateix circuit serveix per als dos casos.

d) Rang de representació

El rang de *h* és $x_s \in [-2^{n-1}, 2^{n-1}-1]$. Per exemple, per a n=8 és x_s ?$\in$ [-128, +127]. El rang no és simètric, i això fa que l'operació "canvi de signe" no és interna al conjunt, ja que l'oposat del -128 és +128, un número fora del rang.

e) Extensió del signe

Donada la representació X d'un enter x_s amb n bits, es demana obtenir la seva representació equivalent amb n+1 bits (amb un rang major). La regla l'anomenem *extensió de signe* i consisteix a afegir un bit a l'esquerra igual al bit de signe, és a dir $X_n=X_{n-1}$.

Demostració

Segons la fórmula (eq4) podem expressar x_s per a n i per a n+1 bits així

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

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

Igualant les dues expressions i eliminant els termes comuns:

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

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

$$X_n = X_{n-1}$$

f) Regla del canvi de signe

Donada la representació $X = X_{n-1} \cdots X_0$ del nombre enter x_s, es demana calcular la representació X' del nombre enter oposat $x' = -x_s$. Si denotem la cadena formada pels seus complementaris $\overline{X} = \overline{X}_{n-1} \cdots \overline{X}_0$, i denotem per $\bar{x}$ l'enter representat per aquesta cadena, es demostra fàcilment que es compleix la següent propietat de l'enter oposat x'

$$x' = \bar{x} + 1 \qquad \textbf{(eq6)}$$

I per tant, podem calcular fàcilment la seva representació complementant els bits i sumant 1

$$X' = \overline{X} + 00...01_2$$

Demostració

Calculem la suma dels enters x_s i $\bar{x}$:

$$x_s + \bar{x} = \left(-X_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} X_i \cdot 2^i\right) + \left(-\overline{X}_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} \overline{X}_i \cdot 2^i\right)$$

$$= -(X_{n-1} + \overline{X}_{n-1}) \cdot 2^{n-1} + \sum_{i=0}^{n-2} (X_i + \overline{X}_i) \cdot 2^i$$

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

$$= -2^{n-1} + (2^{n-1} - 1) = -1$$

I per tant, queda:

$$-x_s = \bar{x} + 1$$

EXEMPLE 19: Sabent que un enter donat x_s = -54 es representa amb 8 bits amb el vector de bits $X = 11001010_2$, volem calcular la representació de $x' = -x_s = 54$. El vector de bits que busquem és:

$X' = \overline{X}+00000001_2 = 00110101_2 + 00000001_2 = 00110110_2$

Cas pràctic. Calcular manualment la interpretació de X com a enter quan té molts bits a 1 a la part alta, fent ús del canvi de signe.

Si X té molts bits a 1 a la part alta, la suma ponderada és costosa de fer a mà, de manera que pot resultar més senzill interpretar X'=$\overline{X}$+1 ja que $\overline{X}$ té pocs bits a la part alta. Per tant, comencem calculant X', l'interpretem com a enter (amb un o amb dos passos), i finalment canviem el signe del resultat.

EXEMPLE 20: Interpretar en Ca2 la cadena de bits $X=11111100_2$ (té 6 bits alts a 1!)

Procedim a calcular la representació del seu oposat, per la regla del canvi de signe:

$X' = \overline{X} + 00000001_2 = 00000011_2 + 00000001_2 = 00000100_2$

Ara interpretem X' (en dos passsos): calculem el valor explícit de x' fent la suma ponderada: $x'_u = 4$ (eq1). I tot seguit apliquem eq3 per calcular l'enter (com que $X'_0 =0$, $x'_s = x'_u = 4$). El resultat que busquem és simplement l'oposat: $x_s = -x'_s = -4$.

Cas pràctic. Calcular manualment la representació de x_s quan és un negatiu de valor absolut relativament petit.

Si x_s és un negatiu de valor absolut relativament petit, el seu valor explícit serà relativament gran, i requerirà moltes divisions successives. Així que farem menys divisions si calculem la representació X' del seu oposat, que té valor explícit menor. Per tant, comencem calculant el valor explícit de l'oposat x'_u, calculem després X' per divisions successives, i li apliquem un canvi de signe: $X=\overline{X}'+1$.

EXEMPLE 21: Representar en Ca2 amb 8 bits l'enter $x_s = -13$ (és un negatiu petit!)

Si busquem el seu valor explícit (eq5) $x_u = -13 + 256 = 243$, veurem que efectivament és un número força gran i que requereix bastantes divisions per representar-lo. Així doncs, procedim a representar l'oposat $x'_s = -x_s = 13$. Calculem el seu valor explícit (eq5) que és $x'_u = x'_s = 13$, un número força més petit que 243. Amb tan sols 4 divisions obtenim la seva representació: $X' = 1101_2 = 00001101_2$. Finalment, aplicant el canvi de signe obtenim:

$X = \overline{X}'+00000001_2 = 11110010_2 + 00000001_2 = 11110011_2$

g) Declaració de variables de tipus enter

En C:

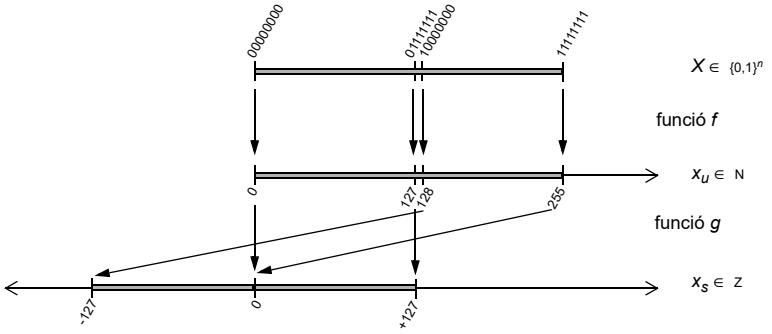
```
char var1 = 0;              // enter de 1 byte
short var2 = 0;             // enter de 2 bytes
int var4 = 0;               // enter de 4 bytes
long long var8 = 0;         // enter de 8 bytes
```

En MIPS, en el cas que les anteriors variables siguin globals:

```
.data
var1: .byte 0
var2: .half 0
var4: .word 0
var8: .dword 0
```

6.2 Enters en complement a 1 (Ca1)

La codificació en Ca1 assigna les representacions que tenen el bit de més pes a 1 (la meitat de totes elles) als enters del $-(2^{n-1}-1)$ al 0. I més concretament, assigna a cada una d'aquestes cadenes de bits un enter que és 1 unitat major que en el cas de Ca2 (recordem que en Ca2 s'assignaven el negatius del -2^{n-1} al -1). Així per exemple, per a n=8 bits, assignaríem els naturals del 128 al 255 (i les seves representacions de la 10000000_2 al 11111111_2), per aquest ordre, als enters del -127 al 0:



Com es pot veure gràficament, la relació entre naturals i enters negatius consisteix en un simple desplaçament (anàleg al Ca2, sols que amb desplaçament 1 unitat més curt). Per als enters negatius de 8 bits es compleix que $x_s = x_u - 255$.

a) Regla d'interpretació: En general, per a n bits, la representació en Ca1 es caracteritza per la seva funció g que ens permet calcular un nombre enter x_s, donat el seu valor explícit x_u. És a dir, $x_s = g(x_u)$ segons la següent regla

$$x_s = \begin{cases} x_u & , \text{ si } \; x_u < 2^{n-1} \\ x_u - (2^n - 1) & , \text{ si } \; x_u \geq 2^{n-1} \end{cases} \qquad \textbf{(eq7)}$$

Que es pot expressar també en un format més compacte així:

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

Anàlogament al cas de Ca2, podem expressar x_s directament en funció dels bits de la seva representació, és a dir, $x_s = h(X)$. Deduirem l'expressió partint del format compacte de eq7 que expressa x_s en funció de x_u:

$$\begin{aligned} x_s &= x_u - X_{n-1} \cdot (2^n - 1) \\ &= (X_{n-1} \cdot 2^{n-1} + \ldots + X_0 \cdot 2^0) - X_{n-1} \cdot (2^n - 1) \\ &= X_{n-1} \cdot (2^{n-1} - 2^n + 1) + \sum_{i=0}^{n-2} X_i \cdot 2^i \\ &= -X_{n-1} \cdot (2^{n-1} - 1) + \sum_{i=0}^{n-2} X_i \cdot 2^i \end{aligned} \qquad \textbf{(eq8)}$$

EXEMPLE 22: Interpretar $X = 10001011_2$ (8 bits).

Solució: calculem primer el valor explícit $x_u = f(X)$ (segons la suma ponderada eq1) i obtenim $x_u = 139$. A continuació calculem $x_s = g(x_u)$ (segons eq7). Al ser $x_u \geq 128$, resulta $x_s = x_u - 255 = 139 - 255 = -116$.

b) Regla inversa, de representació: També podem deduir la regla per a la funció inversa g^{-1}, per a calcular el valor explícit de x_s. És a dir, $x_u=g^{-1}(x_s)$

$$x_u = \begin{cases} x_s & , \text{ si } x_s \geq 0 \\ x_s + (2^n - 1) & , \text{ si } x_s \leq 0 \end{cases} \qquad \textbf{(eq9)}$$

EXEMPLE 23: Representar en Ca1 amb 8 bits l'enter x_s = 29.

Solució: Calculem primer (segons eq9) el valor explícit. Òbviament, x_u=29. Després el representem per divisions successives (eq2): obtenim X=00011101_2.

EXEMPLE 24: Representar en Ca1 amb 8 bits l'enter x_s = -124.

Solució: Procedim de manera anàloga: al ser negatiu, x_u = -124 + 255 = 132, i per divisions successives obtenim X=10000100_2.

c) Propietats: La representació en Ca1 té dos inconvenients: primer, que les dues cadenes de bits $000...0_2$ i $111...1_2$ representen al zero (observem en eq9 els dos valors de x_u per a x_s= 0), la qual cosa complica les operacions de comparació. I segon, que l'operació de suma segueix un algorisme diferent al dels naturals[10] i requereix un circuit diferent.

d) Rang de representació

Al igual que en el cas de Ca2, la codificació en Ca1 representa el signe en el bit de més pes i manté un rang simètric, amb valors absoluts idèntics en els valors extrems del rang, ja que aquest és $x_s \in [-2^{n-1}+1,\ 2^{n-1}-1]$.

e) Regla del canvi de signe

Anàlogament al cas de Ca2, si X és la representació de l'enter x_s, llavors la representació X' del seu oposat $x'= -x_s$ s'obté complementant tots els bits de X. És a dir, sigui la cadena de bits $X_{n-1} \cdots X_0$ que representa x_s i la cadena formada pels seus complementaris $\overline{X} = \overline{X}_{n-1} \cdots \overline{X}_0$ que representa un altre enter que denotarem per $\bar{x}$. Llavors es compleix

$$x' = -x_s = \bar{x} \qquad \textbf{(eq10)}$$

I per tant podem calcular la seva representació així:

$$X' = \overline{X} = \overline{X}_{n-1} \cdots \overline{X}_0$$

Demostració

Escrivim la suma dels enters x_s i $\bar{x}$ aplicant l'expressió eq8

$$\begin{aligned} x_s + \bar{x} &= \left(-X_{n-1} \cdot (2^{n-1} - 1) + \sum_{i=0}^{n-2} X_i \cdot 2^i \right) + \left(-\overline{X}_{n-1} \cdot (2^{n-1} - 1) + \sum_{i=0}^{n-2} \overline{X}_i \cdot 2^i \right) \\ &= -(X_{n-1} + \overline{X}_{n-1}) \cdot (2^{n-1} - 1) + \sum_{i=0}^{n-2} (X_i + \overline{X}_i) \cdot 2^i \\ &= -1 \cdot (2^{n-1} - 1) + \sum_{i=0}^{n-2} 1 \cdot 2^i \\ &= -2^{n-1} + 1 + (2^{n-1} - 1) = 0 \end{aligned}$$

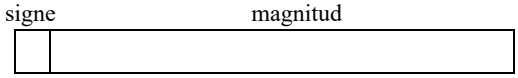
I per tant, obtenim

$$-x_s = \bar{x}$$

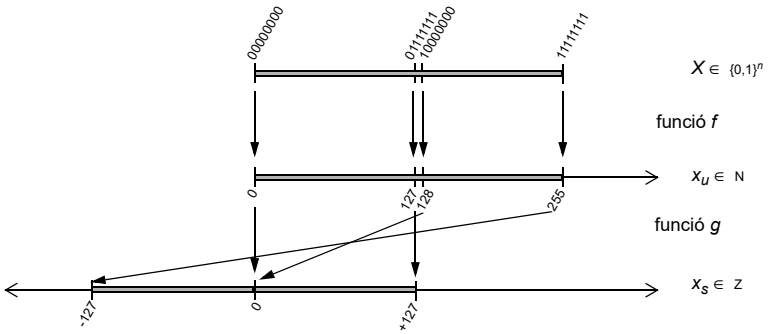
10. Per sumar en Ca1 cal sumar primer els dos vectors de bits com si fossin naturals, i al resultat sumar-li el carry-out de la primera suma (no ho estudiarem en aquest curs).

6.3 Enters "en signe i magnitud"

Consisteix a representar el signe (normalment en el bit de més pes) per separat del valor absolut o magnitud (que es representa en binari, com un natural de n-1 bits).



Les representacions amb el bit de més pes a 1 s'assignen als enters del 0 al -2^{n-1}-1. Així per exemple, per a n=8 bits, assignarem els naturals del 128 al 255 (i les seves representacions del 10000000_2 al 11111111_2), per aquest ordre, als enters del 0 al -127. A diferència de Ca1 o Ca2, on x_s s'obtenia a partir de x_u per un simple desplaçament, en aquest cas les representacions dels negatius s'ordenen al revés que els seus valors explícits.



Aquesta codificació té inconvenients semblants al Ca1: té dues representacions del zero, i l'operació de suma requereix un circuït específic, diferent del de sumar naturals.

6.4 Enters "en excés"

La representació en excés no té com a requisits que el bit de més pes indiqui el signe ni de que els positius es codifiquin igual que els naturals.

a) Propietat

El requisit principal és assegurar que el valor més negatiu tingui la representació $000...0_2$ (x_u=0) i el més positiu la representació $111...1_2$ (x_u=2^n-1), de forma que la seqüència ordenada de valors enters tinguin valors explícits també ordenats, i així es puguin comparar dos enters amb el mateix circuit que compara naturals. La taula a la dreta mostra un exemple per a n=3 bits.

X	x_u	x_s
000	0	-3
001	1	-2
010	2	-1
011	3	0
100	4	1
101	5	2
110	6	3
111	7	4

b) Regles d'interpretació i representació

La funció $x_s = g(x_u)$ es defineix com $x_s = x_u$ - *excés* (i la funció $x_u = g^{-1}(x_s)$ es defineix com $x_u = x_s$ + *excés*), on *excés* és una constant fixa. Per tal d'equilibrar el nombre de positius i negatius es defineix sovint *excés* = 2^{n-1}-1:

$$x_s = x_u - (2^{n-1} - 1) \quad \textbf{(eq11)}$$

$$x_u = x_s + (2^{n-1} - 1) \quad \textbf{(eq12)}$$

Aquesta codificació s'usa per a representar l'exponent dels números en format de coma flotant que estudiarem a fons en el Tema 5.

2.9

7. Representació de caràcters en ASCII

Tradicionalment, una de les utilitzacions dels computadors ha estat el tractament de textos. En general, un text es compon de símbols tipogràfics que poden ser lletres, dígits decimals o símbols de puntuació. Els llenguatges d'alt nivell acostumen a incloure un tipus de dada específic per a representar aquests símbols: el tipus *caràcter*.

La codificació de caràcters consisteix a assignar un codi numèric a cada símbol. Per a aquesta assignació no existeix un criteri únic, de manera que històricament s'han creat múltiples codis (ASCII[11] de 7 bits, ASCII de 8 bits, EBCDIC, Unicode, etc). Emulant a la telegrafia, que ja codificava caràcters molt abans de l'aparició dels computadors, algunes codificacions inclouen al seu repertori no sols símbols tipogràfics sinó també codis *de control* (per al control de la transmissió, del format d'impressió, etc). Molts d'aquests han quedat avui dia obsolets, i alguns tenen usos diferents segons el sistema on s'implanten[12].

En aquest curs estudiarem el codi ASCII de 7 bits. Els caràcters s'emmagatzemen usant 1 byte (8 bits), però el bit de més pes val sempre zero. Els codis del 0 al 31 són caràcters de control i la resta són símbols tipogràfics (imprimibles). La següent taula mostra els codis d'alguns dels caràcters més comuns:

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

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

En C i MIPS, un literal de tipus caràcter s'escriu usant el símbol (entre cometes simples), el seu codi ASCII, o el codi en hexa o octal precedit de barra invertida i entre cometes. Per exemple, la A majúscula es pot escriure: 'A', 65, 0x41, '\x41', o '\101' (octal).

7.1 Propietats de la codificació ASCII

a) Ordre alfabètic. Les lletres minúscules (alfabet anglès) s'assignen en ordre alfabètic a codis consecutius a partir del 97 (0x61). Anàlogament, les majúscules a partir del codi 65 (0x41), i els símbols decimals del '0' al '9' a partir del codi 48 (0x30). És a dir:

- 'a' = 97, 'b' = 98, etc.
- 'A' = 65, 'B' = 66, etc.
- '0' = 48, '1' = 49, etc.

11. American Standard Code for Information Interchange

12. Una línia acaba amb CR o LF? L'estàndard (e.g. en una capçalera HTTP) és finalitzar les línies amb CR+LF (0x0D 0x0A). Això és així ja que en les antigues impressores (sense gràfics) el CR retrocedia el capçal d'impressió al principi de la línia per tatxar o fer dibuixos, i el LF rotava el carro del paper per avançar una línia. En C, el caràcter '\n' s'interpreta en assemblador com el parell de caràcters CR+LF o només LF segons la plataforma (e.g. CR+LF en Windows, LF en Unix, però fins i tot CR en Apple). En l'assignatura traduirem el caràcter '\n' per un simple LF, és a dir pel codi ASCII 0x0A.

b) Conversió majúscula/minúscula. Una lletra passa de majúscula a minúscula (o viceversa) sumant (o restant) 32 al seu codi ASCII. Una altra manera de canviar és invertint el bit 5 del seu codi ASCII (val 0 per a majúscules o 1 per a minúscules). És a dir:

- 'a' = 'A' + 32.
- 'B' = 'b' - 32
- 'c' = 'C' | 00100000_2
- 'D' = 'd' & 11011111_2

c) Conversió dígit ascii/valor numèric. Alguns caràcters són dígits decimals. Cal no confondre'ls amb el *valor numèric* que representen. Per exemple, el caràcter '1' representa al valor 1, però té per codi ASCII el 49 (0x31). La relació és senzilla, si sumem 48 a qualsevol valor entre 0 i 9, convertirem el número en el caràcter ASCII que el representa. També obtenim el mateix resultat sumant '0', ja que '0'=48. És a dir:

- 0 + 48 = '0', 1 + 48 = '1', etc.
- 0 + '0' = '0', 1 + '0' = '1', etc

7.2 Declaració de variables de tipus caràcter

En C, usem el tipus `char`.

```
char lletra = 'R';
```

En MIPS, si la variable `lletra` és global, es declararia així a la secció `.data`:

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

Observa que un enter de 8 bits també es declara com `char`. De fet, als caràcters en C se'ls apliquen els mateixos operadors que als enters (p.ex. 'a' < 'b', 'a'+2 == 'c', etc.).

2.5

8. Format de les instruccions MIPS

Les instruccions, a l'igual que les dades, es representen amb cadenes de bits, s'emmagatzemen a la memòria, i són llegides (i escrites si es tenen els privilegis), igual que si fossin dades. Es codifiquen seguint uns determinats patrons que anomenarem formats. Un format d'instrucció especifica la subdivisió de la instrucció (en aquest cas de 32 bits) en una sèrie de **camps de bits** corresponents al opcode i als operands.

Idealment (per satisfer els principis de disseny 1 i 2) hi hauria un únic format d'instrucció, però hi ha casos que no ho permeten. Per altra banda, tenim un conflicte entre el desig de mantenir totes les instruccions de la mateixa mida i el desig de tenir un únic format d'instrucció. Principi de Disseny 4: *Good design demands good compromises*[13].

MIPS defineix 3 formats d'instrucció, anomenats R (register), I (immediate) i J (jumps). La següent taula mostra els camps de bits de cada format i el seu significat:

		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				

13. El compromís escollit és mantenir totes les instruccions de la mateixa mida i tres formats d'instrucció

El camp *funct* ve a ser una extensió del camp *opcode* (amplia el rang de codis d'operació disponibles). Els camps *rs*, *rt*, i *rd* codifiquen operands en mode registre. El camp *imm16* codifica un operand en mode immediat amb 16 bits. El camp *shamt* (shift amount) codifica també un immediat, usat per les instruccions de desplaçament de bits. El camp *target* codifica els bits del 2 al 27 de l'adreça destinació d'un salt (els bits 0 i 1 es posen a zero, i els bits 28 a 31 es copien dels corresponents bits del PC).

Vegem la codificació de 6 instruccions d'exemple que usen aquests 3 formats:

			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				

EXEMPLE 25. Codificar en binari la instrucció: `addu $t0, $s2, $zero`

- opcode = 0x00 = 000000_2
- rs = $s2 = $18 = 10010_2 ($s0-$s7 es numeren $16-$23, veure apartat 4.1)
- rt = $zero = $0 = 00000_2
- rd = $t0 = $8 = 01000_2 ($t0-$t7 es numeren $8-$15, veure apartat 4.1)
- funct = 0x21 = 100001_2

Resposta: `0000 00|10 010|0 0000| 0100 0|000 00|10 0001 = 0x02404021`

Nou

9. Vectors

Els vectors són agrupacions unidimensionals d'elements de tipus homogenis, els quals s'identifiquen per un índex. Els hem estat usant al llarg d'aquest curs, però ara els estudiarem amb més detall.

9.1 Emmagatzematge i declaració

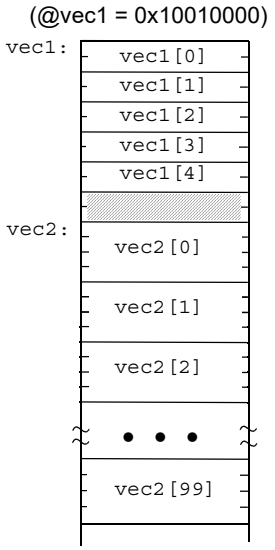
En llenguatge C, els elements d'un vector es guarden en posicions consecutives a partir de l'adreça inicial del vector. Els seus elements s'especifiquen per un índex enter. En C, al primer element li correspon l'índex zero, i els altres es numeren correlativament.

En MIPS, a més a més, els elements han de respectar les regles d'alineació segons els seus tipus respectius. Com que tots els tipus estudiats fins ara tenen mides que són potència de 2, lògicament si el primer element del vector està alineat, també ho estaran tots els altres sense necessitat d'insertar cap espai intermedi entre ells[14].

14. En C no tots els tipus de dades tenen una mida potència de 2. Les tuples (representades en C pel tipus `struct`) són agrupacions d'elements ("camps") de tipus heterogenis, on la mida de la tupla depèn dels tipus dels camps que la formen. Cada camp de la tupla ha d'estar alineat en memòria segons el seu tipus, i per tant poden aparèixer no solament espais de memòria no usats entre els camps, sinó també entre elements consecutius d'un vector de tuples. No estudiarem les tuples en aquest curs.

EXEMPLE 26. En C, declarem les variables globals *vec1* i *vec2*:

```
short vec1[5] = {0, -1, 2, -3, 4};
int vec2[100];
int vec3[5]   = {0, -1};
int vec4[]    = {1, 0, -2};  /* té 3 elements */
```



En C, un vector es pot declarar sense inicialitzar (*vec2*), o bé es poden inicialitzar tots (*vec1*) o una part dels seus elements (*vec3*). Els valors dels elements no inicialitzats són zeros en cas de variables globals o bé indeterminats en cas de variables locals. Normalment cal especificar la dimensió amb una constant literal (mai una veriable), però es pot ometre si el vector està inicialitzat (*vec4*), llavors la dimensió la defineix el nombre de valors inicials.

En MIPS serà:

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

A fi de poder inicialitzar els elements de *vec1*, el declarem amb la directiva `.half`, que a més a més ja fa l'alineació de manera automàtica. En canvi, per declarar *vec2*, com que no està inicialitzat, usem la directiva `.space`, que és més compacta. En contrapartida, aquesta directiva no força cap tipus d'alineació, de manera que l'hem de precedir de la directiva `.align`.

En el diagrama de la dreta hem suposat que *vec1* s'emmagatzema a l'adreça 0x10010000. En aquest cas, la directiva `.align` deixa dos bytes sense usar entremig dels dos vectors a fi d'assegurar que `vec2[0]` ocupi una adreça múltiple de 4 (0x1001000C).

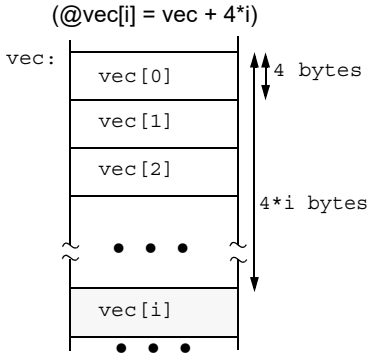
9.2 Accés (aleatori) a un element

Per accedir a l'element i-èssim d'un vector cal calcular prèviament la seva adreça. De les descripcions de l'apartat anterior es desprèn fàcilment que, si la mida d'un element en bytes és *T*, llavors aquesta adreça és:

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

EXEMPLE 27. En C, sigui la sentència:

```
vec[i] = 10;
```



En MIPS serà, suposant que *i* està en $t0 i que *vec* és un vector global d'enters de 32 bits:

```
la      $t2, vec            # vec
sll     $t1, $t0, 2         # 4*i
addu    $t2, $t2, $t1       # vec+4*i
li      $t1, 10
sw      $t1, 0($t2)
```

2.9

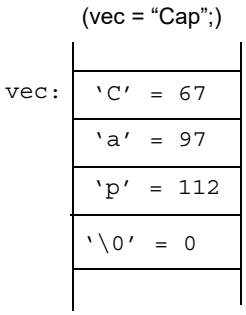
10. Strings

Els *strings* o *cadenes de caràcters* són vectors amb un nombre variable de caràcters.

10.1 Emmagatzematge

Segons en quin llenguatge d'alt nivell escrivim, es poden emmagatzemar:

- En Java: com una tupla, composta d'un enter que n'indica la longitud i un vector de caràcters.
- En C: com un vector de caràcters, però afegint a continuació de l'últim caràcter de la cadena una marca de final o sentinella. Aquest sentinella és el valor 0 (caràcter '\0'). Així doncs, el string "Cap" es representa en C per un vector de 4 bytes, amb valors decimals: 67, 97, 112, 0.



10.2 Declaració

La declaració es fa com un vector de caràcters qualsevol. La particularitat és la manera com l'inicialitzem (si és que ho fem, ja que és opcional):

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

El compilador reservarà 10 bytes (9 caràcters i el sentinella). Equival a

```
char cadena[10] = "Una frase";
```

També equival a la següent declaració, encara que és molt menys compacta

```
char cadena[10] = {'U', 'n', 'a', ' ', 'f', 'r', 'a', 's', 'e', '\0'};
```

La traducció a MIPS, suposant que és una variable global, usant la directiva `.ascii`

```
            .data
cadena:     .ascii  "Una frase"         # Té 9 caràcters
            .byte 0                     # El sentinella
```

Alternativament, usant la directiva `.asciiz`

```
            .data
cadena:     .asciiz "Una frase"         # Té 9 caràcters i el sentinella
```

10.3 Accés als elements

En C, els caràcters es codifiquen en ASCII (1 byte), i s'hi accedeix usant les instruccions `lb` o `sb`. En Java, es codifiquen en Unicode (2 bytes) i s'hi accedeix amb `lhu` o `sh`.

EXEMPLE 28. Fragment en C que compta els caràcters de la cadena *nom* (global):

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

En MIPS, suposant que la variable *num* està en $t0:

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

Nou

11. Punters

Un punter és una variable que conté una adreça de memòria. Per tant, en MIPS ocupa 32 bits. Si el punter *p* conté l'adreça de la variable *v* diem també que *p* "apunta" a *v.*

11.1 Declaració

Aritmèticament, un punter és un simple número natural, però el llenguatge C el considera com un tipus de dada específic, diferent dels naturals, i això es reflecteix en la manera de declarar-lo. A més a més, el llenguatge C distingeix diverses menes de punters segons el tipus de la dada a la qual han d'apuntar, i aquest tipus ha de figurar en la seva declaració. Per exemple:

```
int *p1, *p2;
char *p3;
```

En aquest exemple, *p1* i *p2* són punters del tipus *int* *, és a dir que són punters a variables de tipus *int*, mentre que *p3* és un punter del tipus *char* *, és a dir que és un punter a variables de tipus *char*. Observa que el signe * en les declaracions no denota cap operació, és un declarador. En MIPS, suposant que *p1*, *p2* i *p3* són variables globals, escriurem:

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

Per copiar un punter en un altre cal que els dos apuntin al mateix tipus de variable, altrament el compilador senyala un avís. Per exemple:

```
     p1 = p2;                       /* Correcte */
     p1 = p3;                       /* Assignació errònia! */
```

En el segon cas, el compilador mostraria el següent missatge:

```
myfile.c:12: warning: assignment from incompatible pointer type
```

11.2 Inicialització

En C, un punter es pot inicialitzar assignant-li un altre punter (com en els exemples anteriors), o bé assignant-li l'adreça d'alguna variable. Per denotar "adreça de la variable x", escrivim `&x`, usant l'operador `&` precedint al nom de la variable. Aquest operador no

s'ha de confondre amb el de la "y lògica bit a bit" (que té el mateix símbol), ja que el primer és un operador unari que precedeix a l'operand, mentre que el segon és un operador binari que s'interposa entre dos operands. Vegem un exemple:

EXEMPLE 29. Sigui el següent codi en C amb diverses inicialitzacions de punters, unes en la pròpia declaració i altres en les sentències del programa (en negreta):

```
int v[100];
char a='E', b='K';
char *pglob = &a;
void f()
{
       int *ploc;          /* s'emmagatzemarà en $t0 */
       ...
       ploc = &v[8];
       ...
       pglob = &b;
       ...
}
```

a) Inicialització dins la declaració. Les declaracions globals, en MIPS serien:

```
       .data
v:     .space 400
a:     .byte 'E'
b:     .byte 'K'
pglob: .word a             # traduïm: char *pglob = &a
```

Observem que la declaració de `pglob` usa l'etiqueta `a` com a valor inicial, això és consistent amb el fet que les etiquetes en assemblador equivalen a adreces.

b) Inicialització en una sentència. El codi de la funció *f*, en MIPS seria:

```
f:     ...
       la     $t0, v+32     # calculem @v[8] = @v[0] + 8*4
       ...
       la     $t1, b        # calculem @b
       la     $t2, pglob    # obtenim l'adreça de pglob
       sw     $t1, 0($t2)   # hi guardem el valor @b
       ...
```

Observem que *ploc* resideix al registre $t0 i s'inicialitza simplement calculant l'adreça `@v[8]` amb la macro `la`. En canvi, *pglob* resideix en memòria, i per assignar-li el valor `@b` no n'hi ha prou de calcular-lo amb la macro `la`, sinó que després cal obtenir l'adreça de *pglob* i guardar-hi `@b` amb una instrucció `sw`.

11.3 Operació desreferència

Els punters serveixen per accedir (llegir o escriure) a les variables apuntades. L'operació de *desreferència d'un punter* consisteix a accedir a la posició de memòria apuntada pel punter. S'indica en C precedint el nom del punter amb l'operador *. Així doncs, l'expressió `*p` significa "contingut de la posició de memòria apuntada pel punter `p`". No s'ha de confondre l'operador de desreferència amb el declarador de punters explicat a l'apartat 11.1, que usa el mateix símbol. Mentre que el declarador apareix sempre en una declaració, entre el nom del tipus apuntat i el nom del punter, l'operador de desreferència apareix sempre precedint un punter, en el codi.

EXEMPLE 30. El següent codi en C conté dues desreferències:

```
char *pglob;                /* variable global */
void g()
{
       char tmp, *ploc;     /* es guarden en $t0, $t1 */
       ...
       tmp = *ploc;
       ...
       tmp = *pglob;
       ...
}
```

En MIPS, serà:

```
g:   ...
     lb     $t0, 0($t1)          # desreferenciem el punter ploc
     ...
     la     $t2, pglob
     lw     $t3, 0($t2)          # carreguem el punter en un registre
     lb     $t0, 0($t3)          # desreferenciem el punter pglob
     ...
```

La primera sentència carrega a la variable local *tmp* el contingut de la posició de memòria apuntada per *ploc*. Com que *ploc* està en un registre, n'hi ha prou d'executar una instrucció `lb` on el registre-base sigui el que conté el punter. La segona sentència és similar, però al ser *pglob* global, està emmagatzemada en memòria. Cal prèviament carregar la variable global *pglob* en el registre $t3 (copiem l'etiqueta en $t2 amb la macro `la`, i a continuació accedim al punter en memòria amb `lw`). Un cop tenim el punter *pglob* en registre, fem la desreferència de la mateixa manera que pel cas de *ploc*, amb `lb`.

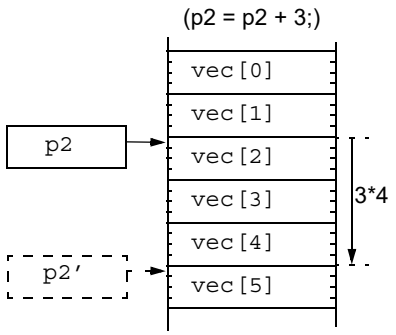
11.4 Operació "aritmètica de punters"

En C es defineix l'operació *aritmètica de punters* com la suma d'un punter *p* més un enter *N*, que dóna com a resultat un altre punter *q* del mateix tipus: $q = p + N$

Si sumem una unitat a *p*, el punter resultant apuntarà a l'element immediatament adjacent en memòria (per exemple, si *p* apuntava a l'element *vec*[2], *p*+1 apuntarà a vec[3]). En aquest cas, suposant que *p* està declarat com a punter a un tipus de dada de mida *T*, la quantitat a sumar-li realment serà *T* bytes. En general, si li sumem un enter *N*, el punter resultant apuntarà a un altre enter situat "*N* elements més endavant". És a dir, que el resultat de sumar un enter *N* a un punter *p* que apunta a elements de mida *T* requereix en realitat sumar-li *N* multiplicat per *T*. L'aritmètica de punters sol usar-se per fer que un punter apunti successivament a diversos elements d'un vector.

EXEMPLE 31. En C, suposem les següents declaracions i sentències:

```
char *p1;          /* char = 1 byte */
int *p2;           /* int = 4 bytes */
long long *p3;     /* long long = 8 */
     ...
p1 = p1 + 3;
p2 = p2 + 3;
p3 = p3 + 3;
```



En MIPS la traducció de les 3 sentències serà, suposant que *p1*, *p2*, *p3* es guarden en $t1, $t2, $t3:

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

11.5 Relació entre punters i vectors

En el llenguatge C, les variables declarades com a vectors tenen el mateix tipus que un punter, és a dir que són pròpiament punters, amb la particularitat que apunten al primer element del vector. Així per exemple, les següents variables *vec* i *p* tenen el mateix tipus:

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

L'única diferència és que el punter *vec* és constant (sempre apuntarà a la mateixa posició de memòria), mentre que *p* pot apuntar a qualsevol posició al llarg del programa.

D'aquesta dualitat es deriven dues conseqüències:

a) Podem inicialitzar un punter assignant-li un altre punter o bé un vector (no obstant, al contrari no funciona perquè no podem assignar una nova adreça a un vector declarat com a tal, ja que apunta a una posició de memòria fixa).

```
p = vec;                    /* fem que p apunti a l'element vec[0] */
```

La traducció en MIPS serà, suposant que *p* resideix en $t0, i *vec* és global:

```
la     $t0, vec
```

b) Als punters els podem aplicar l'operador "[ ]", igual que als vectors:

```
p[8] = 0;                   /* accés a l'element 8 del vector */
```

En MIPS serà, suposant que *p* resideix en $t0:

```
sw     $zero, 32($t0)
```

c) Als vectors els podem aplicar l'operador "*", igual que als punters:

*vec = 0;

En MIPS serà, suposant que *vec* és global:

```
la     $t0, vec
sw     $zero, 0($t0)
```

Per entendre aquesta dualitat, observem que si *p* és un punter, llavors el podem considerar també com un vector, el primer element del qual és l'element apuntat per *p*. I per tant l'expressió *p+i* (aritmètica de punters) és un altre punter que apunta a l'element *i*-èssim del vector *p*. Així doncs, resulta coherent que l'expressió

```
*(p + i) = 0;
```

és totalment equivalent a

```
p[i] = 0;
```

Anàlogament, podríem convertir totes les expressions que contenen l'operador "[ ]" a les expressions equivalents usant l'operador desreferència (i de fet el compilador ho fa així internament).

12. Exemples de revisió

EXEMPLE 32. Traduir a assemblador MIPS el següent programa en C

```
#define N 4
int V[N] = {0,1,2,3};
void main () {
     int suma=0, i=N;
     while ( i != 0) {
            i--;
            suma = suma + V[i];
     }
}
```

La traducció en MIPS serà, suposant que *suma* i *i* ocupen els registres $t0 i $t1:

```
     .eqv  N, 4                 # define N 4
     .data
V:   .word 0, 1, 2, 3           # vector d'enters
     .text
main:
     move   $t0, $zero          # suma = 0
     li     $t1, N              # i = N
     la     $t4, V              # @V[0] = 0x10010004. S'expandeix:
                                #      lui $at,0x10010000
                                #      ori $t4,$at,0x0004
while:
     bne    $t1, $zero,fiwhile  # salta si i!=0
     addiu  $t1, $t1, -1        # i--
     sll    $t3, $t1, 2         # i*4
     addu   $t3, $t3, $t4       # @V[0] + i*4
     lw     $t5, 0($t3)         # V[i]
     addu   $t0, $t0, $t5       # suma = suma + V[i]
     b      while               # salta
fiwhile:
```

Observem que hem aplicat una optimització: l'adreça de V no varia al llarg de totes les iteracions, és el que anomenem un *invariant*. La instrucció que copia aquesta adreça en $t4 es pot situar abans d'entrar al bucle i executar-se una sola vegada. Aquesta optimització rep el nom d'*extracció d'invariants*.

EXEMPLE 33. Traduir el següent codi, equivalent a l'anterior. Aquest programa fa servir un punter per recórrer el vector, i accedeix als seus elements desreferenciant el punter.

```
#define N 4
int V[N] = {0,1,2,3};
void main () {
     int suma=0, i=N, *p;
     p = V;                              /* equival a: p=&V[0]  */
     while ( i != 0) {
            suma = suma + (*p);          /* desreferència de p */
            p++;                         /* fem que p apunti al següent */
            i--;
     }
}
```

La traducció en MIPS serà, suposant que *suma*, *i* i *p* ocupen $t0, $t1 i $t3:

```
     .eqv  N, 4                 # define N 4
     .data
V:   .word 0, 1, 2, 3           # vector d'enters
     .text
main:
     move   $t0, $zero          # suma = 0
     li     $t1, N              # i = N
     la     $t3, V              # p = V
while:
     beq    $t1, $zero, fiwhile # salta si i==0
     lw     $t4, 0($t3)         # *p
     addu   $t0, $t0, $t4       # suma = suma + (*p)
     addiu  $t3, $t3, 4         # p++
     addiu  $t1, $t1, -1        # i--
     b      while               # salta
fiwhile:
```

Obsevem que el cos del bucle de l'exemple anterior tenia 7 instruccions, mentre que aquest en té solament 6. Aquesta optimització rep el nom d'*accés sequencial*, i l'estudiarem amb més detall en el Tema 4. En el Tema 3 veurem també altres maneres d'optimitzar bucles *while*: avaluació de la condició al final del bucle i eliminació de variables d'inducció.