Pràctica 6
Variables

6.1 Objectius

- Entendre la distribució de la memòria en el MIPS.
- Conèixer i utilitzar les instruccions que ens permeten llegir i escriure en la memòria.
- Conèixer com definir dades en la memòria.
- Utilitzar funcions del sistema que permeten l'entrada i eixida de cadenes de caràcters.

6.2 Material

Simulador MARS i un codi font de partida

6.3 Teoria

6.3.1 Introducció

Ha arribat l'hora d'estudiar un element que hem obviat ens les pràctiques precedents, la memòria. Hem suposat fins ara que el programa a executar està emmagatzemat en la memòria i en cada cicle d'instrucció es llig una instrucció i s'executa en el processador. Ara ens preguntem si les dades utilitzades per les instruccions necessàriament han de provenir dels registres del processador o del teclat. Hi ha la possibilitat de que la memòria allotge dades? La resposta és clara, si hi ha dades en la memòria, a més, l'assemblador ens permet tant llegir com escriure valors en la memòria. Això sí, cal tenir present que el MIPS sols permet operar amb l'ALU amb valors immediats o amb dades allotjades en registres (com hem vist fins ara). Això significa que abans d'operar amb una dada de la memòria hem de portar-la a un registre del processador. El MIPS ens proporciona les instruccions que ens possibiliten fer-ho. Però anem per parts. Primer descriurem la memòria del MIPS.

6.3.2 La memòria

El model de la memòria que implementa el MIPS és el d'una memòria plana de 32 bits adreçable per bytes amb adreces de 32 bits. El que vol dir això és que per al programador la memòria del processador MIPS és una memòria contínua de bytes agrupats en paraules de 32 bits que comença en l'adreça 0x00000000 i acaba en l'adreça 0xFFFFFFFF. El programador podria adreçar 4GBytes de dades. Ara bé, això no significa que tota aquesta memòria està a disposició del programador: una part de la memòria està reservada per a utilitzar-la el sistema operatiu (el que s'anomena *kernel data*), una part la utilitza el subsistema d'entrada/eixida, etc. La figura 1 mostra un diagrama de com estan configurats els 4GB de la memòria del MIPS.

La part de la memòria accessible per a l'usuari comprèn el ***Text segment*** (Adreces 0x0040 0000 fins a 0x1000 00000) que és on s'emmagatzema el codi del programa. Com ja hem comentat, cada instrucció s'emmagatzema en una paraula de 32 bits o 4 bytes. Tot programa escrit fins ara comença amb `.text`. Es tracta d'una directiva que defineix el començament del segment de codi. Si no indiquem una adreça, l'assemblador assumeix per defecte el valor 0x00400000.

Les dades estàtiques (adreces 0x1001 0000 fins a 0x1004 0000) és on es guarden les dades del programa. La grandària dels elements d'aquesta secció s'assignen quan el programa s'assembla i es carrega en la memòria. La definició dels elements no es pot modificar durant l'execució del programa. Quan s'escriu un programa podem definir les dades estàtiques mitjançant l'ús de directives que comentarem en el pròxim apartat. De manera similar al *Text segment*, la definició de dades començarà amb la directiva `.data`. Si no indiquem una adreça, l'assemblador assumeix per defecte el valor 0x10010000.

Dades dinàmiques (adreces 0x1004 0000 fins que es troba amb la pila) sempre creix cap a les adreces altes. És on s'emmagatzemen les dades dinàmiques que es van creant per necessitat de l'execució dels programes (per exemple, amb un nou operador en Java).

El ***Stack segment*** (o segment de pila) (adreces 0x7ffffe00 fins que ensopegue amb les dades dinàmiques) sempre creix cap a adreces baixes. Les dades van emmagatzemant-se en aquest segment a mesura que es creen mitjançant les operacions de push i pop per a les subrutines. Ho estudiarem en pràctiques posteriors.

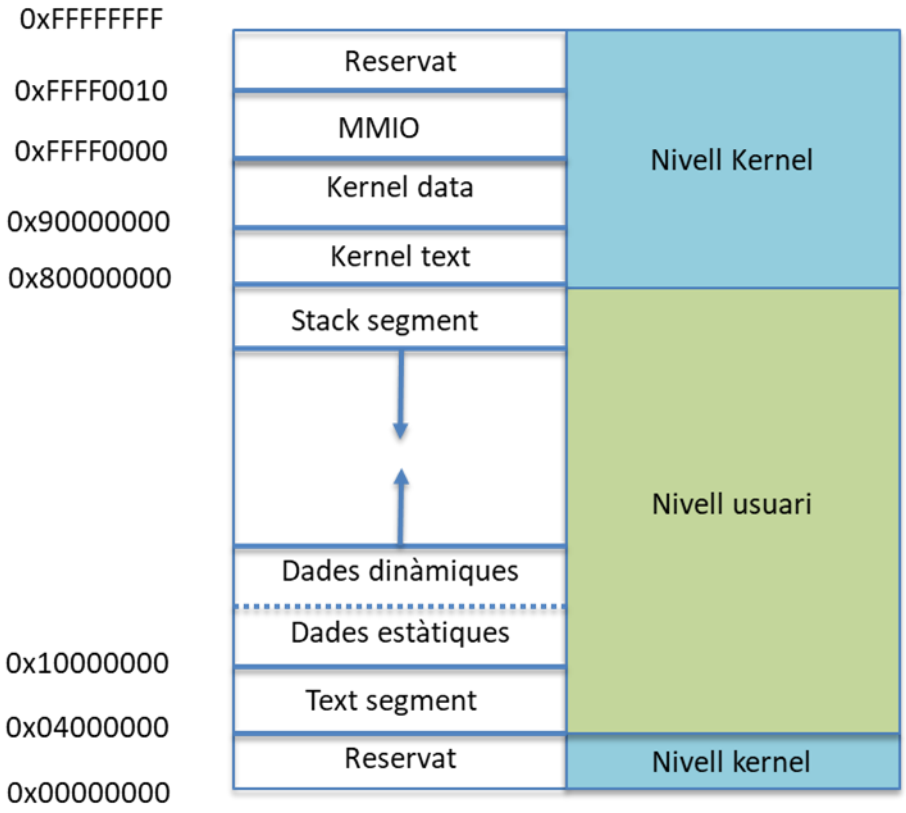

Figura 1. Model de la memòria del MIPS.

6.4 Desenvolupament pràctic

6.4.1 El Text segment

Amb qualsevol dels programes escrits fins ara podem observar com conté el *text segment* el programa en execució, recordeu que tots els programes comencen amb *.text*. Per a observar-ho simplement heu de seleccionar, una vegada assemblat el programa, l'opció de veure el *text segment* en pantalla. Es mostrarà en cada fila 8 posicions de memòria; podeu comprovar que el seu contingut és el mateix que indica la columna *code* de la finestra superior.

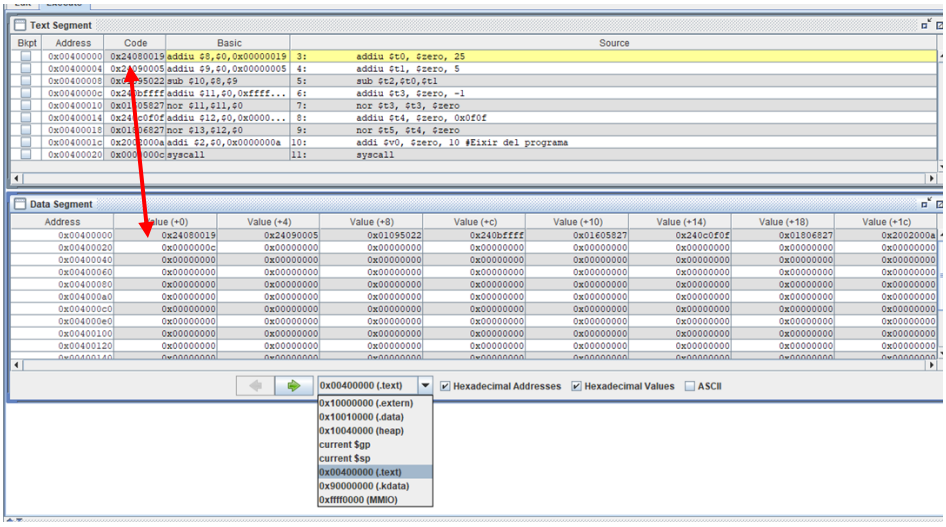

Figura 2. Visualització del text segment de la memòria del MIPS en el MARS

Com veieu, per a facilitar la visualització de la memòria i mostrar més paraules alhora en la pantalla, el MARS mostra 8 paraules per fila en lloc de mostrar cada paraula en una fila, tal com s'indica en la figura 3:

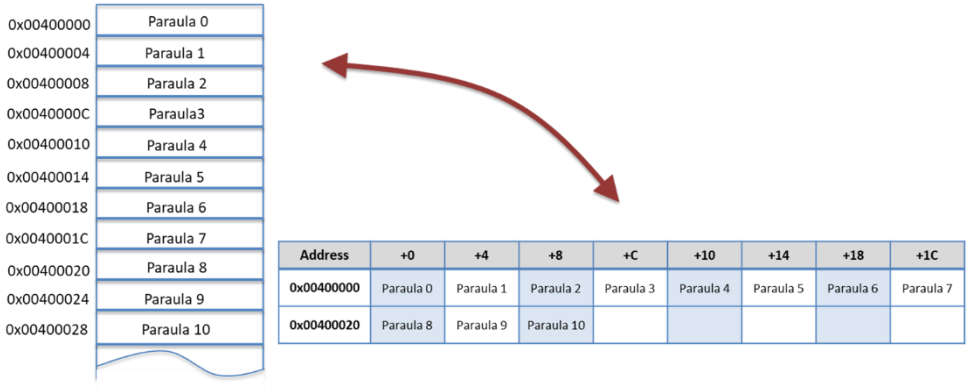

Figura 3. El MARS mostra 8 paraules de la memòria en la mateixa línia de la pantalla.

El processador accedeix al *text segment* de la memòria per a llegir una instrucció amb el registre $pc que indica l'adreça de la paraula on es troba la instrucció a executar.

6.4.2 Adreçament de la memòria

Com hem comentat, el MIPS utilitza un model pla per a la memòria, així, el programador veu la memòria com un successió de bytes als quals pot

accedir mitjançant una adreça. Com que pot accedir a cada byte individual, es diu que la memòria és adreçable per byte. Començant des de l'adreça 0x00000000, aquests bytes s'organitzen en grups de mides diferents: com a bytes individuals, com a parelles de bytes (s'anomenen *mitges paraules* o *half word*), com a grups de 4 bytes (anomenades *paraules* o *word*) i com a grups de 8 bytes (anomenades *dobles paraules*, *double word*). A cadascun d'aquests grups es pot accedir mitjançant l'adreça del byte menys significatiu (anomenat *LSB*) del grup, és a dir, el byte amb l'adreça més baixa. Aquests grups de bytes han d'estar alineats. Això vol dit que per a seleccionar un byte n'hi prou que l'adreça siga múltiple d'1, però per accedir a una mitja paraula ha de ser múltiple de 2, a una paraula ha de ser múltiple de 4 i a una doble paraula ha de ser múltiple de 8, etc. Per exemple, amb l'adreça 0x10001000 podem seleccionar un byte, una mitja paraula o una paraula sencera, però amb l'adreça 0x10001001 sols podem seleccionar un byte. La forma de distingir a quin grup s'està referenciant depèn de l'operador utilitzat per accedir-hi. En la figura 4 es mostra la idea:

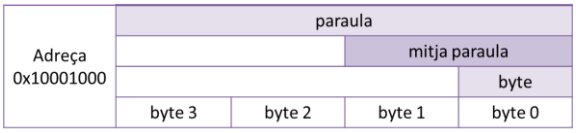

Figura 4. Organització dels bytes en la memòria del MIPS.

6.4.3 Data segment

Les dades en MIPS s'emmagatzemen en el segment de dades (*data segment*). Les dades es defineixen quan el programa s'assembla i es col·loquen en el segment quan el programa es carrega en la memòria i es comença a executar. Per això, aquestes dades són estàtiques, es a dir, la seua grandària no es pot modificar durant l'execució del programa.
Per a definir les dades s'utilitza la directiva `.data <adreça>`. L'adreça és opcional, si no l'indiquem suposa que comença en 0x10001000. A continuació podem definir dades de la grandària que ens interessa.
Per exemple, el codi:

```
########################################
#                                      #
#    Exemple de definició de de dades  #
#                                      #
########################################

.data       #Comença la definició de dades

A: .word 0  #A estarà en l'adreça 0x10001000
B:  half 0  #B estarà en l'adreça 0x10001004
C:  byte 0  #D estarà en l'adreça 0x10001006
```

Defineix a partir de l'adreça 0x10001000 les dades següents: A de grandària paraula (*.word*), B de grandària mitja paraula (*.half*) i C de grandària byte (*.byte*). Fixeu-vos que hem utilitzat etiquetes per a definir les dades (les etiquetes fan referència a l'adreça on està la dada i una vegada definides les podrem utilitzar en qualsevol lloc del programa) i les directives `.word`, `.half` i `.byte` per a delimitar la grandària. El valor que hi ha després, en tot els casos 0, és el valor inicial que li donem.

6.4.4 Ús de la memòria

Utilitzem un codi de partida exemple per clarificar l'ús de la memòria del *data segment* pel MIPS.

```
##########################################
#                                        #
#        Codi de l'activitat 1           #
#        Exemple: Ús de la memòria       #
#                                        #
##########################################

.data                #Comença definició dades
paraula1:.word 15       #Paraula en decimal
paraula2: .word 0x15    #Parula en hexadecimal

mitjaparaula1: .half 2
mitjaparaula2: .half 6
Dosbytes: .byte 3,4

.align 2          #Alinea a paraula (2^2 bytes)

byte1: .byte 8
byte2: .byte 5

espai: .space 4         #Reserva 4 bytes a 0

cadena1: .asciiz "Estructura dels computadors"
```

6.4.4.1 Anàlisi del codi

Aquest codi sols conté directives, és a dir, instruccions a l'assemblador que li indiquen com emmagatzemar dades en el segment de dades de la memòria i, per tant, no es traduiran en codi de màquina. Quan assembleu aquest codi podeu comprovar que el segment de text és buit perquè no hi ha instruccions, però en el segment de dades trobem els valor que hem definit en el programa. Amb les directives es poden definir variables de diferents tipus i grandària i iniciar-les amb valors determinats. Es poden definir paraules (`.word`) de 32 bits de grandària, mitges paraules (`.half`) de 16 bits de grandària, bytes (`.byte`) de 8 bits de grandària i cadenes ASCII (`.asciiz`), en què cada un dels caràcters té 8 bits de grandària. La directiva `.space` permet reservar memòria del segment de dades. És útil

per a definir variables sense inicialitzar i variables de grandària no coneguda. Amb la directiva `.align 2` indiquem a l'assemblador que la variable següent que definim estarà alineada a 2^2 bytes (recorda el que s'ha comentat abans sobre l'adreçament de la memòria)
El segment de dades tindria un aspecte semblant al de la figura 5:

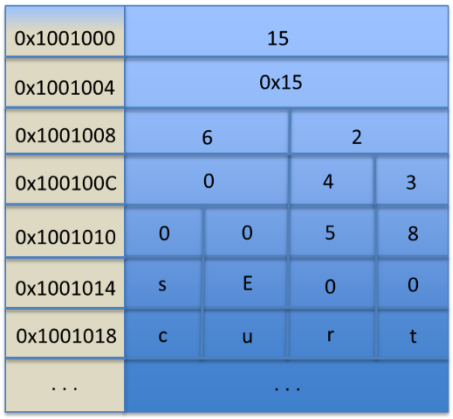

Figura 5. Aspecte de la memòria al fer ús de directives.

6.4.4.2 Activitat 1

- Escriu i assembla el codi *d'exemple de l'ús de la memòria* i comprova com queda el segment de dades.

- Canvia el tipus de visualització de les dades (hexadecimal, decimal, ASCII) marcant les caselles adients.

Qüestió 1

- Quina és l'adreça del byte on està emmagatzemat el caràcter 'l'?

En la taula 1 es resumeixen algunes de les principals directives del MIPS que utilitzarem en les properes pràctiques.

Directiva	Descripció
.align n	Alinea la dada següent sobre un límit de 2n bytes
.ascii "cadena"	Emmagatzema la cadena de caràcters en la memòria, no acaba amb caràcter nul.
.asciiz "cadena"	Emmagatzema la cadena de caràcters en la memòria, acaba amb caràcter nul.
.byte b1,..bn	Emmagatzema n quantitats de 8 bits en posicions consecutives de la memòria.
.data <adreça>	Els elements següent són emmagatzemats en el segment de dades. Si està present l'argument <adreça>, les dades s'emmagatzemen a partir d'aquesta posició.

.double d1,..dn	Emmagatzema n nombres de doble precisió i coma flotant (64 bits) en posicions consecutives de la memòria
.half h1,..hn	Emmagatzema n quantitats de 16 bits en posicions consecutives de memòria.
.space n	Reserva n bytes d'espai en el segment de dades.
.text <adreça>	Defineix el començament del segment de codi. Si s'especifica l'adreça comença a partir d'eixa posició.
.word w1,..wn	Emmagatzema n quantitats de 32 bits en posicions consecutives de la memòria.

Taula 1. Directives del MIPS.

6.4.5 Accés a la memòria

El MIPS sols permet accedir a la memòria mitjançant dues operacions, la lectura i l'escriptura en la memòria. Els operadors que proporciona el MIPS per a fer questes operacions ens permeten accedir a bytes (`lb` i `sb`), mitges paraules (`lh` i `sh`) o paraules (`lw`, `sw`). Vegem ara com fa el MIPS les operacions de lectura i escriptura en memòria:

6.4.5.1 Lectura i escriptura en memòria

Les instruccions de llegir en memòria *lw (load word*), *lh* (*load half*) i *lb* (*load byte*) segueixen la forma general, `op rt,k(rs)`, en què *op* és l'operació (*lw*, *lh*, *lb*) i k és un desplaçament de 16 bits amb signe que se sumarà al contingut del registre *rs* per a formar l'adreça on es troba la dada. Una vegada llegida, s'emmagatzemarà en el registre *rt*. En la figura 6 es pot observar el procés:

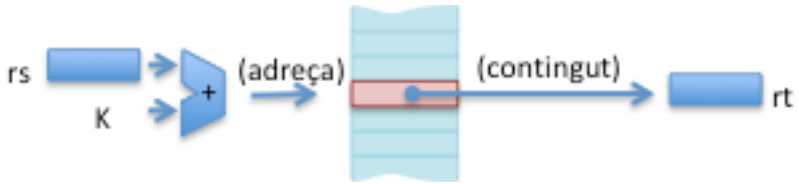

Figura 6. Operació de lectura en memòria (lw, lh, lb).

De manera similar, les instruccions d'escriptura en memòria *sw (store word*), *sh* (*store half*) i *sb* (*store byte*) segueixen la forma general, `op rt,k(rs)`, en què *op* és l'operació (*sw*, *sh*, *sb*) i k és un desplaçament de 16 bits amb signe que es sumarà al registre *rs* per a formar l'adreça on s'emmagatzemarà la dada que hi ha en el registre rt. En la figura 7 es pot observar el procés:

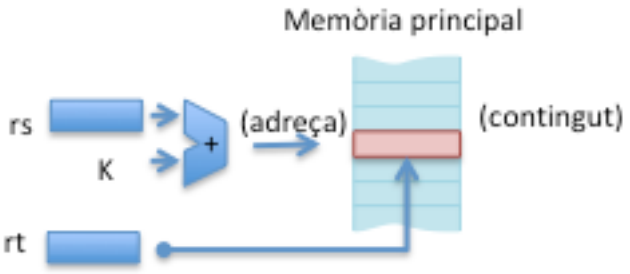

Figura 7. Operació d'escriptura en memòria (sw, sh, sb).

Tant les instruccions de lectura com d'escriptura en memòria segueixen el format d'instrucció I, en el qual, a més del camp *codi d'operació*, hi ha dos camps per a especificar els registres *Rs* i *Rt*, i un camp de 16 bits que indicarà el desplaçament expressat en complement a 2.

A continuació aprofundirem en les instruccions d'accés a la memòria mitjançant un codi exemple. Considereu el programa següent en assemblador:

```
##########################################
#                                        #
#     Codi exemple de l'activitat 2      #
#                                        #
#Accés a la memòria. Lectura i escriptura#
#                                        #
##########################################


.data             #Definició segament de dades
A: .word 25
B: .word 10
C: .word 0

.text             #Comença el programa
la $t0,A
la $t1,B
la $t2,C
lw $s0,0($t0)     #Lectura de la dada A
lw $s1,0($t1)     #Lectura de la dada B
add $s2,$s1,$s0
add $s2,$s2,$s2
sw $s2,0($t2)     #Escriptura de la dada C

li $v0, 10        #Acaba el programa
syscall
```

Com que les tres dades del programa estan representades per les etiquetes A, B i C, podem utilitzar la pseudoinstrucció *la* per a obtenir la seua adreça en memòria i fer-ne ús en qualsevol lloc del programa.

6.4.5.2 Activitat 2

- Analitzeu el programa exemple *Accés a la memòria. Lectura i escriptura*. Què fa el programa?
- Assembleu i executeu el programa. Comproveu que podeu seleccionar per a veure tant el segment de text com el segment de dades.

- Comproveu que podeu observar les dades en hexadecimal i en decimal.

Qüestió 2

- Quants bytes de la memòria principal estan ocupats per dades del programa?

- Quantes instruccions d'accés a la memòria conté el programa?

Qüestió 3

- Quin valor té el registre $t1 quan s'executa la instrucció `lw $s1,0($t1)`?

- En quina adreça s'emmagatzema el resultat?

Qüestió 4

- Substituïu la instrucció `sw $s2,0($t2)` per `sw $s2,2($t2)`? Què passa quan s'intenta executar el programa? Raoneu la resposta.

Qüestió 5

- Quina és la codificació en llenguatge màquina de la instrucció `lw $s1,0($t1)`? Desglosseu la instrucció en els diferents camps del format.

6.4.5.3 Instruccions d'accés a la memòria

En la taula 2 es resumeixen les instruccions de lectura i escriptura en memòria i algunes pseudoinstruccions proporcionades pel MIPS.

Les pseudoinstruccions d'accés a la memòria es recolzen amb el registre $at per a obtenir l'adreça. Igual que havíem comentat quan estudiàvem les instruccions de bot, aneu amb compte a l'hora de programar i procureu no utilitzar aquest registre perquè l'assemblador podria destruir el valor contingut en ell.

Les pseudoinstruccions d'accés a la memòria es recolzen amb el registre $at per a obtenir l'adreça. Igual que havíem comentat quan estudiàvem les instruccions de bot, aneu amb compte a l'hora de programar, perquè l'assemblador podria destruir el valor contingut en aquest registre.

Instrucció	Exemple	Significat	Comentaris
Load word	lw Rt, k(Rs)	Rt ← Memòria[Rs+k]	Carrega la paraula de l'adreça *Rs*+k en el registre *Rt*.
Load halfword	lh $1, k(Rs)	Rt ← Memòria[Rs+k]	Carrega la mitja paraula de l'adreça *Rs*+k en el registre *Rt*.
Load halword unsigned	lhu Rt, k(Rs)	Rt ← Memòria[Rs+k]	Carrega la mitja paraula sense signe de l'adreça *Rs*+k en el registre *Rt*.
Load byte	lb Rt, k(Rs)	Rt ← Memòria[Rs+k]	Carrega el byte de l'adreça *Rs*+k en el registre *Rt*.
Load byte unsigned	lbu Rt, k(Rs)	Rt ← Memòria[Rs+k]	Carrega el byte sense signe de l'adreça *Rs*+k en el registre *Rt*.
Load label	lw Rt, Etiqueta	Rt ← M[Etiqueta]	Pseudoinstrucció. Carrega la paraula assenyalada per l'Etiqueta en el registre *Rt*.
Store word	sw Rt, k(Rs)	Memòria[Rs+k] ← Rt	Emmagatzema en la paraula de l'adreça *Rs*+k el contingut de *Rt*.
Store halfword	sh Rt, k(Rs)	Memòria[Rs+k] ← Rt	Emmagatzema en mitja paraula de l'adreça *Rs*+k el contingut de *Rt*.
Store byte	sb Rt, k(Rs)	Memòria[Rs+k] ← Rt	Emmagatzema en el byte de l'adreça *Rs*+k el contingut de *Rt*.
Store label	sw Rt, Etiqueta	Memòria[Etiqueta] ← Rt	Pseudoinstrucció. Emmagatzema en la paraula assenyalada per l'Etiqueta el contingut del registre *Rt*.

Taula 2. Instruccions i pseudoinstruccions d'accés a la memòria

6.4.6 Els punters

S'anomena *punter* qualsevol variable (registre o bé una posició de memòria) que conté una adreça de la memòria. Un exemple és el registre Comptador de Programa (PC) que conté l'adreça de la instrucció següent a executar. Hi ha processadors en els quals aquest registre rep el nom de *Instruction Pointer* (IP).

La imatge mental que es fan els programadors amb els punters és el d'una fletxa que *apunta* a una posició de memòria i d'ací prové el nom. Així, diem que el PC apunta a la instrucció següent que executarà el processador.
Ja coneixem la pseudointrucció *la* (*load address*) que assigna una adreça a un registre; ara podem dir que aquesta pseudoinstrucció ens serveix perquè un registre *apunte* a una posició de memòria representada per una etiqueta.

Amb els punter podem fer ús d'operacions de suma i resta. Per exemple, el PC s'incrementa en 4 en cada cicle d'instrucció per a apuntar a la instrucció següent. Amb una instrucció de bot se suma una quantitat al PC per a desplaçar-se cap amunt o cap avall en el codi.

Com que les adreces són sempre positives, les operacions amb punters es fan sempre en binari natural i s'utilitzen exclusivament les instruccions addu i addiu, no té sentit parlar de signe.

Observem ara com es representen els punters definits en alt nivell en el cas particular del llenguatge d'assembladors del MIPS. La següent definició de codi en C:

```
int X = 10;
int * p;
```

Té l'equivalència següent en assemblador del MIPS:

```
.data
X: .word 10
p: .space 4
```

La sentència en llenguatge C:

```
p = &X;    /* p apunta a X */
```

Es podria escriure en assemblador MIPS com:

```
la $s0,X
la $s1,p
sw $s0,0($s1)
```

o si utilitzem pseudoinstruccions, es podria escriure com a::

```
.text
la $s0,X
```

```
sw $s0,p
```

La sentència en llenguatge C:

```
*p = *p + 1; /* incrementa l'enter
                a què apunta p */
```

Es tradueix com a:

```
la $s0,p
lw $s1,0($s0)
addi $s1,$s1,1
sw $s1,0($s0)
```

A continuació aprofundirem en el significat dels punters i en les instruccions d'accés a la memòria mitjançant un codi exemple. Considereu el programa següent en assemblador:

```
##########################################
#                                        #
#     Codi exemple de l'activitat 3      #
#                                        #
#     Accés a la memòria. Punters        #
#                                        #
##########################################

.data             #Definició segament de dades
A:    .word 6
B:    .word 8
C:    .space 4
X:    .byte 1

VAE1: .asciiz "Estructures del Computador"
VAE2: .asciiz "Pràctiques de laboratori\n"
VAE3: .asciiz "\n El resultat de la suma és: "

.text             #Comença el programa
la $a0,VAE1       #Començament cadena en $a0
li $v0,4          #Funció 4. Print string
syscall           #Escriu cadena en consola
li $a0,'\n'
li $v0,11
syscall
la $a0,VAE2
li $v0,4
syscall

la $a0,VAE3
li $v0,4
syscall

la $t0,A          # $t0 = @A
lw $t1,0($t0)     # $t1 = *$t1
lw $t2,4($t0)     # $t1 = *($t0+4)
add $t3,$t1,$t2
```

```
        sw $t3,8($t0)
        move $a0, $t3
        addi $v0,$0,1
        syscall          #Escriu un valor

        li $v0, 10       #Acaba el programa
        syscall
```

6.4.6.1 Activitat 3

- Analitzeu el codi i esbrineu què fa.
- Assembleu el codi i executeu-lo. Què fa la funció 4 per a la instrucció syscall?
- En quina adreça es guarda el resultat de la suma?

Qüestió 6

- Quina és la codificació màquina de la instrucció syscall?

Com hem pogut experimentar en les pràctiques desenvolupades fins ara, la instrucció syscall ens permet interaccionar amb el teclat i la consola per a llegir i escriure diferents tipus de dades, incloses les cadenes de caràcters com heu vist en l'activitat 3. En la taula 3 es recullen algunes de les funcions de crida més utilitzades amb la instrucció syscall.

Servei	Codi de crida	Arguments	Resultat
Print_int	1	$a0=enter	Imprimeix en la consola l'enter en $a0
print_string	4	$a0=Adreça del començament de la cadena	Imprimeix en la consola el string que comença en $a0
Read_int	5		Llig de la consola un enter i el guarda en $v0
Read_string	8	$a0=Adreça del buffer d'entrada $a1=longitud	Llig de la consola un string que no pot ser més llarg de $a1-1 i el guarda a partir de $a0
Exit	10		Finalitza l'execució
Print_char	11	$a0=Caràcter	Imprimeix en la consola el byte de menor pes de $a0
Read_char	12		Llig de la consola un caràcter i el guarda en $v0

Print_int_hex	34	$a0 = enter	Imprimeix un valor de 8 dígits en hexadecimal. Ompli amb zeros a l'esquerra si fóra necessari
Print_int_bin	35	$a0 = enter	Imprimeix un valor de 32 bits i ompli amb ceros a l'esquerra si fóra necessari
Print_int_unsig	36	$a0 = enter	Imprimeix un valor decimal sense signe

Taula 3. Algunes de les funcions Syscall disponible en MARS.

Qüestió 7

- Feu el codi que llig dos enters del teclat. Amb aquesta finalitat heu de mostrar dos missatges en la consola: un primer que demane a l'usuari que introduïsca un valor i una vegada llegit, que mostre un altre missatge demanant el segon valor. Les dades s'emmagatzemaran en posicions consecutives de la memòria; per això, prèviament haureu reservat espai en el segment de dades amb la directiva *.space*. A continuació el programa llegirà els valors guardats en la memòria i els mostrarà en la pantalla.

Qüestió 8

- Modifiqueu el programa de la qüestió 7 perquè mostre en la pantalla les dues dades guardades en la memòria ordenades de menor a major valor.

Qüestió 9

- Modifiqueu el codi de la qüestió 8 en el qual s'afegia una funció que anomenarem *SWAP*. Aquesta funció ha d'intercanviar el contingut de les dues posicions de memòria on estan emmagatzemats els valors llegits de teclat. El programa principal cridarà la funció SWAP amb la instrucció *jal* abans de mostrar les dades en la pantalla.

6.5 Resum

- El codi d'assemblador s'emmagatzema en una zona de la memòria del MIPS anomenada *text segment* i les dades en una zona anomenada *data segment*.

- L'accés a la memòria es realitza mitjançant operacions de lectura i escriptura.

- La memòria de MIPS permet accés per byte, mitja paraula, paraula i doble paraula.

- Les dades definides en la memòria no es poden utilitzar en operacions aritmètiques o lògiques, s'han de portar primer a un registre del banc de registres.