Quadern de laboratori
Estructura de Computadors

Emilio Castillo
José María Cela
Montse Fernández
David López
Joan Manuel Parcerisa
Angel Toribio
Rubèn Tous
Jordi Tubella
Gladys Utrera

Departament d'Arquitectura de Computadors
Facultat d'Informàtica de Barcelona
Quadrimestre de Primavera - Curs 2014/15

Aquest document es troba sota una llicència Creative Commons

Licencia Creative Commons

Esta obra está bajo una licencia Reconocimiento-No comercial-Compartir bajo la misma licencia 2.5 España de Creative Commons. Para ver una copia de esta licencia, visite

`http://creativecommons.org/licenses/by-nc-sa/2.5/es/`

o envie una carta a

Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Usted es libre de:

- copiar, distribuir y comunicar públicamente la obra
- hacer obras derivadas

Bajo las condiciones siguientes:

- **Reconocimiento**. Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).
- **No comercia**l. No puede utilizar esta obra para fines comerciales.
- **Compartir bajo la misma licencia**. Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.
- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor
- Nada en esta licencia menoscaba o restringe los derechos morales del autor.

Advertencia: Este resumen no es una licencia. Es simplemente una referencia práctica para entender el Texto Legal (la licencia completa).
Los derechos derivados de usos legítimos u otras limitaciones reconocidas por ley no se ven afectados por lo anterior.

Sessió 0: Introducció

Objectiu: En aquesta sessió coneixereu l'entorn de treball que fareu servir a les sessions de laboratori. Mitjançant el programa MARS, editareu un programa senzill en assemblador MIPS, generareu un programa equivalent en llenguatge màquina i simulareu la seva execució. Durant la sessió també aprendreu técniques bàsiques de depuració de programes, que us permetran corregir els errors de forma eficient i autònoma.

Lectura prèvia

Desenvolupament de software

A les classes de laboratori escriureu programes en assemblador MIPS i en simulareu l'execució. Per fer-ho utilitzareu el programa MARS[1], que integra les diferents eines que habitualment s'utilitzen per a programar en assemblador, i en simplifica les etapes[2]. Les fases que seguireu per realitzar un programa són les següents: primer es crea el programa escrivint-lo en llenguatge assemblador per mitjà d'un editor de text (el propi MARS en proporciona un). El resultat és un fitxer font en un llenguatge que pot entendre l'usuari, però no la màquina. Per traduïr-lo a llenguatge màquina, el MARS ens proporciona una eina que integra, sota un únic pas, el que normalment serien les etapes d'assemblatge, muntatge i càrrega. El resultat serà el programa en llenguatge màquina tal i com quedaria ubicat en la memòria d'una computadora MIPS just al començament de la seva execució. A partir d'aquest moment, el MARS proporciona eines per simular l'execució del programa pas a pas, i per inspeccionar els efectes que aquesta té sobre els registres i la memòria de la computadora.

Durant el procés de creació d'un programa se solen produir errors. Hi ha dos tipus d'errors: els sintàctics o detectables en temps de traducció, i els semàntics o detectables en temps d'execució. Els errors sintàctics són, per exemple, escriure malament una instrucció o fer una operació entre dos tipus de dades incompatibles. Aquests errors són detectats pel traductor i els hem de solucionar abans de poder generar un executable.

1. Si voleu treballar a casa, podeu descarregar el MARS des de la pàgina web de l'assignatura: http://docencia.ac.upc.edu/FIB/grau/EC
2. A les classes de teoria coneixereu les etapes reals en la transformació d'un programa assemblador en un programa en execució (assemblatge, muntatge i càrrega/execució)

Un cop tenim un programa sintàcticament correcte el podeu executar, però això no implica que el programa sigui correcte. Totes les instruccions poden ser correctes, però pot faltar la condició de final d'un bucle (i que aquest no acabi mai) o que, senzillament, el programa no faci el que volem. Aquests errors només es poden detectar en temps d'execució, i per tal d'eliminar-los haureu de fer servir les eines de depuració de programes que proporciona el MARS[1]. Les eines de depuració permeten executar el programa instrucció a instrucció i veure tots els valors que es van calculant, de manera que podeu trobar els errors.

Al laboratori, l'entorn de desenvolupament dels nostres programes serà sota **Linux**, però el programa que farem servir està escrit en Java i per tant a casa el podreu fer servir en qualsevol plataforma (Windows, Mac, etc.).

El llenguatge assemblador emprat al laboratori serà el MIPS (és el que usen processadors com el MIPS R4000 de 32 bits). Podeu trobar-ne una descripció detallada a l'apèndix B.10 del llibre de Patterson & Hennessy[2] així com una referència ràpida al menú d'ajuda del programa MARS (tecla **F1**), i també en un opuscle que publiquem a la web d'EC.

Introducció al Llenguatge Assemblador MIPS R4000

El codi incomplet de la Figura 0.1 mostra l'esquelet d'un programa en assemblador MIPS.

```
     .data
nomvariable1: .word 0            # comentari
nomvariable2: .word 0, 0, 0, 0   # vector d'enters

#    ...                           altres variables
     .text
     .globl main                 # fem main global per cridar-la de fora
main:
     addu   $t0, $t0, $t1        # Instrucció suma
#    ...                           altres instruccions
etiq:                            # Etiqueta per a un salt
#    ...                           altres instruccions
     b      etiq                 # Salt a etiq

#    ...                           altres instruccions

     jr     $ra                  # Retorna
```

Figura 0.1: Esquelet d'un programa en assemblador MIPS.

Observeu que els comentaris s'escriuen utlitzant el símbol "#", i que les paraules començades amb un "." són directives del llenguatge. És important tenir en compte que el llenguatge és *case-sensitive*, és a dir, que diferencia entre majúscules i minúscules.

1. Les eines de depuració normalment les proporciona un programa independent (depurador o debugger)
2. D. A. Patterson and J. L. Hennessy. Computer Organization and Design. The Hardware/Software Interface, 4th edition, Morgan Kaufmann.

Quan el programa es tradueix a codi màquina, el muntador hi afegeix un petit fitxer amb l'anomenat codi de "startup" (*startup.s*), que en el nostre cas conté el següent codi:

```
jal    main
li     $v0, 10
syscall
```

La 1a instrucció simplement salta a l'etiqueta *main* (fa una "crida a *main*", s'estudiarà al tema 3). Per tal que l'etiqueta main sigui referenciable des d'un fitxer extern com aquest cal que la declarem global, i és per això que al inici del nostre programa hem escrit: `.globl main`.
Al final de la funció *main* observaràs que apareix la instrucció: jr $ra. Aquesta instrucció retorna al punt on ha estat invocada la funció main (s'estudiarà també al tema 3), de manera que s'executaran les dues darreres instruccions del codi de "startup". Aquestes invoquen la "crida al sistema" `exit()`, que finalitza el programa, allibera els recursos que estava usant (memòria, etc.) i retorna el control al sistema operatiu. S'explicarà més endavant en el curs.

Registres

MIPS és una arquitectura del tipus *load store*, això vol dir que únicament les instruccions load i store (lw, sw, etc.) accedeixen a memòria. Totes les instruccions de còmput (aritmètiques, etc.) treballen exclusivament sobre els 32 registres de 32 bits de ofereix el processador (a banda dels disponibles als coprocessadors, que es veuran més endavant). Els registres tenen 2 noms alternatius, número ($0, $1, etc.) i nom alfanumèric ($zero, $at, etc.), tots dos es poden usar al codi. A continuació es mostra la llista dels 32 registres (els que fareu servir en aquesta sessió apareixen ressaltats en negreta):

```
Número    : Nom
$0        : $zero   (conté el valor 0, es pot llegir però no modificar)
$1        : $at     (registre temporal reservat per a pseudoinstruccions)
$2-$3     : $v0-$v1 (resultats de funcions i expressions)
$4-$7     : $a0-$a3 (paràmetres en una subrutina)
$8-$15    : $t0-$t7 (temporals)
$16-$23   : $s0-$s7 (temporals que es preserven en una crida a una subrutina)
$24-$25   : $t8-$t9 (temporals)
$26-$27   : $k0-$k1 (reservats per al S.O.)
$28       : $gp     (global pointer)
$29       : $sp     (stack pointer)
$30       : $fp     (frame pointer)
$31       : $ra     (return address)
```

Durant aquesta sessió farem servir els registres temporals $t0..$t7.

Instruccions

L'ajuda del programa MARS (tecla **F1**) us mostra un llistat de les instruccions que soporta el simulador. El següent exemple mostra la traducció d'una expressió en C a assemblador MIPS, suposant que f, g, h, i, j ocupen $s0, $s1, $s2, $s3, $s4:

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

En MIPS:

```
addu    $t0, $s1, $s2              # $t0 = g + h
addu    $t1, $s3, $s4              # $t1 = i + j
subu    $s0, $t0, $t1              # f = (g + h) - (i + j)
```

Les següents taules mostren un resum de les instruccions i les pseudoinstruccions MIPS que s'empraran en aquesta sessió de laboratori. Un llistat complet de les que es veuran a EC el pots trobar al document "Instruccions i Macros MIPS" penjat a la web d'EC:

Instruccions MIPS		
SINTAXI	SEMÀNTICA	COMENTARI
#load upper immediate `lui rd, imm16`	rd<31..16> = imm16 rd<15..0> = 0x0000	Copia imm16 a la part alta i posa a 0 la part baixa
#OR immediate `ori rd, rs, imm16`	rd = rs or ZeroExtImm16	Operació bit a bit. Imm16 s'extén a zeros a la part alta
#add unsigned `addu rd, rs, rt`	rd = rs + rt	- Dóna el mateix resultat que add, per realitzar suma enters, però no causa excepció d'overflow.
#add immediate unsigned `addiu rt, rs, imm16`	rt = rs + SignExtImm16	- S'extén el bit de signe de l'immediat (de 16 bits) a la part alta. - Dóna el mateix resultat que addi, per realitzar suma d'enters, però no causa excepció d'overflow.
#load word `lw rt, offset6(rs)`	rt = M[rs + SignExtOffset6]	Llegeix un word a la posició de memòria donada per rs + SignExtOffset6
#store word `sw rt, offset6(rs)`	M[rs + SignExtOffset6] = rt	Escriu un word a la posició de memòria donada per rs + SignExtOffset6
#set less than `slt rd, rs, rt`	Si (rs < rt) llavors rd=1 sino rd=0	Comparació d'enters
#branch if equal `beq rs, rt, etiq`	Si (rs == rt) llavors salta a etiq	
#branch if not equal `bne rs, rt, etiq`	Si (rs != rt) llavors salta a etiq	
#shift left logical `sll rd, rs, shamt`	rd = rs << shamt	Desplaçament lògic a l'esquerra de shamt posicions. Shamt és un número natural de 5 bits. Els desplaçaments negatius no estan permesos.

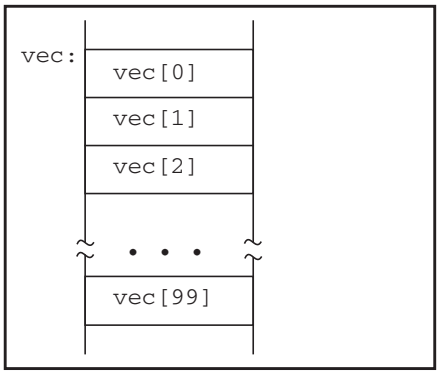
Pseudoinstruccions MIPS		
SINTAXI	SEMÀNTICA	EXPANSIÓ
#load immediate `li rdest, imm`	rdest = imm Carrega l'immediat a un registre	si (imm >16bits) lui $at, hi(imm) ori rdest, $at, lo(imm) si (imm <=16bits) addiu rdest, $szero, imm
#load address `la rdest, etiq`	rdest = adreça d'etiq	lui $at, hi(etiq) ori rdest, $at, lo(etiq) Copia l'adreça d'una etiqueta a un registre
#branch `b etiq`	Salt incondicional a etiq	beq $0,$0 etiq

Declaració i emmagatzematge d'un vector

Els programes d'exemple d'aquesta sessió de laboratori fan un ús bàsic d'un vector. Els vectors són agrupacions unidimensionals d'elements de tipus homogenis, els quals s'identifiquen per un índex. Els elements es guarden en posicions consecutives de memòria, a partir de l'adreça inicial del vector, respectant les regles d'alineació dels elements.

EXEMPLE: En C, declarem la variable global *vec* com un vector de 100 elements *int* així (*vec* es guardarà en una adreça múltiple de 4):

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



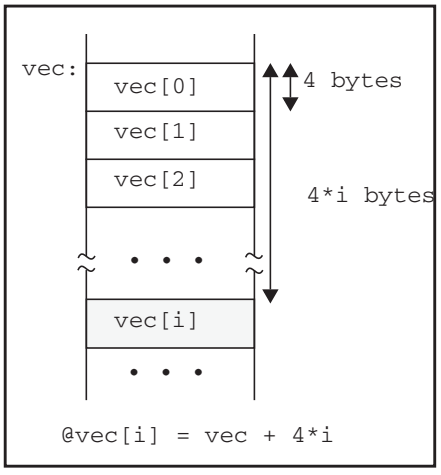
En MIPS serà:

```
     .data
vec: .space 400
```

Accés (aleatori) a un element d'un vector

Els elements d'un vector s'especifiquen per un índex. En C, el primer element sempre té índex zero. Per exemple, la sentència:

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



En MIPS serà, suposant que *i* està en $s1 i que vec és global:

```
     .data
     la     $t0, vec            # vec
     sll    $t1, $s1, 2         # 4*i
     addu   $t0, $t0, $t1       # @vec[i]
     li     $t1, 10
     sw     $t1, 0($t0)
```

Enunciats de la sessió

Activitat 0.A: Autentificació i primers passos de la sessió

Les pràctiques de l'assignatura es poden fer en els laboratoris del DAC, al mòdul D6 o bé en els de la FIB, als mòduls A5, B5 o C6. La identificació en el sistema és diferent segons l'aula on es treballi. Quan es posen en marxa surt un menú que permet triar el sistema operatiu en el que volem treballar.

A les **aules del DAC** heu de triar la imatge **EC** i es carregarà el sistema Linux. Un cop carregat (pot trigar uns quants minuts), pot ser que us demani autentificació, i heu d'entrar al sistema amb el nom d'usuari **`alumne`**, i amb clau d'accés **`alumne`**. Per fer les sessions normals en tenim prou amb aquest compte. Per fer la sessió d'examen entrarem en uns comptes especials que s'explicaran en el mateix moment de l'examen. Un cop dins el sistema cal saber que teniu accessible el disc dur local del PC en què treballeu per guardar-hi els fitxers de treball. Aquesta zona de disc pertany a l'usuari genèric "alumne", hi té accés qualsevol, i s'esborra cada cop que es reinicialitza. Per tant, la manera de conservar els fitxers creats d'una sessió per una altra és endur-se'ls en una clau de memòria USB. Sols cal insertar-la i en pocs instants estarà visible el seu contingut, des del navegador, o bé des d'una finestra de terminal.

Cada vegada que s'engega el PC s'ha de carregar la imatge del sistema operatiu triat. Aquesta operació és costosa: pot trigar de l'ordre de 3 minuts a baixar la imatge, descomprimir els fitxers i posar en marxa el sistema operatiu. Només ens podrem estalviar aquesta operació quan l'últim usuari que hagi estat en el mateix PC, hagi utilitzat Linux, i hagi sortit de la sessió sense apagar el PC: amb l'opció "Surt... -> Sortir de la sessió" que es troba associada a una icona amb el nom "alumne" a l'extrem dret de la barra de menús de l'escriptori (o bé amb la comanda **`logout`**, si es fa des d'una finstra de terminal).

A les **aules de la FIB** heu de triar la imatge 'Linux'. Un cop carregada heu d'entrar al sistema amb el vostre nom d'usuari i password personal. Un cop engegat el sistema, tindreu accés a la vostra zona de disc personal al servidor, on anireu guardant els vostres fitxers de treball durant el curs. També en aquest cas podem llegir i copiar fitxers en una clau USB, però no és indispensable. El procediment requereix insertar la clau, i clicar la icona dels "dispositius USB". Abans de retirar la clau és convenient seguir el mateix procés, però seleccionant "Desmuntar dispositiu".

En aquest curs usarem la "finestra de terminal", un tipus de programa que ens ofereix una interfície de comandes per activar altres programes del sistema i gestionar fitxers. S'activa des d'una opció del menú, o simplement clicant la icona de terminal a la barra d'aplicacions.

Un cop oberta la finestra de terminal, podem **l**li**s**tar els fitxers del nostre directori principal, i observar-ne la grandària, les dates de la darrera modificació i altres informacions:

```
$ ls -l
```

També podem crear (**m**a**k**e) o esborrar (**r**e**m**ove) subdirectoris:

```
$ mkdir nomcarpeta
$ rmdir nomcarpeta
```

Podem **c**anviar de **d**irectori de treball, a un subdirectori, esbrinar quin és el seu camí complet des de l'arrel del disc (**p**rint **w**orking **d**irectory), i retornar al directori 'pare' inicial:

```
$ cd nomcarpeta
$ pwd
$ cd ..
$ pwd
```

Si hem insertat la clau de memòria seguint les indicacions de la pàgina anterior (depèn de l'aula), podem examinar els fitxers de la clau de memòria i **c**o**p**iar-hi qualsevol fitxer del nostre directori. A les aules de la FIB les comandes serien:

```
$ ls -l /media/usb
$ cp fitxer_a_salvar /media/usb
$ ls -l /media/usb
```

Al DAC, el directori de la clau de memòria es diu: '/media/USB MEMORY'

A través de la xarxa tindreu accés a un directori anomenat **`/assig/ec`**. Aquest directori només és de lectura i no hi podeu escriure. En aquest directori hi haurà el software de simulació i depuració de programes escrits en MIPS, així com els fitxers-plantilla de cada sessió. Al principi de cada sessió, inclosa aquesta sessió 0, heu de copiar els fitxers corresponents al vostre espai de treball. Executeu la següent comanda (atenció! el segon argument de la comanda **cp** és un punt):

```
cp  /assig/ec/sessio0/* .
```

Si ara mireu el contingut del vostre directori de treball amb la comanda

```
ls -l
```

veureu alguns fitxers com el **`s0a.s`**. Aquest fitxer conté el programa assemblador que farem servir posteriorment (Figura 0.2). A partir d'ara veurem el procés de desenvolupament d'un programa en llenguatge assemblador MIPS.

Activitat 0.B: Edició de fitxers

Verifiqueu que heu copiat els fitxers plantilla al vostre directori personal tal com s'explica a l'activitat 0.A. Totes les etapes (inclosa l'edició del programa en assemblador o font) les realitzarem amb el programa MARS. Per tal d'executar aquest programa caldrà fer[1]:

```
$ java -jar /assig/ec/Mars/Mars.jar
```

Alternativament, podeu obrir la carpeta fent doble clic a la icona *EC* de l'escriptori, navegar a la subcarpeta *Mars*, i fer clic amb el botó dret del ratolí sobre el fitxer *Mars.jar*, seleccionant *Executar amb Java 1.6* d'entre les opcions que es mostren.

Un cop obert el simulador Mars, seleccioneu l'opció de menú *File/Open,* per obrir el fitxer font **s0a.s**, que prèviament heu copiat en algun dels vostres directoris:

```
     .data
V:   .word -1, -2, -3, -4, -5, -6, -7, -8, -9, -10    # vector d'enters

     .text
     .globl main
main:
     li     $t0, 0               # $t0 = 0 (variable suma)
     li     $t1, 0               # $t1 = 0 (comptador del bucle i)
     li     $t2, 10              # $t2 = 10 (límit del comptatge)
bucle:
     slt    $t3, $t1, $t2
     beq    $t3, $zero, fibucle  # Després de 10 voltes acaba
     la     $t3, V               # $t3 = Adreça inicial de V
     sll    $t4, $t1, 2          # $t4 = 4*i (cada element ocupa 4 bytes)
     addu   $t3, $t3, $t4        # @V[i] = @V + 4*i
     lw     $t3, 0($t3)          # Llegim un element $t3 = V[i]
     addu   $t0, $t0, $t3        # Acumulem. suma = suma + V[i]
     addiu  $t1, $t1, 1          # i++
     b      bucle                # salt incondicional
fibucle:
```

Figura 0.2: Programa `s0a.s`, que suma els elements de V, i ho guarda a `$t0`.

```
int V[N] = {-1, -2, -3, -4, -5, -6, -7, -8, -9, -10};
void main() {
     int suma = 0, i = 0;
     while ( i < 10) {
          suma = suma + V[i];
          i++;
     }
}
```

Figura 0.3: Programa equivalent en C, que calcula la suma dels elements de V.

1. Als laboratoris ja tindreu instal·lat el Java, i la ruta a l'executable *java* correctament afegida a la variable d'entorn *path.* Si ho voleu provar a casa, potser us haureu de descarregar el Java (http://java.sun.com/javase/downloads/widget/jdk6.jsp) i modificar la variable *path* (http://www.troubleshooters.com/linux/prepostpath.htm)

Activitat 0.C: Assemblatge i depuració

La generació de codi màquina a partir del codi assemblador la farem seleccionant l'opció de menú *Run/Assemble* o prement la tecla *F3*. Aquesta comanda assemblarà el fitxer de codi font `s0a.s` i carregarà a la memòria el codi màquina resultant. Cal tenir en compte que el procés d'assemblatge pot generar errors. Sempre que ens trobem amb un error, el que haurem de fer és fixar-nos en el comentari que l'acompanya i anotar el número de línia on s'ha detectat. Llavors cal intentar corregir-lo editant novament el fitxer i tornant a assemblar el programa. Això és un procés iteratiu fins que no s'obtinguin errors.

Per exemple, editeu el fitxer `s0a.s`, i canvieu `li $t2,10` per la instrucció `li $T2,10`. A continuació torneu a assemblar el programa i observeu el missatge d'error. Ara corregiu l'error, i torneu-lo a assemblar. A continuació s'explicaran breument les comandes més usuals de depuració. És convenient cridar totes les comandes de depuració amb el teclat (no amb el ratolí!), mitjançant combinacions de tecles (hotkeys). La majoria estan disponibles en el menú *Run*. Per exemple, prement la tecla **F5** executareu el vostre programa tot d'una tirada.

Activitat 0.D: Panells del MARS

Un cop assemblat el programa, Mars mostra automàticament la pestanya "Execute" amb l'aparença inicial que es pot veure a la Figura 0.4, i està formada per 5 *panells* o finestres. Cada

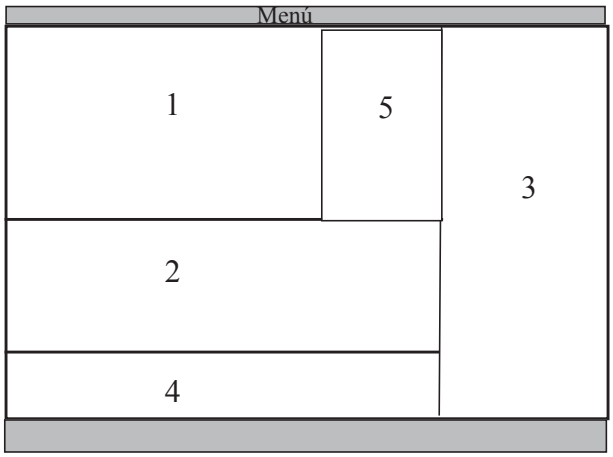


Figura 0.4: Els 5 panells

panell conté un tipus d'informació. Completeu l'exercici 0.1

Exercici 0.1: Podeu provar per exemple a modificar la configuració inicial de panells sel·leccionant/desel·leccionant al menú *Settings* l'opció "*Show labels window*" per fer aparèixer o desaparèixer el panell 5. Els canvis que hi feu es conservaran d'una sessió per una altra.

Activitat 0.E: Panell del Codi Desassemblat

La Vista del Codi Desassemblat es troba inicialment al panell 1, i mostra una porció de la memòria desassemblada, una instrucció per línia. Cada línia està composta de cinc columnes. A la segona columna apareixen les adreces de memòria en hexadecimal. A la tercera columna hi ha el contingut de memòria en hexadecimal (el codi de cada instrucció), i a continuació la instrucció en assemblador. A la darrera columna es mostra la instrucció del *programa font* tal com està escrita en el fitxer del disc, amb les etiquetes, comentaris, etc. En molts casos les dues darreres columnes seran idèntiques, però en altres es veurà que una instrucció del *programa font* (una pseudoinstrucció per exemple) pot donar lloc a una o més instruccions diferents de les que el programador ha escrit. Completeu l'exercici 0.2

Exercici 0.2: Torneu a l'editor i canvieu la instrucció `li $t2, 10` per `li $t2, 65536`. A continuació torneu a assemblar el programa i observeu com la pseudoinstrucció `li` ara s'ha expandit en dues instruccions d'assemblador en comptes de només una. Escriviu quines instruccions són:

lui $at, 0x00000001
ori $t2, $at, 0x00000000

Activitat 0.F: Vista de Dades de memòria

Es troba en el panell 2, mostra el contingut d'una porció de la memòria, en un format compacte: en cada línia de pantalla es mostra primer una adreça, i a continuació les 8 dades de mida word (32 bits) emmagatzemades en aquesta i en les següents adreces de memòria, fins a omplir la línia de pantalla (32 bytes). Per tant, l'adreça mostrada en primer lloc correspon tan sols a la primera de les dades de la línia. Les dades es poden mostrar en decimal o hexadecimal seleccionant-ho a les opcions que apareixen a sota. Completeu l'exercici 0.3

Exercici 0.3: Intenteu localitzar a la vista de dades les 10 cel·les que contenen el vector V (valors -1, -2, etc). Feu clic al símbol V al panell 5 'Labels window' i mireu com es ressalta el primer element de V a la vista de dades (panell 2). Proveu a canviar la visualització dels valors de hexadecimal a decimal per tal de veure el contingut del vector en diferents formats. En quina adreça de memòria està guardat V[0]?

hexa: 0x10010000
decimal: 268500992

Activitat 0.G: Vista dels Registres

Es troba en el panell 3 i consta de 3 pestanyes. A la primera pestanya anomenada '*Registers*' es mostra el contingut dels 32 registres de propòsit general, el `pc`, i els registres `hi` i `lo` que s'estudiaran més endavant. Els seus valors es poden veure en decimal o hexadecimal, seleccionant-ho a les opcions que apareixen a la vista de dades (panell 2)Les altres dues pestanyes '*Coproc0*' i '*Coproc1*' mostren els registres dels coprocessadors en hexadecimal i decimal.

Activitat 0.H: Execució de programes

Aquesta és la funcionalitat més important del MARS. Hi ha dues maneres principals d'executar un programa: execució pas a pas o bé execució contínua amb punts d'aturada. Haureu de saber fer servir les dues per poder depurar errors de forma eficient.

a) Execució Pas a pas: El PC sempre conté l'adreça de la següent instrucció a executar, la qual està resaltada en color al panell 1 (punt d'execució). Prement la tecla **F7** s'executa la instrucció apuntada pel PC i s'actualitza automàticament el valor dels registres i dades de memòria en tots els panells. Si s'ha fet un accés a un registre, aquest queda ressaltat. Aquest és el mètode més útil per a programes petits com els que nosaltres escriurem. Completeu l'exercici 0.4

Exercici 0.4: Deixeu el programa `s0a.s` tal com estava a l'inici, i executeu-lo pas a pas prement F7. Observeu com es ressalten els diferents registres i com el registre $t0 va acumulant el resultat.

- **Reinicialització** dels registres (també del PC) i variables de memòria del programa: Premeu **F12** per tornar a executar el programa des del principi.

b) Execució contínua: Prement la tecla **F5**, el programa s'executarà fins al final (o fins al primer punt d'aturada que trobi, segons veureu al següent apartat). És el mètode més útil per veure ràpidament els resultats d'un programa llarg, suposant que no té errors. Completeu l'exercici 0.5:

Exercici 0.5: Premeu F12 per recarregar el programa i executeu-lo fins al final amb F5. Quin és el valor del registre $t0? 0xFFFFFFC9

Observeu que el valor de la suma de tots els elements del vector es -55.

c) Execució contínua amb punts d'aturada: Sovint, els programes tenen errors i cal executar-los pas a pas per descobrir l'error. Però si són molt llargs, resulta pesat i és molt més pràctic executar-los de forma contínua, però aturant-nos en algun punt clau. Aquest punt rep el nom de punt d'aturada, i és una marca situada en una instrucció perquè el programa s'aturi abans d'executar-la. Per afegir un punt d'aturada sols cal marcar la instrucció on volem ubicar-lo a la columna de l'esquerra del panell de Codi Desassemblat. Es poden afegir i treure tants punts d'aturada com es vulguin o eliminar-los tots de cop amb l'opció del menú corresponent. Polsant **F5** s'executarà el programa de forma contínua fins al primer punt d'aturada que trobi. Tornant a polsar F5, el programa continua executant-se fins al següent punt d'aturada que trobi, ja sigui el mateix o un altre. Completeu l'exercici 0.6

Exercici 0.6: Reinicieu de nou el programa amb F12. Al panell 1 (codi) afegiu un punt d'aturada sobre la instrucció `addu $t0, $t0, $t3` (situada després del `lw`). A continuació executeu fins al punt d'aturada polsant F5. Observeu com es resalta la instrucció que heu marcat. Observeu a la vista de Registres que $t3 ha estat modificat. El seu valor és `-1`, ja que és el primer element del vector. Si aneu polsant F5 repetidament, anireu executant iteració per iteració i podreu comprovar quins valors llegeix de la memòria la instrucció `lw`. Quins valors pren el registre $t3 al llarg de l'execució de tot el programa?: -1, 1, 0x10010000, 0x10010004, -2, 0x10010008, -3, 0x1001000C, -4, 0x10010010, -5, 0x10010014, -6, 0x10010018, -7, 0x1001001C, -8, 0x10010020, -9, 0x10010024, -10, 0x00000000

Activitat 0.I: Modificació de dades de memòria i de registres

Per poder modificar el contingut d'un registre o posició de memòria únicament heu de sel·leccionar la cel·la del panell corresponent, esborrar el seu valor i a continuació introduir el nou valor. Completeu l'exercici 0.7

Exercici 0.7: Recarregueu el programa de nou (F12). Modifiqueu el contingut d'alguns elements del vector. Per exemple, modifiqueu el primer element del vector, posant-li valor 0x00000001. Després, modifiqueu el darrer element del vector posant-li el valor -11.

A continuació executeu el programa fins al final (F5). Comproveu que el resultat de la suma que tenim a la variable $t0 ha de ser ara -54.

Activitat 0.J: Depuració de programes erronis pas a pas

El programa que us donem en s0a.s ja és correcte. Tanmateix, en moltes ocasions haurem programat un codi sense cap error d'assemblatge però que després no fa la tasca esperada. Ens caldrà utilitzar les eines de depuració per trobar l'error. Tanmateix, no hi ha un procediment universal de depuració, ja que depèn de cada cas. El més recomanable és anar executant el programa pas a pas o amb punts d'aturada, i comprovant que cada pas fa el que s'espera que faci, per detectar el punt on comença a fallar. Descobrir els *bugs* és tot un art.

El fitxer `s0b.s` conté una nova versió del programa `s0a.s` (vegeu Figura 0.5), la suma dels elements del vector, però fent alguna optimització. L'avaluació de la condició s'ha ubicat al final (com en un do-while de C) i es fa l'accés als elements del vector començant pel final (V[9], V[8], etc.). Malauradament, el programa conté alguns errors.

```
     .data
V:   .word -1, -2, -3, -4, -5, -6, -7, -8, -9, -10 # vector d'enters

     .text
     .globl main
main:
     li     $t0, 0              # Posem $t0 a 0 (variable suma)
     li     $t1, 9              # Posem $t1 a 9 (comptador del bucle i)
     la     $t3, V              # $t3 = Adreça inicial de V
bucle:
     sll    $t4, $t1, 2         # $t4 = 4*i (cada element ocupa 4 bytes)
     addu   $t4, $t3, $t4       # @V[i]
     lw     $t4, 0($t4)         # Llegim un element $t4 = V[i]
     addu   $t0, $t0, $t4       # Acumulem suma = suma + V[i]
     addiu  $t1, $t1, -1        # i--
     bne    $t1, $zero, bucle
fibucle:
     jr     $ra                 # Retorna
```

Figura 0.5: Programa (erroni) `s0b.s`, que suma els elements de V, i ho guarda a `$t0`.

Completeu l'exercici 0.8 per localitzar l'error:

Exercici 0.8: Depurarem (pas a pas) el programa erroni.

- Obriu el programa s0b.s al simulador i assembleu-lo amb F3.
- Executeu-lo amb F5 i observeu que el resultat de $t0 no és -55 sinó -54.
- Recarregueu el programa amb F3. Ara executeu-lo pas a pas (amb F7) intentant esbrinar què provoca l'error. Quin és exactament l'error que hem introduït?

no se l'hi suma el primer element del vector (0x10010000

L'error que heu detectat es pot solventar de moltes maneres, però ens interessa fer-ho sense incrementar el número d'instruccions del bucle. Una possible solució seria la que a continuació es mostra en alt nivell, que és correcte, però no es correspon amb el programa en assemblador:

```
int V[N] = {-1, -2, -3, -4, -5, -6, -7, -8, -9, -10};
void main() {
     int suma, i;
     suma = 0;
     i = 10;
     do {
          i--;
          suma = suma + V[i];
     } while (i != 0);
}
```

Figura 0.6: Programa (correcte) equivalent en C, que calcula la suma dels elements de V.

Completeu l'exercici 0.9

Exercici 0.9: Correcció del programa anterior.

- Basant-vos en la versió proposada en alt nivell, corregiu el programa del fitxer s0b.s. Únicament heu de canviar una línia de codi de posició i modificar una constant. Quina és aquesta instrucció?

  li $t1, 9 —> li $t1, 10
  addiu $t1, $t1, -1 —> principi del bucle

- Executeu-lo amb F5 i observeu que el resultat final en $t0 és -55.

Activitat 0.K: Depuració de programes mitjançant punts d'aturada

Aquest exercici de depuració parteix del programa en C (correcte), que es mostra a la Figura 0.7 i que calcula la mitjana aritmètica dels elements d'un vector.

El programa comença sumant a la variable *suma* els elements d'un vector (com l'altre), i calculant a la variable *i* el nombre total d'elements. Després calcula la mitjana fent la divisió *suma* / *i*, però aquest programa té la particularitat que fa la divisió per mitjà un bucle que va restant el divisor (el nombre d'elements del vector) al dividend (la suma) tantes vegades com sigui possible.

```
int V[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

void main()
{
     int suma, i, dividend, divisor, quocient;
     suma = 0;

     /* Calculem la suma */
     i = 0;
     while ( i < 10) {
          suma = suma + V[i];
          i++;
     }

     /* Calculem la mitjana aritmètica (suma/i) */
     dividend = suma;
     divisor = i;
     quocient = 0;                /* això serà la mitjana */
     while (dividend >= divisor) {
          dividend = dividend - divisor;
          quocient++;
     }
}
```

Figura 0.7: Programa (correcte) en C que calcula la mitjana dels elements de V.

Se suposa que un programador l'ha traduït a assemblador MIPS, i el resultat es mostra a la Figura 0.8. El programa sembla correcte però el programador ha comès un error, que haurem d'intentar trobar. Observem que la variable *suma* ocupa el registre $t0, la variable *i* ocupa el registre $t1, i el *quocient* de la divisió queda en el registre $t3.

```
     .data
V:   .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10       # vector d'enters
     .text
     .globl main
main:
     li     $t0, 0               # suma = 0
     li     $t1, 0               # i = 0
     li     $t2, 10              # $t2 = 10 (límit del comptatge)
bucle:
     slt    $t3, $t1, $t2        # condició i<10
     beq    $t3, $zero, fibucle  # si és falsa (zero), surt del bucle
     la     $t3, V               # $t3 = adreça inicial de V
     sll    $t4, $t1, 2          # $t4 = 4*i (cada element ocupa 4 bytes)
     addu   $t3, $t3, $t4        # @V[i] = @V + 4*i
     lw     $t3, 0($t3)          # llegim V[i] a la memòria
     addu   $t0, $t0, $t3        # suma = suma + V[i]
     addiu  $t1, $t1, 1          # i++
     b      bucle                # salt enrera incondicional
fibucle:

     li     $t3, 0               # $t3 = 0 (quocient suma/i)
bucle2:
     slt    $t4, $t0, $t1
     bne    $t4, $zero, fibucle2
     subu   $t0, $t1, $t0
     addiu  $t3, $t3, 1
     b      bucle2
fibucle2:
     jr     $ra
```

Figura 0.8: Programa equivalent (erroni) `s0c.s`, que calcula la mitjana dels elements de V.

El programa es troba ja escrit en el fitxer `s0c.s`. Obriu-lo dins el simulador, estudieu-lo bé per entendre el que fa. Completeu l'exercici 0.10.

Exercici 0.10: Per localitzar l'errada del programa utilitzarem punts d'aturada.

- Obriu el programa s0c.s a l'editor i executeu-lo d'una tirada amb F5. Observeu que el resultat a $t3 dóna 1, quan en realitat hauria de donar 5 (55 div 10).
- Recarregueu el programa amb F3. Marqueu un punt d'aturada a la línia que conté la instrucció `li $t3, 0`. Ara executeu tota la primera part (la que calcula la suma dels elements de V) fent F5.
- Un cop arribats al punt d'aturada, executeu la part final pas a pas (amb F7) intentant esbrinar quina línia de codi provoca l'error. Si heu d'inicialitzar el simulador, millor feu F12 en comptes de F3, ja que així conservareu el punt d'aturada.
- Quina instrucció cal modificar? Escriu-la un cop corregida:

subu $t0, $t1, $t0 —> subu $t0, $t0, $t1