ESTRUCTURA DE COMPUTADORS
Grau en Enginyeria Informàtica

Sessió de laboratori número 1

REGISTRES I MEMÒRIA PRINCIPAL

Objectius

- Entrar de nou en contacte amb el simulador SPIM.
- Presentar els aspectes bàsics de l'arquitectura MIPS: banc de registres, ALU, memòria principal.
- Repassar les bases de la programació en assemblador de l'arquitectura MIPS.
- Reprendre contacte amb la manipulació de registres, pseudoinstruccions i instruccions màquina, adreces de memòria principal i dades enteres en l'assemblador del MIPS.
- Comprovar el funcionament del mecanisme de crida al sistema mitjançant la impressió d'un nombre enter.

Bibliografia

- D.A. Patterson i J. L. Hennessy, *Estructura y diseño de computadores*, Reverté, capítol 2, 2011.

Introducció teòrica

El banc de registres de propòsit general

El processador MIPS té altres bancs de registres, però no anem a estudiar-los ara mateix. El banc d'enters, o de registres de propòsit general, permet fer càlculs aritmètics sobre adreces i sobre dades de tipus enter.

Amb excepció de **`$0`** i **`$31`**, tots els registres són idèntics i aprofiten per a les mateixes coses. El registre **`$0`** conté sempre el valor zero i el registre **`$31`** està lligat a la instrucció **`jal`**. Tanmateix, per facilitar la compilació habitual dels programes escrits en alt nivell, s'ha fet un repartiment **convencional** dels registres que es detallarà en aquesta sèrie de pràctiques. L'ús del conveni es fa més senzill amb el nou bateig dels registres, reconegut per l'assemblador del simulador PCSpim. Els detalls més útils del conveni d'ús dels registres s'explicarà al llarg d'aquestes i altres pràctiques quan farà falta.

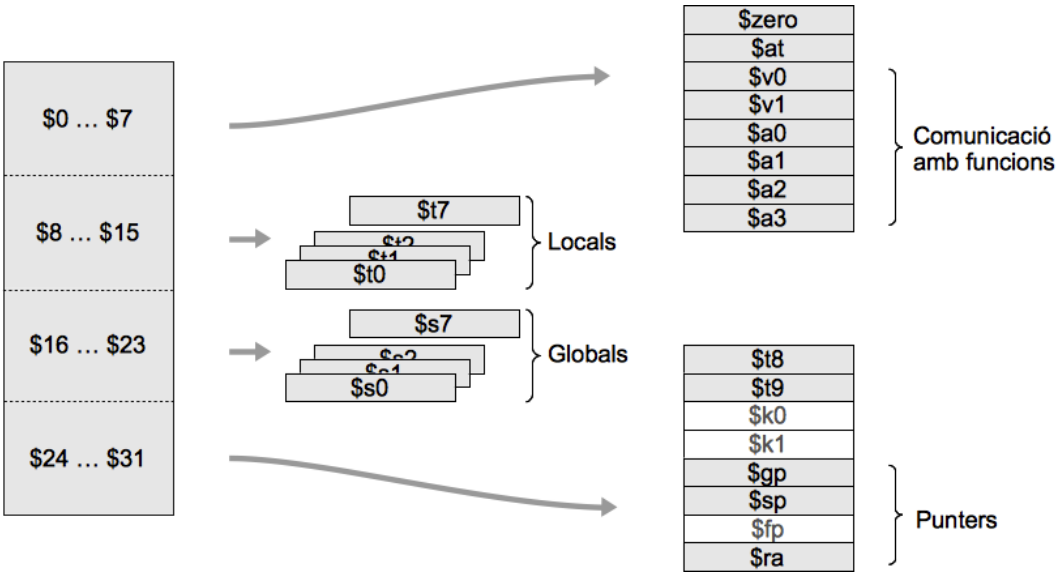

Figura 1. Convenció d'ús dels registres. Hi podem fer-ne quatre grans blocs de 8 registres: comunicació amb les funcions ($0 a $7), variables locals ($8 a $15), variables globals ($16 a $23) i diversos ($24 a $31). Entre els primers hi ha els dos registres $zero i $at; entre els últims hem marcat els registres que utilitzareu en aquestes pràctiques.

Instruccions i pseudoinstruccions

A cada cicle d'instrucció, el processador executa una instrucció màquina. Moltes instruccions fan per sí mateix una operació elemental amb utilitat ben definida expressada en el seu mnemònic (una operació aritmètica, un accés de lectura o escriptura en la memòria, un salt); però hi ha casos en què una acció elemental s'obté fent un ús molt particular d'una instrucció màquina de propòsit molt general i hi ha d'altres en què l'acció necessita dos o tres instruccions de màquina per limitacions del joc d'instruccions del processador.

Les pseudoinstruccions permeten expressar aquestes accions elementals en una línia amb mnemònic i operands que segueixen la sintaxi comú reconeguda per l'assemblador però que no representen una instrucció nova. El resultat és molt més llegible i s'ajusta millor a l'univers mental del programador en assemblador. Vegem-ne un exemple de cada cas:

- La pseudoinstrucció **`move rs,rt`** expressa la operació de còpia entre registres (*rs* = *rt*). Aquesta operació es pot obtenir com a cas particular d'una instrucció més general, com **`or rs,rt,$zero`** o **`addi rs,rt,0`**.
- La pseudoinstrucció **`li`** (*load immediate*) resol un problema de programació bàsic: assignar una constant **`K`** a un registre. El programador en assemblador només ha d'escriure una línia: **`li $rt,K`**. Depenent del valor de **`K`**, tindrem els tres casos que mostra la taula següent. Cal entendre que **`K`** pot tindre fins a 32 bits, que **`Kh`** representa els 16 bits més significatius de **`K`** i **`Kl`** els 16 bits menys significatius:

Cas	Exemple	Traducció
Kh=0	**K = 0x00000100** **Kl = 0x0100**	**ori $rt,$0,Kl**
Kl=0	**K = 0x00040000** **Kh = 0x0004**	**lui $rt,Kh**
Kh≠0 i Kl≠0	**K = 0x00040010** **Kl = 0x0010** **Kh = 0x0004**	**lui $at,Kh** **ori $rs,$at,Kl**

Figura 2. Diverses traduccions de **li $rt,K**, on la constant **K** de 32 bits es descompon en la part alta **Kh** i la part baixa **Kl**, totes dues de 16 bits.

- La pseudoinstrucció **la** (*load address*) també resol un altre problema de programació bàsic i, al seu torn, importantíssim: copia l'adreça de memòria d'una etiqueta en un registre. Aquesta funcionalitat facilita enormement la programació en assemblador perquè permet al programador oblidar-se de les adreces de memòria exactes on s'hi ubiquen les variables i centrar-se només en el nom que aquestes reben en el programa. Per exemple, si en un programa hi ha la declaració següent:

```
        .data 0x2000B000
aux:    .word -1
cadena: .asciiz "Hola món"
```

  la pseudoinstrucció **la $t0,aux** copiarà en **$t0** el valor **0x2000B000**, que és l'adreça de memòria principal corresponent a l'etiqueta anomenada **aux** que ve definida com un enter amb signe de valor –1 (en hexadecimal, **0xFFFFFFFF**); per una altra banda, la pseudoinstrucció **la $t1,cadena** copiarà en **$t1** el valor **0x2000B004**, que correspon a l'adreça on s'emmagatzema el codi ASCII del caràcter "**H**" de la cadena "**Hola món**".

  En definitiva, si ens hi fixem, estem en un cas semblant al de la pseudoinstrucció **li**, però en aquest cas el valor a escriure és una dada de 32 bits que ara correspon a una informació interpretada com una adreça de memòria. Consegüentment, la traducció en instruccions màquina també es farà per mitjà de **lui** i **ori**. Així, la traducció de les dues pseudoinstruccions **la** anteriors en instruccions màquina es pot fer així:

```
lui $at,0x2000       # $at = 0x20000000
ori $t0,$at,0xB000   # $t0 = 0x2000B000
ori $t1,$at,0xB004   # $t1 = 0x2000B004
```

Vegeu que les pseudoinstruccions poden fer ús del registre **$1** reservat per conveni. De fet, el pseudònim equivalent, **$at**, ve d'*assembler temporary*. En condicions normals de treball, els programes no poden fer ús explícit d'aquest registre.

Variables en la memòria principal i directives relacionades

L'assemblador del MIPS ofereix aquests recursos per a descriure les variables estàtiques d'un programa en la memòria:

- El segment **.data** on ubicar les dades en la memòria.
- La directiva **.space** permet reservar memòria del segment de dades. Útil per a declarar variables sense inicialitzar.
- Les directives **.byte**, **.half**, **.word**, **.ascii** i **.asciiz** permeten definir variables i inicialitzar-les.

Instruccions i pseudoinstruccions d'accés a la memòria de dades

El joc d'instruccions del MIPS per a lectura i escriptura de dades en la memòria comprèn vuit instruccions:

unitat	restriccions sobre l'adreça	lectura amb extensió de signe	lectura sense extensió de signe	escriptura
byte	cap	**lb**	**lbu**	**sb**
halfword	múltiple de 2	**lh**	**lhu**	**sh**
word	múltiple de 4	**lw**		**sw**

Taula 1. Les instruccions de lectura i escriptura d'enters

Totes elles són del format I i en assemblador s'escriuen de la forma **op rt,D(rs)**. **D** és un **desplaçament** de 16 bits (amb signe) que se suma al contingut del registre **base rs** per tal de formar l'adreça de la memòria on es llegeix o s'hi escriu. Aquesta manera d'especificar l'adreça permet accedir a la memòria amb diverses intencions que referim tot seguit.

L'**adreçament absolut** es fa a una paraula fixa en la memòria de la què es coneix la posició *A*: en principi només caldria considerar la constant *A* com desplaçament i el registre **$zero** com a base. Amb aquest propòsit us convé utilitzar les pseudoinstruccions de la forma **op rs,A**, que es descomponen en les instruccions de màquina adients quan *A* ocupa més de 16 bits.

Per exemple, la pseudoinstrucció **lw $rt,A** permet expressar la càrrega d'un registre amb el valor d'una variable en memòria ubicada en la posició que s'ha etiquetat com **"A"**. Aquesta línia pot traduir-se en una o més instruccions màquina, depenent del valor de l'etiqueta:

- Si **A** és un nombre expressable en 16 bits, la traducció és **lw $rt,A($0).**
- Si **A** és massa gran per això, l'assemblador la descompon en la part alta **Ah** i la part baixa **Al**. Una traducció pot ser:

```
    lui $at,Ah
    lw $rt,Al($at)
```

Adreçament indirecte, quan la posició de la variable està en un registre. És la visió del programador quan ha d'accedir a una adreça calculada pel programa, o per a seguir un punter (en parlarem més avall) o per a recórrer variables estructurades.

Adreçament relatiu a registre, quan un registre conté una adreça de referència i el programador pensa en desplaçaments respecte d'ella. Aquest adreçament també s'utilitza per a accedir a variables estructurades: el registre conté l'adreça de la variable i el desplaçament és el corresponent al camps a què s'accedeix.

Exercicis de laboratori

En iniciar el simulador, comproveu que la configuració definida en *Simulator->Settings...* és similar a la mostrada en la Figura 3, especialment les opcions de l'apartat *Execution*. Mireu també que en la secció *Display* es pot triar la manera de visualitzar el contingut dels registres del processador g(decimal o hexadecimal).

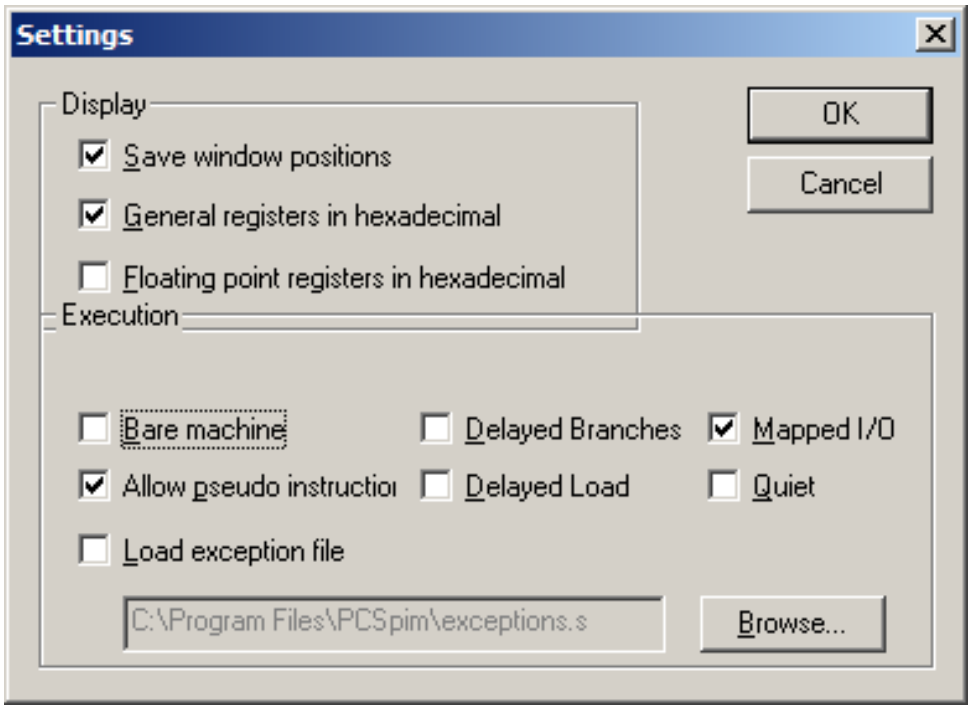

Figura 3 Paràmetres de configuració del simulador

Exercici 1: variables en registres i pseudoinstruccions

Volem calcular el perímetre d'un rectangle en funció dels seus dos costats de longituds 25 i 30 amb aquest programa:

```
        .globl __start
        .text 0x00400000
```

```
__start: li $t0,25
         li $t1,30
         add $s0,$t1,$t0
         add $s0,$s0,$s0
```

Obriu el simulador i introduïu el codi anterior. Vegeu que aquest programa fa el càlcul del perímetre mitjançant l'expressió **`$s0 = 2·($t0+$t1)`**. Noteu que els costats estan emmagatzemats en els registres **`$t0`** i **`$t1`** i el resultat queda en **`$s0`**. Pareu atenció, també, en la manera en què el programa ha estat assemblat en el simulador, això és, com s'ha dut a terme la *traducció* del codi que conforma el nostre programa en instruccions màquina.

- ¿Quantes instruccions màquina comprèn el programa?
- ¿En quines instruccions màquina es tradueixen les pseudoinstruccions presents?
- ¿En quina adreça de memòria es troba la instrucció **`add $s0,$s0,$s0`**?
- ¿Quina instrucció del programa es codifica com **`0x01288020`**?
- Digueu en hexadecimal el valor del perímetre calculat pel programa.
- Modifiqueu el programa perquè calcule el perímetre d'un rectangle amb costats 75369 i 12976 i comproveu el resultat. En aquest cas, ¿gen quines instruccions màquina es tradueixen les pseudoinstruccions presents? Raoneu-ne la resposta.

Exercici 2: variables en la memòria

Tot seguit treballareu amb un programa semblant a l'anterior, amb la diferència que els costats del rectangle i el perímetre calculat estan ubicats en la memòria principal.

```
         .globl __start
         .data 0x10000000
A:       .word 25
B:       .word 30
P:       .space 4
         .text 0x00400000
__start: la $t0,A
         la $t1,B
         la $t2,P
         lw $s0,0($t0)
         lw $s1,0($t1)
         add $s2,$s1,$s0
         add $s2,$s2,$s2
         sw $s2,0($t2)
```

Noteu que les tres variables del programa han estat representades per etiquetes (A, B, P). Aquestes etiquetes fan referència, en realitat, a posicions de memòria, i poden ser usades en el codi del programa. Cal notar que la pseudoinstrucció **`la`** (*load address*) serveix per carregar en un registre l'adreça a què fa referència una etiqueta; aquesta adreça sol rebre el nom de *punter* a l'etiqueta.

- ¿Quants bytes de la memòria principal estan ocupats per les variables del programa?
- ¿Quantes instruccions d'accés a la memòria conté el programa?
- ¿En quina adreça s'escriu el valor del perímetre?
- ¿Per què la pseudoinstrucció **`la $t0,A`** es tradueix en només una instrucció màquina i **`la $t1,B`** ho fa en dues?
- Justifiqueu el valor (4) que apareix en la directiva **`.space 4`**.
- ¿Afectaria al valor final de *P* si en compte de la directiva **`.space 4`** férem ús de la directiva **`.word 0`**?
- ¿Quin valor conté el registre **`$t1`** quan s'executa la instrucció **`lw $s1,0($t1)`**?

Exercici 3: impressió del perímetre

La interacció amb l'usuari (lectura de dades des del teclat i impressió de valors en pantalla) es fa utilitzant les anomenades crides al sistema (*system calls*). Aquestes operacions representen en realitat una interacció amb el sistema operatiu des del programa de l'usuari i es duen a terme mitjançant la combinació d'una instrucció màquina **`syscall`** i d'un conjunt de registres.

En aquest exercici anem a mostrar el mecanisme per tal d'imprimir el valor del perímetre del rectangle tan bon punt s'ha calculat. Ara no cal que pareu massa atenció als detalls (això ho farem en les sessions posteriors de laboratori), només és important adonar-se del mecanisme general de crida al sistema.

Afegiu el codi següent al final del programa de l'exercici anterior i comproveu que el valor del perímetre s'imprimeix en la pantalla:

```
        move $a0,$s2   # copia el perímetre en $a0
        li $v0,1       # codi de print_int
        syscall        # crida al sistema
```

Aquest codi està format per tres instruccions. La primera deixa el valor a imprimir en el registre **`$a0`**. La segona posa un 1 en el registre **`$v0`**; aquest valor indicarà que, d'entre totes les crides al sistema possibles, el que demanem al sistema operatiu és que imprimisca un valor enter (això vol dir que el sistema operatiu interpretarà el valor contingut en **`$a0`** com un enter amb signe codificat en complement a dos). Finalment, la tercera instrucció és la que du a terme la petició al sistema operatiu i desencadena totes les operacions adients perquè l'usuari trobe en la pantalla el valor que vol.

En definitiva, una crida al sistema sol combinar tres elements: un conjunt de paràmetres que s'hi especifiquen en **`$a0`** (i potser també en **`$a1`**), un codi o número que identifica el tipus de crida al sistema i que s'escriu en **`$v0`** (imprimir un enter, detenir l'execució del programa, llegir

una cadena de caràcters, llegir un *float*, etc.) i, finalment, la instrucció **`syscall`** que desencadena la crida al sistema.

- ¿Quina és la codificació de la instrucció màquina **`syscall`**?
- ¿En quina instrucció màquina s'ha traduït la pseudoinstrucció **`move $a0,$a1`**?
- Substituïu ara la instrucció **`sw $s2,0($t2)`** per **`sw $s2,2($t2)`**. ¿Què ocorre quan s'intenta executar el programa? Raoneu-ne la resposta.

Exercici 4: qüestions a resoldre

Algunes d'aquestes qüestions es poden resoldre amb l'ajuda del simulador.

1. ¿Quina d'aquestes instruccions és una traducció incorrecta de la pseudoinstrucció de moviment de dades **`move $t0,$t1`** (copia el contingut de **`$t1`** en **`$t0`**)?
   - **`add $t0, $t1, $zero`**
   - **`addi $t0, $t1, 0`**
   - **`sub $t0, $t1, $zero`**
   - **`and $t0, $t1, $zero`**
   - **`or $t0, $t1, $zero`**
   - **`andi $t0, $t1, 0xFFFF`**

2. Quines d'aquestes instruccions són bones traduccions de la pseudoinstrucció **`li $t0,100`** (emmagatzema el valor decimal 100 en **`$t0`**)?
   - **`ori $t0, $zero, 0x64`**
   - **`andi $t0, $zero, 0x64`**
   - **`addi $t0, $zero, 0x64`**
   - **`ori $t0, $zero, 100`**
   - **`addi $t0, 0x64, $zero`**
   - **`xori $t0, $zero, 100`**
   - **`andi $t0, $zero, 100`**
   - **`addi $t0, $zero, 100`**

3. El codi següent origina un error durant la seua execució. On i per què es produeix?

```
        li $t0, 0x10003000
        lw $t1, 2($t0)
```

4. Encara que la lletra ela ("**`l`**") de les instruccions **`lui`** i **`lw`** (i també **`lh`** i **`lb`**) vol dir *load* en anglès, quina diferència fonamental hi ha entre aquestes dues instruccions?

5. Suposeu que tenim una variable N declarada de la manera següent:

```
        .data 0x10000000
N:      .space 4
```

   Indiqueu la instrucció o instruccions màquina adients per tal d'assignar-li els valors següents:
   - **`N = 0`**
   - **`N = -1`**

- `N = 0x100000`
- `N = 0x100040`
- `N = 200000` (en decimal)

6. Com traduiríeu a instruccions màquina la pseudoinstrucció `li $t0,-1`?

7. Suposeu que el segment de dades d'un programa d'assemblador MIPS conté aquesta descripció:

```
        .data 0x10000000
X:      .space 4
```

En cadascun dels casos següents, indiqueu quin valor escriuen les instruccions sobre la posició de memòria `X`:

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

```
la $t0,X
sh $zero,0($t0)
sh $zero,2($t0)
```

```
la $t0,X
sb $zero,0($t0)
sb $zero,1($t0)
sb $zero,2($t0)
sb $zero,3($t0)
```

```
la $t0, X
lui $t1, 0x0001
sw $t1,0($t0)
```

```
la $t0,X
lui $t1,0xFFFF
ori $t1,$t1,0xFFFF
sw $t1,0($t0)
```

```
lui $t0,0x1000
andi $t1,$t1,0x0000
sw $t1,0($t0)
```

```
la $t0,X
```

```
lw $t1,0($t0)
sw $t1,0($t0)
```

```
li $t1,50
sw $t1,X
```

```
li $t0,0x50
sw $t0,X
```

```
li $t0,0x10000000
li $t1,0xFFFFFFFF
sw $t1,0($t0)
```