ESTRUCTURA DE COMPUTADORES
Grado en Ingeniería Informática

Sesión de laboratorio número 1

REGISTROS Y MEMORIA PRINCIPAL

Objetivos

- Entrar de nuevo en contacto con el simulador SPIM.
- Presentar los aspectos básicos de la arquitectura MIPS: banco de registros, ALU, memoria principal.
- Repasar las bases de la programación en ensamblador de la arquitectura MIPS.
- Retomar contacto con la manipulación de registros, pseudoinstrucciones e instrucciones máquina, direcciones de memoria principal y datos enteros en el ensamblador del MIPS.
- Comprobar el funcionamiento del mecanismo de llamada al sistema mediante la impresión de un número entero.

Bibliografía

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

Introducción teórica

El banco de registros de propósito general

El procesador MIPS tiene otros bancos de registros, pero no vamos a estudiarlos ahora mismo. El banco de enteros, o de registros de propósito general, permite hacer cálculos aritméticos sobre direcciones y sobre datos de tipo entero.

A excepción de **`$0`** y **`$31`**, todos los registros son idénticos y sirven para las mismas cosas. El registro **`$0`** contiene siempre el **`valor`** cero y el registro **`$31`** está ligado a la instrucción **`jal`**. Sin embargo, para facilitar la compilación habitual de los programas escritos en alto nivel, se ha hecho un reparto **convencional** de los registros que se detallará en esta serie de prácticas. El uso del convenio se hace más sencillo con la nueva denominación de los registros, reconocida por el ensamblador del simulador PCSpim. Los detalles más útiles del convenio de uso de los registros se explicarán a lo largo de estas y otras prácticas cuando sea necesario.

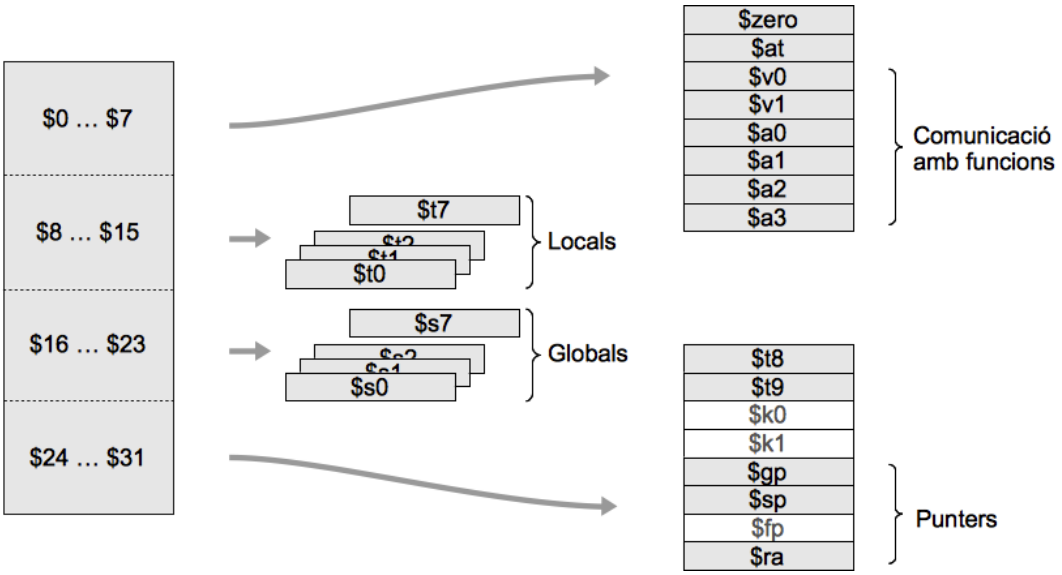

Figura 1. Convención de uso de los registros. Podemos formar cuatro grandes bloques de 8 registros: comunicación con las funciones ($0 a $7), variables locales ($8 a $15), variables globales ($16 a $23) y diversos ($24 a $31). Entre los primeros están los dos registros $zero y $at; entre los últimos hemos marcado los registros que se utilizarán en estas prácticas.

Instrucciones y pseudoinstrucciones

A cada ciclo de instrucción, el procesador ejecuta una instrucción máquina. Muchas instrucciones hacen por sí mismas una operación elemental con utilidad bien definida, expresada en su nemotécnico (una operación aritmética, un acceso de lectura o escritura en la memoria, un salto); pero hay casos en que una acción elemental se obtiene haciendo un uso muy particular de una instrucción máquina de propósito muy general y otros en que la acción necesita dos o tres instrucciones máquina por limitaciones del juego de instrucciones del procesador.

Las pseudoinstrucciones permiten expresar estas acciones elementales en una línea con nemotécnico y operandos que siguen la sintaxis común reconocida por el ensamblador pero que no representan una instrucción nueva. El resultado es mucho más legible y se ajusta mejor al universo mental del programador en ensamblador. Veamos un ejemplo de cada caso:

- La pseudoinstrucción **`move rs,rt`** expresa la operación de copia entre registros (*rs* = *rt*). Esta operación se puede obtener como un caso particular de una instrucción más general, como **`or rs,rt,$zero`** o **`addi rs,rt,0`**.
- La pseudoinstrucción **`li`** (*load immediate*) resuelve un problema de programación básico: asignar una constante **`K`** a un registro. El programador en ensamblador sólo debe escribir una línea: **`li $rt,K`**. Dependiendo del valor de **`K`**, tendremos los tres

casos que muestra la tabla siguiente. Debe entenderse que **K** puede tener hasta 32 bits, que **Kh** representa los 16 bits más significativos de **K** y **Kl** los 16 bits menos significativos:

Caso	Ejemplo	Traducción
Kh=0	**K = 0x00000100** **Kl = 0x0100**	`ori $rt,$0,Kl`
Kl=0	**K = 0x00040000** **Kh = 0x0004**	`lui $rt,Kh`
Kh≠0 y Kl≠0	**K = 0x00040010** **Kl = 0x0010** **Kh = 0x0004**	`lui $at,Kh` `ori $rs,$at,Kl`

Figura 2. Diversas traducciones de **li $rt,K**, donde la constante **K** de 32 bits se descompone en la parte alta **Kh** y la parte baja **Kl**, ambas de 16 bits.

- La pseudoinstrucción **la** (*load address*) también resuelve otro problema de programación básico y, a su vez, importantísimo: copia la dirección de memoria de una etiqueta en un registro. Esta funcionalidad facilita enormemente la programación en ensamblador porque permite al programador olvidarse de las direcciones de memoria exactas donde se ubican las variables y centrarse solamente en el nombre que éstas reciben en el programa. Por ejemplo, si en un programa está la declaración siguiente:

```
        .data 0x2000B000
aux:    .word -1
cadena: .asciiz “Hola mundo”
```

la pseudoinstrucción **la $t0,aux** copiará en **$t0** el valor **0x2000B000**, que es la dirección de memoria principal correspondiente a la etiqueta llamada **aux** que viene definida como un entero con signo de valor -1 (en hexadecimal, **0xFFFFFFFF**); por otra parte, la pseudoinstrucció **la $t1,cadena** copiará en **$t1** el valor **0x2000B004**, que corresponde a la dirección donde se almacena el código ASCII del carácter "**H**" de la cadena "**Hola mundo**".

En definitiva, si nos fijamos, estamos en un caso similar al de la pseudoinstrucció **li**, pero en este caso el valor a escribir es un dato de 32 bits que ahora corresponde a una información interpretada como una dirección de memoria. Por consiguiente, la traducción en instrucciones máquina también se hará por medio de **lui** y **ori**. Así, la

traducción de las dos pseudoinstruccions **la** anteriores en instrucciones máquina se puede hacer así:

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

Obsérvese que las pseudoinstrucciones pueden utilizar el registro **$1** reservado por convenio. De hecho, el pseudónimo equivalente, **$at**, viene de *assembler temporary*. En condiciones normales de trabajo, los programas no pueden usar explícitamente este registro.

Variables en la memoria principal y directivas relacionadas

El ensamblador del MIPS ofrece estos recursos para describir las variables estáticas de un programa en la memoria:

- El segmento **.data** donde ubicar los datos en la memoria.
- La directiva **.space** permite reservar memoria del segmento de datos. Útil para declarar variables sin inicializar.
- Las directivas **.byte**, **.half**, **.word**, **.ascii** y **.asciiz** permiten definir variables y inicializarlas.

Instrucciones y pseudoinstrucciones de acceso a la memoria de datos

El juego de instrucciones del MIPS para lectura y escritura de datos en la memoria comprende ocho instrucciones:

unidad	restricciones sobre la dirección	lectura con extensión de signo	lectura sin extensión de signo	Escritura
byte	ninguna	**lb**	**lbu**	**sb**
halfword	múltiplo de 2	**lh**	**lhu**	**sh**
word	múltiplo de 4	**lw**		**sw**

Tabla 1. Las instrucciones de lectura y escritura de enteros

Todas ellas son del formato I y en ensamblador se escriben de la forma **op rt,D(rs)**. **D** es un **desplazamiento** de 16 bits (con signo) que se suma al contenido del registro **base rs** para formar la dirección de memoria donde se lee o escribe. Esta manera de especificar la dirección permite acceder a memoria con diversas intenciones que referimos a continuación.

El **direccionamiento absoluto** se realiza a una palabra fija en la memoria de la que se conoce la posición *A*: en principio sólo sería necesario considerar la constante *A* como desplazamiento y el registro **$zero** como base. Con este propósito conviene utilizar las

pseudoinstrucciones de la forma **`op rs,A`**, que se descomponen en las instrucciones máquina correspondientes cuando *A* ocupa más de 16 bits.

Por ejemplo, la pseudoinstrucción **`lw $rt,A`** permite expresar la carga de un registro con el valor de una variable en memoria ubicada en la posición que se ha etiquetado como **`"A"`**. Esta línea puede traducirse en una o más instrucciones máquina, dependiendo del valor de la etiqueta:

- Si **`A`** es un número expresable con 16 bits, la traducción es **`lw $rt,A($0).`**
- Si **`A`** es demasiado grande para eso, el ensamblador la descompone en la parte alta **`Ah`** y la parte baja **`Al`**. Una traducción puede ser:

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

Direccionamiento indirecto, cuando la posición de la variable está en un registro. Es la visión del programador cuando debe acceder a una dirección calculada por el programa, o para seguir un puntero (hablaremos de eso más abajo) o para recorrer variables estructuradas.

Direccionamiento relativo a registro, cuando un registro contiene una dirección de referencia y el programador piensa en desplazamientos respecto de ella. Este direccionamiento también se utiliza para acceder a variables estructuradas: el registro contiene la dirección de la variable y el desplazamiento es el correspondiente al campo al que se accede.

Ejercicios de laboratorio

Al iniciar el simulador, comprueba que la configuración definida en *Simulator->Settings...* es similar a la mostrada en Figura 3, especialmente las opciones del apartado *Execution*. Nótese también que en la sección *Display* se puede escoger la manera de visualizar el contenido de los registros del procesador (decimal o hexadecimal).

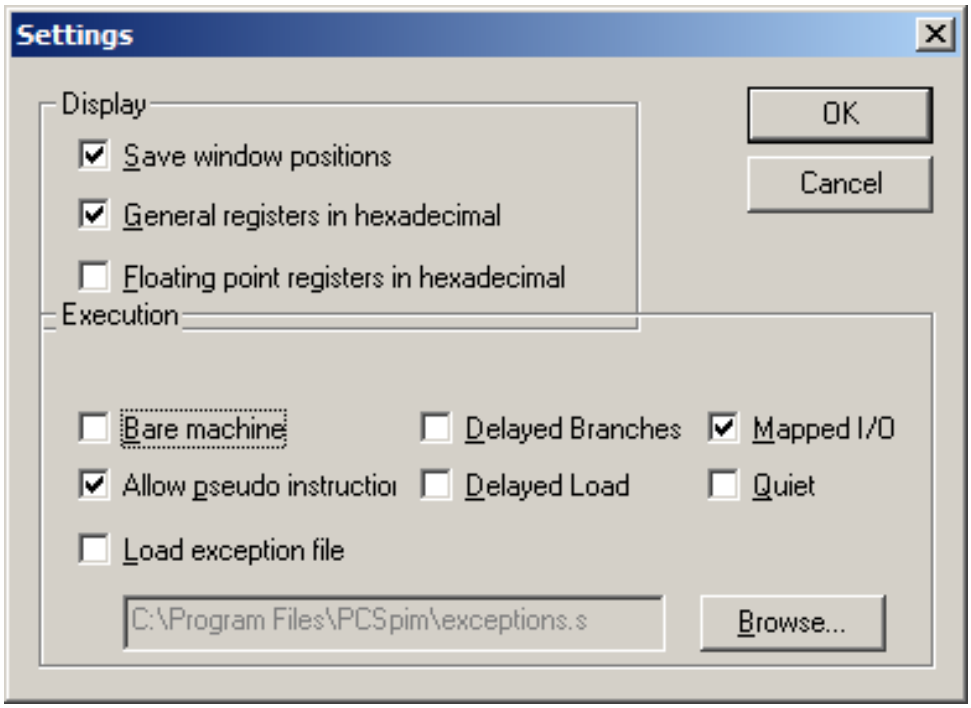

Figura 3 Parámetros de configuración del simulador

Ejercicio 1: variables en registros y pseudoinstrucciones

Queremos calcular el perímetro de un rectángulo en función de sus dos lados de longitudes 25 y 30 con este programa:

```
        .globl __start
        .text 0x00400000
__start: li $t0,25
        li $t1,30
        add $s0,$t1,$t0
        add $s0,$s0,$s0
```

Abra con el simulador e introduzca el código anterior. Véase que este programa hace el cálculo del perímetro mediante la expresión **`$s0 = 2·($t0+$t1)`**. Nótese que los lados están almacenados en los registros **`$t0`** y **`$t1`** y el resultado queda en **`$s0`**. Prestad atención, también, en la manera en que el programa ha sido ensamblado en el simulador, esto es, como se ha llevado a cabo la *traducción* del código que conforma nuestro programa en instrucciones máquina.

- ¿Cuántas instrucciones máquina comprende el programa? 4
- ¿En qué instrucciones máquina se traducen las pseudoinstruccions presentes? ori, ori
- ¿En qué dirección de memoria se encuentra la instrucción **`add $s0,$s0,$s0`**? 0x0040000c
- ¿Qué instrucción del programa se codifica como **`0x01288020`**? add $16, $9, $8
- Diga en hexadecimal el valor del perímetro calculado por el programa. 0000006e

- Modifique el programa para que calcule el perímetro de un rectángulo con lados 75369 y 12976 y compruebe el resultado. En este caso, ¿en qué instrucciones máquina se traducen las pseudoinstruccions presentes? Razone su respuesta.

  La asignación del primer número se traduce en Lui Ori pues es requerido modificar los 32 bits para settear el número, la segunda asignación se sirve solamente de una ori

Ejercicio 2: variables en la memoria

A continuación trabajarás con un programa similar al anterior, con la diferencia de que los lados del rectángulo y el perímetro calculado están ubicados en la memoria 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)
```

Nótese que las tres variables del programa han sido representadas por etiquetas (A, B, P). Estas etiquetas se refieren, en realidad, a posiciones de memoria, y pueden ser usadas en el código del programa. Nótese que la pseudoinstrucción **`la`** (*load address*) sirve para cargar en un registro la dirección a la que hace referencia una etiqueta; esta dirección suele recibir el nombre de puntero a la etiqueta.

- ¿Cuántos bytes de la memoria principal están ocupados por las variables del programa? 4, 4, 4
- ¿Cuántas instrucciones de acceso a la memoria contiene el programa? 3, lw, lw, sw
- ¿En qué dirección se escribe el valor del perímetro? 0x1000008
- ¿Por qué la pseudoinstrucció **`la $t0,A`** se traduce en sólo una instrucción máquina y **`la $t1,B`** lo hace en dos? la dirección de A es 0x1000000, para cargar la solo es requerido modificar los bits más significativos, es decir, una lui(load upper inmediate) mientras que para cargar la dirreción de B hay que cambiar tanto los bits más significativos como los menos por lo que se traduce tanto en lui como en ori
- Justificar el valor (4) que aparece en la directiva **`.space 4`**? Como el valor del perímetro es desconocido se guardan 4 bytes, 32 bits (una word) en lugar de asignar un valor aleatorio, se guarda un espacio del tamaño de una word
- Afectaría al valor final de P si en lugar de la directiva **`.space 4`** hiciésemos uso de la directiva **`.word 0`**? No afectaría porque carga la misma cantidad de bytes pero 0 es un valor falso,
- ¿Qué valor contiene el registro **`$t1`** cuando se ejecuta la instrucción **`lw $s1,0($t1)`**? 0x1000004

Ejercicio 3: impresión del perímetro

La interacción con el usuario (lectura de datos desde el teclado e impresión de valores en pantalla) se hace utilizando las denominadas llamadas al sistema (*system calls*). Estas operaciones representan en realidad una interacción con el sistema operativo desde el

programa del usuario y se llevan a cabo mediante la combinación de una instrucción máquina **`syscall`** y de un conjunto de registros.

En este ejercicio vamos a mostrar el mecanismo para imprimir el valor del perímetro del rectángulo después de haberlo calculado. Ahora no hay que poner demasiada atención en los detalles (eso lo haremos en las sesiones posteriores de laboratorio), solo es importante darse cuenta del mecanismo general de llamada al sistema.

Agregue el código siguiente al final del programa del ejercicio anterior y compruebe que el valor del perímetro se imprime en la pantalla:

```
        move $a0,$s2   # copia el perímetro en $a0
        li $v0,1       # código de print_int
        syscall        # llamada al sistema
```

Este código está formado por tres instrucciones. La primera deja el valor a imprimir en el registro **`$a0`**. La segunda pone un 1 en el registro **`$v0`**; este valor indicará que, de entre todas las llamadas al sistema posibles, lo que pedimos al sistema operativo es que imprima un valor entero (esto quiere decir que el sistema operativo interpretará el valor contenido en **`$a0`** como un entero con signo codificado en complemento a dos). Finalmente, la tercera instrucción es la que lleva a cabo la petición al sistema operativo y desencadena todas las operaciones adecuadas para que el usuario encuentre en la pantalla el valor que quiere.

En definitiva, una llamada al sistema suele combinar tres elementos: un conjunto de parámetros que se especifican en **`$ a0`** (y quizás también en **`$ a1`**), un código o número que identifica el tipo de llamada al sistema y que se escribe en $ v0 (imprimir un entero, detener la ejecución del programa, leer una cadena de caracteres, leer un *float*, etc) y, finalmente, la instrucción **`syscall`** que desencadena la llamada al sistema.

- ¿Cuál es la codificación de la instrucción máquina **`syscall`**? 0x0000000c syscall
- ¿En qué instrucción máquina se ha traducido la pseudoinstrucción **`move $a0,$a1`**? addu - add unsigned
- Substituya ahora la instrucción **`sw $s2,0($t2)`** por **`sw $s2,2($t2)`**. ¿Qué ocurre cuando se intenta ejecutar el programa? Razone la respuesta.

  Ocurre la excepción unaligned access ya que la dirección al desplazarla 2 bytes quedaría en la dirección no múltiplo de 4 por lo que no puedes cargar una word en ella, si se diese el caso de que fuera múltiplo de 4 lo guardaría en una dirección que tampoco podrías usar porque no está tenido en cuenta en el programa, lo cargarías donde no se ha guardado espacio dentro de los datos

Ejercicio 4: cuestiones a resolver

Algunas de estas cuestiones se pueden resolver con la ayuda del simulador.

1. ¿Cuál de estas instrucciones es una traducción incorrecta de la pseudoinstrucción de movimiento de datos **`move $t0,$t1`** (copia el contenido 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. ¿Cuáles de las siguientes instrucciones son buenas traducciones de la pseudoinstrucción `li $t0,100` (almacena el valor decimal 100 en `$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 código siguiente origina un error durante su ejecución. ¿Dónde y por qué se produce?

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

4. Aunque la letra ele ("l") de las instrucciones `lui` y `lw` (y también `lh` y `lb`) significa *load* en inglés, ¿qué diferencia fundamental hay entre estas dos instrucciones?

5. Suponga que tenemos una variable N declarada de la siguiente manera:

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

   Indica la instrucción o instrucciones máquina adecuadas para asignarle los siguientes valores:
   - `N = 0`
   - `N = -1`
   - `N = 0x100000`
   - `N = 0x100040`
   - `N = 200000` (en decimal)

6. Como traducirías a instrucciones máquina la pseudoinstrucción `li $t0,-1`?
7. Supón que el segmento de datos de un programa en ensamblador MIPS contiene la siguiente descripción:

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

   En cada uno de los siguientes casos, indica qué valor escriben las instrucciones sobre la posición de memoria `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)
```