

Práctica 1

Instrucciones y registros

Objetivos

- Toma de contacto con aspectos básicos de la arquitectura del MIPS
- Toma de contacto con un lenguaje ensamblador
- Toma de contacto con el simulador MARS
- Escribir el primer programa en ensamblador

Materiales

Simulador MARS y código Fuente de partida

Teoría

Introducimos a continuación una serie de conceptos que nos ayudarán a comprender mejor la práctica.

La máquina von Neumann

La máquina de Von Neumann es el diseño teórico de un computador de programa almacenado realizado en 1945 por el físico y matemático John von Neumann (1903-1957). Este diseño es la base de la práctica totalidad de los computadores de hoy en día. El concepto básico que se encuentra tras este diseño es la posibilidad de almacenar instrucciones de programas y los datos en memoria y la habilidad de que esas instrucciones operen con los datos.

Una máquina de Von Neumann está formada por tres componentes distintos (o subsistemas): Una unidad central de procesamiento (CPU), una memoria y interfaces de entrada y salida.

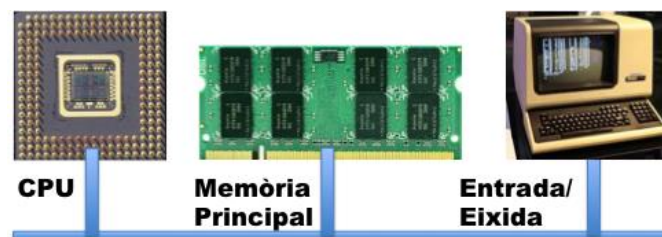


Figura 1: Componentes de una arquitectura Von Neumann



La Unidad Central de Procesamiento se dedica al procesamiento de la información y está formada por tres Componentes principales, la unidad aritmética y lógica, la unidad de control y un conjunto de registros.

La memoria principal se utiliza para almacenar las instrucciones de los programas y los datos.

La entrada y salida permiten a la memoria del computador recibir información y enviar datos a dispositivos externos. También permite al computador comunicarse con el usuario y con dispositivos externos de almacenamiento.

Los tres componentes citados se comunican a través del bus del sistema.

Programa almacenado

Un concepto inherente a la introducción de la máquina de Von Neumann es el de programa almacenado, que significa que tanto las instrucciones del programa como los datos sobre los que operará están almacenados en memoria. Esto quiere decir, por ejemplo, que pueden ser modificados independientemente de la máquina a diferencia de los computadores anteriores que tenían que programarse mediante conmutadores y cables de interconexión.

Programa de alto nivel versus código máquina

Cuando escribimos un programa en un lenguaje de alto nivel (por ejemplo C o Java), el programa no es más que una cadena de caracteres ASCII que hemos introducido a través de teclado y hemos almacenado en un fichero.

Para que podamos ejecutar ese fichero necesitamos traducirlo a un conjunto de instrucciones ejecutables (*código máquina*, formado por ceros y unos) que indiquen a un computador determinado que tiene que hacer. El programa tiene que traducirse en instrucciones que pueda entender el procesador particular que contiene el computador, bien sea, por ejemplo, de AMD, de Intel, etc. Esa traducción la realiza el *compilador*.

Lenguaje ensamblador

En los primeros computadores construidos en los años 40 del siglo pasado los programadores escribían los programas directamente en código máquina, asignando directamente los valores de 0 y 1 mediante la activación de interruptores físicos. Para facilitar esta tarea se introdujo el lenguaje ensamblador. Un programa en lenguaje ensamblador es un fichero ASCII que necesita ser traducido a código máquina, pero esta traducción es relativamente fácil y directa. La traducción a lenguaje máquina la realiza un programa llamado *Ensamblador*. No estudiaremos como lo hace exactamente el Ensamblador pero sí estudiaremos como programar en un lenguaje ensamblador y entenderemos la correspondencia exacta del programa inteligible por el procesador. En la figura podéis observar el proceso de traducción de un compilador y de un ensamblador.

Programa
en lenguaje
de alto
nivel
(en C)

```
swap(int v[], int k)
{int temp;
 temp = v[k];
 v[k] = v[k+1];
 v[k+1] = temp;
}
```



Programa
En lenguaje
Ensamblador
(para MIPS)

```
swap:
    muli $2, $5, 4
    add $2, $4, $2
    lw $15, 0($2)
    lw $16, 4($2)
    sw $16, 0($2)
    sw $15, 4($2)
    jr $31
```



Programa
En lenguaje
Máquina
(para MIPS)

```
000000001010000100000000000011000
000000001000111000011000000100001
100011000110001000000000000000000
100011001111001000000000000000100
101011001111001000000000000000000
101011000110001000000000000000100
00000011111000000000000000001000
```

Figura 2: Proceso de compilación y ensamblado

Ciclo de instrucciones.

Todo procesador requiere de un ciclo de instrucción para ejecutar una instrucción en particular. El ciclo de instrucción constará de una serie de pasos que permiten al procesador leer la instrucción almacenada en la memoria y realizar las acciones oportunas sobre los operandos requeridos por la instrucción.

MIPS

En este curso vamos a fijarnos en una máquina particular, el procesador MIPS (*Microprocessor without Interlocked Pipeline Stages*), y a pesar de que hay varias versiones (32 bits (R2000 y R3000), 64 bits (R4000)) vamos a centrarnos en el MIPS R2000 y aprenderemos a programar en su lenguaje ensamblador llamado también MIPS.

Antes de nada tenemos que introducir el modelo de la máquina que tiene que conocer el programador de ensamblador de esta máquina. Los elementos del procesador que son visibles al programador son:

- Un banco de registros de enteros o de registros de propósito general que está formado por 32 registros de 32 bits. Eso significa que necesitamos 5 bits para seleccionar cada registro. A pesar de que MIPS tiene otros bancos de registros, los dejaremos de lado y los estudiaremos más adelante.
- Una memoria principal formada por palabras de 32 bits. La memoria se puede direccionar per byte, media palabra o palabra completa. Para seleccionar una palabra se necesitan direcciones de 32 bits. En posteriores prácticas estudiaremos como hacer el acceso a memoria.
- Una unidad aritmética y lógica que permite realizar operaciones aritméticas y lógicas sobre los datos de entrada.
- Un conjunto de funciones de entrada/salida del sistema. Es un software básico que viene de fábrica.
- El registro PC (*Program Counter*) que siempre contiene la dirección en memoria donde está la instrucción a ejecutar. También hay otros registros especiales que iremos introduciendo en el momento que sean necesarios.

En la figura siguiente se muestra el modelo del procesador MIPS simplificado:

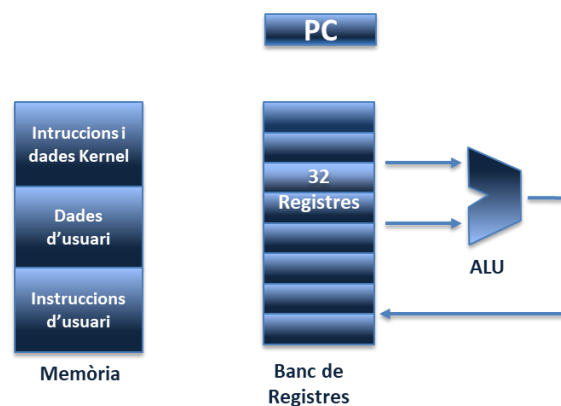


Figura 3. Modelo del procesador MIPS

Modelo del simulador MARS

El MARS es un simulador de programación que permite ejecutar programas escritos en ensamblador MIPS R2000/R3000. Se trata de un entorno de desarrollo interactivo que permite la introducción del código en ensamblador. Con este simulador podemos hacer el seguimiento de la ejecución de los programas y ver en todo momento el contenido de los registros del procesador.

Es un programa libre implementado en Java y se puede descargar desde la página siguiente: <http://courses.missouristate.edu/kenvollmar/mars/index.htm>

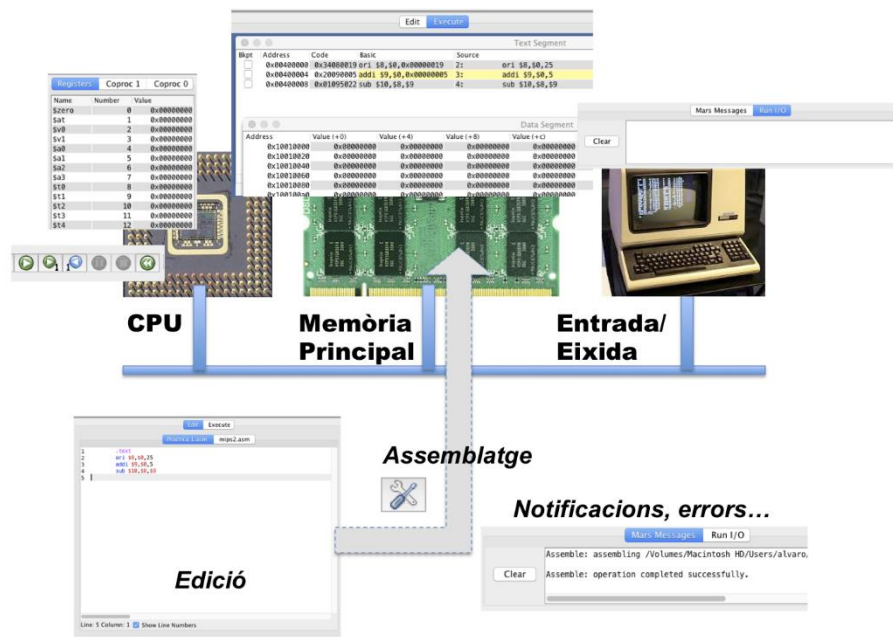


Figura 4. Visión general del procesador.

El vocabulario general que se usará al programar en ensamblador se recoge a continuación:

- *Leer* (un registro, una palabra) es seleccionar un registro o una palabra, tomar su contenido y llevarlo a otro lugar: registro, palabra u operador. Leer nunca cambia el contenido.
- *Escribir* (un registro, una palabra) es seleccionar un registro o una palabra y cambiar su contenido por un nuevo valor proveniente de otro registro, palabra o operador.
- *Operador*: componente del procesador que hace un cálculo. A partir de una o dos palabras el operador suministra el resultado del cálculo. Ejemplo: la suma.
- *Operación*: acción que hace el procesador. Puede implicar uno o más operadores. Las operaciones más habituales tienen un significado aritmético.
- *Instrucción*: En el caso del MIPS, es una palabra de 32 bits que codifica una operación y los operandos correspondientes. Veremos que hay distintas formas de especificar operandos.

n	Contingut	adreça	Contingut	\$v0	Nom
0	0x00000000	0x00000000	0x00000000	1	Print Integer
1	0xFFFFFFFF	0x00000004	0x00000000	2	Print float
2		0x00000008	0x00000000		
3					
31		0xFFFFFFFF	0x00000000		

Figura 5. Elementos del procesador.

Realización de la práctica.

La pantalla principal del MARS se dividió en diferentes partes.

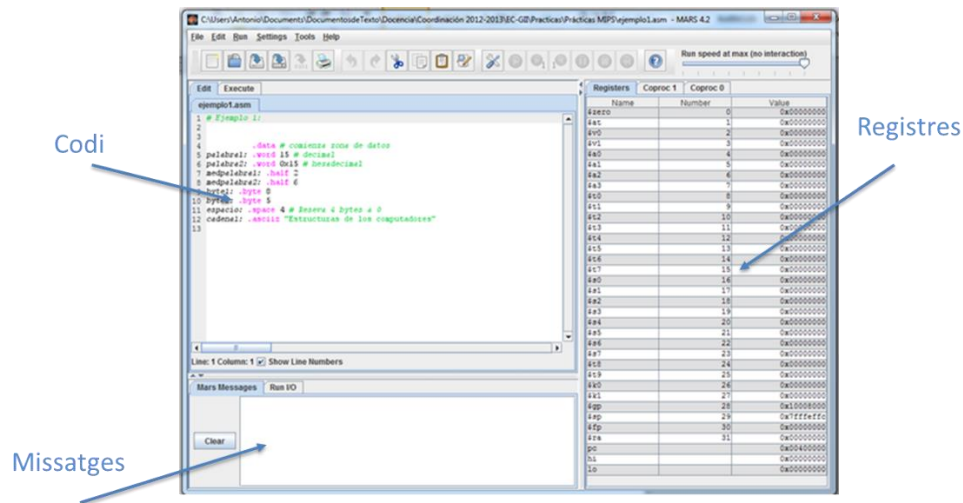


Figura 6. Pantalla principal del simulador MARS.

1. La ventana *Registers*

Observad que los registros \$0 a \$31 tienen dos nombres y que, quitado del \$sp (*Stack pointer*), \$gp (*Pointer to global area*) y \$pc (*Program counter*) valen 0.

Podéis escoger la base de numeración activando o desactivando la casilla correspondiente que la podréis encontrar en (*Settings>Values Displayed in Hexadecimal*) o cambiando a la pestaña *Execute* (*Hexadecimal Values*).

Actividad 1

- Probad a modificar el contenido de algún registro. Notad que podéis escribir en decimal o hexadecimal y que no podéis cambiar \$0, \$31 ni \$pc.
- Probad a escribir valores negativos en los registros.

Cuestión 1.

- ¿Cuál es el mayor positivo que puede contener un registro del MIPS? Basta con que lo digas en hexadecimal.
- ¿Cuál es el mayor negativo que puede contener un registro del MIPS? Basta con que lo digas en hexadecimal.

2. El primer programa – análisis

Partiremos del siguiente programa:

```
#####
#                                     #
#   Primer programa                   #
#                                     #
#####

.text 0x00400000
addi $9,$8,25
addi $10,$8,5
```

El símbolo # marca el inicio de un comentario. El ensamblador ignorará lo que haya a la derecha de este símbolo.

La línea `.text 0x00400000` dice en que dirección de la memoria comienza el programa. `0x00400000` es la dirección por defecto, si no introducimos nada el ensamblador asume este valor.

La instrucción *addi* (*suma inmediata*) se escribe en lenguaje ensamblador de la forma

addi rt, rs, k

donde *rt* y *rs* pueden ser cualquier número de registro del 0 al 31 y *K* cualquier nombre codificado en complemento a 2 con un tamaño de 16 bits. La instrucción hace $rt = rs + K$, es decir, lee un registro fuente (*rs*), hace la suma del su contenido y la constante *K* y escribe el resultado de la suma en el registro destino (*rt*).

Gráficamente podemos expresarlo como:



Figura 7. Representación gráfica de la instrucción *addi*.

Encontramos dos maneras de expresar los operandos: el número de un registro como pueda ser \$8 o \$9, y constantes como 25 o 5. Por lo tanto, el programa hace la suma del contenido de \$8 y la constante 25 y almacena el resultado en el registro \$9. Después hace la suma del contenido de \$8 y 5 y el resultado lo guarda en \$10.

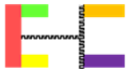
Las instrucciones se codifican en binario para almacenarlas en la memoria. Todas las instrucciones del MIPS se codifican en 32 bits. La instrucción *addi \$9, \$8, 25* se codifica de la siguiente forma:

Codigo op (6 bits)	Rs (5 bits)	Rt (5 bits)	K
0010 00	01 000	0 1001	0000 0000 0001 1001

En hexadecimal quedaría como: `0x21090019`

El primer campo es el código de operación de 6 bits e indica que se hará la suma del registro fuente *rs* y el número *K*.

Los siguientes dos campos son los numero de los registros fuente y destino, en el ejemplo el 8 y el 9. El tercer campo es el valor en complemento a dos de la constante *K*, en el ejemplo el valor 25.



Cuestión 2.

- ¿Cómo se codifica la instrucción *addi \$10,\$8,5*? Escribid el código resultante en hexadecimal

3. Ensamblado

Escribid, denominad y guardad el código fuente en un archivo de texto. Ensamblad (*Run->Asemeje*) y atended a la ventana *Mars Messages*. Observad si dice *Assemble: operation completed successfully*.

Al ensamblar aparecen dos ventanas llamadas *Text segment* y *Data Segment*. De momento nos fijaremos únicamente en la de *Text segment*.

4. La ventana *text segment*



Si nos fijamos en esta ventana vemos que la información está tabulada, donde podemos ver la dirección en hexadecimal donde se encuentra la instrucción en memoria (*columna Address*) o la instrucción ensamblada en código máquina (*columna Code*). Además nos señalará realzado en amarillo que es la instrucción que va a ejecutarse.

Actividad 2

- Observa la ventana *Text Segment*. ¿En qué dirección se almacena cada instrucción del programa?
- Comprobad la codificación de las instrucciones en código máquina.
- Observa la ventana *Registers*. ¿Qué vale el PC?

5. El ciclo de instrucción

Haced los pasos que se indican a continuación:

1. Escribid un valor en el registro \$8 en la ventana *Registers*
2. Buscan la acción *Step* (*Run->Step*, F7, ) y hacedla una vez. ¡Habéis simulado un ciclo de instrucción! Notad que el PC ha avanzado y que el contenido del registro \$9 ha cambiado.
3. Completad los dos de instrucción y confirmad el resultado.
4. Buscad la acción *Reset* (*Run->Reset*, F12, ) y dad un valor inicial a \$8. Ejecutad el programa entero (*Run->Go*, F5)

Actividad 3

- Dad el siguiente valor inicial a \$8 = 0x7FFFFFFF y ejecutad de nuevo el programa paso a paso fijándoos en la ventana *Mars Messages*. ¿Qué ha ocurrido?

Cuestión 3

- ¿Cuál es el valor más grande que podrá contener \$8 para que no se aborte el programa?

6. Usos alternativos de addi

Podemos utilizar la instrucción `addi` para usos distintos al de la propia suma como puede ser almacenar una constante K en un registro R : `addi $R, $0, K`. O para copiar el contenido de un registro R a otro registro T : `addi $T, $R, 0`

Actividad 4

- Modificad el programa para dar un valor inicial al registro $\$8$ utilizando `addi`.
- Añadid una instrucción para que el resultado final se encuentre en $\$12$
- ¿Se podría utilizar la instrucción `addi` para hacer una resta?

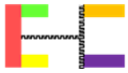
Cuestión 4

- ¿Cómo se escribe la instrucción que hace $\$8 = \$8 - 1$?
- ¿Cómo quedaría su codificación en binario?

7. Ayudas a la programación. Nombres alternativos de los registros.

El banco de registros de propósito general del MIPS está formado por 32 registros. Con estos registros se pueden realizar cálculos con direcciones y con números enteros. Recordad que el MIPS también tiene otros bancos pero ahora mismo no los estudiaremos, lo dejaremos para más adelante. Ciertos registros del banco se utilizan para propósitos muy específicos facilitando de esta forma la compilación habitual de programas de alto nivel. Este convenio está aceptado por los programadores en MIPS, es por esto que nos resultará muy conveniente seguir estas reglas o convenios de utilización de los registros. Cada registro del banco tiene un nombre convencional reconocido por el ensamblador y será el que de ahora en adelante utilizaremos. En la siguiente tabla los podéis ver:

Número del registro	Nombre convencional	Utilización
\$0	\$zero	Siempre contiene el valor 0.
\$1	\$at	Reservado para el ensamblador
\$2 - \$3	\$v0 - \$v1	Utilizado para resultados y evaluación de expresiones
\$4 - \$7	\$a0-\$a3	Utilizado para pasar argumentos a rutinas.
\$8 - \$15	\$t0-\$t7	Temporales usados para la evaluación de expresiones (no se mantienen a través de llamadas a procedimientos)
\$16 - \$23	\$s0-\$s7	Registros guardados. (Se mantienen a través de llamadas a procedimientos)
\$24 - \$25	\$t8-\$t9	Más temporales (no se mantienen a través de llamadas a procedimientos)
\$26 - \$27	\$k0 - \$k1	Reservados para al núcleo del sistema operativo.
\$28	\$gp	Contiene el puntero global.
\$29	\$sp	Contiene el puntero de pila (stack pointer)



\$30	\$fp	Contiene el puntero de marco.
\$31	\$ra	Contiene la dirección de retorno usada en las llamadas a procedimientos.

Tabla 1. Número de registros, nombre y utilización

En adelante, para el cálculo de cualquier expresión, utilizad los registros temporales \$t0...\$t7

Cuestión 5

- Reescribid el programa anterior utilizando el convenio de registros y vuelve a ejecutarlo.

8. Más sobre la suma inmediata.

En el apartado 5 habéis visto que la ejecución se aborta cuando se produce desbordamiento (overflow) al hacer la suma y mostrar en la consola el mensaje *Error: Runtime exception at 0x0----: arithmetic overflow*. En ciertas ocasiones puede interesarle al programador que la ejecución del programa continúe y no aborte. En ese caso es responsabilidad de programador tomar las acciones necesarias si fuera necesario. El MIPS nos proporciona una instrucción que lo permite, es la instrucción *addiu rt, rs, K*.

Actividad 5

- Volved a escribir el programa cambiando *addi* or *addiu* y dad como valor inicial de \$t0 el positivo más grande posible (\$t0 = 0x7FFFFFFF) y ejecutadlo observando el contenido de \$t1 en hexadecimal y en decimal. ¿Qué ha ocurrido?
- Si el programador considera que está operando con número naturales, el resultado que hay en \$t1 sería correcto? ¿Cuál sería su valor en decimal?

Cuestión 6

- Escribe el código que haga las siguientes acciones utilizando el convenio de registros y utilizando la instrucción *addi*:

\$t2=5

\$t0= 8

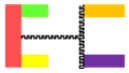
\$t3=\$t2 + 10

\$t0=\$t0 - 4

\$t4=\$t3 - 30

\$t5=\$t0

- Ensamblad y ejecutad el programa y comprobad que el resultado final es \$t7 = \$t2 = 4, \$t6=-15, \$t4=5,\$t5=15.



Cuestión 7

- ¿Se podría escribir el mismo código utilizando la instrucción *addiu*? Haz la prueba.
- ¿Cuál es el código de operación de la instrucción *addiu*?
- Codifica en binario la instrucción *addiu \$v0, \$zero, 1*.

Resumen

- Las instrucciones se codifican en binario, el ensamblador es el encargado de codificarlas.
- La memoria principal tiene direcciones de 32 bits que seleccionan palabras de 32 bits.
- El banco de registros tiene 32 registros de 32 bits.
- Hay un ciclo de instrucción en el que el procesador procesa la instrucción indicada por el registro PC.
- Hay instrucciones que calculan la suma de un registro con una constante.