

Práctica 1

Instrucciones y registros

Realización de la práctica.

La pantalla principal del MARS se divide en diferentes partes.

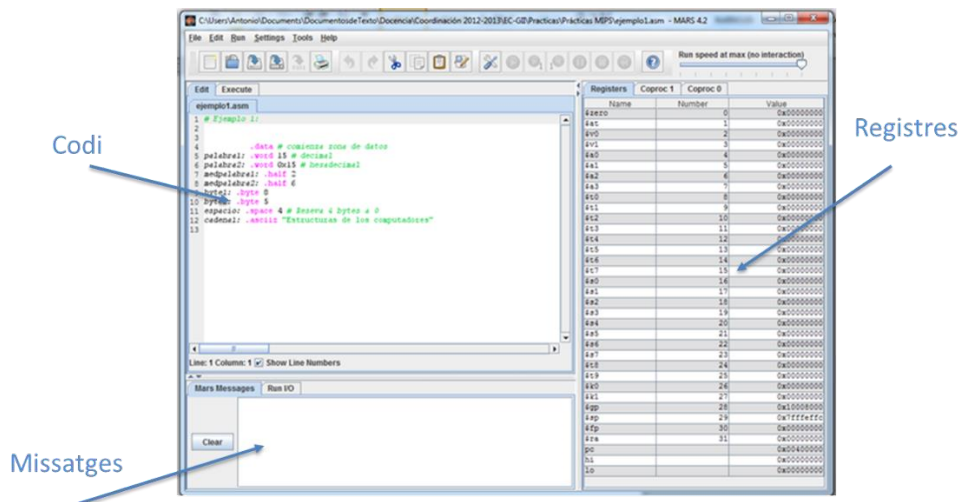


Figura 6. Pantalla principal del simulador MARS.

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.

No deja pinchar para cambiar las memorias \$0, \$31 ni \$pc

`$a0` 4 `0x00018a92`

- Probad a escribir valores negativos en los registros.

Para escribir números negativos en hexadecimal al pasarlo a binario el primer número puede ser tanto 8 como 9 como, a, b, c, d, e, f. Por lo que cualquier número hexadecimal que empiece por esos números será negativo

`$a2` 6 `0xff000000`

Cuestión 1.

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

El valor más alto que puede contener una memoria sería el 0x7fffffff, todo unos y el primer número 0

`$v1` 3 `0x7fffffff`

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

El mayor número negativo sería el 0x80000000 un 1 y todo ceros en binario

`$v1` 3 `0x7fffffff`

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 qué 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 número 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 de 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:

Código 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*.

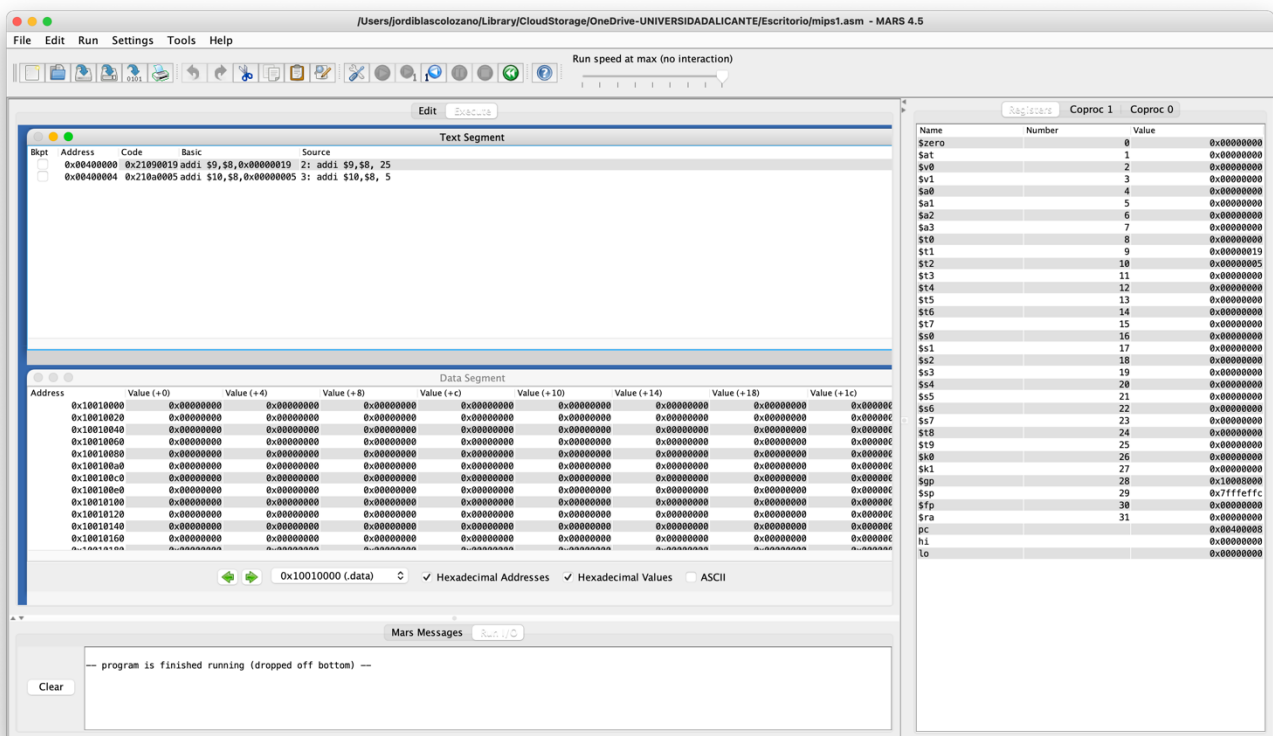
Los siguientes dos campos son los números de los registros fuente y destino, en el ejemplo el 8 y el 9. El tercer campo es el valor en complemento a dos de la constante *K*, en el ejemplo el valor 25.

Cuestión 2.

- ¿Cómo se codifica la instrucción *addi \$t0, \$s8, 5*? Escribid el código resultante en hexadecimal

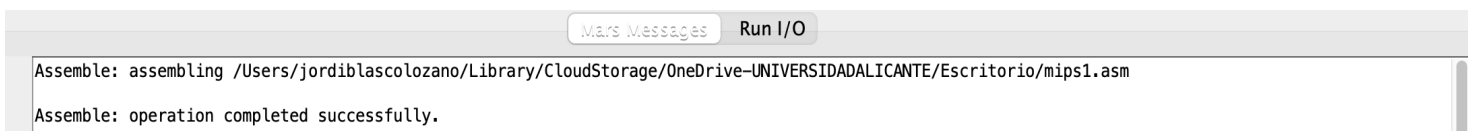
He realizado la primera y la segunda instrucción y se han guardado en las nuevas memorias las sumas que corresponden

\$t1	9	0x00000019
\$t2	10	0x00000005



3. Ensamblado

Escribid, nombrad y guardad el código fuente en un archivo de texto (*File->Save*). Ensamblad (*Run->Assemble*) 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*.



Si nos fijamos en esta ventana vemos que la información está tabulada, en ella 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 realzada en amarillo cual es la instrucción que va a ejecutarse.

Actividad 2

- Observa la ventana *Text Segment*. ¿En qué dirección se almacena cada instrucción del programa?

La dirección se almacena en la columna *Address*

- Comprobad la codificación de las instrucciones en código máquina.

La codificación se encuentra en la columna *code*

Text Segment				
Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x21090019	addi \$9,\$8,0x00000019	2: addi \$9,\$8, 25
<input type="checkbox"/>	0x00400004	0x210a0005	addi \$10,\$8,0x00000005	3: addi \$10,\$8, 5



- Observa la ventana *Registers*. ¿Qué vale el PC?

pc	0x00400000
----	------------

Vale 0x00400000

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 ciclos 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 de la siguiente manera: *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 $\$8 = \$8 - 1$ utilizando *addi*? ➤
¿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 tabla 1 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 el 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 convenio de 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 se muestra 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 se aborta el programa 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* por *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 números 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*:

\$12=5

\$10= 8

\$13=\$12 + 10

\$10=\$10 - 4

\$14=\$13 - 30

\$15=\$10

- 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.