



Universidad de Córdoba

Arquitectura y Tecnología de Computadores

Práctica 1

Manejo del simulador MARS

Arquitecturas Avanzadas de Procesadores

D. Miguel Ángel Montijano Vizcaíno (el1movim@uco.es)

D. Héctor Martínez Pérez (el2mapeh@uco.es)

Curso 2025/2026

1 Introducción

Esta práctica consiste en el aprendizaje del diseño y la implementación en el simulador de procesadores MIPS MARS, desarrollado por Peter Sanderson y Kenneth Vollmar de Missouri State University.

Los objetivos principales de la práctica son:

- Introducción al simulador MARS.
- Codificación, formatos de instrucción y tipos de datos del MIPS.
- Instrucciones básicas aritméticas, lógicas, de salto, de acceso a memoria, etc.

2 Instalación del software

Puesto que MARS está escrito en Java, para poder ejecutarlo será necesario disponer de una versión actualizada de Java Runtime Environment en el sistema operativo.

La última versión de MARS puede descargarse desde la página web de Missouri State University:

<https://computerscience.missouristate.edu/mars-mips-simulator.htm>

O directamente desde su repositorio *github* (ya sea bien, el código fuente o un JAR precompilado):

<https://github.com/dpetersanderson/MARS/>

Una vez instalada/actualizada la versión de Java y disponer del JAR de MARS descargado/compilado, puede lanzarse en Windows con doble clic (si se tiene asociado el SDK de JAVA a los ficheros con extensión .jar) o en Linux mediante el comando `java -jar Mars4_5.jar`.

3 MARS

3.1 Intefaz inicial del entorno MARS

Al arrancar MARS se observará la interfaz principal de edición como puede observarse en la Figura 1. En esta pantalla, pueden diferenciarse tres zonas principales:

- La zona de edición o simulación (dependiendo de la pestaña activa). Esta ocupa la mayor parte de la pantalla.
- La zona de mensajes MARS y entrada/salida de ejecución en la parte inferior.
- La zona de registros en la parte derecha. Esta zona será importante en la simulación.

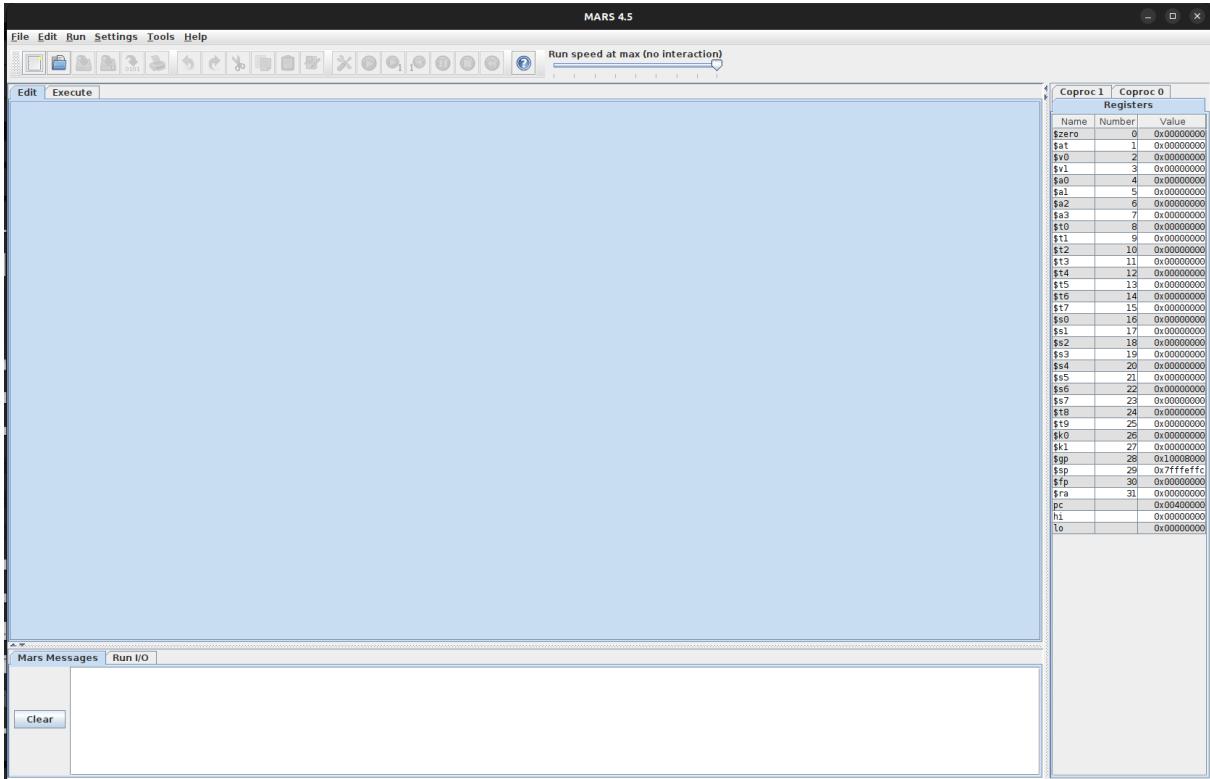


Figura 1: Interfaz principal del entorno MARS.

3.2 Carga/Almacenamiento del código fuente

El siguiente paso consistirá en llevar a cabo la carga de un código fuente en el entorno MARS. Para ello, en primer lugar, se debe descargar el recurso *"Ejemplo: Reducción de un vector en MIPS"* disponible en la PRÁCTICA 1 de Moodle. Una vez descargado el fichero denominado *vreduction.asm*, se procederá a la carga del mismo desde el entorno MARS. Para ello, mediante la opción *File → Open*, tal y como puede observarse en la Figura 3, se seleccionará el fichero correspondiente para su apertura.

Puede observarse en la pestaña principal *Edit* cómo se ha cargado el código fuente que contiene el fichero *asm*, tal y como muestra la Figura 2. Esta pestaña se trata de un editor de código fuente que permitirá al usuario hacer las modificaciones pertinentes.

Por otro lado, el usuario puede crear sus propios ficheros de código fuente mediante la opción *File → New* y almacenarlos mediante *File → Save as...*. Si únicamente se trata de cambios sobre un fichero ya existente, debe seleccionarse la opción *File → Save*.

```

/vreduction.asm
1  # Vector Reduction Example
2  # @Hector Martinez Perez (el2mapeh@uco.es)
3  # October 2026
4  #
5  # The C/C++ equivalent of this MIPS program is:
6  #
7  #     #define SIZE 4
8  #     int data[SIZE] = {0, 1, 2, 3};
9  #     int value, i;
10 #
11 #
12 #         value = 0;
13 #         for (i = 0; i < SIZE; i++)
14 #             value += data[i];
15 #
16 #
17 # data
18 # array: .word 0, 1, 2, 3
19 # size: .word 4
20 #print: .asciz "The reduction is "
21 #
22 #test
23 li $t0, size    #Load size=4 into $t0
24 sll $t0, $t0, 2  #Left shift (4bytes per element)
25 li $t1, 0        #Set $t1 = 0
26 li $t2, 0        #Set $t2 = value
27 #
28 loop:
29 la $t3, array($t1) #Load a word from array
30 add $t2, $t2, $t3 #Add a new vector element
31 add $t1, $t1, 4   #Update displacement direction
32 bne $t1, $t0, loop #Branch if not equal
33 #
34 fmr $t0, $t1      #FPU to integer conversion
35 li $v0, print    #I/O syscall: code 4 for a string
36 li $v0, 4        #I/O syscall: code 1 for an int (a word of 4 bytes in this case)
37 syscall
38 #
39 move $a0, $t2    #I/O syscall: code 10 for exit
40 li $v0, 1        #I/O syscall
41 syscall
42 #
43 li $v0, 10       #I/O syscall: code 10 for exit
44 syscall
45

```

Registers window:

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10000000
\$sp	29	0x7fffffc0
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Figura 2: Editor de código fuente.

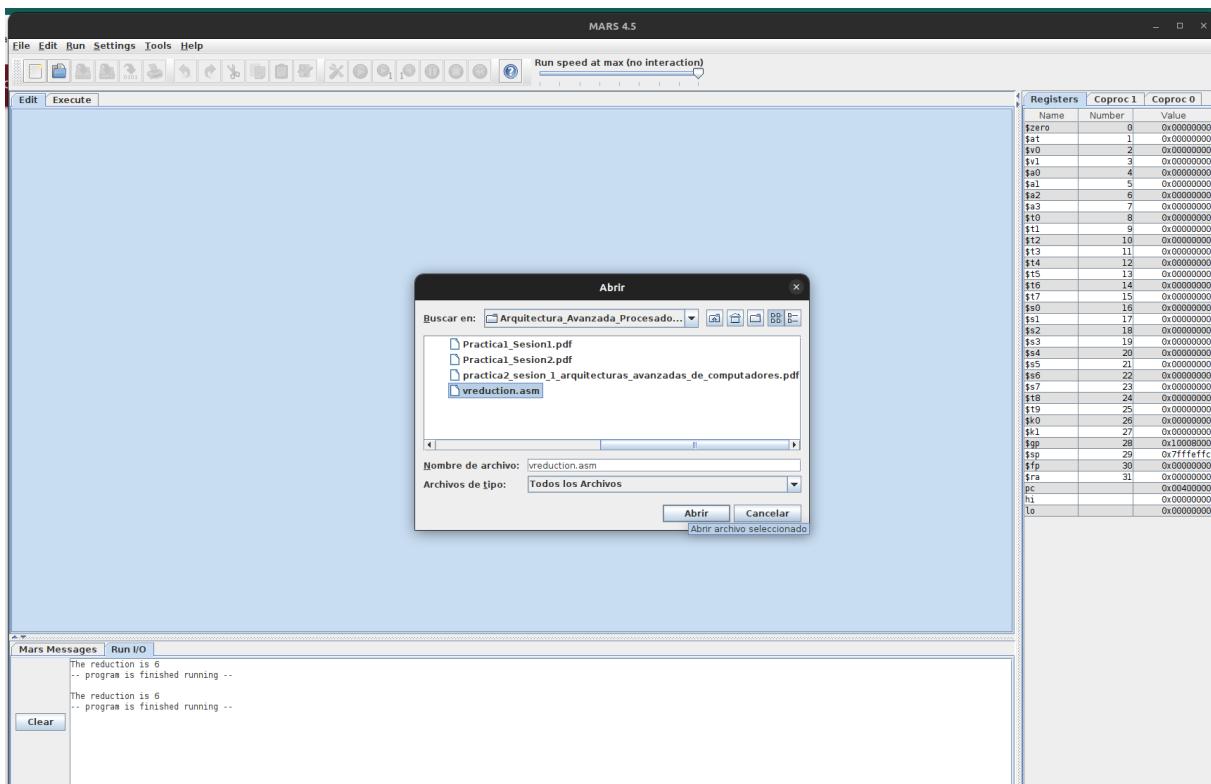


Figura 3: Carga del código fuente.

3.3 Ensamblaje del código fuente

El siguiente paso consiste en ensamblar el código escrito en lenguaje ensamblador traduciéndolo a código máquina. Esto lo haremos mediante la opción *Run → Assemble*. La Figura 4 muestra el ensamblaje del código fuente del ejemplo.

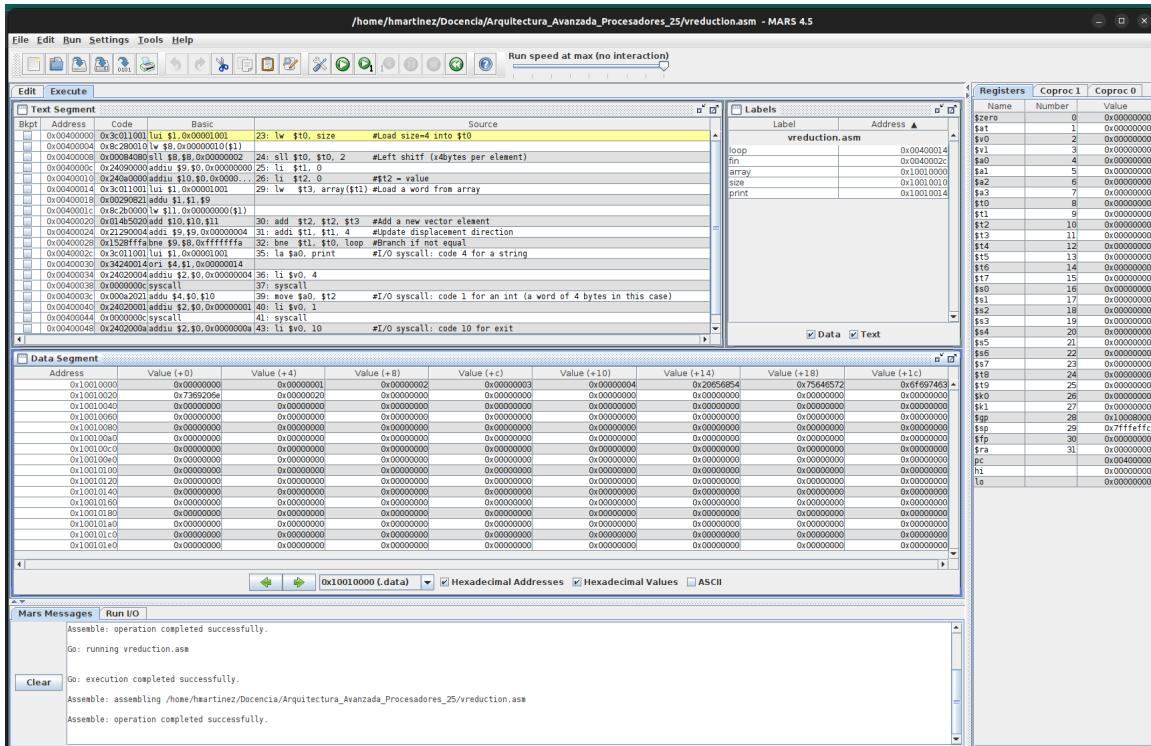


Figura 4: Ensamblaje del código fuente.

Se observa que aparece activa la zona de ejecución. Esta zona de ejecución está dividida en tres ventanas:

- Ventana de segmento de código "*text segment*": Se pueden encontrar las instrucciones del programa y las direcciones de memoria que les corresponden.
- Ventana de segmentos de datos "*data segment*": Se encuentran las variables ubicadas en memoria.
- Tabla de símbolos "*labels*": Aparecen los nombres de las variables y etiquetas de saltos junto con sus respectivas direcciones a memoria. **Nota:** En caso de que dicha ventana no aparezca, puede ser activada desde la opción *Settings → Show Labels Window (symbol table)*.

La zona de código muestra: el código fuente *source*, las instrucciones ensamblador *basic*, las instrucciones en código máquina representadas en hexadecimal *code* y las direcciones de las mismas *address*.

Entre las instrucciones ensamblador y máquina, existe una correspondencia biunívoca, no así entre el código fuente y aquellas. Esto último se debe a que el lenguaje ensamblador de MIPS ofrece pseudoinstrucciones, es decir, operaciones con apariencia de instrucciones,

pero que no están soportadas en la circuitería. Cuando se emplea una pseudoinstrucción, el ensamblador la traduce a una o varias instrucciones.

A la vista del código fuente y las instrucciones ensamblador, en MARS resulta muy sencillo identificar cuándo una operación es una pseudoinstrucción y cómo se ha traducido esta en instrucciones máquina. En el ejemplo, la instrucción *la* es una pseudoinstrucción que se traduce a su vez en dos instrucciones *lui* y *ori*.

Por defecto, las direcciones, el contenido de los registros, etc. Se muestran en hexadecimales; MARS no ofrece la posibilidad de visualizar esta información en base 10.

En esta práctica es interesante identificar la ubicación de las etiquetas correspondientes a nombres de variables, funciones, saltos, etc. Para ello, puede consultarse la ventana *Labels*, tal y como muestra la Figura 5. Esto ayudará a seguir la evolución de sus contenidos a lo largo del programa. Así, vemos que la función *loop* se encuentra en la dirección hexadecimales 0x00400014, mientras que la etiqueta *print* se ubica en la dirección 0x10010014.

Label	Address
<i>vreduction.asm</i>	
loop	0x00400014
fin	0x0040002c
array	0x10010000
size	0x10010010
print	0x10010014

Data Text

Figura 5: Ejemplo de ubicación de las funciones.

La ventana *Text Segment* contiene la lista de instrucciones que van a ir ejecutándose. Por defecto aparecen almacenadas a partir de la dirección 0x00400000, tal y como puede observarse en la Figura 6. En esta misma Figura, puede apreciarse las cinco columnas principales que a continuación se describen:

- *Bkpt*: Columna usada para establecer puntos de quiebre *breakpoints* a la hora de llevar a cabo la simulación del programa.
- *Address*: Representa la dirección hexadecimal en memoria correspondiente a cada instrucción.
- *Code*: Código hexadecimal del código máquina para cada una de las instrucciones.

- *Basic*: Lenguaje ensamblador de la instrucción, usando la representación numérica de los registros y valores indicados por cada instrucción.
- *Source*: Código fuente introducido en el editor (puede incluir mnemónicos de los registros y/o opcodes de la psudo-instrucción).

		Code	Basic	Source
Bkpt	Address			
	0x00400000	0x3c011001	lui \$1,0x00000001	23: lw \$t0, size #Load size=4 into \$t0
	0x00400004	0x8c280010	lw \$8,0x00000010(\$1)	
	0x00400008	0x000840B0	sll \$8,\$8,0x00000002	24: sll \$t0, \$t0, 2 #Left shift (x4bytes per element)
	0x0040000c	0x24090000	addiu \$9,\$0,0x00000000	25: li \$t1, 0
	0x00400010	0x240a0000	addiu \$10,\$0,0x0000...	26: li \$t2, 0 #\$t2 = value
	0x00400014	0x3c011001	lui \$1,0x00000001	29: lw \$t3, array(\$t1) #Load a word from array
	0x00400018	0x00290821	addi \$1,\$1,\$9	
	0x0040001c	0x8c2b0000	lw \$11,0x00000000(\$1)	
	0x00400020	0x014b5620	add \$10,\$10,\$11	30: add \$t2, \$t2, \$t3 #Add a new vector element
	0x00400024	0x21290004	addi \$9,\$9,0x00000004	31: addi \$t1, \$t1, 4 #Update displacement direction
	0x00400028	0x1528ffffab	ne \$9,\$8,0xffffffff	32: bne \$t1, \$t0, loop #Branch if not equal
	0x0040002c	0x3c011001	lui \$1,0x00000001	35: la \$a0, print #I/O syscall: code 4 for a string
	0x00400030	0x34240014	ori \$4,\$1,0x00000014	
	0x00400034	0x24020004	addiu \$2,\$0,0x00000004	36: li \$v0, 4
	0x00400038	0x0000000c	syscall	37: syscall
	0x0040003c	0x000a2021	addu \$4,\$0,\$10	39: move \$a0, \$t2 #I/O syscall: code 1 for an int (a word of 4 bytes in this case)
	0x00400040	0x24020001	addiu \$2,\$0,0x00000001	40: li \$v0, 1
	0x00400044	0x0000000c	syscall	41: syscall
	0x00400048	0x240200a0	addiu \$2,\$0,0x00000000	43: li \$v0, 10 #I/O syscall: code 10 for exit

Figura 6: Conjunto de instrucciones a ejecutarse.

La ventana llamada *Data Segment* representa los datos almacenados en memoria, tal y como puede observarse en la Figura 7. El área de datos comienza por defecto en la dirección 0x10010000. Dicha ventana representa la memoria del sistema organizando de manera consecutiva y mediante ocho columnas las direcciones de los datos ubicados en memoria. Cada columna representa un espacio de memoria de cuatro bytes. [Los cuatro bytes corresponde a las palabras de 32 bits usadas en el procesador MIPS](#).

Los botones inferiores con flechas a izquierda y derecha, son usados para retroceder o avanzar, respectivamente, a través de la memoria. Además, el menú desplegable muestra las direcciones de inicio de otras porciones de memoria como *.extern*, *.data*, *heap*, etc. Así como los valores actuales de los registros del apuntador de pila, *Stack Pointer (\$sp)*, y del apuntador global, *Global Pointer (\$gp)*.

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000001	0x00000002	0x00000003	0x00000004	0x20656854	0x75646572	0x6f697463
0x10010020	0x7369206e	0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010180	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100101e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Figura 7: Representación de los datos almacenados en el banco de memoria.

3.4 Simulación en MARS

Antes de ejecutar el código ensamblado, debe seleccionarse la pestaña llamada *Registers* en la ventana ubicada en la derecha de MARS. Nótese que la mayoría de los registros, exceptuando *Global Pointer (\$gp)*, *Stack Pointer (\$sp)* y *Program Pointer (\$pc)*, contienen cero. Los valores de los registros que no están a cero son mostrados en la Tabla 1.

MIPS Register	\$gp	\$sp	pc
Value	0x10008000	0x7fffffc	0x00400000

Tabla 1: Valores iniciales de registros en MIPS.

A continuación, agregue *breakpoints* en las últimas dos líneas del programa (para observar su comportamiento), marcando las respectivas casillas asociadas a cada instrucción. A continuación, seleccione la opción *Run → Go* y arrancará la ejecución del programa ensamblado parándose en el primer *breakpoint*.

En este punto, y sin modificar el estado de la ejecución, se llevará a cabo el primer ejercicio propuesto en esta práctica.

Ejercicio 1

1.1. Observe los valores actuales de los registros. ¿Cuántos registros han cambiado sus valores? Indíquelos junto con su valor en decimal y hexadecimal, siguiendo el esquema de la siguiente tabla.

Nombre del registro	Valor Hexadecimal	Valor Decimal

1.2. ¿Cuál es el nombre completo del GPR (*General Purpose Register*) que contiene la dirección donde se ha detenido el programa?

1.3. ¿Cuál es la instrucción encargada del salto condicional del bucle que recorre el vector?

1.4. ¿Cuál es el nombre del registro encargado de acumular la suma de los elementos del vector?

1.5. ¿Cuál es el propósito de la instrucción *sll*? ¿Por qué se aplica sobre el registro \$t0?

Ejercicio 2

Cree un archivo de edición seleccionando *File → New* en la parte superior de la ventana del MARS. Esto hará que se abra un documento en blanco que llenará la ventana de edición del MARS mostrando el cursor listo para escribir un nuevo programa.

Escriba las cuatro líneas mostradas en el Código 1 y seleccione la opción *File → Save as...* e inserte un nombre de interés para el archivo. Por convención, los archivos fuente en lenguaje ensamblador usan extensiones *.asm* o *.s*. **Nota: El archivo debe ser almacenado antes de que el código fuente pueda ser ensamblado.**

Código 1: Muestra simple de código MIPS.

```
1 addi $t0, $zero, 100
2 addi $t1, $zero, 50
3 add $t2, $t0,    $t1
4 sub $t3, $t0,    $t1
```

A continuación, seleccione *Run → Assembler* del menú de la parte superior del MARS. Note el cambio en la apariencia del MARS. Antes de ejecutar el código ensamblado, seleccione la pestaña *Registers*.

2.1. Indique los registros actuales que no contienen cero y su valor actual.

2.2. Agregue dos *breakpoints* en las últimas dos líneas del programa. A continuación, ejecute el programa e indique en una tabla similar a la del ejercicio 1.1 los valores actuales de los registros.

2.3. El programa se ha detenido en la instrucción iluminada en amarillo ¿A sido ejecutada dicha instrucción?

2.4. Vuelva a ejecutar el programa, es decir, avanzando un paso hasta el siguiente *breakpoint*. Indique de nuevo los registros actuales que han cambiado de valor en una tabla similar a la del ejercicio 1.1.

2.5. Finalmente, vuelve a avanzar un paso en la ejecución e indique de nuevo mediante una tabla similar a la del ejercicio 1.1 los valores de los registros que se han visto afectados.

Ejercicio 3

3.1. El ISA del MIPS tiene una convención para enumerar y nombrar los registros. Complete la siguiente tabla prestando atención a la descripción de cada uno de los diferentes tipos de registros.

Nombre del registro	Número del registro	Descripción
\$zero		
\$at		
\$v0,\$v1		
\$a0...\$a3		
\$t0...\$t9		
\$s0...\$s7		
\$k0,\$k1		
\$gp		
\$sp		
\$fp		
\$ra		

3.2. ¿Cuál es el propósito de la instrucción *addi \$t1, \$zero, 50*?

3.3. ¿Y el de *add \$t2, \$t0, \$t1* y *sub \$t3, \$t0, \$t1*?