

Prácticas de ensamblador de Estructura y Tecnología de Computadores

Febrero de 2016

Práctica 1

USO DEL SIMULADOR MARS

1.1 OBJETIVOS

Los objetivos de esta sesión son que el alumno se familiarice con el simulador MARS y que sea capaz de seguir la ejecución de programas sencillos en ensamblador de MIPS.

1.2 PREREQUISITOS

- Lectura de los apuntes de teoría, en especial las secciones relativas a la introducción a ensamblador, repertorios de instrucciones, operandos del computador, llamadas al sistema operativo e ISA de MIPS.


1.3 PLAN DE TRABAJO

El plan de trabajo de esta sesión será el siguiente:

1. Lectura por parte del alumno de las secciones 1.4 y 1.5.
2. Realización, en grupos de dos personas, de los ejercicios propuestos en el boletín (con supervisión del profesor).

1.4 EL SIMULADOR MARS

MARS es un programa para simular el ISA de MIPS diseñado para uso educativo. Puede ser descargado libremente de la dirección <http://cs.missouristate.edu/MARS/>, aunque para la realización de las prácticas de la asignatura se deberá utilizar siempre la versión disponible en la página web de la asignatura (<http://ditec.um.es/etc/>). Esta última versión incluye algunas modificaciones necesarias para los ejercicios y el proyecto que se pondrán a lo largo del curso.

Cuando se arranca el programa y se abre el fichero `ejemplo1.s` (cuyo contenido se muestra en la figura 1.3), la ventana de MARS tiene el aspecto que se puede ver en la figura 1.1. Podemos editar directamente el programa usando MARS, o usar cualquier otro editor de texto. También se puede ensamblar el programa usando la tecla `F3` o el botón  de la barra de herramientas, tras lo que el programa tendrá el aspecto de la figura 1.2.

MARS incluye en su distribución una documentación escueta pero útil. Esta documentación está accesible desde el menú «Help» o pulsando la tecla `F1`. Se recomienda al alumno que se familiarice

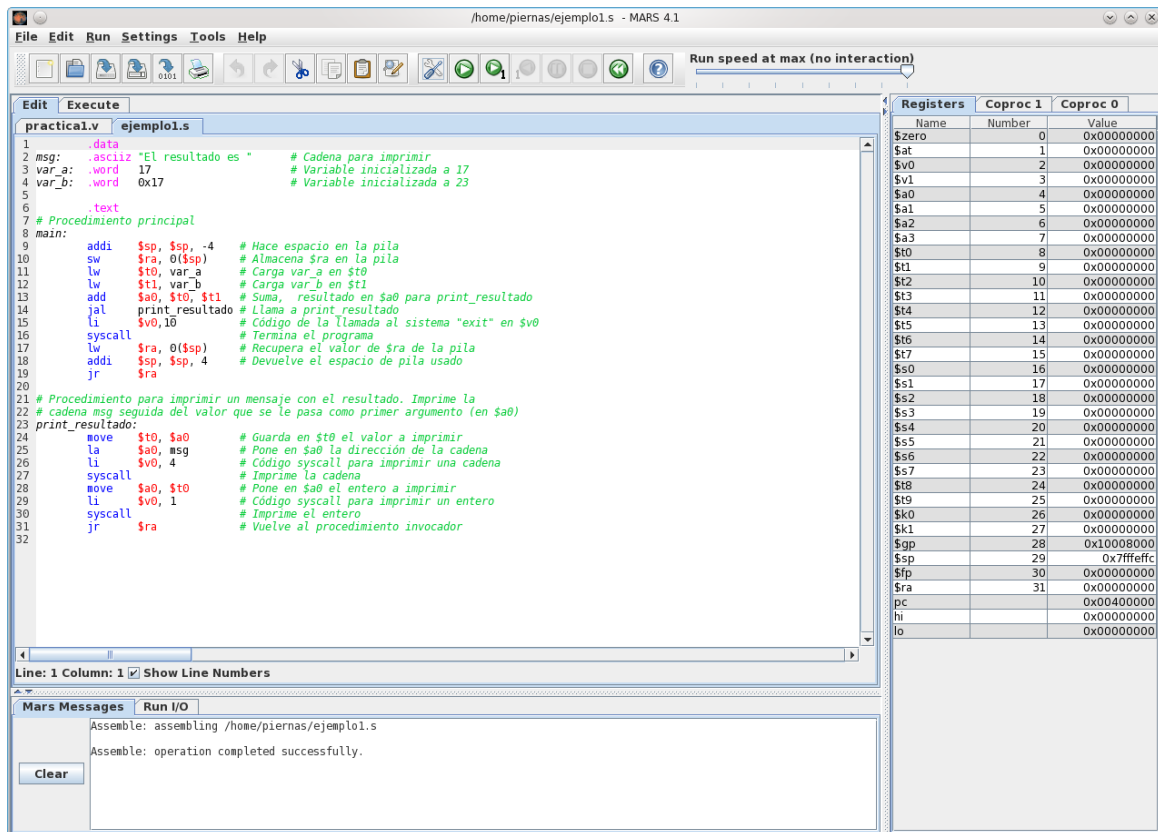


Figura 1.1: Aspecto de MARS después de cargar un programa.

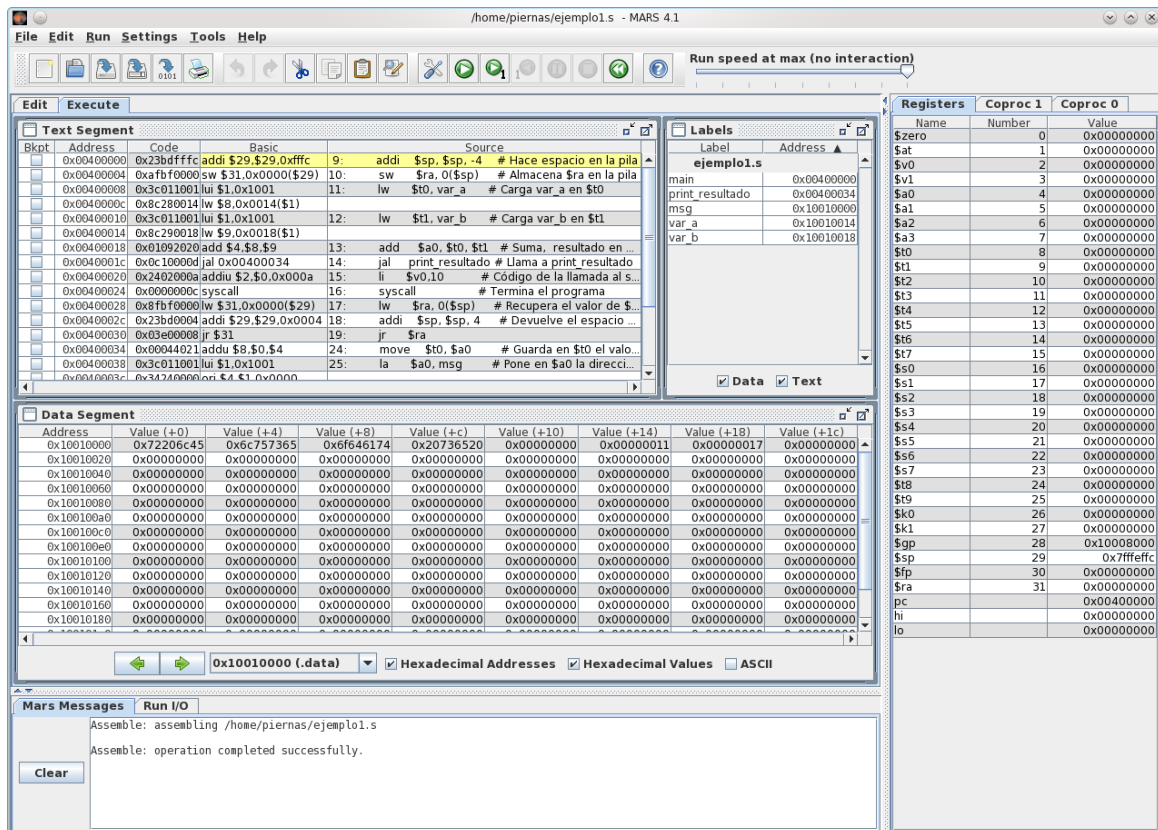


Figura 1.2: Aspecto de MARS después de ensamblar un programa.

con la documentación disponible. Ésta incluye un listado de las instrucciones, pseudoinstrucciones, directivas y llamadas al sistema disponibles en MARS.

En la figura 1.2 podemos ver que MARS nos muestra varias ventanas y paneles de información:




- La ventana titulada «*Text Segment*» nos permite ver el contenido del *segmento de texto*, es decir, de la zona de memoria que contiene el programa recién ensamblado. Cada línea de la tabla se corresponde con una palabra de 4 bytes, es decir: una instrucción de código máquina. De cada palabra, podemos ver su dirección, su contenido en hexadecimal, la instrucción correspondiente tal y como la vería el procesador, y la línea de ensamblador que generó la instrucción. En el caso de pseudoinstrucciones que generan varias instrucciones en código máquina, la línea de ensamblador aparece asociada solo a la primera instrucción real (como es el caso de las instrucciones almacenadas en las direcciones 0x00400008 y 0x0040000c, que se generan a partir de la línea 11 de `ejemplo1.s`).
- En la ventana «*Data Segment*» podemos ver el contenido de la zona de memoria que almacena los datos. De nuevo, la información aparece separada en palabras de 4 bytes. Esta ventana también nos permite modificar el contenido de la memoria escribiendo directamente en la celda correspondiente.
- La ventana «*Labels*» nos muestra la dirección de memoria que el ensamblador ha asociado con cada una de las etiquetas declaradas en el programa.
- A la derecha de la ventana principal disponemos de un panel que muestra el contenido de los registros del procesador y permite modificarlo. La información está dividida en 3 pestañas:
 - Registers:** muestra el contenido de los 32 registros de propósito general y de tres registros especiales: el contador de programa (`pc`) y los registros `hi` y `lo` usados por las instrucciones de multiplicación y división.
 - Coproc0:** muestra algunos de los registros del coprocesador 0 relacionados con el manejo de excepciones.
 - Coproc1:** muestra el banco de registros en coma flotante. Los registros aparecen tanto individualmente (para los valores de simple precisión) como en parejas (para los valores de doble precisión).
- En la parte inferior de la ventana disponemos de un panel con 2 pestañas. La primera de ellas muestra los mensajes de MARS, incluyendo los errores encontrados al ensamblar un programa. La segunda muestra la salida del programa simulado.


Si observamos la información mostrada por MARS justo después de ensamblar el programa, podemos ver, por ejemplo, que la etiqueta `var_b` está asociada a la dirección 0x1001018. Si miramos en la ventana «*Data Segment*» el valor almacenado en dicha dirección, vemos que es 0x17, tal y como se especifica en la línea 4 de `ejemplo1.s`. Análogamente, la etiqueta `print_resultado` está asociada con la dirección 0x00400034, la cual podemos comprobar en la ventana «*Text Segment*» que contiene la primera instrucción del procedimiento, definida en la línea 24 de `ejemplo1.s`.

También podemos ver que el registro `pc` contiene el valor 0x0040000, que se corresponde con el comienzo del segmento de texto. Cuando se ensambla un programa, MARS asigna a dicho registro la dirección del procedimiento principal¹ o la dirección de inicio del segmento de texto si no se

¹Para que el procedimiento principal sea reconocido como tal debe tener una etiqueta llamada «`main`» que haya sido declarada como global (con la directiva «`.global`»). En particular, el procedimiento «`main`» de `ejemplo1.s` no sería reconocido como procedimiento principal porque no se ha incluido la directiva «`.global`» correspondiente.

encuentra un procedimiento principal. Por su parte, el registro `sp` se inicializa al valor `0x7ffeffc` donde se encuentra la cima de la pila vacía.

Podemos iniciar la ejecución del programa actual con la tecla `F5` o el botón . MARS permite ralentizar la ejecución del programa para facilitar su seguimiento paso a paso usando el control etiquetado «*Run speed*» situado a la derecha de la barra de herramientas. La ejecución también se puede realizar instrucción a instrucción usando la tecla `F7` o el botón , e incluso hacia atrás con la tecla `F8` o el botón .

Si ejecutamos el programa después de ensamblarlo, aparecerá en la pestaña «*Run I/O*» el mensaje «El resultado es 40» seguido de un mensaje indicando que el programa ha acabado correctamente. Para volver a ejecutar el programa, podemos volver a ensamblar el programa o devolver el simulador a su estado original con la tecla `F12` o el botón .

También podemos usar la casilla «*Bkpt*» que se encuentra a la izquierda de cada instrucción en la ventana «*Text segment*» para conseguir que la ejecución se pare cada vez que se llegue a esa instrucción.

Además de simular el ISA de MIPS de forma suficientemente completa para ejecutar programas simples, MARS también pone a disposición del programador un conjunto de llamadas al sistema. En una máquina real, estas llamadas al sistema serían provistas por el sistema operativo.

Las llamadas al sistema disponibles incluyen las necesarias para realizar la entrada y salida. El listado completo de llamadas disponibles puede consultarse en la documentación de MARS (tecla `F1`). En la sección 1.5 se comentará el uso de las llamadas al sistema usadas por `ejemplo1.s` (por ejemplo, para mostrar el mensaje con el resultado).

1.5 ANATOMÍA DE UN PROGRAMA EN ENSAMBLADOR

En la figura 1.3 se muestra el programa `ejemplo1.s`. Este programa es muy simple: realiza la suma de dos valores almacenados en sendas variables e imprime un mensaje con el resultado.

El programa está dividido en dos secciones: una de datos y otra de código. Las directivas `.data` y `.text` marcan el inicio de cada una de ellas, respectivamente.

En la sección de datos se definen tres etiquetas. La primera de ellas (`msg`) apuntará a una cadena que se usará posteriormente para mostrar un mensaje. El mensaje se codifica con la directiva `.ascii`, que define una cadena codificada en ASCII y terminada en 0. Las dos restantes (`var_a` y `var_b`) apuntan al espacio reservado para las dos variables. Este espacio se reserva y se inicializa mediante las directivas `.word`.

En el segmento de código se definen dos procedimientos: el procedimiento principal `main` y el procedimiento `print_resultado`, que es llamado desde `main`.

El procedimiento `main` es el que se empieza a ejecutar al principio del programa. Este procedimiento, al igual que cualquier otro, podemos dividirlo en tres partes:

Prólogo: de la línea 9 a la 10. En él, se guarda en la pila el valor de los registros de tipo *preservados entre llamadas* que se van a modificar en el cuerpo del procedimiento.

Cuerpo: de la línea 11 a la 18. Se realiza el trabajo del procedimiento.

Epílogo: de la línea 19 a la 21. Se recuperan los valores guardados anteriormente por el prólogo y se retorna el control al procedimiento invocador.

En el prólogo de este procedimiento se guarda en la pila² sólo el registro `$ra`, ya que es el único del conjunto de registros de tipo *preservados entre llamadas* que se modifica en el cuerpo del

²En el caso concreto de este procedimiento, no sería estrictamente necesario guardar nada, ya que el programa acaba siempre en la línea 18, antes de que se llegue a ejecutar el epílogo.

```

1      .data
2 msg:  .ascii "El resultado es "      # Cadena para imprimir
3 var_a: .word 17                      # Variable inicializada a 17
4 var_b: .word 0x17                    # Variable inicializada a 23
5
6      .text
7 # Procedimiento principal
8 main:
9      addi    $sp, $sp, -4            # Hace espacio en la pila
10     sw      $ra, 0($sp)             # Almacena $ra en la pila
11     lw      $t0, var_a              # Carga var_a en $t0
12     lw      $t1, var_b              # Carga var_b en $t1
13     add     $a0, $t0, $t1           # Suma, resultado en $a0 para print_resultado
14     jal     print_resultado         # Llama a print_resultado
15     li      $v0, 10                 # Código de la llamada al sistema "exit" en $v0
16     syscall                                # Termina el programa
17     lw      $ra, 0($sp)             # Recupera el valor de $ra de la pila
18     addi    $sp, $sp, 4             # Devuelve el espacio de pila usado
19     jr      $ra
20
21 # Procedimiento para imprimir un mensaje con el resultado. Imprime la
22 # cadena msg seguida del valor que se le pasa como primer argumento (en $a0)
23 print_resultado:
24     move     $t0, $a0               # Guarda en $t0 el valor a imprimir
25     la       $a0, msg               # Pone en $a0 la dirección de la cadena
26     li       $v0, 4                 # Código syscall para imprimir una cadena
27     syscall                                # Imprime la cadena
28     move     $a0, $t0               # Pone en $a0 el entero a imprimir
29     li       $v0, 1                 # Código syscall para imprimir un entero
30     syscall                                # Imprime el entero
31     jr      $ra                     # Vuelve al procedimiento invocador

```

Figura 1.3: ejemplo1.s

procedimiento (en concreto, se modifica en la línea 15).

El cuerpo del procedimiento carga el valor almacenado en las variables `var_a` y `var_b`, lo suma y llama al procedimiento `print_resultado`. Obsérvese que el resultado de la suma (línea 13) se almacena en el registro `$a0` para que sea el primer argumento de `print_resultado`. Por último, el cuerpo del procedimiento invoca la llamada al sistema número 10, la cual finaliza la ejecución del programa.

Por otro lado, el procedimiento `print_resultado` se limita a recibir un argumento, imprimir un mensaje e imprimir el valor del argumento recibido. Obsérvese que el prólogo de este procedimiento está vacío dado que no modifica el valor de ningún registro de tipo *preservado entre llamadas* y el epílogo se limita a devolver el control al procedimiento invocador mediante la instrucción «`jr $ra`».

1.6 EJERCICIOS

Para cada ejercicio, el portafolios deberá incluir una explicación de cómo se ha resuelto y, en caso de que se realice o modifique algún programa, el código realizado y al menos un ejemplo de su ejecución.

1. Cargue, ensamble y ejecute el programa `ejemplo1.s`. Compruebe que el mensaje mostrado

en la pestaña «*Run I/O*» es «El resultado es 40». Utilice MARS para, sin modificar el código en ensamblador, conseguir que la salida del programa en la siguiente ejecución sea «El resultado es 5».

2. Modifique el programa `ejemplo1.s` para que, en vez de sumar el contenido de las variables `var_a` y `var_b`, le pregunte cada vez al usuario qué números sumar.
3. Escriba un programa para sumar fracciones. El programa le debe preguntar al usuario el valor de cuatro variables (a , b , c y d) y debe calcular y mostrar en forma de fracción el resultado de $\frac{a}{b} + \frac{c}{d}$ (es decir, $\frac{a \times d + c \times b}{b \times d}$). No es necesario que la fracción resultante aparezca simplificada y se puede asumir que todos los resultados y valores intermedios caben en 32 bits.

Práctica 2

CONDICIONALES, BUCLES Y ACCESO A ARRAYS EN ENSAMBLADOR

2.1 OBJETIVOS

Los objetivos de esta sesión son que el alumno sea capaz de escribir pequeños procedimientos en ensamblador utilizando expresiones condicionales, bucles y acceso a arrays.

2.2 PREREQUISITOS

- Lectura de los apuntes de teoría sobre ensamblador, especialmente las secciones relativas a las instrucciones de salto y las de acceso a memoria.

2.3 PLAN DE TRABAJO

El plan de trabajo de esta sesión será el siguiente:

1. Lectura por parte del alumno de la sección 2.4.
2. Realización, en grupos de dos personas, de los ejercicios propuestos en el boletín (con supervisión del profesor).

2.4 CONVERSIÓN DE ENTEROS A CADENAS

Como todos sabemos, el computador representa internamente los números enteros con signo utilizando la representación binaria en complemento a dos. Por tanto, cada vez que se desea mostrar (o leer) un número, es necesario realizar la conversión desde la representación interna a una cadena de dígitos que pueda ser interpretada fácilmente por un ser humano (o viceversa).

Estas conversiones las realizan internamente las funciones de la librería del sistema. Por ejemplo, la realiza la función `printf` de C cuando se utiliza el formato «%d» para imprimir un entero. En el simulador MARS, se ofrece la llamada al sistema número 1 para mostrar un entero por pantalla, la cual también realiza esta conversión. En ambos casos se utiliza la base 10, aunque otras bases también serían posibles¹. Después de realizar la conversión, la cadena resultante se muestra por pantalla.

En esta práctica vamos a implementar en ensamblador un procedimiento capaz de realizar la conversión anteriormente descrita. Además, nuestro procedimiento será capaz de utilizar cualquier base entre 2 y 36.

¹Por ejemplo, podemos usar la base 16 usando el formato «%x» con `printf`.

La idea principal del algoritmo que utilizaremos es la misma que cuando realizamos manualmente un cambio de base. A partir del número que se desea convertir, lo iremos dividiendo repetidamente por la base hasta que el cociente sea 0. El resto de cada una de las divisiones se corresponderá con un dígito del resultado, obteniéndose primero los dígitos menos significativos.

Por ejemplo, para obtener la representación en base 10 de 1234, deberemos realizar cuatro iteraciones:

Operación	Cociente	Resto
$1234 \div 10$	123	4
$123 \div 10$	12	3
$12 \div 10$	1	2
$1 \div 10$	0	1

Los restos obtenidos en cada iteración son los dígitos de 1234 en base 10 y en orden de menos significativo a más significativo (es decir, al revés de como se escribe normalmente).

El mismo algoritmo se puede utilizar para obtener la representación en base 16 de 1234:

Operación	Cociente	Resto
$1234 \div 16$	77	2
$77 \div 16$	4	13
$4 \div 16$	0	4

De donde deducimos que 1234 en base 16 es 4d2.

El algoritmo anterior lo podríamos intentar implementar con el código en C de la figura 2.1. La función de la figura recibe tres argumentos: el número a traducir, la base deseada y la dirección de memoria de un buffer en el que se debe almacenar el resultado (es decir, un puntero al buffer). Se supone que el buffer siempre tiene suficiente tamaño². El procedimiento define un puntero (`p`) que inicializa con la dirección de memoria de comienzo del buffer. A lo largo de la ejecución del procedimiento, el puntero apuntará a la siguiente posición libre en el buffer. El bucle de las líneas 3 a la 6 realiza la conversión propiamente dicha. La variable `i`, que se inicializa al valor del número a traducir, se utiliza para controlar el bucle. Éste se ejecuta mientras que `i` valga más que 0, y al final de cada iteración `i` se actualiza dividiéndolo por `base`. En el cuerpo del bucle, se calcula un dígito cada vez. Para ello, en la línea 4 se calcula el resto de la división de `i` entre la base y el resultado, que es el valor numérico del dígito, se suma al carácter `'0'` para obtener el carácter correspondiente. En la misma línea, el resultado se almacena en la posición del buffer apuntada por `p`, y en la línea 5 se incrementa `p` para que apunte al siguiente hueco en el buffer. Finalmente, la última línea escribe el byte 0 (carácter `'\0'` en C, el cual es importante no confundirlo con el byte 48, que se corresponde con el carácter `'0'`) para marcar el fin de la cadena.

El procedimiento descrito anteriormente tiene varias deficiencias. La más obvia de todas es que la cadena del resultado se queda almacenada en `buf` en orden inverso. Además, no funciona correctamente para los números negativos o el 0, y obtiene resultados incorrectos para bases superiores a 10.

En la figura 2.2 se muestra la traducción a ensamblador del procedimiento anterior. En la línea 2 se almacena la dirección de memoria de comienzo del buffer en `$t0`, que será el registro que almacenará el puntero `p`³. Las líneas desde la 3 a la 13 implementan el bucle del procedimiento, usando el registro

²En el peor caso, dado que el número de entrada se representa con 32 bits en complemento a 2, se necesitarían 34 bytes para representar el número más largo posible, que sería -2^{31} en base 2. De los 34 bytes, el primero sería el `-`, el segundo sería un 1, los 31 siguientes serían los caracteres 0 y el último sería para el byte 0 final (`'\0'`).

³Se podría haber utilizado directamente el registro `$a2`, pero el uso del registro `$t0` facilitará los ejercicios.

```

1 void integer_to_string(int n, int base, char* buf) {
2     char *p = buf;
3     for (int i = n; i > 0; i = i / base) {
4         *p = (i % base) + '0';
5         ++p;
6     }
7     *p = '\0';
8 }

```

Figura 2.1: integer_to_string_v0.c

\$t1 para la variable *i*. La línea 5 es necesaria para que el bucle no se ejecuta ninguna vez si *n* es 0 o menor. En la línea 6 se calculan tanto el cociente como el resto de la división de *i* entre *base*, quedando en los registros especiales LO y HI, respectivamente. En la línea 7, *i* se actualiza con el cociente y en la línea 8 se copia el resto a \$t2. La línea 9 calcula el carácter correspondiente al dígito actual y la línea 10 lo almacena en la posición actual del bucle, incrementándose el puntero *p* en la siguiente línea para que apunte a la siguiente posición vacía del bucle. La línea 12 comprueba si se debe ejecutar alguna iteración más del bucle, lo que ocurre si *i* es aún mayor que 0. Finalmente, la línea 14 almacena en el bucle la marca de fin (byte 0) y la línea 15 finaliza el procedimiento y vuelve al procedimiento llamante.

```

1 integer_to_string_v0:                # ($a0, $a1, $a2) = (n, base, buf)
2     move    $t0, $a2                # char *p = buff
3     # for (int i = n; i > 0; i = i / base) {
4     move    $t1, $a0                # int i = n
5 B0_3:    blez    $t1, B0_7            # si i <= 0 salta el bucle
6     div     $t1, $a1                # i / base
7     mflo    $t1                    # i = i / base
8     mfhi    $t2                    # d = i % base
9     addiu   $t2, $t2, '0'           # d + '0'
10    sb      $t2, 0($t0)              # *p = $t2
11    addiu   $t0, $t0, 1              # ++p
12    j       B0_3                    # sigue el bucle
13    # }
14 B0_7:    sb      $zero, 0($t0)       # *p = '\0'
15 B0_10:    jr      $ra

```

Figura 2.2: integer_to_string_v0.s

En el fichero `integer_to_string_incompleto.s` se puede encontrar el procedimiento `integer_to_string_v0` junto con un procedimiento principal, dos procedimientos de prueba (`test1` y `test2`) y algunos procedimientos auxiliares para facilitar las pruebas.

2.5 EJERCICIOS

Para cada ejercicio, el portafolios deberá incluir una explicación de cómo se ha resuelto y, en caso de que se realice o modifique algún programa, el código realizado y algunos ejemplos de su ejecución.

1. Modifique el procedimiento `integer_to_string_v0` para que el resultado quede almacenado en el buffer en el orden correcto. Para ello, añada al final del procedimiento un bucle

para darle la vuelta al contenido del buffer, excepto al último byte (que es el 0 final). Llame al procedimiento modificado `integer_to_string_v1`.

IMPORTANTE: Las etiquetas utilizadas en cada procedimiento deben ser distintas, y hay que tener cuidado de renombrarlas (tanto donde se declaran como donde se usan en alguna instrucción) cuando se copie el código de un procedimiento a otro tanto en este ejercicio como en los siguientes. En ningún caso una instrucción de salto puede saltar a una etiqueta contenida en otro procedimiento salvo para llamar a otro procedimiento utilizando correctamente los convenios de programación del ABI de MIPS.

2. Modifique el procedimiento realizado en el apartado anterior para que funcione correctamente para los valores negativos. Para ello, la variable de control del bucle se deberá inicializar al valor absoluto de `n` (en lugar de `a n`). Además, se deberá poner el carácter «-» al principio del resultado. Téngase en cuenta que los dígitos se calculan en orden inverso, por lo que lo más cómodo será añadir el guión justo después de calcular la cifra más significativa y antes de darle la vuelta al contenido del buffer. Llame al procedimiento modificado `integer_to_string_v2`.
3. Modifique el procedimiento realizado en el apartado anterior para que funcione correctamente con el valor 0. Para ello, añada una comprobación al principio del procedimiento para tratar este caso especial. Llame al procedimiento modificado `integer_to_string_v3`.
4. Modifique el procedimiento realizado en el apartado anterior para que funcione correctamente para bases superiores a 10. Para ello, se deberá modificar el cuerpo del bucle cuando se le suma al resto obtenido de la división el carácter «0» (código 48). Ahora, se deberá sumar el carácter «0» para los dígitos menores que 10 y el carácter «A» (código 65) menos 10 (ya que el dígito 10 se debe representar con una A, el 11 con una B...). Llame al procedimiento modificado `integer_to_string_v4`.

Práctica 3

CONVENCIONES DE PROGRAMACIÓN EN MIPS

3.1 OBJETIVOS

El objetivo de la sesión es que el alumno sea capaz de escribir procedimientos que realicen tareas sencillas utilizando correctamente las convenciones de usos de registros y de llamadas a procedimientos de MIPS.

3.2 PREREQUISITOS

- Lectura de los apuntes de ensamblador, en especial las secciones relativas al uso de los registros y las llamadas a procedimientos.

3.3 PLAN DE TRABAJO

El plan de trabajo de esta sesión será el siguiente:

1. Lectura por parte del alumno de la sección 3.4.
2. Realización, en grupos de dos personas, de los ejercicios propuestos en el boletín (con supervisión del profesor).

3.4 EJEMPLOS DE PASO DE PARÁMETROS

Vamos a comenzar analizando un pequeño programa de ejemplo, cuyo código fuente se encuentra disponible en el fichero `procedimientos.s`. En este programa se incluyen, entre otras cosas, los siguientes procedimientos:

main: el procedimiento principal que, en este caso, se limita a llamar a `proc1` y a finalizar el programa después.

proc1: es un procedimiento que no recibe ningún argumento ni devuelve ningún resultado. Sin embargo, realiza llamadas a otros procedimientos (figura 3.1).

proc2: un procedimiento que recibe tres parámetros y devuelve dos. Siguiendo el convenio de paso de parámetros del ABI de MIPS, los parámetros se comunicarán a `proc2` usando los registros `$a0`, `$a1` y `$a2`. Por su parte, los resultados se recibirán en los registros `$v0` y `$v1` (figura 3.2).

proc3: un procedimiento que recibe seis parámetros y devuelve uno. Siguiendo los mismos convenios, los cuatro primeros parámetros se comunicarán usando los registros `$a0`, `$a1`, `$a2` y `$a3`, mientras que el quinto y el sexto parámetro se pasarán usando la pila (como se describe más adelante). Además, se utilizará el registro `$v0` para recibir el resultado del procedimiento (figura 3.3).

printlnInt: procedimiento que imprime un entero (que recibe en `$a0`) y salta a la línea siguiente de la consola (figura 3.4).

El procedimiento `proc1` comienza guardando en la pila los registros *preservados entre llamadas* que modifica (`$ra`, `$s0` y `$s1`), los cuales se desapilan al final del procedimiento. A continuación, este procedimiento realiza varias llamadas consecutivas a procedimientos:

1. A `proc2` pasándole los parámetros 3, 4 y 5. Para ello, simplemente se colocan dichos valores en los registros `$a0`, `$a1` y `$a2`, y se utiliza la instrucción `jal` para guardar el contador de programa en `$ra` y saltar a la primera instrucción de `proc2`. Obsérvese que los valores devueltos por `proc2` mediante los registros `$v0` y `$v1` se copian a los registros `$s0` y `$s1`. Esto se hace así para evitar que las llamadas posteriores a otros procedimientos sobrescriban esos valores.
2. A `printlnInt`, pasándole en el registro `$a0` el valor almacenado en `$s1`, es decir, el segundo valor que había sido devuelto anteriormente por `proc2`.
3. A `printlnInt` de nuevo, pasándole ahora el valor almacenado en `$s0`, es decir, el primer valor que había sido devuelto anteriormente por `proc2`. Obsérvese que, si no se hubiera guardado anteriormente en `$s0` el valor que se quiere imprimir ahora, éste hubiera sido sobrescrito por la ejecución anterior de `printlnInt`.
4. A `proc3` pasándole los parámetros 8, 7, 6, 5, 4 y 3. Debido a que el número de parámetros que recibe `proc3` es mayor que el número de registros dedicados para el paso de parámetros (4), es necesario utilizar la pila para comunicar algunos de ellos. Los cuatro primeros parámetros se comunican usando los registros `$a0`, `$a1`, `$a2` y `$a3`, mientras que el quinto y el sexto parámetros se apilan. Para ello, se decrementa `$sp` en 8 unidades (porque cada parámetro va a ocupar 4 bytes de la pila) y se mueve el quinto parámetro a la posición de memoria 0 (`$sp`) y el sexto a 4 (`$sp`). Es decir, la pila quedará como si se hubiera apilado primero el sexto parámetro y después el quinto¹. El procedimiento que realiza la llamada (`proc3` en este caso) es el responsable de devolver el espacio a la pila una vez que recupera el control (línea 55).

El procedimiento `proc2` se limita a realizar las operaciones aritméticas requeridas para calcular los resultados que devuelve en `$v0` y `$v1`. No necesita manipular la pila, ya que no escribe en ningún registro preservado entre llamadas. De hecho, sólo escribe en `$v0`, `$v1` y los registros `HI` y `LO`.

El procedimiento `proc3` tiene la peculiaridad de que recibe más de 4 argumentos. El quinto y el sexto argumentos los recibe mediante la pila, cuyo estado al comienzo del procedimiento será el siguiente:

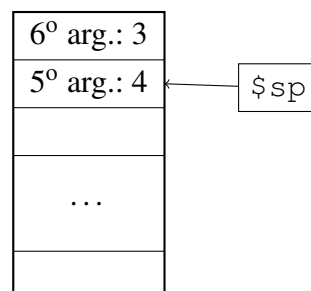
¹Aunque apilar los parámetros en orden inverso puede parecer extraño, es lo habitual en la mayoría de ABIs porque facilita la implementación de funciones que reciben un número variable de argumentos (como por ejemplo, la función `printf` de C).

```

22 # proc1: no recibe ni devuelve nada
23 proc1:
24     addi    $sp, $sp, -12
25     sw      $s1, 8($sp)
26     sw      $s0, 4($sp)
27     sw      $ra, 0($sp)
28
29     # llama a proc2(3, 4, 5)
30     li      $a0, 3
31     li      $a1, 4
32     li      $a2, 5
33     jal     proc2
34     move    $s0, $v0    # Primer valor devuelto por proc2
35     move    $s1, $v1    # Segundo valor devuelto por proc2
36
37     # imprime los resultados en orden inverso
38     move    $a0, $s1
39     jal     printlnInt
40     move    $a0, $s0
41     jal     printlnInt
42
43     # llama a proc3(8, 7, 6, 5, 4, 3)
44     li      $a0, 8
45     li      $a1, 7
46     li      $a2, 6
47     li      $a3, 5
48     # Los dos argumentos restantes se pasan usando la pila
49     addi    $sp, $sp, -8 # Hace sitio en la pila
50     li      $t0, 4
51     sw      $t0, 0($sp) # Quinto argumento: 4
52     li      $t0, 3
53     sw      $t0, 4($sp) # Sexto argumento: 3
54     jal     proc3
55     addi    $sp, $sp, 8  # Devuelve el espacio a la pila
56
57     # imprime el resultado
58     move    $a0, $v0
59     jal     printlnInt
60
61     lw      $ra, 0($sp)
62     lw      $s0, 4($sp)
63     lw      $s1, 8($sp)
64     addi    $sp, $sp, 12
65     jr      $ra

```

Figura 3.1: procedimientos.s – proc1



```

68 # proc2: recibe tres enteros (x, y, z) y devuelve dos (x + y + z, x * y * z)
69 proc2:
70     add    $v0, $a0, $a1
71     add    $v0, $v0, $a2
72     mul    $v1, $a0, $a1
73     mul    $v1, $v1, $a2
74     jr     $ra

```

Figura 3.2: procedimientos.s – proc2

```

77 # proc3: recibe seis enteros y devuelve uno (la suma de todos)
78 proc3:
79     add    $v0, $a0, $a1
80     add    $v0, $v0, $a2
81     add    $v0, $v0, $a3
82     lw     $t0, 0($sp)    # El quinto parámetro está en la cima de la pila
83     add    $v0, $v0, $t0
84     lw     $t1, 4($sp)    # Y el sexto parámetro en la siguiente posición
85     add    $v0, $v0, $t1
86     jr     $ra

```

Figura 3.3: procedimientos.s – proc3

```

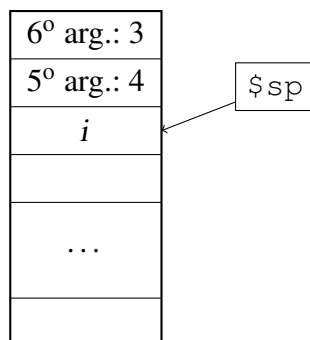
89 # imprime un entero (recibido en $a0) y un retorno de carro
90 printlnInt:
91     li     $v0, 1 # el entero a imprimir ya se encuentra en $a0
92     syscall
93     la     $a0, msg_cr # dirección de la cadena "\n"
94     li     $v0, 4
95     syscall
96     jr     $ra

```

Figura 3.4: procedimientos.s – printlnInt

Es decir, `proc3` debe acceder a 0 (`$sp`) para leer su quinto argumento y a 4 (`$sp`) para el sexto. Los cuatro primeros argumentos estarán, como es habitual, en los registros `$a0`, `$a1`, `$a2` y `$a3`.

Sin embargo, en estos casos se ha de tener en cuenta que si `proc3` necesita apilar algo, la posición de los argumentos respecto de `$sp` cambiará. Por ejemplo, el estado de la pila si apilamos un entero *i* será:



y a partir de ese momento `proc3` deberá acceder a 4 (`$sp`) para leer su quinto argumento y a 8 (`$sp`) para el sexto.

Alternativamente, se puede utilizar el registro `$fp` para almacenar el valor de `$sp` a la entrada del procedimiento y usar dicho registro como base para acceder a los parámetros extra (o variables locales) con desplazamientos estables independientemente de lo que se apile en el cuerpo del procedimiento. En este último caso, habrá que guardar previamente el valor de `$fp` en la pila porque es un registro preservado entre llamadas.

En algunos casos, podríamos pensar que no es necesario seguir estrictamente los convenios de llamada a procedimiento y las reglas de uso de la pila. Por ejemplo, `proc1` guarda en la pila el valor de los registros `s0` y `s1` antes de modificarlos pero el procedimiento `main` que llama a `proc1` no usa dichos registros, por lo que no se vería afectado si `proc1` no los guardara. Sin embargo, si hiciéramos eso, `proc1` no podría ser llamado desde otro procedimiento que sí usara esos registros (reduciendo la reusabilidad) y, si en algún momento modificáramos el procedimiento `main`, tendríamos que recordar no usar esos registros (arriesgándonos a introducir un error en caso contrario).

Es importante recordar que es necesario cumplir las convenciones de llamadas siempre, incluso aunque en algunos casos sepamos que el programa funcionaría correctamente sin ellas. Estas convenciones aseguran que procedimientos escritos por distintos programadores (o compilados por distintos compiladores) pueden llamarse unos a otros correctamente.

3.5 EJERCICIOS

Los ficheros `programa-ejercicio.c` y `programa-ejercicio.s` son dos versiones incompletas del mismo programa de ejemplo: una en C y otra en ensamblador del MIPS. El programa, una vez completo, mostrará un menú donde se nos ofrecerán tres opciones:

1. Comparar los elementos del vector con un escalar. Esta opción comparará cada elemento de un vector con un escalar que se pide por teclado. Mostrará el resultado de todas las comparaciones en una sola cadena, indicando `<`, `=` o `>` en la posición de la cadena correspondiente al elemento comparado.
2. Rellenar el vector con valores aleatorios. Nos pedirá el número de elementos que debe tener el vector, y el máximo valor absoluto de los valores aleatorios que contendrá.
3. Sale del programa.

El programa utiliza un tipo de datos llamado `VectorInt` que es una estructura que contiene un array con un tamaño fijado por una constante llamada `NUM_DATOS_MAX` y una variable que guarda el número de elementos válidos que contiene el array.

En C el tipo de datos `VectorInt` es definido como sigue:

```
typedef struct {
    int tam;
    int datos[NUM_DATOS_MAX];
} VectorInt;
```

Donde `NUM_DATOS_MAX` está definida con un valor de 255 (`#define NUM_DATOS_MAX 255`). Así mismo la variable `enteros` se define de tipo `VectorInt` de la siguiente manera:

```
VectorInt enteros = {
    5,
    { 2, 7, 3344, 655, -74 }
};
```

En ensamblador una variable de este tipo se define como:

```

enteros:
    .word    5 # Número de elementos
    .word    2
    .word    7
    .word    3344
    .word    655
    .word    4294967222
    .space   1000      # (255 - 5) elementos de 4 bytes

```

El código C y ensamblador que se proporciona incluye algunos procedimientos auxiliares y de prueba, además de los que se deberán implementar en ensamblador.

A partir de toda la información anterior, se pide la realización de los siguientes ejercicios:

1. Traduzca las funciones `compara_enteros` y `compara_vector_con_escalar` que se proporcionan en C en el fichero `programa-ejercicio.c` a ensamblador en el fichero `programa-ejercicio.s`. El resultado deberá reflejar fielmente la versión en C.

- La función `compara_enteros` recibe dos enteros y devuelve -1, 0 o 1 si el primer parámetro es menor, igual o mayor que el segundo respectivamente.
- La función `compara_vector_con_escalar` recibe como parámetro un entero y no devuelve nada. Accede al vector global `enteros` y genera una cadena en la variable global `cadena_resultado` que contiene los caracteres <, = o > en la posición correspondiente al elemento del vector comparado si éste es menor, igual o mayor al parámetro recibido.

2. Implemente en `programa-ejercicio.s` la función `inicializa_vector`. Esta función no recibe ningún parámetro ni devuelve ningún valor. Se encarga de inicializar una variable global `enteros` de tipo `VectorInt` con valores aleatorios. Para ello, primero pide por teclado dos datos (mostrando mensajes adecuados para cada uno): el número de elementos (N) y el máximo valor absoluto de los valores a generar (R). Los valores generados estarán entre -R y R inclusive.

Por ejemplo, si introducimos que N vale 30 y R vale 20, se generarán 30 números entre -20 y 20.

Si el valor de N es menor que 1 o mayor que `NUM_DATOS_MAX` o si el valor de R es menor que 1 o mayor que 1500, debe mostrar un mensaje de error (diferente en cada caso) y terminar sin modificar el vector.

Para obtener números aleatorios entre 0 y R, se debe usar la función `random_int_max(R)`, la cual devolverá un valor entre 0 y R-1. Para obtener valores entre -R y R se debe generar un número aleatorio entre 0 y $2 \times R$ y después restarle R (es decir: calcular `random_int_max(2 * R + 1) - R`).

Se aconseja utilizar las funciones `print_string`, `print_integer`, `read_string` y `read_integer`.

Si no se ha cometido ningún error, el resultado de la función será que se habrá actualizado el tamaño del vector `enteros` y se habrán almacenado en él los nuevos valores.

Consejo: Compruebe que la opción 1 sigue funcionando correctamente aún después de haber utilizado la opción 2 y haber generado un vector con menos de 16 elementos. Para cada ejercicio, el portafolios deberá incluir una explicación de cómo se ha resuelto y, en caso de que se realice o modifique algún programa, el código realizado y al menos un ejemplo de su ejecución.