Estructura de Computadores (2016-2017)
Subgrupo C3
Grado en Ingeniería Informática
Universidad de Granada

Práctica 2: Programación en ensamblador x86 Linux

Mario Rodríguez Ruiz

30 de octubre de 2016

Índice

Índice de figuras

1. Diario de trabajo

1.1. Primera sesión

Finalizar el seminario de la práctica 1. Utilización del **ddd** con el programa **hola32**, para visualizar los registros y direcciones de memoria de éste. Empezar con la práctica 2a(tutorial) que incluía la descripción de las tareas a realizar de la práctica 2b. Compilación y prueba del programa **suma**.

1.2. Segunda sesión

Demostración de las instrucciones que se pueden utilizar en los distintos programas de la práctica 2.
Visualización de manuales de varias instrucciones para comprobar si modifican los flags. Comienzo del programa suma. Aprender el funcionamiento de algunas instrucciones como **adc** e **idiv** y los registros que hay disponibles para los programas. Función printf y cómo hay que alojar en la pila los valores a mostrar por pantalla, así como el formato éstos.

1.3. Tercera sesión

Realización de las preguntas teóricas así como la finalización de los ejercicios suma con signo y media.

2. Ejercicio 5.1: Suma SIN signo en 32 bits

```
1  # Mario Rodríguez Ruiz
2  # Suma de enteros sin signo de 32 bits en una plataforma de
       32 bits sin perder precisión.
3  .section .data
4      .macro linea
5          #.int 1,1,1,1
6          .int 2,2,2,2
7          #.int 1,2,3,4
8          #.int -1,-1,-1,-1
9          #.int 0xffffffff,0xffffffff,0xffffffff,0xffffffff
10         #.int 0x08000000,0x08000000,0x08000000,0x08000000
11         #.int 0x10000000,0x20000000,0x40000000,0x80000000
12     .endm
13 lista:  .irpc i,12345678
14         linea
15     .endr
16
17 # Calcula la longitud de la lista
18 longlista:  .int (.-lista)/4
```

```
19
20 resultado:  .quad 0x0123456789ABCDF    # Valor que ocupa 8B
21
22 # Formato para mostrar resultados
23 formato:  .ascii "Decimal = %lld \nHexadecimal = 0x%llx \n\0"
24
25 .section .text
26 main: .global main
27     # Se guarda en ebx la posición donde comienza la
28     # lista de enteros en memoria
29   mov $lista, %ebx
30
31   # Se guarda en ecx el numero de enteros de la lista
32   mov longlista, %ecx
33
34   # Llamada a la función suma
35   call suma
36
37   # Se guarda el valor de los 32 bit menos significativos
38   # del número calculado en suma en resultado.
39   mov %eax, resultado
40
41   # Se guarda en la posición resultado más 4B el valor de los
         32 bit
42   # más significativos del número calculado en suma.
43   mov %edx, resultado+4
44
45   # Como el sistema utilizado es little endian metemos en la
        pila los
46   # 32bit mas significativos primero de nuestro número
47   push resultado+4
48
49   # Se guardan en la pila los otros 32 bit (los menos
        significativos)
50   push resultado
51   push resultado+4
52   push resultado
53   push $formato          # Parámetros a leer en la pila
54   call printf            # Llamada a la función printf
55   mov $1, %eax           # Llamada al sistema
56   mov $0, %ebx
57   int $0x80
58
59 suma:
```

```
60    mov $0, %eax
61    mov $0, %edx
62    mov $0, %esi            # Establece el índice a 0
63 bucle:
64      # Se suma al acumulador eax los números de la lista y,
65      # como son enteros de 4B, se realiza un desplazamiento
66      # de 4 en 4 Bytes
67    add (%ebx,%esi,4), %eax
68
69    # Se suma el acarreo en el registro edx (registro para
70    # los 32bit más signficativos)
71    adc $0,%edx
72
73    # Incrementa en uno el registro esi (contador) cada vez
74    # que se suma un número
75    inc %esi
76
77    # Si el registro esi es menor que ecx (longitud de lista)
78    # salta al bucle para seguir sumando.
79    cmp  %esi,%ecx
80    jne bucle
81
82    ret
```

ejercicio5.1/suma_unsigned.s

```
[mario@manjario ejercicio5.1]$ make
cc -g -std=c11 -Wall -m32 -nostartfiles suma_unsigned.s -o suma_unsigned
/usr/bin/ld: aviso: no se puede encontrar el símbolo de entrada _start;
81f0
[mario@manjario ejercicio5.1]$ ./suma_unsigned
Decimal = 64
Hexadecimal = 0x40
[mario@manjario ejercicio5.1]$
```

Figura 2.1: Ejecución del programa "suma"SIN signo.

3. Ejercicio 5.2: Suma CON signo en 32 bits

```
1  # Mario Rodríguez Ruiz
2  # Suma de enteros con signo de 32 bits en una plataforma de
       32 bits sin perder precisión.
3  .section .data
4      .macro linea
5          #.int -1,-1,-1,-1
6          #.int 1,-2,1,-2
7          .int 1,2,-3,-4
8          #.int 0x7FFFFFFF,0x7FFFFFFF,0x7FFFFFFF,0x7FFFFFFF
9          #.int 0X80000000,0X80000000,0X80000000,0X80000000
10         #.int 0X04000000,0X04000000,0X04000000,0X04000000
11         #.int 0X08000000,0X08000000,0X08000000,0X08000000
12         #.int 0xFC000000,0xFC000000,0xFC000000,0xFC000000
13         #.int 0xF8000000,0xF8000000,0xF8000000,0xF8000000
14         #.int 0xF0000000,0xE0000000,0xE0000000,0xD0000000
15     .endm
16 lista:  .irpc i,12345678
17         linea
18     .endr
19
20 # Calcula la longitud de la lista
21 longlista:  .int (.-lista)/4
22 resultado:  .quad 0x0123456789ABCDF    # Valor que ocupa 8B
23 # Formato para mostrar resultados
24 formato:  .ascii "Decimal = %lld \nHexadecimal = 0x%llx \n\0"
25 .section .text
26 main: .global main
27     # Se guarda en ebx la posición donde comienza la
28     # lista de enteros en memoria
29   mov    $lista, %ebx
30
31   # Se guarda en ecx el numero de enteros de la lista
32   mov longlista, %ecx
33
34   # Llamada a la función suma
35   call suma
36
37   # Se guarda el valor de los 32 bit menos significativos
38   # de nuestro número calculado en suma en resultado.
39   mov %eax, resultado
40
```

```
41    # Se guarda en la posición resultado más 4B el valor de los
          32 bit
42    # más significativos de nuestro número calculado en suma.
43    mov %edx, resultado+4
44
45      # Como el sistema utilizado es little endian metemos en
            la pila los
46    # 32bit mas significativos primero de nuestro número
47    push resultado+4
48
49    # Se guardan en la pila los otros 32 bit (los menos
         significativos)
50    push resultado
51    push resultado+4
52    push resultado
53    push $formato       # Parámetros a leer en la pila
54    call printf         # Llamada a la función printf
55    mov $1, %eax        # Llamada al sistema
56    mov $0, %ebx
57    int $0x80
58
59  suma:
60    mov $0, %ebp
61    mov $0, %edi
62    mov $0, %esi     # Establece el índice a 0
63  bucle:
64      # Se guarda en eax cada número de la lista
65    mov (%ebx,%esi,4), %eax
66
67    # Se extiende el valor de eax a un número utilizando tambié
         n el registro edx para el signo
68    cltd
69
70    # Se suma el valor existente en eax con el valor de edi
         manteniendo la suma en él
71    add %eax,%edi
72
73    # Se suman los bits de signo de edx con el valor de ebp más
          el acarreo si lo hubiera.
74    adc %edx,%ebp
75
76      # Incrementa en uno el registro esi (contador) cada vez
77    # que se suma un número
78    inc %esi
```

```
79
80    # Si el registro esi es menor que ecx (longitud de lista)
81    # salta al bucle para seguir sumando.
82    cmp %esi,%ecx
83    jne bucle
84
85    # Se mueven los bits de signo a edx que son los 32 bits más
          significativos
86    mov %ebp,%edx
87
88    # Se mueve la suma total al acumulador eax
89    mov %edi,%eax
90    ret
```

ejercicio5.2/suma_signo.s

```
[mario@manjario ejercicio5.2]$ make
cc -g -std=c11 -Wall -m32 -nostartfiles suma_signo.s -o suma_signo
/usr/bin/ld: aviso: no se puede encontrar el símbolo de entrada _start;
81f0
[mario@manjario ejercicio5.2]$ ./suma_signo
Decimal = -32
Hexadecimal = 0xffffffffffffffe0
[mario@manjario ejercicio5.2]$
```

Figura 3.1: Ejecución del programa suma CON signo.

4. Ejercicio 5.3: Media CON signo en 32 bits

```
1  # Mario Rodríguez Ruiz
2  # Media de enteros con signo de 32 bits en una plataforma de
       32 bits sin perder precisión.
3  .section .data
4      .macro linea
5          #.int 0,-2,-1,-1
6          #.int 1,-2,1,-2
7          .int 1,2,-3,-4
8          #.int 0x7FFFFFFF,0x7FFFFFFF,0x7FFFFFFF,0x7FFFFFFF
9          #.int 0X80000000,0X80000000,0X80000000,0X80000000
10         #.int 0xF0000000,0xE0000000,0xE0000000,0xD0000000
11     .endm
12
13 lista:
14     linea
15     .irpc i,1234567
16         linea
17     .endr
18
19 # Calcula la longitud de la lista
20 longlista:  .int (.-lista)/4
21
22 media: .int 0
23 resto: .int 0
24 # Formato para mostrar resultados
25 formato: .ascii "Media decimal = %d \nMedia hexadecimal = 0x%
       x \n"
26       .ascii "Resto decimal = %d \nResto hexadecimal = 0x%x \n
           \0"
27 .section .text
28 main: .global main
29
30   # Se guarda en ebx la posición donde comienza la
31     # lista de enteros en memoria
32   mov    $lista, %ebx
33
34   # Se guarda en ecx el numero de enteros de la lista
35   mov longlista, %ecx
36
37   # Llamada a la función suma
38   call suma
```

```
39
40    # Se guarda el valor de la media de todos los elementos en
          eax.
41    mov %eax,media
42
43      # Se guarda el valor del resto del resultado de hacer la
            media en la variable edx
44    mov %edx,resto
45    # Se incluye dos veces el resto y la media
46      # porque se mostrará en decimal y hexadecimal.
47    push resto
48    push resto
49    push media
50    push media
51    push $formato       # Parámetros a leer en la pila
52    call printf         # Llamada a la función printf
53    mov $1, %eax        # Llamada al sistema
54    mov $0, %ebx
55    int $0x80
56
57  suma:
58    mov $0, %ebp
59    mov $0, %edi
60    mov $0, %esi        # Establece el índice a 0
61  bucle:
62    # Se guarda en eax cada número de la lista
63    mov (%ebx,%esi,4), %eax
64
65    # Se extiende el valor de eax a un número utilizando tambié
          n el registro edx para el signo
66    cltd
67
68    # Se suma el valor existente en eax con el valor de edi
          manteniendo la suma en él
69    add %eax,%edi
70
71    # Se suman los bits de signo de edx con el valor de ebp más
            el acarreo si lo hubiera.
72    adc %edx,%ebp
73
74      # Incrementa en uno el registro esi (contador) cada vez
75    # que se suma un número
76    inc %esi
77
```

```
78  # Si el registro esi es menor que ecx (longitud de lista)
79  # salta al bucle para seguir sumando.
80  cmp %esi,%ecx
81  jne bucle
82
83  # Se mueven los bits de signo a edx que son los 32 bits más
        significativos
84  mov %ebp,%edx
85
86  # Se mueve la suma total al acumulador eax
87  mov %edi,%eax
88
89  # Se divide el número alojado en entre el numero de
       elementos de la lista(ecx),
90    # El cociente se guarda en eax y el resto en edx.
91  idiv %ecx
92  ret
```

ejercicio5.3/media.s

```
[mario@manjario ejercicio5.3]$ make
cc -g -std=c11 -Wall -m32 -nostartfiles media.s -o media
/usr/bin/ld: aviso: no se puede encontrar el símbolo de entrada _start;
81f0
[mario@manjario ejercicio5.3]$ ./media
Media decimal = -1
Media hexadecimal = 0xffffffff
Resto decimal = 0
Resto hexadecimal = 0x0
[mario@manjario ejercicio5.3]$
```

Figura 4.1: Ejecución del programa media con signo.

5. Preguntas de Auto-comprobación

5.1. Sesión de depuración saludo.s

1. ***¿Qué contiene EDX tras ejecutar mov longsaludo, %edx? ¿Para qué necesitamos esa instrucción, o ese valor? Responder no sólo el valor concreto (en decimal y hex) sino también el significado del mismo (¿de dónde sale?). Comprobar que se corresponden los valores hexadecimal y decimal mostrados en la ventana Status->Registers.***

   EDX, tras ejecutar **mov longsaludo**, contiene el valor 28 (hexadecimal 0x1C) como puede apreciarse en la Figura 5.1

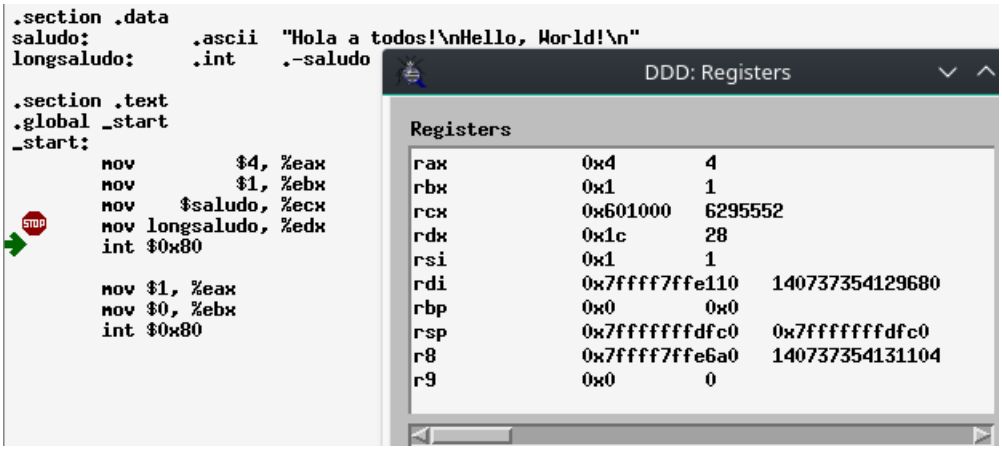


Figura 5.1: Depuración de saludo.s

EDX contiene este valor porque al alojarle el valor de la variable **longsaludo** que es el contador de posiciones de saludo: el número de bytes que ocupará. Es por ello que se necesite el uso de este valor ya que por medio ahora de esta instrucción se podrá indicar la que se escribirán 28 bytes.

2. ***¿Qué contiene ECX tras ejecutar mov $saludo, %ecx? Indicar el valor en hexadecimal, y el significado del mismo.***

   Como muestra la Figura 5.1, ECX contiene el valor hexadecimal 0x601000. Este valor define la dirección en memoria del contenido de la cadena **saludo**.

3. ***¿Qué sucede si se elimina el símbolo de dato inmediato ($) de la instrucción anterior? (mov saludo, %ecx). Realizar la modificación, indicar el contenido de ECX en hexadecimal, explicar por qué no es lo mismo en ambos casos. Concretar de dónde viene el nuevo valor (obtenido sin usar $).***

   Si se elimina el símbolo de dato inmediato $ de la instrucción **mov $saludo, %ecx** en vez de mover la dirección de memoria de **saludo** al registro **ecx**, será el contenido

de **saludo** el que se guarde en dicho registro.

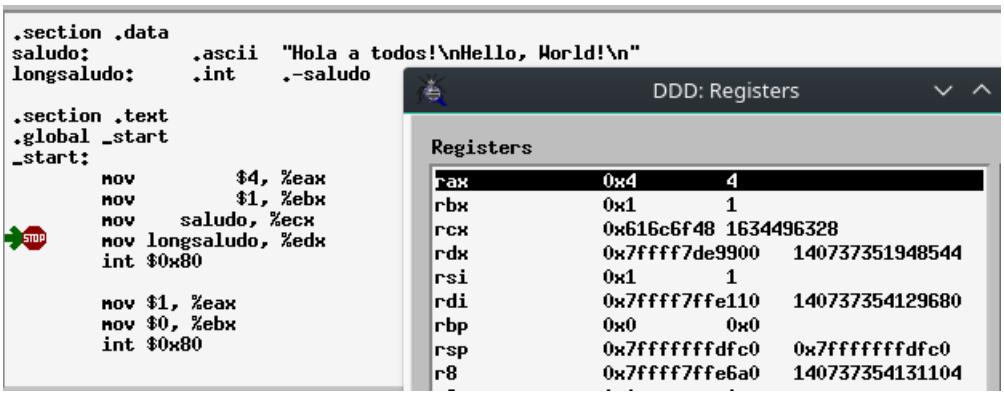


Figura 5.2: Depuración de saludo.s

Como se observa en la Figura 5.2, el valor de **ECX** ahora es **0x616c6f48**, que es la representación en hexadecimal de la cadena de texto almacenada en **saludo**.

4. ***¿Cuántas posiciones de memoria ocupa la instrucción mov $1, %ebx? ¿Cómo se ha obtenido esa información? Indicar las posiciones concretas en hexadecimal.***

   La instrucción mov $1, %ebx ocupa cinco posiciones de memoria.

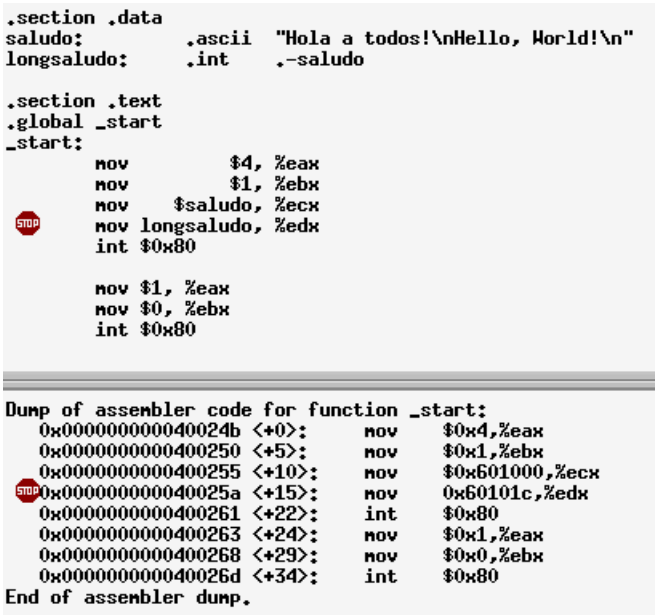


Figura 5.3: Desensamblado de saludo.s

Como se puede observar en la Figura 5.3 su posición se encuentra desde la dirección 0x0400250 hasta la 0x0400255. Esta información se obtiene desde la opción **"View ->Machine Code Window"** de **ddd**.

5. ***¿Qué sucede si se elimina del programa la primera instrucción int 0x80? ¿Y si se elimina la segunda? Razonar las respuestas***

   Si se elimina la primera instrucción **int 0x80** el programa finaliza aparentemente sin errores, sin embargo no se imprimen los mensajes por pantalla. Este hecho se produce porque con la instrucción **mov $4, %eax** se realiza una llamada a **write** (encargado de imprimir mensajes en pantalla). Ésta se pierde con la instrucción siguiente **mov $1, %eax**.

   Si eliminamos la segunda instrucción **int 0x80**, en esta ocasión sí que se producen errores en la finalización de la ejecución del programa, concretamente un fallo de segmentación. Este hecho se produce porque con la instrucción **mov $1, %eax** se realiza una llamada a **exit** (encargado de finalizar la ejecución de un programa) Al no producirse esta llamada el programa no sabe cómo establecer el final de la ejecución y produce un fallo de segmentación.

6. ***¿Cuál es el número de la llamada al sistema READ (en kernel Linux 32bits)? ¿De dónde se ha obtenido esa información?***

   El número de la llamada al sistema READ (en kernel Linux 32bits) es el 3.

```
[mario@manjario practica2]$ cat /usr/include/asm/unistd_32.h
#ifndef _ASM_X86_UNISTD_32_H
#define _ASM_X86_UNISTD_32_H 1

#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
```

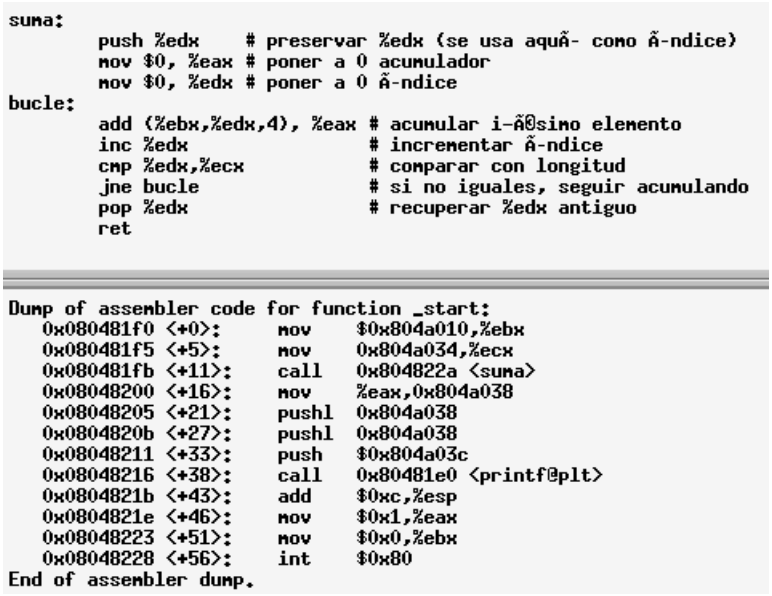
Figura 5.4: Llamadas al sistema

   Esta información puede obtenerse, por ejemplo, en el archivo **unistd_32.h** del directorio /**usr**/**include**/**asm**/ como muestra la Figura 5.4

5.2. Sesión de depuración suma.s

1. ***¿Qué dirección se le ha asignado a la etiqueta suma? ¿Y a bucle? ¿Cómo se ha obtenido esa información?***

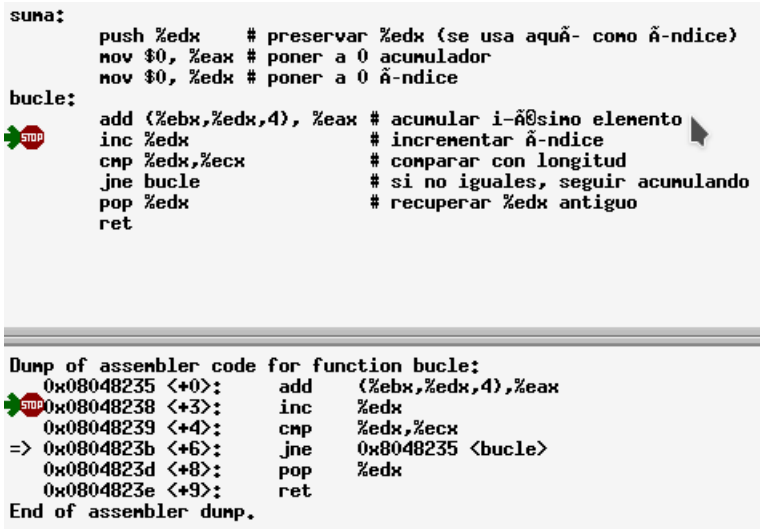
   La dirección que se le ha asignado a la etiqueta **suma** es **0x804822a**(Figura 5.5) y **0x8048235** a **bucle** (Figura 5.6).



```
suma:
        push %edx    # preservar %edx (se usa aquÃ- como Ã-ndice)
        mov $0, %eax # poner a 0 acumulador
        mov $0, %edx # poner a 0 Ã-ndice
bucle:
        add (%ebx,%edx,4), %eax # acumular i-Ã®simo elemento
        inc %edx                # incrementar Ã-ndice
        cmp %edx,%ecx           # comparar con longitud
        jne bucle               # si no iguales, seguir acumulando
        pop %edx                # recuperar %edx antiguo
        ret
```

```
Dump of assembler code for function _start:
   0x080481f0 <+0>:     mov    $0x804a010,%ebx
   0x080481f5 <+5>:     mov    0x804a034,%ecx
   0x080481fb <+11>:    call   0x804822a <suma>
   0x08048200 <+16>:    mov    %eax,0x804a038
   0x08048205 <+21>:    pushl  0x804a038
   0x0804820b <+27>:    pushl  0x804a038
   0x08048211 <+33>:    push   $0x804a03c
   0x08048216 <+38>:    call   0x80481e0 <printf@plt>
   0x0804821b <+43>:    add    $0xc,%esp
   0x0804821e <+46>:    mov    $0x1,%eax
   0x08048223 <+51>:    mov    $0x0,%ebx
   0x08048228 <+56>:    int    $0x80
End of assembler dump.
```

Figura 5.5: Desesamblado de suma.s



```
suma:
        push %edx    # preservar %edx (se usa aquÃ- como Ã-ndice)
        mov $0, %eax # poner a 0 acumulador
        mov $0, %edx # poner a 0 Ã-ndice
bucle:
        add (%ebx,%edx,4), %eax # acumular i-Ã®simo elemento
        inc %edx                # incrementar Ã-ndice
        cmp %edx,%ecx           # comparar con longitud
        jne bucle               # si no iguales, seguir acumulando
        pop %edx                # recuperar %edx antiguo
        ret
```

```
Dump of assembler code for function bucle:
   0x08048235 <+0>:     add    (%ebx,%edx,4),%eax
   0x08048238 <+3>:     inc    %edx
   0x08048239 <+4>:     cmp    %edx,%ecx
=> 0x0804823b <+6>:     jne    0x8048235 <bucle>
   0x0804823d <+8>:     pop    %edx
   0x0804823e <+9>:     ret
End of assembler dump.
```

Figura 5.6: Desesamblado de suma.s

Esta información se ha obtenido desde la opción **"View ->Machine Code Window"** de **ddd**.

2. ***¿Para qué usa el procesador los registros EIP y ESP?***

   El procesador usa el registro **EIP** como puntero a la siguiente dirección de memoria que va a ejecutar y el registro **ESP** lo usa como puntero al inicio de la pila del

programa.

3. ***¿Qué registros modifica la instrucción CALL? Explicar por qué necesita CALL modificar esos registros.***

   La instrucción **CALL** modifica los registros **EAX** (al funcionar como acumulador debe ir modificándose tras cambiar el resultado de la suma), **EDX** (se modifica porque es un índice que muestra la siguiente dirección de memoria a leer), **ESP** (es modificado ya que en él se alojará la dirección de retorno al pasar por RET) y **EIP** y EIP (se altera porque en él se encuentra la dirección de la instrucción siguiente que se va a ejecutar).

4. ***¿Qué registros modifica la instrucción RET? Explicar por qué necesita RET modificar esos registros.***

   La instrucción **RET** modifica los registros **ESP** y **EIP** ya que éstos actúan como punteros y deben cambiar su contenido para que el programa pueda seguir con su ejecución desde el punto en el que se encontraba antes de llamar a la subrutina.

5. ***¿Qué ocurriría si se eliminara la instrucción RET? Razonar la respuesta. Comprobarlo usando ddd.***

   Si se eliminara la instrucción **RET** sería imposible devolver la ejecución al programa principal desde el que se llamó a la subrutina. Esto daría lugar a un error en la ejecución del programa provocando un fallo de segmentación.