

# PRÁCTICA 2

## Programación en ensamblador x86 Linux

### SESIÓN DE DEPURACIÓN DE `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 corresponde con los valores hexadecimal y decimal mostrados en la ventana Status->Registers.

Contiene el valor 28 en decimal, que equivale a 0x1c en hexadecimal, siendo este el tamaño del mensaje *saludo*, el número de bytes a escribir, es decir, el número total de letras, espacios y salto de línea (\n) que forman *saludo*.

Este valor es necesario para poder llamar a `write(stdout, &saludo, longsaludo)`, escribiendo *longsaludo* bytes a partir de *&saludo* en el descriptor para la salida estándar (*STDOUT\_FILENO*).

2. ¿Qué contiene ECX tras ejecutar `mov $saludo,%ecx`? Indicar el valor en hexadecimal, y el significado del mismo. Realizar un dibujo a escala de la memoria del programa, indicando dónde empieza el programa (*\_start*, *.text*), dónde empieza *saludo* (*.data*) y dónde está el tope de pila (*%esp*).

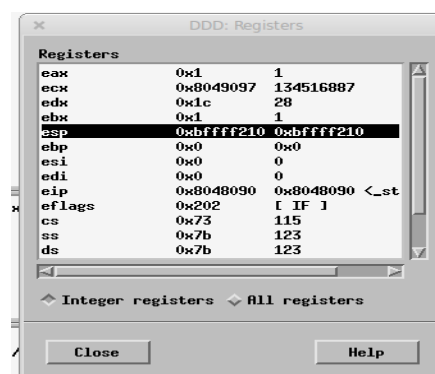
ECX contiene el valor 0x8049097, que es la dirección del texto a escribir.

```
Dump of assembler code for function _start:
0x08048074 <+0>:    nov    $0x4,%eax
0x08048079 <+5>:    nov    $0x1,%ebx
0x0804807e <+10>:   nov    $0x8049097,%ecx
0x08048083 <+15>:   nov    $0x80490b3,%edx
0x08048089 <+21>:   int     $0x80
0x0804808b <+23>:   nov    $0x1,%eax
0x08048090 <+28>:   nov    $0x0,%ebx
0x08048095 <+33>:   int     $0x80
End of assembler dump.
```

El programa *\_start* empieza en 0x08048074

Saludo empieza en 0x8049097

```
(gdb) x /32cb $saludo
0x8049097:  72 'H' 111 'o' 108 'l' 97 'a' 32 ' ' 97 'a' 32 ' ' 116 't'
0x804909f:  111 'o' 100 'd' 111 'o' 115 's' 33 '!' 10 '\n' 72 'H' 101 'e'
0x80490a7:  108 'l' 108 'l' 111 'o' 44 ',' 32 ' ' 87 'H' 111 'o' 114 'r'
0x80490af:  108 'l' 100 'd' 33 '!' 10 '\n' 28 '\034' 0 '\000' 0 '\000'
(gdb)
```



El tope de la pila está en 0xbffff210

## Programación en ensamblador x86 Linux

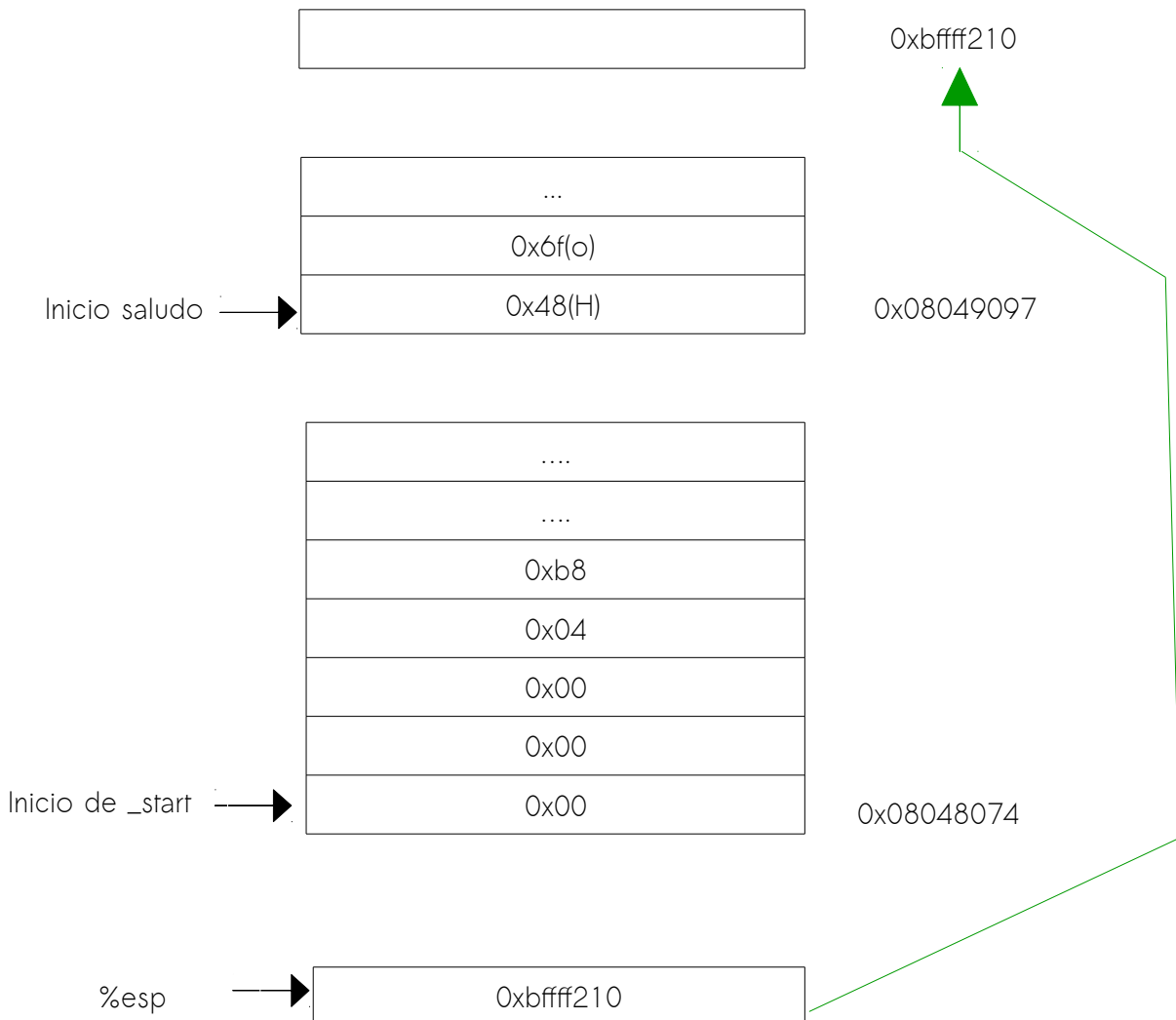
Desensamblado de la sección .text:

```
+ X ...Practicas/Practica 1
mar@mar-SATELLITE-L750:~/Dropbox/Ugr/2º Curso de Doble Grado/1º cuatrimestre
/EC/documentos/Practicas/Practica 1$ objdump -d saludo.o

saludo.o:      formato del fichero elf32-i386

Desensamblado de la sección .text:

00000000 <_start>:
  0:  b8 04 00 00 00      mov     $0x4,%eax
  5:  bb 01 00 00 00      mov     $0x1,%ebx
  a:  b9 00 00 00 00      mov     $0x0,%ecx
  f:  8b 15 1c 00 00 00    mov     0x1c,%edx
 15:  cd 80               int     $0x80
 17:  b8 01 00 00 00      mov     $0x1,%eax
 1c:  bb 00 00 00 00      mov     $0x0,%ebx
 21:  cd 80               int     $0x80
mar@mar-SATELLITE-L750:~/Dropbox/Ugr/2º Curso de Doble Grado/1º cuatrimestre
/EC/documentos/Practicas/Practica 1$ _
```



## Programación en ensamblador x86 Linux

3. ¿Qué sucede si se elimina el símbolo de dato inmediato (\$) de la instrucción anterior? (*mov saludo, %ecx*) Realiza 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 \$ quedando *mov saludo,%ecx*, el valor de ECX se modifica siendo este 0x616c6f48 ya que interpreta string como un entero, en cambio \$saludo indica una dirección en memoria.

4. ¿Cuántas posiciones de memoria ocupa la variable *longsaludo*? ¿Y la variable *saludo*? ¿Cuántos bytes ocupa por tanto la sección de datos? Comprobar con un volcado Data->Memory mayor que la zona de datos antes de hacer Run.

La variable *longsaludo* ocupa 1 bytes y *saludo* 28 bytes, por tanto, la sección de datos ocupa en total 29 bytes.

5. Añadir dos volcados Data->Memory de la variable *longsaludo*, uno como entero hexadecimal, y otro como 4 bytes hex. Teniendo en cuenta lo mostrado en esos volcados... ¿Qué dirección de memoria ocupa *longsaludo*? ¿Cuál byte está en la primera posición, el más o menos significativo? ¿Los procesadores de la línea x86 usan el criterio de extremo mayor(big-endian) o menos(little-endian)? Razonar la respuesta.

La variable *longsaludo* ocupa la dirección de memoria 0x80490b3.

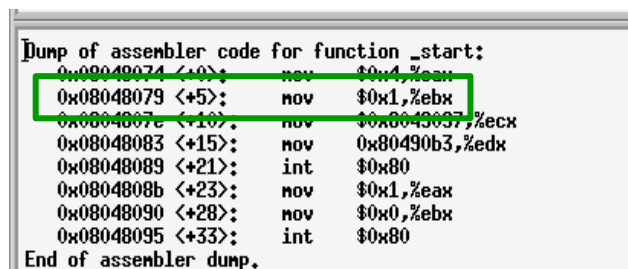
En la primera posición se encuentra el menos significativo al usar el criterio de extremo menor (little-endian).

Los procesadores de la línea x86 usan el criterio de extremo menor (little-endian) ya que el byte de menor peso se almacena en la dirección más baja de memoria y el byte más significativo en la más alta.

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

Ocupa 5 posiciones de memoria:

0x08048079  
0x0804807a  
0x0804807b  
0x0804807c  
0x0804807d



```
Dump of assembler code for function _start:
0x08048074 <+0>:  mov    $0x4,%eax
0x08048079 <+5>:  mov    $0x1,%ebx
0x0804807e <+10>:  mov    $0x0490b3,%ecx
0x08048083 <+15>:  mov    0x80490b3,%edx
0x08048089 <+21>:  int     $0x80
0x0804808b <+23>:  mov    $0x1,%eax
0x08048090 <+28>:  mov    $0x0,%ebx
0x08048095 <+33>:  int     $0x80
End of assembler dump.
```

Se ha obtenido a través de ddd, escribiendo *disassemble* en *View->GDB Console* o equivalentemente en *View->Machine Code Window*

## Programación en ensamblador x86 Linux

7. ¿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` no se realizaría la escritura de `&saludo` en el descriptor para la salida estándar (`STDOUT_FILENO`) ya que no se llamaría a `write(stdout, &saludo, longsaludo)` que es lo que realiza esta primera instrucción, al realizarse las anteriores sentencias:

```
mov $4, %eax      # write
mov $1, %ebx      # fd: descriptor del texto a escribir
mov $saludo, %ecx  # dirección del texto a escribir
mov $longsaludo, %edx # número de bytes a escribir
```

Si se elimina la segunda no se terminaría el programa ya que esta segunda instrucción llama a `exit(0)` al realizarse la instrucciones:

```
mov $1, %eax      # exit
mov $0, %ebx      # código a retornar, en este caso 0
```

8. ¿Cuál es el número de la llamada al sistema `READ`(en kernel Linux 32bits)?

El número de la llamada al sistema `READ` es 3.

## Programación en ensamblador x86 Linux

### SESIÓN DE DEPURACIÓN suma.s

1. ¿Cuál es el contenido de EAX justo antes de ejecutar la instrucción RET, para esos componentes de lista concretos? Razonar la respuesta, incluyendo cuanto valen  $0b10$ ,  $0x10$  y  $(.-lista)/4$

El contenido de EAX es 37 que es el resultado de sumar todos los elementos de la lista.

Sabiendo que:

$0b10 = 2$

$0x10 = 16$

$(.-lista)/4 = 9$  (Número total de elementos de la lista)

$1+2+10+1+2+2(0b10)+1+2+16(0x10) = 37$ , que equivale a 25 en hexadecimal.

2. ¿Qué valor en hexadecimal se obtiene en *resultado* si se usa la lista de 3 elementos: `.int 0xffffffff, 0xffffffff, 0xffffffff`? ¿Por qué es diferente del que se obtiene haciendo la suma a mano? NOTA: Indicar qué valores va tomando EAX en cada iteración del bucle, como los muestra la ventana Status->Registers, en hexadecimal y decimal (con signo). Fijarse también si se van activando los flags CF y OF o no tras cada suma. Indicar también qué valor muestra *resultado* si se vuelca con Data->Memory como decimal (con signo) o unsigned (sin signo).

Se obtiene resultado = `0xffffffffd` si se usa la lista de 3 elementos especificada en el enunciado.

Es diferente de la suma haciéndola a mano que nos da que resultado = `0x2ffffffffd` porque resultado es una variable de 32 bits, por tanto el acarreo generado al hacer la suma se pierde.

Valores de EAX en cada iteración:

1º) EAX = `0xffffffff` = -1

2º) EAX = `0xffffffe` = -2

3º) EAX = `0xffffffd` = -3

Decimal (Con signo)

→ resultado = -3

Unsigned (Sin signo):

→ resultado = 4294967293

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

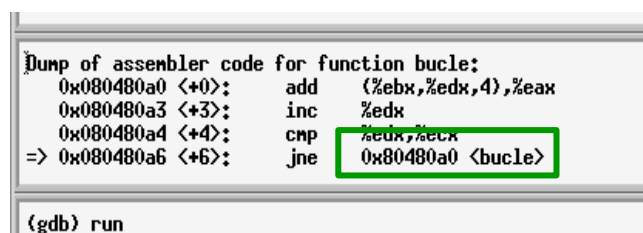
Se le ha asignado la dirección `0x8048095` a la etiqueta *suma*.

```
~5~
Dump of assembler code for function _start:
0x08048074 <+0>:  mov     $0x80490aa,%ebx
0x08048079 <+5>:  mov     0x00430000,%ecx
0x0804807f <+11>: call    0x8048095 <suma>
0x08048084 <+16>:  mov     %eax,%eax

(gdb) print suma
$1 = {<text variable, no debug info>} 0x8048095 <suma>
(gdb) watch suma
```

## Programación en ensamblador x86 Linux

Se le ha asignado la dirección 0x80480a0 a la etiqueta *bucle*.



```
Dump of assembler code for function bucle:
0x080480a0 <+0>:  add    (%ebx,%edx,4),%eax
0x080480a3 <+3>:  inc     %edx
0x080480a4 <+4>:  cmp     %edx,%ecx
=> 0x080480a6 <+6>:  jne     0x80480a0 <bucle>

(gdb) run
```

La información se ha obtenido a través de la ventana de Machine Code Window.

### 4. ¿Para qué usa el procesador los registros EIP y ESP?

EIP la usa el procesador para saber la dirección donde se encuentra la instrucción que se está ejecutando en cada momento.

ESP la utiliza para saber donde está el tope de pila, ya que apunta al último elemento insertado.

### 5. ¿Cuál es el valor de ESP antes de ejecutar CALL, y cuál antes de ejecutar RET? ¿En cuánto se diferencian ambos valores? ¿Por qué? ¿Cuál de los dos valores de ESP apunta a algún dato de interés para nosotros? ¿Cuál es ese dato?

Antes de ejecutar CALL el valor de ESP = 0xbffff220 y antes de RET, ESP = 0xbffff21c, por tanto la diferencia entre ambos es 4 ya que al ejecutar CALL es como si se hiciera un PUSH EIP, por tanto, se decrementa ESP en el número de posiciones de memoria que ocupa el dato a insertar, en este caso, 4 pero también se ejecuta la instrucción PUSH entre estas dos instrucciones volviéndose a decrementar ESP otros 4, y POP justo antes de RET, por tanto, se incrementa 4. Luego la diferencia es 4.

El segundo valor de ESP es de interés para nosotros ya que apunta a la dirección donde se encuentra EIP, para poder retornar posteriormente de la subrutina.

### 6. ¿Qué registros modifica la instrucción CALL? Explica porque necesita CALL modificar esos registros.

CALL modifica EIP ya que guarda la dirección de retorno en la pila antes de saltar a la subrutina indicada como argumento.

### 7. ¿Qué registros modifica la instrucción RET? Explica porque necesita RET modificar esos registros.

RET modifica EIP (equivale a POP EIP) para recuperar de pila la dirección de retorno.

## Programación en ensamblador x86 Linux

8. Indicar qué valores se introducen en la pila durante la ejecución del programa, y en qué direcciones de memoria queda cada uno. Realizar un dibujo de la pila con dicha información. NOTA: en los volcados Data->Memory se puede usar \$esp para referirse a donde apunta el registro ESP.

Se introducen en pila primero con la instrucción CALL (PUSH EIP), la dirección de retorno, que en este caso es, 0x08048084 (*mov %eax, resultado*) en la dirección de memoria 0xbffff21c. Y con la instrucción PUSH, se introduce EDX en la dirección 0xbffff218.

9. ¿Cuántas posiciones de memoria ocupa la instrucción *mov \$0, %edx*? ¿Y la instrucción *inc %edx*? ¿Cuáles son sus respectivos códigos máquina? Indicar como se ha obtenido. NOTA: en los volcados Data->Memory se puede usar una dirección hexadecimal 0x... para indicar la dirección del volcado. Recordar la ventana View->Machine Code Window. Recordar también la herramienta objdump.

La instrucción *mov \$0, %edx* ocupa 5 posiciones de memoria ya que se almacena en pila desde la dirección 0x0804809b hasta 0x080480a0 e *inc %edx* ocupa 1 ya que solo se almacena en 0x080480a3.

Se ha obtenido a través de la ventana Machine Code Window.

Código máquina:

- *mov \$0, %edx* = ba 00 00 00 00
- *inc %edx* = 42

```
miniaspireone@miniaspireone-Aspire-one ~/Dropbox/Ugr/2º Curso de Doble Grado/1º
cuatrimestre/EC/documentos/Practicas/practica 2 $ objdump -d suma.o
suma.o:      formato del fichero elf32-i386

Desensamblado de la sección .text:
00000000 <start>:
0:  bb 00 00 00 00      mov     $0x0,%ebx
5:  8b 0d 0c 00 00 00    mov     0xc,%ecx
b:  e8 11 00 00 00      call   21 <suma>
10: a3 10 00 00 00      mov     %eax,0x10
15: b8 01 00 00 00      mov     $0x1,%eax
1a: bb 00 00 00 00      mov     $0x0,%ebx
1f:  cd 80               int     $0x80
00000021 <suma>:
21:  52                 push    %edx
22:  b8 00 00 00 00      mov     $0x0,%eax
27:  ba 00 00 00 00      mov     $0x0,%edx
0000002c <bucle>:
2c:  03 04 93           add     (%ebx,%edx,4),%eax
2f:  42                 inc     %edx
30:  39 d1              cmp     %edx,%ecx
32:  75 f8              jne     2c <bucle>
34:  5a                 pop     %edx
35:  c3                 ret
```

10. ¿Qué ocurriría si se eliminara la instrucción RET? Razonar la respuesta. Comprobar usando ddd.

Si se eliminara la instrucción RET no se podría recuperar de pila la dirección de retorno por lo tanto continuaría el programa, pero se encontraría con que las direcciones siguientes no tienen instrucciones. Luego el programa no podría continuar aunque tampoco podría finalizar de forma normal.

## Programación en ensamblador x86 Linux

### CUESTIONES SOBRE suma64signed.s

1. Para  $N=32$ , ¿cuántos bits adicionales pueden llegar a necesitarse para almacenar el resultado? Dicho resultado se alcanzaría cuando todos los elementos tomaran el valor máximo sin signo. ¿Cómo se escribe ese valor en hexadecimal? ¿Cuántos acarrees se producen? ¿Cuánto vale la suma (indicarla en hexadecimal)? Comprobarlo usando ddd.

El máximo valor sin signo es  $0xffffffff = 2147483647$ .

Con este valor se producen 15 acarrees, provocándose estos cuando el contador es impar, es decir, en 3,5,7,...,31, valiéndose la suma  $0xffffffff0$ . Por tanto, en este caso se necesita 1 bits más para almacenar el resultado, pero si realizamos la suma con  $0xffffffff$  se necesitarían 2 bits, resultando la suma  $0x1ffffffff0$ .

2. Si nos proponemos obtener sólo 1 acarreo con una lista de 32 elementos iguales, el objetivo es que la suma alcance  $2^{32}$  (que ya no cabe en 32 bits). Cada elemento debe valer por tanto  $2^{32}/32 = 2^{32}/2^5 = ?$ . ¿Cómo se escribe ese valor en hexadecimal? Inicializar los 32 elementos de la lista con ese valor y comprobar cuándo se produce el acarreo.

$$2^{32}/32 = 2^{32}/2^5 = 2^{27} = 0x08000000.$$

El acarreo se produce en la última suma, resultado  $2^{32} = 16^8 = 0x100000000$ .

3. Por probar valores intermedios: si la lista se inicializa con valores  $0x10000000$ ,  $0x20000000$ ,  $0x40000000$ ,  $0x80000000$ , repetidas cíclicamente, ¿qué valor tomaría la suma de los 32 elementos? ¿Cuándo se producirían los acarrees? Comprobarlo con ddd.

$$0x10000000 + 0x20000000, 0x40000000, 0x80000000 = 0xf0000000$$

$$0xf0000000 \cdot 8 = 0x780000000$$

Por tanto, suma de 32 elementos sería:

$$\rightarrow \text{resultado} = 0x780000000$$

Los acarrees se producirían cuando el contador vale: 5, 10, 14, 19, 23, 27 y 31.



CUESTIONES SOBRE suma64signed.s

1. ¿Cuál es el máximo entero positivo que se puede representar (escribirlo en hexadecimal)? Si se sumaran los N=32 elementos de la lista inicializados a ese valor ¿qué resultado se obtendría (en hexadecimal)? ¿Qué valor aproximado tienen el elemento y la suma (indicarlo en múltiplos de potencias binarias Ki, Mi, Gi)? Comprobarlo con ddd.

El máximo entero que se puede representar es 0x7fffffff ( = 2147483648 =  $2^{31}$  Ki =  $2^{11}$  Mi = 2 Gi).. Obteniéndose de resultado 0x0000000fffffe0 (= -68719476736 =  $-2^{36}$  =  $-2^{16}$  Ki = -2<sup>6</sup> Mi = 64 Gi).

2. Misma pregunta respecto a negativos: menor valor negativo en hexadecimal, suma, valores decimales aprox., usar ddd.

El menor negativo es 0x80000000 ( = -2147483648 =  $-2^{31}$  Ki =  $-2^{11}$  Mi = -2 Gi). Obteniendo al sumarlo 32 veces 0xffffffff00000000 (= -68719476736 =  $-2^{36}$  =  $-2^{16}$  Ki = -2<sup>6</sup> Mi = 64 Gi)

3. Si nos proponemos obtener sólo 1 acarreo con una lista de 32 elementos positivos iguales, se podría pensar que el objetivo es que la suma alcance  $2^{31}$  (que no cabe en 32 bits como número positivo en complemento a dos). Aparentemente, cada elemento debe valer por tanto  $2^{31}/32 = 2^{31}/2^5 = ?$ . ¿Cómo se escribe ese valor en hexadecimal? Inicializar los 32 elementos de la lista con ese valor y comprobar si se produce acarreo.

$$2^{31}/32 = 2^{31}/2^5 = 2^{26} = 0x04000000.$$

→ resultado = 0x80000000

Por tanto, no se produce acarreo al sumar los 32 elementos de la lista con ese valor.

4. Repetir el ejercicio anterior de forma que sí se produzca acarreo desde los 32 bits inferiores a los superiores. ¿Cuál es el valor del elemento requerido? ¿Por qué es incorrecto el razonamiento anterior? Indicar los valores decimales aproximados (múltiplos de potencias de 10) del elemento y de la suma. Comprobar usando ddd.

$$2^{32}/32 = 2^{32}/2^5 = 2^{27} = 0x08000000 \quad (2^{27} \approx 10^8)$$

El razonamiento anterior es falso ya que  $2^{32} = 16^8$  pero  $2^{31} = 8 \cdot 16^7$  por tanto no se produce acarreo.

$$\rightarrow \text{resultado} = 0x100000000 = 2^{32} (\approx 4 \cdot 10^9)$$

## Programación en ensamblador x86 Linux

5. Respecto a negativos,  $-2^{31}$  sí cabe en 32 bits como número negativo en complemento a dos. Calcular qué valor de elementos requiere para obtener como suma  $-2^{31}$ , y para obtener  $-2^{32}$ . Comprobarlo usando ddd.

$$-2^{31}/32 = -2^{31}/2^5 = -2^{26} = -67108864 = \text{Oxfc000000}$$

$$-2^{32}/32 = -2^{32}/2^5 = -2^{27} = -134217728 = \text{Oxf8000000}$$

Se requiere el valor de  $\text{Oxfc000000}$  para obtener  $-2^{31}$  y  $\text{Oxf8000000}$  para obtener  $-2^{32}$ .

6. Por probar valores intermedios: si la lista se inicializa con los valores  $\text{0xF0000000}$ ,  $\text{0xE0000000}$ ,  $\text{0xE0000000}$ ,  $\text{0xD0000000}$ , repetidos cíclicamente, ¿qué valor tomaría la suma de los 32 elementos (en hex)? Comprobarlo con ddd.

La suma tomaría el valor:

$$\rightarrow \text{resultado} = ((-1) \cdot (1 \cdot 16^7) + (-1) \cdot (2 \cdot 16^7) \cdot 2 + (-1) \cdot (3 \cdot 16^7)) \cdot 8 = -1.717986918 \cdot 10^{10} = \text{Oxfffffff000000000}$$

## Programación en ensamblador x86 Linux

### CUESTIONES SOBRE media.s

1. Rellenando la lista al valor -1, la media es -1. Cambiando un elemento a 0, la media pasa a valer 0. ¿Por qué? Consultar el manual de intel sobre la instrucción de división. ¿Cuánto vale el resto de la división en ambos casos? Probarlo con ddd.

La media sale cero ya que el cociente de dividir  $(0+(-1)\cdot 32)$  entre 33 es cero, al ser  $(0+(-1)\cdot 32)/33 = -0.969696\dots$ , siendo, por lo tanto, el resto igual a -32.

2. También se obtiene cociente 0 si se cambia  $\text{lista}[0] = 1$ , o  $\text{lista}[0] = 2$ , o  $\text{lista}[0] = 3\dots$  Comprobarlo con ddd. La siguiente pregunta lógica es hasta cuando se puede incrementar  $\text{lista}[0]$  sin que cambie el cociente=0.

Para facilitar el cálculo mental, podemos ajustar  $\text{lista}[1] = -2$ , y así la suma de todo el array vale  $\text{lista}[0]-32$ , resultando más fácil calcular el resto. ¿Para que rango de valores de  $\text{lista}[0]$  se obtiene cociente 0? ¿Cuánto vale el resto a lo largo de ese rango? Comprobar que coinciden los signos del dividendo(suma) y del resto.

NOTA: Para evitar el ciclo editar-ensamblar-enlazar-depurar, se pueden poner un par de breakpoints antes de llamar a la subrutina que calcula la media. Tras encontrar el primer breakpoint se puede modificar  $\text{lista}[0]$  con el comando `set var lista = <valor>`. Pulsar Cont para llegar al segundo breakpoint y ver en EAX y EDX los resultados retornados por la subrutina (que no debe hacer PUSH/POP EDX ya que no se pretende conservar el valor de EDX sino retornar el resto). Para hacer muchas ejecuciones seguidas, puede merecer la pena (re)utilizar la línea de comandos `(run/set var.../cont)` en lugar del ratón.

Para el rango de valores comprendidos en el intervalo  $[0, 67]$  se obtiene cociente 0.

Y el resto =  $\text{lista}[0] + \text{lista}[1] + \text{lista}[2]\cdot \text{longlista}$ , siendo  $\text{lista}[1] = -2$ .

3. ¿Para qué rango de valores de  $\text{lista}[0]$  se obtiene media 1? ¿Cuánto vale el resto de ese rango? Comprobarlo con ddd, y notar que tanto los dividendos como los restos son positivos (el cociente se redondea hacia cero).

Para el rango de valores comprendidos en el intervalo  $[68, 101]$  se obtiene cociente 0.

Y el resto =  $\text{lista}[0] + \text{lista}[1]\cdot 2 + \text{lista}[2]\cdot \text{longlista}\cdot 2$ , siendo  $\text{lista}[1] = -2$ .

4. ¿Para qué rango de valores de  $\text{lista}[0]$  se obtiene -1? ¿Cuánto vale el resto en ese rango? Comprobarlo con ddd, y notar que tanto los divisores como los restos son negativos (el cociente se redondea hacia cero).

Para el rango de valores comprendidos en el intervalo  $[-33, 0]$  se obtiene cociente 0.

Y el resto =  $\text{lista}[0]$ .