**PRACTICA 2 – PREGUNTAS DE AUTOCOMPROBACION**

**Davide Gallitelli, Giulia Angarano**

**Sesión de depuración saludo.s**

1.

Ejecutando “mov longsaludo, %edx”, movimos el valor de la variable “longsaludo” al registro “edx”. Entonces este ahora contiene el valor hexadecimal 0x1C (valor decimal 28), porque “longsaludo” es una variable entera que almacena “.‐saludo” (el número de bytes que ocupa saludo). Esta instrucción sirve para indicar que se van a escribir 28 bytes.

2.

Ejecutando “mov $saludo, %ecx”, movimos la dirección de la variable “saludo” al registro “ecx”. Este ahora contiene el valor hexadecimal 0x8049098, correspondiente a la dirección en memoria en la cual empieza la cadena de texto “saludo”.

3.

Si se elimina el símbolo de dato inmediato ($), en vez de mover la dirección de memoria de “saludo” al registro “ecx”, lo que haremos será mover el contenido de saludo al registro. Como podemos comprobar el valor hexadecimal del registro “ecx” pasará de ser 0x8049098 (con el símbolo $) a 0x616c6f48, que es la representación hexadecimal de la cadena de texto almacenada en la variable “saludo”.

4.

La variable “longsaludo” ocupa 1B en memoria, mientras que la variable “saludo” ocupa 28B, entonces la seccion de datos ocupa 29B en memoria.

Para comprobarlo, examinamos “32 char bytes from &saludo”, obteniendo la siguiente salida:

```

(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 'W' 111 'o' 114 'r'

0x80490af : 108 'l' 100 'd' 33 '!' 10 '\n' 28 '\034' 0 '\000' 0 '\000' 0 '\000'

```

Ahora comprobaremos que la variable “longsaludo” tiene un tamaño de 1B, examinando “4 decimal bytes from &longsaludo” y obteniendo la siguiente salida:

```

(gdb) x /4db &longsaludo

0x80490b3: 28 0 0 0

```

5.

Hacemos los volcados de la variable “longsaludo”:

```

(gdb) x /1xb &longsaludo

0x80490b4 : 0x1c

(gdb) x /4xb &longsaludo

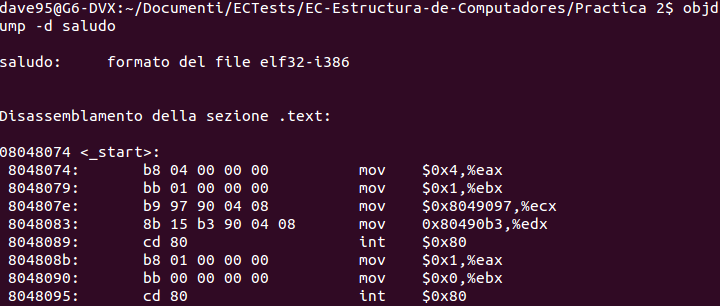
0x80490b4 : 0x1c 0x00 0x00 0x00

```

“longsaludo” ocupa la dirección 0x80490b4, siendo el primer byte el menos significativo. Los procesadores x86, usan el criterio menor (little‐endian), porque primero escriben el bit menos significativo. Por esto un cambio en un bit de la derecha tiene mayor repercusión que un cambio en un bit de la izquierda.

6.

La instrucción “mov $1, %ebx” ocupa 5 posiciones de memoria, desde la 0x8048079 hasta la ox804807d. Esta información la obtenemos haciendo el comando “objdump –d saludo”, como en figura:



7.

Si eliminamos la primera instruccion “int 0x80”, eliminamos la interrupción para hacer la llamada al sistema write (servicio 4 del kernel de Linux indicado con la instrucción “mov $4, %eax”) y se produce la siguiente salida:

```

(gdb) run

Program exited normally.

```

El programa finaliza correctamente, pero no muestra los mensajes por pantalla.

Si eliminamos la segunda instrucción “int 0x80”, obtenemos la próxima salida:

```

(gdb) run

Hola a todos!

Hello, World!

Program received signal SIGSEGV, Segmentation fault.

0x08048095 in ?? ()

```

La instrucción “mov $1, %eax”, indica que vamos a llamar al servicio 1 del kernel de Linux, “exit” en este caso, que es el encargado de finalizar la ejecución de un programa. Como no se produce esta invocación, el programa no sabe como acabar y produce un fallo de segmentación.

8.

El número de llamada al sistema READ es el 3. Hemos obtenido esta información del archivo “/usr/include/asm/unistd\_32.h”:

#define \_\_NR\_read 3

**Sesión de depuración suma.s**

1.

EAX es el registro que hemos usado como acumulador, por eso antes de ejecutar la instrucción su valor será el resultado de las sumas realizadas: 37. [Hemos sumado 1, 2, 10, 1, 2, 0b10 (2 en binario), 1, 2, 0x10 (16 en hexadecimal)]

La expresión (.‐lista)/4 significa que todo el espacio en memoria ocupado de lista (36B) lo dividamos en grupos de 4, como estamos trabajando con enteros de 32 bits (ocupan 4 B cada uno). Entonces la expresión valdrá 9.

2.

Con un lista de 3 elementos como la indicada, el valor de la variable resultado en hexadecimal es 0xFFFFFFFD. Con signo su valor será ‐3, mientras sin signo su valor será 253, la diferencia entre el resultado con signo y sin signo está en que con signo el 0xFFFFFFFF al realizarle el complemento a 2, su valor representado es ‐1, mientras que sin signo su valor representado es 2^8 – 1, que es 255.

Por iteraciones el valor de EAX es 0xffffffff en hexadecimal y ‐1 en decimal con signo, 0xfffffffe (‐2) y en la última, 0xfffffffd (‐3). Como en la suma no hemos controlado el acarreo, el resultado con signo será ‐3, pero sin signo será 2^8 – 3, que es 253.

3.

La etiqueta suma tiene la dirección 0x8048095 y la etiqueta bucle la dirección 0x80480a0. Esta información se puede obtener del volcado del código ensamblador para el bucle de funciones, al que podemos acceder desde el menú, View → Machine Code Window.

```

0x0804807f <+11>: call 0x8048095 <suma>

0x080480a6 <+6>: jne 0x80480a0 <bucle>

```

4.

El procesador usa el registro EIP como puntero a la siguiente dirección de memoria que va a ser ejecutada por el procesador, y el registro ESP es usado como puntero al inicio de la pila del programa.

5.

El valor de ESP antes de ejecutar CALL es 0xFFFFD4A0 y antes de ejecutar RET es 0xFFFFD49C. La diferencia es de 4 y es producida porque al ejecutar CALL, el tamaño de la pila se modifica porque se guarda en ella la dirección de retorno a la función que será necesaria cuando se ejecute RET.

6.

La instrucción CALL modifica los registros EAX, EDX, ESP y EIP: EAX es modificado porque en nuestro programa funcionará como acumulador para ir guardando el resultado de las sumas, EDX es modificado porque funcionará como índice para conocer la siguiente dirección de memoria a leer, ESP es modificado porque almacenará la dirección a la que retornar cuando se ejecute la instrucción RET, y EIP es modificado porque almacenará la dirección de la próxima instrucción a ser ejecutada.

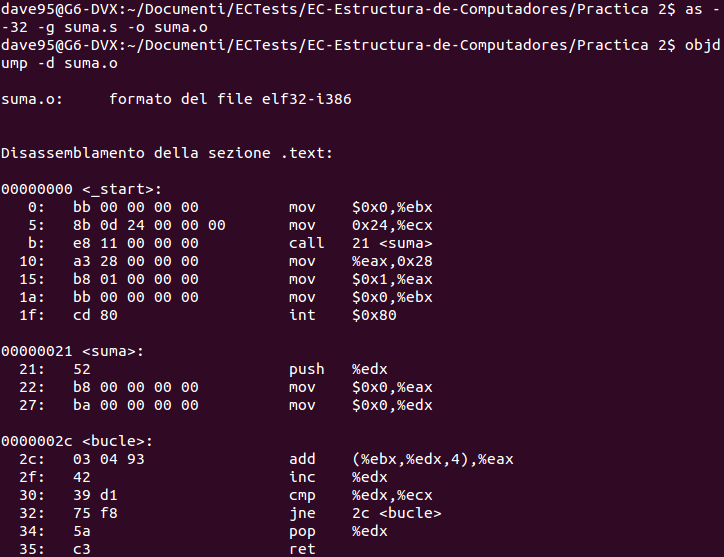
7.

La instrucción RET modifica los registros ESP y EIP porque los punteros deberán cambiar para que el programa pueda continuar con su ejecución al punto donde estaba antes de llamar a la subrutina.

8.

Cuando se inicia al programa, vemos que el registro ESP apunta a la dirección 0xfffd4a0 con el valor 0x00000001, y una vez dentro de la subrutina suma, en la dirección 0xffffd49c tenemos el valor 0x08048084, a la siguiente instrucción tenemos en la dirección 0xffffd498 el valor 0x00000000, justo después de hacer POP, en la dirección 0xffffd49c tenemos el valor 0x08048084 y, finalmente, al volver al programa principal en la dirección 0xffffd4a0 tenemos el valor 0x00000001.

9.



10.

Que no se podría volver la ejecución al programa principal desde el que se llamó a la subrutina, y como no sabría cómo volver, produciría un error en la ejecución del programa:

```

(gdb) run

Program received signal SIGSEGV, Segmentation fault.

0x080480a9 in ?? ()

```

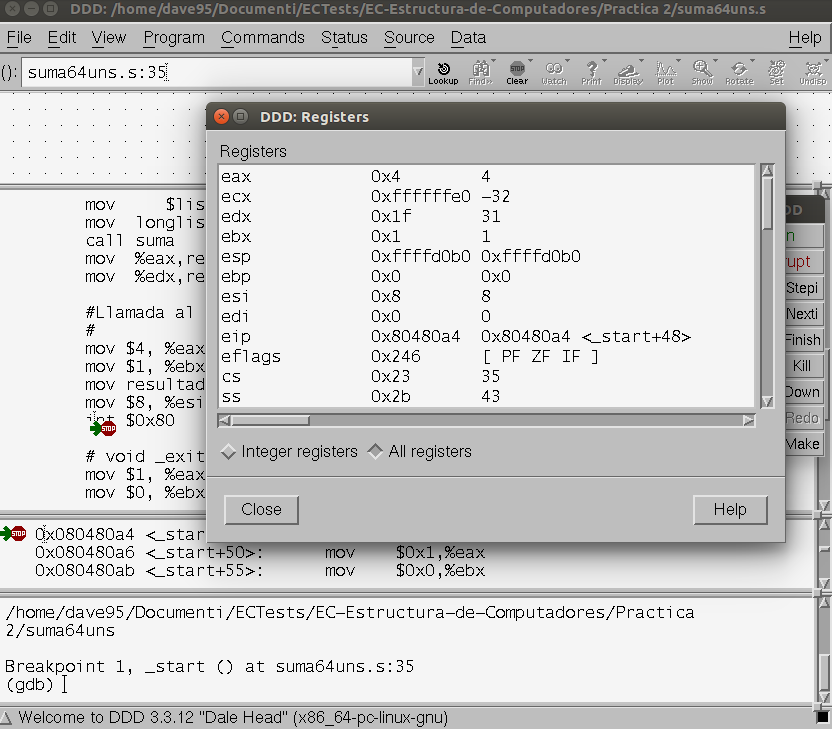
**Cuestiones sobre suma64unsigned.s**

1.

El valor maximo de cada elemento de 32 bit es 2^32 - 1 = 4294967295. En hexadecimal es 0xFFFFFFFF.

Se producen 31 acarreos.

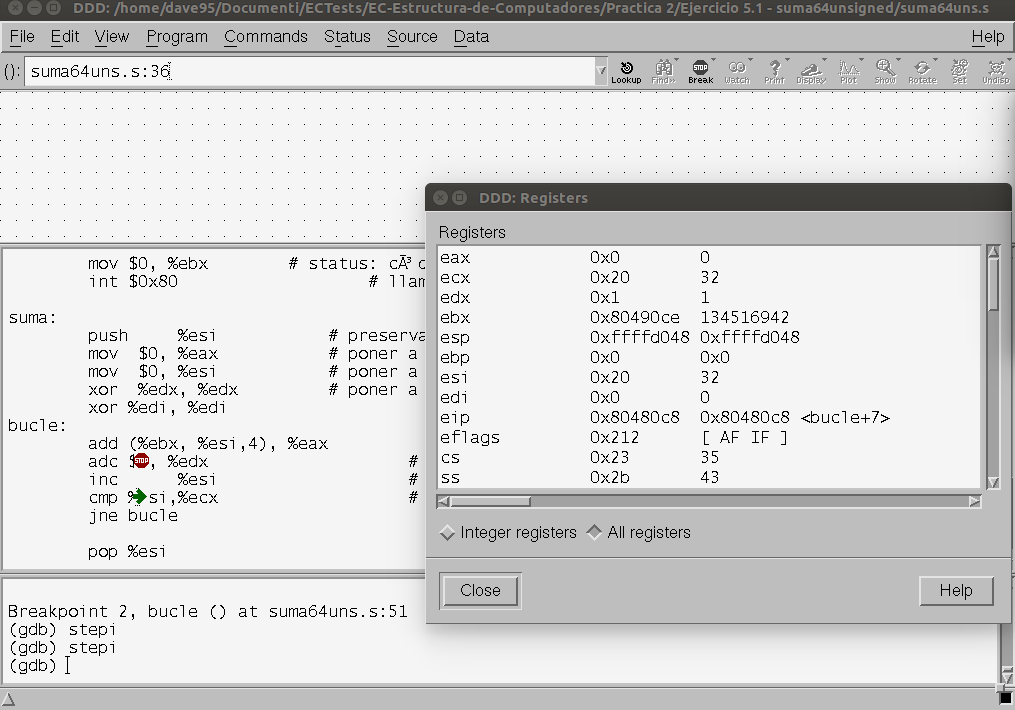
La suma es 137438953440, en hexadecimal es 0x1FFFFFFFE0.



2.

Para obtener solo 1 acarreo cada elemento tiene que tener el valor 2^32/2^5 = 2^27 = 0x8000000

Si realizamos la suma paso a paso comprobamos que el acarreo se produce en la última suma, dando como resultado 0x100000000.



3.

0x10000000 +

0x20000000 +

0x40000000 +

0x80000000 =

0Xf0000000 +

0x10000000 =

**0x100000000** + ACARREO

0x20000000 +

0x40000000 +

0x80000000 =

0x1E0000000 +

0x10000000 +

0x20000000 +

**0x210000000** + ACARREO

0x40000000 +

0x80000000 =

0x2D0000000 +

0x10000000 +

0x20000000 +

**0X300000000** + ACARREO

0x40000000 +

0x80000000 =

0x3C0000000 +

0x10000000 +

0x20000000 +

0x40000000 +

**0x430000000** + ACARREO

0x80000000 =

0x4B0000000 +

0x10000000 +

0x20000000 +

0x40000000 +

**0x520000000** + ACARREO

0x80000000 =

0x5A0000000 +

0x10000000 +

0x20000000 +

0x40000000 +

**0x610000000** + ACARREO

0x80000000 =

0x690000000 +

0x10000000 +

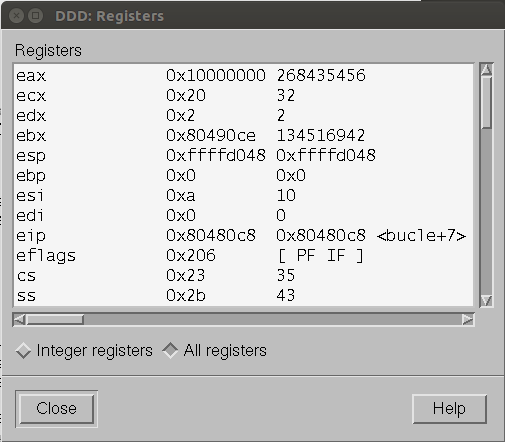
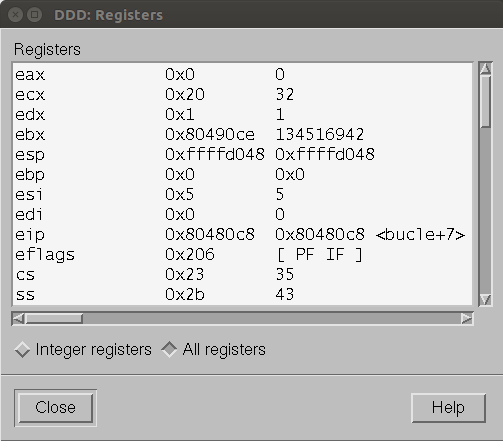
0x20000000 +

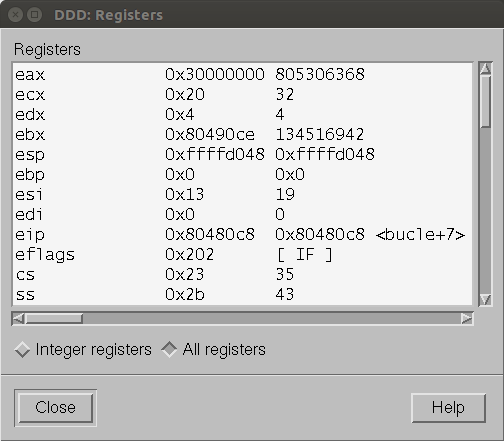
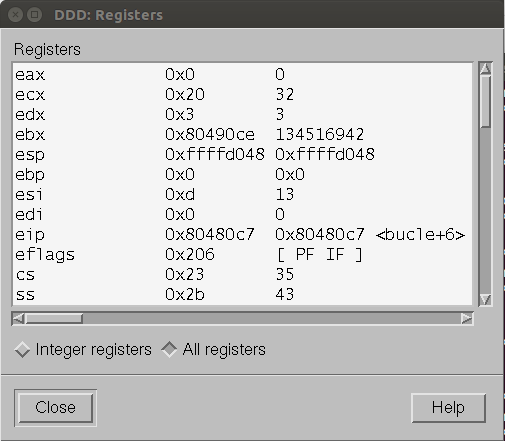
0x40000000 +

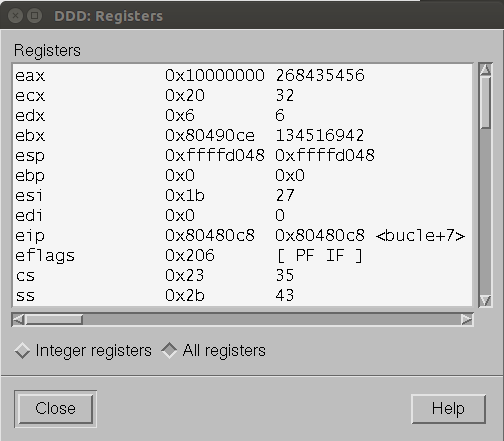
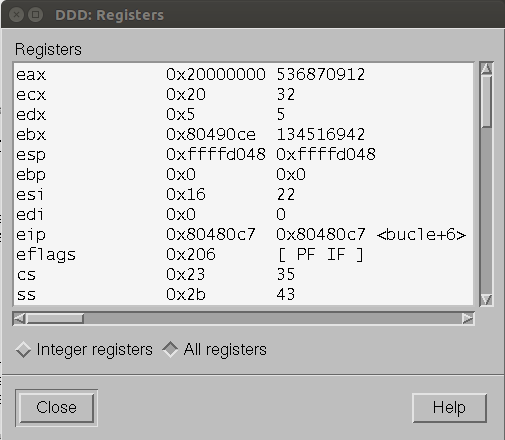
**0x700000000** + ACARREO

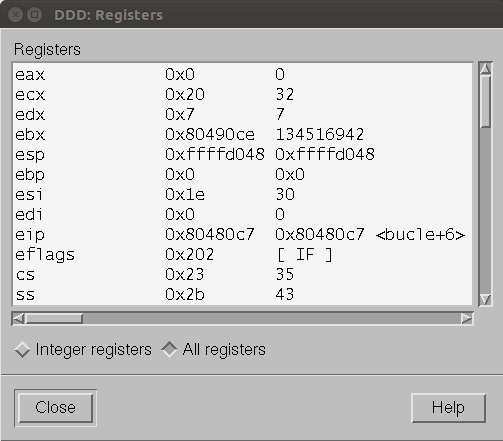
0x80000000 =

0x780000000 SUMA









**Cuestiones sobre suma64signed.s**

1. El maximo valor que cada elemento puede tomar es (2^32)/2 – 1 = 2147483647 (aprox. 2 Mi) y en hexadecimal es 0x7FFFFFFF.

El resultado seria 68719476704 (aprox. 67 Mi) y en hexadecimal 0xFFFFFFFE0.

2. El minimo valor que cada elemento puede tomar es – (2^32)/2 = - 2147483648 (aprox. -2 Mi) y en hexadecimal es 0x80000000.

El resultado seria -68719476736 (aprox. -67 Mi).

3. 2^31 / 2^5 = 2^26, en hexadecimal 0x4000000.

Sumando 32 elementos que toman el valor 0x4000000 se produce 0x80000000.