Alberto Jesús Durán López. Estructura de computadores: Práctica 1.

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

Esta instrucción mueve el valor de la variable "longsaludo" al registro "edx"
Una vez depurado, obtenemos que tiene un valor decimal de 28 y en hexadecimal 0x1C
Esta variable almacena 28 bytes (número de bytes que ocupa ".-saludo"

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).

Esta instrucción mueve el valor de la dirección de memoria de "saludo" al registro "ecx".
Al comprobar su valor vemos que tiene un valor en hexadecimal de 0x8049098 que es el valor de la dirección de memoria de 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 le quita el $ a la instrucción anterior, en vez de indicar la dirección de memoria , indicaría el valor de la variable "saludo" por lo que el valor del registro "ecx" al realizar la instrucción será diferente al que teníamos anteriormente.

4. ¿Cuántas posiciones de memoria ocupa 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.

Comprobamos cuantas posiciones de memoria ocupan:
"longsaludo" ocupa 1B, "saludo" ocupa 28B por lo que la sección de datos ocupa 29B que es la suma de amba variables.
Esto se puede comprobar con un volcado de datos.

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 direcciones de memoria ocupa longsaludo? ¿Cuál byte está en la primera posición, el más o el menos significativo? ¿Los procesadores de la línea x86 usan el criterio del extremo mayor (big-endian) o menor (little-endian)? Razonar la respuesta.

Hacemos los volcados de “longsaludo”, en gdb escribimos lo siguiente:

```
(gdb) z /1xb &longsaludo
0x80490b4 <longsaludo>: 0x1c
(gdb) x /4xb &longsaludo
0x80490b4 <longsaludo>: 0x1c 0x00 0x00 0x00#
```

Comprobamos que “longsaludo” ocupa la dirección 0x80490b4, que es el primer byte menos significativo.
Los procesadores x86 usan el criterio menor (little-endian) ya que en binario, al cambiar un bit en la derecha se produce un mayor cambio que en la izquierda.

6. ¿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 5 posiciones de momoria. Se puede saber gracias al comando “disassemble” de donde tenemos que varía desde la 0x08048079 hasta la 0x0804807d.

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 muestra la siguiente salida:

(gdb) run
Program exited normally.

Si se elimina la segunda se muestra:

(gdb) run
Hola a todos!
Hello, World!

Program received signal SIGSEGV, Segmentation fault.
0x08048095 in ?? ()#

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

El número de llamada al sistema READ es 3 que se obtiene del archivo “/user/include/asm/unistd_32.h” que contiene los números de llamadas al sistema de READ. → #define __NR_read   3

Tabla 4: preguntas de autocomprobación (suma.s) Sesión de depuración suma.s

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

Justo antes de ejecutar RET, el contenido de EAX es de 37 que es la suma de 1,2,10,1,2, 0b10,1,2,0x10.
0b10=valor decimal 2 representado en binario.
0x10=valor decimal 16 en hexadecimal.

(.-lista)/4 es la suma de todo entre 4. Como estamos trabajando con enteros de 32 bits, estos ocupan 4B, como lista ocupa 36B por lo que la expresión vale 9 (la lista tiene 9 números).

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 en 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).

Con una lista de 3 elementos, el valor de resultado en hexadecimal es 0xFFFFFFFD, con valor decimal con signo sería -3 y sin signo sería 253. La diferencia entre estos dos últimos es que con signo el 0xFFFFFFF al realizarle el complemento a 2, su valor representado es -1, mientras que sin signo su valor es 2^8-1 = 255.

El valor de EAX es 0xffffffff en hexadecimal y -1 en decimal con signo, 0xfffffffe en hecadecimal y 2 en decimal con signo. En la última, 0xfffffffd en hexadecimal y -3 en decimal con signo. Como no hemos tenido en cuenta el acarreo, el resultado con signo será -3, pero sin signo será 2^8-3 =253

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

Se le asigna la dirección 0x8048095 a la suma y al bucle 0x80480a0.
Dicha información se optiene del volcado del código ensamblador para el bucle de funciones.

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

El registro EIP lo usa como puntero a la siguiente dirección de memoria que va a ejecutar el procesador mientras que el registro ESP se usa como puntero al inicio de la pila del programa.

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 es 0xFFFFD4A0 y antes de ejecutar RET es 0xFFFFD49C.
Estos dos valores se diferencia en 4 que es la diferencia producida al ejecutar CALL ya que el tamaño de la pila se modifica porque se guarda en ella la dirección de retorno.

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

La instrucción CALL modifica los registros EAX, EDX, ESP, EIP.
EAX: Para guardar el resultado de las sumas.
EDX: Para conocer la siguiente dirección de memoria a leer.
ESP: Almacena la dirección a la que retornar cuando se ejecute RET.
EIP: Almacenará la dirección de la prox. Instrucción al ser ejecutada.

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

La instrucción RET modifica los registros ESP y EIP.
Se necesitan modificar estos registros para que el programa pueda seguir con su ejecución.

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 Dta->Memory se puede usar $esp para referirse a donde apunta el registro ESP.

El registro ESP apunta a la dirección 0xfffd4a0 con el valor 0x00000001 y una vez dentro de suma, en la dirección 0xffffd49c se tiene el valor 0x08048084.
En la siguiente instrucción se tiene el valor 0xffffd498 con valor 0x00000000. Justo antes de POP, se tiene el valor 0x08048084 en la dirección 0xffffd4a0.
Finalmente, al volver al programa principal, en la dirección 0xffffd4a0 se tiene el valor 0x00000001.

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

Si se elimina la instrucción RET no se podría volver la ejecución del programa por lo que daría un error en la ejecución del programa.

(gdb) run

Program received signal SIGSEGV, Segmentation fault.
0x080480a9 in ?? ()#

Código suma64uns.s

```
.section .data
lista:          .int    4294967295,4294967295,1,2,3,4,5,6
                        .int    0b111,0b1000,0b1001,0b1010,0b1011,0b1100,0b1101,0b1110
                        .int    017,020,021,022,023,024,025,026
                        .int    0x17,0x18,0x19,0x1A,0x1B,0x1C,0x1D,0x1E

longlista:      .int    (.-lista)/4
resultado:      .quad   -1      # Usaremos un número de 64 bits.

.section .text
_start: .global _start

            mov     $lista, %ebx
            mov     $0, %edx        # Ponemos a 0 el registro (del acarreo)
            mov     longlista, %ecx
            call    suma

            # Moveremos el resultado desde el registro acumulador a la variable declarada.
            # Movemos el acarreo.
            # Generado a la dirección que empieza 4 posiciones por detrás del resultado.

            mov     %eax, resultado
            mov     %edx, resultado+4

            mov     $1, %eax
            mov     $0, %ebx
            int     $0x80
suma:
            push    %esi        # Usaremos %esi como índice.
            mov     $0, %eax
            mov     $0, %esi
bucle:
            add     (%ebx, %esi, 4), %eax
            jnc     acarreo     # Saltaremos si no se ha producido acarreo en la operación anterior.
            inc     %edx        # Si no se produce el salto, hay que incrementar el acarreo.
acarreo:
            inc     %esi
            cmp     %esi,%ecx
            jne     bucle

            pop     %esi
            ret
```