ESTRUCTURA DE COMPUTADORES
DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS
PREGUNTAS PRÁCTICA 1
Juan Antonio Villegas Recio

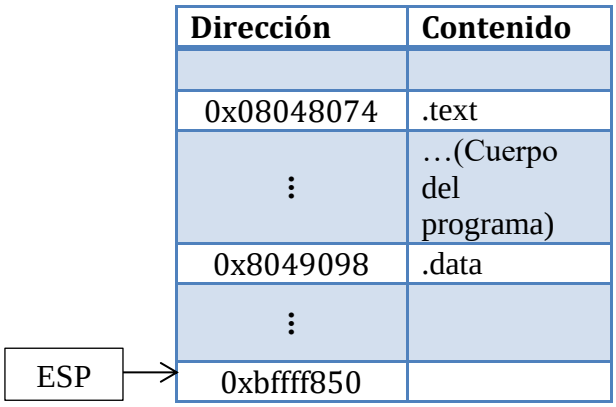
Preguntas de autocomprobación saludo.s

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.**

   0x1c = 28, que es la longitud de la cadena "Hola a todos!\nHello, World!\n". Se necesita para pasarla como argumento a la instrucción write. Ese valor se obtiene de .-saludo, que corresponde a restarle a la dirección actual la dirección donde comienza saludo, como está declarado justo después de saludo, se obtiene así la longitud de la cadena saludo y se guarda en la variable longsaludo, que posteriormente se almacena en EDX.

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 0x8049098, que es la dirección de memoria en la que comienza la cadena saludo.

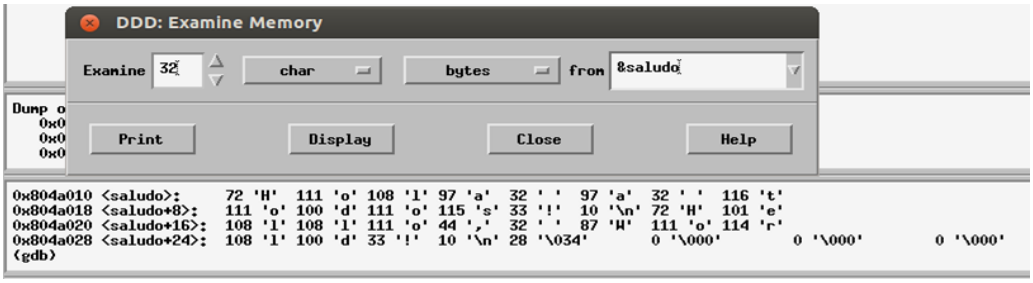


3. **¿Qué sucede si se elimina el símbolo de dato inmediato ($) de la instrucción anterior? (mov saludo, %ecx) Realizar la modificación, indicar el contenido de ECX en hexadecimal, explicar por qué no es lo mismo en ambos casos. Concretar de dónde viene el nuevo valor (obtenido sin usar $).**

   En este caso ECX contiene 0x616c6f48, contenido de saludo, es decir, la cadena contenida en la variable saludo, no su dirección de memoria.
   No es lo mismo porque lo que utiliza el programa para imprimir la cadena es su dirección de memoria, si se pasa la cadena en sí, el programa no funcionará correctamente.

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.**

   La variable longsaludo ocupa 1B y la variable saludo 28B, uno por cada carácter de la cadena, por tanto la sección data ocupa 29 bytes. En la siguiente imagen se puede apreciar que volcando 32 char bytes desde &saludo obtenemos los 28 caracteres de la cadena y por último el 28 que representa la longitud de la cadena, puesto justo a continuación de la declaración de saludo.



Preguntas de autocomprobación 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 cuánto valen 0b10, 0x10, y (.-lista)/4.**

   El valor de EAX es 0x25=37. El valor de 0b10 (valor en binario) en decimal es 2 y el de 0x10 (valor en hexadecimal) en decimal es 16. Teniendo esto en cuenta, la suma de todos los valores es 1+2+10+1+2+2+1+2+16=37, que es precisamente el valor que tenía EAX antes del retorno, y esto se debe a que la función devuelve el valor guardado en EAX, así que necesariamente antes del retorno tenía que estar guardada en EAX el valor que devuelve la función, es decir, la suma de todos los elementos, 37 en este caso.

   (.-lista)/4 es equivalente a restarle la dirección actual la dirección en la que empezara lista y dividir ese valor entre 4. Obteniendo así las posiciones de memoria que ocupa lista, y dividiendo entre el número de bytes que ocupa cada elemento de la lista, 4 por ser valores enteros, obtenemos el número de elementos que tiene la lista.

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).**

Valores de EAX	Hexadecimal	Decimal	Flags de estado
Iteración 1	0xFFFFFFFF	-1	
Iteración 2	0xFFFFFFFE	-2	CF
Iteración 3	0xFFFFFFFD	-3	CF

Haciendo a mano la suma obtenemos un resultado diferente porque el programa toma el valor 0xFFFFFFFF como un número en complemento a 2, y su complemento a dos es -1, de ahí que al sumarlo 3 veces se obtenga -3.
Haciendo el volcado de memoria obtenemos -3 con signo y 4294967293 sin signo.

```
Breakpoint 1, _start () at suma.s:14
(gdb) print resultado
$1 = -3
(gdb)
```

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

   A suma se le ha asignado la dirección 0x8048095 y a bucle la dirección 0x80480a0. La información la he obtenido posicionando el cursor en la función suma y con view → machine code window, se obtienen las direcciones de las diversas instrucciones de la etiqueta suma. La primera de ellas se corresponde con la dirección de suma. Análogamente posicionando el cursor en bucle, se obtiene la dirección de bucle.

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

   EIP es el registro de instrucción, almacena la dirección de memoria de la siguiente instrucción a ejecutar. ESP es el Stack Pointer o puntero de pila, almacena la dirección de memoria de 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?**

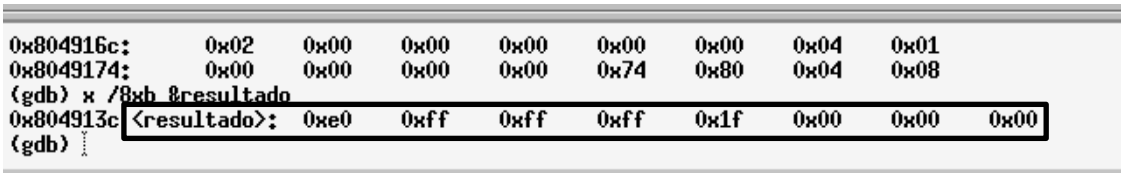
   Antes de ejecutar CALL el valor de ESP es 0xBFFFF870 y antes de ejecutar RET es 0xBFFFF86C. La diferencia es 4, porque cuando se hace una llamada a una función, se guarda en la pila la dirección de retorno de la función, que es un valor entero que ocupa 4 bytes, por tanto el puntero de pila aumenta en 4 su valor. El valor que nos interesa es el valor de ESP antes de ejecutar RET, porque conociendo el valor que está en la posición de memoria a la que apunta ESP podemos conocer la dirección de retorno de la función.

```
(gdb) x /16xb $esp
0xbffff870:     0x01    0x00    0x00    0x00    0xe4    0xf9    0xff    0xbf
0xbffff878:     0x00    0x00    0x00    0x00    0x03    0xfa    0xff    0xbf
(gdb)
```

Peguntas de autocomprobación sobre suma64unsigned.s:

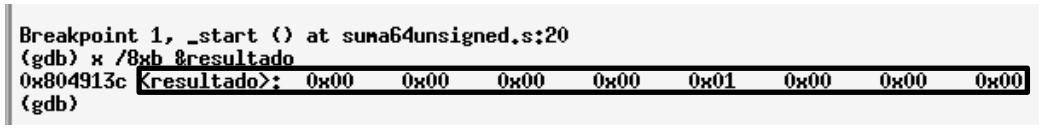
1. **Para N≈32, ¿cuántos bits adicionales pueden llegar a necesitarse para almacenar el resultado? Dicho resultado se alcanzaría cuando todos los elementos tomaran el valor máximo sin signo. ¿Cómo se escribe ese valor en hexadecimal? ¿Cuántos acarreos se producen? ¿Cuánto vale la suma (indicarla en hexadecimal)?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

   En hexadecimal, el máximo valor sin signo es 0xFFFFFFFF, que en decimal es 4294967295. Por cada suma se produciría un acarreo, por tanto, al ser 32 elementos se producirían 32 acarreos. La suma sería igual a 32*4294967295=137438953440, que en hexadecimal es 0x1FFFFFFFE0.



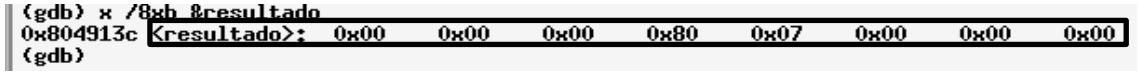
2. **Si nos proponemos obtener sólo 1 acarreo con una lista de 32 elementos iguales, el objetivo es que la suma alcance 2^{32} (que ya no cabe en 32bits). Cada elemento debe valer por tanto $2^{32}/32 = 2^{32}/2^{?}$=? ¿Cómo se escribe ese valor en hexadecimal? Inicializar los 32 elementos de la lista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

   Como $32=2^5$, entonces $2^{32}/2^5=2^{27}$=134217728, en hexadecimal, 0x8000000.
   El acarreo se produce en la última suma, que es la que desborda los 32 bits, y el resultado es 0x100000000.



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

   La suma toma el valor 0x780000000 y los acarreos se producirían en cada ciclo, ya que la suma de estos 4 valores desborda los 32 bits.



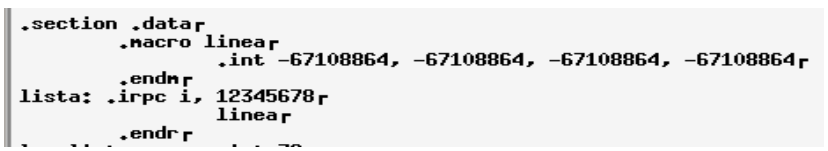
Preguntas de autocomprobación sobre suma64signed.s:

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 ya no cabe en 32bits como número positivo en complemento a dos). Aparentemente, cada elemento debe valer por tanto $2^{31}/32 = 2^{31}/2^{?}$ = ?. ¿Cómo se escribe ese valor en hexadecimal? Inicializar los 32 elementos de la lista con ese valor y comprobar si se produce el acarreo.**

   Como $32=2^5$, entonces $2^{31}/2^5=2^{26}$=67108864, en hexadecimal, 0x4000000.

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

   Para obtener -2^{31}=-2147483648, debemos sumar 32 veces el valor $-2^{31}/2^5$=-67108864.



Y para obtener -2^{32}, seguimos el mismo razonamiento para obtener que debemos sumar 32 veces -134217728.
Al hacerlo en ddd, obtuve el valor 0, debido a que -2^{32} en complemento a 2 desborda los 64 bits.

6. **Por probar valores intermedios: si la lista se inicializara con los valores 0xF0000000, 0xE0000000, 0xE0000000, 0xD0000000, repetidos cíclicamente, ¿qué valor tomaría la suma de los 32 elementos (en hex)? Comprobarlo con ddd.**
   El valor obtenido ha sido 0xCF000000 en complemento a 2.

```
.section .data
        .macro linea
                .int 0xF0000000, 0xE0000000, 0xE0000000, 0xD0000000
        .endm
lista: .irpc i, 12345678
                linea
```

```
Breakpoint 1, _start () at suma64signed.s:20
(gdb) x /8xb &resultado
0x804914c <resultado>:  0x00    0x00    0x00    0x00    0xfc    0xff    0xff    0xff
(gdb)
```

Preguntas de autocomprobación sobre media64signed.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? Probar con ddd.**

   Poniendo todos los valores a -1:

```
.section .data
        .macro linea
                .int -1, -1, -1, -1
        .endm
lista: .irpc i, 12345678
                linea
        .endr
```

```
(gdb) x /1dw &media
0x8049150 <media>:      -1
(gdb) x /1dw &resto
0x8049154 <resto>:      0
(gdb)
```

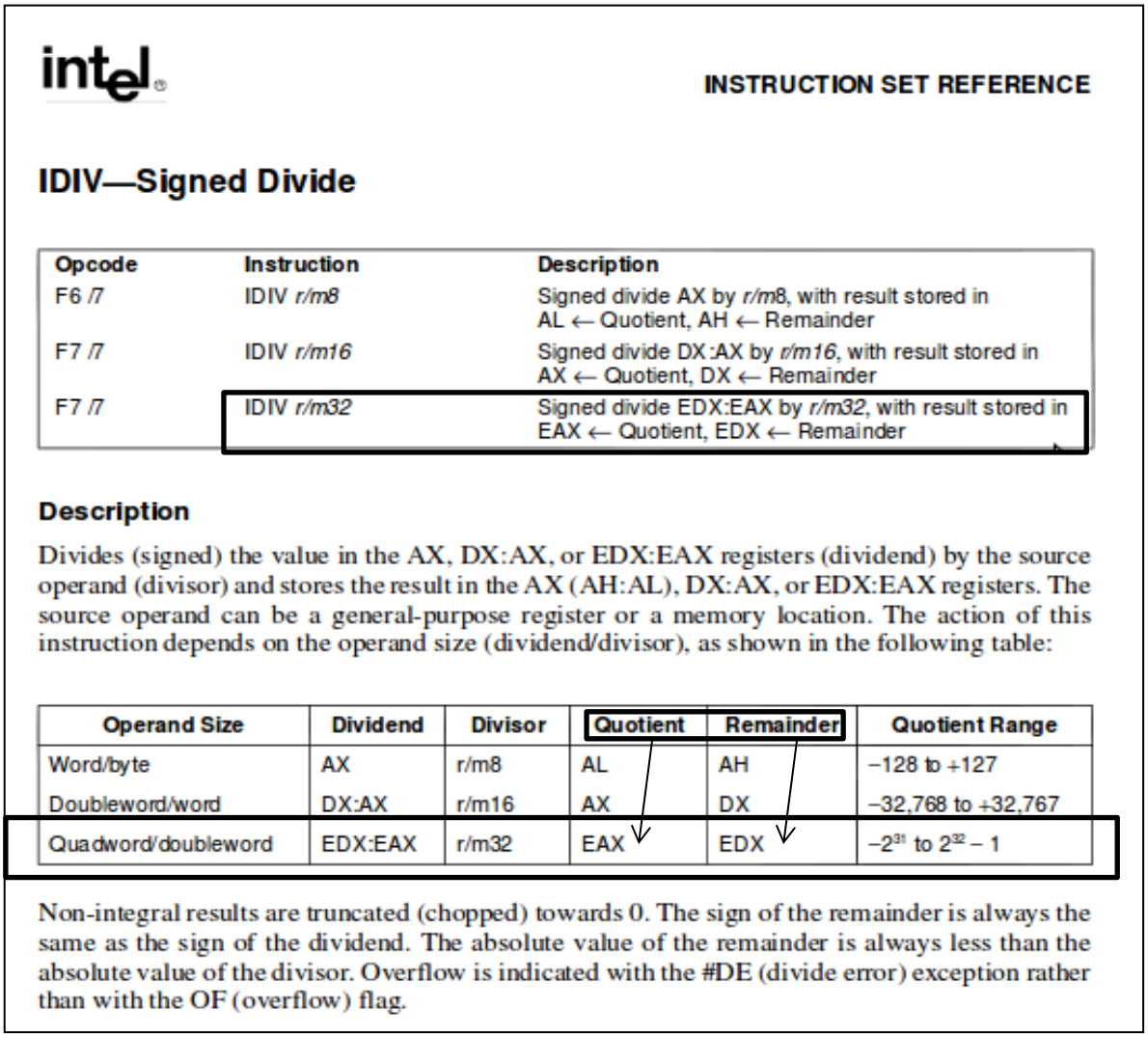
   Y poniendo un elemento a 0[1] (Nota a pie de página):

```
.section .data
        .macro linea
                .int -1, -1, -1, 0
        .endm
```

```
(gdb) print media
$1 = 0
(gdb) print resto
$2 = -24
(gdb)
```

[1] Realmente estamos poniendo 8 elementos a 0, porque esa secuencia de 4 elementos se repiten 8 veces.

Según el manual de Intel:



intel

INSTRUCTION SET REFERENCE

IDIV—Signed Divide

Opcode	Instruction	Description
F6 /7	IDIV *r/m8*	Signed divide AX by *r/m8*, with result stored in AL ← Quotient, AH ← Remainder
F7 /7	IDIV *r/m16*	Signed divide DX:AX by *r/m16*, with result stored in AX ← Quotient, DX ← Remainder
F7 /7	IDIV *r/m32*	Signed divide EDX:EAX by *r/m32*, with result stored in EAX ← Quotient, EDX ← Remainder

Description

Divides (signed) the value in the AX, DX:AX, or EDX:EAX registers (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, or EDX:EAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor), as shown in the following table:

Operand Size	Dividend	Divisor	Quotient	Remainder	Quotient Range
Word/byte	AX	r/m8	AL	AH	−128 to +127
Doubleword/word	DX:AX	r/m16	AX	DX	−32,768 to +32,767
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	-2^{31} to $2^{32}-1$

Non-integral results are truncated (chopped) towards 0. The sign of the remainder is always the same as the sign of the dividend. The absolute value of the remainder is always less than the absolute value of the divisor. Overflow is indicated with the #DE (divide error) exception rather than with the OF (overflow) flag.

Esto se produce porque, al ser el divisor 32, el dividendo no llega a superarlo, por lo que idiv guarda en EAX el cociente de la división, 0 en este caso y el resto, -24, en el registro EDX.