

5 Trabajo a realizar

Se propone mejorar este último programa (Sección 4, Figura 15, `suma.s`) para calcular la media de una lista de N enteros realizando los cálculos sobre registros enteros. Entendemos por enteros los `.int` normales de la plataforma x86-64 (4B), y por cálculos las operaciones aritmético-lógicas (no las instrucciones de movimiento que obviamente utilizarán direcciones de 64 bits). Se debe considerar la posibilidad de desbordamiento aritmético (*overflow*), respondiendo siempre con el resultado correcto.

Por comodidad usaremos $N=16$ pero el código no puede asumir que el tamaño es fijo, y debe funcionar correctamente para cualquier tamaño $N>0$. Por explicar razonadamente la condición de usar registros de 32 bits, que en principio pudiera parecer arbitraria, piénsese en la dificultad de escribir valores de prueba para ejercitar la posibilidad de *overflow* si se usaran enteros `.quad` de 64 bits. También es un ejercicio interesante pensar en cómo se deben tratar adecuadamente los acarreo en aritmética multi-precisión con signo (que requiere extender el signo de los datos al tamaño del resultado), frente al tratamiento sin signo (que puede realizarse por extensión con ceros, o sencillamente acumulando acarreo). Adicionalmente, podemos aprovechar la plataforma de 64 bits para utilizar un registro `.quad` en donde realizar los cálculos directamente (sin aritmética multi-precisión) y comparar este resultado con lo calculado en 32 bits.

Según la temporización de cada curso, se procurarán realizar guiadamente (como Seminario Práctico) los Ejercicios 1-4 (e incluso `suma.s` si sobrara tiempo). Aunque no diera tiempo a tanto, comprender los programas mostrados (Figura 10, Figura 15) y ejercitarse en el uso de las herramientas (Figura 11, Figura 12, Figura 13, Figura 14) son competencias que cada uno debe conseguir personalmente.

Al objeto de facilitar el desarrollo progresivo de la práctica, se sugiere realizar en orden las siguientes tareas:

5.1 Sumar N enteros sin signo de 32 bits sobre dos registros de 32 bits usando uno de ellos como acumulador de acarreo ($N \approx 16$)

En su forma actual `suma.s` ya permite un tamaño variable de la lista. La primera modificación que podemos hacer es cambiar la lista de nueve elementos a una lista que repita 16 veces el número 1. Podemos salvarlo con el nombre `media.s` y ejecutarlo para comprobar si `./media; echo $?` devuelve 16.

Notar que la suma puede necesitar más de 32 bits, incluso si la lista fuera de 2 números (0xffff ffff y 1, o 0x8000 0000 y 0x8000 0000). *¿Qué valor mínimo* habría que repetir 16 veces para producir acarreo? Es 0x1000 0000. Modificar la lista y comprobar que responde que el resultado es 0. Si no se entiende por qué, conviene ejecutarlo paso a paso con el depurador. Seguramente conviene dividir la definición de la lista en 4 líneas, cada una de ellas de 4 enteros. Sólo la primera necesita etiqueta `lista:`. Si hubiéramos rellenado la lista con 16 valores 0xffffffff no hubiera llegado a producir acarreo (*¿por qué?*), y el resultado sería correcto. Comprobarlo. Es fácil calcular mentalmente el resultado en hexadecimal.

Quien lea el guión sin realizar las comprobaciones no habrá caído en la cuenta de que el programa retorna 0 y 240 para los dos ejemplos mencionados. La página `man 2 exit` explica *por qué* se retornan valores tan bajos. Se puede depurar el programa para comprobar que todo iba bien hasta la llamada a `EXIT`. Pero si imprimiéramos el resultado por pantalla usando `printf()` de `libC` evitaríamos este problema con `EXIT`, además de facilitar la comprobación de si el programa es correcto tras cada modificación que hagamos (si se imprime el resultado correcto, asumimos que no hay fallos). Para ello podemos descomentar el formato sugerido, cambiar la etiqueta `_start` por `main`, borrar el cuerpo de `main` y sustituirlo por los cuerpos de `trabajar`, `imprim_C` y `acabar_C` uno tras otro (se pueden borrar esas etiquetas, los respectivos `ret` y toda la subrutina `acabar_L`, y se puede retornar 0 en `acabar_C`), y dejar para el final la subrutina `suma` con su bucle y su `ret`. Tal y como se sugiere en el comentario 4) del fuente, estas modificaciones nos permiten “compilar” el programa ensamblador usando `gcc` (recordar que con el Ubuntu 18.04 que usamos en prácticas, es necesario usar la opción `-no-pie` para que se genere un ejecutable, en lugar de un objeto compartido). Ahora es más fácil comprobar los dos ejemplos mencionados, 16 elementos a 0x1000 0000 o a 0x0fff ffff. Comprobar si los resultados se imprimen correctamente por pantalla con el formato deseado, sin necesidad de entrar en el depurador. El primer ejemplo pierde un bit de acarreo, y nuestro objetivo es conservar ese bit.



La suma en doble precisión de números *sin signo* puede realizarse conservando los *acarreo* que de otra forma se hubieran perdido (instrucción etiquetada `bucle:` en `suma.s`, Figura 15, tras la cual no se comprueba si hay acarreo saliente de EAX). Si los vamos acumulando en otro registro de 32 bits (p. ej. EDX), la concatenación de ambos (EDX:EAX) puede almacenar el resultado (sin signo, de 64 bits). Para comprobar si hay acarreo y acumularlo sólo cuando lo haya, inmediatamente después de la suma etiquetada `bucle:` puede usarse un salto condicional JNC para saltarse una instrucción INC (*pensarlo*) y llegar a una etiqueta colocada tras ese INC. Necesitaremos usar otro registro como índice (tal vez RSI) en lugar de RDX/EDX, e inicializar los registros apropiadamente (acumuladores e índice a 0). Cuando veamos la instrucción DIV para calcular la media, se entenderá el interés de acumular en EDX:EAX.

Tomando como punto de partida el programa `media.s`, con directivas `.int` adicionales para los $N=16$ valores (4 líneas de 4 valores `0x1000 0000`), realizamos las modificaciones indicadas (JNC, INC, etiqueta) y cambiamos el tamaño de `resultado` a 64 bits (consultar en Apéndice 2, Tabla 9, la directiva `as` para ello). La subrutina `suma` debería devolver el resultado en los acumuladores EDX:EAX (deben eliminarse los `push/pop %rdx`), y el programa principal almacenarlo en la variable `resultado` de 64 bits, según el criterio del extremo menor (recordar que la familia x86-64 es *little-endian*). La dirección que empieza 4 posiciones detrás de `resultado` se indica `resultado+4`, naturalmente. Desde `gdb` debería poder visualizarse el valor correcto de la suma con `p/x(long)resultado`, o con `x/1xg &resultado`. Desde `ddd` se puede conseguir el mismo volcado de memoria con el menú *Data→Memory→Examine 1 hex giant from &resultado*.

Aunque consigamos calcular correctamente el resultado `0x1 0000 0000` en EDX:EAX, por pantalla seguimos imprimiendo enteros de 4 bytes. Es inmediato cambiar los especificadores de formato a `"%1u"` y `"%1x"`, así como los argumentos de `printf` a `%rsi` y `%rdx`, para conseguir imprimir el resultado de 8 bytes en decimal sin signo y en hexadecimal. Tras conseguir imprimir el resultado correcto para el ejemplo más pequeño que produce acarreo, probar con el ejemplo más grande posible. El resultado debe ser `0xf ffff fff0` (*¿por qué?*).

5.2 Sumar N enteros *sin* signo de 32 bits sobre dos registros de 32 bits mediante extensión con ceros ($N \approx 16$)

Un número E sin signo de N bits puede ser extendido a N+M bits sin más que añadirle M bits a cero como parte más significativa (a la izquierda). En nuestro caso, deseamos sumar al acumulador EDX:EAX de 32+32 bits un elemento de la lista, E, de 32 bits. Como consideramos que E no tiene signo, podemos extenderlo a 64 bits *imaginándonos* que tuviera otros 32 bits a 0 a la izquierda. Para sumar estos dos números de 64 bits en doble precisión (usando sumas de 32 bits), empezaremos sumando las partes menos significativas (E y EAX), puesto que puede haber acarreo. Luego sumaremos las partes más significativas (0 y EDX... y el acarreo si lo hubiera).

Afortunadamente existe la instrucción de suma con acarreo ADC (consultar Apéndice 2, Tabla 4) que permite sumar dos valores y el flag CF. La modificación necesaria es por tanto eliminar el salto JNC, el INC y la etiqueta posterior, y sustituirlos por una suma con acarreo sobre EDX (*pensarlo*) inmediatamente posterior a la suma de las partes menos significativas. Volver a probar el programa con los cuatro ejemplos usados hasta ahora: 16 veces 1, 16 veces `0x0fff ffff`, 16 veces `0x1000 0000`, 16 veces `0xffff ffff`. A lo mejor conviene imprimir EDX y EAX separadamente para facilitar la lectura.

Un problema que tenemos es que cada vez que cambiamos de ejemplo tenemos que volver a teclear los 16 valores. Una solución hubiera sido dejar comentada la lista completa de cada ejemplo, y descomentar en cada momento sólo el ejemplo deseado. O tal vez, como se trata de cuatro líneas iguales, dejar comentada una línea de cada ejemplo, para poder copiarla rápidamente cuatro veces cuando sea necesario.

Otra recomendación que facilita enormemente incorporar una batería de test al código fuente es la mostrada en la Figura 16, basada en las directivas `.irpc` y `.macro` de `as` (ver Apéndice 2, Tabla 9). Como la mayoría de los ejemplos consisten en repetir 4 veces una línea que emite 4 enteros, se pueden definir separadamente las líneas deseadas, pudiendo cambiar de ejemplo sin más que descomentar la definición deseada.

```

.section .data
    .macro linea
        # .int 1,1,1,1
        # .int 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff
        ...
        # .int 5000000000,5000000000,5000000000,5000000000
    .endm
lista: .irpc i,1234
        linea
    .endr

```

Figura 16: uso de las directivas `.irpc` y `.macro` para preparar una batería de pruebas

Cambiar de ejemplo se reduce entonces a comentar una línea y descomentar otra. Para aún mayor comodidad, se podría acomodar esta batería para compilación condicional con el preprocesador `cpp`, aprovechando que de todas formas ya estamos “compilando” el programa usando `gcc` para conseguir el enlazado con `libC` automáticamente. La batería y el formato `printf` podrían quedar entonces de la siguiente manera:

```

.section .data
#ifdef TEST
#define TEST 9
#endif
    .macro linea                                # Resultado - Comentario
        #if TEST==1                            // 16 - ejemplo muy sencillo
            .int 1,1,1,1
        #elif TEST==2                          // 0x0 ffff fff0, casi acarreo
            .int 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff
        ...
        #elif TEST==8                          / 11 280 523 264 << 16x5e9= 80e9
            .int 5000000000,5000000000,5000000000,5000000000
        #else
            .error "Definir TEST entre 1..8"
        #endif
    .endm
    ...
formato: .ascii "resultado \t = %18lu (uns)\n"
          .ascii "\t\t = 0x%18lx (hex)\n"
          .asciz "\t\t = 0x %08x %08x \n"

```

Figura 17: batería de pruebas acomodada para compilación condicional con `cpp`, y formato recomendado

En la Figura 17 se muestran los resultados incorporados al propio código fuente de forma que quede éste autodocumentado, en un formato muy similar a la propia definición de la batería en la página 16. Con esta recomendación se puede ejecutar toda la batería con una única línea de comando en el `shell` (un bucle `for` del `bash`), como el mostrado en la Figura 18.

```

ubuntu@ubuntu-VirtualBox:~/estruct/p2$ for i in $(seq 1 9); do
> rm media
> gcc -x assembler-with-cpp -D TEST=$i -no-pie media.s -o media
> printf "__TEST%02d__%35s\n" $i " " | tr " " "-" ; ./media
> done
rm: cannot remove 'media': No such file or directory
__TEST01__-----
resultado          =                16 (uns)
                  = 0x                10 (hex)
                  = 0x 00000000 00000010
__TEST02__-----
...
ubuntu@ubuntu-VirtualBox:~/estruct/p2$

```

Figura 18: uso de un bucle `for` para comprobar la batería de tests aprovechando compilación condicional

Comprobar el correcto funcionamiento del programa por lo menos para la siguiente batería de 8 tests. Notar que cuando el ejemplo se ofrece en decimal se espera que el resultado también se exprese en decimal, y similarmente para hexadecimal (ejemplo hex \Rightarrow resultado hex). Notar que para facilitar la lectura de las cantidades, se espera que los dígitos decimales se agrupen de 3 en 3, y los hexadecimales de 4 en 4. Salvo el 5º ejemplo en decimal (que es idéntico al 4º, escrito en hex) y el último (que incluye un error de representación numérica), todos los ejemplos se pueden calcular mentalmente sin

necesidad de calculadora, motivo por el cual unos están en decimal y otros en hexadecimal. Usar el accesorio “Calculadora” de Ubuntu en modo programador si surgen dudas sobre algún resultado.

Test #	Ejemplo	Resultado	Comentario
TEST01	[1, ...]	16	Todos los elementos a 1
Ejemplos fáciles de calcular en hexadec.			
TEST02	[0x0FFF FFFF, ...]	0x? ???? ????	Máximo elem e para que 16e no produzca acarreo
TEST03	[0x1000 0000, ...]	0x? ???? ????	Mínimo elem e para que 16e sí produzca 1 acarreo
TEST04	[0xFFFF FFFF, ...]	0x? ???? ????	Notar agrupación de 4 en 4 dígitos hexadec.
TEST05	[-1, ...]	?? ??? ??? ???	Notar agrupación de 3 en 3 dígitos decimales
Ejemplos fáciles de calcular en decimal			
TEST06	[200 000 000, ...]	? ??? ??? ???	Máx e $A \cdot 10^B$ p/suma representable uns32b $\leq 4Gi-1$
TEST07	[300 000 000, ...]	? ??? ??? ???	Mín el $A \cdot 10^B$ p/suma produzca 1CF uns32b $(\geq 4Gi)$
TEST08	[5 000 000 000, ...]	?? ??? ??? ???	Mín elm tipo $A \cdot 10^B$ no representable uns32b $\geq 4Gi$
	+ TEST05	+ ?? ??? ??? ???	si al resultado incorrecto del TEST08
	+ skin 16	+ 16	se le suma el TEST05 y const. Skinner 16
	=	80 000 000 000	se obtiene el resultado correcto. <i>¿Por qué?</i>

Se han tachado los resultados del TEST05 y TEST08 porque son claramente erróneos, debido a la incorrecta interpretación y/o representación de los elementos de la lista. Coincide que sumando esos resultados tachados junto con el valor 16 se obtiene el resultado correcto del TEST08. *¿Por qué?* ¿Funciona el 16 como constante de Skinner, o se puede explicar por qué hace falta?

5.3 Sumar N enteros con signo de 32 bits sobre dos registros de 32 bits (mediante extensión de signo, naturalmente) ($N \approx 16$)

Se desea ahora interpretar los números de la lista como enteros con signo. Correspondientemente, la variable de 64 bits en donde se almacene el resultado también se interpretará con signo. Eso implica realizar cambios al programa. En concreto, el flag de acarreo (CF) ya no significa que haya error en el resultado, e insistir en acumular los acarreo produciría un resultado sólo comprensible si los datos originales vuelven a interpretarse sin signo (reparar el TEST04 y TEST05 de la anterior batería de tests).

Notar que para una lista tan sencilla como por ejemplo 16 veces -1, el resultado del programa con signo debería ser -16 (0xfffffff ffffffff), mientras que con nuestro programa actual (sin signo) se obtiene casi 64Giga (=0xf ffff fff0). La diferencia radica en que nuestro programa sin signo extiende con ceros (*aunque sea imaginariamente* en la solución de la sección 5.1) los datos originales, mientras que para poder acumular números de 32 bits con un acumulador de 64 bits se debería realizar extensión de signo 32→64 bits al dato original antes de sumarlo al acumulador. Si aún no se ve claro, otro ejemplo más convincente por su sencillez sería `lista[]={0,-1}`, para el cual nuestro programa actual calcula `0x00000000 ffffffff= 232-1 = 4Giga-1` (tras haber extendido -1 con ceros, interpretándolo por tanto sin signo), cuando lo que deseamos sería `0xfffffff ffffffff=-1` (*extendiendo -1 con unos y 0 con ceros,* interpretando por tanto ambos elementos con signo).

Tomando como partida el programa ensamblador del apartado anterior (en donde $0-1 \neq -1$ porque -1 se interpreta sin signo), nos interesaría añadir alguna instrucción de extensión de signo (Apéndice 2, Tabla 3 y Tabla 5) tras *leer* (no *sumar*) cada elemento de la lista, para pasarlo a doble precisión *antes* de acumular. Las instrucciones que nos interesan no son las de tipo MOVX ni CBW/cbtw, porque no queremos usar los registros de 64 bits, sino las de tipo CWD/cwtq, porque queremos pasar de uno a dos registros de 32 bits. Probablemente querremos *leer (no sumar)* el elemento en EAX, extender a dos registros de 32 bits, sumar esos 64 bits (ADD para los LSB, ADC para los MSB) con otros 2 registros usados como acumulador (se puede usar EBP si fuera necesario), y antes de retornar, mover la suma acumulada de 64 bits a EDX:EAX, que es donde la espera el programa principal. Como ya dijimos, cuando veamos la instrucción de división quedará claro por qué se desea la suma en EDX:EAX.



cambiado. Coincide que sumándole (2 x TEST06) al TEST19, se obtiene el resultado correcto del TEST19. ¿Por qué? ¿Funciona (2 x TEST09) como constante de Skinner, o se puede explicar por qué hace falta?

En la Figura 19 se muestran los resultados incorporados al propio código fuente de forma que quede éste autodocumentado, en un formato muy similar a la propia definición de la batería de tests.

```
.section .data
#ifdef TEST
#define TEST 20
#endif

    .macro linea                                # Resultado: Comentario
    #if TEST==1                                // -16: ejemplo muy sencillo
        .int -1,-1,-1,-1
    #elif TEST==2                               // 0x0 4000 0000: incluso sgn32b
        .int 0x04000000, 0x04000000, 0x04000000, 0x04000000
    ...
    #elif TEST==19                              // 20 719 476 736>>16x-3e9= -48e9
        .int -3000000000,-3000000000,-3000000000,-3000000000
    #else
        .error "Definir TEST entre 1..19"
    #endif
    .endm

...
...
formato:    .ascii "resultado \t =  %18ld (sgn)\n"
            .ascii      "\t\t = 0x%18lx (hex)\n"
            .asciz      "\t\t = 0x %08x %08x \n"
```

Figura 19: batería de pruebas acomodada para compilación condicional con cpp, y formato recomendado

5.4 Media y resto de N enteros con signo de 32 bits calculada usando registros de 32 bits (N≈16)

Modificar el programa anterior para que calcule la media y resto de la lista de números. La instrucción de división a añadir (Apéndice 2, Tabla 5) viene obligada, teniendo en cuenta que se trata de números con signo (*¿cuál debe ser?*). También viene obligado en qué registros debe ponerse el dividendo, y en dónde quedarán el cociente y el resto (*¿dónde?*), que ya no son de 64 bits. Añadir la instrucción de división en la subrutina, y salvar el cociente y el resto tras volver de la subrutina, en variables del tamaño (y tipo) apropiado.

Aunque el libro [1] utilizado en las clases no explica la extensión de signo con la instrucción CLTD, sí que explica las instrucciones CLTQ (pág. 221) y CQTO (pág.234), mostrando un ejemplo de su uso antes de IDIV (pág. 236. En la página 221 menciona la posibilidad de extender con MOVSLQ, y también es posible realizar esa operación con MOV y SAR \$31 (*pensarlo*), forma usada frecuentemente por gcc. La explicación empieza en la página 233, Sección 3.5.5 “*Special Arithmetic Operations*”, y concluye con el Problema 3.12 en la página 236 (resuelto en la página 365), sugiriendo las modificaciones que habría que realizar al código anterior (pág. 235) para que la división fuera sin signo. El ejemplo anterior (pág.234, desensamblado en la pág.235) explica la multiplicación sin signo en doble precisión.

Los ejemplos que hemos sugerido para las baterías anteriores consistían en repetir siempre el mismo número, buscando casos que producirían *overflow* o acarreo en caso de realizarse sobre un acumulador de 32 bits. Con esos ejemplos no aprenderemos mucho sobre la división, ya que la media debe ser siempre dicho valor repetido. Sugerimos esta vez casos más sencillos para descubrir si la instrucción de división entera IA32 funciona como la división euclidiana “normal” usada en aritmética modular (cociente redondeado hacia $\pm\infty$ según se obtenga resto positivo) o truncada (cociente redondeado hacia 0, resto del mismo signo que el dividendo). Como informáticos, deberíamos conocer con precisión las reglas de evaluación aritmética de los lenguajes en los que aprendemos a programar, incluyendo la precedencia de los distintos operadores, el tratamiento del acarreo/*overflow*, el signo del resto en la división entera (¿mismo signo que numerador?), etc.

Recordar que recomendamos “*compilar*” usando gcc para aprovechar *printf* de *libc* y la compilación condicional de *cpp* para la batería de tests. Para que *printf* imprima ahora resultados de 32 bits con signo se debe modificar de nuevo el formato para ajustarse al tipo de datos del cociente y del resto. Se

pueden imprimir ambos en una misma llamada *printf*, de nuevo en decimal y hexadecimal, en dos líneas de pantalla si se desea (usando los formatos "\n" y "\t" para saltar de línea y alinear con tabulaciones), controlando incluso los dígitos de anchura (y ceros a la izquierda) de cada número ("%11d", "0x%08x").

Comprobar el correcto funcionamiento del programa por lo menos para la siguiente batería de 42 tests. Recordar que el resultado se debe expresar en la misma base en que se ofrece el ejemplo (decimal o hexadecimal), que todos los ejemplos son fáciles de calcular mentalmente (por eso unos están en decimal y otros en hexadecimal), y que para facilitar la lectura de las cantidades, los dígitos decimales deben agruparse de 3 en 3 y los hexadecimales de 4 en 4. Siempre se puede usar el accesorio "Calculadora" de Ubuntu en modo programador si surgen dudas sobre algún resultado.

Test #	Ejemplo	Media	Resto	Comentario
TEST01	[1, 2, 1, 2, ...]	1	8	Cíclicamente 1,2... euclídea o truncada?
TEST02	[-1, -2, -1, -2, ...]	-?	-?	Cíclicamente -1,-2... aclarado
Ejemplos fáciles de calcular en hexadecimal				
TEST03	[0x7FFF FFFF, ...]	0x???? ?????	0x?	...positivo grande (máx. elem. sgn32b)
TEST04	[0x8000 0000, ...]	0x???? ?????	0x?	...negativo grande (mín. elem. sgn32b)
TEST05	[0xFFFF FFFF, ...]	0xFFFF FFFF	0x0	...negativo más pequeño (equiv. TEST15)
Ejemplos fáciles de calcular en decimal				
TEST06	[2 000 000 000, ...]	? ??? ? ? ? ? ?	? ? ? ? ? ?	...fácil ver q. suma cabe sgn32b (<=2Gi-1)
TEST07	[3 000 000 000, ...]	-? ? ? ? ? ? ? ? ?	? ? ? ? ? ?	...pos+gran A·10 ^B suma uns32b (<=4Gi-1)
TEST08	[-2 000 000 000, ...]	-? ? ? ? ? ? ? ? ?	? ? ? ? ? ?	...pos+peq A·10 ^B suma no uns32b(>=4Gi)
TEST09	[-3 000 000 000, ...]	+? ? ? ? ? ? ? ? ?	? ? ? ? ? ?	...pos+gran A·10 ^B repr. sgn32b (<=2Gi-1)
Ejemplos fáciles decimal: división eucl. vs. trunc.				
TEST10	[0, 2, 1, 1, 1, 1, ...]	?	?	Toda la lista a 1 salvo 2º elemento a 2, y
TEST11	[1, 2, 1, 1, 1, 1, ...]	?	?	barrido del 1º elemento a 0,1,8,15,16
TEST12	[8, 2, 1, 1, 1, 1, ...]	?	?	(barrido para ver si resto eucl./trunc.)
TEST13	[15, 2, 1, 1, 1, 1, ...]	?	??	
TEST14	[16, 2, 1, 1, 1, 1, ...]	?	?	Ventajoso pasar 1º "1" a 2º elemento
TEST15	[0,-2,-1,-1, -1,-1, ...]	-1	0	Igual pero en negativo (equiv. a TEST05)
TEST16	[-1,-2,-1,-1, -1,-1, ...]	?	?	
TEST17	[-8,-2,-1,-1, -1,-1, ...]	?	?	
TEST18	[-15,-2,-1,-1, -1,-1, ...]	?	??	
TEST19	[-16,-2,-1,-1, -1,-1, ...]	?	?	Ventajoso pasar 1º "-1" a 2º elemento

Como *podemos comprobar*, la instrucción de división en Intel 64 se corresponde con lo que la Wikipedia denomina división truncada (buscar *Módulo*), lo cual tiene una implicación respecto al signo del módulo (*¿cuál?*). *¿Coincide IDIV* con los operadores / (división entera) y % (módulo) en los lenguajes C/C++?

En la Figura 20, además del formato *printf* recomendado y la técnica de autodocumentación del código fuente (incorporando los resultados), se muestra también una posible técnica con dos macros para implementar los últimos tests a partir del 10, en donde toda la lista vale un mismo valor (± 1) salvo los dos primeros elementos. ¿Por qué se afirma en los comentarios de los tests 14 y 19 que es "ventajoso acumular el 1º elemento al 2º"? ¿Qué significa esa frase?

```
.section .data
#ifdef TEST
#define TEST 20
#endif
# doble macro: lista definida por una linea0 y 3 lineas normales
# mayoría ejemplos linea0 = linea => lista tiene 4 lineas normales
```



```

        .macro linea                                # Media / Resto - Comentario
        #if TEST==1                                // 1 / 8
            .int 1, 2, 1, 2
        #elif TEST==2                              // -1 / -8
            .int -1,-2,-1,-2
        ...
        ...
        #elif TEST>=10 && TEST<=14                # linea0 + 3lineas, casi todo a 1
            .int 1, 1, 1, 1
        #elif TEST>=15 && TEST<=19                # linea0 + 3lineas, casi todo -1
            .int -1,-1,-1,-1
        #else
            .error "Definir TEST entre 1..19"
        #endif
        .endm

        # mayoría ejemplos linea0 = linea => lista tiene 4 lineas normales

        .macro linea0                              # Media / Resto - Comentario
        #if TEST>=1 && TEST<=9
            linea                                # casi siempre 4 lineas iguales
        #elif TEST==10                             // 1 / 0 - equiv toda lista 1
            .int 0, 2, 1, 1
        ...
        #elif TEST==15                             // -1 / 0 - equiv TEST05
            .int 0,-2,-1,-1
        ...
        #elif TEST==19                             // ? / ?
            .int -16,-2,-1,-1
        #else
            .error "Definir TEST entre 1..19"
        #endif
        .endm

        # mayoría ejemplos linea0 = linea => lista tiene 4 lineas normales

lista:      linea0
            .irpc i,123
            linea
            .endr

longlista:  .int  (.-lista)/4
media:     .int  0
resto:     .int  0

formato:   .ascii "media \t = %11d \t resto \t = %11d \n"
           .asciz  "\t = 0x %08x \t \t = 0x %08x\n"

```

Figura 20: batería de pruebas acomodada para compilación condicional con cpp, y formato recomendado

5.5 Media y resto de N enteros calculada en 32 y en 64 bits (N≈16)

Añadir al programa anterior otra subrutina `sumaq`: que repita la acumulación, pero esta vez sobre un registro de 64 bits, no sobre dos de 32 bits. La instrucción de *extensión* de signo es ahora *distinta* (Apéndice 2, Tabla 5). Por hacer mínimos cambios, si se extiende EAX→RAX tal vez convendría acumular sobre RDI (dejando sin uso RBP), y moverlo al dividendo RAX (asegurándose de *extenderlo* a RDX:RAX) antes de dividir por el divisor de 64 bits. Aunque el cociente y el resto (*¿dónde estarán?*) sean de 64 bits, es correcto aprovechar solo las partes menos significativas de 32 bits (*¿por qué?*). Se puede seguir salvando el cociente y el resto, tras volver de la subrutina, en las mismas variables, e imprimirlas con el mismo formato (o tal vez con otra copia `formatq`: en la que avise de que los cálculos se realizaron en 64 bits). Lo que sí necesitaremos cambiar será la etiqueta `bucle`: (a `bucleq`:, por ejemplo).

El programa llamaría a `suma`: y a `printf` (con el mismo `formato`: del Apartado 5.4), y posteriormente llamaría a `sumaq`: y a `printf` con el nuevo `formatq`:, confiando en que ambos mensajes produzcan el mismo resultado impreso. Se puede reutilizar la misma batería de tests, naturalmente.

6 Entrega del trabajo desarrollado

Los distintos profesores de prácticas indicarán las normas de entrega para cada grupo, incluyendo qué se ha de entregar, cómo, dónde y cuándo.

Por ejemplo, puede que en un grupo el profesor habilite que los estudiantes que cumplan el régimen de asistencia puedan entregar en SWAD, hasta medianoche del domingo de la última semana de esta práctica, el fichero ensamblador correspondiente al último Apartado que hayan conseguido implementar (código fuente, listo para “compilar” con `gcc -D TEST=$i` y para reproducir la batería con `for i in $(seq 1 ...);` usando la técnica mostrada en la Figura 18) y un fichero .txt con los resultados de la batería de test, volcado íntegramente de la pantalla (incluso redireccionando la salida estándar) en el orden y con el formato indicado en este guión de prácticas. Puede que en otro grupo el profesor de prácticas proporcione otras instrucciones distintas.

En cualquier caso, **el examen de prácticas** a final del curso (Test TP de 4 puntos) **es el mismo para todos los grupos**, lo cual significa que independientemente del profesor de prácticas, cada estudiante es responsable de comprender todo lo explicado en estos **guiones de prácticas (que son comunes a todos los grupos y son materia de examen)** y desarrollar las habilidades que se ejercitan en los mismos (que también son materia de examen).

7 Bibliografía

- [1] Apuntes y presentaciones de clase
Libro CS:APP: Randal E. Bryant, David R. O’Hallaron: “Computer Systems: A Programmer’s Perspective”, 3rd Ed., Pearson, 2015. <http://csapp.cs.cmu.edu/>
- [2] Jonathan Bartlett, “Programming from the Ground Up”. Ed. Dominick Bruno, Jr.
<http://savannah.nongnu.org/projects/pgubook/>
- [3] Manuales de Intel sobre IA-32 e Intel64, en concreto el volumen 2: “Instruction Set Reference”
<https://software.intel.com/en-us/articles/intel-sdm>
<https://software.intel.com/en-us/articles/intel-sdm#three-volume>
<https://software.intel.com/sites/default/files/managed/a4/60/325383-sdm-vol-2abcd.pdf>
- [4] Resumen del repertorio de instrucciones básico x86 (mencionado en transparencias clase)
<http://www.jegerlehner.ch/intel/IntelCodeTable.pdf>
http://www.jegerlehner.ch/intel/IntelCodeTable_es.pdf
- [5] GNU binutils, artículo Wikipedia: http://en.wikipedia.org/wiki/GNU_Binutils
Sitio web <http://www.gnu.org/software/binutils/>
Manuales (incluyendo gas, ld, nm...) <http://sourceware.org/binutils/docs/>
- [6] GNU Debugger, GDB, Wikipedia <http://en.wikipedia.org/wiki/Gdb>
Sitio web <http://www.gnu.org/s/gdb/>
Manuales <http://sourceware.org/gdb/current/onlinedocs/gdb/>
Chuletario <http://www.csd.uoc.gr/~hy255/refcards/gdb-refcard.pdf>
Resumen comandos <http://web.cecs.pdx.edu/~jrb/cs201/lectures/handouts/gdbcomm.txt>
- [7] Data Display Debugger, DDD, Wiki http://en.wikipedia.org/wiki/Data_Display_Debugger
Sitio web <http://www.gnu.org/s/ddd/>
Manuales (incluye tutorial) http://www.gnu.org/s/ddd/manual/html_mono/ddd.html
http://www.gnu.org/s/ddd/manual/html_mono/ddd.html#Sample%20Session
- [8] Código ASCII, Wikipedia <http://en.wikipedia.org/wiki/ASCII>
Código UTF-8 <http://en.wikipedia.org/wiki/UTF-8>
Tabla ASCII http://en.wikipedia.org/wiki/File:ASCII_Code_Chart-Quick_ref_card.png

Apéndice 1. Tabla de caracteres ASCII

En algún momento (Figura 11) se han interpretado 4 bytes de un *string* como un entero. Para comprobar que el resultado obtenido no es en absoluto impredecible, sino que es matemáticamente preciso, se proporciona la tabla de códigos ASCII de 7 bits en la que están presentes todos los caracteres utilizados en aquel *string*.

	0	1	2	3	4	5	6	7
0	NUL	DEL	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(8	H	X	h	x
9	HT	EM)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[k	{
C	FF	FS	,	<	L	\	l	
D	CR	GS	-	=	M]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Por ejemplo, si un programa definiera un *string* "Hello", y posteriormente el *string* se interpretara como entero, el valor entendido por el procesador dependería del tamaño de entero (usualmente la longitud de palabra) y el ordenamiento de bytes (si la memoria está organizada en bytes). Los procesadores x86-64 siguen la convención del extremo menor, así que el *string* interpretado como entero de 4B sería 0x6c6c6548. Observar cómo se toman las primeras cuatro letras "Hell", y el byte menos significativo es el que está almacenado en la posición de memoria más baja "H".

Apéndice 2. Resumen de la arquitectura de los repertorios x86 y x86-64, y del ensamblador GNU AS.

Los 8 registros x86 de propósito general se pueden acceder en tamaño byte (AH...BL), palabra de 16 bits (AX...BP, para modo 16 bits), y doble palabra de 32 bits (EAX...EBP). Hay un registro de flags (EFLAGS) y un contador de programa (EIP).

Registros enteros IA-32			nombre
%eax	%ax	%ah %al	acumulador
%ecx	%cx	%ch %cl	contador
%edx	%dx	%dh %dl	datos
%ebx	%bx	%bh %bl	base
%esi	%si		índice fuente
%edi	%di		índice destino
%esp	%sp		puntero pila
%ebp	%bp		puntero base

También existen registros para datos en punto flotante de 64/80 bits (FP0...FP7), registros MMX de 64 bits (MM0...MM7), registros SSE de 128 bits (XMM0...XMM7), etc, que se usan con instrucciones específicas del repertorio.

Existen también registros de segmento (CS,SS,DS,ES,FS,GS, aunque en Linux se usa un modelo de memoria plano, no segmentado). Y registros de control (CRi), depuración (DRI), específicos (MSRi), etc.

Los 16 registros x86-64 de propósito general son de tamaño cuádruple palabra (RAX...RBP + R8...R15), y también puede accederse en tamaños menores: 32 bits (EAX...EBP + R8D...R15D), 16 bits (AX...BP + R8W...R15W), 8 bits (AL...BL + SIL...BPL + R8B...R15B). RFLAGS y RIP también son de 64 bits.

Registros enteros x86-64

%rax	%eax	%ax	%al	%r8	%r8d	%r8w	%r8b
%rcx	%ecx	%cx	%cl	%r9	%r9d	%r9w	%r9b
%rdx	%edx	%dx	%dl	%r10	%r10d	%r10w	%r10b
%rbx	%ebx	%bx	%bl	%r11	%r11d	%r11w	%r11b
%rsi	%esi	%si	%sil	%r12	%r12d	%r12w	%r12b
%rdi	%edi	%di	%dil	%r13	%r13d	%r13w	%r13b
%rsp	%esp	%sp	%spl	%r14	%r14d	%r14w	%r14b
%rbp	%ebp	%bp	%bpl	%r15	%r15d	%r15w	%r15b

Se muestra a continuación un brevísimo resumen de las instrucciones más frecuentemente usadas (en sintaxis Intel como aparecen en el manual, y con ejemplos AT&T como suelen utilizarse en Linux). Recordar que, como norma general, en las instrucciones con 2 operandos sólo 1 puede ser memoria. Siempre se pueden consultar los manuales de Intel para comprobar qué modos de direccionamiento son válidos para cada argumento, y qué flags de estado resultan afectados por cada instrucción.

Instrucciones de movimiento de datos

Instrucción	Efecto	Descripción	Ejemplos
MOV	D, S	$D \leftarrow S$	Mover fuente a destino (mismo tamaño). Variantes AT&T: tamaño byte, word, long... movb %al, %bl movl \$0xf02, %ecx
MOVS	D, S	$D \leftarrow \text{ExtSigno}(S)$	Mover con extensión de signo (aumentando tamaño) Variantes AT&T: byte-to-word, byte-long, word-long... movsbw %al, %bx movslq %ecx, %rdx
MOVZ	D, S	$D \leftarrow \text{ExtCero}(S)$	Mover rellenando con ceros (aumentando tamaño) movzbq %al, %rbx movzbl %cx, %edx movzwl %cx, %edx
PUSH	S	$\text{ESP} \leftarrow \text{ESP} - S $ $M[\text{ESP}] \leftarrow S$	Meter fuente en pila pushq \$8
POP	D	$D \leftarrow M[\text{ESP}]$ $\text{ESP} \leftarrow \text{ESP} + S $	Sacar de pila a destino popq %rbp
LEA	D, S	$D \leftarrow \&S$	Cargar dirección efectiva leaq tabla(%rax,4), %rsi

Tabla 3: Instrucciones x86 - transferencia

Observar que la sintaxis AT&T intercambia el orden de los argumentos (S, D) e incorpora un sufijo con el tamaño de operando (byte, word, long) o incluso de la conversión realizada (byte-to-long...). Notar que los datos inmediatos se prefijan con \$, los registros con %, y la sintaxis para direccionamiento a memoria es `Desplaz(Rbase, Ríndice, FactEscala)`, pudiendo omitirse los componentes no deseados.

Instrucciones aritmético-lógicas

Instrucción	Efecto	Descripción	Instrucción	Efecto	Descripción
INC	D	$D \leftarrow D + 1$	NEG	D	$D \leftarrow -D$
DEC	D	$D \leftarrow D - 1$	NOT	D	$D \leftarrow \sim D$
ADD	D, S	$D \leftarrow D + S$	XOR	D, S	$D \leftarrow D \wedge S$
SUB	D, S	$D \leftarrow D - S$	OR	D, S	$D \leftarrow D \vee S$
ADC	D, S	$D \leftarrow D + S + C$	AND	D, S	$D \leftarrow D \& S$
SBB	D, S	$D \leftarrow D - (S + C)$	IMUL	D, S	$D \leftarrow D * S$
SAL	D, k	$D \leftarrow D \ll k$	RCL	D, k	$D \leftarrow D \cup_c k$
SHL	D, k	$D \leftarrow D \ll k$	ROL	D, k	$D \leftarrow D \cup k$
SAR	D, k	$D \leftarrow D \gg_{\text{A}} k$	RCR	D, k	$D \leftarrow D \cup_c k$
SHR	D, k	$D \leftarrow D \gg_{\text{L}} k$	ROR	D, k	$D \leftarrow D \cup k$

Tabla 4: Instrucciones x86 - aritmético-lógicas

No confundir la negación aritmética con el complemento lógico. Recordar que los desplazamientos a derecha pueden ser aritméticos o lógicos según se inserten ceros o copias del bit de signo. Las rotaciones pueden incluir o no el acarreo en el conjunto de bits a desplazar. La instrucción `IMUL` es particular, consultar en el manual de Intel sus diversos modos de direccionamiento (1, 2, 3 operandos).

Instrucciones aritméticas especiales					
Instrucción		Efecto	Descripción	Operandos / Variantes	
MUL	S	$AC_{D:A} \leftarrow AC_A * S$	Multiplicación sin signo	$AX \leftarrow AL * r/m8$ $DX:AX \leftarrow AX * r/m16$	$EDX:EAX \leftarrow EAX * r/m32$ $RDX:RAX \leftarrow RAX * r/m64$
DIV	S	$AC_A \leftarrow AC_{D:A} / S$ $AC_D \leftarrow AC_{D:A} \% S$	División sin signo	$AL(AH) \leftarrow AX / r/m8$ $AX(DX) \leftarrow DX:AX / r/m16$	$EAX(EDX) \leftarrow EDX:EAX / r/m32$ $RAX(RDX) \leftarrow RDX:RAX / r/m64$
IMUL	S Dr,S Dr,S,I	$AC_{D:A} \leftarrow AC_A * S$ $D \leftarrow D * S$ $D \leftarrow D * S * Im$	Multiplicación con signo	1 operando: Como MUL, nbit x nbit = 2nbits 2 operandos: Reg * (R/M/Inmediato), n x n = nbits 3 operandos: Reg = Reg * (R/M/Inm), n x n = nbits	
IDIV	S	$AC_A \leftarrow AC_{D:A} / S$ $AC_D \leftarrow AC_{D:A} \% S$	División con signo	Como DIV	
CBW		$AC_{2n} \leftarrow ExtSign(AC_n)$	Extensión de signo acum.	$AX \leftarrow ExtSign(AL)$	cbtw
CWDE			word-to-long-to-quad	$EAX \leftarrow ExtSign(AX)$	cwtl
CDQE				$RAX \leftarrow ExtSign(EAX)$	cltq
CWD		$AC_{D:A} \leftarrow ExtSign(AC_A)$	Extensión de signo acum.	$DX:AX \leftarrow ExtSign(AX)$	cwtd
CDQ			x-to-double	$EDX:EAX \leftarrow ExtSign(EAX)$	cltd
CQO				$RDX:RAX \leftarrow ExtSign(RAX)$	cqto

Tabla 5: Instrucciones x86 - aritmética especial

En general los productos y divisiones usan implícitamente los registros A y D del tamaño deseado para multiplicar $A(nbits) \times S(nbits) = D:A(2nbits)$, o dividir $D:A(2n) / S(nbits)$ produciendo cociente y resto en A y D(nbits). Las extensiones de signo en acumulador (registros A y D) pueden redactarse en sintaxis Intel o AT&T, `gas` entiende ambas versiones. En modo 32 bits será `c1td` la que probablemente usemos más.

Comparaciones y saltos/ajustes/movimientos condicionales							
Instrucción		Cálculo	Efecto	Instrucción		Efecto	Descripción
CMP	S_1, S_2	$S_1 - S_2$	Ajustar flags según cálculo	SETcc	D (r/m8)	$D \leftarrow 0/1$ según cc se cumpla o no	Códigos: e,ne,z,nz (Z) s,ns (S) o,no (O), c,nc (C), p,np (P) [n]a[e],[n]b[e] (sin signo), [n]l[e],[n]g[e] (con signo)
TEST	S_1, S_2	$S_1 \& S_2$	Ajustar flags según cálculo	CMOVcc	Dreg, S	$D \leftarrow S$ según cc	Mismos códigos
JMP	label		Salto incondicional directo Intel: "relative short/near"	Jcc	label	Saltar label según cc	Mismos códigos
jmp	*Ptr	(AT&T)	Salto incondic. Indirecto				
JMP	PtrDst	(Intel)	"near, absolute, indirect"				

Tabla 6: Instrucciones x86 - comparaciones y condicionales

Las comparaciones y tests permiten calcular qué condiciones se cumplen entre los 2 operandos (recordar que AT&T invierte el orden de los operandos), y posteriormente se puede (des)activar un byte (`SETcc`), realizar o no un movimiento (`CMOVcc`), o saltar a una etiqueta del programa (`Jcc`) según alguna de esas condiciones. Los códigos de condición "cc" pueden referirse a flags sueltos como (Z)ero, (S)ign, (O)verflow, (C)arry, (P)arity, y a combinaciones interpretadas sin signo (Above/Below) y con signo (Less/Greater). En sintaxis AT&T, `jmp *%eax` sería saltar a la dirección indicada en el registro, y `jmp *(%eax)` sería leer la dirección de salto de memoria, donde apunta `%eax`.

Otras instrucciones de control de flujo, y miscelánea						
Instrucción		Efecto	Descripción	Instrucción	Efecto	Descripción
CALL	label	PUSH rip rip ← label	Llamada a subrutina Intel: "near relative"	INT v	PUSH rflags CALL ISR#v	Llamada a ISR Interrupción "Interrupción software"
call	*Ptr	(AT&T)				
CALL	PtrDst	(Intel)	Intel: "near absol. indirect"			
RET		POP rip	Retorno de subrutina	IRET	RET POP rflags	Retorno de ISR
LEAVE		RSP ← RBP POP RBP	Libera marco pila Para usar con ENTER	NOP	No-op	
CLC		$C \leftarrow 0$	Ajustes del flag acarreo	CLI	$I \leftarrow 0$	Ajustes del flag Interrupt
STC		$C \leftarrow 1$		STI	$I \leftarrow 1$	(des)habilitar IRQs
CMC		$C \leftarrow \sim C$				

Tabla 7: Instrucciones x86 - control y miscelánea

El registro de flags (EFLAGS en 32 bits, RFLAGS en 64 bits) contiene (entre otros) los bits aritméticos C(arry), O(verflow), S(ign), Z(ero), un bit P(arity) ajustado a impar, y un bit de habilitación I(nterrupt).

