5 Trabajo a realizar

Se propone mejorar este último programa (Sección 4, Figura 12, **suma.s**) para calcular la media de una lista de N enteros, evitando la posibilidad de desbordamiento aritmético (*overflow*). Necesitaremos utilizar las instrucciones de división con signo, extensión de signo y suma con acarreo, para realizar la suma en múltiple (doble) precisión.

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, responder a las preguntas (Tabla 10, Tabla 11), comprender los programas mostrados (Figura 10, Figura 12) y ejercitarse en el uso de las herramientas 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:

5.0 Contestar las preguntas de autocomprobación (saludo.s, suma.s)

El objetivo es comprender con detalle el funcionamiento de las instrucciones mostradas, la ubicación de código y datos en memoria, el funcionamiento de la pila y llamadas y retornos de subrutinas, y adquirir habilidad en el manejo de las herramientas usadas (compilador, ensamblador, enlazador y depurador).

5.1 Sumar N enteros *sin* signo de 32bits en una plataforma de 32bits sin perder precisión (N≈32)

Realizar un programa ensamblador con una función que sume una lista de N números en binario natural (sin signo) de 32bits, sin perder dígitos. Notar que la suma puede necesitar más de 32bits. En las preguntas de autocomprobación del Apéndice 3, Tabla 12, se pide calcular cuántos bits adicionales pueden llegar a necesitarse para almacenarse el resultado, si N=32. Razonar que si se hace una suma en doble precisión (32+32bits) no habrá desbordamiento para cualquier valor práctico de N.

La suma en doble precisión implica conservar los acarreos que de otra forma se hubieran perdido (instrucción etiquetada `bucle:` en `suma.s`, Figura 12, tras la cual no se comprueba si hay acarreo saliente de EAX). Si los vamos acumulando en otro registro de 32bits (p. ej. EDX), la concatenación de ambos (EDX:EAX) puede almacenar el resultado (sin signo, de 64bits). Para comprobar y/o sumar el acarreo pueden usarse la suma con acarreo ADC o el salto condicional según haya acarreo o no JC/JNC, posiblemente combinado con INC. Probablemente necesitemos usar otro registro como índice (tal vez ESI) en lugar de EDX. Cuando veamos la instrucción DIV para calcular la media, se entenderá el interés de acumular en EDX:EAX.

Se puede tomar como punto de partida el programa `suma.s`, añadiendo directivas `.int` adicionales para los N=32 valores (por ejemplo, 8 líneas de 4 valores) y cambiando el tamaño de `resultado` a 64bits (consultar en Apéndice 2, Tabla 9, la directiva `as` para ello). La subrutina debería devolver el resultado en EDX:EAX, y el programa principal almacenarlo en la variable `resultado` de 64bits, según el criterio del extremo menor (recordar que la familia x86 es *little-endian*). La dirección que empieza 4 posiciones detrás de `resultado` se indica `resultado+4`, naturalmente. Desde ddd debería poder visualizarse el valor correcto de la suma con Data->Memory->*Examine 1 decimal giant from &resultado*.

También se podría redactar el progama en C y observar qué código genera gcc con distintos niveles de optimización (`-O1`, `-O2`). La función se declararía como `unsigned long long suma(unsigned* lista, int longlista)`. La lista se podría inicializar con la sintaxis `unsigned lista[]={1,2 [,…] }`, y se podría imprimir el resultado con `printf("resultado=%llu\n", suma(lista,longlista))`. También se podría usar el formato `"%llx"` para ver el resultado en hexadecimal (o usar ddd).

Para saber si el resultado es correcto, se pueden probar ejemplos pequeños (todos los elementos a 1 suman 32, todos a 2 suman 64, cíclicamente 1,2,3,4 suman 80, etc.) pero es costumbre usar ejemplos que ejerciten todas las posibilidades del código. En este caso interesan ejemplos que produzcan acarreos, incluso varios acarreos, hasta uno por cada suma. En las preguntas de autocomprobación, Apéndice 3, Tabla 12, se sugieren ejemplos con más posibilidades de detectar algún problema en nuestro programa, si lo hubiera.

Para probar una batería de ejemplos es relativamente fácil modificar los datos en el código fuente y volver a ensamblar y enlazar el programa, pero depurarlo para ver el resultado final puede volverse tedioso. Para facilitar (de hecho, evitar) esta tarea se propone utilizar la llamada printf de la librería libC. En la Figura 12 aparecen comentados el string de formato, el punto de entrada (main) y la secuencia de llamada para introducir en la pila el argumento a imprimir (dos veces) y el propio formato (para imprimir el argumento tanto en decimal como en hexadecimal). Si se sigue esta recomendación, será necesario compilar el programa con gcc. Comprobar primero que printf funciona con el programa `suma.s`. Para que funcione con esta modificación (5.1) cuando el resultado sobrepase los 32bits, será necesario meter en la pila los 64bits de la suma (2 veces) y modificar el formato para que maneje datos de tipo `unsigned long long` (`"%llu"`, `"%llx"`). Si se prefiriera usar `"%08x,%08x"` en lugar de `"%llx"`, los 64bits se meterían en orden inverso, primero los 32bits menos significativos, para que se impriman primero los 32bits más significativos. Cuando los 64bits forman una sola palabra, se meten primero los MSB (*little-endian*).

Comprobar el correcto funcionamiento del programa por lo menos para la siguiente batería de 6 tests:

Ejemplo	Resultado	Comentario
[1, …]	32	Todos los elementos a 1
[2, …]	64	Todos los elementos a 2
[1, 2, 3, 4, 1, …]	80	Cíclicamente los valores 1,2,3,4
[-1, …] **o también [0xFFFFFFFF, …]**	?	Todos los elementos al valor indicado
[0x08000000, …]	?	Todos los elementos al valor indicado
[0x10000000, 0x20…, 0x40…, 0x80…, 0x10…]	?	Cíclicamente los valores indicados

Otra recomendación que facilita enormemente modificar los datos en el código fuente es la mostrada en la Figura 13, basada en las directivas `.macro` y `.irpc` (ver Apéndice 2, Tabla 9). Como la mayoría de los ejemplos consiste en repetir 8 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 2,2,2,2
         #     .int 1,2,3,4
         #     .int -1,-1,-1,-1
         #     .int 0xffffffff,0xffffffff,0xffffffff,0xffffffff
         #     .int 0x08000000,0x08000000,0x08000000,0x08000000
         #     .int 0x10000000,0x20000000,0x40000000,0x80000000
        .endm
lista: .irpc i,12345678
            linea
        .endr
```

Figura 13: uso de las directivas .macro y .irpc para preparar una batería de pruebas

5.2 Sumar N enteros *con* signo de 32bits en una plataforma de 32bits

Modificar el programa anterior para que los números de la lista se consideren como enteros con signo. Correspondientemente, la variable de 64bits en donde se almacene el resultado también será con signo.

En lenguaje C la declaración de la función cambiaría a `long long suma(int* lista, int longlista)`, y la de los datos a `int lista[]={1,2 [,…] }`. Si se redacta el programa C (para observar qué código se genera), comparar el resultado de ambas versiones, con y sin signo, para una lista sencilla como por ejemplo {-1,-1,-1,-1}. El resultado del programa con signo debe ser -4 (0xffffffff fffffffc), mientras que sin signo se obtiene casi 16Giga (=0x00000003 fffffffc). La diferencia está en que no se realiza extensión de signo 32→64bits para sumar números sin signo. Si aún no se ve claro, otro ejemplo más sencillo sería {0,-1}, en donde se obtiene 0x00000000 ffffffff= 2^{32}-1 = 4Giga-1 sin signo, y 0xffffffff ffffffff=-1 con signo.

Tomando como partida el programa ensamblador del apartado anterior (en donde -1+0≠-1 porque -1 se interpreta sin signo), nos interesaría añadir alguna instrucción de extensión de signo (Apéndice 3, Tabla 3 y Tabla 5) tras leer cada elemento del array, para pasarlo a doble precisión antes de acumular. Las instrucciones que nos interesan no son las de tipo MOVSX ni CxxE, porque no queremos usar los registros de 64bits en una plataforma de 32bits, sino las de tipo CBW, porque queremos pasar de uno a dos registros de 32bits. Probablemente querremos *leer (**no** sumar)* el elemento en EAX, extender a dos registros de 32bits, sumar esos 64bits (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 64bits 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.

Para saber si el resultado es correcto, se pueden probar ejemplos pequeños (todos los elementos a -1 suman -32, 16 parejas [1,-2] suman -16, cíclicamente 1,2,-3,-4 suman -32, etc.) pero en las preguntas de autocomprobación se sugieren ejemplos con más posibilidades de detectar algún problema en nuestro programa, si lo hubiera.

Recordar que será necesario compilar el programa con gcc, si se sigue la recomendación de usar la llamada libC printf. Para que printf funcione con esta modificación (5.2) sigue siendo necesario meter en la pila los 64bits de la suma y modificar el formato, en este caso para que maneje datos (con signo) de tipo long long (”%lld”, ”%llx”). Recordar también la técnica de la Figura 13, que resulta más útil conforme mayor es el número de ejemplos que desean probarse.

Comprobar el correcto funcionamiento del programa por lo menos para la siguiente batería de tests:

Ejemplo	Resultado	Comentario
[-1, …] o también [0xFFFFFFFF, …]	-32	Todos los elementos a -1
[1, -2, 1, -2, …]	-16	Cíclicamente los valores 1,-2
[1, 2, -3, -4, 1, …]	-32	Cíclicamente los valores 1,2,-3,-4
[0x7FFFFFFF, …]	?	Todos un gran positivo
[0x80000000, …]	?	Todos un gran negativo
[0x04000000, …]	?	Positivo menor
[0x08000000, …]	?	Positivo intermedio
[0xFC000000, …]	?	Negativo menor
[0xF8000000, …]	?	Negativo intermedio
[0xF0000000, 0xE0…, 0xE0…, 0xD0…, 0xF0…]	?	Cíclicamente los valores indicados

5.3 Media de N enteros *con* signo de 32bits, en plataforma de 32bits

Modificar el programa anterior para que calcule la media 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 quedará el cociente (¿dónde?), que ya no es de 64bits. 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.

En lenguaje C sólo se puede devolver directamente un resultado. La declaración de la función cambiaría a `int media(int* lista, int longlista)`, y la función podría acabar con `return suma/longlista` si se ha acumulado en `long long suma=0`. Para devolver también el resto, se podría pasar como argumento la dirección de la variable resto (`int media(int* lista, int longlista, int* resto)`), y sería la subrutina la responsable de asignar el valor del resto.

Para saber si el resultado es correcto, se pueden probar todos los ejemplos sugeridos anteriormente. En la mayoría de ellos la lista repite 32 veces un mismo elemento, que termina siendo el valor de la media (en el caso cíclico 1,2,-3,-4, la media es -1). En las preguntas de autocomprobación sugerimos esta vez

casos más sencillos, pero que hay que probar con ddd para aprender cómo funciona la instrucción de división.

Recordar que será necesario compilar el programa con gcc, si se sigue la recomendación de usar la llamada libC printf. Para usar printf provechosamente con esta modificación (5.3) 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), incluso controlando los dígitos de anchura de cada número ("%8d", "0x%08x"). Recordar también la técnica de la Figura 13, que resulta más útil conforme mayor es el número de ejemplos que desean probarse.

Comprobar el correcto funcionamiento del programa por lo menos para la siguiente batería de tests:

Ejemplo	Media	Resto	Comentario
[1, -2, 1, -2, …]	0	-16	Cíclicamente los valores 1,-2
[1, 2, -3, -4, 1, …]	-1	0	Cíclicamente los valores 1,2,-3,-4
[0x7FFFFFFF, …]	?	0	Todos un gran positivo
[0x80000000, …]	?	0	Todos un gran negativo
[0xF0…, 0xE0…, 0xE0…, 0xD0…, …]	?	?	Cíclicamente los valores indicados
[-1, …]	-1	0	Todos los elementos a -1
[0, -1, -1, -1, …]	?	?	Igual salvo elemento 1º a 0
[0, -2, -1, -1, …]	-1	0	Recuperar el 1º sumándoselo al 2º
[1, -2, -1, -1, …]		?	Habiendo quedado el elemento 1º libre (a 0), hacerlo barrer los valores 0…63, anotando qué se obtiene como cociente y resto
…	…	…	
[32, -2, -1, -1, …]	?	?	
…	…	…	
[63, -2, -1, -1, …]	?	?	
[64, -2, -1, -1, …]	?	?	Continuar el barrido hasta 95
…	…	…	
[95, -2, -1, -1, …]	?	?	
[-31, -2, -1, -1, …]	?	?	Continuar el barrido desde -31
…	…	…	
[0, -2, -1, -1, …]	-1	0	

Como podemos comprobar, la instrucción de división en IA-32 se corresponde con lo que la Wikipedia denomina división truncada (buscar *Módulo*), lo cual tiene una implicación respecto al signo del módulo (¿cuál?). ¿Coinciden estos resultados con el funcionamiento de los operadores / (división entera) y % (módulo, o resto entero) en lenguajes C/C++? ¿En qué consisten las diferencias?

6 Entrega del trabajo desarrollado

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

Por ejemplo, puede que en un grupo se deba entregar en un documento PDF las respuestas a las preguntas de autocomprobación (Apéndice 3, Tabla 10-Tabla 14), los listados de los programas ensamblador (5.1, 5.2, 5.3), y unos breves comentarios sobre por qué se realizó cada modificación (instrucciones añadidas o eliminadas, registros cambiados, etc.) partiendo del programa original `suma.s`, subiéndolo al SWAD hasta 3 días después de la última sesión de prácticas dedicada a esta práctica, con penalización creciente por entrega tardía hasta 1 semana posterior.

Puede que en otro grupo se pueda trabajar y entregar por parejas, pero que el profesor de prácticas visite cada puesto al final de cada sesión comprobando si ambos estudiantes saben responder a las

preguntas, programar en ensamblador y utilizar las herramientas, permitiendo que se suba al SWAD el trabajo en caso afirmativo.

Puede que en otros grupos se deba subir a SWAD el código programado en ensamblador y un documento de texto en PDF indicando el resultado obtenido por cada programa en cada uno de los ejemplos sugeridos en los apartados 5.1, 5.2, 5.3, y en las preguntas de autocomprobación en Apéndice 3, Tabla 12-Tabla 14, y comentando si ha servido o no de ayuda ver el código generado por gcc.

Los profesores de teoría y prácticas de cada grupo acordarán cómo entregará ese grupo el trabajo desarrollado.

7 Bibliografía

[1] Apuntes y presentaciones de clase
Libro CS:APP: Randal E. Bryant, David R. O'Hallaron: "Computer Systems: A Programmer's Perspective", 2nd Ed., Pearson, 2011. 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"
http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-vol-2a-2b-instruction-set-a-z-manual.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 7bits 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 IA-32 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 16bits (AX…BP, para modo 16bits), y doble palabra de 32bits (EAX…EBP). Hay un registro de flags (EFLAGS) y un contador de programa (EIP).

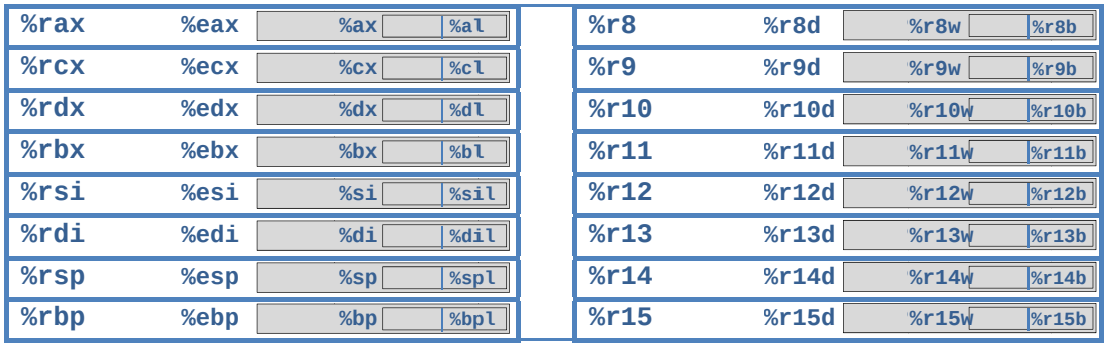
Registros enteros IA-32				nombre
%eax	%ax	%ah	%al	acumulador
%ecx	%cx	%ch	%cl	contador
%edx	%dx	%dh	%dl	datos
%ebx	%bx	%bh	%bl	base
%esi	%si			índice fuente
%edi	%di			índice destino
%esp	%sp			puntero pila
%ebp	%bp			puntero base

También existen registros para datos en punto flotante de 64/80bits (FP0…FP7), registros MMX de 64bits (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: 32bits (EAX…EBP + R8D…R15D), 16bits (AX…BP + R8W…R15W), 8 bits (AL…BL + SIL…BPL + R8B…R15B). RFLAGS y RIP también son de 64bits.

Registros enteros x86-64



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 ← S	Mover fuente a destino (mismo tamaño). Variantes AT&T: tamaño byte, word, long…	movb %al, %bl movl $0xf02, %ecx
MOVS	D, S	D ← 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 ← ExtCero(S)	Mover rellenando con ceros (aumentando tamaño) movzbw, movzbl, movzwl…	movzbq %al, %rbx movzwl %cx, %edx
PUSH	S	ESP ← ESP-|S| M[ESP] ← S	Meter fuente en pila	pushl $4
POP	D	D ← M[ESP] ESP ←ESP+|S|	Sacar de pila a destino	popl %ebp
LEA	D, S	D ← &S	Cargar dirección efectiva	leal tabla(,%eax,4), %esi

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,Rindice,FactEscala)`, pudiendo omitirse los componentes no deseados.

Instrucciones aritmético-lógicas

Instrucción		Efecto	Descripción	Instrucción		Efecto	Descripción
INC	D	D ← D + 1	Incrementar	NEG	D	D ← -D	Negar (aritmético)
DEC	D	D ← D - 1	Decrementar	NOT	D	D ← ~D	Complementar (lógico)
ADD	D, S	D ← D + S	Sumar	XOR	D, S	D ← D ^ S	O-exclusivo
SUB	D, S	D ← D - S	Restar	OR	D, S	D ← D | S	O lógico
ADC	D, S	D ← D+S+C	Sumar con acarreo	AND	D, S	D ← D & S	Y Lógico
SBB	D, S	D ←D-(S+C)	Restar con débito	IMUL	D, S	D ← D * S	Multiplicar
SAL	D, k	D ← D << k	Desplazamiento a izq.	RCL	D, k	D ← D $\circlearrowright_C$ k	Rotación izq. incl. acarreo
SHL	D, k	D ← D << k	Desplazamiento a izq.	ROL	D, k	D ← D $\circlearrowright$ k	Rotación a izquierda
SAR	D, k	D ← D$>>_A$ k	Desplaz. aritm. derecha	RCR	D, k	D ← D $\circlearrowleft_C$ k	Rotación der. incl. acarreo
SHR	D, k	D ← D $>>_L$ k	Desplaz. lógico derecha	ROR	D, k	D ← D $\circlearrowleft$ k	Rotación a derecha

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 ← AL*r/m8 EDX:EAX ← EAX*r/m32 DX:AX ← AX*r/m16 RDX:RAX ← RAX*r/m64
DIV	S	$Ac_A \leftarrow Ac_{D:A} / S$ $Ac_D \leftarrow Ac_{D:A} \% S$	División sin signo	AL(AH)← AX / r/m8 EAX(EDX)←EDX:EAX / r/m32 AX(DX)←DX:AX / r/m16 RAX(RDX)←RDX:RAX / r/m64
IMUL	S Dr,S Dr,S,I	$Ac_{D:A} \leftarrow Ac_A * S$ $D \leftarrow D * S$ $D \leftarrow D*S*Inm$	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 CWDE CDQE		$Ac_{2n} \leftarrow ExtSign(Ac_n)$	Extensión de signo acum. word-to-long-to-quad	AX ← ExtSigno(AL) cbtw EAX ← ExtSigno(AX) cwtl RAX ← ExtSigno(EAX) cltq
CWD CDQ CQO		$Ac_{D:A} \leftarrow ExtSign(Ac_A)$	Extensión de signo acum. x-to-double	DX:AX ← ExtSigno(AX) cwtd EDX:EAX ← ExtSigno(EAX) cltd RDX:RAX ← ExtSigno(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) x 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 32bits será `cltd` 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 ← 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 ← S según cc	Mismos códigos
JMP	label		Salto incondicional directo Intel: "relative short/near"	Jcc	label	Saltar label según cc	Mismos códigos
jmp JMP	*Ptr PtrDst	(AT&T) (Intel)	Salto incondic. Indirecto "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. Las condiciones "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 eip eip ← label	Llamada a subrutina Intel: "near relative"	INT v	PUSH eflags CALL ISR#v	Llamada a ISR Interrupción "Interrupción software"
call CALL	*Ptr PtrDst	(AT&T) (Intel)	Intel: "near absol. indirect"			
RET		POP eip	Retorno de subrutina	IRET	RET POP eflags	Retorno de ISR
LEAVE		ESP ← EBP POP EBP	Libera marco pila Para usar con ENTER	NOP	No-op	
CLC STC CMC		C ← 0 C ← 1 C ← ~C	Ajustes del flag acarreo	CLI STI	I ← 0 I ← 1	Ajustes del flag Interrupt (des)habilitar IRQs

Tabla 7: Instrucciones x86 - control y miscelánea

El registro de flags (EFLAGS en 32bits, RFLAGS en 64bits) 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).

Registro EFLAGS, algunos bits interesantes

31	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					OF	DF	IF	TF	SF	ZF				PF		CF

Recordar también que en Linux se usa preferentemente sintaxis AT&T, distinta a la usada por Intel en sus manuales. Las diferencias más significativas se destacan en el manual de AS en §9.13:

Diferencia	Intel	AT&T

Constantes decimal, hex, octal, binario
Constantes char, string
Prefijo de valor inmediato
	`65,41h,101o,01000001b`
	`'A', "Hello"`
	`-65`
		`65,0x41,0101,0b01000001`
		`'A, "Hello"`
		`$-65`

Prefijo de registro	`eax`	`%eax`
Orden de operandos fuente y destino	`mov ebx, 0f02h`	`movl $0xf02, %ebx`
Dirección de variable	`mov ecx, offset var`	`movl $var, %ecx`

Sufijos/Directivas de tamaño de operandos
AT&T: b, w, l, q
Intel: (byte/word/dword/qword) ptr
	`mov ah,byte ptr [ebx]`
	`mov cx, dx`
	`mov esi, edi`
	`mov rbp, rsp`
		`movb (%ebx), %ah`
		`movw %dx, %cx`
		`movl %edi, %esi`
		`movq %rsp, %rbp`

Sintaxis modos direccionamiento
Despl=constante o nombre variable (su dirección)
Rb/Ri registros base/índice. S factor escala=1, 2, 4, 8
	`Despl[Rb+Ri*s]`
	`mov edx, array[eax*4]`
	`mov eax,[ebp-4]`
		`Despl(Rb,Ri,s)`
		`movl array(,%eax,4),%edx`
		`movl -4(%ebp), %eax`

Prefijo transferencias control (JMP/CALL) absolutas	`jmp eax` `call tabla[esi*4]`	`jmp *%eax` `call *tabla(,%esi,4)`

Mnemotécnicos transferencias control lejanas inm.	`jmp far sect:offset`
	`call far sect:offset`
	`ret far 4`
		`ljmp $sect,$offset`
		`lcall $sect,$offset`
		`lret $4`

Extensiones con y sin signo (MOVSX, MOVZX)
AT&T: Sufijos bw, bl, bq, wl, wq, (lq sólo con movs) y sin sufijo X
	`movsx bx, al`
	`movzx ebx, al`
	`movsx rbx, al`
	`movzx ecx, bx`
	`movsx rcx, bx`
	`mov rdx, ecx`
		`movsbw al, bx`
		`movzbl al, ebx`
		`movsbq al, rbx`
		`movzwl bx, ecx`
		`movswq bx, rcx`
		`movslq ecx, rdx`

Extensiones de signo en acumuladores	`cbw; cwde; cdqe` `cwd; cdq; cqo`	`cbtw; cwtl; cltq` `cwtd; cltd; cqto`

Tabla 8: diferencias ensambladores Intel - AT&T

Por último, destacar de la sección §7 del manual de AS las directivas más comúnmente utilizadas:

Directiva	Utilidad
.ascii *"string" [, ...]*	Ensambla cada string (pasando a código ASCII cada letra) en direcciones consecutivas
.byte *expr [, ...]*	Ensambla cada expresión en el siguiente byte (dirección)
.word *expr [, ...]* **.short** *expr [, ...]*	Emite los números indicados (enteros de 2bytes) en direcciones consecutivas
.long *expr [, ...]* **.int** *expr [, ...]*	Emite los números indicados (enteros de 4bytes) en direcciones consecutivas
.quad *bignum [, ...]*	Emite los números indicados (enteros de 8bytes) en direcciones consecutivas
.equ *symbol, expr* **.set** *symbol, expr*	Asigna a *symbol* el valor *expr* (numérico o carácter). En sucesivas apariciones, se sustituye el símbolo por el valor.
.macro *name [,args]* **.endm**	Define una *macro(instrucción)* que genera código ensamblador. En sucesivas llamadas, se sustituye el nombre por el código definido. Los argumentos se referencian como \arg.
.irp[c] *symbol, values* **.endr**	Asigna a *symbol* los valores *values* (numérico o carácter). La directiva *instruction repeat [char]* repite el código definido para los valores indicados, sustituyendo \symbol por cada valor.
.global *symbol* **.globl** *symbol*	Hace *symbol* visible desde `ld`. Si *symbol* se define en este módulo, se podrá referenciar desde otros módulos enlazados con éste. Si no, este módulo puede usar la definición de otro módulo.
.p2align *pow [,fill [,max]]*	Salta el contador de posiciones hasta próxima posición múltiplo de 2^{pow}. Las posiciones saltadas se rellenan con *fill*. Si hubiera que saltar más de *max* posiciones, se cancela el alineamiento.
.att_syntax *[[no]prefix]* **.intel_syntax** *[[no]prefix]*	Conmutar a sintaxis AT&T o Intel a partir del punto donde aparece la directiva. Se puede escoger si los registros deben llevar prefijo % o no.
.section *name [,flags[,@.]]* **.text** **.data**	Continúa ensamblando en la sección llamada *name*. Son secciones comunes: `.text` (para código) `.data` (para datos) y `.bss` (datos globales sin inicializar, `.bss` se inicializa a cero). `.text` y `.data` son tan habituales que se puede omitir `.section`.
.type *name, @type*	Marca (para `ld`) el símbolo *name* como objeto o función *(@function, @object)*
.size *name, expr*	Fija el tamaño asociado con el símbolo *name*. Se puede usar aritmética de etiquetas incluyendo el contador de posiciones . en la *expr*.

Tabla 9: directivas GNU AS

Con estas directivas sobra para redactar nuestros programas de prácticas. Si leyendo algún ensamblado gcc apareciera alguna directiva que ignoráramos, siempre podríamos consultarla en el manual de as.

Apéndice 3. Preguntas de Autocomprobación

Para evitar inquietudes sobre si es están comprendiendo bien los tutoriales, se proporcionan a continuación una serie de preguntas sobre los mismos, que pueden considerarse como ejercicios de autocomprobación.

Las siguientes preguntas se refieren a la sección "Ejercicio 4: ddd", Figura 11, sesión de depuración del programa **saludo.s**. Aunque algunas puedan responderse genéricamente, para proporcionar valores y resultados concretos es necesario comprobar las respuestas usando ddd., y tal vez `objdump -s/nm`.

	Sesión de depuración 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
2	¿Qué contiene ECX tras ejecutar **mov $saludo, %ecx**? Indicar el valor en hexadecimal, y el significado del mismo. Realizar un dibujo a escala de la memoria del programa, indicando dónde empieza el programa (_start, .text), dónde empieza saludo (.data), y dónde está el tope de pila (%esp)
3	¿Qué sucede si se elimina el símbolo de dato inmediato ($) 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 $)
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.
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
6	¿Cuántas posiciones de memoria ocupa la instrucción **mov $1, %ebx**? ¿Cómo se ha obtenido esa información? Indicar las posiciones concretas en hexadecimal.
7	¿Qué sucede si se elimina del programa la primera instrucción **int 0x80**? ¿Y si se elimina la segunda? Razonar las respuestas
8	¿Cuál es el número de la llamada al sistema *READ* (en kernel Linux 32bits)? ¿De dónde se ha obtenido esa información?

Tabla 10: preguntas de autocomprobación (saludo.s)

Las siguientes preguntas se refieren a la sección 4 "Llamadas a funciones", Figura 12, sesión de depuración del programa **suma.s**. Aunque algunas puedan responderse genéricamente, para proporcionar valores y resultados concretos es necesario comprobar las respuestas usando ddd.

	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 cuánto valen 0b10, 0x10, y (.-lista)/4
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).
3	¿Qué dirección se le ha asignado a la etiqueta **suma**? ¿Y a **bucle**? ¿Cómo se ha obtenido esa información?
4	¿Para qué usa el procesador los registros EIP y ESP?
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?
6	¿Qué registros modifica la instrucción CALL? Explicar por qué necesita CALL modificar esos registros
7	¿Qué registros modifica la instrucción RET? Explicar por qué necesita RET modificar esos registros
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
9	¿Cuántas posiciones de memoria ocupa la instrucción **mov $0, %edx**? ¿Y la instrucción **inc %edx**? ¿Cuáles son sus respectivos códigos máquina? Indicar cómo se han 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.
10	¿Qué ocurriría si se eliminara la instrucción RET? Razonar la respuesta. Comprobarlo usando ddd

Tabla 11: preguntas de autocomprobación (suma.s)

Las siguientes preguntas se refieren a la sección 5.1 "Sumar N enteros *sin* signo de 32bits en una plataforma de 32bits sin perder precisión (N≈32)", programa que denominaremos **suma64uns.s**. Aunque es posible calcular los valores solicitados a mano, conviene comprobar las respuestas usando ddd. Aún más, los ejemplos pueden servir para detectar algún error en nuestro programa.

	Cuestiones 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.
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.
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.

Tabla 12: preguntas de autocomprobación (suma64uns.s)

Las siguientes preguntas se refieren a la sección 5.2 "Sumar N enteros *con* signo de 32bits en una plataforma de 32bits", programa que denominaremos **suma64sgn.s**. Aunque es posible calcular los valores solicitados a mano, conviene comprobar las respuestas usando ddd. Aún más, los ejemplos pueden servir para detectar algún error en nuestro programa.

	Cuestiones sobre suma64signed.s
1	¿Cuál es el máximo entero positivo que puede representarse (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 usando ddd.
2	Misma pregunta respecto a negativos: menor valor negativo en hexadecimal, suma, valores decimales aprox., usar ddd.
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.
4	Repetir el ejercicio anterior de forma que sí se produzca acarreo desde los 32bits inferiores a los superiores. ¿Cuál es el valor de 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. Comprobarlo usando ddd.
5	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.
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.

Tabla 13: preguntas de autocomprobación (suma64sgn.s)

Las siguientes preguntas se refieren a la sección 5.3 “Media de N enteros *con* signo de 32bits, en plataforma de 32bits”, programa que denominaremos **media.s**. Conviene comprobar las respuestas usando ddd, no sólo porque los ejemplos pueden servir para detectar algún error en nuestro programa, sino porque también se preguntan detalles de funcionamiento de la instrucción de división que conviene conocer.

	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.
2	También se obtiene cociente 0 si se cambia lista[0]=1, o si lista[0]=2, ó si lista[0]=3… Comprobarlo con ddd. La siguiente pregunta lógica es hasta cuánto se puede incrementar lista[0] sin que cambie cociente=0. Para facilitar el cálculo mental, podemos ajustar lista[1]=-2, y así la suma de todo el array vale lista[0]-32, resultando más fácil calcular el resto. ¿Para qué rango de valores de 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 y después de llamar la subrutina que calcula la media. Tras encontrar el primer breakpoint se puede modificar 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.
3	¿Para qué rango de valores de lista[0] se obtiene media **1**? ¿Cuánto vale el resto en ese rango? Comprobarlo con ddd, y notar que tanto los dividendos como los restos son positivos (el cociente se redondea hacia cero).
4	¿Para qué rango de valores de lista[0] se obtiene media **-1**? ¿Cuánto vale el resto en ese rango? Comprobarlo con ddd, y notar que tanto los dividendos como los restos son negativos (el cociente se redondea hacia cero).

Tabla 14: preguntas de autocomprobación (media.s)