Lenguaje Ensamblador para PC

Paul A. Carter

9 de agosto de 2007

Traducido al español por Leonardo Rodríguez Mújica. Sus comentaros y sugerencias acerca de la traducción por favor a: lrodri@udistrital.edu.co

Prefacio

Propósito

El propósito de este libro es dar al lector un mejor entendimiento de cómo trabajan realmente los computadores a un nivel más bajo que los lenguajes de alto nivel como Pascal. Teniendo un conocimiento profundo de cómo trabajan los computadores, el lector puede ser más productivo desarrollando software en lenguajes de alto nivel tales como C y C++. Aprender a programar en lenguaje ensamblador es una manera excelente de lograr este objetivo. Otros libros de lenguaje ensamblador aún enseñan a programar el procesador 8086 que usó el PC original en 1981. El procesador 8086 sólo soporta el modo *real*. En este modo, cualquier programa puede acceder a cualquier dirección de memoria o dispositivo en el computador. Este modo no es apropiado para un sistema operativo multitarea seguro. Este libro, en su lugar discute cómo programar los procesadores 80386 y posteriores en modo *protegido* (el modo en que corren Windows y Linux). Este modo soporta las características que los sistemas operativos modernos esperan, como memoria virtual y protección de memoria. Hay varias razones para usar el modo protegido

1. Es más fácil de programar en modo protegido que en el modo real del 8086 que usan los otros libros.

2. Todos los sistemas operativos de PC se ejecutan en modo protegido.

3. Hay disponible software libre que se ejecuta en este modos.

La carencia de libros de texto para la programación en ensamblador de PC para modo protegido es la principal razón por la cual el autor escribió este libro.

Cómo lo dicho antes, este libro hace uso de Software Libre: es decir el ensamblador NASM y el compilador de C/C++ DJGPP. Ambos se pueden descargar de Internet. El texto también discute cómo usar el código del ensamblador NASM bajo el sistema operativo Linux y con los compiladores de C/C++ de Borland y Microsoft bajo Windows. Todos los

ejemplos de estas plataformas se pueden encontrar en mi sitio web: `http://www.drpaulcarter.com/pcasm`. Debe descargar el código de los ejemplos, si desea ensamblar y correr los muchos ejemplos de este tutorial.

Tenga en cuenta que este libro no intenta cubrir cada aspecto de la programación en ensamblador. El autor ha intentado cubrir los tópicos más importantes que *todos* los programadores deberían tener

Reconocimientos

El autor quiere agradecer a los muchos programadores alrededor del mundo que han contribuido al movimiento de Software Libre. Todos los programe y aún este libro en sí mismo fueron producidos usando software libre. El autor desearía agradecerle especialmente a John S. Fine, Simon Tatham, Julian Hall y otros por desarrollar el ensamblador NASM ya que todos los ejemplos de este libro están basados en él; a DJ Delorie por desarrollar el compilador usado de C/C++ DJGPP; la numerosa gente que ha contribuido al compilador GNU gcc en el cual está basado DJGPP; a Donald Knuth y otros por desarrollar los lenguajes de composición de textos TEX y LATEX 2_ε que fueron usados para producir este libro; a Richar Stallman (fundador de la Free Software Fundation), Linus Torvalds (creador del núcleo de Linux) y a otros que han desarrollado el software que el autor ha usado para producir este trabajo.

Gracias a las siguientes personas por correcciones:

- John S. Fine
- Marcelo Henrique Pinto de Almeida
- Sam Hopkins
- Nick D'Imperio
- Jeremiah Lawrence
- Ed Beroset
- Jerry Gembarowski
- Ziqiang Peng
- Eno Compton
- Josh I Cates
- Mik Mifflin
- Luke Wallis

- Gaku Ueda
- Brian Heward
- Chad Gorshing
- F. Gotti
- Bob Wilkinson
- Markus Koegel
- Louis Taber
- Dave Kiddell
- Eduardo Horowitz
- Sébastien Le Ray
- Nehal Mistry
- Jianyue Wang
- Jeremias Kleer
- Marc Janicki

Recursos en Internet

Página del autor	`http://www.drpaulcarter.com/`
Página de NASM en SourceForge	`http://nasm.sourceforge.net/`
DJGPP	`http://www.delorie.com/djgpp`
Ensamblador con Linux	`http://www.linuxassembly.org/`
The Art of Assembly	`http://webster.cs.ucr.edu/`
USENET	`comp.lang.asm.x86`
Documentación de Intel	`http://www.intel.com/design/Pentium4/documentation.htm`

Comentarios

El autor agradece cualquier comentario sobre este trabajo.

E-mail: `pacman128@gmail.com`
WWW: `http://www.drpaulcarter.com/pcasm`

Capítulo 1

Introducción

1.1. Sistemas de numeración

La memoria en un computador está compuesta de números. La memoria del computador no almacena estos números en decimal (base 10). Porque se simplifica mucho el hardware, los computadores almacenan toda la información en binario (base 2). Primero haremos una revisión del sistema decimal.

1.1.1. Decimal

Los números con base 10 están compuestos de 10 posibles dígitos (0-9). Cada dígito de un número tiene una potencia de 10 asociada con él, basada en su posición en el número. Por ejemplo:

$$234 = 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

1.1.2. Binario

Los números en base dos están compuestos de dos posibles dígitos (0 y 1). Cada dígito de un número tiene una potencia de 2 asociada con él basada en su posición en el número. Por ejemplo:

$$\begin{aligned} 11001_2 &= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 16 + 8 + 1 \\ &= 25 \end{aligned}$$

Esto muestra cómo los números binarios se pueden convertir a decimal. El Cuadro 1.1 muestra cómo se representan los primeros números en binario.

La Figura 1.1 muestra cómo se suman los dígitos binarios individuales (bits). A continuación un ejemplo:

Decimal	Binario		Decimal	Binario
0	0000		8	1000
1	0001		9	1001
2	0010		10	1010
3	0011		11	1011
4	0100		12	1100
5	0101		13	1101
6	0110		14	1110
7	0111		15	1111

Cuadro 1.1: Decimal de 0 a 15 en binario

No hay carry antes				Sí hay carry antes			
0	0	1	1	0	0	1	1
+0	+1	+0	+1	+0	+1	+0	+1
0	1	1	0	1	0	0	1
			c		c	c	c

Figura 1.1: Suma binaria (c es *carry*)

$$\begin{array}{r} 11011_2 \\ +10001_2 \\ \hline 101100_2 \end{array}$$

Si uno considera la siguiente división decimal:

$$1234 \div 10 = 123\ r\ 4$$

podemos ver que esta división suprime el dígito del extremo derecho del número y desplaza los otros dígitos una posición a la derecha. Dividiendo por dos hacemos una operación similar, pero para los dígitos binarios de un número. Consideremos la siguiente división binaria[1]:

$$1101_2 \div 10_2 = 110_2\ r\ 1$$

Este hecho se puede usar para convertir un número decimal a su representación equivalente en binario como muestra la Figura 1.2. Este método encuentra primero el bit del extremo derecho, llamado *bit menos significativo* (lsb). El bit del extremo izquierdo es llamado *bit más significativo* (msb). La unidad básica de memoria está compuesta de 8 bits y es llamado *byte*

[1]El subíndice 2 se usa para mostrar que el número está representado en binario no en decimal

Decimal	Binario
$25 \div 2 = 12\ r\ 1$	$11001 \div 10 = 1100\ r\ 1$
$12 \div 2 = 6\ r\ 0$	$1100 \div 10 = 110\ r\ 0$
$6 \div 2 = 3\ r\ 0$	$110 \div 10 = 11\ r\ 0$
$3 \div 2 = 1\ r\ 1$	$11 \div 10 = 1\ r\ 1$
$1 \div 2 = 0\ r\ 1$	$1 \div 10 = 0\ r\ 1$

Así $25_{10} = 11001_2$

Figura 1.2: Conversión a decimal

1.1.3. Hexadecimal

Los número hexadecimales tienen base 16. Los hexadecimales (o *hex*) se pueden usar como una representación resumida de los números binarios. Los números hexadecimales tienen 16 dígitos posibles. Esto crea un problema ya que no hay símbolos para estos dígitos adicionales después del nueve. Por convención se usan letras para estos dígitos adicionales. Los 16 dígitos hexadecimales son: 0-9 y luego A, B, C, D, E, F. El dígito A equivale a 10 en decimal, B es 11 etc. Cada dígito de un número hexadecimal tiene una potencia de 16 asociada con él. Por ejemplo:

$$\begin{aligned} 2\text{BD}_{16} &= 2 \times 16^2 + 11 \times 16^1 + 13 \times 16^0 \\ &= 512 + 176 + 13 \\ &= 701 \end{aligned}$$

Para convertir de decimal a hex use la misma idea que la empleada para la conversión binaria excepto que se divide por 16. Vea la Figura 1.3 para un ejemplo.

La razón por la cual los hexadecimales son útiles es que hay una manera fácil para convertir entre hex y binario. Los número binarios se tornan largos y molestos rápidamente. La representación hexadecimal es una manera mucho más compacta de representar los números binarios.

Para convertir un número hexadecimal a binario simplemente convierta cada dígito hexadecimal a un número binario de 4 bits. Por ejemplo, 24D_{16} es convertido en $0010\ 0100\ 1101_2$. Observe que ¡los ceros delanteros son importantes! Si los ceros del dígito de la mitad de 24D_{16} no se usan el resultado es erróneo. Convertir de binario a hex es igual de fácil; uno hace el proceso

$$\begin{aligned} 589 \div 16 &= 36\ r\ 13 \\ 36 \div 16 &= 2\ r\ 4 \\ 2 \div 16 &= 0\ r\ 2 \end{aligned}$$

Así $589 = 24\mathrm{D}_{16}$

Figura 1.3:

inverso, convierte cada segmento de 4 bits a hexadecimal comenzando desde el extremo derecho, no desde el izquierdo, del número binario. Esto asegura que el segmento de 4 bits es correcto[2]. Ejemplo:

110	0000	0101	1010	0111	1110_2
6	0	5	A	7	E_{16}

Un número de 4 bits es llamado *nibble* . Así cada dígito hexadecimal corresponde a un nibble. Dos nibbles conforman un byte y por lo tanto un byte puede ser representado por dos dígitos hexadecimales. Los valores de un byte van de 0 a 11111111 en binario, 0 a FF en hex y 0 a 255 en decimal.

1.2. Organización del computador

1.2.1. La Memoria

La memoria es medida en unidades de kilobytes ($2^{10} = 1,024$ bytes), mega bytes ($2^{20} = 1,048,576$ bytes) y Gigabytes ($2^{30} = 1,073,741,824$ bytes).

La unidad básica de memoria es el byte. Un computador con 32 Mega bytes de memoria puede almacenar aproximadamente 32 millones de bytes de información. Cada byte está etiquetado por un número único conocido como su dirección. Tal como lo muestra la Figura 1.4.

Dirección	0	1	2	3	4	5	6	7
Memoria	2A	45	B8	20	8F	CD	12	2E

Figura 1.4: Direcciones de Memoria

A menudo la memoria se usa en trozos más grandes que un byte. En la arquitectura del PC, los nombres que se le han dado a estas secciones de

[2]Si no es claro porque el punto de inicio hace la diferencia, intente convertir el ejemplo comenzando desde la izquierda

inglés	unidad	español
word	2 bytes	palabra
double word	4 bytes	palabra doble
quad word	8 bytes	palabra cuádruple
paragraph	16 bytes	párrafo

Cuadro 1.2: Unidades de memoria

memoria más grandes se muestran en el Cuadro 1.2 [3].

Todos los datos en la memoria son numéricos. Los caracteres son almacenados usando un *código de caracteres* que traduce un número en un carácter. Uno de los códigos de caracteres más conocido es el *ASCII* (*American Standar Code for Information Interchange*). Un nuevo código, más completo, que está reemplazando al ASCII es el Unicode. Una diferencia clave entre los dos códigos es que el ASCII usa un byte para codificar un carácter, pero Unicode usa dos bytes (o una *palabra*) por carácter. Por ejemplo ASCII decodifica el byte 41_{16} (65_{10}) como la *A* mayúscula. Unicode la codifica con la palabra 0041_{16}. Ya que ASCII usa un byte está limitado a sólo 256 caracteres diferentes.[4] Unicode amplía los valores ASCII a palabras y permite que se representen muchos más caracteres. Esto es importante para representar los caracteres de todas las lenguas del mundo.

1.2.2. La CPU

La Unidad Central de Procesamiento (CPU) es el dispositivo físico que ejecuta las instrucciones. Las instrucciones que ejecuta la CPU son por lo general muy simples. Las instrucciones pueden requerir datos que estén en un lugar especial de almacenamiento de la CPU en sí misma llamados *registros*. La CPU puede acceder a los datos en los registros mucho más rápido que en la memoria. Sin embargo el número de registros en la CPU es limitado, así el programador debe tener cuidado de dejar en los registros sólo los datos que esté usando.

Las instrucciones que un tipo de CPU ejecuta las hace en *lenguaje de máquina*. Los programas en lenguaje de máquina tienen una estructura mucho más básica que los lenguajes de alto nivel. Las instrucciones en lenguaje de máquina son codificadas como números, no en formatos de texto amigables. Una CPU debe estar en capacidad de decodificar una instrucción muy rápidamente para ejecutarse eficientemente. EL lenguaje de máquina es diseñado con este objetivo en mente, no para ser fácilmente descifrados

[3] N del T: En la traducción se usarán los nombres de las unidades de memoria en español, aunque en la literatura técnica seguramente los encontrarán en inglés

[4] De hecho ASCII sólo usa los 7 bits más bajos y sólo tiene 128 valores diferentes

por humanos. Los programas escritos en otros lenguajes deben ser convertidos en lenguaje de máquina nativo de la CPU para que se ejecute en el computador. Un *compilador* es un programa que traduce programas escritos en un lenguaje de programación al lenguaje de máquina de una arquitectura en particular de un computador. En general cada tipo de CPU tiene su propio y único lenguaje de máquina. Esa es una de las razones por las cuales programas escritos para un Mac no corren en un PC tipo IBM

GHz significa Gigahertz o mil millones de ciclos por segundo. Una CPU de 1.5 GHz *tiene mil quinientos millones de pulsos de reloj por segundo.*

Los computadores usan un *reloj* para sincronizar la ejecución de las instrucciones. El *reloj* pulsa a una frecuencia fija conocida como velocidad del reloj. Cuando Ud. compra un computador de 1.5 GHz, la frecuencia de su reloj es 1.5 GHz. Actualmente, los pulsos del reloj son usados por muchos componentes de un computador. Con frecuencia, los otros componentes usan unas velocidades de reloj diferentes que la CPU. El reloj no marca los minutos y los segundos, simplemente toca a una razón constante. La electrónica de la CPU usa los toques para realizar sus operaciones correctamente, como los toques de un metrónomo para la interpretación de música al ritmo correcto. El número de toques (o como a ellos se les llama comúnmente *ciclos*) que una instrucción requiere depende del modelo de la CPU, de la instrucción anterior y de otros factores.

1.2.3. La familia de CPU 80x86

Las PC de tipo IBM tienen una CPU de la familia Intel (o un clon de ellas) Las CPU de esta familia todas tienen algunas características comunes incluyendo el lenguaje de máquina básico. Sin embargo los miembros más recientes amplían mucho las características.

8888,8086: Estas CPU desde el punto de vista de la programación son iguales. Ellas fueron las CPU usadas en las primeras PC. Ellas suministran varios registros de 16 bits: AX, BX, CX, DX, SI, DI, BP, SP, CS, DS, SS, ES, IP, FLAGS. Ellas solo soportan hasta 1 Mega byte de memoria y sólo operan en *modo real*. En este modo un programa puede acceder a cualquier dirección de memoria, ¡aún a la memoria de otros programas! Esto hace la depuración y seguridad muy difícil. También la memoria del programa tiene que ser dividida en *segmentos*. Cada segmento no puede ser más largo que 64 KB

80286: Esta CPU se usa en los PC tipo AT. Agrega unas instrucciones nuevas al lenguaje de máquina base del 8080/86. Sin embargo la característica principal nueva es el modo *protegido de 16 bits*. En este modo puede acceder hasta 16 Mega bytes de memoria y proteger a los programas del acceso de otros. Sin embargo los programas todavía están divididos en segmentos que no pueden ser más grandes de 64K.

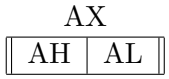


Figura 1.5: El registro AX

80386: Esta CPU es una gran ampliación del 80286. Primero extiende muchos de los registros para almacenar 32 bits (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP) y añade dos nuevos registros de 16 bits FS y GS. También añade un nuevo modo *protegido de 32 bits*. En este modo pueden acceder hasta 4 Gigabyes. Los programas otra vez están divididos en segmentos, pero ahora cada segmento también puede tener hasta un tamaño de 4 Gigabytes.

80486/Pentium/Pentium Pro: Estos miembros de la familia 80x86 añaden muy pocas características nuevas. Ellos principalmente aceleran la ejecución de las instrucciones.

Pentium MMX: Este procesador añade instrucciones MMX (eXtensiones MultiMedia) al Pentium. Estas instrucciones pueden acelerar instrucciones comunes gráficas.

Pentium II: Este es el procesador Pentium Pro con las instrucciones MMX añadidas (El pentium III es esencialmente sólo un Pentium II rápido.

1.2.4. Registros de 16 bits del 8086

La CPU original 8086 suministra 4 registros de 16 bits de propósito general AX, BX, CX y DX. Cada uno de esos registros puede ser descompuesto en dos registros de 8 bits. Por ejemplo el registro AX se puede descomponer en los registros AL y AH que muestra la Figura 1.5. El registro AH contiene los 8 bits superiores de AX y AL contiene los 8 bits inferiores de AX. A menudo AH y AL son usados como registros independientes de 8 bits; sin embargo es importante tener en cuenta que ellos no son independientes de AX. Cambiando el valor de AX cambiará AH y AL y *viceversa*. Los registros de propósito general son usados en muchos movimientos de datos e instrucciones aritméticas.

Hay dos registros de índice de 16 bits SI y DI. Ellos son a menudo usados como apuntadores, pero pueden ser usados para muchos de los propósitos de los registros generales. Sin embargo, ellos no se pueden descomponer en registros de 8 bits.

Los registros de 16 bits BP y SP son usados para señalar a los datos en la pila y son llamados Apuntador Base (*Base Pointer*) y apuntador a la pila (*Stack Pointer*), respectivamente. Ellos se discutirán luego.

Los registros de 16 bits CS, DS, SS y ES son *registros de segmento*. Ellos especifican qué memoria es usada por diferentes partes de un programa. CS significa segmento de código (*Code Segment*), DS segmento de datos (*Data Segment*), SS Segmento de la pila (*Stack Segment*) y ES segmento adicional (*Extra Segment*). ES es usado como un registro temporal. Los detalles de estos registros están en las Secciones 1.2.6 y 1.2.7.

El registro IP, Apuntador a la instrucción (*Instruction Pointer*) es usado con el registro CS para obtener la dirección de la siguiente instrucción a ser ejecutada por la CPU. Normalmente cuando se ejecuta una instrucción IP avanza hasta señalar a la siguiente instrucción en memoria.

El registro FLAGS almacena información importante sobre los resultados de una instrucción anterior. Estos resultados son almacenados como bits individuales en el registro. Por ejemplo, el bit Z es 1 si el resultado de la instrucción anterior fue cero o 0 si el resultado no fue cero. No todas las instrucciones modifican bits en FLAGS, consulte el cuadro en el apéndice para ver cómo instrucciones específicas afectan el registro FLAGS

1.2.5. Registros de 32 bits del 80386

El 80386 y los procesadores posteriores tienen registros extendidos. Por ejemplo el registro de 16 bits AX se extendió para ser de 32 bits. Para la compatibilidad con sus predecesores, AX se refiere al registro de 16 bits y EAX se usa para referirse al registro extendido de 32 bits. AX son los 16 bits inferiores de EAX tal como AL son los 8 bits inferiores de AX (y EAX). No hay forma de acceder directamente a los 16 bits superiores de EAX. Los otros registros extetendidos son EBX, ECX, EDX, ESI and EDI.

Muchos de los otros registros se extienden también. BP se convierte en EBP, SP se convierte en ESP, FLAGS en EFLAGS e IP en EIP. Sin embargo, son diferentes los registros de índice y los de propósito general, en el modo protegido de 32 bits (discutidos luego) sólo se usan las versiones extendidas de estos registros.

Los registros de segmento continúan siendo de 16 bits en el 80386. Hay también dos nuevos registros de segmento: FS y GS. Sus nombres no significan nada. Ellos son registros adicionales para segmentos temporales (como ES).

Una de las definiciones del térmno *word* se refiere a el tamaño del registro de datos de la CPU. Para la familia 80x86, el término es ahora un poco confuso. En el Cuadro 1.2, uno ve que *word* está definida para ser 2 bytes (o 16 bits). Este fue el significado que se le dio, cuando se lanzó la primera vez el 8086. Cuando se desarrolló el 80386, se decidió dejar la definición de *word* sin cambio, auque el tamaño del registro cambió.

1.2.6. Modo Real

¿De dónde viene el infame límite de 640K de DOS? La BIOS requerida alguno-de 1M para el código y para los dispositivos de hardware como la pantalla de video

En el modo real la memoria está limitada a sólo 1 mega byte (2^{20} bytes) Las direcciones válidas están desde 0000 hasta FFFFF. (en hexadecimal) Estas direcciones requieren un número de 20 bits Obviamente un número de 20 bits no cabrá en ningún registro de 16 bits. Intel solucionó este problema usando 2 valores de 16 bits para determinar una dirección. El primer valor de 16 bits es llamado *seleccionador selector*. Los valores del seleccionador deben estar almacenados en registros de segmento El segundo valor de 16 bits es llamado *desplazamiento* (*offset*) La dirección física referenciada por un par *seleccinador:desplazamiento* es calculada por la fórmula:

$$16 * \text{seleccionador} + \text{desplazamiento}$$

multiplicar por 16 en hexadecimal es muy fácil, es sólo añadir un 0 a la derecha del número. Por ejemplo la dirección física referenciada por 047C:0048 está dada por:

$$\begin{array}{r} 047C0 \\ +0048 \\ \hline 04808 \end{array}$$

En efecto, el valor seleccionador es un número párrafo (vea el Cuadro 1.2).
direcciones reales segmentadas tienen desventajas:

- Un único valor del seleccionador sólo puede hacer referencia a 64K de memoria (el límite superior del desplazamiento de 16 bits). ¿Qué pasa si un programa tiene más de 64K de código? Un solo valor en CS no se puede usar para toda la ejecución del programa. El programa se debe dividir en secciones (llamadas *segmentos*) menores de 64K en tamaño. Cuando la ejecución se mueve de un segmento a otro, los valores de CS se deben cambiar. Problemas similares ocurren con grandes cantidades de datos y el registro DS Esto puede ser muy incómodo
- Cada byte de memoria no tiene una sola dirección segmentada. La dirección física 04804 puede ser referenciada por 047C:0048, 047D:0038, 0047E:0028 o 047B:0058. Esto puede complicar la comparación de direcciones segmentadas.

1.2.7. Modo protegido de 16 bits

En el modo protegido del 80286 los valores del seleccionador son interpretados completamente diferente que en el modo real. En este modo, un valor del seleccionador es un número de párrafo de memoria física. En el modo protegido un valor seleccionador es un *índice* en una *tabla de descriptores*.

En ambos modos, los programas son divididos en segmentos. En modo real estos segmentos están en posiciones fijas en la memoria física y el seleccionador denota el número de párrafo de comienzo del segmento. En modo protegido los segmentos no están en posiciones fijas en la memoria física. De hecho no tiene que estar todo el segmento en memoria.

El modo protegido usa una técnica llamada *memoria virtual*. La idea básica de un sistema de memoria virtual, es dejar sólo en la memoria los datos y el código que los programas están usando en un momento dado. Los otros datos y el código son almacenados temporalmente en el disco hasta que ellos se necesiten de nuevo. Cuando retorna un segmento a la memoria del disco, es muy probable que se coloque en un área diferente de memoria en el que estuvo antes de ser enviada al disco. Todo esto es hecho transparementemente por el sistema operativo. El programa no se tiene que escribir de otra manera para que la memoria virtual trabaje.

En el modo protegido a cada segmento se le asigna una entrada en una tabla de descriptores. Esta entrada tiene toda la información que el sistema necesita conocer sobre el segmento. Esta información incluye: si está actualemente en memoria, si es así dónde está, permiso de acceso (ejem: sólo lectura). El índice de la entrada del segmento es el valor del seleccionador que está almacendo en los registros de segmento.

Un conocido columnista de PC llamó al 286 "cerebro muerto."

Una gran desventaja del modo protegido es que los desplazamientos están aún en cantidades de 16 bits Como una consecuencia de esto, los tamaños de los segmentos están todavía limitados a un máximo de 64K. Esto hace problemático el uso de arreglos grades.

1.2.8. Modo protegido de 32 bits

El 80386 introdujo el modo protegido de 32 bits. Hay dos grandes diferencias entre los modos protegidos de un 386 de 32 bits y un 286 de 16 bits

1. Los desplazamientos se amplían a 32 bits. Esto permite un rango de desplazamiento hasta 4 millardos. Así los segmentos pueden tener tamaños hasta de 4 Gigabytes.

2. Los segmentos pueden ser divididos en unidades más pequeñas de 4K llamadas *páginas*. El sistema de memoria virtual trabaja ahora con páginas en lugar de segmentos. Esto significa que partes de un segmento pueden estar en memoria. En el modo de 16 bits del 286 o todo el segmento está en memoria o no está. Esto no es práctico con los grandes segmentos que permite el modo de 32 bits. Windows 9x, Windows NT/200/XP, OS/2 y Linux todos se ejecutan en modo protegido de 32 bits

En Windows 3.x el *modo standar* se refiere al modo protegido de 16 bits del 286 y el *modo ampliado* se refiere al modo de 32 bits.

1.2.9. Interrupciones

Algunas veces el flujo ordinario de un programa debe ser interrumpido para procesar eventos que requieren una respuesta rápida. El hardware de un computador provee un mecanismo llamado *interrupción* para manipular estos eventos. Por ejemplo cuando se mueve el ratón la interrupción de hardware del ratón es el programa actual para manejar el movimiento del ratón (para mover el cursor del mouse, etc) Las interrupciones hacen que el control se pase a un *manipulador de interrupciones*. Los manipuladores de interrupciones son rutinas que procesan la interrupción. A cada tipo de interrupción se le asigna un número entero. En el comienzo de la memoria física una tabla de *vectores de interrupción* que contiene la dirección del segmento de los manipuladores de la interrupción. El número de la interrupción es escencialmente un índice en esta tabla.

Las interrupciones externas son levantadas desde el exterior de la CPU (el ratón es un ejemplo de este tipo de interrupción). Muchos dispositivos de E/S levantan interrupciones (teclado, temporizador, disco duro CD ROM y tarjetas de sonido) Las interrupciones internas son levantadas desde la CPU, por una instrucción de error o por una instrucción de interrupción. Las instrucciones de error también se llaman *trampas*. Las interrupciones generadas desde la instrucción de interrupción son llamadas *interrupciones* de sofware. DOS usa estas interrupciones para implementar su API (Interfaz de programas de Aplicación). Sistemas operativos más modernos (como Windows y Linux) usan una interfaz basada en C [5]

Muchos manipuladores de interrupción devuelven el control al programa interrumpido cuando ella culmina. Ella restaura todos los registros con los mismos valores que tenían antes que ocurriera la interrupción. Así el programa interrumpido se ejecuta como si nada hubiese pasado (excepto que se perdieron algunos ciclos de CPU) Las trampas generalmente no retornan. A menudo ellas acaban el programa.

1.3. Lenguaje ensamblador

1.3.1. Lenguaje de máquina

Cada tipo de CPU entiende su propio lenguaje de máquina. Las instrucciones en lenguaje de máquina son números almacenados como bytes en memoria. Cada instrucción tiene su propio y único código llamado *código*

[5] Sin embargo, ellas pueden usar una interfaz de bajo nivel (a nivel del kernel)

de operación u opcode. Las instrucciones del procesador 80x86 varían en tamaño. El opcode está siempre al inicio de la instrucción. Muchas instrucciones incluyen también datos (vgr. constantes o direcciones) usados por las instrucciones.

El lenguaje de máquina es muy difícil de programar directamente. Descifrar el significado de las instrucciones codificadas numéricamente es tedioso para los humanos. Por ejemplo, la instrucción que suma los registros EAX y EBX y almacena el resultado en EAX está codificada por los siguientes códigos hexadecimales

```
03 C3
```

Esto no es obvio. Afortunadamente, un programa llamado *ensamblador* puede hacer este aburrido trabajo para el programador.

1.3.2. Lenguaje ensamblador

Un programa Escrito en lenguaje ensamblador es almacenado como texto (tal como programas de alto nivel). Cada instrucción representa exactamente una instrucción de la máquina. Por ejemplo, la instrucción de suma descrita arriba podría ser representada en lenguaje ensamblador como:

```
add eax, ebx
```

Acá el significado de la instrucción es *mucho* más claro que el código de la máquina. La palabra `add` es el *nemónico* para la instrucción de suma . La forma general de una instrucción de ensamblador es:

```
mnemonico operando(s)
```

Un *ensamblador* es un programa que lee un archivo de texto con instrucciones de ensamblador y convierte el ensamblador en código de máquina. Los *compiladores* son programas que hacen conversiones similares para lenguajes de programación de alto nivel. Un ensamblador es mucho más simple que un compilador. Cada instrucción de lenguaje ensamblador representa una sola instrucción de la máquina. Las instrucciones de un lenguaje de alto nivel son *mucho* más complejas y pueden requerir muchas instrucciones de máquina.

Les tomó varios años a los científicos de la computación imaginarse cómo escribir un compilador

Otra diferencia importante entre los lenguajes ensamblador y de alto nivel es que debido a que cada tipo de CPU tiene su propio lenguaje de máquina, también tiene su propio lenguaje ensamblador. Trasladar programas entre arquitecturas de computador diferentes es *mucho* más difícil que en un lenguaje de alto nivel.

En los ejemplos de este libro se usa Netwide Assembler o NASM . Está disponible libremente en Internet (vea el prefacio para la URL). Los ensambladores más comunes son el ensamblador de Microsoft (MASM) y el de Borland (TASM) . Hay algunas diferencias en la sintaxis del ensamblador de NASM, MASM y TASM .

1.3.3. Operandos de las instrucciones

Los códigos de las instrucciones de máquina tienen una variedad de tipos y operandos; sin embargo, en general cada instrucción en sí misma tiene un número fijo de operandos (0 a 3). Los operandos pueden tener los siguientes tipos:

registro: Estos operandos se refieren directamente al contenido de los registros de la CPU.

memoria: Estos se refieren a los datos en la memoria. La dirección de los datos puede ser una constante escrita en la instrucción o puede ser calculada usando los valores de los registros. Las direcciones son siempre desplazamientos relativos al comienzo de un segmento.

immediato: Estos son valores fijos que están listados en la instrucción en sí misma. Ellos son almacenados en la instrucción en si misma (en el segmento de código), no en el segmento de datos.

implícito: Estos operandos no se muestran explícitamente. Por ejemplo, la instrucción de incremento añade uno a un registro o a memoria. El uno está implícito.

1.3.4. instrucciones básicas

La instrucción esencial es `MOV` . Ella translada datos de un lugar a otro (como el operador de asignación en un lenguaje de alto nivel). Toma dos operandos:

```
mov dest, src
```

El dato especificado por *src* es copiado a *dest*. Una restricción es que los dos operandos no pueden ser operandos de memoria. Esto señala otra peculiaridad del ensamblador. Hay a menudo algunas reglas arbitrarias sobre cómo se usan las instrucciones. Los operandos deben tener el mismo tamaño. El valor de AX no puede ser almacenado en BL.

Acá hay un ejemplo(los ; inician un comentario)

```
mov    eax, 3   ; almacena 3 en el registro EAX (3 es el operando inmediato)
mov    bx, ax   ; almacena el valor de AX en el registro BX
```

La instrucción `ADD` se usa para sumar enteros.

```
add    eax, 4   ; eax = eax + 4
add    al, ah   ; al = al + ah
```

La instrucción `SUB` resta enteros.

```
sub    bx, 10   ; bx = bx - 10
sub    ebx, edi ; ebx = ebx - edi
```

Las instrucciones INC y DEC incrementan o decrementan valores en uno. Ya que el uno es un operando implícito, el código de de máquina para INC y el DEC es más pequeño que los de las instrucciones ADD y SUB.

```
inc    ecx      ; ecx++
dec    dl       ; dl--
```

1.3.5. Directivas

Una directiva es un artificio del ensamblador no de la CPU. Ellas se usan generalmente para decirle al ensamblador que haga alguna cosa o informarle al ensamblador de algo. Ellas no se traducen en código de máquina. Los usos comunes de las directivas son:

- Definir constantes
- Definir memoria para almacenar datos en ella
- Definir la memoria para almacenar datos en ella
- Agrupar la memoria en segmentos
- Incluír código fuente condicionalmente
- Incluír otros archivos

El código de NASM pasa a través de un preprocesador tal como C. Tiene muchas de las órdenes del preprocesador tal como C. Sin embargo las directivas del preprocesador de NASM comienzan con un como en C.

directiva equ

La directiva equ se puede usar para definir un símbolo. Los símbolos son constantes con nombre que se pueden emplear en el programa ensamblador. El formato es:

símbolo equ *valor*

Los valores de los símbolos no se pueden redefinir posteriormente.

La directiva %define

Esta directiva es parecida a la #define de C. Se usa normalmente para definir macros tal como en C.

```
%define SIZE 100
      mov    eax, SIZE
```

Unidad	Letra
byte	B
palabra	W
palabra doble	D
palabra cuádruple	Q
diez bytes	T

Cuadro 1.3: Letras para las directivas `RESX` y `DX`

El código de arriba define un macro llamado size y muestra su uso en una instrucción MOV. Los macros son más flexibles que los símbolos de dos maneras. Los macros se pueden redefinir y pueden ser más que simples constantes númericas.

Directivas de datos

Las directivas de datos son usadas en segmentos de datos para definir espacios de memoria. Hay dos formas en que la memoria puede ser reservada. La primera es solo definir el espacio para los datos; la segunda manera define el espacio y el valor inicial. El primer método usa una de las directivas `RESX`. La `X` se reemplaza con una letra que determina el tamaño del objeto (u objetos) que será almacenados. El Cuadro 1.3 muestra los valores posibles.

El segundo método (que define un valor inicial también) usa una de las directivas DX. Las `X` son las mismas que las de la directiva `RESX`.

Es muy común marcar lugares de memoria con etiquetas. Las etiquetas le permiten a uno referirse fácilmente a lugares de la memoria en el código. Abajo hay varios ejemplos.

```
L1    db     0         ; byte etiquetado como L1 con valor inicial 0
L2    dw     1000      ; palabra etiquetada como L2 con valor inicial de 1000
L3    db     110101b   ; byte con valor inicial binario de 110101 (53 en decimal)
L4    db     12h       ; byte con valor inicial hex de 12 (18 en decimal)
L5    db     17o       ; byte con valor inicial octal de 17 (15 en decimal)
L6    dd     1A92h     ; plabra doble con valor inicial hex de 1A92
L7    resb   1         ; un byte sin valor inicial
L8    db     "A"       ; byte con valor inicial del código ASCII para A (65)
```

Las comillas dobles o simples se interpretan igual. Las definiciones consecutivas de datos se almacenarán secuencialmente en memoria. Así que, la palabra L2 se almacenará inmediatamente después que la L1. Se pueden definir también secuencias de memoria.

```
L9    db     0, 1, 2, 3              ; define 4 bytes
```

```
L10   db     "w", "o", "r", 'd', 0   ; define una cadena tipo C = "word"
L11   db     'word', 0               ; igual que  L10
```

La directiva DD se puede usar para definir o enteros o constantes de punto flotante de presición simple.[6] Sin embargo DQ solo se puede usar para definir constantes de punto flotante de doble precisión.

Para secuencias largas la directiva TIMES de NASM es a menudo útil. Esta directiva repite su operando un número especificado de veces por ejemplo:

```
L12   times 100 db 0                 ; equivalente a 100 veces db 0
L13   resw   100                     ; reserva lugar para 100 palabras
```

Recuerde que las etiqueta pueden ser usadas para referirse a datos en el código. Si se usa una etiqueta ésta es interpretada como la dirección (o desplazamiento) del dato. Si la etiqueta es colocada dentro de paréntesis cuadrados ([]), se interpreta como el dato en la dirección. En otras palabras, uno podría pensar de una etiqueta como un apuntador al dato y los paréntesis cuadrados como la des referencia al apuntador tal como el asterisco lo hace en C (MASM y TASM siguen una convención diferente). En el modo de 32 bits las direcciones son de 32 bits. A continuación, algunos ejemplos.

```
1        mov    al, [L1]       ; copia el byte que está en  L1 en AL
2        mov    eax, L1        ; EAX = dirección del byte en L1
3        mov    [L1], ah       ; copia AH en el byte en L1
4        mov    eax, [L6]      ; copia la palabra doble en L6 en EAX
5        add    eax, [L6]      ; EAX = EAX + la palabra doble en L6
6        add    [L6], eax      ; la palabra doble en L6 += EAX
7        mov    al, [L6]       ; copia el primer byte de la palabra doble en L6 en AL
```

La línea 7 de los ejemplos muestra una propiedad importante de NASM. El ensamblador no recuerda el tipo de datos al cual se refiere la etiqueta. De tal forma que el programador debe estar seguro que usa la etiqueta correctamente. Luego será común almacenar direcciones de datos en registros y usar los registros como una variable apuntador en C. Una vez más no se verifica que el apuntador se use correctamente. De este modo el ensamblador es mucho más propenso a errores aún que C.

Considere la siguiente instrucción:

```
mov    [L6], 1              ; almacena 1 en  L6
```

Esta instrucción produce un error de tamaño no especificado. ¿Por qué? Porque el ensamblador no sabe si almacenar el 1 como byte, palabra o palabra doble. Para definir esto, se añade un especificador de tamaño .

[6] Punto flotante de presición simple es equivalente a la variable float en C.

```
mov    dword [L6], 1        ; almacena 1 at L6
```

Esto le dice al ensamblador que almacene un 1 en la palabra doble que comienza en L6. Otros especificadores son: BYTE, WORD, QWORD Y TWORD. [7]

1.3.6. Entrada y Salida

La entrada y salida son acciones muy dependientes del sistema. Involucra comunicarse con el hardware del sistema. Los lenguquajes del alto nivel, como C, proveen bibliotecas normalizadas de rutinas que suministran una interfaz de programación simple y uniforme para la E/S. Los lenguajes ensamblador no disponen de bibliotecas normalizadas. Ellos deben acceder directamente al hardware (que es una operación privilegiada en el modo protegido) o usar las rutinas de bajo nivel que provea el sistema operativo.

Es muy común que se interfacen rutinas de ensamblador con C. Una de las ventajas de esto es que el código en ensamblador puede usar las rutinas E/S de las bibliotecas estandar de C. Sin embargo uno debe conocer las reglas que usa C para pasar información entre rutinas. Estas reglas son muy complicadas para cubrir acá (ellas se verán luego). Para simplificar la E/S el autor ha desarrollado sus propias rutinas que ocultan las complejas reglas de C y provee una interfas mucho más simple. El Cuadro 1.4 describe las rutinas suministradas. Todas las rutinas preservan el valor de todos los registros, excepto las rutinas de lectura. Estas rutinas modifican el valor del registro EAX. Para usar estas rutinas uno debe incluir un archivo con la información que el ensamblador necesita para poder usarlas. Para incluir un archivo en NASM use la directiva del preprocesador `%include`. La siguiente línea incluye el archivo necesario para las rutinas de E/S hechas por el autor[8]:

```
%include "asm_io.inc"
```

Para usar una de las rutinas print, uno carga EAX con el valor correcto y usa la instrucción `CALL` para invocarla. La instrucción `CALL` es equivalente a un llamado de función en un lenguaje de alto nivel. Hace un salto en la ejecución hacia otra sección de código pero después retorna al origen luego que la rutina a culminado. El programa muestra varios ejemplos de llamadas de estas rutinas de E/S.

[7] TWORD define un área de memoria de 10 bytes. El coprocesador de punto flotante usa este tipo de dato.

[8] El archivo `asm_io.inc` (y el archivo objeto asm_io.o archivo que `asi_io.inc` necesita) están en los ejemplos que se encuentan en la página web del autor: `http://www.drpaulcarter.com/pcasm`

print_int	imprime en la pantalla el valor del entero almacendo en EAX
print_char	imprime en la pantalla el caracter cuyo código ASCII esté almacendo en AL
print_string	imprime en la pantalla el contenido de la cadena en la *dirección* almacenada en EAX. La cadena debe ser tipo C, (terminada en NULL).
print_nl	imprime en pantalla el caracter de nueva línea.
read_int	lee un entero del teclado y lo almacena en el registro.
read_char	lee un solo caracter del teclado y almacena el código ASCII en el registro EAX.

Cuadro 1.4: Rutinas de E/S en ensamblador

1.3.7. Depuración

La biblioteca del autor también contiene algunas rutinas útiles para depurar los programas. Estas rutinas de depuración muestran información sobre el estado del computador sin modificar su estado. Estas rutinas son en realidad *macros* que muestran el estado de la CPU y luego hacen un llamado a una subrutina. Los macros están definidos en el archivo `asm_io.inc` discutido antes. Los matros se usan como instrucciones normales. Los operandos de los macros se separan con comas.

Hay cuatro rutinas de depuración llamadas `dump_regs`, `dump_mem`, `dump_stack` and `dump_math`; Ellas muestran los valores de los registros, memoria, pila y el coprocesador matemático respctivamente

dump_regs Este macro imprime los valores de los registros (en hexadecimal) del computador `stdout` (la pantalla). También imprime el estado de los bits del registto FLAGS[9]. Por ejemplo si la bandera cero es 1 se muestra `ZF`. Si es cero no se muestra nada. Torma un solo entero como parámetro que luego se imprime. Este entero se puede usar para distinguir la salida de diferentes órdenes `dump_regs`.

dump_mem Este macro imprime los valores de una región de memoria (en hexadecimal) y también como caracteres ASCII. Toma tres argumentos delimitados por comas. El primero es un entero que es usado para identificar la salida (tal cual como el argumento de dump_regs). El segundo argumento es la dirección a mostrar (ésta puede ser una etiqueta). El último argumento es un número de párrafos de l6 bytes para mostrar luego de la direccción. La memoria mostrada comenzará en el primer límite de párrafo antes de la dirección solicitada.

[9]El Capítulo 2 discute este registro

dump_stack Este macro imprime los valores de la pila de la CPU (la pila se verá en el Capítulo 4). La pila está organizada como palabras dobles y está rutina las mostrará de esta forma. Toma tres parámetros separados por comas. El primero es un identificador entero (como `dump_regs`). El segundo es el número de palabras dobles para mostrar *antes* de la dirección que tenga almacenada el registro EBP, y el tercer argumento es el número de palabras dobles a imprimir *luego* de la dirección de EBP.

dump_math Este macro imprime los valores de los registros del coprocesador matemático. Toma un solo parámetro entero como argumento que se usa para identificar la salida tal como el argumento de `dump_regs` lo hace.

1.4. Creando un programa

Hoy día no es común crear todo un programa independiente escrito totalmente en lenguaje ensamblador. El ensamblador es usado para desarrollar ciertas rutinas crítica. ¿Por qué? Es *mucho* más fácil programar en un lenguaje de alto nivel que en ensamblador. También al usar ensamblador es muy difícil transportar el programa a otras plataformas. De hecho, es raro usar el ensamblador en cualquier circunstancia.

Así, ¿Por qué alguien quisiera aprender ensamblador?

1. Algunas veces el código escrito en ensamblador puede ser más rápido y pequeño que el código generado por un compilador.

2. El ensamblador permite acceder directamente a características del hardware del sistema que puede ser difícil o imposible de usar desde un lenguaje de alto nivel.

3. Aprender a programar en ensamblador le ayuda a uno a ganar un entendimiento profundo de cómo trabaja el computador.

4. Aprender a programar en ensamblador ayuda a entender mejor cómo trabajan los compiladores y los lenguajes de alto nivel como C.

Los últimos dos puntos demuestran que aprender ensamblador puede ser útil aún si uno nunca programa en él posteriormente. De hecho, el autor raramente programa en ensamblador pero usa las ideas aprendidas de él todos los días.

```
int main()
{
  int ret_status ;
   ret_status = asm_main();
  return ret_status ;
}
```

Figura 1.6: código de `driver.c`

1.4.1. Primer programa

Los primeros programas en este texto comenzarán todos con un programa sencillo de C mostrado en la Figura 1.6. Simplemente llama otra función llamada `asm_main`. Esta es la rutina escrita en ensamblador. Hay varias ventajas de usar este programa en C. Primero dejamos que C fije todos los parámetros para que el programa se ejecute correctamente en el modo protegido. Todos los segmentos y sus correspondientes registros de segmento serán iniciados por C. El código en ensamblador no necesita preocuparse de nada de esto. Segundo, las bibliotecas de C estarán disponibles para ser usadas en el código de ensamblador. Las rutinas de E/S del autor aprovechan esto. Ellas usan las funciones de E/S de C (`printf`, etc.). Ahora se muestra un programa elemental en ensamblador.

first.asm

```
1  ; Archivo: first.asm
2  ; Primer programa en ensamblador. Este programa pide dos
3  ; enteros como entrada e imprime su suma
4
5  ; Para crear el ejecutable usando djgpp:
6  ;
7  ; nasm -f coff first.asm
8  ; gcc -o first first.o driver.c asm_io.o
9
10 %include "asm_io.inc"
11 ;
12 ; Los datos iniciados se colocan en el segmento .data
13 ;
14 segment .data
15 ;
16 ; Estas etiquetas se refieren a las cadenas usadas para la salida
17 ;
18 prompt1 db    "Digite un número: ", 0       ; no olvide el fin de cadena
19 prompt2 db    "Digite otro número: ", 0
```

```
20 outmsg1 db    "Ud. ha digitado ", 0
21 outmsg2 db    " y ", 0
22 outmsg3 db    ", la suma es ", 0
23
24 ;
25 ; Los datos no iniciados se colocan en el segmento .bss
26 ;
27 segment .bss
28 ;
29 ; Estas etiquetas señalan a palabras dobles usadas para almacenar los datos
30 ; de entrada
31 ;
32 input1  resd 1
33 input2  resd 1
34
35 ;
36 ; El código se coloca en el segmento .text
37 ;
38 segment .text
39         global  _asm_main
40 _asm_main:
41         enter   0,0               ; setup routine
42         pusha
43
44         mov     eax, prompt1      ; print out prompt
45         call    print_string
46
47         call    read_int          ; lee un entero
48         mov     [input1], eax     ; lo almacena en input1
49
50         mov     eax, prompt2      ; print out prompt
51         call    print_string
52
53         call    read_int          ; lee un entero
54         mov     [input2], eax     ; lo almacena en input2
55
56         mov     eax, [input1]     ; eax = dword en input1
57         add     eax, [input2]     ; eax += dword en  input2
58         mov     ebx, eax          ; ebx = eax
59
60         dump_regs 1                ; imprime los valores de los registros
61         dump_mem  2, outmsg1, 1    ; imprimer la memoria
```

```
62  ;
63  ; ahora, se imprimen los resultados en una serie de pasos
64  ;
65          mov     eax, outmsg1
66          call    print_string      ; se imprime el primer mensaje
67          mov     eax, [input1]
68          call    print_int         ; se imprime input1
69          mov     eax, outmsg2
70          call    print_string      ; se imprime el segundo mensaje
71          mov     eax, [input2]
72          call    print_int         ; se imprime input2
73          mov     eax, outmsg3
74          call    print_string      ; se imprime el tercer mensaje
75          mov     eax, ebx
76          call    print_int         ; se imprime la suma (ebx)
77          call    print_nl          ; se imprime una nueva linea
78
79          popa
80          mov     eax, 0            ; retorna a  C
81          leave
82          ret
```

first.asm

La línea 13 del programa define una sección del programa que especifica la memoria del segmento de datos (cuyo nombre es `.data`). Solo los datos iniciados se deberían definir en este segmento. En las líneas 17 a 21 se declaran varias cadenas. Ellas serán impresas con las bibliotecas de C y como tal deben estar terminadas con el caracter *null* (el código ASCII 0). Recuerde que hay una gran diferencia entre `0` y `'0'`.

Los datos no iniciados deberían declararse en el segmento bss (llamado `.bss` en la línea 26). Este segmento toma su nombre de un operador de ensamblador basado en UNIX que significa "*block started by simbol*". Existe también el segmento de la pila. Será discutido después.

El segmento de código es llamado `.text` por razones históricas. Acá es donde se colocan las instrucciones. Observe que la etiqueta de la rutina principal (línea 38) tiene un prefijo de guión bajo. Esto es parte de las *convenciones de llamado de C*. Esta convención especifica las reglas que usa C cuando compila el código. Es muy importante conocer esta convención cuando se interfaza C con ensamblador. Luego se presentara toda la convención; sin embargo por ahora uno solo necesita conocer que todos los símbolos de C (funciones y variables globales) tienen un guión bajo como prefijo anexado a ellos por el compilador de C. (Esta regla es específica para DOS/Windows, el compilador de C de Linux no antepone nada a los nombres de los símbolos).

La directiva `global` en la línea 37 le dice al ensamblador que tome la etiqueta `_asm_main` como universal. A diferencia de C, las etiquetas tienen un *alcance interno* por omisión. Esto significa que solo el código en el mismo módulo puede usar la etiqueta. La directiva `global` da a la(s) etiqueta(s) específicada(s) un *alcance externo*. Este tipo de etiqueta puede ser conocido por cualquier módulo en el programa. El módulo `asm_io` declara las etiquetas `print_int` etc., globales. Este es el por qué uno puede usarlas en el módulo `first.asm`.

1.4.2. Dependencias del compilador

El código de ensamblador de arriba es específico del compilador libre GNU C/C++[10] DJGPP .[11] Este compilador puede ser descargado libremente de Internet. Requiere un PC 386 o posterior y se ejecuta bajo DOS, Windows 95/98 o NT. Este compilador usa archivos objeto con formato COFF (i*Common Objet File Format*). Para ensamblar este formato use la opción `-f coff` con `nasm` (como se muestra en los comentarios del código). La extensión del archivo objeto resultante será `o`.

El compilador de C de Linux también es GNU. Para convertir el código para que corra bajo Linux simplemente quita los guión bajos de prefijos en las líneas 37 y 38. Linux usa el formato ELF (*Excecutable an Linkable Format*) para los archivos objetos. Use la opción `-f elf` para Linux. También produce un objeto con una extensión `o`.

Los archivos de un compilador dado están disponibles en la página web del autor ya modificados para que trabajen con el compilador apropiado.

Borland C/C++ es otro compilador popular. Usa el formato de OMF de microsoft para los archivos objeto. Use la opción `-f obj` para los compiladores de Borland. La extensión del archivo objeto será `obj`. El formato OMF utiliza unas directivas de `segmento` diferentes que los otros formatos de objetos. El segmento de datos (línea 13) se debe cambiar a:

```
segment _DATA public align=4 class=DATA use32
```

el segmento bss (line 26) se debe cambiar a:

```
segment _BSS public align=4 class=BSS use32
```

El segmento text (line 36) se debe cambiar a:

```
segment _TEXT public align=1 class=CODE use32
```

Además se debe añadir una nueva línea antes de la línea 36.

```
group DGROUP _BSS _DATA
```

[10] GNU es un proyecto de la Free Software Foundation (`http://www.fsf.org`)

[11] `http://www.delorie.com/djgpp`

El compilador de Microsoft C/C++ puede usar el formato OMF o el Win32 para los archivos objeto (si le dan un formato OMF, él convierte la información internamente en Win32). El formato Win32 permite que los segmentos se definan tal como DJGPP y Linux. Use la opción `-f win32` para este formato. La extensión del archivo objeto será `obj`.

1.4.3. Ensamblando el código

El primer paso es ensamblar el código. Desde la línea de orden digite:

```
nasm -f formato-de-objeto first.asm
```

Donde el *formato del objeto* es *coff*, *elf*, *obj* o *win32* dependiendo que compilador de C será usado. (Recuerde que también se deben cambiar los archivos fuente para el caso de Linux y Borland).

1.4.4. Compilando el código de C

Compile el archivo `driver.c` usando un compilador de C. Para DJGPP, digite:

```
gcc -c driver.c
```

La opción `-c` significa que solo compile, no intente encadenar aún. Esta misma opción trabaja en los compiladores de Linux, Borland, y Microsoft también.

1.4.5. encadenando los archivos objeto

El encadenamiento es un proceso de combinar el código de máquina y los datos en archivos objeto con archivos de biblioteca para crear un archivo ejecutable. Como se verá adelante, este proceso es complicado.

El código de C requieren la biblioteca estandar de C y un *código de inicio* especial para ejecutarse. Es *mucho* más fácil dejar que el compilador de C llame al encadenador con los parámetros correctos que intentar llamar al encadenador directamente. Por ejemplo encadenar el código para el primer programa utilizando DJGPP, digite:

```
gcc -o first driver.o first.o asm_io.o
```

Esto crea un ejecutable llamado `first.exe` (o solo `first` bajo Linux).

Con Borland uno usaría:

```
bcc32 first.obj driver.obj asm_io.obj
```

Borland usa el nombre del primer archivo en la lista para determinar el nombre del ejecutable. Así en el caso anterior el programa debería llamarse `first.exe`.

Es posible combinar el paso de compilar y encadenar. Por ejemplo:

```
gcc -o first driver.c first.o asm_io.o
```

Ahora gcc compilará driver.C y entonces lo encadenará.

1.4.6. Entender un archivo de listado de ensamblador

La opción `-l` *`archivo-de-listado`* se puede usar para decirle a `nasm` que cree un archivo de listado con un nombre dado. Este archivo muestra cómo se ensambló el código. Se muestra cómo las líneas 17 y 18 (en el segmento data) aparecen en el archivo de listado. Los números de las líneas están en el archivo de listado; sin embargo observe que los números de las líneas en el archivo fuente pueden no ser los mismas que las del archivo de listado.

```
48 00000000 456E7465722061206E-     prompt1 db    "Enter a number: ", 0
49 00000009 756D6265723A2000
50 00000011 456E74657220616E6F-     prompt2 db    "Enter another number: ", 0
51 0000001A 74686572206E756D62-
52 00000023 65723A2000
```

La primera columna en cada línea es el número de línea y la segunda es el desplazamiento (en hex) de los datos en el segmento. La tercera columna muestra los valores en hexadecimal que serán almacenados. En este caso el dato hexadecimal corresponde a códigos ASCII. Finalmente en la línea se muestra el texto del código fuente. Los desplazamientos mostrados en la segunda columna es muy probable que *no* sean los desplazamientos reales, de los datos que serán colocados en el programa completo. Cada módulo puede definir sus propias etiquetas en el segmento de datos (y en los otros segmentos también). En el paso de encadenamiento vea la Sección 1.4.5, todas estas definiciones de segmentos y etiquetas son combinadas para formar un solo segmento de datos. El encadenador entonces calcula el desplazamiento definitivo.

Se muestra una pequeña sección (líneas 54 a 56 del archivo fuente) del segmento de texto en el archivo de listado.

```
94 0000002C A1[00000000]          mov     eax, [input1]
95 00000031 0305[04000000]        add     eax, [input2]
96 00000037 89C3                  mov     ebx, eax
```

La tercera columna muestra el código de máquina generado por el ensamblador. A menudo el código completo de una instrucción no se puede calcular aún. Por ejemplo, en la línea 94 el desplazamiento (o dirección) de `input1` no se conoce hasta que el código se encadene. El ensamblador puede calcular el código de la instrucción `mov` (que del listado es A1), pero escribe el desplazamiento en paréntesis cuadrados porque el valor exacto no se puede calcular en este momento. En este caso se utiliza un desplazamiento temporal de 0 porque `input1` está al inicio de la parte del segmento bss definido en este archivo. Recuerde que esto *no* significa que estará al comienzo del segmento bss definitivo del programa. Cuando el código es encadenado, el encadenador insertará el desplazamiento en la posición correcta. Otras instrucciones como la línea 96 no hacen referencia a ninguna etiqueta. En este caso el ensamblador puede calcular el código de máquina completo.

Representaciones Big y Little Endian

Si uno mira de cerca en la línea 95 hay algo muy extraño sobre el desplazamiento en los paréntesis cuadrados del código de máquina. La etiqueta `input2` tiene un desplazamiento de 4 (como está definido en este archivo); sin embargo, el desplazamiento que aparece en la memoria no es 0000004, pero 04000000. ¿Por qué? Diferentes procesadores almacenan enteros de varios bytes en ordenes diferentes en la memoria. Existen dos métodos populares de almacenar enteros: *big endian* y *littel endian*. Big endian es el método que se ve más natural. El byte mayor (*más significativo*) se almacena primero, y luego los siguientes. Por ejemplo, la palabra doble 00000004 se debería almacenar como los cuatro bytes 00 00 00 04. Los mainframes IBM, la mayoría de los procesadores RISC y los procesadores Motorola todos ellos usan el método de Big endian. Sin embargo los procesadores Intel usán el método little endian. De tal manera que se almacena primero el byte menos significativo. Así 00000004 se almacenará en memoria como 04 00 00 00. Este formato está cableado en la CPU y no se puede cambiar. Normalmente el programador no necesita preocuparse sobre que formato está usando. Sin embargo hay circunstancias donde esto es importante.

Endian se pronuncia como indian.

1. Cuando un dato binario es transfiere entre computadores diferentes (o de archivos o a través de una red)

2. Cuando un dato binario es escrito fuera de la memoria como un entero multibyte y luego se vuelve a leer como bytes individuales o *vice versa*

Lo Endian no se aplica al orden de los elementos de un arreglo. El primer elemento de un arreglo está siempre en la dirección menor. Esto se aplica a cadenas (que sólo son arreglos de caracteres). Lo Endian sólo se aplica a los elementos individuales de un arreglo.

skel.asm

```
1  %include "asm_io.inc"
2  segment .data
3  ;
4  ; los datos iniciados se colocan en el segmento de
5  ; datos acá
6  ;
7
8  segment .bss
9  ;
10 ; Datos no iniciados se colocan en el segmento bss
11 ;
12 segment .text
13         global  _asm_main
14 _asm_main:
15         enter   0,0               ; rutina de
16         pusha
17
18 ;
19 ; El código está colocado en el segmento de texto. No modifique el
20 ; código antes o después de este comentario
21 ;
22         popa
23         mov     eax, 0            ; retornar a C
24         leave
25         ret
```

skel.asm

Figura 1.7: Programa esqueleto

1.5. Archivo esqueleto

La Figura 1.7 muestra un archivo esqueleto que se puede usar como punto de partida para escribir programas en ensamblador

Capítulo 2

Lenguaje ensamblador básico

2.1. Trabajando con enteros

2.1.1. Representación de enteros

Hay dos tipos de enteros: sin signo y con signo. Los enteros sin signo (que son no negativos) están representados de una manera muy directa en binario natural. El número 200 como un byte entero sin signo sería representado como 11001000 (o C8 en hex).

Los enteros con signo (que pueden ser positivos o negativos) se representan de maneras más complejas. Por ejemplo considere -56 $+56$ como byte sería representado por 00111000. En el papel uno podría representar -56 como -111000, pero ¿Cómo podría representarse esto en un byte en la memoria del computador? ¿Cómo se almacenaría el signo menos?

Hay 3 técnicas que se han usado para representar números enteros en la memoria del computador. Todos estos métodos usan el bit más significativo como un *bit de signo*. Este bit es 0 si el número es positivo y 1 si es negativo.

Magnitud y signo

El primer método es el más elemental y es llamado *magnitud y signo*. Representa los enteros como dos partes. La primera es el bit de signo y la segunda es la magnitud del entero. Así 56 sería representado como el byte $\underline{0}0111000$ (el bit de signo está subrayado) y -56 sería $\underline{1}0111000$. El mayor valor de un byte sería $\underline{0}1111111$ o $+127$ y el menor valor sería $\underline{1}1111111$ o -127. Para negar un valor, se cambia el bit del signo. Este método es directo, pero tiene sus inconvenientes. Primero hay dos valores posibles de cero $+0$ ($\underline{0}0000000$) y -0 ($\underline{1}0000000$). Ya que cero no es ni positivo ni negativo, las dos representaciones podrían servir igual. Esto complica la lógica de la aritmética para la CPU. Segundo, la aritmética general también es complicada. Si se

añade 10 a −56, esta operación debe transformarse en la resta de 10 y 56. Una vez más, esto complica la lógica de la CPU.

Complemento a uno

El segundo método es conocido como complemento a uno. El complemento a uno de un número se encuentra invirtiendo cada bit en el número. (Otra manera de ver esto es que el nuevo valor del bit es $1 - \text{elvalorantiguodelbit}$). Por ejemplo el complemento a uno de $\underline{0}0111000$ (+56) es $\underline{1}1000111$. En la notación de complemento a uno calcular el complemento a uno es equivalente a la negación. Así $\underline{1}1000111$ es la representación de −56. Observe que el bit de signo fue cambiado automáticamente por el complemento a uno y que como se esperaría al aplicar el complemento a 1 dos veces produce el número original. Como el primer método, hay dos representaciones del cero $\underline{0}0000000$ (+0) y $\underline{1}1111111$ (−0). La aritmética con números en complemento a uno es complicada.

Hay un truco útil para encontrar el complemento a 1 de un número en hexadecimal sin convertirlo a binario. El truco es restar el dígito hexadecimal de F (o 15 en decimal). Este método supone que el número de dígitos binarios en el número es un múltiplo de 4. Un ejemplo: +56 se representa por 38 en hex. Para encontrar el complemento a uno reste F de cada dígito para obtener C7 en hexadecimal. Esto es coherente con el resultado anterior.

Complemento a dos

Los dos primeros métodos descritos fueron usados en los primeros computadores. Los computadores modernos usan un tercer método llamado la representación en complemento a dos. El complemento a dos de un número se halla con los dos pasos siguientes:

1. Hallar el complemento a uno del número.

2. Sumar uno al resultado del paso 1.

Acá está un ejemplo usando $\underline{0}0111000$ (56). Primero se calcula el complemento a uno: $\underline{1}1000111$. Entonces se añade uno:

$$\begin{array}{rr} & \underline{1}1000111 \\ + & \qquad\quad 1 \\ \hline & \underline{1}1001000 \end{array}$$

En la notación de complemento a dos, calcular el complemento a dos es equivalente a negar el número. Así $\underline{1}1001000$ es la representación en complemento a dos de −56. Dos negaciones deberían reproducir el número original.

Número	Representación Hex
0	00
1	01
127	7F
-128	80
-127	81
-2	FE
-1	FF

Cuadro 2.1: Representación de complemento a dos

Sorprendentemente el complemento a dos no reúne este requisito. Tome el complemento a dos de $\underline{1}1001000$ añadiendo uno al complemento a uno.

$$\begin{array}{rr} & \underline{0}0110111 \\ + & \qquad 1 \\ \hline & \underline{0}0111000 \end{array}$$

Cuando realizamos la suma en una operación en complemento a dos, la suma del bit del extremo izquierdo puede producir un carry. Este carry *no* se usa. Recuerde que todos los datos en el computador son de un tamaño fijo (en términos de números de bits). Sumar dos bytes siempre produce como resultado un byte (tal como sumar dos palabras produce otra palabra, etc.). Esta propiedad es importante para la notación en complemento a dos. Por ejemplo, considere el cero como un número en complemento a dos de un byte ($\underline{0}0000000$). Calcular el complemento a 2 produce la suma:

$$\begin{array}{rr} & \underline{1}1111111 \\ + & \qquad 1 \\ \hline c & \underline{0}0000000 \end{array}$$

donde c representa un carry (luego se mostrará como detectar este carry, pero no se almacena en el resultado). Así en la notación de complemento a dos existe solo un cero. Esto hace la aritmética de complemento a dos más simple que los métodos anteriores.

Usando la notación en complemento a dos, un byte con signo se puede usar para representar los números desde -128 hasta $+127$. El Cuadro 2.1 muestra algunos valores seleccionados. Si se usan 16 bits, se pueden representar los números con signo desde $-32{,}768$ hasta 32,767 que está representado por 7FFF, $-32{,}768$ por 8000, -128 como FF80 y -1 como FFFF. Los números de 32 bits en complemento a dos están en el rango de -2 mil millones a $+2$ mil millones aproximadamente.

La CPU no tiene ni idea que supuesta representación tiene un byte en particular (palabra, o palabra doble). El ensamblador no tiene ni idea de los

tipos de datos que tienen los lenguajes de alto nivel. Cómo se interpretan los datos depende de qué instrucción se usa con el dato. Si el valor FF representa −1 o +255 depende del programador. El lenguaje C define tipos de entero con y sin signo (`signed`, `unisigned`). Esto le permite al compilador determinar las instrucciones correctas a usar con el dato.

2.1.2. Extensión del signo

En ensamblador, todos los datos tienen un tamaño determinado. No es raro necesitar cambiar el tamaño del dato para usarlo con otro dato. Reducir el tamaño es fácil.

Reduciendo el tamaño de los datos

Para reducir el tamaño del dato simplemente quite los bits más significativos del dato. Un ejemplo trivial:

```
mov    ax, 0034h      ; ax = 52 (almacenado en16 bits)
mov    cl, al         ; cl = los 8-bits inferiores de ax
```

Claro está, si el número no se puede representar correctamente en el tamaño más pequeño, la reducción de tamaño no funcionará. Por ejemplo si `AX` era 0134h (o 308 en decimal) entonces el código anterior almacenaría en `CL` 34h. Este método trabaja con números con o sin signo. Considere números con signo, si `AX` era FFFFh (-1 como palabra), entonces code CL sería FFh (−1 como byte). Sin embargo, observe que ¡esto no es correcto si el valor en `AX` era sin signo!

La regla para números sin signo es que todos los bits al ser quitados deben ser 0 para que la conversión sea correcta. La regla para los números con signo es que los bits que sean quitados deben ser o todos 1 o todos 0. Además el primer bit no se debe quitar pero debe tener el mismo valor que los bits quitados. Este bit será el nuevo bit de signo del valor más pequeño. Es importante que sea el bit del signo original.

Aumentando el tamaño de los datos

Incrementar el tamaño de los datos es más complicado que disminuirlo. Considere el byte hex FF. Si se extiende a una palabra, ¿Qué valor debería tener la palabra? Depende de cómo se interprete la palabra. Si FF es un byte sin signo (255 en decimal), entonces la palabra debería ser 00FF; sin embargo, si es un byte con signo (−1 en decimal), entonces la palabra debería ser FFFF.

En general, para extender un número sin signo, uno hace cero todos los bits nuevos del número extendido. Así FF se convierte en 00FF. Sin

embargo, para extender un número con signo uno debe *extender* el bit de signo. Esto significa que los nuevos bits se convierten en copias del bit de signo. Ya que el bit de signo de FF es 1, los nuevos bits deben ser todos unos, para producir FFFF. Si el número con signo 5A (90 en decimal) fue extendido, el resultado sería 005A.

Existen varias instrucciones que suministra el 80386 para la extensión de los números. Recuerde que el computador no conoce si un número está con o sin signo. Es responsabilidad del programador usar la instrucción adecuada.

Para números sin signo, uno puede simplemente colocar ceros en los bits superiores usando una instrucción `MOV`. Por ejemplo, para extender el byte en AL a una palabra sin signo en AX:

```
mov    ah, 0   ; cero los  8-bits superiores
```

Sin embargo, no es posible usar la instrucción `MOV` para convertir la palabra sin signo en AX a una palabra doble en EAX. ¿Por qué no? No hay manera de referirse a los 16 bits superiores de EAX con una instrucción `MOV`. El 80386 resuelve este problema con una nueva instrucción `MOVZX`. Esta instrucción tiene dos operandos. El destino (primer operando) debe ser un registro de 16 o 32 bits. La fuente (segundo operando) puede ser un registro de 8 o 16 bits o un byte o una palabra en memoria. La otra restricción es que el destino debe ser mayor que la fuente. (La mayoría de instrucciones requieren que la fuente y el destino sean del mismo tamaño). Algunos ejemplos:

```
movzx  eax, ax      ; extiende ax en eax
movzx  eax, al      ; extiende al en eax
movzx  ax, al       ; extiende al en ax
movzx  ebx, ax      ; extiende ax en ebx
```

Para números con signo, no hay una manera fácil de usar la instrucción `MOV`. EL 8086 suministra varias instrucciones para extender números con signo. La instrucción `CBW` (*Convert Byte to Word*) extiende el registro AL en AX. Los operandos son implícitos. La instrucción CWD (*Convert Word to Double Word*) extiende AX en DX:AX. La notación DX:AX implica interpretar los registros DX y AX como un registro de 32 bits con los 16 bits superiores almacenados en DX y los 16 bits inferiores en AX. (Recuerde que el 8086 no tenía ningún registro de 32 bits). El 80386 añadió varias instrucciones nuevas. La instrucción `CWDE` (*Convert Word to Double word Extended*) extiende AX en EAX. La instrucción `CDQ` (*Convert Double word to Quad word*) extiende EAX en EDX:EAX (¡64 bits!). Finalmente, la instrucción `MOVSX` trabaja como `MOVZX` excepto que usa las reglas para números con signo.

```
unsigned char uchar = 0xFF;
signed char   schar = 0xFF;
int a = (int) uchar;      /* a = 255 (0x000000FF) */
int b = (int) schar;      /* b = -1 (0xFFFFFFFF) */
```

Figura 2.1:

```
char ch;
while( (ch = fgetc(fp)) != EOF ) {
  /* hace algo con ch */
}
```

Figura 2.2:

Aplicación a la programación en C

ANSI C no define si el tipo `char` *es con signo o no, es responsabilidad de cada compilador decidir esto. Esta es la razón por la cual el tipo está explícitamente definido en la Figura 2.1.*

Extender enteros con y sin signo también ocurre en C. Las variables en C se pueden declarar como `int` signed o unsigned (int es signed). Considere el código de la Figura 2.1. En la línea 3, la variable `a` se extiende usando las reglas para valores sin signo (usando `MOVZX`), pero en la línea 4 se usan las reglas con signo para `b` (usando `MOVSX`).

Hay un error muy común en la programación en C que tiene que ver con esto directamente. Considere el código de la Figura 2.2. El prototipo de `fgetc()` es:

```
int fgetc( FILE * );
```

Uno podría preguntar ¿Por qué la función retorna un `int` siendo que lee caracteres? La razón es que normalmente retorna un `char` (extendido a un valor entero usando la extensión cero). Sin embargo hay un valor que puede retornar que no es un carácter, `EOF`. Este es un macro que normalmente se define como -1. Así `fgetc()` o retorna un carácter extendido a entero (que es como `000000`xx en hex) o `EOF` (que es `FFFFFFF` en hex).

El problema principal con el programa de la Figura 2.2 es que `fgetc()` retorna un entero, pero este valor se almacena en un `char`. C truncará los bits de mayor peso para que el entero quepa en el caracter. El único problema es que los números (en hex) `000000FF` y `FFFFFFFF` ambos se truncarán al byte FF. Así el ciclo while no puede distinguir entre el byte `FF` y el fin de archivo (`EOF`).

Lo que sucede exactamente en este caso, depende de si el `char` es con signo o sin signo ¿por qué? Porque en la línea 2 `ch` es comparada con `EOF`. Ya que `EOF` es un valor `int`[1], `ch` será extendido a un `int` de modo que los

[1]Es un concepto erróneo pensar que los archivos tienen un carácter EOF al final. ¡Esto

dos valores comparados sean del mismo tamaño[2]. como se muestra en la Figura 2.1, donde si la variable es con signo o sin signo es muy importante.

Si `char` es unsigned, `FF` se extenderá a `000000FF`. Esto es comparo con `EOF` (`FFFFFFFF`) y encontrará que no es igual. Así, ¡el bucle nunca terminará!

Si `char` es signed se extenderá a `FFFFFFFF`. Esto se compara como igual y el bucle finaliza. Sin embargo, ya que el byte `FF` puede haber sido leído de un archivo, el bucle podría terminar prematuramente.

La solución a este problema es definir la variable `ch` como un `int` no como un `char`. Cuando esto se hace no se truncará o extenderá en la línea 2. Dentro del bucle es seguro truncar el valor ya que ahí `ch` *debe* ser un simple byte.

2.1.3. Aritmética de complemento a dos

Como se vio al principio, la instrucción `add` efectúa una suma y la instrucción `sub` efectúa una resta. Dos de los bits en el registro FLAGS, que se alteran con estas instrucciones son las banderas de *desborde* y *carry*. La bandera de desborde se fija si el resultado verdadero de la operación es muy grande para caber en el destino para aritmética con signo. La bandera de carry se fija si hay carry en el bit más significativo de una suma o un préstamo en el bit más significativo de una resta. Así, él se puede usar para detectar un desborde para aritmética sin signo. Los usos de la bandera de carry para aritmética con signo se verán dentro de poco. Una de las grandes ventajas del complemento a 2 es que las reglas para la adición y sustracción son exactamente las mismas que para los enteros sin signo. Así `add` y `sub` se pueden usar con enteros con o sin signo.

$$\begin{array}{rr} 002C & 44 \\ +\ \text{FFFF} & +\ (-1) \\ \hline 002B & 43 \end{array}$$

Hay un carry generado, pero no es parte de la respuesta.

Hay dos instrucciones diferentes para multiplicar y dividir. Primero para multiplicar use la instrucción `MUL` o `IMUL`. La instrucción `MUL` se emplea para multiplicar números sin signo e IMUL se usa para multiplicar enteros con signo. ¿Por qué se necesitan dos instrucciones diferentes? Las reglas para la multiplicación son diferentes para números en complemento a dos con signo o sin signo. ¿Cómo así? Considere la multiplicación del byte FF con sí mismo dando como resultado una palabra. Usando una multiplicación sin signo es 255 veces 255 o 65025 (o FE01 en hex). Usando la multiplicación con signo es -1 veces -1 (o 0001 en hex).

---

no es verdad!

[2]La razón para este requerimiento se mostrará luego.

dest	fuente1	fuente2	Acción
	reg/mem8		AX = AL*fuente1
	reg/mem16		DX:AX = AX*fuente1
	reg/mem32		EDX:EAX = EAX*fuente1
reg16	reg/mem16		dest *= fuente1
reg32	reg/mem32		dest *= fuente1
reg16	immed8		dest *= immed8
reg32	immed8		dest *= immed8
reg16	immed16		dest *= immed16
reg32	immed32		dest *= immed32
reg16	reg/mem16	immed8	dest = fuente1*fuente2
reg32	reg/mem32	immed8	dest = fuente1*fuente2
reg16	reg/mem16	immed16	dest = fuente1*fuente2
reg32	reg/mem32	immed32	dest = fuente1*fuente2

Cuadro 2.2: instrucciones `imul`

Hay varias formas de las instrucciones de multiplicación. La más antigua es:

```
mul   fuente
```

fuente es un registro o una referencia a memoria. No puede ser un valor inmediato. Exactamente qué multiplicación se realiza depende del tamaño del operando fuente. Si el operando es de un byte, éste es multiplicado por el byte del registro AL y el resultado se almacena en los 16 bits de AX. Si la fuente es de 16 bits, se multiplica por la palabra en AX y el resultado de 32 bits se almacena en DX:AX. Si la fuente es de 32 bits éste se multiplica por EAX y el resultado de 64 bits se almacena en EDX:EAX.

La instrucción `IMUL` tiene la misma forma de `MUL`, pero también tiene algunos otros formatos. Hay dos y tres formas de operandos.

```
imul   dest, fuente1
imul   dest, fuente1, fuente2
```

El Cuadro 2.2 muestra las posibles combinaciones.

Los dos operadores para la división son `DIV` e `IDIV`. Ellas efectúan la división sin signo y con signo respectivamente. El formato general es:

```
div   fuente
```

Si la fuente es de 8 bits, entonces AX es dividido por el operando. El cociente se almacena en AL y el residuo en ah. Si la fuente es de 16 bits, entonces DX:AX se divide por el operando. El cociente se almacena en AX y el residuo en DX. Si la fuente es de 32 bits, entonces EDX:EAX se divide por el

operando y el cociente se almacena en EAX y el residuo en EDX. La instrucción `IDIV` trabaja de la misma manera. No hay instrucciones especiales `IDIV` como las especiales en `IMUL`. Si el cociente es muy grande para caber en el registro o el divisor es cero, el programa se interrumpe y termina. Un error muy común es olvidar iniciar DX o EDX antes de la división.

La instrucción `NEG` niega su operando calculando su complemento a dos. El operando puede ser cualquier registro de 8, 16 o 32 bits o un lugar de memoria.

2.1.4. Programa de ejemplo

math.asm

```
%include "asm_io.inc"
segment .data         ; Cadenas de salida
prompt          db    "Digite un número: ", 0
square_msg      db    "La entrada al cuadrado es ", 0
cube_msg        db    "La entrada al cubo es ", 0
cube25_msg      db    "La entrada al cubo 25 veces es ", 0
quot_msg        db    "El cociente del cubo/100 es ", 0
rem_msg         db    "El residuo del cube/100 es ", 0
neg_msg         db    "La negación del residuo es ", 0

segment .bss
input   resd 1

segment .text
        global  _asm_main
_asm_main:
        enter   0,0               ; rutina de inicio
        pusha

        mov     eax, prompt
        call    print_string

        call    read_int
        mov     [input], eax

        imul    eax               ; edx:eax = eax * eax
        mov     ebx, eax          ; guarda la respuesta en  ebx
        mov     eax, square_msg
        call    print_string
        mov     eax, ebx
        call    print_int
```

```
32          call    print_nl
33
34          mov     ebx, eax
35          imul    ebx, [input]      ; ebx *= [input]
36          mov     eax, cube_msg
37          call    print_string
38          mov     eax, ebx
39          call    print_int
40          call    print_nl
41
42          imul    ecx, ebx, 25      ; ecx = ebx*25
43          mov     eax, cube25_msg
44          call    print_string
45          mov     eax, ecx
46          call    print_int
47          call    print_nl
48
49          mov     eax, ebx
50          cdq                       ; inicia edx con la extensión de signo
51          mov     ecx, 100          ; no puede dividirse por el valor inmediato
52          idiv    ecx               ; edx:eax / ecx
53          mov     ecx, eax          ; guarda el cociente ecx
54          mov     eax, quot_msg
55          call    print_string
56          mov     eax, ecx
57          call    print_int
58          call    print_nl
59          mov     eax, rem_msg
60          call    print_string
61          mov     eax, edx
62          call    print_int
63          call    print_nl
64
65          neg     edx               ; niega el residuo
66          mov     eax, neg_msg
67          call    print_string
68          mov     eax, edx
69          call    print_int
70          call    print_nl
71
72          popa
73          mov     eax, 0            ; retorna a  C
```

```
74          leave
75          ret
```
math.asm

2.1.5. Aritmética de precisión extendida

El lenguaje ensamblador también suministra instrucciones que le permitan a uno hacer operaciones de suma y resta de números más grandes que palabras dobles. Como se vio antes las instrucciones `ADD` y `SUB` modifican la bandera de carry si se ha generado un carry o un préstamo respectivamente. Esta información almacenada en la bandera de carry se puede usar para sumar o restar números grandes dividiendo la operación en piezas pequeñas de palabras dobles (o menores).

Las instrucciones `ADC` y `SBB` usan esta información en la bandera de carry. La instrucción `ADC` hace la siguiente operación:

operando1 = *operando1* + bandera de carry + *operando2*

La instrucción SBB realiza:

operando1 = *operando1* - bandera de flag - *operando2*

¿Cómo se usan? Considere la suma de enteros de 64 bits en EDX:EAX y EBX:ECX . El siguiente código podía almacenar la suma en EDX:EAX

```
1   add    eax, ecx       ; suma los 32-bits inferiores
2   adc    edx, ebx       ; suma los 32-bits superiores y el carry
3                             ; de la suma anterior
```

La resta es muy similar. El siguiente código resta EBX:ECX de EDX:EAX

```
1   sub    eax, ecx       ; resta los 32-bits inferiores
2   sbb    edx, ebx       ; resta los 32-bits superiores y el préstamo
```

Para números *realmente* grandes, se puede usar un bucle (vea sección 2.2). Para el bucle suma sería conveniente usar la instrucción ADC para cada iteración (en todas menos la primera). Esto se puede hacer usando la instrucción `CLC` (*CLear Carry*) antes que comience el bucle para iniciar la bandera de carry a cero. Si la bandera de carry es cero no hay diferencia entre `ADD` y `ADC`. La misma idea se puede usar también para la resta.

2.2. Estructuras de control

Los lenguajes de alto nivel suministran estructuras del alto nivel (vgr instrucciones *if while*), que controlan el flujo de ejecución. El lenguaje ensamblador no suministra estas complejas estructuras de control. En lugar

de ello usa el difamado *goto* y su uso inapropiado puede resultar en un ¡código spaghetti! Sin embargo *es* posible escribir programas estructurados en ensamblador. El procedimiento básico es diseñar la lógica del programa usando las estructuras de control de alto nivel y traducir el diseño en lenguaje ensamblador apropiado (parecido a lo que haría el compilador).

2.2.1. Comparaciones

Las estructuras de control deciden que hacer basados en la comparación de datos. En ensamblador, el resultado de una comparación se almacenan en el registro FLAGS para usarlas luego. El 80x86 suministra la instrucción `CMP` para realizar comparaciones. El registro FLAGS se fija basado en la diferencia de los dos operandos de la instrucción `CMP`. Los operandos se restan y se fija el registro FLAGS basado en el resultado, pero el resultado *no* se almacena en ninguna parte. Si necesita el resultado use la instrucción SUB en lugar de la instrucción `CMP`.

Para enteros sin signos hay dos banderas (bits en el registro FLAGS) que son importante: cero (ZF) y carry (CF). La bandera cero se fija (1) si el resultado de la resta es cero. La bandera carry se usa como bandera de préstamo para la resta. Considere una comparación como:

```
cmp    vleft, vright
```

Se calcula la diferencia de `vleft - vright` y las banderas se fijan de acuerdo al resultado. Si la diferencia de `CMP` es cero, `vleft = vright`, entonces ZF se fija (ZF=1) y CF se borra (CF=0). Si `vleft > vright`, entonces ZF se borra y CF también (no hay préstamo). Si `vleft < vright`, entonces ZF se borrará y CF se fijará (hay préstamo).

¿Por qué hace SF = OF si `vleft > vright`*? Si no hay desborde, entonces la diferencia tendrá el valor correcto y debe ser no negativo. Así, SF = OF = 0. Sin embargo, si hay un desborde, la diferencia no tendrá el valor correcto (y de hecho será negativo). Así SF = OF = 1.*

Para enteros con signo, hay tres banderas que son importante: la bandera cero (ZF), la bandera de desborde (OF) y la bandera de signo (SF). La bandera de desborde se fija si el resultado de una operación se desborda (o underflows). La bandera de signo se fija si el resultado de una operación es negativo. Si `vleft = vright`, la ZF se fija (tal como para los enteros sin signo). Si `vleft > vright`, ZF es cero y SF = OF. Si `vleft < vright`, ZF es cero y SF $\neq$ OF.

No olvide que otras instrucciones también cambian el registro FLAGS, no solo CMP.

2.2.2. Instrucciones de ramificación

Las instrucciones de ramificación pueden transferir la ejecución del programa a un punto arbitrario. En otras palabras funcionan como *goto*. Hay dos tipos de ramificaciones: condicionales e incondicionales. Una ramificación incondicional es tal cual como goto, siempre hace el salto. Una ramificación

condicional puede o no hacer el salto dependiendo de las banderas del registro FLAGS. Si una ramificación condicional no hace el salto, el control pasa a la siguiente instrucción.

La instrucción `JMP` (acrónimo de *jump*) hace ramificaciones incondicionales. Su argumento normalmente es la *etiqueta de código* de la instrucción a la cual debe saltar. El ensamblador o el encadenador reemplazará la etiqueta con la dirección correcta de la instrucción. Esta es otra de las labores aburridas que realiza el ensamblador para hacer la vida del programador más fácil. Es importante tener en cuenta que la instrucción inmediatamente después de la instrucción `JMP` nunca se ejecutará a menos que otra instrucción salte a ella.

Hay variaciones de la instrucción de salto.

SHORT Este salto es de tamaño muy limitado, solo se puede mover arriba o abajo 128 bytes en memoria. La ventaja de este tipo es que usa menos memoria que otros. Usa un byte con signo para almacenar el *desplazamiento* del salto. El desplazamiento es cuántos bytes se mueve adelante o atrás. (El *desplazamiento* se añade a EIP). Para especificar un salto corto, use la palabra `SHORT` inmediatamente antes de la etiqueta en la instrucción `JMP`.

NEAR Este salto es el tipo por omisión en las ramificaciones condicionales e incondicionales y se puede usar para saltar a cualquier lugar del segmento. Actualmente el 80386 soporta 2 tipos de saltos cercanos. Uno usa dos bytes para el desplazamiento. Esto le permite a uno moverse aproximadamente 32000 bytes arriba o abajo. El otro tipo usa cuatro bytes para el desplazamiento, que le permite a uno moverse a cualquier lugar en el segmento de código. El tipo de 4 bytes es el de defecto en el modo protegido del 386. El tipo de 2 bytes se puede especificar colocando la palabra WORD antes de la etiqueta en la instrucción `JMP`.

FAR Este salto permite moverse a otro segmento de código. Este es una cosa muy rara para hacerla en el modo protegido del 386.

Las etiquetas de código válidas siguen las mismas reglas que las etiquetas de datos. Las etiquetas de código están definidas para colocarlas en el segmento de código al frente de la instrucción sus etiquetas. Dos puntos se colocan al final de la etiqueta en este punto de la definición. Los dos puntos *no* son parte del nombre.

Hay muchas instrucciones de ramificación condicional diferentes. Ellas también toman una etiqueta como su operando. Las más sencillas solo ven una bandera en el registro FLAGS para determinar si salta o no. Vea el Cuadro 2.3 para una lista de estas instrucciones (PF es la *bandera de paridad*

JZ	salta solo si ZF es uno
JNZ	salta solo si ZF es cero
JO	salta solo si OF es uno
JNO	salta solo si OF es cero
JS	salta solo si SF es uno
JNS	salta solo si SF es cero
JC	salta solo si CF es uno
JNC	salta solo si CF es cero
JP	salta solo si PF es uno
JNP	salta solo si PF es cero

Cuadro 2.3: Saltos condicionales simples

que indica si el número de unos en los 8 bit inferiores del resultado de la operación es par o impar).

El siguiente pseudocódigo:

```
if ( EAX == 0 )
  EBX = 1;
else
  EBX = 2;
```

Se podría escribir en ensamblador como:

```
1        cmp    eax, 0            ; establece las banderas
2                                      ; (ZF se fija si  eax - 0 = 0)
3        jz     thenblock         ; si  ZF es 1 salta a thenblock
4        mov    ebx, 2            ; parte ELSE del if
5        jmp    next              ; salta sobre la parte THEN del IF
6  thenblock:
7        mov    ebx, 1            ; parte THEN del IF
8  next:
```

Las otras comparaciones no son fáciles usando las ramificaciones condicionales de el Cuadro 2.3 Para ilustrar esto considere el siguiente pseudocódigo:

```
if ( EAX >= 5 )
  EBX = 1;
else
  EBX = 2;
```

Si EAX es mayor que o igual a 5, ZF debe estar fija o borrada y SF será igual a OF. A continuación está el código en ensamblador que prueba estas condiciones (asumiendo que EAX es con signo):

Signed		Unsigned	
JE	salta si `vleft = vright`	JE	salta si `vleft = vright`
JNE	salta si `vleft` $\neq$ `vright`	JNE	salta si `vleft` $\neq$ `vright`
JL, JNGE	salta si `vleft <vright`	JB, JNAE	salta si `vleft <vright`
JLE, JNG	salta si `vleft` $\leq$ `vright`	JBE, JNA	salta si `vleft` $\leq$ `vright`
JG, JNLE	salta si `vleft >vright`	JA, JNBE	salta si `vleft >vright`
JGE, JNL	salta si `vleft` $\geq$ `vright`	JAE, JNB	salta si `vleft` $\geq$ `vright`

Cuadro 2.4: Instrucciones de comparación con y sin signo

```
1        cmp    eax, 5
2        js     signon            ; salta a signon si SF = 1
3        jo     elseblock         ; salta a elseblock si OF = 1 y SF = 0
4        jmp    thenblock         ; salta a thenblock si SF = 0 y OF = 0
5   signon:
6        jo     thenblock         ; salta a thenblock si SF = 1 y OF = 1
7   elseblock:
8        mov    ebx, 2
9        jmp    next
10  thenblock:
11       mov    ebx, 1
12  next:
```

El código anterior es muy complicado. Afortunadamente, el 80x86 suministra otras instrucciones de ramificación que hace este tipo de pruebas *mucho* más fácil. Hay versiones con y sin signo para cada tipo. El Cuadro 2.4 muestra estas instrucciones. Las ramificaciones igual y no igual (JE y JNE) son idénticas para enteros con y sin signo. (De hecho JE y JNE son idénticas a JZ y JNZ respectivamente). Cada una de las otras instrucciones de ramificación tienen dos sinónimos. Por ejemplo observe que JL (*Jump Less than*) y JNGE (*Jump Not Greater than or Equal to*). Son la misma instrucción porque:

$$x < y \Longrightarrow \mathbf{not}(x \geq y)$$

Las ramificaciones sin signo usan A por *above* y B por *below* en lugar de L (*Less*) y G(*Greater*).

Usando estas nuevas instrucciones de ramificación, el pseudocódigo de arriba puede ser traducido a ensamblador mucho más fácil.

```
1        cmp    eax, 5
2        jge    thenblock
3        mov    ebx, 2
4        jmp    next
```

```
5  thenblock:
6        mov    ebx, 1
7  next:
```

2.2.3. Instrucciones de bucle

El 80x86 suministra varias instrucciones para implementar bucles del tipo *for*. Cada una de esas instrucciones tiene una etiqueta como único operando.

LOOP Decrementa ECX, si ECX $\neq 0$, salta a la etiqueta

LOOPE, LOOPZ Decrementa ECX (el registro FLAGS no se modifica), si ECX $\neq 0$ y ZF $= 1$, salta

LOOPNE, LOOPNZ Decrementa ECX (FLAGS sin cambio), si ECX $\neq 0$ y ZF $= 0$, salta

Las dos últimas instrucciones de bucle son útiles para bucles de búsqueda secuencial. El siguiente pseudocódigo:

```
sum = 0;
for( i=10; i >0; i-- )
  sum += i;
```

Podría ser traducido a ensamblador como:

```
1        mov    eax, 0          ; eax es sum
2        mov    ecx, 10         ; ecx es i
3  loop_start:
4        add    eax, ecx
5        loop   loop_start
```

2.3. Traducir estructuras de control estándares

Esta sección muestra cómo las estructuras de control estándares de los lenguajes de alto nivel se pueden implementar en lenguaje ensamblador.

2.3.1. instrucciones if

El siguiente pseudocódigo:

```
if ( condición )
  bloque entonces;
else
  bloque else;
```

podría implementarse como:

```
        ; code to set FLAGS
        jxx    else_block    ; selecciona xx tal que salta si la
                                 ; condición es falsa
        ; código para bloque entonces
        jmp    endif
else_block:
        ; código para bloque else
endif:
```

Si no hay else, entonces el salto al `else_block` puede ser reemplazado por un salto a `endif`.

```
        ; Código para establecer FLAGS
        jxx    endif          ; selecciona xx tal que salta si la condición es falsa
        ; código para el bloque entonces
endif:
```

2.3.2. bucles while

El bucle *while* se prueba al inicio del bucle:

while(condición) {
  cuerpo del bucle;
}

Esto podría traducirse en:

```
while:
        ; código que fija FLAGS basado en la condición
        jxx    endwhile       ; selecciona xx tal que salte si es falso
        ; Cuerpo del bucle
        jmp    while
endwhile:
```

2.3.3. bucles do while

El bucle *do while* se prueba al final del bucle:

do {
  cuerpo del bucle;
} **while**(condición);

Esto podría traducirse en:

```
unsigned guess;   /* La conjetura actual para el primo      */
unsigned factor;  /* el posible factor      */
unsigned limit;   /* encontrar primos hasta este valor */

printf("Find primes up to: ");
scanf("%u", &limit);
printf("2\n");    /* trata los dos primeros primos  */
printf("3\n");    /* como caso especial             */
guess = 5;        /* conjetura inicial */
while ( guess <= limit ) {
  /* busca un factor */
  factor = 3;
  while ( factor*factor < guess &&
          guess % factor != 0 )
    factor += 2;
  if ( guess % factor != 0 )
    printf("%d\n", guess);
  guess += 2;   /* sólo busca en los números impares */
}
```

Figura 2.3:

```
do:
      ; cuerpo del bucle
      ; código para fijar FLAGS basado en la condición
      jxx    do          ; seleccionar xx tal que salte si es verdadero
```

2.4. Ejemplo: hallar números primos

Esta sección muestra un programa que encuentra números primos. Recuerde que un número primo es divisible sólo por 1 y por sí mismo. No hay fórmula para hacer esto. El método básico de este programa es encontrar los factores de todos los números impares[3] bajo un límite dado. Si no se puede encontrar un factor para un número impar, es primo La Figura 2.3 muestra el algoritmo básico escrito en C.

A continuación la versión en ensamblador

prime.asm

```
%include "asm_io.inc"
segment .data
```

[3] 2 es el único número par.

```
3  Message         db      "Halle primos hasta: ", 0
4
5  segment .bss
6  Limit           resd    1               ; halle primos hasta este límite
7  Guess           resd    1               ; la conjetura actual para el primo
8
9  segment .text
10         global  _asm_main
11 _asm_main:
12         enter   0,0               ; rutina de inicio
13         pusha
14
15         mov     eax, Message
16         call    print_string
17         call    read_int            ; scanf("%u", & limit );
18         mov     [Limit], eax
19
20         mov     eax, 2              ; printf("2\n");
21         call    print_int
22         call    print_nl
23         mov     eax, 3              ; printf("3\n");
24         call    print_int
25         call    print_nl
26
27         mov     dword [Guess], 5    ; Guess = 5;
28 while_limit:                        ; while ( Guess <= Limit )
29         mov     eax,[Guess]
30         cmp     eax, [Limit]
31         jnbe    end_while_limit     ; se usa jnbe ya que los números son sin signo
32
33         mov     ebx, 3              ; ebx es factor = 3;
34 while_factor:
35         mov     eax,ebx
36         mul     eax                 ; edx:eax = eax*eax
37         jo      end_while_factor    ; Si la respuesta no cabe en eax
38         cmp     eax, [Guess]
39         jnb     end_while_factor    ; if !(factor*factor < guess)
40         mov     eax,[Guess]
41         mov     edx,0
42         div     ebx                 ; edx = edx:eax % ebx
43         cmp     edx, 0
44         je      end_while_factor    ; if !(guess % factor != 0)
```

```
45
46          add     ebx,2                   ; factor += 2;
47          jmp     while_factor
48  end_while_factor:
49          je      end_if                  ; if !(guess % factor != 0)
50          mov     eax,[Guess]             ; printf("%u\n")
51          call    print_int
52          call    print_nl
53  end_if:
54          add     dword [Guess], 2        ; guess += 2
55          jmp     while_limit
56  end_while_limit:
57
58          popa
59          mov     eax, 0              ; retorna a C
60          leave
61          ret
```

prime.asm

Capítulo 3

Operaciones con bits

3.1. Operaciones de desplazamientos

El lenguaje ensamblador le permite al programador manipular bits individuales de los datos. Una operación común es llamada un *desplazamiento*. Una operación de desplazamiento mueve la posición de los bits de algún dato. Los desplazamientos pueden ser hacia la izquierda (hacia el bit más significativo) o hacia la derecha (el bit menos significativo).

3.1.1. Desplazamientos lógicos

Un desplazamiento lógico es el tipo más simple de desplazamiento. Desplaza de una manera muy directa. La Figura 3.1 muestra un ejemplo del desplazamiento de un byte.

Original	1	1	1	0	1	0	1	0
Desplazado a la izquierda	1	1	0	1	0	1	0	0
Desplazado a la derecha	0	1	1	1	0	1	0	1

Figura 3.1: Desplazamientos lógicos

Observe que los nuevos bits que entran son siempre cero. Se usan las instrucciones `SHL` y `SHR` para realizar los desplazamientos a la izquierda y derecha respectivamente. Estas instrucciones le permiten a uno desplazar cualquier número de posiciones. El número de posiciones puede ser o una constante o puede estar almacenado en el registro `CL`. El último bit desplazado se almacena en la bandera de carry. A continuación, algunos ejemplos:

```
1  mov    ax, 0C123H
2  shl    ax, 1           ; desplaza un bit a la izquierda,
3                         ; ax = 8246H, CF = 1
4  shr    ax, 1           ; desplaza un bit a la derecha,
```

```
5                               ; ax = 4123H, CF = 0
6       shr     ax, 1           ; desplaza un bit a la derecha,
7                               ; ax = 2091H, CF = 1
8       mov     ax, 0C123H
9       shl     ax, 2           ; desplaza dos bit a la izquierda,
10                              ;ax = 048CH, CF = 1
11      mov     cl, 3
12      shr     ax, cl          ; desplaza tres bit a la derecha,
13                              ; ax = 0091H, CF = 1
```

3.1.2. Uso de los desplazamientos

Los usos más comunes de las operaciones de desplazamientos son las multiplicaciones y divisiones rápidas. Recuerde que en el sistema decimal la multiplicación y división por una potencia de 10 es sólo un desplazamiento de los dígitos. Lo mismo se aplica para las potencias de dos en binario. Por ejemplo para duplicar el número binario 1011_2 (o 11 en decimal), al desplazar una vez a la izquierda obtenemos 10110_2 (o 22). El cociente de una división por una potencia de dos es el resultado de un desplazamiento a la derecha. Para dividir por 2 solo haga un desplazamiento a la derecha; para dividir por 4 (2^2) desplace los bits 2 lugares; para dividir por 8 desplace 3 lugares a la derecha etc. Las instrucciones de desplazamiento son muy elementales y son *mucho* más rápidas que las instrucciones correspondientes `MUL` y `DIV` .

Los desplazamientos lógicos se pueden usar para multiplicar y dividir valores sin signo. Ellos no funcionan, en general, para valores con signo. Considere el valor de dos bytes FFFF (-1). Si éste se desplaza lógicamente a la derecha una vez ¡el resultado es 7FFF que es $+32,767$! Se pueden usar otro tipo de desplazamientos para valores con signo.

3.1.3. Desplazamientos aritméticos

Estos desplazamientos están diseñados para permitir que números con signo se puedan multiplicar y dividir rápidamente por potencias de 2. Ellos aseguran que el bit de signo se trate correctamente.

SAL (*Shift aritmetic left*). Esta instrucción es solo sinónimo para `SHL`. Se ensambla con el mismo código de máquina que SHL. Como el bit de signo no se cambia por el desplazamiento, el resultado será correcto. SAR

SAR (*Shift Arithmetic Right*). Esta es una instrucción nueva que no desplaza el bit de signo (el bit más significativo) de su operando. Los otros bits se desplazan como es normal excepto que los bits nuevos que entran por la derecha son copias del bit de signo (esto es, si el

bit de signo es 1, los nuevos bits son también 1). Así, si un byte se desplaza con esta instrucción, sólo los 7 bits inferiores se desplazan. Como las otras instrucciones de desplazamiento, el último bit que sale se almacena en la bandera de carry.

```
1  mov    ax, 0C123H
2  sal    ax, 1          ; ax = 8246H, CF = 1
3  sal    ax, 1          ; ax = 048CH, CF = 1
4  sar    ax, 2          ; ax = 0123H, CF = 0
```

3.1.4. Desplazamientos de rotación

Los desplazamientos de rotación trabajan como los desplazamientos lógicos excepto que los bits perdidos en un extremo del dato se desplazan al otro lado. Así, el dato es tratado como si fuera una estructura circular. Las dos rotaciones más simples son `ROL` y `ROR` , que hacen rotaciones a la izquierda y a la derecha respectivamente. Tal como los otros desplazamientos, estos desplazamientos dejan una copia del último bit rotado en la bandera de carry.

```
1  mov    ax, 0C123H
2  rol    ax, 1          ; ax = 8247H, CF = 1
3  rol    ax, 1          ; ax = 048FH, CF = 1
4  rol    ax, 1          ; ax = 091EH, CF = 0
5  ror    ax, 2          ; ax = 8247H, CF = 1
6  ror    ax, 1          ; ax = C123H, CF = 1
```

Hay dos instrucciones de rotación adicionales que desplazan los bits en el dato y la bandera de carry llamadas `RCL` y `RCR`. . Por ejemplo, si el registro `AX` rota con estas instrucciones, los 17 bits se desplazan y la bandera de carry se rota.

```
1  mov    ax, 0C123H
2  clc                   ; borra la bandera de carry (CF = 0)
3  rcl    ax, 1          ; ax = 8246H, CF = 1
4  rcl    ax, 1          ; ax = 048DH, CF = 1
5  rcl    ax, 1          ; ax = 091BH, CF = 0
6  rcr    ax, 2          ; ax = 8246H, CF = 1
7  rcr    ax, 1          ; ax = C123H, CF = 0
```

3.1.5. Aplicación simple

A continuación está un fragmento de código que cuenta el número de bits que están “encendidos” (1) en el registro EAX.

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

Cuadro 3.1: La operación AND

	1	0	1	0	1	0	1	0
AND	1	1	0	0	1	0	0	1
	1	0	0	0	1	0	0	0

Figura 3.2: ANDdo un byte

```
1       mov    bl, 0            ; bl contendrá el número de bits prendidos
2       mov    ecx, 32          ; ecx es el contador del bucle
3  count_loop:
4       shl    eax, 1           ; desplaza los bits en la bandera de carry
5       jnc    skip_inc         ; si CF == 0, va a skip_inc
6       inc    bl
7  skip_inc:
8       loop   count_loop
```

El código anterior destruye el valor original de `EAX` (`EAX` es cero al final del bucle). Si uno desea conservar el valor de EAX, la línea 4 debería ser reemplazada con `rol eax,1`.

3.2. Operaciones booleanas entre bits

Hay cuatro operadores booleanos básicos *AND*, *OR*, *XOR* y *NOT*. Una *tabla de verdad* muestra el resultado de cada operación por cada posible valor de los operandos.

3.2.1. La operación *AND*

El resultado del *AND* de dos bits es 1 solo si ambos bits son 1, si no el resultado es cero como muestra el Cuadro 3.1

Los procesadores tienen estas operaciones como instrucciones que actúan independientemente en todos los bits del dato en paralelo. Por ejemplo, si el contenido de `AL` y `BL` se les opera con *AND*, la operación se aplica a cada uno de los 8 pares de bits correspondientes en los dos registros como muestra la Figura 3.2. A continuación un código de ejemplo:

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

Cuadro 3.2: La operación OR

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Cuadro 3.3: La operación XOR

```
1        mov    ax, 0C123H
2        and    ax, 82F6H          ; ax = 8022H
```

3.2.2. La operación *OR*

El *O* inclusivo entre dos bits es 0 solo si ambos bits son 0, si no el resultado es 1 como se muestra en el Cuadro 3.2 . A continuación un código de ejemplo:

```
1        mov    ax, 0C123H
2        or     ax, 0E831H          ; ax = E933H
```

3.2.3. La operación *XOR*

El *O* exclusivo entre 2 bits es 0 si y solo si ambos bits son iguales, sino el resultado es 1 como muestra el Cuadro 3.3. Sigue un código de ejemplo:

```
1        mov    ax, 0C123H
2        xor    ax, 0E831H          ; ax = 2912H
```

3.2.4. La operación *NOT*

La operación *NOT* es *unaria* (actúa sobre un solo operando, no como las operaciones *binarias* como *AND*). El *NOT* de un bit es el valor opuesto del bit como se muestra en el Cuadro 3.4. Sigue un código de ejemplo:

```
1        mov    ax, 0C123H
2        not    ax                  ; ax = 3EDCH
```

X	NOT X
0	1
1	0

Cuadro 3.4: La operación NOT

Prende el bit i	*OR* el número con 2^i (es el número con únicamente el bit i-ésimo prendido)
Apaga el bit i	*AND* el número binario con sólo el bit i apagado . Este operando es a menudo llamado *máscara*
Complementa el bit i	*XOR* el número con 2^i

Cuadro 3.5: Usos de las operaciones booleanas

Observe que *NOT* halla el complemento a 1. A diferencia de las otras operaciones entre bits, la instrucción `NOT` no cambian ninguno de los bits en el registro `FLAGS`.

3.2.5. La instrucción `TEST`

La instrucción `TEST` realiza una operación *AND*, pero no almacena el resultado. Solo fija las banderas del registro `FLAGS` dependiendo del resultado obtenido (muy parecido a lo que hace la instrucción `CMP` con la resta que solo fija las banderas). Por ejemplo, si el resultado fuera cero, `ZF` se fijaría.

3.2.6. Usos de las operaciones con bits

Las operaciones con bits son muy útiles para manipular bits individuales sin modificar los otros bits. El Cuadro 3.5 muestra los 3 usos más comunes de estas operaciones. Sigue un ejemplo de cómo implementar estas ideas.

```
1  mov    ax, 0C123H
2  or     ax, 8           ; prende el bit 3,   ax = C12BH
3  and    ax, 0FFDFH      ; apaga el bit 5,  ax = C10BH
4  xor    ax, 8000H       ; invierte el bit 31,   ax = 410BH
5  or     ax, 0F00H       ; prende el nibble,  ax = 4F0BH
6  and    ax, 0FFF0H      ; apaga nibble, ax = 4F00H
7  xor    ax, 0F00FH      ; invierte nibbles,  ax = BF0FH
8  xor    ax, 0FFFFH      ; complemento a uno ,  ax = 40F0H
```

La operación *AND* se puede usar también para hallar el residuo de una división por una potencia de dos. Para encontrar el residuo de una división por 2^i, efectúa un AND entre el dividendo y una máscara igual a $2^i - 1, AND$

el número con una máscara igual a $2^i - 1$. Esta máscara contendrá unos desde el bit 0 hasta el bit $i - 1$. Son solo estos bits los que contienen el residuo. El resultado de la *AND* conservará estos bits y dejará cero los otros. A continuación un fragmento de código que encuentra el cociente y el residuo de la división de 100 por 16.

```
1  mov    eax, 100        ; 100 = 64H
2  mov    ebx, 0000000FH  ; mácara = 16 - 1 = 15 or F
3  and    ebx, eax        ; ebx = residuo = 4
```

Usando el registro `CL` es posible modificar arbitrariamente bits. El siguiente es un ejemplo que fija (prende) un bit arbitrario en `EAX`. El número del bit a prender se almacena en `BH`.

```
1  mov    cl, bh          ;
2  mov    ebx, 1
3  shl    ebx, cl         ; se desplaza a la derecha cl veces
4  or     eax, ebx        ; prende el bit
```

Apagar un bit es solo un poco más difícil.

```
1  mov    cl, bh          ;
2  mov    ebx, 1
3  shl    ebx, cl         ; se desplaza a la derecha cl veces
4  not    ebx             ; invierte los bits
5  and    eax, ebx        ; apaga el bit
```

El código para complementar un bit arbitrario es dejado como ejercicio al lector.

Es común ver esta instrucción en un programa 80x86.

```
xor    eax, eax         ; eax = 0
```

Un número XOR con sigo mismo, el resultado es siempre cero. Esta instrucción se usa porque su código de máquina es más pequeño que la instrucción MOV equivalente.

3.3. Evitando saltos condicionales

Los procesadores modernos usan técnicas muy sofisticadas para ejecutar el código tan rápido como sea posible. Una técnica común se conoce como *ejecución especulativa*. Esta técnica usa las capacidades de procesamiento paralelo de la CPU para ejecutar múltiples instrucciones a la vez. Las instrucciones condicionales tienen un problema con esta idea. El procesador, en general, no sabe si se realiza o no la ramificación.Si se efectúa, se ejecutará un conjunto de instrucciones diferentes que si no se efectúa (el salto).

```
1       mov    bl, 0           ; bl contendrá el número de bits prendidos
2       mov    ecx, 32         ; ecx es el bucle contador
3  count_loop:
4       shl    eax, 1          ; se desplaza el bit en la bandera de carry
5       adc    bl, 0           ; añade solo la bandera de carry a bl
6       loop   count_loop
```

Figura 3.3: Contando bits con `ADC`

El procesador trata de predecir si ocurrirá la ramificación o no. Si la predicción fue errónea el procesador ha perdido su tiempo ejecutando un código erróneo.

Una manera de evitar este problema, es evitar usar ramificaciones condicionales cuando es posible. El código de ejemplo en 3.1.5 muestra un programa muy simple donde uno podría hacer esto. En el ejemplo anterior, los bits "encendidos" del registro EAX se cuentan. Usa una ramificación para saltarse la instrucción `INC`. La figura 3.3 muestra cómo se puede quitar la ramificación usando la instrucción `ADC` para sumar el bit de carry directamente.

Las instrucciones `SET`*xx* suministran una manera de suprimir ramificaciones en ciertos casos. Esta instrucción fija el valor de un registro o un lugar de memoria de 8 bits a cero, basado en el estudio del registro FLAGS. Los caracteres luego de `SET` son los mismos caracteres usados en los saltos condicionales. Si la condición correspondiente de `SET`*xx* es verdadero, el resultado almacenado es uno, si es falso se almacena cero. Por ejemplo,

```
setz   al        ; AL = 1 if Z flag is set, else 0
```

Usando estas instrucciones, uno puede desarrollar algunas técnicas ingeniosas que calculan valores sin ramificaciones.

Por ejemplo, considere el problema de encontrar el mayor de dos valores. La aproximación normal para resolver este problema sería el uso de `CMP` y usar un salto condicional y proceder con el valor que fue más grande. El programa de ejemplo de abajo muestra cómo se puede encontrar el mayor sin ninguna ramificación.

```
1  ; Archivo: max.asm
2  %include "asm_io.inc"
3  segment .data
4
5  message1 db "Digite un número: ",0
```

```
6  message2 db "Digite otro número: ", 0
7  message3 db "El mayor número es: ", 0
8
9  segment .bss
10
11 input1  resd    1        ; primer número ingresado
12
13 segment .text
14         global  _asm_main
15 _asm_main:
16         enter   0,0              ;
17         pusha
18
19         mov     eax, message1    ; imprime el primer mensaje
20         call    print_string
21         call    read_int         ; ingresa el primer número
22         mov     [input1], eax
23
24         mov     eax, message2    ; imprime el segundo mensaje
25         call    print_string
26         call    read_int         ; ingresa el segundo número (en  eax)
27
28         xor     ebx, ebx         ; ebx = 0
29         cmp     eax, [input1]    ; compara el segundo y el primer número
30         setg    bl               ; ebx = (input2 > input1) ?          1 : 0
31         neg     ebx              ; ebx = (input2 > input1) ? 0xFFFFFFFF : 0
32         mov     ecx, ebx         ; ecx = (input2 > input1) ? 0xFFFFFFFF : 0
33         and     ecx, eax         ; ecx = (input2 > input1) ?     input2 : 0
34         not     ebx              ; ebx = (input2 > input1) ?          0 : 0xFFFFFFFF
35         and     ebx, [input1]    ; ebx = (input2 > input1) ?          0 : input1
36         or      ecx, ebx         ; ecx = (input2 > input1) ?     input2 : input1
37
38         mov     eax, message3    ; imprime los resultado
39         call    print_string
40         mov     eax, ecx
41         call    print_int
42         call    print_nl
43
44         popa
45         mov     eax, 0           ; retorna a C
46         leave
47         ret
```

El truco es crear una máscara de bits que se pueda usar para seleccionar el valor mayor. La instrucción `SETG` en la línea 30 fija BL a 1. Si la segunda entrada es mayor o 0 en otro caso. Esta no es la máscara deseada. Para crear la máscara de bits requerida la línea 31 usa la instrucción `NEG` en el registro EBX. (Observe que se borró EBX primero). Si EBX es 0 no hace nada; sin embargo si EBX es 1, el resultado es la representación en complemento a dos de -1 o 0xFFFFFFFF. Esta es la máscara que se necesita. El resto del código usa esta máscara para seleccionar la entrada correcta como e la mayor.

Un truco alternativo es usar la instrucción `DEC`. En el código de arriba, si NEG se reemplaza con un `DEC`, de nuevo el resultado será 0 o 0xFFFFFFFF. Sin embargo, los valores son invertidos que cuando se usa la instrucción `NEG`.

3.4. Manipulando bits en C

3.4.1. Las operacones entre bits de C

A diferencia de algunos lenguajes de alto nivel C suministra operadores para operaciones entre bits. La operación `AND` se representa con el operador `&`[1]. La operación *OR* se representa por el operador binario `|`. La operación *XOR* se representa con el operador binario `^` . Y la operación *NOT* se representa con el operador unario `~` .

Los desplazamientos son realizados por C con los operadores binarios `<<` y `>>` . El operador `<<` realiza desplazamientos a la izquierda y el operador `>>` hace desplazamientos a la derecha. Estos operadores toman 2 operandos. El de la derecha es el valor a desplazar y el de la izquierda es el número de bits a desplazar. Si el valor a desplazar es un tipo sin signo, se realiza un desplazamiento lógico. Si el valor es con signo (como `int`), entonces se usa un desplazamiento aritmético a continuación, un ejemplo en C del uso de estos operadores:

```
short int s;          /* se asume que short int es de 16 bits */
short unsigned u;
s = -1;               /* s = 0xFFFF (complemento a dos) */
u = 100;              /* u = 0x0064 */
u = u | 0x0100;       /* u = 0x0164 */
s = s & 0xFFF0;       /* s = 0xFFF0 */
s = s ^ u;            /* s = 0xFE94 */
u = u << 3;           /* u = 0x0B20 (desplazamiento lógico) */
s = s >> 2;           /* s = 0xFFA5 (desplazamiento aritmético) */
```

[1] ¡Este operador es diferente del operador binario && y del unario &!

Macro	Meaning
S_IRUSR	el propietario puede leer
S_IWUSR	el propietario puede escribir
S_IXUSR	el propietario puede ejecutar
S_IRGRP	el grupo de propietario puede leer
S_IWGRP	el grupo del propietario puede escribir
S_IXGRP	el grupo del propietario puede ejecutar
S_IROTH	los otros pueden leer
S_IWOTH	los otros pueden escribir
S_IXOTH	los otros pueden ejecutar

Cuadro 3.6: Macros POSIX para permisos de archivos

3.4.2. Usando las operaciones entre bits en C

Los operadores entre bits se usan en C para los mismos propósitos que en lenguaje ensamblador. Ellos le permiten a uno manipular bits individuales y se pueden usar para multiplicaciones y divisiones rápidas. De hecho, un compilador de C inteligente usará desplazamientos automáticamente para multiplicaciones como `X*=2`, Muchos API[2] (Como *POSIX* [3] y Win 32). tienen funciones que usan operandos que tienen datos codificados como bits. Por ejemplo, los sistemas POSIX mantienen los permisos de los archivos para 3 tipos diferentes de usuarios (un mejor nombre sería *propietario*), *grupo* y *otros*. A cada tipo de usuario se le puede conceder permisos para leer, escribir o ejecutar un archivo. Para cambiar los permisos de un archivo requiere que el programador de C manipule bits individuales. POSIX define varios macros para ayudar (vea el Cuadro 3.6). La función `chmod` se puede usar para establecer los permisos de un archivo. Esta función toma dos parámetros, una cadena con el nombre del archivo sobre el que se va a actuar y un entero[4] Con los bits apropiados para los permisos deseados. Por ejemplo, el código de abajo fija los permisos para permitir que el propietario del archivo leerlo y escribirlo, a los usuarios en e, grupo leer en archivo y que los otros no tengan acceso.

```
chmod("foo", S_IRUSR | S_IWUSR | S_IRGRP );
```

La función POSIX `stat` se puede usar para encontrar los bits de permisos actuales de un archivo. Usada con la función `chmod`, es posible modificar algunos de los permisos sin cambiar los otros. A continuación un ejemplo que quita el acceso de la escritura a los otros y añade el accesode lectura.

[2] Aplication Programming Interface

[3] Significa Portable Operatting System Interface for Computer Enviroments. Una norma desarrollada por el IEEE basado en UNIX.

[4] Actualmente un parámetro de tipo `mode_t` que es un typedef a un tipo integral.

Los otros permisos no son alterados Los otros permisos no se alteran.

```
struct stat   file_stats ;      /* estructura usada por stat() */
stat("foo", & file_stats );   /* lee la información del archivo.
                                 file_stats .st_mode holds permission bits */
chmod("foo", ( file_stats .st_mode & ~S_IWOTH) | S_IRUSR);
```

3.5. Representaciones Littel Endian y Big Endian

El Capítulo 1 introdujo el concepto de las representaciones big y little endian de datos multibyte. Sin embargo, el autor ha encontrad que este tema confunde a muchas personas. Esta sección cubre el tópico con más detalle.

El lector recordará que lo endian se refiere al orden en que los bytes individuales se almacenan en memoria (*no* bits) de un elemento multibyte se almacena en memoria. Big endian es el método más directo. Almacena el byte más significativo primero, luego el siguiente byte en peso y así sucesivamente. En otras palabras los bits de *más peso* se almacenan primero. Little endian almacena los bytes en el orden contrario (primero el menos significativo). La familia de procesadores X86 usa la representación little endian.

Como un ejemplo, considere la palabra doble 12345678_{16}. En la representación big endian, los bytes se almacenarían como 12 34 56 78. En la representación little endian los bytes se almacenarían como 78 56 34 12.

El lector probablemente se preguntará así mismo, ¿por qué cualquier diseñador sensato de circuitos integrados usaría la representación little endian? ¿Eran los ingenieros de Intel sádicos para infligir esta confusa representación a multitud de programadores? Parecería que la CPU no tiene que hacer trabajo extra para almacenar los bytes hacia atrás en la memoria (e invertirlos cuando los lee de la memoria). La respuesta es que la CPU no hace ningún trabajo extra para leer y escribir en la memoria usando el formato little endian. Uno sabe que la CPU está compuesta de muchos circuitos electrónicos que simplemente trabajan con bits. Los bits (y bytes) no están en un orden en particular en la CPU.

Considere el registro de 2 bytes `AX`. Se puede descomponer en registros de un byte (`AH` y `AL`). Hay circuitos en la CPU que mantienen los valores de `AH` y `AL`. Los circuitos no están en un orden particular en la CPU. Esto significa que los circuitos para `AH` no están antes o después que los circuitos para `AL`.Una instrucción mov que y el valor de AX en memoria el valor de Al y luego AH Quiere decir que, no es difícil para la CPU hacer que almacene `AH` primero.

El mismo argumento se aplica a los bits individuales dentro de un byte, no hay realmente ningún orden en los circuitos de la CPU (o la memoria). Sin

```
unsigned short word = 0x1234;  /* se asume sizeof(short) == 2 */
unsigned char * p = (unsigned char *) &word;

if ( p[0] == 0x12 )
   printf ("Máquina Big Endian\n");
else
   printf ("Máquina Little Endian\n");
```

Figura 3.4: Cómo determinar lo Endianness

embargo, ya que los bits individuales no se pueden direccionar directamente en la CPU o en la memoria, no hay manera de conocer que orden parece que conservaran internamente en la CPU.

El código en C en la Figura 3.4 muestra cómo se puede determinar lo endian de una CPU. El apuntador `p` trata la variable `word` como dos elementos de un arreglo de caracteres. Así, `p[0]` evalúa el primer byte de `word` en la memoria que depende de lo endian en la CPU.

3.5.1. Cuando tener cuidado con Little and Big Endian

Para la programación típica, lo endian de la CPU no es importante. La mayoría de las veces esto es importante cuando se transfiere datos binarios entre sistemas de cómputo diferente. Esto es ocurre normalmente usando un medio de datos físico (como un disco) o una red. Ya que el código ASCII es de 1 byte la característica endian no le es importante.

Ahora con los conjuntos de caracteres multibyte como UNICODE , lo endian es importante aún para texto. UNICODE soporta ambos tipos de representación y tiene un mecanismo para especificar cuál se está usando para representar los datos.

Todos los encabezados internos de TCP/IP almacena enteros en big Endian (llamado *orden de byte de la red*). Y las bibliotecas de TCP/IP suministran funciones de C para tratar la cuestión endian de una manera portátil. Por ejemplo la función `htonl()` convierte una palabra doble (o long int) del formato del *host* al de *red*. La función `ntohl()` hace la transformación inversa.[5] Para un sistema big endian, las dos funciones sólo retornan su entrada sin cambio alguno. Esto le permite a uno escribir programas de red que compilarán y se ejecutarán correctamente en cualquier sistema sin importar lo endian. Para más información sobre lo endian programación de redes vea el excelente libro de W. Richard Steven *UNIX Network Programming*.

La Figura 3.5 muestra una función de C que invierte lo endian de una palabra doble. El 486 ha introducido una nueva instrucción de máquina llamada `BSWAP` que invierte los bytes de cualquier registro de 32 bits. Por ejemplo:

[5] Ahora, invertir lo endian de un entero simplemente coloca al revés los bytes; así convertir de big a little o de little a big es la misma operación. Así, ambas funciones hacen la misma cosa.

```
unsigned invert_endian( unsigned x )
{
  unsigned invert;
  const unsigned char * xp = (const unsigned char *) &x;
  unsigned char * ip = (unsigned char *) & invert;

  ip[0] = xp[3];    /* invierte los bytes individuales */
  ip[1] = xp[2];
  ip[2] = xp[1];
  ip[3] = xp[0];

  return invert;    /* retorna los bytes invertidos */
}
```

Figura 3.5: Función invert_endian

```
bswap   edx          ; intercambia los bytes de edx
```

Esta instrucción no se puede usar en los registros de 16 bits, sin embargo la instrucción `XCHG` se puede usar para intercambiar los bytes en los registros de 16 bits que se pueden descomponer en registros de 8 bits. Por ejemplo:

```
xchg    ah,al        ; intercambia los bytes de ax
```

3.6. Contando bits

Al principio se dio una técnica directa para contar el número de bits que están "encendidos" en una palabra doble. Esta sección mira otros métodos menos directos de hacer esto, como un ejercicio que usa las operaciones de bits discutidas en este capítulo.

3.6.1. Método uno

El primer método es muy simple, pero no obvio. La figura 3.6 muestra el código.

¿Cómo trabaja este método? En cada iteración el bucle, se apaga un bit de data `dato`. Cuando todos los bits se han apagado (cuando el `dato` es cero), el bucle finaliza. El número de iteraciones requerido para hacer el `dato` cero es igual al número de bits en el valor original de data `data`.

La línea 6 es donde se apaga un bit del `dato`. ¿Cómo se hace esto? Considere la forma general de la representación en binario del `dato` y el 1 del extremo derecho de esta representación. Por definición cada bit después

```
int count_bits( unsigned int data )
{
  int cnt = 0;

  while( data != 0 ) {
    data = data & (data − 1);
    cnt++;
  }
  return cnt;
}
```

Figura 3.6: Contando bits: método uno

de este 1 debe ser cero. Ahora, ¿Cuál será la representación de `data -1`? Los bits a la izquierda del 1 del extremo derecho serán los mismos que para `data`, pero en el punto del 1 del extremo derecho ellos serán el complemento de los bits originales de `data`. Por ejemplo:

```
data       =  xxxxx10000
data - 1   =  xxxxx01111
```

donde X es igual para ambos números. Cuando se hace `data` *AND* `data -1`, el resultado será cero el 1 del extremo derecho en `data` y deja todos los otros bits sin cambio.

3.6.2. Método dos

Una búsqueda en una tabla se puede usar para contar bits de una palabra doble arbitraria. La aproximación directa sería precalcular el número de bits para cada palabra doble y almacenar esto en un arreglo. Sin embargo, hay dos problemas relacionados con esta aproximación. ¡Hay alrededor de *4 mil millones* de palabras dobles! Esto significa que el arreglo será muy grande e iniciarlo consumiría mucho tiempo. (de hecho, a menos que uno vaya a utilizar realmente el arreglo de más que 4 mil millones de veces, se tomará más tiempo iniciando el arreglo que el que se requeriría para calcular la cantidad de bits usando el método uno).

Un método más realista sería precalcular la cantidad de bits para todos los valores posibles de un byte y almacenar esto en un arreglo. Entonces la palabra doble se puede dividir en 4 bytes. Se hallan los b y se suman para encontrar la cantidad de bits de la palabra doble original. La figura 3.7 muestra la implementación de esta aproximación.

La función `initialize_count_bits` debe ser llamada antes, del primer llamado a la función `count_bits` . Esta función inicia el arreglo global `byte_bit_count`. La función `count_bits` mira la variable `data` no como una

```
static unsigned char byte_bit_count [256];   /* lookup table */

void   initialize_count_bits  ()
{
  int cnt, i, data;

  for( i = 0; i < 256; i++ ) {
    cnt = 0;
    data = i;
    while( data != 0 ) {          /* método uno */
      data = data & (data - 1);
      cnt++;
    }
    byte_bit_count [i] = cnt;
  }
}

int  count_bits ( unsigned int data )
{
  const unsigned char * byte = ( unsigned char *) & data;

  return  byte_bit_count [byte [0]]  +  byte_bit_count [byte [1]]  +
          byte_bit_count [byte [2]]  +  byte_bit_count [byte [3]];
}
```

Figura 3.7: Método dos

palabra doble, sino como un arreglo de 4 bytes. El apuntador `dword` actúa como un apuntador a este arreglo de 4 bytes. Así `dword [0]` es uno de los bytes en `data` (el byte menos significativo o el más significativo dependiendo si es little o big endian respectivamente). Claro está uno podría usar una instrucción como:

```
(data >> 24) & 0x000000FF
```

Para encontrar el byte más significativo y hacer algo parecido con los otros bytes; sin embargo, estas construcciones serán más lentas que una referencia al arreglo.

Un último punto, se podría usar fácilmente un bucle `for` para calcular la suma en las líneas 22 y 23. Pero el bucle `for` incluiría el trabajo extra de iniciar el índice del bucle, comparar el índice luego de cada iteración e incrementar el índice. Calcular la suma como la suma explícita de 4 valores será más rápido. De hecho un compilador inteligente podría convertir la

```
int count_bits(unsigned int x )
{
  static unsigned int mask[] = { 0x55555555,
                                 0x33333333,
                                 0x0F0F0F0F,
                                 0x00FF00FF,
                                 0x0000FFFF };
  int i;
  int shift;    /* número de posiciones a desplazarse a la derecha */

  for( i=0, shift=1; i < 5; i++, shift *= 2 )
    x = (x & mask[i]) + ( (x >> shift) & mask[i] );
  return x;
}
```

Figura 3.8: Método tres

versión del bucle `for` a la suma explícita. Este proceso de reducir o eliminar iteraciones de bucles es una técnica de optimización conocida como *loop unrolling*.

3.6.3. Método tres

Hay otro método ingenioso de contar bits que están en un dato. Este método literalmente añade los unos y ceros del dato unido. Esta suma debe ser igual al número de unos en el dato. Por ejemplo considere calcular los unos en un byte almacenado en una variable llamada `data`. El primer paso es hacer la siguiente operación:

```
data = (data & 0x55) + ((data >> 1) & 0x55);
```

¿Qué hace esto? La constante hexadecimal `0X55` es 01010101 en binario. En el primer operando de la suma `data` es *AND* con él, los bits en las posiciones pares se sacan. El Segundo operando (`data >> 1 & 0x55`), primero mueve todos los bits de posiciones pares a impares y usa la misma máscara para sacar estos mismos bits. Ahora, el primer operando contiene los bits pares y el segundo los bits impares de `data`. Cuando estos dos operandos se suman, se suman los bits pares e impares de `data`. Por ejemplo si data es 10110011_2, entonces:

$$\begin{array}{r} \texttt{data \& } 01010101_2 \\ +\ \texttt{(data >> 1) \& } 01010101_2 \\ \hline \end{array} \quad \text{or} \quad \begin{array}{r|cccc} & 00 & 01 & 00 & 01 \\ + & 01 & 01 & 00 & 01 \\ \hline & 01 & 10 & 00 & 10 \end{array}$$

La suma de la derecha muestra los bits sumados. Los bits del byte se dividen en 4 campos de 2 bits para mostrar que se realizan 4 sumas independientes. Ya que la mayoría de estas sumas pueden ser dos, no hay posibilidad de que la suma desborde este campo y dañe otro de los campos de la suma.

Claro está, el número total de bits no se ha calculado aún. Sin embargo la misma técnica que se usó arriba se puede usar para calcular el total en una serie de pasos similares. El siguiente paso podría ser:

`data = (data & 0x33) + ((data >> 2) & 0x33);`

Continuando con el ejemplo de arriba (recuerde que `data` es ahora 01100010_2):

`data & 00110011`$_2$		0010	0010
+ `(data >> 2) & 00110011`$_2$	or +	0001	0000
		0011	0010

Ahora hay 2 campos de 4 bits que se suman independientemente.

El próximo paso es sumar estas dos sumas unidas para conformar el resultado final:

`data = (data & 0x0F) + ((data >> 4) & 0x0F);`

Usando el ejemplo de arriba (con `data` igual a 00110010_2):

`data & 00001111`$_2$		00000010
+ `(data >> 4) & 00001111`$_2$	or +	00000011
		00000101

Ahora `data` es 5 que es el resultado correcto. La Figura 3.8 muestra una implementación de este método que cuenta los bits en una palabra doble. Usa un bucle `for` para calcular la suma. Podría ser más rápido deshacer el bucle; sin embargo, el bucle clarifica cómo el método generaliza a diferentes tamaños de datos.

Capítulo 4

Subprogramas

Este capítulo se concentra en el uso de subprogramas para hacer programas modulares e interfaces con programas de alto nivel (como C). Las funciones y los procedimientos son ejemplos, en lenguajes de alto nivel, de subprograma.

El código que llama el subprograma y el subprograma en sí mismo deben estar de acuerdo en cómo se pasarán los datos entre ellos. Estas reglas de cómo se pasarán el dato son llamadas *convenciones de llamado*. Una gran parte de este capítulo tratará de las *convenciones de llamado* estándares de C, que se pueden usar para interfasar subprogramas de ensamblador con programas de C. Estas (y otras convenciones) a menudo pasan direcciones de datos (apuntadores) para permitirle al subprograma acceder a datos en la memoria.

4.1. Direccionamiento indirecto

El direccionamiento indirecto le permite a los registros comportarse como variables apuntador. Para indicar que un registro se va a usar indirectamente como apuntador, se encierra entre paréntesis cuadrados (`[]`) por ejemplo:

```
1  mov    ax, [Data]     ; Direccionamiento directo de memoria de una palabra
2  mov    ebx, Data      ; ebx = & Data
3  mov    ax, [ebx]      ; ax = *ebx
```

Debido a que AX almacena una palabra, la línea 3 lee una palabra comenzando en la dirección almacenada en EBX. Si AX fuera reemplazando con AL, se leería un solo byte. Es importante notar que los registros no tienen tipos como lo hacen las variables en C. A lo que EBX se asume que señala está totalmente determinada por qué instrucciones se usan. Si EBX se utiliza incorrectamente, a menudo no habrá error en el ensamblador; sin embargo, el programa no trabajará correctamente. Esta es una de las muchas razones

por la cual el ensamblador es más propenso a errores que los lenguajes de alto nivel.

Todos los registros de 32 bits de propósito general (EAX, EBX, ECX, EDX) y los registros de índice (ESI y EDI) se pueden usar para el direccionamiento indirecto. En general los registros de 8 y 16 bits no.

4.2. Sencillo subprograma de ejemplo

Un subprograma es una unidad independiente de código que puede ser usada desde diferentes partes de un programa. En otras palabras, un subprograma es como una función en C. Se puede usar un salto para invocar el subprograma, pero el retorno representa un problema. Si el subprograma es usado en diferentes partes del programa debe retornar a la parte del código desde la que se la invocó. Por lo tanto, el salto de retorno desde el subprograma no puede ser a una etiqueta. El código siguiente muestra cómo se puede realizar esto usando una forma indirecta de la instrucción `JMP`. Esta forma de la instrucción JMP usa el valor de un registro para determinar a dónde saltar (así, el registro se comporta muy parecido a un *apuntador a una función* en C). A continuación, está el primer programa del Capítulo 1 reescrito para usarlo como subprograma.

sub1.asm

```
1  ; file: sub1.asm
2  ; Subprograma programa de ejemplo
3  %include "asm_io.inc"
4
5  segment .data
6  prompt1 db    "Ingrese un número: ", 0       ; no olvide el NULL
7  prompt2 db    "Ingrese otro número: ", 0
8  outmsg1 db    "Ud. ha ingresado ", 0
9  outmsg2 db    " y ", 0
10 outmsg3 db    ", la suma de ellos es ", 0
11
12 segment .bss
13 input1  resd 1
14 input2  resd 1
15
16 segment .text
17         global  _asm_main
18 _asm_main:
19         enter   0,0               ; setup routine
20         pusha
21
```

```
22          mov     eax, prompt1      ; imprime el prompt
23          call    print_string
24
25          mov     ebx, input1       ; almacena la dirección de input1 en ebx
26          mov     ecx, ret1         ; almacena la dirección de retorno en ecx
27          jmp     short get_int     ; lee un entero
28  ret1:
29          mov     eax, prompt2      ; imprime el prompt
30          call    print_string
31
32          mov     ebx, input2
33          mov     ecx, \$ + 7        ; ecx = esta dirección + 7
34          jmp     short get_int
35
36          mov     eax, [input1]     ; eax = palabra doble en  input1
37          add     eax, [input2]     ; eax += palabra doble en  input2
38          mov     ebx, eax          ; ebx = eax
39
40          mov     eax, outmsg1
41          call    print_string      ; imprime el primer mensaje
42          mov     eax, [input1]
43          call    print_int         ; imprime input1
44          mov     eax, outmsg2
45          call    print_string      ; imprime el segundo mensaje
46          mov     eax, [input2]
47          call    print_int         ; imprime input2
48          mov     eax, outmsg3
49          call    print_string      ; imprime el tercer mensaje
50          mov     eax, ebx
51          call    print_int         ; imprime sum (ebx)
52          call    print_nl          ; imprime nueva línea
53
54          popa
55          mov     eax, 0            ; retorno a C
56          leave
57          ret
58  ; subprograma get_int
59  ; Parámetros:
60  ;   ebx - dirección de la palabra doble que almacena el entero
61  ;   ecx - dirección de la instrucción a donde retornar
62  ; Notes:
63  ;   el valor de eax se destruye
```

```
64 get_int:
65         call    read_int
66         mov     [ebx], eax          ; almacena la entrada en memoria
67         jmp     ecx                 ; salta al llamador
```
sub1.asm

El subprograma `get_int` usa una convención de llamado simple basada en un registro. Ella espera que el registro EBX almacene la dirección de la palabra doble del número de entrada y que el registro ECX a almacene el código de la instrucción a saltar. En las líneas 25 a 28, el operador `$` se usa para calcular la dirección de retorno. El operador `$` retorna la dirección de la línea en que aparece. La expresión `$ + 7` calcula la dirección de la instrucción `MOV` de la línea 36.

Los dos cálculos de la dirección de retorno son complicados. El primer método requiere que una etiqueta se defina en cada llamado a subprograma. El segundo método no requiere una etiqueta, pero requiere un tratamiento cuidadoso. Si se usó un salto largo en lugar de uno corto ¡el número a añadirle a `$` podría no ser 7! Afortunadamente hay una manera mucho más simple de invocar subprogramas. Este método usa la *pila*.

4.3. La pila

Muchas CPU tienen soporte para una pila. Una pila es una lista *LIFO* (Last In Firist Out). La pila es un arca de memoria que está organizada de esta manera. La instrucción `PUSH` añade datos a la pila y la instrucción `POP` quita datos. El dato extraído siempre es el último dato insertado (esta es la razón por la cual es llamado FIFO).

El registro de segmento SS especifica el segmento de datos que contiene la pila. (Normalmente este es el mismo segmento de datos). El registro ESP contiene la dirección del dato que sería quitado de la pila. Los datos sólo se pueden añadir en unidades de palabras dobles. Esto es, que no se puede insertar un solo byte en la pila.

La instrucción PUSH inserta una palabra doble[1] en la pila restándole 4 a ESP y entonces almacena la palabra doble en `[ESP]`. La instrucción `POP` lee la palabra doble almacenada en `[ESP]` y luego añade 4 a ESP. El código siguiente demuestra cómo trabajan estas instrucciones asumiendo que el valor inicial de ESP es `1000H`.

```
1  push   dword 1    ; 1 almacendao en  0FFCh, ESP = 0FFCh
2  push   dword 2    ; 2 almacenado en 0FF8h, ESP = 0FF8h
3  push   dword 3    ; 3 almacenado en  0FF4h, ESP = 0FF4h
```

[1]También se pueden empujar palabras, pero en el modo protegido de 32 bits es mejor trabajar sólo con palabras dobles en la pila.

```
4       pop     eax         ; EAX = 3, ESP = 0FF8h
5       pop     ebx         ; EBX = 2, ESP = 0FFCh
6       pop     ecx         ; ECX = 1, ESP = 1000h
```

La pila se puede usar como un almacén de datos temporal muy conveniente. También se usa para el llamado a subprogramas, pasando parámetros y variables locales.

El 80x86 también suministra la instrucción `PSHA` que empuja el valor de los registros: EAX, EBX, ECX, EDX, ESI, EOI y EBP (no en este orden). La instrucción `POPA` se puede usar para devolver todos estos registros a su valor anterior.

4.4. Las instrucciones CALL y RET

El 80x86 suministra dos instrucciones que usa la pila para hacer llamados a subprogramas rápido y fácil. La instrucción CALL hace un salto incondicional a un subprograma y *empuja* en la pila la dirección de la próxima instrucción. La instrucción RET *saca* una dirección de la pila y salta a esta dirección. Cuando se usa esta instrucción, es muy importante que uno administre la pila correctamente ya que la instrucción RET debe extraer de la pila el número correcto.

El programa anterior se puede reescribir usando estas nuevas instrucciones cambiando las líneas 25 a 34 por:

```
        mov     ebx, input1
        call    get_int

        mov     ebx, input2
        call    get_int
```

y cambiando el subprograma `get_int` a:

```
get_int:
        call    read_int
        mov     [ebx], eax
        ret
```

Hay varias ventajas de CALL y RET

- Es simple
- Permite a los subprogramas hacer llamados anidados fácilmente. Observe que `get_int` llama `read_int`. Esta llamada empuja otra dirección

en la pila. Al final del código de `red_int` hay un RET que saca la dirección de retorno y que salta de nuevo al código de `get_int`. Cuando la instrucción RET de `get_int` se ejecuta, saca la dirección de retorno que salta de nuevo a `asm_main`. Esto trabaja correctamente por la propiedad LIFO de la pila.

Recuerde es *muy* importante sacar todos los datos que se han empujado en la pila. Por ejemplo considere lo siguiente:

```
1 get_int:
2       call   read_int
3       mov    [ebx], eax
4       push   eax
5       ret                  ; ¡¡saca el valor de EAX,
6                            ; no la dirección de retorno!!
```

Este código no retornaría correctamente.

4.5. Convenciones de llamado

Cuando un subprograma se invoca, el código llamado y el subprograma (el *llamador*) deben estar de acuerdo en cómo se pasan datos entre ellos. Los lenguajes de alto nivel tienen modos normalizados de pasarse datos, conocidas como *convenciones de llamado*. Para interfasar código de alto nivel con lenguaje ensamblador, éste debe usar las mismas convenciones que el lenguaje de alto nivel. Las convenciones de llamado pueden diferir de compilador a compilador o pueden variar dependiendo de cómo se compila el código (si se ha optimizado o no). Una convención universal es que el código será invocado con la instrucción `CALL` y retornará con `RET`.

Todos los compiladores de C para PC soportan una convención de llamado que será descrita en el resto del capítulo por etapas. Estas convenciones le permiten a uno crear subprogramas que sean *reentrantes*. Un subprograma reentrante puede ser llamado en cualquier punto del programa con seguridad (aún dentro del subprograma mismo).

4.5.1. Pasando parámetros en la pila

Los parámetros a un subprograma se pueden pasar en la pila. Ellos se empujan en la pila antes de la instrucción `CALL`. Tal como en C, si el parámetro es cambiado por el subprograma se debe pasar la *dirección* del dato no su *valor*. Si el tamaño del parámetro es menor que una palabra doble, se debe convertir a palabra doble antes de ser empujado en la pila.

Los parámetros no son sacados de la pila por el subprograma, en lugar de ello son accedidos desde la pila misma.¿Por qué?

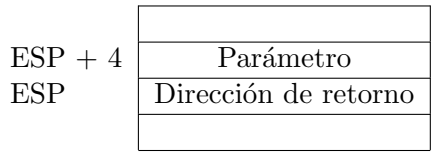


Figura 4.1:

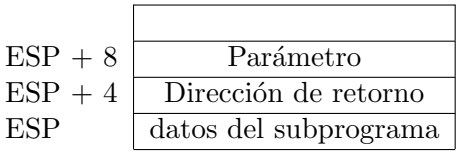


Figura 4.2:

- Ya que ellos se han empujado a la pila antes de la instrucción `CALL`, la dirección de retorno tendría que haberse sacado primero (y luego metido otra vez).

- A menudo los parámetros tendrán que usarse en varios lugares en el subprograma. Normalmente, ellos no se pueden dejar en un registro durante todo el subprograma y tendría que almacenarse en memoria. Dejándolos en la pila tenemos una copia del dato en memoria que se puede acceder en cualquier parte del subprograma.

Considere un subprograma al que se le pasa un solo parámetro en la pila. Cuando el subprograma se invoca, la pila se ve como en la Figura 4.1. Se puede acceder al parámetro usando direccionamiento indirecto (`[ESP+4]`)[2].

Si la pila se usa dentro del subprograma para almacenar datos, el número necesario a ser agregado a ESP cambiará. Por ejemplo, la Figura 4.2 muestra cómo se ve la pila si una palabra doble se empuja en ella. Ahora el parámetro es `ESP + 8` y no `ESP + 4`. Así, esto puede ser muy propenso a errores usar ESP cuando uno se refiere a parámetros. Para resolver este problema, el 80386 suministra otro registro: EBP. El único propósito de este registro es referenciar datos en la pila. La convención de llamado de C ordena que un subprograma primero guarde el valor de EBP en la pila y luego lo haga igual a ESP. Esto le permite a ESP cambiar cuando los datos se empujen o se saquen de la pila sin modificar EBP. Al final del subprograma, se debe restaurar el valor de EBP (esta es la razón por la cual se guarda el valor al principio del subprograma). La Figura 4.3 muestra la forma general de un subprograma que sigue estas convenciones.

Cuando se usa direccionamiento indirecto, el procesador 80x86 accede a segmentos diferentes dependiendo de qué registros se usan en la expresión de direccionamiento indirecto. ESP (y EBP) usan el segmento de la pila mientras que EAX, EBX, ECX y EDX usan el segmento de datos. Sin embargo, esto normalmente no tiene importancia para la mayoría de los programas en modo protegido, porque para ellos los segmentos de datos y de la pila son los mismos.

[2]Es válido añadir una constante a un registro cuando se usa direccionamiento indirecto. Se pueden construir expresiones más complicadas también. Este tópico se verá en el capítulo siguiente.

```
1 subprogram_label:
2       push   ebp      ; guarda el valor original de EBP en la pila
3       mov    ebp, esp ; nuevo EBP = ESP
4 ; subprogram code
5       pop    ebp      ; restaura el valor original de EBP
6       ret
```

Figura 4.3: Forma general de un subprograma

ESP + 8	EBP + 8	Parámetro
ESP + 4	EBP + 4	Dirección de retorno
ESP	EBP	EBP guardado

Figura 4.4:

Las líneas 2 y 3 de la Figura 4.3 componen el *prólogo* general de un subprograma. Las líneas 5 y 6 conforman el *epílogo*. La figura 4.4 muestra cómo se ve la pila inmediatamente después del prólogo. Ahora los parámetros se pueden acceder con `[EBP + 8]` en cualquier lugar del subprograma sin importar qué haya empujado en la pila el subprograma.

Luego que el subprograma culmina, los parámetros que se empujan en la pila se deben quitar. La convención de llamado de C especifica que el código llamador debe hacer esto. Otras convenciones son diferentes. Por ejemplo la convención de llamado de Pascal especifica que el subprograma debe quitar los parámetros de la pila (hay otra forma de la instrucción RET que hace esto fácil). Algunos compiladores de C soportan esta convención también. El identificador `pascal` es usado en la definición del prototipo de la función para decirle al compilador que emplee esta convención. De hecho, la convención `stdcall` que usan las funciones de C del API de MS Windows trabajan de esta forma. ¿Cuál es la ventaja de este modo? Es un poco más eficiente que la convención de llamado de C. Entonces, ¿Por qué todas las funciones no usan esta convención? En general C le permite a una función tener un número variable de argumentos (`printf` y `scanf` son ejemplos). Para este tipo de funciones, la operación de quitar los parámetros de la pila variará de un llamado a otro. La convención de C permite realizar esta operación fácilmente de un llamado a otro. Las convenciones de Pascal y stdcall hacen esta operación muy difícil. Así, la convención de Pascal (como el lenguaje Pascal) no permite este tipo de funciones. MS Windows puede usar esta convención ya que ninguna de las funciones del API toma un número variable de argumentos.

```
1       push   dword 1        ; pasa 1 como parámetro
2       call   fun
3       add    esp, 4         ; quita el parámetro de la pila
```

Figura 4.5: Muestra del llamado a un subprograma

La Figura 4.5 muestra como sería invocado un subprograma usando la convención de llamado de C. La línea 3 quita los parámetros de la pila manipulando directamente el apuntador de la pila. Una instrucción `POP` se podría usar para hacer esto pero requeriría que el resultado inútil se almacene en un registro. Actualmente, para este caso en particular muchos compiladores podrían usar una instrucción `POP ECX` para quitar el parámetro. El compilador usaría `POP` en lugar de `ADD` porque `ADD` requiere más bytes para la instrucción. Sin embargo, `POP` también altera el valor de ECX. A continuación está otro programa de ejemplo con dos subprogramas que usan la convención de llamado de C discutida arriba. La línea 54 (y otras líneas) muestran que se pueden declarar varios segmentos de datos y texto en un solo archivo fuente. Ellos serán combinados en un solo segmento de texto y datos en el proceso de encadenamiento. Dividir el código y los datos en segmentos separados permite que los datos de un subprograma se definan cerca del código del subprograma.

sub3.asm

```
1  %include "asm_io.inc"
2
3  segment .data
4  sum     dd   0
5
6  segment .bss
7  input   resd 1
8
9  ;
10 ; pseudo-código
11 ; i = 1;
12 ; sum = 0;
13 ; while( get_int(i, &input), input != 0 ) {
14 ;   sum += input;
15 ;   i++;
16 ; }
17 ; print_sum(num);
18 segment .text
```

```
19          global  _asm_main
20  _asm_main:
21          enter   0,0               ; setup routine
22          pusha
23
24          mov     edx, 1            ; edx es 'i' en el pseudocódigo
25  while_loop:
26          push    edx               ; guarda 'i' en la pila
27          push    dword input       ; empuja la dirección de input en la pila
28          call    get_int
29          add     esp, 8            ; quita i e &input de la pila
30
31          mov     eax, [input]
32          cmp     eax, 0
33          je      end_while
34
35          add     [sum], eax        ; sum += input
36
37          inc     edx
38          jmp     short while_loop
39
40  end_while:
41          push    dword [sum]       ; empuja el valor de sum  en la pila
42          call    print_sum
43          pop     ecx               ; quita [sum] de la pila
44
45          popa
46          leave
47          ret
48
49  ; subprograma get_int
50  ; Paramétros (en el orden que es empujan en la pila)
51  ;   número de input (en [ebp + 12])
52  ;   dirección de input en [ebp + 8])
53  ; Notas:
54  ;   Los valores de eax y ebx se destruyen
55  segment .data
56  prompt  db      ") Ingrese un entero (0 para salir): ", 0
57
58  segment .text
59  get_int:
60          push    ebp
```

```
61          mov     ebp, esp
62
63          mov     eax, [ebp + 12]
64          call    print_int
65
66          mov     eax, prompt
67          call    print_string
68
69          call    read_int
70          mov     ebx, [ebp + 8]
71          mov     [ebx], eax          ; almacena input en memoria
72
73          pop     ebp
74          ret                         ; retorna al llamador
75
76  ; subprograma print_sum
77  ; imprime la suma
78  ; Parameter:
79  ;   suma a imprimir (en [ebp+8])
80  ; Nota: destruye el valor de eax
81  ;
82  segment .data
83  result  db      "La suma es ", 0
84
85  segment .text
86  print_sum:
87          push    ebp
88          mov     ebp, esp
89
90          mov     eax, result
91          call    print_string
92
93          mov     eax, [ebp+8]
94          call    print_int
95          call    print_nl
96
97          pop     ebp
98          ret
```

sub3.asm

```
1  subprogram_label:
2        push   ebp                ; guarda el valor original de EBP
3                                  ; en la pila
4        mov    ebp, esp           ; nuevo EBP = ESP
5        sub    esp, LOCAL_BYTES   ; = # de bytes necesitados por las
6                                  ; variables locales
7  ; subprogram code
8        mov    esp, ebp           ; libera las variables locales
9        pop    ebp                ; restaura el valor original de EBP
10       ret
```

Figura 4.6: Forma general de un subprograma con variables locales

```
void calc_sum( int n, int * sump )
{
  int i, sum = 0;

  for( i=1; i <= n; i++ )
    sum += i;
  *sump = sum;
}
```

Figura 4.7: versión de C de sum

4.5.2. Variables locales en la pila

La pila se puede usar como un lugar adecuado para las variables locales. Ahí es exactamente donde C almacena las variables normales (o *automatic* como se dice en C). Usar la pila para las variables es importante si uno desea que el programa sea reentrante. Un programa reentrante trabajará si es invocado en cualquier lugar, incluido en subprograma en sí mismo. En otras palabras, los programas reentrantes pueden ser invocados *recursivamente*. Usar la pila para las variables también ahorra memoria. Los datos no almacenados en la pila están usando la memoria desde el comienzo hasta el final del programa (C llama este tipo de variable *global* o *static*). Los datos almacenados en la pila solo usan la memoria cuando el subprograma que los define está activo.

Las variables locales son almacenadas justo después que se guardó el valor EBP en la pila. Ellas son colocadas restando el número de bytes requeridos de ESP en el prólogo del subprograma. La Figura 4.6 muestra el nuevo esqueleto del subprograma. El registro EBP se usa para acceder a las variables locales.

```
1  cal_sum:
2        push   ebp
3        mov    ebp, esp
4        sub    esp, 4               ; hace espacio para la sum local
5
6        mov    dword [ebp - 4], 0   ; sum = 0
7        mov    ebx, 1               ; ebx (i) = 1
8  for_loop:
9        cmp    ebx, [ebp+8]         ; es i <= n?
10       jnle   end_for
11
12       add    [ebp-4], ebx         ; sum += i
13       inc    ebx
14       jmp    short for_loop
15
16 end_for:
17       mov    ebx, [ebp+12]        ; ebx = sump
18       mov    eax, [ebp-4]         ; eax = sum
19       mov    [ebx], eax           ; *sump = sum;
20
21       mov    esp, ebp
22       pop    ebp
23       ret
```

Figura 4.8: Versión en ensamblador de sum

Considere la función de C en la Figura 4.7. La Figura 4.8 muestra cómo se podría escribir un programa equivalente en ensamblador.

La Figura 4.9 muestra como se ve la pila luego del prólogo del programa en la Figura 4.8. Esta parte de la pila que contiene la información de retorno, los parámetros y las variables locales es llamado *marco de pila* (*stack frame*). Cada invocación a una función de C crea un nuevo marco de la pila en la pila.

El prólogo y el epílogo de un subprograma se pueden simplificar usando dos instrucciones especiales que están diseñadas específicamente para este propósito. La instrucción `ENTER` ejecuta el código del prólogo y `LEAVE` ejecuta el epílogo. La instrucción `ENTER` toma dos operandos inmediatos. Para la convención de llamado de C, el segundo operando es siempre 0. El primer operando es el número de bytes necesarios para las variables locales. La instrucción `LEAVE` no tiene operandos. La Figura 4.10 muestra cómo se usan

A pesar del hecho que `ENTER` *y* `LEAVE` *simplifican el prólogo y el epílogo ellos no se usan muy a menudo. ¿Por qué? Porque ellas son más lentas que instrucciones equivalentes más simples. Este es un ejemplo de cuando uno no puede asumir que una sola instrucción es más rápida que una secuencia de instrucciones.*

ESP + 16	EBP + 12	`sump`
ESP + 12	EBP + 8	`n`
ESP + 8	EBP + 4	Dirección de retorno
ESP + 4	EBP	EBP guardado
ESP	EBP - 4	`sum`

Figura 4.9:

```
1  subprogram_label:
2        enter  LOCAL_BYTES, 0     ; = número de bytes necesitados
3                                  ; por las variables locales
4  ; subprogram code
5        leave
6        ret
```

Figura 4.10: Forma general de un subprograma con variables locales usando `ENTER` and `LEAVE`

estas instrucciones. Observe que el programa esqueleto (Figura 1.7) también usa `ENTER` y `LEAVE`.

4.6. Programas Multinmódulo

Un *programa multimódulo* es uno que está compuesto de más de un archivo objeto. Todos los programas presentados acá han sido multimódulo. Ellos están compuestos del archivo objeto driver y el archivo objeto de ensamblador (más los archivos objeto de las bibliotecas de C). Recuerde que el encadenador combina los archivos objeto en un solo programa ejecutable. El encadenador debe emparejar las referencias hechas para cada etiqueta en un módulo (archivo objeto) con su definición en otro módulo. Para que el módulo A use la etiqueta definida en el módulo B, se debe usar la directiva `extern`. Luego de la directiva `extern` viene una lista de etiquetas separadas por comas. La directiva le dice al ensamblador que trate esas etiquetas como *externas* al módulo. O sea, esas son etiquetas que se pueden usar en este módulo pero están definidas en otro. El archivo `asm_io.inc` define las rutinas `read_int`, etc. como externas.

En ensamblador, no se puede acceder externamente a las etiquetas por omisión. Si una etiqueta puede ser accedida desde otros módulos diferentes al cual se definió, debe declararse *global* en su módulo. La directiva `global` hace esto, la línea 13 del programa esqueleto listado en la Figura 1.7 muestra que la etiqueta `_asm_main` está definida como global. Sin esta declaración, el

encadenador debería generar un error. ¿Por qué? Porque el código de C no podría hacer referencia a la etiqueta *interna* `_asm_main`.

A continuación está el código del ejemplo anterior, reescrito para usar dos módulos. Los dos subprogramas (`get_int` y `print_sum`) están en archivos fuentes diferentes que la rutina `_asm_main`.

main4.asm

```
1  %include "asm_io.inc"
2
3  segment .data
4  sum     dd   0
5
6  segment .bss
7  input   resd 1
8
9  segment .text
10         global  _asm_main
11         extern  get_int, print_sum
12 _asm_main:
13         enter   0,0               ; setup routine
14         pusha
15
16         mov     edx, 1            ; edx es 'i' en el pseudocódigo
17 while_loop:
18         push    edx               ; guarda i en la pila
19         push    dword input       ; empuja la dirección input en la pila
20         call    get_int
21         add     esp, 8            ; quita i e &input de la pila
22
23         mov     eax, [input]
24         cmp     eax, 0
25         je      end_while
26
27         add     [sum], eax        ; sum += input
28
29         inc     edx
30         jmp     short while_loop
31
32 end_while:
33         push    dword [sum]       ; empuja el valor de sum de la pila
34         call    print_sum
35         pop     ecx               ; quita sum de la pila
36
```

```
37          popa
38          leave
39          ret
```

main4.asm

sub4.asm

```
1   %include "asm_io.inc"
2
3   segment .data
4   prompt  db      ") Ingrese un número entero (0 para salir): ", 0
5
6   segment .text
7           global  get_int, print_sum
8   get_int:
9           enter   0,0
10
11          mov     eax, [ebp + 12]
12          call    print_int
13
14          mov     eax, prompt
15          call    print_string
16
17          call    read_int
18          mov     ebx, [ebp + 8]
19          mov     [ebx], eax          ; almacena input en memoria
20
21          leave
22          ret                         ; retorna
23
24  segment .data
25  result  db      "La suma es ", 0
26
27  segment .text
28  print_sum:
29          enter   0,0
30
31          mov     eax, result
32          call    print_string
33
34          mov     eax, [ebp+8]
35          call    print_int
36          call    print_nl
37
38          leave
```

```
39          ret
```
sub4.asm

El ejemplo anterior solo tiene etiquetas de código global sin embargo las etiquetas de datos global trabajan exactamente de la misma manera.

4.7. Interfazando ensamblador con C

Hoy día, pocos programas están escritos completamente en ensamblador. Los compiladores son muy buenos en convertir código de alto nivel en un código de máquina eficiente. Ya que es mucho más fácil escribir código en un lenguaje de alto nivel, es más popular. Además, el código de alto nivel es *mucho* más portátil que el ensamblador.

Cuando se usa ensamblador, se usa a menudo solo para pequeñas partes de código. Esto se puede hacer de dos maneras: llamando rutinas de ensamblador desde C o ensamblado en línea. El ensamblado en línea le permite al programador colocar instrucciones de ensamblador directamente en el código de C. Esto puede ser muy conveniente; sin embargo hay desventajas del ensamblado en línea. El código en ensamblador se debe escribir en el formato que usa el compilador. No hay compilador que en el momento soporte el formato de NASM. Los diferentes compiladores requieren diferentes formatos. Borland y Microsoft requieren el formato NASM . DJGPP y el gcc de Linux requieren el formato GAS[3]. La técnica de llamar una rutina en ensamblador está mucho más generalizada en el PC.

Las rutinas de ensamblador comúnmente se usan con C por las siguientes razones:

- Se necesita acceso directo a características del hardware del computador que es imposible o difícil acceder desde C.
- La rutina debe ser lo más rápida posible y el programador puede optimizar a mano el código mejor que el compilador

La última razón no es tan válida como una vez lo fue. La tecnología de los compiladores se ha mejorado con los años y a menudo generan un código muy eficiente. (Especialmente si se activan las optimizaciones del compilador). Las desventajas de las rutinas en ensamblador son: portabilidad reducida y lo poca legibilidad.

La mayoría de las convenciones de llamado ya se han especificado. Sin embargo hay algunas características adicionales que necesitan ser descritas.

[3]GAS es el ensamblador que usan todos los compiladores GNV. Usa la sintaxis AT&T que es muy diferente de la sintaxis relativamente similares de MASM, TASM y NASM.

```
1  segment .data
2  x            dd     0
3  format       db     "x = %d\n", 0
4
5  segment .text
6  ...
7        push   dword [x]     ; empuja el valor de x
8        push   dword format  ; empuja la dirección de la cadena
9                             ; con formato
10       call   _printf       ; observe el guión bajo
11       add    esp, 8        ; quita los parámetros de la pila
```

Figura 4.11: Llamado a `printf`

EBP + 12	valor de `x`
EBP + 8	dirección de la cadena con formato
EBP + 4	Dirección de retorno
EBP	EBP guardado

Figura 4.12: Pila dentro de `printf`

4.7.1. Ahorrando registros

La palabra reservada `register` *se puede usar en una declaración de una variable de C para sugerirle al compilador que use un registro para esta variable en vez de un lugar en memoria. Ellas se conocen como variables register. Los compiladores modernos hacen esto automáticamente sin requerir ninguna sugerencia.*

Primero, C asume que una subrutina conserva los valores de los siguientes registros: EBX, ESI, EDI, EBP, CS, DS, SS, ES. Esto no significa que la subrutina no pueda cambiarlos internamente. En vez de ello, significa que si se hace un cambio en sus valores, deben restablecer los valores originales antes que la subrutina retorne. Los valores en EBX, ESI y EDI no se deben modificar porque C usa esos registros para *variables register*. Normalmente la pila se usa para guardar los valores originales de estos registros.

4.7.2. Etiquetas de funciones

La mayoría de compiladores anteponen un guión bajo (_) al inicio de los nombres de funciones y variables globales o static. Por ejemplo una función llamada `f` se le asignará la etiqueta `_f`. Así, si ésta es una rutina en ensamblador se debe llamar `_f` no `f`. El compilador gcc de Linux *no* antepone ningún carácter. Bajo los ejecutables ELF de Linux, uno simplemente usaría la etiqueta `f` para la función de C `f`. Sin embargo el gcc de DJGPP antepone un guión bajo. Observe que en el programa esqueleto de ensamblador (Figura 1.7) la etiqueta de la rutina principal es `_asm_main`.

4.7.3. Pasando parámetros

Bajo la convención de llamado de C, los argumentos de una función se empujan en la pila en el orden *inverso* que aparecen en el llamado a la función.

Considere la siguiente instrucción en C: `printf ("x=%d\n",x);` la Figura 4.11 muestra como se compilaría esto (mostrado en el formato equivalente de NASM). La Figura 4.12 muestra cómo se ve la pila luego del prólogo dentro de la función `printf`. La función `printf` es una de las funciones de la biblioteca de C que puede tomar cualquier número de argumentos. Las reglas de la convención de llamado de C fueron escritas específicamente para permitir este tipo de funciones. Ya que la dirección de la cadena con formato se empuja de último, éste lugar en la pila será *siempre* `EBP + 8` no importa cuantos parámetros se le pasan a la función. El código `printf` puede ver en la cadena con formato cuántos parámetros se le debieron haber pasado y verlos en la en la pila.

No es necesario usar ensamblador para procesar un número arbitrario de parámetros en C. El archivo de cabecera `stdarg.h` *define marcos que se pueden usar para procesarlos con portabilidad. Vea cualquier buen libro de C para los detalles*

Claro está, si se comete un error, `printf ("x=%d\n");` el código de `printf` esperará imprimir una palabra doble en `[EBP+12]`. Sin embargo este no será el valor de `x`.

4.7.4. Calculando las direcciones de las variables locales

Hallar la dirección de una variable local definida en el segmento `data` o `bss` es sencillo, básicamente el encadenador hace esto. Sin embargo calcular la dirección de una variable local (o parámetro) en la pila no es directo. Sin embargo, es una necesidad muy común cuando se llaman subrutinas. Considere el caso de pasar la dirección de una variable (la llamaremos `x`) a una función (que llamaremos `foo`). Si x está en EBP − 8 en la pila, uno no puede usar:

```
mov    eax, ebp - 8
```

¿Por qué? El valor que `MOV` almacena en EAX debe ser calculado por el ensamblador (esto es, debe ser una constante). Sin embargo, hay una instrucción que hace el cálculo deseado. Es llamada `LEA` (*Load Efective Adress*). Lo siguiente calcularía la dirección de `x` y la almacena en EAX:

```
lea    eax, [ebp - 8]
```

Ahora EAX almacena la dirección de `x` y podría ser empujada en la pila cuando se llame la función `foo`. No se confunda, parece como si esta instrucción estuviera leyendo el dato en [EBP−8]; sin embargo esto *no* es verdad. ¡La instrucción `LEA` *nunca* lee la memoria! Solo calcula la dirección que sería leída por otra instrucción y almacena esta dirección en el primer operando de registro. Ya que no se hace ninguna lectura a memoria, no hay necesidad, ni está permitido definir el tamaño de la memoria (`dword` u otros).

4.7.5. Retornando valores

Las funciones diferentes a void retornan un valor. La convención de llamado de C especifica cómo se hace esto. Los valores de retorno se pasan a través de registros. Todos los tipos enteros (`char`, `int`, `enum`, etc.) se retornan en el registro EAX. Si son más pequeños que 32 bits, ellos son extendidos a 32 bits cuando se almacenan en EAX. (el cómo se extienden depende de si ellos tipos son con o sin signo.) Los valores de 64 bits se retornan en el par de registros EDX:EAX. Los valores tipo apuntador también se almacenan en EAX. Los valores de punto flotante son almacenados en el registro ST0 del coprocesador matemático. (Este registro se discute en el capítulo de punto flotante).

4.7.6. Otras convenciones de llamado

Las reglas anteriores describen la convención de llamado estándar de C que es soportada por todos los compiladores de C para 80x86. A menudo los compiladores soportan otras convenciones de llamado también. Cuando se interfaza con lenguaje ensamblador es *muy* importante conocer que convención de llamado está usando el compilador cuando llama su función. Normalmente, por omisión se usa la convención de llamado estándar; sin embargo no siempre es este el caso[4]. Los compiladores que usan varias convenciones a menudo tienen opciones de la línea de órdenes que se pueden usar para cambiar la convención por omisión. Ellos también le suministran extensiones a la sintaxis de C para asignar explícitamente convenciones de llamado a funciones individuales. Sin embargo, estas extensiones no están normalizadas y pueden variar de un compilador a otro.

El compilador GCC permite diferentes convenciones de llamado. La convención de una función se puede declarar explícitamente usando la extensión `__attribute__`. Por ejemplo para declarar una función void que usa la convención de llamado estándar llamada `f` que toma un parámetro int, use la siguiente sintaxis para su prototipo:

```
void f( int ) __attribute__((cdecl));
```

GCC también soporta la convención de *llamado estándar*. La función de arriba se podría declarar para usar esta convención reemplazando `cdecl` con `stdcall`. La diferencia entre `stdcall` y `cdecl` es que `stdcall` requiere que la subrutina quite los parámetros de la pila (como lo hace la convención de llamado de Pascal). Así, la convención `stdcall` sólo se puede usar con

[4]El compilador de C de Watcom es un ejemplo de uno que *no* usa la convención de llamado estándar por omisión. Vea el código fuente de ejemplo para Watcom para los detalles

funciones que tomen un número fijo de parámetros (ejm. unas que no sean como `printf` y `scanf`).

GCC también soporta un atributo adicional llamado `regparam` que le dice al compilador que use los registros para pasar hasta 3 argumentos enteros a su función en lugar de usar la pila. Este es un tipo común de optimización que soportan muchos compiladores.

Borland y Microsoft usan una sintaxis común para declarar convenciones de llamado. Ellas añaden a las palabras reservadas `__cdecl __stdcall` a C. Estas palabras reservadas actúan como modificadoras de funciones y aparecen inmediatamente antes nombre de la función en un prototipo. Por ejemplo, la función `f` de arriba se podría definir para Borland y Microsoft así:

```
void __cdecl f( int );
```

Hay ventajas y desventajas para cada convención de llamado. La principal ventaja de la convención `cdecl` es que es simple y muy flexible. Se puede usar para cualquier tipo de función de C y cualquier compilador de C. Usar otras convenciones puede limitar la portabilidad de la subrutina. Su principal desventaja es que puede ser más lenta que alguna de las otras y usa más memoria (ya que cada vez que se invoca de la función requiere código para quitar los parámetros de la pila).

Las ventajas de la convención `stdcall` es que usa menos memoria que `cdecl`. No se requiere limpiar la pila después de la instrucción `CALL`. Su principal desventaja es que no se puede usar con funciones que tengan un número variable de argumentos.

La ventaja de usar una convención que use registros para pasar enteros es la velocidad. La principal desventaja es que la convención es más compleja. Algunos parámetros pueden estar en los registros y otros en la pila.

4.7.7. Ejemplos

El siguiente es un ejemplo que muestra cómo una rutina de ensamblador se puede interfasar con un programa de C (observe que este programa no usa el programa esqueleto de ensamblador (Figura 1.7) o el módulo driver.c)

main5.c

```
#include <stdio.h>
/* prototipo para la rutina en ensamblador */
void calc_sum( int, int * ) __attribute__((cdecl));

int main( void )
{
```

```
  int n, sum;

  printf("Sumar enteros hasta: ");
  scanf(" %d", &n);
  calc_sum(n, &sum);
  printf("Sum is %d\n", sum);
  return 0;
}
```

main5.c

sub5.asm

```
; subroutinea _calc_sum
; halla la suma de los enteros de 1 hasta n
; Parametros:
;   n    - hasta dónde sumar (en [ebp + 8])
;   sump - apuntador a un int para almacenar sum (en [ebp + 12])
; pseudocódigo en C:
; void calc_sum( int n, int * sump )
; {
;   int i, sum = 0;
;   for( i=1; i <= n; i++ )
;     sum += i;
;   *sump = sum;
; }

segment .text
        global  _calc_sum
;
; local variable:
;   sum at [ebp-4]
_calc_sum:
        enter   4,0               ; hace espacio para sum en la pila
        push    ebx               ; IMPORTANTE!

        mov     dword [ebp-4],0   ; sum = 0
        dump_stack 1, 2, 4        ; imprime la pila desde ebp-8 hasta ebp+16
        mov     ecx, 1            ; ecx es i en el pseudocódigo
for_loop:
        cmp     ecx, [ebp+8]      ; cmp i y n
        jnle    end_for           ; si no i <= n, sale

        add     [ebp-4], ecx      ; sum += i
```

```
Sumar enteros hasta: 10
Stack Dump # 1
EBP = BFFFFB70 ESP = BFFFFB68
 +16  BFFFFB80  080499EC
 +12  BFFFFB7C  BFFFFB80
  +8  BFFFFB78  0000000A
  +4  BFFFFB74  08048501
  +0  BFFFFB70  BFFFFB88
  -4  BFFFFB6C  00000000
  -8  BFFFFB68  4010648C
Sum is 55
```

Figura 4.13: Muestra de la ejecución del programa sub5

```
32          inc     ecx
33          jmp     short for_loop
34
35  end_for:
36          mov     ebx, [ebp+12]     ; ebx = sump
37          mov     eax, [ebp-4]      ; eax = sum
38          mov     [ebx], eax
39
40          pop     ebx               ; restaura ebx
41          leave
42          ret
```

sub5.asm

¿Por qué la línea 22 de `sub5.asm` es importante? Porque la convención de llamado de C requiere que el valor de EBX no se modifique por la función llamada. Si esto no se hace muy probable que el programa no trabaje correctamente.

La línea 25 demuestra como trabaja el macro `dump_stack`. Recuerde que el primer parámetro es sólo una etiqueta numérica, y el segundo y tercero determinan cuántas palabras dobles se muestran antes y después de EBP respectivamente. La Figura 4.13 muestra un ejemplo de la ejecución del programa. Para este volcado, uno puede ver que la dirección de la palabra doble que almacena la suma es BFFFFB80 (en EBP + 12); el número a sumar es 0000000A (en EBP + 8); la dirección de retorno para la rutina es 08048501 (en EBP + 4); el valor guardado de EBP es BFFFFB88 (en EBP); el valor de la variable local es 0 en (EBP- 4); y finalmente el valor guardado de EBX es 4010648C (en EBP- 8).

La función `calc_sum` podría ser rescrita para devolver la suma como un valor de retorno en lugar de usar un parámetro apuntador. Ya que la suma es un valor entero, la suma se podría dejar en el registro EAX. La línea 11 del archivo `main5.c` debería ser cambiada por:

```
sum = calc_sum(n);
```

También, el prototipo de `calc_sum` debería ser alterado. A continuación, el código modificado en ensamblador.

sub6.asm

```
1  ; subroutina _calc_sum
2  ; halla la suma de los enteros de 1 hasta  n
3  ; Parámetros:
4  ;   n    - hasta dónde se suma (en [ebp + 8])
5  ; Valor de retorno:
6  ;   la suma
7  ; pseudocódigo en C:
8  ; int calc_sum( int n )
9  ; {
10 ;   int i, sum = 0;
11 ;   for( i=1; i <= n; i++ )
12 ;     sum += i;
13 ;   return sum;
14 ; }
15 segment .text
16         global  _calc_sum
17 ;
18 ; local variable:
19 ;   sum at [ebp-4]
20 _calc_sum:
21         enter   4,0               ; hace espacio en la pila para sum
22
23         mov     dword [ebp-4],0   ; sum = 0
24         mov     ecx, 1            ; ecx es i en el pseudocódigo
25 for_loop:
26         cmp     ecx, [ebp+8]      ; cmp i and n
27         jnle    end_for           ; si no i <= n, sale
28
29         add     [ebp-4], ecx      ; sum += i
30         inc     ecx
31         jmp     short for_loop
32
33 end_for:
```

```
segment .data
format       db "%d", 0

segment .text
...
      lea    eax, [ebp-16]
      push   eax
      push   dword format
      call   _scanf
      add    esp, 8
...
```

Figura 4.14: Llamando `scanf` desde ensamblador

```
        mov     eax, [ebp-4]      ; eax = sum

        leave
        ret
```
sub6.asm

4.7.8. Llamando funciones de C desde ensamblador

Una gran ventaja de interfasar C y ensamblador es que le permite al código de ensamblador acceder a la gran biblioteca de C y usar funciones escritas por el usuario. Por ejemplo, si uno desea llamar la función `scanf` para leer un entero del teclado. La Figura 4.14 muestra el código que hace esto. Un punto muy importante para recordar es que `scanf` sigue la convención de llamado de C normalizada. Esto significa que preserva los valores de los registros EBX, ESI y EDI; sin embargo los registros EAX, ECX y EDX se pueden modificar. De hecho, EAX definitivamente será cambiado, ya que él contendrá el valor de retorno del llamado a `scanf`. Para otros ejemplos del uso de interfaces con C, vea el código en `asm_io.asm` que fue usado para crear `asm_io.obj`.

4.8. Subprogramas reentrantes y recursivos

Un programa reentrante debe satisfacer las siguientes propiedades:

- No debe modificar ninguna instrucción del programa. En un lenguaje de alto nivel esto podría ser difícil, pero en ensamblador no es difícil para un programa intentar modificar su propio código. Por ejemplo:

```
mov    word [cs:\$+7], 5      ; copia 5 en la palabra 7
                              ; bytes adelante
add    ax, 2                  ; la instrucción anterior
                              ; cambia 2 por 5
```

Este código trabajaría en modo real, pero en sistemas operativos con modo protegido el segmento de código está marcado de solo lectura. Cuando se ejecute la primera línea, el programa se abortará en estos sistemas. Esto es una mala forma de programar por muchas razones. Es confuso, difícil de mantener y no permite compartir el código (vea más adelante).

- No debe modificar datos globales (tal como los datos que están en el segmento `data` y `bss`). Todas las variables son almacenadas en la pila.

Hay varias ventajas de escribir código reentrante:

- Un subprograma reentrante se puede llamar recursivamente.
- Un programa reentrante puede compartirse en múltiples procesos. En muchos sistemas operativos multitarea, si están ejecutandose varias instancias solo *una* copia del código está en memoria. Las bibliotecas compartidas y DLL (*Dinamic Link Libraries*) usan esta idea también.
- Los subprogramas reentrantes trabajan mucho mejor en programas *multihilo*[5] Windows 9x/NT y la mayoría de los sistemas operativos tipo UNIX (Solaris, Linux, etc.) soportan programas multihilo.

4.8.1. Subprogramas recursivos

Estos tipos de subprogramas se invocan así mismos. La recursión puede ser *directa* o *indirecta*. La recursión directa ocurre cuando un subprograma, digamos `foo`, se invoca así mismo dentro del cuerpo de `foo`. La recursión indirecta ocurre cuando un subprograma no se llama así mismo directamente, pero otro subprograma lo llama. Por ejemplo, el subprograma `foo` podría llamar a `bar` y `bar` podría llamar a `foo`.

Los programas recursivos deben tener una *condición de terminación*. Cuando esta condición es verdadera, no se necesita hacer más llamadas. Si la rutina recursiva no tiene una condición de terminación o la condición nunca se vuelve verdadera, la recursión nunca termina (muy parecido a un bucle infinito).

La Figura 4.15 muestra una función que calcula el factorial recursivamente. Puede ser llamado desde C con:

[5] un programa multihilo tiene varios hilos de ejecución. Esto es, el programa en sí mismo es multitarea.

```
1  ; finds n!
2  segment .text
3        global _fact
4  _fact:
5        enter  0,0
6
7        mov    eax, [ebp+8]    ; eax = n
8        cmp    eax, 1
9        jbe    term_cond       ; si n <= 1, termina
10       dec    eax
11       push   eax
12       call   _fact           ; eax = fact(n-1)
13       pop    ecx             ; respuesta en  eax
14       mul    dword [ebp+8]   ; edx:eax = eax * [ebp+8]
15       jmp    short end_fact
16 term_cond:
17       mov    eax, 1
18 end_fact:
19       leave
20       ret
```

Figura 4.15: Función factorial recursiva

	n(3)
marco n=3	Dirección de retorno
	EBP guardado
	n(2)
marco n=2	Dirección de retorno
	EBP guardado
	n(1)
marco n=1	Dirección de retorno
	EBP guardado

Figura 4.16: Marco de la pila para la funición factorial

```
void f( int x )
{
  int i;
  for( i=0; i < x; i++ ) {
     printf (" %d\n", i);
     f(i);
  }
}
```

Figura 4.17: Otro ejemplo (versión de C)

```
x = fact(3);          /* find 3! */
```

La Figura 4.16 muestra cómo se ve la pila en el punto más profundo del llamado de la función anterior.

Las Figuras 4.17 y 4.18 muestran otro ejemplo recursivo más complicado en C y en ensamblador respectivamente. ¿Cuál es la salida para `f(3)`? Observe que la instrucción `ENTER` crea un nuevo `i` en la pila para cada llamado recursivo. Así cada instancia recursiva de `f` tiene su propia variable independiente `i`. Definiendo `i` como una palabra doble en el segmento `data` no trabajará igual.

4.8.2. Revisión de tipos de variables según su alcance en C

C suministra varios tipos de almacenamiento para las variables.

global Estas variables están definidas fuera de cualquier función y están almacenadas en lugares fijos de memoria (en los segmentos `data` o `bss`) y existen desde el inicio hasta el final del programa. Por omisión ellas pueden ser accedidas por cualquier función en el programa; sin embargo, si ellas están declaradas como `static`, solo las funciones en el mismo módulo pueden acceder a ellas (en términos de ensamblador la etiqueta es interna, no externa).

static Estas son variables *locales* de una función que son declaradas `static` (desafortunadamente, C usa la palabra reservada `static` para dos propósitos diferentes). Estas variables se almacenan también en lugares fijos de memoria (en `data` o `bss`), pero solo se pueden acceder directamente en las funciones en donde ellas se definieron.

automatic Este es el tipo por omisión para una variable definida dentro de una función. Estas variables son colocadas en la pila cuando la función en la cual están definidas se invocada y quitadas cuando la función retorna. Así, ellas no tienen un lugar fijo en memoria.

```
1  %define i ebp-4
2  %define x ebp+8          ; macros útiles
3  segment .data
4  format       db "%d", 10, 0     ; 10 = '\n'
5  segment .text
6        global _f
7        extern _printf
8  _f:
9        enter  4,0           ; toma espacio en la pila para i
10
11       mov    dword [i], 0  ; i = 0
12 lp:
13       mov    eax, [i]      ; es i < x?
14       cmp    eax, [x]
15       jnl    quit
16
17       push   eax           ; llama printf
18       push   format
19       call   _printf
20       add    esp, 8
21
22       push   dword [i]     ; llama f
23       call   _f
24       pop    eax
25
26       inc    dword [i]     ; i++
27       jmp    short lp
28 quit:
29       leave
30       ret
```

Figura 4.18: Otro ejemplo (versión de ensamblador)

register Esta palabra reservada le pregunta al compilador usar un registro para el dato de esta variable. Esto es solo una *solicitud*. El compilador *no* tiene que hacerlo. Si la dirección de la variable se usa en cualquier parte del programa la variable no será de este tipo (ya que los registros no tienen dirección). También, solo tipos enteros pueden ser valores tipo register. Tipos estructurados no lo pueden ser, ¡ellos no caben en un registro! Los compiladores de C a menudo convertirán variables normales automatic en variables register sin ningún aviso al programador.

volatile Esta palabra clave le dice al compilador que el valor en la variable puede cambiar en cualquier momento. Esto significa que el compilador no puede hacer ninguna suposición sobre cuándo se modifica la variable. A menudo un compilador podría almacenar el valor de una variable en un registro temporalmente y usar el registro en vez de la variable en una parte del código. No se pueden hacer este tipo de optimizaciones con variables `volatile`. Un ejemplo común de una variable volatile podría ser una que se alterara por dos hilos de un programa multihilo. Considere el siguiente código:

```
x = 10;
y = 20;
z = x;
```

Si `x` pudiera ser alterada por otro hilo, es posible que otro hilo cambie `x` entre las líneas 1 y 3 así que `z` podría no ser 10. Sin embargo, si `x` no fue declarada volatile, el compilador podría asumir que `x` no cambia y fija `z` a 10.

If `x` could be altered by another thread, it is possible that the other thread changes `x` between lines 1 and 3 so that `z` would not be 10. However, if the `x` was not declared volatile, the compiler might assume that `x` is unchanged and set `z` to 10.

Otro uso de `volatile` es evitar que el compilador use un registro para una variable.

Capítulo 5

Arreglos

5.1. Introducción

Un *arreglo* es un bloque contiguo de una lista de datos en la memoria. Cada elemento de la lista debe ser del mismo tipo y usar exactamente el mismo número de bytes de memoria para almacenarlo. Por estas propiedades, los arreglos permiten un acceso eficiente de los datos por su posición (o índice) en el arreglo. La dirección de cualquier elemento se puede calcular conociendo tres hechos:

- La dirección del primer elemento del arreglo.
- El número de bytes de cada elemento.
- El índice del elemento.

Es conveniente considerar el índice del primer elemento como 0 (tal como C). Es posible usar otros valores para el primer índice, pero esto complica los cálculos.

5.1.1. Definir arreglos

Definir arreglos en los segmentos `data` y `bss`

Para definir un arreglo con valor inicial en el segmento `data`, use las directivas normales: `db`, `dw`, etc. NASM también suministra la directiva llamada `TIMES` que se puede usar para repetir una instrucción muchas veces sin tener que duplicar las instrucciones a mano. La Figura 5.1 muestra varios ejemplos de estos.

Para definir un arreglo sin valor inicial en el segmento `bss`, use las directivas `resb`, `resw`, etc. Recuerde que estas directivas tienen un operando

```
1  segment .data
2  ; define un arreglo de 10 plabras dobles con valores
3  ; iniciales de 1,2,..,10
4  a1           dd   1, 2, 3, 4, 5, 6, 7, 8, 9, 10
5  ; define un arreglo de 10 palabras inicadas todas con  0
6  a2           dw   0, 0, 0, 0, 0, 0, 0, 0, 0, 0
7  ; lo mismo de antes usando TIMES
8  a3           times 10 dw 0
9  ; define un arreglo de bytes con 200 ceros y luego 100 con unos
10 a4           times 200 db 0
11              times 100 db 1
12
13 segment .bss
14 ; define un arreglo de 10 palabras dobles sin valor inicial
15 a5           resd  10
16 ; define un arreglo de 100 palabras dobles sin valor inicial
17 a6           resw  100
```

Figura 5.1: Definir arreglos

que especifica cuántas unidades de memoria reservar. La Figura 5.1 muestra también ejemplos de este tipo de definiciones

Definir arreglos como variables locales en la pila

No hay una manera directa de definir una arreglo como variable local en la pila. Como antes, uno calcula el total de bytes necesarios para *todas* las variables locales, incluidos los arreglos, y resta esto de ESP (o directamente usando la instrucción `ENTER`). Por ejemplo, si una función necesita una variable caracter, dos enteros y 50 elementos de un arreglo de palabras, uno necesitaría $1 + 2 + 50 = 109$ bytes. Sin embargo, el número restado de ESP debería ser un múltiplo de cuatro (112 en este caso) para que ESP esté en el límite de una palabra doble. Uno podría formar las variables dentro de estos 109 bytes de varias maneras. La Figura 5.2 muestra dos maneras posibles. La parte sin uso del primer ordenamiento está para dejar las palabras dobles en los límites de palabras dobles para optimizar la velocidad del acceso a la memoria.

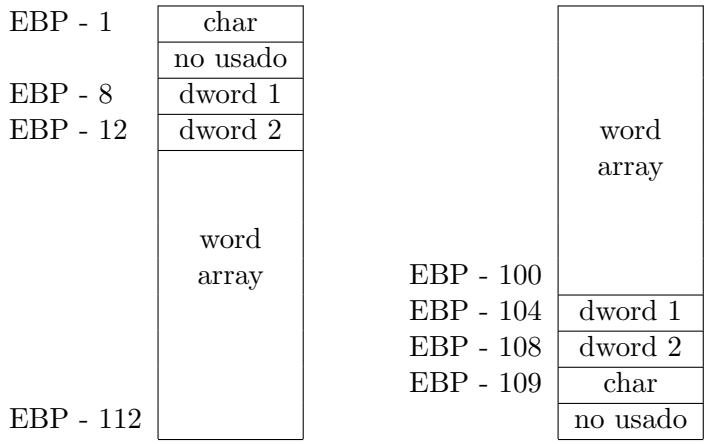


Figura 5.2: Disposición de la pila

5.1.2. Acceder a elementos de los arreglos

No existe el operador `[ ]` en el lenguaje ensamblador como en C. Para acceder a un elemento del arreglo, se debe calcular su dirección. Considere las dos definiciones siguientes de arreglos:

```
array1      db     5, 4, 3, 2, 1     ; arreglo de  bytes
array2      dw     5, 4, 3, 2, 1     ; arreglo de palabras
```

A continuación, algunos ejemplos que utilizan estos arreglos.

```
1       mov    al, [array1]            ; al = array1[0]
2       mov    al, [array1 + 1]        ; al = array1[1]
3       mov    [array1 + 3], al        ; array1[3] = al
4       mov    ax, [array2]            ; ax = array2[0]
5       mov    ax, [array2 + 2]        ; ax = array2[1] (NO es array2[2]!)
6       mov    [array2 + 6], ax        ; array2[3] = ax
7       mov    ax, [array2 + 1]        ; ax = ??
```

En la línea 5, se referencia el elemento 1 del arreglo, no el segundo. ¿Por qué? Las palabras son unidades de 2 bytes, para moverse al siguiente elemento del arreglo uno se debe desplazar 2 bytes adelante, no uno. La línea 7 leerá un byte del primer elemento y uno del segundo. En C, el compilador mira el tipo de apuntador para determinar cuántos bytes mover en una expresión aritmética para que el programador no tenga que hacerla. Sin embargo en ensamblador es él el que calcula el tamaño de los elementos del arreglo cuando se mueve de elemento en elemento.

La Figura 5.3 muestra un fragmento de código que suma todos los elementos de `array1` del ejemplo anterior. En la línea 7, a AX se le suma DX. ¿Por qué no AL? Primero, los dos operandos de la instrucción `ADD` deben

```
1       mov    ebx, array1          ; ebx = dirección de array1
2       mov    dx, 0                ; dx almacenará sum
3       mov    ah, 0                ; ?
4       mov    ecx, 5
5   lp:
6       mov    al, [ebx]            ; al = *ebx
7       add    dx, ax               ; dx += ax (not al!)
8       inc    ebx                  ; bx++
9       loop   lp
```

Figura 5.3: Sumar elementos de un arreglo (Versión 1)

```
1       mov    ebx, array1          ; ebx = dirección de array1
2       mov    dx, 0                ; dx almacenará sum
3       mov    ecx, 5
4   lp:
5       add    dl, [ebx]            ; dl += *ebx
6       jnc    next                 ; if no hay carry vaya a next
7       inc    dh                   ; inc dh
8   next:
9       inc    ebx                  ; bx++
10      loop   lp
```

Figura 5.4: Sumar elementos de un arreglo (Versión 2)

ser del mismo tamaño. Segundo, es probable que al sumar bytes y lleguemos a un resultado que no quepa en un byte. Usando DX, se permite una suma hasta 65.535. Sin embargo es importante ver que se está sumando AH también. Esta es la razón por la cual se fija AH a cero.[1] en la línea 3.

Las Figuras 5.4 y 5.5 muestran dos alternativas para calcular la suma. Las líneas en itálica reemplazan las líneas 6 y 7 de la Figura 5.3.

5.1.3. Direccionamiento indirecto más avanzado

No es sorprendente que se use direccionamiento indirecto con arreglos. La forma más general de una referencia indirecta memoria es

*[reg base + factor*reg índice + constante]*

[1] Fijando AH a cero se asume implícitamente que AL es un número sin signo. Si es con signo, la acción apropiada sería insertar una instrucción `CBW` entre las líneas 6 y 7

```
1       mov     ebx, array1             ; ebx = dirección de  array1
2       mov     dx, 0                   ; dx almacenará sum
3       mov     ecx, 5
4   lp:
5       add     dl, [ebx]               ; dl += *ebx
6       adc     dh, 0                   ; dh += bandera de carry + 0
7       inc     ebx                     ; bx++
8       loop    lp
```

Figura 5.5: Sumar elementos de un arreglo (Versión 3)

donde:

reg base es uno de los registros EAX, EBX, ECX, EDX, EBP, ESP, ESI o EDI.

factor es 1, 2, 4 o 8. (Es 1, si el factor se omite)

index reg es uno de estos registros EAX, EBX, ECX, EDX, EBP, ESI, EDI. (observe que ESP no está en la lista.) (o expresión).

5.1.4. Ejemplo

Se presenta un ejemplo que usa un arreglo y se pasa a la función. Usa el programa `array1c.c` (listado abajo) como driver y no el programa `driver.c`.

array1.asm

```
1   %define ARRAY_SIZE 100
2   %define NEW_LINE 10
3
4   segment .data
5   FirstMsg        db   "Primeros 10 elementos del arreglo", 0
6   Prompt          db   "Ingrese el índice del elemento a mostrar: ", 0
7   SecondMsg       db   "Elemento %d es %d", NEW_LINE, 0
8   ThirdMsg        db   "Elementos 20 hasta 29 del arreglo", 0
9   InputFormat     db   "%d", 0
10
11  segment .bss
12  array           resd ARRAY_SIZE
13
14  segment .text
15          extern  _puts, _printf, _scanf, _dump_line
```

```
16          global  _asm_main
17  _asm_main:
18          enter   4,0               ; variable local en  EBP - 4
19          push    ebx
20          push    esi
21
22  ; Se inicia el arreglo con 100, 99, 98, 97, ...
23
24          mov     ecx, ARRAY_SIZE
25          mov     ebx, array
26  init_loop:
27          mov     [ebx], ecx
28          add     ebx, 4
29          loop    init_loop
30
31          push    dword FirstMsg         ; imprime FirstMsg
32          call    _puts
33          pop     ecx
34
35          push    dword 10
36          push    dword array
37          call    _print_array           ; imprime los 10 primeros
38                                         ; elementos del arreglo
39          add     esp, 8
40
41  ; Le pregunta al usuario el índice del elemento
42  Prompt_loop:
43          push    dword Prompt
44          call    _printf
45          pop     ecx
46
47          lea     eax, [ebp-4]      ; eax = dirección de la variable local
48          push    eax
49          push    dword InputFormat
50          call    _scanf
51          add     esp, 8
52          cmp     eax, 1              ; eax = valor de retorno de scanf
53          je      InputOK
54
55          call    _dump_line  ; vacía el resto de la línea y vuelve a
56                                      ; comenzar
57          jmp     Prompt_loop          ; si la entrada no es válida
```

```
58
59  InputOK:
60          mov     esi, [ebp-4]
61          push    dword [array + 4*esi]
62          push    esi
63          push    dword SecondMsg      ; imprime el valor del elemento
64          call    _printf
65          add     esp, 12
66
67          push    dword ThirdMsg       ; imprime los elementos  20 a 29
68          call    _puts
69          pop     ecx
70
71          push    dword 10
72          push    dword array + 20*4     ; dirección de array[20]
73          call    _print_array
74          add     esp, 8
75
76          pop     esi
77          pop     ebx
78          mov     eax, 0            ; retorna a C
79          leave
80          ret
81
82  ;
83  ; rutina _print_array
84  ; Rutina llamable desde C que imprime elementos de arreglos de palabras
85  ; dobles como enteros con signo
86  ; Prototipo de C:
87  ; void print_array( const int * a, int n);
88  ; Parámetros:
89  ;   a - Apuntador al arreglo a imprimir (en ebp+8 en la pila)
90  ;   n - número de enteros a imprimir (en ebp+12 en la pila)
91
92  segment .data
93  OutputFormat    db   "%-5d %5d", NEW_LINE, 0
94
95  segment .text
96          global  _print_array
97  _print_array:
98          enter   0,0
99          push    esi
```

```
100         push    ebx
101
102         xor     esi, esi                ; esi = 0
103         mov     ecx, [ebp+12]           ; ecx = n
104         mov     ebx, [ebp+8]            ; ebx = dirección del arreglo
105 print_loop:
106         push    ecx                     ; ¡printf podría cambiar ecx!
107
108         push    dword [ebx + 4*esi]     ; empuja a la pila array[esi]
109         push    esi
110         push    dword OutputFormat
111         call    _printf
112         add     esp, 12                 ; quita los parámetros (¡deja ecx!)
113
114         inc     esi
115         pop     ecx
116         loop    print_loop
117
118         pop     ebx
119         pop     esi
120         leave
121         ret
```

array1.asm

array1c.c

```
#include <stdio.h>

int asm_main( void );
void dump_line( void );

int main()
{
  int ret_status ;
   ret_status = asm_main();
  return ret_status ;
}

/*
 * función dump_line
 * Vacía todos los caracteres de la línea del búfer de entrada
 */
void dump_line()
```

```
{
  int ch;

  while( (ch = getchar()) != EOF && ch != '\n')
    /* cuerpo vacío*/ ;
}
```

array1c.c

Revisión de la instrucción `LEA`

La instrucción `LEA` se puede usar para otros propósitos que sólo calcular direcciones. Un uso común es para cálculos rápidos. Considere lo siguiente:

```
lea    ebx, [4*eax + eax]
```

Esto efectivamente almacena el valor de $5 \times$ `EAX` en EBX. Usando `LEA` para hacer esto, es más fácil y rápido que usar `MUL`. Sin embargo, uno debe tener en cuenta que la expresión dentro de los paréntesis cuadrados *debe* ser una dirección indirecta correcta. Así por ejemplo, esta instrucción no se puede usar para multiplicar por 6 rápidamente.

5.1.5. Arreglos multidimiensionales

Un arreglo multidimensional no es realmente muy diferente que los arreglos unidimensionales ya discutidos. De hecho, están representados en memoria tal como un arreglo unidimensional.

Arreglos bidimensionales

No debe sorprender que el arreglo multidimiensional más elemental es un arreglo bidimensional. Un arreglo bidimensional se presenta a menudo como una malla de elementos. Cada elemento está identificado por un par de índices. Por convención, el primer índice es identificado con la fila del elemento y el segundo índice es la columna.

Considere un arreglo con tres filas y dos columnas definidas como:

```
int a [3][2];
```

El compilador de C reservaría espacio para un arreglo de 6 ($= 2 \times 3$) enteros y ordena los elementos como sigue:

Índice	0	1	2	3	4	5
Elemento	a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]

La tabla intenta mostrar cómo el elemento referenciado como `a[0][0]` es

```
1    mov    eax, [ebp - 44]          ; ebp - 44 es el sitio de i
2    sal    eax, 1                   ; multiplica i por 2
3    add    eax, [ebp - 48]          ; add j
4    mov    eax, [ebp + 4*eax - 40]  ; ebp - 40 es la dirección a[0][0]
5    mov    [ebp - 52], eax          ; Almacena el resultado en  ebp - 52
```

Figura 5.6: Ensamblador para `x = a[i][j]`

almacenado en el comienzo de los 6 elementos de un arreglo unidimensional. El elemento `a[0][1]` se almacena en la siguiente posición (índice 1) y así sucesivamente. Cada fila del arreglo bidimensional se almacena contiguamente en memoria. El último elemento de una fila es seguido por el primer elemento de la próxima fila. Esto es conocido como la representación rowwise del arreglo y es como un compilador de C/C++ podría representar el arreglo.

¿Cómo el compilador determina dónde aparece `a[i][j]` en la representación *rowwise*? Una fórmula elemental calculará el índice de `i` y `j`. La fórmula en este caso es $2i + j$. No es difícil ver cómo se deriva esta fórmula. Cada fila es de dos elementos; así, el primer elemento de la fila `i` está en la posición $2i$. Entonces la posición de la columna j se encuentra sumando j a $2i$. Este análisis también muestra cómo la fórmula se generaliza para un arreglo con `N` columnas. $N \times i + j$. Observe que la fórmula *no* depende del número de filas.

Como un ejemplo, veamos cómo compila *gcc* el siguiente código (usando el arreglo `a` definido antes):

```
x = a[i][j];
```

La Figura 5.6 muestra cómo el ensamblador trabaja esto, Así, el compilador esencialmente convierte el código a:

```
x = *(&a[0][0] + 2*i + j);
```

y de hecho, el programador podría escribir de esta manera con los mismos resultados.

No hay nada mágico sobre la escogencia de la representación *rowwise* del arreglo. Una representación *columnwise* podría trabajar bien:

Index	0	1	2	3	4	5
Element	a[0][0]	a[1][0]	a[2][0]	a[0][1]	a[1][1]	a[2][1]

En la representación *columnwise*, cada columna se almacena contigua. El elemento `[i][j]` se almacenan en la posición $i + 3j$. Otros lenguajes (FORTRAN, por ejemplo) usan la representación *columnwise*. Esto es importante cuando se interfaza el código con varios lenguajes.

Dimensiones mayores que dos

Para las dimensiones mayores de 2, se aplica la misma idea básica. Considere el arreglo tridimensional:

```
int b [4][3][2];
```

Este arreglo podría ser almacenado como si fueran cuatro arreglos bidimensionales cada uno de tamaño `[3][2]` consecutivamente en memoria. La tabla de abajo muestra como comienza él:

Index	0	1	2	3	4	5
Element	b[0][0][0]	b[0][0][1]	b[0][1][0]	b[0][1][1]	b[0][2][0]	b[0][2][1]
Index	6	7	8	9	10	11
Element	b[1][0][0]	b[1][0][1]	b[1][1][0]	b[1][1][1]	b[1][2][0]	b[1][2][1]

La fórmula para calcular la posición de `b[i][j][k]` es $6i + 2j + k$. El 6 está determinado por el tamaño de los arreglos `[3][2]`. En general, para un arreglo `a[i][j][k]` será: $M \times N \times i + N \times j + k$. Observe que nuevamente la primera dimensión `L` no aparece en la fórmula.

Para dimensiones mayores, se generaliza el mismo proceso. Para un arreglo n dimensional de dimensiones D_1 a D_n, la posición del elemento denotada por los índices i_1 a i_n está dada por la fórmula:

$$D_2 \times D_3 \cdots \times D_n \times i_1 + D_3 \times D_4 \cdots \times D_n \times i_2 + \cdots + D_n \times i_{n-1} + i_n$$

o puede ser escrito más corto como:

$$\sum_{j=1}^{n} \left(\prod_{k=j+1}^{n} D_k \right) i_j$$

La primera dimensión D_1 no aparece en la fórmula:

Por una representación *columnwise* la fórmula general sería:

$$i_1 + D_1 \times i_2 + \cdots + D_1 \times D_2 \times \cdots \times D_{n-2} \times i_{n-1} + D_1 \times D_2 \times \cdots \times D_{n-1} \times i_n$$

Aquí es donde puede decirle al autor que fue un mejor físico (o ¿fue la referencia a FORTRAN un regalo?)

En la notación matemática griega de über:

$$\sum_{j=1}^{n} \left(\prod_{k=1}^{j-1} D_k \right) i_j$$

En este caso, es la última dimensión D_n, la que no aparece en la fórmula.

Pasar arreglos multidimensionales como parámetros en C

La representación *rowwise* de arreglos multidimensionales tiene un efecto directo en la programación en C. Para arreglos unidimensionales, no se necesita el tamaño del arreglo para calcular donde está localizado un elemento en

memoria. Esto no es cierto para arreglos multidimensionales. Para acceder a los elementos de estos arreglos el compilador debe conocer todo menos la primera dimensión. Esto se vuelve aparente cuando consideramos el prototipo de la función que toma un arreglo multidimensional como parámetro. Lo siguiente no compilará:

```
void f( int a[ ][ ] );  /* No hay información de la dimensión */
```

Sin embargo, lo siguiente compila:

```
void f( int a[ ][2] );
```

Cualquier arreglo bidimensional con dos columnas puede ser pasado a esta función. La primera dimensión no se necesita.[2]

No se confunda con una función con este prototipo:

```
void f( int * a[ ] );
```

Esto define un arreglo unidimensional de apuntadores enteros (que incidentalmente puede ser usado para crear un arreglo de arreglos que se comporta muy parecido a un arreglo bidimensional).

Para arreglos de dimensiones mayores, se debe especificar todo menos la primera dimensión. Por ejemplo, un arreglo de 4 dimensiones se podría pasar como:

```
void f( int a[ ][4][3][2] );
```

5.2. Instrucciones de arreglos/cadenas

La familia de procesadores 80x86 suministran varias instrucciones que están diseñadas para trabajar con arreglos. Estas instrucciones son llamadas *instrucciones de cadena*. Ellas usan los registros de índice (ESI y EDI) para realizar una operación y entonces automáticamente incrementar o decrementar uno o los dos registros de índice. La *bandera dirección* (DF) en el registro FLAGS determina si los registros de índice son incrementados o decrementados. Hay dos instrucciones que modifican la bandera dirección:

CLD Borra la bandera de dirección. En este estado, los registros de índice se incrementan.

STD Establece la bandera de dirección. En este estado los registros de índice se decrementan.

Un error *muy común* en la programación de 80x86 es olvidar colocar la bandera de dirección en el estado correcto de manera explícita. Esto a menudo

[2] Se puede especificar un tamaño allí, pero será ignorado por el compilador.

LODSB AL = [DS:ESI] ESI = ESI ± 1	STOSB [ES:EDI] = AL EDI = EDI ± 1
LODSW AX = [DS:ESI] ESI = ESI ± 2	STOSW [ES:EDI] = AX EDI = EDI ± 2
LODSD EAX = [DS:ESI] ESI = ESI ± 4	STOSD [ES:EDI] = EAX EDI = EDI ± 4

Figura 5.7: Instrucciones de cadena de lectura y escritura

```
1  segment .data
2  array1  dd  1, 2, 3, 4, 5, 6, 7, 8, 9, 10
3
4  segment .bss
5  array2  resd 10
6
7  segment .text
8        cld                    ; ¡No olvide esto!
9        mov    esi, array1
10       mov    edi, array2
11       mov    ecx, 10
12 lp:
13       lodsd
14       stosd
15       loop  lp
```

Figura 5.8: Ejemplo de carga y almacenamiento

hace que el código trabaje la mayor parte del tiempo (cuando sucede que la bandera de dirección está en el estado deseado) pero no trabaja *todo* el tiempo.

5.2.1. Leer y escribir en la memoria

Las instrucciones más elementales leen o escriben en memoria. Ellas pueden leer o escribir un byte, palabra o palabra doble de una vez. La Figura 5.7 muestra estas instrucciones con una corta descripción en pseudocódigo de qué hace ella. Hay varios puntos para tener en cuenta. Primero, ESI se usa para leer y EDI para escribir. Es fácil recordar esto si uno tiene en cuenta que SI significa *Source Index* y DI significa *Destination Index*. Ahora, observe que el registro que almacena el dato es fijo (AL, AX o AEX). Finalmente, observe que las instrucciones de almacenamiento usan ES para determinar

MOVSB	byte [ES:EDI] = byte [DS:ESI] ESI = ESI ± 1 EDI = EDI ± 1
MOVSW	word [ES:EDI] = word [DS:ESI] ESI = ESI ± 2 EDI = EDI ± 2
MOVSD	dword [ES:EDI] = dword [DS:ESI] ESI = ESI ± 4 EDI = EDI ± 4

Figura 5.9: Instrucciones de cadena de movimiento de memoria

```
1  segment .bss
2  array  resd 10
3
4  segment .text
5        cld                    ; ¡No olvide esto!
6        mov    edi, array
7        mov    ecx, 10
8        xor    eax, eax
9        rep stosd
```

Figura 5.10: Ejemplo de arreglo cero

el segmento donde escribir, no DS. En la programación en modo protegido esto normalmente no es un problema, ya que hay sólo un segmento de datos y ES debería ser iniciado automáticamente para referenciarlo. (tal como lo es DS). Sin embargo, en la programación en modo real es *muy* importante para el programador iniciar ES con el valor de selector del segmento correcto.[3] La Figura 5.8 muestra un ejemplo del uso de estas instrucciones que copia un arreglo en otro.

La combinación de una instrucción `LODSx` y `STOSx` (en las líneas 13 y 14 de la Figura 5.8) es muy común. De hecho, esta combinación se puede realizar con una sola instrucción para cadenas `MOVSx`. La Figura 5.9 describe las operaciones que estas instrucciones realizan. Las líneas 13 y 14 de la Figura 5.8 podría ser reemplazada con una sola instrucción `MOVSD` con el mismo efecto. La única diferencia sería que el registro EAX no sería usado

[3]Otra complicación es que uno no puede copiar el valor de DS en el registro de ES directamente usando la instrucción `MOV`. En lugar de ello, el valor DS debe copiarse a un registro de propósito general (como AX) y entonces copiado de este registro a ES usando dos instrucciones `MOV`.

CMPSB	compara el byte [DS:ESI] y el byte [ES:EDI] ESI = ESI ± 1 EDI = EDI ± 1
CMPSW	compara la palabra [DS:ESI] y la word [ES:EDI] ESI = ESI ± 2 EDI = EDI ± 2
CMPSD	compara la dword [DS:ESI] y la dword [ES:EDI] ESI = ESI ± 4 EDI = EDI ± 4
SCASB	compara AL y [ES:EDI] EDI ± 1
SCASW	compara AX y [ES:EDI] EDI ± 2
SCASD	compara EAX y [ES:EDI] EDI ± 4

Figura 5.11: Comparación de las instrucciones de cadena

en el bucle.

5.2.2. El prefijo de instrucción `REP`

La familia 80x86 suministra un prefijo de instrucción especial[4] llamado `REP` que se puede usar con las instrucciones anteriores Este prefijo le dice a la CPU que repita la próxima instrucción de cadena un número especificado de veces. El registro ECX se usa para contar las iteraciones (tal como la instrucción `LOOP`). Usando el prefijo `REP`, el bucle en la Figura 5.8 (líneas 12 a 15) se podría reemplazar con una sola línea:

```
rep movsd
```

La Figura 5.10 muestra otro ejemplo que llena de ceros el contenido de un arreglo.

5.2.3. Comparación de las instrucciones de cadena

La Figura 5.11 muestra varias instrucciones de cadena nuevas que se pueden usar para comparar memoria con otra memoria o un registro. Ellas son útiles para comparar o buscar en arreglos. Ellas fijan el registro FLAGS

[4]un prefijo de instrucción no es una instrucción, es un byte especial que se coloca antes de una instrucción de cadena que modifica el comportamiento de la instrucción. Se usan otros prefijos para sobreescribir los segmentos por omisión de los accesos a memoria

```
1  segment .bss
2  array        resd 100
3
4  segment .text
5        cld
6        mov    edi, array    ; apuntador al inicio del arreglo
7        mov    ecx, 100      ; número de elementos
8        mov    eax, 12       ; número a buscar
9  lp:
10       scasd
11       je     found
12       loop   lp
13   ; código a ejecutar si no se encontró
14       jmp    onward
15 found:
16       sub    edi, 4         ; edi apunta ahora a 12 en array
17   ; código a ejecutar si se encontró
18 onward:
```

Figura 5.12: Ejemplo de búsqueda

`REPE`, `REPZ`	repite la instrucciónn mientras la bandera Z esté fija o un máximo de ECX veces
`REPNE`, `REPNZ`	repite la instrucción mientras la bandera Z esté borrada o un máximo de ECX veces

Figura 5.13: Prefijos de instrucción `REPx`

tal como la instrucción `CMP`. Las instrucciones `CMPSx` comparan el lugar de memoria correspondiente y `SCASx` comparan lugares de memoria en busca de un valor específico.

La Figura 5.12 muestra una porción de código que busca el número 12 en un arreglo de palabras dobles, la instrucción `SCASD` en la línea 10 siempre le suma 4 a EDI aún si el valor buscado se encuentra. Así si uno desea encontrar la dirección de donde está el 12 es necesario sustraer 4 de EDI (como lo hace la línea 16).

5.2.4. Prefijos de instrucción `REPx`

Hay varios prefijos de instrucción del tipo `REP` que se pueden usar con las instrucciones de comparación de cadenas. La Figura 5.13 muestra los

```
1  segment .text
2        cld
3        mov    esi, block1 ; dirección del primer bloque
4        mov    edi, block2 ; dirección del segundo bloque
5        mov    ecx, size   ; tamaño del bloque en bytes
6        repe   cmpsb       ; repita mientras la bandera Z esté fija
7        je     equal       ; Si Z está fija, los bloque son iguales
8     ; código para ejecutar si los bloques no son igual
9        jmp    onward
10 equal:
11    ; código para ejecutar si los bloques son iguales
12 onward:
```

Figura 5.14: Comparando bloques de memoria

dos nuevos prefijos y describe su operación. `REPE` y `REPZ` son sinónimos para el mismo prefijo (como `RPNE` y `REPNZ`) si la comparación repetida de la instrucción de cadena se detiene por el resultado de la comparación, el registro o registros índice se incrementan y ECX se decrementa; sin embargo, el registro FLAGS almacenará el estado que terminó la repetición. Así es posible usar la bandera Z para determinar si las comparaciones repetidas pararon por una comparación o porque ECX es cero.

¿Por qué uno no solo ve si ECX es cero después de repetir la comparación?

La Figura 5.14 muestra una porción código de ejemplo que determina si dos bloques de memoria son iguales. El `JE` en la línea 7 del ejemplo examina el resultado de la instrucción anterior. Si la comparación repetida se detiene porque encontró dos bytes diferentes, la bandera Z continuará borrada y no se hace el salto; sin embargo, si las comparaciones paran porque ECX se vuelve cero, la bandera Z estará fijada y el código salta a la etiqueta `equal`.

5.2.5. Ejemplo

Esta sección contiene un archivo fuente en ensamblador con varias funciones que implementan operaciones con arreglos usando instrucciones de cadena. Muchas de las funciones duplican las funciones de las bibliotecas conocidas de C.

memory.asm

```
1  global _asm_copy, _asm_find, _asm_strlen, _asm_strcpy
2
3  segment .text
4  ; función _asm_copy
5  ; copia un bloque de memoria
```

```
6  ; prototipo de C
7  ; void asm_copy( void * dest, const void * src, unsigned sz);
8  ; parámetros:
9  ;   dest - apuntador del búfer para copiar a
10 ;   src  - apuntador del búfer para copiar desde
11 ;   sz   - número de bytes a copiar
12
13 ; A continuación, se definen algunos símbolos útiles
14
15 %define dest [ebp+8]
16 %define src  [ebp+12]
17 %define sz   [ebp+16]
18 _asm_copy:
19         enter   0, 0
20         push    esi
21         push    edi
22
23         mov     esi, src        ; esi = dirección del búfer para copiar desde
24         mov     edi, dest       ; edi = dirección del búfer para copia a
25         mov     ecx, sz         ; ecx = número de bytes a copiar
26
27         cld                     ; borrar la bandera de dirección
28         rep     movsb           ; ejecute movsb ECX veces
29
30         pop     edi
31         pop     esi
32         leave
33         ret
34
35
36 ; function _asm_find
37 ; Busca en la memoria un byte dado
38 ; void * asm_find( const void * src, char target, unsigned sz);
39 ; parámetros
40 ;   src    - apuntador al lugar dónde buscar
41 ;   target - valor del byte a buscar
42 ;   sz     - tamaño en bytes de dónde se busca
43 ; valor de retorno
44 ;   si  target se encuentra, apunta a la primera ocurrencia de target
45 ;
46 ;   si no
47 ;     retorna NULL
```

```
48  ; NOTA: target es un valor de un byte, pero  se empuja en la pila como
49  ; una palabra doble. El byte se almacena en los 8 bits inferiores.
50  ;
51  %define src    [ebp+8]
52  %define target [ebp+12]
53  %define sz     [ebp+16]
54
55  _asm_find:
56          enter   0,0
57          push    edi
58
59          mov     eax, target     ; al tiene el valor a buscar
60          mov     edi, src
61          mov     ecx, sz
62          cld
63
64          repne   scasb           ; busca hasta que ECX == 0 or [ES:EDI] == AL
65
66          je      found_it        ; Si la bandera Z se fija, entonces se
67                                  ; encontró el valor
68          mov     eax, 0          ; Si no se encontró, retorna un apuntador NULL
69          jmp     short quit
70  found_it:
71          mov     eax, edi
72          dec     eax              ; Si se encuentra retorna (DI - 1)
73  quit:
74          pop     edi
75          leave
76          ret
77
78
79  ; function _asm_strlen
80  ; retorna el tamaño de una cadena
81  ; unsigned asm_strlen( const char * );
82  ; parámetro
83  ;   src - apuntador a la cadena
84  ; valor de retorno:
85  ;   número de caracteres en la cadena (sin contar el 0 final) (en EAX)
86
87  %define src [ebp + 8]
88  _asm_strlen:
89          enter   0,0
```

```
90          push    edi
91
92          mov     edi, src        ; edi = apuntador a la cadena
93          mov     ecx, 0FFFFFFFFh ; usa el mayor valor posible en ECX
94          xor     al,al           ; al = 0
95          cld
96
97          repnz   scasb           ; busca un 0 para terminar
98
99  ;
100 ; repnz tendrá un trayecto muy largo, así el tamaño será
101 ; FFFFFFFE - ECX, y no FFFFFFFF - ECX
102 ;
103         mov     eax,0FFFFFFFEh
104         sub     eax, ecx        ; length = 0FFFFFFFEh - ecx
105
106         pop     edi
107         leave
108         ret
109
110 ; función _asm_strcpy
111 ; copia una cadena
112 ; void asm_strcpy( char * dest, const char * src);
113 ; parámetros
114 ;   dest - apuntador a la cadena de destion
115 ;   src  - apuntador a la cadena origen
116 ;
117 %define dest [ebp + 8]
118 %define src  [ebp + 12]
119 _asm_strcpy:
120         enter   0,0
121         push    esi
122         push    edi
123
124         mov     edi, dest
125         mov     esi, src
126         cld
127 cpy_loop:
128         lodsb                   ; Caraga AL e incrementa SI
129         stosb                   ; almacena AL e incrementa DI
130         or      al, al          ; fija las banderas de condición
131         jnz     cpy_loop        ; si no más allá de terminar 0, continúa
```

```
132
133         pop     edi
134         pop     esi
135         leave
136         ret
```

memory.asm

memex.c

```
#include <stdio.h>

#define STR_SIZE 30
/* prototipos */

void asm_copy( void *, const void *, unsigned ) __attribute__((cdecl));
void * asm_find( const void *,
                 char target, unsigned ) __attribute__((cdecl));
unsigned asm_strlen( const char * ) __attribute__((cdecl));
void asm_strcpy( char *, const char * ) __attribute__((cdecl));

int main()
{
  char st1[STR_SIZE] = "test string";
  char st2[STR_SIZE];
  char * st;
  char  ch;

  asm_copy(st2, st1, STR_SIZE);  /* copia todos los 30 caracteres de la
                                    cadena*/
   printf (" %s\n", st2);

   printf ("Digite un caracter: ");  /* busca un byte en la cadena */
  scanf(" %c%*[^\n]", &ch);
  st = asm_find(st2, ch, STR_SIZE);
  if ( st )
     printf ("Encontrado en : %s\n", st);
  else
     printf ("No encontrado\n");

  st1[0] = 0;
   printf ("Digite una Cadena:");
  scanf(" %s", st1);
   printf ("len = %u\n", asm_strlen(st1));
```

```
  asm_strcpy( st2, st1 );      /* copia un dato significativo en string */
   printf (" %s\n", st2 );

  return 0;
}
```

memex.c

Capítulo 6

Punto flotante

6.1. Representación de punto flotante

6.1.1. Números binarios no enteros

Cuando se discutieron los sistemas numéricos en el primer capítulo, sólo se trataron los valores enteros. Obviamente, debe ser posible representar números no enteros en otras bases y así como en decimal. En decimal, los dígitos a la derecha del punto decimal tienen asociados potencias de 10 negativas.

$$0{,}123 = 1 \times 10^{-1} + 2 \times 10^{-2} + 3 \times 10^{-3}$$

No es sorprendente que los números binarios trabajen parecido.

$$0{,}101_2 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0{,}625$$

Esta idea se puede combinar con los métodos para enteros del Capítulo 1 para convertir un número.

$$110{,}011_2 = 4 + 2 + 0{,}25 + 0{,}125 = 6{,}375$$

Convertir de decimal a binario no es muy difícil. En general divida el número decimal en dos partes: entero y fracción. Convierta la parte entera usando a binario usando los métodos del Capítulo 1. La parte fraccionaria se convierte usando el método descrito a continuación:

Considere una fracción binaria con los bits etiquetados como $a, b, c, \ldots$ entonces el número se ve como:

$$0.abcdef\ldots$$

Multiplique el número por dos. La representación binaria del nuevo número será:

$$a.bcdef\ldots$$

$$
\begin{array}{rcl}
0{,}5625 \times 2 & = & 1{,}125 \\
0{,}125 \times 2 & = & 0{,}25 \\
0{,}25 \times 2 & = & 0{,}5 \\
0{,}5 \times 2 & = & 1{,}0
\end{array}
\qquad\qquad
\begin{array}{rcl}
\text{first bit} & = & 1 \\
\text{second bit} & = & 0 \\
\text{third bit} & = & 0 \\
\text{fourth bit} & = & 1
\end{array}
$$

Figura 6.1: Convertir 0.5625 a binario

$$
\begin{array}{rcl}
0{,}85 \times 2 & = & 1{,}7 \\
0{,}7 \times 2 & = & 1{,}4 \\
0{,}4 \times 2 & = & 0{,}8 \\
0{,}8 \times 2 & = & 1{,}6 \\
0{,}6 \times 2 & = & 1{,}2 \\
0{,}2 \times 2 & = & 0{,}4 \\
0{,}4 \times 2 & = & 0{,}8 \\
0{,}8 \times 2 & = & 1{,}6
\end{array}
$$

Figura 6.2: Convertir 0.85 a binario

Observe que el primer bit está ahora en el lugar del uno. Reemplace a con 0 y obtiene:

$$0.bcdef\ldots$$

y multiplique por dos otra vez para obtener:

$$b.cdef\ldots$$

Ahora el segundo bit (b) está en la primera posición. Este procedimiento se puede repetir hasta los bits que se deseen encontrar. La Figura 6.1 muestra un ejemplo real que convierte 0.5625 a binario. El método culmina cuando la parte fraccionaria es cero.

Como otro ejemplo, considere convertir 23.85 a binario. Es fácil convertir la parte entera ($23 = 10111_2$), pero ¿Qué pasa con la parte fraccionaria (0,85)? La Figura 6.2 muestra el comienzo de este cálculo. Si uno mira cuidadosamente los números, se encuentra con un bucle infinito. Esto significa

que 0.85 en un binario periódico (a diferencia de los decimales periódicos en base 10).[1] Hay un patrón de los números calculados. Mirando en el patrón, uno puede ver que $0{,}85 = 0{,}11\overline{0110}_2$. Así $23{,}85 = 10111{,}11\overline{0110}_2$.

Una consecuencia importante del cálculo anterior es que 23.85 no se puede representar *exactamente* en binario usando un número finito de bits. (Tal como $\frac{1}{3}$ no se puede representar en decimal con un número finito de dígitos). Como muestra este capítulo, las variables `float` y `double` en C son almacenados en binario. Así, valores como 23.85 no se pueden almacenar exactamente en estas variables. Sólo se puede almacenar una aproximación a 23.85

Para simplificar el hardware, los números de punto flotante se almacenan con un formato consistente. Este formato usa la notación científica (pero en binario, usando potencias de dos, no de diez). Por ejemplo 23.85 o $10111{,}11011001100110\ldots_2$ se almacenará como:

$$1{,}011111011001100110\ldots \times 2^{100}$$

(Donde el exponente (100) está en binario). Un número de punto flotante *normalizado* tiene la forma:

$$1.sssssssssssssss \times 2^{eeeeeee}$$

Dónde $1.sssssssssssss$ es la *mantisa* y *eeeeeeee* es el *exponente*.

6.1.2. Representación IEEE de punto flotante

El IEEE (Institute of Electrical and Electronic Engineer) es una organización internacional que ha diseñado formato binarios específicos para almacenar números de punto flotante. Este formato se usa en la mayoría (pero no todos) los computadores hechos hoy día. A menudo es soportado por el hardware de computador en sí mismo. Por ejemplo el coprocesador numérico (o matemático) de Intel (que está empotrado en todas las CPU desde el Pentium) lo usa. El IEEE define dos formatos con precisión diferente: precisión simple y doble. La precisión simple es usada por las variables `float` en C y la precisión doble es usada por la variable `double`.

El coprocesador matemático de Intel utiliza un tercer nivel de mayor precisión llamado *precisión extendida*. De hecho, todos los datos en el coprocesador en sí mismo están en esta precisión. Cuando se almacenan en memoria desde el coprocesador se convierte a precisión simple o doble automáticamente.[2] La precisión extendida usa un formato general ligeramente diferente

[1] No debería sorprenderse de que un número pudiera repetirse en una base, pero no en otra. Piense en $\frac{1}{3}$, es periódico en decimal, pero en ternario (base3) sería $0{,}1_3\overline{4}$.

[2] Algunos compiladores (como Borland) los tipos `long double` usa esta precisión extendida. Sin embargo, otros compiladores usan la precisión doble para `double` y `long double`. (Esto es permitido por ANSI C).

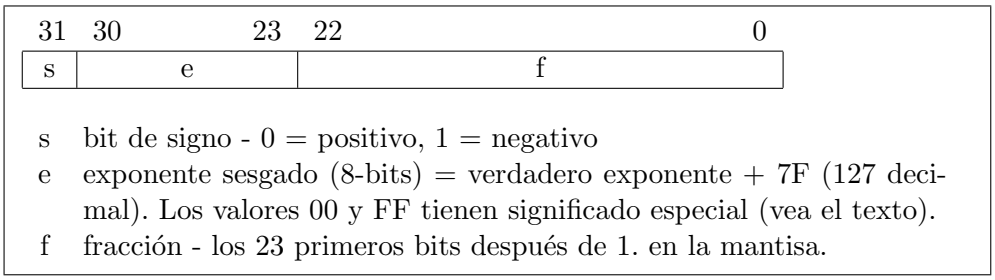


Figura 6.3: Precisión simple de la IEEE

que los formatos float y double de la IEEE y no será discutido acá.

Presición simple IEEE

El punto flotante de precisión simple usa 32 bits para codificar el número. Normalmente son exactos los primeros 7 dígitos decimales. Los números de punto flotante son almacenados en una forma mucho más complicada que los enteros. La Figura 6.3 muestra la forma básica del formato de precisión simple del IEEE. Hay varias peculiaridades del formato. Los números de punto flotante no usan la representación en complemento a 2 para los números negativos. Ellos usan la representación de magnitud y signo. El bit 31 determina el signo del número como se muestra en la figura.

El exponente binario no se almacena directamente. En su lugar, se almacena la suma del exponente y 7F en los bits 23 al 30. Este *exponente sesgado* siempre es no negativo.

La parte fraccionaria se asume que es normalizada (en la forma $1.sssssssss$). Ya que el primer bit es siempre uno, éste uno *no se almacena*. Esto permite el almacenamiento de un bit adicional al final y así se incrementa un poco la precisión. Esta idea se conoce como la *representación oculta del uno*.

Uno debería tener en cuenta que los bytes 41 BE CC CD se pueden interpretar de diferentes maneras dependiendo qué hace un programa con ellos. Como número de punto flotante de precisión simple, ellos representan 23,850000381*, pero como un entero, ellos representan* $1,103,023,309$*. La CPU no conoce cuál es la interpretación correcta.*

¿Cómo se podría almacenar 23.85? Primero es positivo, así el bit de signo es 0, ahora el exponente verdadero es 4, así que el exponente es $7F + 4 = 83_{16}$. Finalmente la fracción es (recuerde el uno de adelante está oculto). Colocando todo esto unido (para ayudar a aclarar las diferentes secciones del formato del punto flotante, el bit de signo y la fracción han sido subrayados y los bits se han agrupado en nibles):

$$\underline{0}\,100\ 0001\ 1\,\underline{011\ 1110\ 1100\ 1100\ 1100\ 1100}_2 = 41\text{BECCCC}_{16}$$

Esto no es exactamente 23.85 (ya que es un binario periódico). Si uno convierte el número anterior a decimal, uno encuentra que es aproximadamente 23,849998474. Este número es muy cercano a 23.85, pero no es exacto. En C, 23.85 no se puede representar exactamente. Ya que el bit del extremo izquierdo que fue truncado de la representación exacta es 1, el último bit es

$e = 0$	and	$f = 0$	representa el número cero (que no puede ser normalizado) Observe que hay un +0 y -0.
$e = 0$	and	$f \neq 0$	representa un *número sin normalizar*. Estos se discuten en la próxima sección.
$e = \text{FF}$	and	$f = 0$	representa infinito (∞). Hay infinitos positivo y negativo.
$e = \text{FF}$	and	$f \neq 0$	representa un resultado indefinido, Conocido como *NaN* (Not a Number).

Cuadro 6.1: Valores especiales de f y e

redondeado a 1. Así 23.85 podría ser representado como 41 BE CC CD en hexadecimal usando precisión simple. Convirtiendo esto a decimal el resultado y que es una aproximación ligeramente mejor de 23.85.

¿Cómo se podría representar -23.85? sólo cambie el bit de signo: C1BECCCD ¡*no* tome el complemento a 2!

Ciertas combinaciones de e y f tienen significado especial para los float IEEE. El Cuadro 6.1 describe estos valores especiales. Un infinito se produce por un desborde o por una división por cero. Un resultado indefinido se produce por una operación no válida como tratar de encontrar la raíz cuadrada de un número negativo, sumar dos infinitos, etc.

Los números de precisión simple están en el rango de $1{,}0 \times 2^{-126}$ ($\approx 1{,}1755 \times 10^{-35}$) to $1{,}11111\ldots \times 2^{127}$ ($\approx 3{,}4028 \times 10^{35}$).

Números sin normalizar

Los números sin normalizar se pueden usar para representar números con magnitudes más pequeñas que los normalizados (menores a $1{,}0 \times 2^{-126}$). Por ejemplo, considere el número $1{,}001_2 \times 2^{-129}$ ($\approx 1{,}6530 \times 10^{-39}$). En la forma normalizada, el exponente es muy pequeño. Sin embargo, se puede representar de una forma no normalizada como: $0{,}01001_2 \times 2^{-127}$. Para almacenar este número, el exponente sesgado se fija 0 (ver Cuadro 6.1) y la fracción es la mantisa completa del número escrito multiplicado por 2^{-127} (todos los bits son almacenados incluyendo el 1 a la izquierda del punto decimal). La representación de $1{,}001 \times 2^{-129}$ es entonces:

$$\underline{0}\,000\ 0000\ 0\,\underline{001\ 0010\ 0000\ 0000\ 0000\ 0000}$$

Doble precisión IEEE

La doble precisión IEEE usa 64 bits para representar números y normalmente son exactos los 15 primeros dígitos decimales. Como muestra la

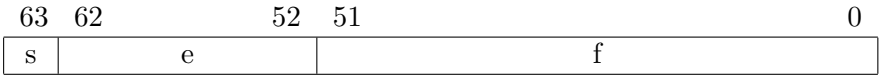


Figura 6.4: Precisión doble del IEEE

Figura 6.4, el formato básico es muy similar a la precisión simple. Se usan más bits para el exponente (ll) y la fracción (52) que para la precisión simple.

El gran rango para el exponente sesgado tiene dos consecuencias. La primera es que se calcula como la suma del exponente real y 3FF (1023) (no 7F como para la precisión simple). Segundo, se permite un gran rango de exponentes verdaderos (y así usa un gran rango de magnitudes). Las magnitudes de precisión doble van de 10^{-308} hasta 10^{308} aproximadamente.

En el campo de la fracción el responsable del incremento en el número de dígitos significativos para los valores dobles.

Como un ejemplo, considere nuevamente 23.85 otra vez. El exponente sesgado será 4 + 3FF = 403 en hexadecimal. Así la representación doble sería:

0 100 0000 0011 0111 1101 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1010

O 40 37 D9 99 99 99 99 9A en hexadecimal. Si uno convierte esto a decimal uno encuentra 23.8500000000000014 (¡hay 12 ceros!) que es una aproximación mucho mejor de 23.85.

La precisión doble tiene los mismos valores especiales que la precisión simple.[3] Los números no normalizados son muy similares también. La principal diferencia es que los números dobles sin normalizados usan 2^{-1023} en lugar de 2^{-127}.

6.2. Aritmética de punto flotante

La aritmética de punto flotante en un computador diferente que en la matemática continua. En matemáticas, todos los números pueden ser considerados exactos. Como se muestra en la sección anterior, en un computador muchos números no se pueden representar exactamente con un número finito de bits. Todos los cálculos se realizan con una precisión limitada. En los ejemplos de esta sección, se usarán números con una mantisa de 8 bits por simplicidad.

[3]La única diferencia es que para los valores de infinito e indefinido, el exponente sesgado es 7FF y no FF.

6.2.1. suma

Para sumar dos números de punto flotante, los exponentes deben ser iguales. Si ellos, no son iguales, entonces se deben hacer iguales, desplazando la mantisa del número con el exponente más pequeño. Por ejemplo, considere $10{,}375 + 6{,}34375 = 16{,}71875$ o en binario:

$$\begin{array}{rr} & 1{,}0100110 \times 2^3 \\ + & 1{,}1001011 \times 2^2 \\ \hline \end{array}$$

Estos dos números no tienen el mismo exponente así que se desplaza la mantisa para hacer iguales los exponentes y entonces sumar:

$$\begin{array}{rr} & 1.0100110 \times 2^3 \\ + & 0.1100110 \times 2^3 \\ \hline & 10.0001100 \times 2^3 \end{array}$$

Observe que el desplazamiento de $1{,}1001011 \times 2^2$ pierde el uno delantero y luego de redondear el resultado se convierte en $0{,}1100110 \times 2^3$. El resultado de la suma, $10{,}0001100 \times 2^3$ (o $1{,}00001100 \times 2^4$) es igual a $10000{,}110_2$ o 16.75. Esto *no* es igual a la respuesta exacta (16.71875) Es sólo una aproximación debido al error del redondeo del proceso de la suma.

Es importante tener en cuenta que la aritmética de punto flotante en un computador (o calculadora) es siempre una aproximación. Las leyes de las matemáticas no siempre funcionan con números de punto flotante en un computador. Las matemáticas asumen una precisión infinita que un computador no puede alcanzar. Por ejemplo, las matemáticas enseñan que $(a+b)-b = a$; sin embargo, esto puede ser exactamente cierto en un computador.

6.2.2. Resta

La resta trabaja muy similar y tiene los mismos problemas que la suma. Como un ejemplo considere $16{,}75 - 15{,}9375 = 0{,}8125$:

Subtraction works very similarly and has the same problems as addition. As an example, consider $16{,}75 - 15{,}9375 = 0{,}8125$:

$$\begin{array}{rr} & 1{,}0000110 \times 2^4 \\ - & 1{,}1111111 \times 2^3 \\ \hline \end{array}$$

Desplazando $1{,}1111111 \times 2^3$ da (redondeando arriba) $1{,}0000000 \times 2^4$

$$\begin{array}{rr} & 1{,}0000110 \times 2^4 \\ - & 1{,}0000000 \times 2^4 \\ \hline & 0{,}0000110 \times 2^4 \end{array}$$

$0{,}0000110 \times 2^4 = 0{,}11_2 = 0{,}75$ que no es exacto.

6.2.3. Multiplicación y división

Para la multiplicación, las mantisas son multiplicadas y los exponentes son sumados. Considere $10{,}375 \times 2{,}5 = 25{,}9375$:

$$\begin{array}{rr} & 1{,}0100110 \times 2^3 \\ \times & 1{,}0100000 \times 2^1 \\ \hline & 10100110 \\ + & 10100110 \\ \hline & 1{,}10011111000000 \times 2^4 \end{array}$$

Claro está, el resultado real podría ser redondeado a 8 bits para dar:

$$1{,}1010000 \times 2^4 = 11010{,}000_2 = 26$$

La división es más complicada, pero tiene problemas similares con errores de redondeo.

6.2.4. Instrucciones condicionales

El principal punto de esta sección es que los cálculos de punto flotante no son exactos. El programador necesita tener cuidado con esto. Un error común que el programador hace con números de punto flotante es compararlos asumiendo que el cálculo es exacto. Por ejemplo considere una función llamada f(x) que hace un cálculo complejo y un programa está tratando de encontrar las raíces de la función.[4]. Uno podría intentar usar la siguiente instrucción para mirar si x es una raíz:

```
if ( f(x) == 0.0 )
```

¿Pero si f(x) retorna 1×10^{-30}? Esto probablemente significa que x es una *muy* buena aproximación a una raíz verdadera; sin embargo, la igualdad será falsa. Puede no haber ningún valor de punto flotante IEEE de x que retorne cero exactamente, debido a los errores de redondeo en f(x).

Un método mucho mejor sería usar:

```
if ( fabs(f(x)) < EPS )
```

Dónde EPS es un macro definido para ser un valor positivo muy pequeño (como 1×10^{-10}). Esto es cierto si f(x) está muy cercano a cero. En general, para comparar un valor de punto flotante (digamos x) con otro (y) use:

```
if ( fabs(x - y)/fabs(y) < EPS )
```

[4]Una raíz de una función es un valor x tal que $f(x) = 0$

6.3. El coprocesador numérico

6.3.1. Hardware

Los primeros procesadores Intel no tenían soporte de hardware para las operaciones de punto flotante. Esto no significa que ellos no podían efectuar operaciones de punto flotante. Esto sólo significa que ellas se realizaban por procedimientos compuestos de muchas instrucciones que no son de punto flotante. Para estos primeros sistemas, Intel suministraba un circuito integrado adicional llamado *coprocesador matemático*. Un coprocesador matemático tiene instrucciones de máquina que realizan instrucciones de punto flotante mucho más rápido que usando procedimientos de software (en los primeros procesadores se realizaban al menos 10 veces más rápido). El coprocesador para el 8086/8088 fue llamado 8087. Para el 80286 era el 80287 y para el 80386 un 80387. El procesador 80486DX integró el coprocesador matemático en el 80486 en sí mismo[5]. Desde el Pentium, todas las generaciones del 80x86 tienen un coprocesador matemático empotrado; sin embargo él todavía se programa como si fuera una unidad separada. Inclusive en los primeros sistemas sin un coprocesador se puede instalar un software que emula el coprocesador matemático. Estos emuladores se activan automáticamente cuando un programa ejecuta una instrucción del coprocesador y corre un procedimiento que obtiene los mismos resultados que el coprocesador (mucho más lento claro está).

El coprocesador numérico tiene ocho registros de punto flotante. Cada registro almacena 80 bits. Los números de punto flotante se almacenan en estos registros *siempre* como números de 80 bits de precisión extendida. Los registros se llaman `ST0`, `ST1`, ... `ST7`. Los registros de punto flotante se usan diferente que los registros enteros en la CPU. Los registros de punto flotante están organizados como una pila. Recuerde que una *pila* es una lista LIFO (*Last In Firist Out*). `ST0` siempre se refiere al valores el tope de la pila. Todos los números nuevos se añaden al tope de la pila. Los números existentes se empujan en la pila para dejarle espacio al nuevo número.

Hay también un registro de estado en el coprocesador numérico. Tiene varias banderas. Sólo las 4 banderas usadas en comparaciones serán estudiadas. C_0, C_1, C_2 and C_3. El uso de ellas se discutirá luego.

6.3.2. Instrucciones

Para distinguir fácilmente las instrucciones normales de la CPU de las del coprocesador, todos los nemónicos del coprocesador comienzan por `F`.

[5]Sin embargo el 80486SX no tiene el un coprocesador integrado. Existía un chip separado para estas máquinas (el 80487SX)

Carga y almacenamiento

Hay varias instrucciones que cargan datos en el tope de la pila de registro del coprocesador:

`FLD` *`fuente`*	Carga un número de punto flotante de la memoria en el tope de la pila. *fuente* puede ser un número de precisión simple doble o extendida o un registro del coprocesador
`FILD` *`fuente`*	Lee un *entero* de memoria, lo convierte a punto flotante y almacena el resultado en el tope de la pila. *fuente* puede ser una palabra, palabra doble o una palabra cuádruple.
`FLD1`	almacena un uno en el tope de la pila.
`FLDZ`	almacena un cero en el tope de la pila.

Hay también varias instrucciones que almacenan datos de la pila en memoria. Algunas de estas instrucciones también *sacan* el número de la pila.

`FST` *`dest`*	Almacena el tope de la pila (`ST0`) en memoria. El *destino* puede ser un número de precisión simple o doble o un registro de coprocesador.
`FSTP` *`dest`*	Almacena el tope de la pila en la memoria tal como `FST`; sin embargo luego de que se almacena el número, su valor se saca de la pila. El *destino* puede ser un número de precisión simple o doble o un registro del coprocesador.
`FIST` *`dest`*	Almacena el valor del tope de la pila convertido a entero en memoria. El *destino* puede ser una palabra o palabra doble. La pila no sufre ningún cambio. Cómo se convierte el número de punto flotante a entero depende de algunos bits de la *palabra de control* del coprocesador. Este es un registro especial (no de punto flotante) que controla cómo trabaja el coprocesador. Por omisión, la palabra de control se inicia de tal manera que redondea al entero más cercano cuando se convierte a entero. Sin embargo las instrucciones `FSTCW` (*Store Control Word*) y `FDLCW` (*Load Control Word*) se pueden usar para cambiar este comportamiento.
`FISTP` *`dest`*	Lo mismo que `FIST`, excepto por dos cosas. El tope de la pila se saca y el *destino* también puede ser una palabra cuádruple.

Hay otras dos instrucciones que pueden mover o quitar datos de la pila en sí misma.

`FXCH ST`*`n`*	intercambia los valores en `ST0` y `ST`*`n`* en la pila (donde n es el número del registro de 1 a 7).
`FFREE ST`*`n`*	libera un registro en la pila, marcando el registro como no usado o vacío.

```
1  segment .bss
2  array        resq SIZE
3  sum          resq 1
4
5  segment .text
6        mov    ecx, SIZE
7        mov    esi, array
8        fldz                  ; ST0 = 0
9  lp:
10       fadd   qword [esi]    ; ST0 += *(esi)
11       add    esi, 8         ; se mueve al próximo dobule
12       loop   lp
13       fstp   qword sum      ; almacena el resultado en sum
```

Figura 6.5: Ejemplo sumar arreglo

Suma y resta

Cada una de las instrucciones de suma calcula la suma de `ST0` y otro operando, el resultado siempre se almacena en un registro del coprocesador.

`FADD` *`fuente`*	`ST0 +=` *`fuente`*. *fuente* puede ser cualquier registro del coprocesador o un número de precisión simple o doble en memoria.
`FADD` *`dest`*`, ST0`	*`dest`* `+= ST0`. El *destino* puede ser cualquier registro del coprocesador
`FADDP` *`dest`* o `FADDP` *`dest`*`, ST0`	*`dest`* `+= ST0` y luego se saca de la pila. El *destino* puede ser cualquier registro del coprocesador.
`FIADD` *`fuente`*	`ST0 += (float)` *`fuente`*. suma un entero con `ST0`. *fuente* debe ser una palabra o una palabra doble en memoria.

Hay el doble de instrucciones de resta que de suma porque el orden de los operandos es importante. (¡$a + b = b + a$, pero $a - b \neq b - a$!). Por cada instrucción, hay una alterna que resta en el orden inverso. Esas instrucciones al revés todas culminan con `R` o `RP`. La Figura 6.5 muestra un pequeño código que suma los elementos de un arreglo de dobles. En las líneas 10 y 13, uno debe especificar el tamaño del operando de memoria. De otra forma el ensamblador no conocería si el operando de memoria era un float (dword) o double (qword).

`FSUB` *fuente*	`ST0 -=` *fuente*. *fuente* puede ser cualquier registro del coprocesador o un número de presición doble o simple en memoria.
`FSUBR` *fuente*	`ST0 =` *fuente* `- ST0`. *fuente* puede ser cualquier registro del coprocesador o un número de presición doble o simple en memoria.
`FSUB` *dest*`, ST0`	*dest* `-= ST0`. *dest* puede ser cualquier registro del coprocesador.
`FSUBR` *dest*`, ST0`	*dest* `= ST0 -` *dest*. *dest* puede ser cualquier registro del coprocesador.
`FSUBP` *dest* o `FSUBP` *dest*`, ST0`	*dest* `-= ST0` y luego sale de la pila. *dest* puede ser cualquier registro del coprocesador.
`FSUBRP` *dest* o `FSUBRP` *dest*`, ST0`	*dest* `= ST0 -` *dest* y luego sale de la pila. *dest* puede ser cualquier registro del coprocesador.
`FISUB` *fuente*	`ST0 -= (float)` *fuente*. Resta un entero de `ST0`. *fuente* debe ser una palabra o palabra doble en memoria.
`FISUBR` *fuente*	`ST0 = (float)` *fuente* `- ST0`. Resta `ST0` de un entero. *fuente* debe ser una palabra o palabra doble en memoria.

Multiplicación y división

La instrucción de multiplicación son totalmente análogas a las instrucciones de suma.

`FMUL` *fuente*	`ST0 *=` *fuente*. *fuente* puede ser cualquier registro del coprocesador o un número de precisión simple o doble en memoria.
`FMUL` *dest*`, ST0`	*dest* `*= ST0`. *dest* puede ser cualquier registro del coprocesador.
`FMULP` *dest* o `FMULP` *dest*`, ST0`	*dest* `*= ST0` y luego sale de la pila. *dest* puede ser cualquier registro del coprocesador.
`FIMUL` *fuente*	`ST0 *= (float)` *fuente*. Multiplica un entero con `ST0`. *fuente* debe ser una palabra o palabra doble en memoria.

No es sorprendente que las instrucciones de división sean análogas a las instrucciones de resta. La división por cero da infinito.

`FDIV` *fuente*	`ST0 /=` *fuente*. *fuente* puede ser cualquier registro del coprocesador o un número de precisión simple o doble en memoria.
`FDIVR` *fuente*	`ST0 =` *fuente* `/ ST0`. *fuente* puede ser cualquier registro del coprocesador o un número de precisión simple o doble en memoria.
`FDIV` *dest*`, ST0`	*dest* `/= ST0`. *dest* puede ser cualquier registro del coprocesador.
`FDIVR` *dest*`, ST0`	*dest* `= ST0 /` *dest*. *dest* puede ser cualquier registro del coprocesador.
`FDIVP` *dest* o `FDIVP` *dest*`, ST0`	*dest* `/= ST0` y luego sale de la pila. *dest* puede ser cualquier registro del coprocesador.
`FDIVRP` *dest* o `FDIVRP` *dest*`, ST0`	*dest* `= ST0 /` *dest* y luego sale de la pila. *dest* puede ser cualquier registro del coprocesador.
`FIDIV` *fuente*	`ST0 /= (float)` *fuente*. Divide `ST0` por un entero. *fuente* debe ser una palabra o palabra doble en memoria.
`FIDIVR` *fuente*	`ST0 = (float)` *fuente* `/ ST0`. Divide un entero por `ST0`. *fuente* debe ser una palabra o palabra doble en memoria.

Comparaciones

El coprocesador también realiza comparaciones de números de punto flotante. La familia de instrucciones `FCOM` hacen esta operación.

`FCOM` *fuente*	compara `ST0` y *fuente*. *fuente* puede ser un registro del coprocesador o un float o un double en memoria.
puede ser un `FCOMP` *fuente*	compara `ST0` y *fuente*, luego sale de la pila. *fuente* puede ser un registro del coprocesador o un float o double en memoria.
`FCOMPP`	compara `ST0` y `ST1`, y luego sale de la pila dos veces.
`FICOM` *fuente*	compara `ST0` y `(float)` *fuente*. *fuente* puede ser una palabra o palabra doble en memoria.
`FICOMP` *fuente*	compara `ST0` y `(float)`*fuente*, y luego sale de la pila. *fuente* puede ser una palabra o palabra doble entera en memoria.
`FTST`	compara `ST0` and 0.

Estas instrucciones cambian los bits C_0, C_1, C_2 y C_3 del registro de estado del coprocesador. Desafortunadamente no es posible que la CPU acceda a estos bits directamente. Las instrucciones de salto condicional usan el registro FLAGS, no el registro de estado del coprocesador. Sin embargo es relativamente fácil transferir los bits de la palabra de estado a los bits correspondientes del registro FLAGS usando algunas instrucciones nuevas.

```
1  ;     if ( x > y )
2  ;
3        fld    qword [x]       ; ST0 = x
4        fcomp  qword [y]       ; compara ST0 e y
5        fstsw  ax              ; mueve los bits C a FLAGS
6        sahf
7        jna    else_part       ; si x no es mayor que y,
8                                ; vaya a else_part
9  then_part:
10       ; código para parte de entonces
11       jmp    end_if
12 else_part:
13       ; código para parte si no
14 end_if:
```

Figura 6.6: Ejemplo de comparación

`FSTSW` *dest*	Almacena la palabra de estado del coprocesador in memoria o en el registro AX.
`SAHF`	Almacena el registro AH en el registro FLAGS.
`LAHF`	Carga el registro AH con los bits del registro FLAGS.

La Figura 6.6 muestra un ejemplo cortico. Las líneas 5 y 6 transfieren los bits C_0, C_1, C_2 y C_3 de la palabra de estado del coprocesador al registro FLAGS. Los bits son transferidos tal que ellos son análogos al resultado de una comparación de dos enteros *sin signo*. Este es el porque la línea 7 usa la instrucción JNA.

El Pentium Pro (y procesadores posteriores II y III) soportan 2 nuevos operadores de comparación que directamente modifican el registro FLAGS de la CPU.

`FCOMI` *fuente*	compara `ST0` y *fuente*. *fuente* debe ser un registro del coprocesador.
`FCOMIP` *fuente*	COMPARA `ST0` y *fuente*, y luego sale de la pila. *fuente* debse ser un registro del coprocesador.

La Figura 6.7 muestra una subrutina de ejemplo que encuentran el máximo de dos números tipo double usando la instrucción `FCOMIP`. No confundan esta instrucción con la función de comparación de enteros (`FICOMP` y `FICOM`).

Instrucciones miscelaneas

Esta sección cubre las otras instrucciones que suministra el procesador:

`FCHS`	`ST0 = - ST0` cambia el bit de signo de `ST0`
`FABS`	`ST0` $= \lvert$`ST0`$\rvert$ Toma el valor absoluto de `ST0`
`FSQRT`	`ST0` $= \sqrt{\texttt{ST0}}$ Toma la raíz cuadrada de `ST0`
`FSCALE`	`ST0` $=$ `ST0` $\times 2^{\lfloor \texttt{ST1} \rfloor}$ multiplica `ST0` por una potencia de dos rápidamente. `ST1` no se quita de la pila del coprocesador . La Figure 6.8 muestra un ejemplo de cómo usar esta instrucción.

6.3.3. Ejemplos

6.3.4. Fórmula cuadrática

El primer ejemplo muestra cómo se puede programar la fórmula cuadrática en ensamblador. Recuerde que la fórmula cuadrática calcula la solución de una ecuación cuadrática:

$$ax^2 + bx + c = 0$$

La fórmula en sí misma da dos soluciones para x: x_1 y x_2.

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

La expresión dentro de la raíz cuadrada ($b^2 - 4ac$) es llamada *discriminante*, su valor es útil para determinar las 3 posibilidades para la solución:

1. Hay solo un real en la solución degenerada. $b^2 - 4ac = 0$
2. Hay dos soluciones reales. $b^2 - 4ac > 0$
3. Hay dos soluciones complejas. $b^2 - 4ac < 0$

Este es un pequeño programa en C que usa la subrutina en ensamblador.

quadt.c

```
#include <stdio.h>

int quadratic( double, double, double, double *, double *);

int main()
{
  double a,b,c, root1, root2;

  printf("Enter a, b, c: ");
  scanf(" %lf %lf %lf", &a, &b, &c);
  if (quadratic( a, b, c, &root1, &root2) )
```

```
    printf ("roots: %.10g %.10g\n", root1, root2);
  else
    printf ("No real roots\n");
  return 0;
}
```

quadt.c

A continuación está la rutina en ensamblador.

quad.asm

```
; función quadratic
; Halla la solución de la ecuación cuadrática:
;       a*x^2 + b*x + c = 0
; Prototipo de C:
;   int quadratic( double a, double b, double c,
;                  double * root1, double *root2 )
; Parámetros:
;   a, b, c - Coeficientes de la ecuación cuadrática (ver arriba)
;   root1   - Apuntador al double que almacena la primera raíz
;   root2   - Apuntador al double que almacena la segunda raíz
; Valor de retorno:
;   devuelve 1 si las raíces son reales si no 0

%define a               qword [ebp+8]
%define b               qword [ebp+16]
%define c               qword [ebp+24]
%define root1           dword [ebp+32]
%define root2           dword [ebp+36]
%define disc            qword [ebp-8]
%define one_over_2a     qword [ebp-16]

segment .data
MinusFour       dw      -4

segment .text
        global  _quadratic
_quadratic:
        push    ebp
        mov     ebp, esp
        sub     esp, 16         ; asigna 2 doubles (disc & one_over_2a)
        push    ebx             ; debe guardar el valor original de ebx

```

```
33          fild    word [MinusFour]; pila -4
34          fld     a               ; pila: a, -4
35          fld     c               ; pila: c, a, -4
36          fmulp   st1             ; pila: a*c, -4
37          fmulp   st1             ; pila: -4*a*c
38          fld     b
39          fld     b               ; pila: b, b, -4*a*c
40          fmulp   st1             ; pila: b*b, -4*a*c
41          faddp   st1             ; pila: b*b - 4*a*c
42          ftst                    ; prueba con 0
43          fstsw   ax
44          sahf
45          jb      no_real_solutions ; if disc < 0, no hay soluciones reales
46          fsqrt                   ; pila: sqrt(b*b - 4*a*c)
47          fstp    disc            ; almacena y saca de la pila
48          fld1                    ; pila: 1.0
49          fld     a               ; pila: a, 1.0
50          fscale                  ; pila: a * 2^(1.0) = 2*a, 1
51          fdivp   st1             ; pila: 1/(2*a)
52          fst     one_over_2a     ; pila: 1/(2*a)
53          fld     b               ; pila: b, 1/(2*a)
54          fld     disc            ; pila: disc, b, 1/(2*a)
55          fsubrp  st1             ; pila: disc - b, 1/(2*a)
56          fmulp   st1             ; pila: (-b + disc)/(2*a)
57          mov     ebx, root1
58          fstp    qword [ebx]     ; almacena en *root1
59          fld     b               ; pila: b
60          fld     disc            ; pila: disc, b
61          fchs                    ; pila: -disc, b
62          fsubrp  st1             ; pila: -disc - b
63          fmul    one_over_2a     ; pila: (-b - disc)/(2*a)
64          mov     ebx, root2
65          fstp    qword [ebx]     ; almacena en *root2
66          mov     eax, 1          ; valor de retorno es 1
67          jmp     short quit
68
69  no_real_solutions:
70          mov     eax, 0          ; valor de retorno es 0
71
72  quit:
73          pop     ebx
74          mov     esp, ebp
```

```
75          pop     ebp
76          ret
```
quad.asm

6.3.5. Leer arreglos de archivos

Este ejemplo, es una rutina en ensamblador que lee doubles de un archivo. Sigue un pequeño programa de prueba en C.

readt.c

```
/*
 * Este programa prueba el procedimiento en ensamblador de 32 bits read_doubles() .
 * Lee doubles de stdin . (use la redireccióon para leer desde un archivo .)
 */
#include <stdio.h>
extern int read_doubles( FILE *, double *, int );
#define MAX 100

int main()
{
  int i,n;
  double a[MAX];

  n = read_doubles(stdin , a, MAX);

  for( i=0; i < n; i++ )
     printf (" %3d %g\n", i, a[i]);
  return 0;
}
```

readt.c

A continuación la rutina en ensamblador

read.asm

```
1  segment .data
2  format  db      "%lf", 0        ; formato para fscanf()
3
4  segment .text
5          global  _read_doubles
6          extern  _fscanf
7
8  %define SIZEOF_DOUBLE   8
```

```
9  %define FP              dword [ebp + 8]
10 %define ARRAYP          dword [ebp + 12]
11 %define ARRAY_SIZE      dword [ebp + 16]
12 %define TEMP_DOUBLE     [ebp - 8]
13
14 ;
15 ; función _read_doubles
16 ; prototipo de C:
17 ;   int read_doubles( FILE * fp, double * arrayp, int array_size );
18 ; Esta función lee dobles de un archivo de texto hasta EOF o hasta
19 ; que se llene el arreglo
20 ; Parámetros:
21 ;   fp         - apuntador FILE pointer a leer desde (se debe abrir para entrada)
22 ;   arrayp     - apuntador a un arreglo de doubles para leer en él
23 ;   array_size - número de elementos en el arreglo
24 ; Valor de retorno:
25 ;   número de doubles almacenado en el arreglo (en EAX)
26
27 _read_doubles:
28         push    ebp
29         mov     ebp,esp
30         sub     esp, SIZEOF_DOUBLE      ; define un double en la pila
31
32         push    esi                     ; guarda esi
33         mov     esi, ARRAYP             ; esi = ARRAYP
34         xor     edx, edx                ; edx = índice del arreglo (inicialmente en 0)
35
36 while_loop:
37         cmp     edx, ARRAY_SIZE         ; si edx < ARRAY_SIZE?
38         jnl     short quit              ; si no, salir del bucle
39 ;
40 ; llama a fscanf() para leer un double en TEMP_DOUBLE
41 ; fscanf() podría cambiar  edx y así guardarlo
42 ;
43         push    edx                     ; guarda edx
44         lea     eax, TEMP_DOUBLE
45         push    eax                     ; push &TEMP_DOUBLE
46         push    dword format            ; push &format
47         push    FP                      ; push el apuntador al archivo
48         call    _fscanf
49         add     esp, 12
50         pop     edx                     ; restaura edx
```

```
51          cmp     eax, 1                  ; fscanf  retornó 1?
52          jne     short quit              ; si no, salir del bucle
53
54  ;
55  ; copia TEMP_DOUBLE en ARRAYP[edx]
56  ; (Los ocho bytes del double son copiados por las dos copias de 4 bytes)
57  ;
58          mov     eax, [ebp - 8]
59          mov     [esi + 8*edx], eax      ; primero copia los 4 bytes inferiores
60          mov     eax, [ebp - 4]
61          mov     [esi + 8*edx + 4], eax  ; ahora copia los 4 bytes superiores
62
63          inc     edx
64          jmp     while_loop
65
66  quit:
67          pop     esi                     ; restaura esi
68
69          mov     eax, edx                ; almacena el valor de retorno en eax
70
71          mov     esp, ebp
72          pop     ebp
73          ret
```

read.asm

6.3.6. Hallar primos

Este último ejemplo halla números primos una vez más. Esta implementación es más eficiente que la anterior. Almacena los primos que ha encontrado en un arreglo y solo divide por los primos que ha encontrado antes, en lugar de cada número impar, para encontrar nuevos primos.

Otra diferencia es que el programa calcula la raíz cuadrada del número examinado para buscar el próximo primo para determinar en qué punto parar la búsqueda de factores. Altera la palabra de control de coprocesador tal que cuando almacena la raíz cuadrada como un entero, y la trunca en lugar de redondearla. Esto es controlado por los bits 10 y 11 de la palabra de control. Estos bits se llaman los bits RC (Raunding Control). Si ellos son ambos 0 (por omisión) el coprocesador redondea cuando convierte a entero. Si ellos son ambos 1, el coprocesador trunca las conversiones enteras. Observe que la rutina se cuida de guardar la palabra de control original y restaurarla antes de retornar.

A continuación el programa en C:

fprime.c

```
#include <stdio.h>
#include <stdlib.h>
/*
 * función find_primes
 * Halla los números primos indicados
 * Parámetros:
 *    a – arreglo que almacena los primos
 *    n – cuántos primos encontrar
 */
extern void find_primes ( int * a, unsigned n );

int main()
{
  int status;
  unsigned i;
  unsigned max;
  int * a;

   printf ("¿Cuántos primos desea encontrar Ud.? ");
  scanf(" %u", &max);

  a = calloc( sizeof(int), max);

  if ( a ) {

    find_primes (a,max);

    /* imprime los 20 últimos primos encontrados */
    for(i= ( max > 20 ) ? max - 20 : 0; i < max; i++ )
      printf (" %3d %d\n", i+1, a[i]);

    free (a);
    status = 0;
  }
  else {
     fprintf ( stderr , "No se puede crear el arreglo de %u ints\n", max);
    status = 1;
  }

  return status;
}
```

fprime.c

A continuación la rutina en ensamblador:

prime2.asm

```
segment .text
        global  _find_primes
;
; function find_primes
; Encuentra el número indicado de primos
; Parámetros:
;   array  - arreglo que almacena los primos
;   n_find - cuántos primos encontrar
; Prototipo de C
;extern void find_primes( int * array, unsigned n_find )
;
%define array         ebp + 8
%define n_find        ebp + 12
%define n             ebp - 4           ; número de primos a encontrar
%define isqrt         ebp - 8           ; el piso de la raíz cudrada de guess
%define orig_cntl_wd  ebp - 10          ; palabra original de control
%define new_cntl_wd   ebp - 12          ; nueva palabra de control

_find_primes:
        enter   12,0                    ; Hace espacio para las variables locales

        push    ebx                     ; guarda las posibles variables registro
        push    esi

        fstcw   word [orig_cntl_wd]     ; toma la palabra de control actual
        mov     ax, [orig_cntl_wd]
        or      ax, 0C00h               ; fija el redondeo de los bits a 11 (truncar)
        mov     [new_cntl_wd], ax
        fldcw   word [new_cntl_wd]

        mov     esi, [array]            ; esi apunta a array
        mov     dword [esi], 2          ; array[0] = 2
        mov     dword [esi + 4], 3      ; array[1] = 3
        mov     ebx, 5                  ; ebx = guess = 5
        mov     dword [n], 2            ; n = 2
;
; Este bucle externo encuentra un nuevo primo en cada iteración,
; que se añade al final del arreglo. A diferencia del primer programa
```

```
39 ; que halla primos, esta función no determina los primos dividiendo
40 ; por todos los números impares. Sólo divide  por los números primos
41 ; que ya se han encontrado. (Esta es la razón por la cual ellos se han
42 : almacenado en el arreglo)
43
44 while_limit:
45         mov     eax, [n]
46         cmp     eax, [n_find]           ; while ( n < n_find )
47         jnb     short quit_limit
48
49         mov     ecx, 1                  ; ecx se usa como índice del arreglo
50         push    ebx                     ; almacena guess en la pila
51         fild    dword [esp]             ; carga guess en la pila del coprocesador
52         pop     ebx                     ; saca guess de la pila
53         fsqrt                           ; halla sqrt(guess)
54         fistp   dword [isqrt]           ; isqrt = floor(sqrt(quess))
55
56 ; Este b1ucle interno divide guess por los números primos ya encontrados
57 ; hasta que encuentre un factor primo de guess (que significa que guess
58 ; no es primo) o hasta que número primo a dividir sea más grande que
59 ; floor(sqrt(guess))
60 ;
61 while_factor:
62         mov     eax, dword [esi + 4*ecx]        ; eax = array[ecx]
63         cmp     eax, [isqrt]                    ; while ( isqrt < array[ecx]
64         jnbe    short quit_factor_prime
65         mov     eax, ebx
66         xor     edx, edx
67         div     dword [esi + 4*ecx]
68         or      edx, edx                        ; && guess % array[ecx] != 0 )
69         jz      short quit_factor_not_prime
70         inc     ecx                             ; intenta el próximo primo
71         jmp     short while_factor
72
73 ;
74 ; found a new prime !
75 ;
76 quit_factor_prime:
77         mov     eax, [n]
78         mov     dword [esi + 4*eax], ebx        ; suma al final de arreglo
79         inc     eax
80         mov     [n], eax                        ; inc n
```

```
81
82  quit_factor_not_prime:
83          add     ebx, 2                          ; intenta con el impar siguiente
84          jmp     short while_limit
85
86  quit_limit:
87
88          fldcw   word [orig_cntl_wd]             ; restaura la palabra de control
89          pop     esi                             ; restaura las variables de registro
90          pop     ebx
91
92          leave
93          ret
```

prime2.asm

```
1  global _dmax
2
3  segment .text
4  ; función _dmax
5  ; retorna el mayor de dos argumentos double
6  ; Prototipo de C
7  ; double dmax( double d1, double d2 )
8  ; Parámetros:
9  ;   d1   - primer double
10 ;   d2   - segundo double
11 ; Valor de retorno:
12 ;   El mayor de d1 y d2 (en ST0)
13 %define d1   ebp+8
14 %define d2   ebp+16
15 _dmax:
16         enter   0, 0
17
18         fld     qword [d2]
19         fld     qword [d1]          ; ST0 = d1, ST1 = d2
20         fcomip  st1                 ; ST0 = d2
21         jna     short d2_bigger
22         fcomp   st0                 ; pop d2 de la pila
23         fld     qword [d1]          ; ST0 = d1
24         jmp     short exit
25 d2_bigger:                          ; si d2 is max, no hace nada
26 exit:
27         leave
28         ret
```

Figura 6.7: Ejemplo de FCOMIP

```
1  segment .data
2  x            dq  2.75          ; convertido a formato double
3  five         dw  5
4
5  segment .text
6        fild   dword [five]      ; ST0 = 5
7        fld    qword [x]         ; ST0 = 2.75, ST1 = 5
8        fscale                   ; ST0 = 2.75 * 32, ST1 = 5
```

Figura 6.8: Ejemplo de `FSCALE`

Capítulo 7

Estructuras y C++

7.1. Estructuras

7.1.1. Introducción

Las estructuras se usan en C para agrupar datos relacionados en una variable compuesta. Esta técnica tiene varias ventajas:

1. Clarifica el código mostrando que los datos definidos en la estructura están relacionados íntimamente.

2. Simplifica el paso de datos a las funciones. En lugar de pasar muchas variables separadas, ellas se pueden pasar como una variable.

3. Incrementar la *localidad* del código.[1]

Desde el punto de vista del ensamblador una estructura se puede considerar como un arreglo con elementos de *diferente* tamaño. Los elementos de los arreglos reales son siempre del mismo tipo y mismo tamaño. Esta característica es la que le permite a uno calcular la dirección de cualquier elemento conociendo la dirección de inicio del arreglo, el tamaño de los elementos y el índice del elemento que se desea.

En una estructura los elementos no tienen que tener el mismo tamaño (normalmente no). Por esto cada elemento de una estructura debe ser definido explícitamente mediante un *nombre*, en lugar de un índice numérico.

En ensamblador los elementos de una estructura se accederán de una manera parecida que los elementos de un arreglo. Para acceder a un elemento uno debe conocer la dirección de inicio de la estructura y el *desplazamiento relativo* desde el comienzo de la estructura, sin embargo, a diferencia de los arreglos donde este desplazamiento se puede calcular por el índice del

[1] vea la sección de manejo de memoria virtual de cualquier libro de texto de sistemas operativos para la discusión del término.

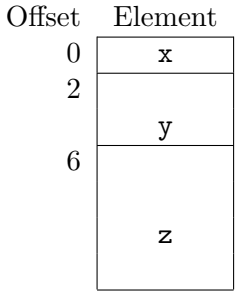


Figura 7.1: Estructura S

elemento, al elemento de una estructura el compilador le asigna un desplazamiento.

Por ejemplo considere la siguiente estructura:

```
struct S {
  short int x;     /* entero de 2 bytes */
  int       y;     /* entero de 4 bytes */
  double    z;     /* float de 8 bytes  */
};
```

La Figura 7.1 muestra cómo una variable de tipo `S` podría verse en la memoria del computador. La norma ANSI dicta que los elementos de una estructura son colocados en memoria en el mismo orden que ellos son definidos en la `estructura`. También dicta que el primer elemento está al comienzo de la estructura (desplazamiento cero). También define otro macro útil en el archivo de cabecera `stddef.h` llamado `offsetof()`. Este macro calcula y retorna el desplazamiento de cualquier elemento de una estructura. Este macro toma dos parámetros, el primero es el nombre de *tipo* de la estructura, el segundo del nombre del elemento al cual encontrarle el desplazamiento. Así el resultado de `offsetof(S,y)` podría ser 2 según la Figura 7.1.

7.1.2. Alineamiento de la memoria

Recuerde que una dirección está en los límites de una palabra doble si es divisible por cuatro.

Si uno usa el macro `offsetof` para encontrar el desplazamiento de `y` usando el compilador *gcc* uno encontrará que retorna 4 y no 2. ¿Por qué? Porque *gcc* (y muchos otros compiladores) alinean por omisión las variables a los límites de palabras dobles. En el modo protegido de 32 bits, la CPU lee la memoria más rápido si el dato comienza en el límite de una palabra doble. La Figura 7.2 muestra cómo se ve realmente la estructura `S` usando *gcc*. El compilador inserta dos bytes no usados en la estructura para alinear `y`(y `z`) en un límite de palabra doble. Esto muestra por qué es buena idea usar `offsetof` para calcular los desplazamientos en lugar de calcularlos uno mismo cuando se usan estructuras definidas en C.

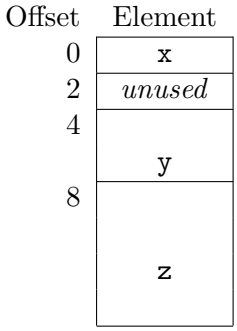


Figura 7.2: Estructura S

Claro está, si la estructura solo se usa en ensamblador el programador puede determinar el desplazamiento él mismo. Sin embargo si uno está interfazando C y ensamblador es muy importante que los dos códigos estén de acuerdo en los desplazamientos de los elementos de la estructura. Una complicación es que diferentes compiladores de C pueden tener desplazamientos diferentes de los elementos. Por ejemplo, como se ha visto, el compilador *gcc* crea una estructura `S` que se ve como la Figura 7.2; sin embargo, el compilador de Borland podría crear una estructura como la de la Figura 7.1. Los compiladores de C suministran maneras de especificar el alineamiento usado para los datos. Sin embargo la norma ANSI C no específica cómo se hace esto y así, cada compilador lo hará a su manera.

El compilador *gcc* tiene un método flexible y complicado de especificar el alineamiento. El compilador le permite a uno especificar el alineamiento de cualquier tipo usando una sintaxis especial. Por ejemplo, la siguiente línea:

```
typedef short int unaligned_int __attribute__((aligned(1)));
```

Define un nuevo tipo llamado `unaligned_int` que se alinea en los límites de los bytes (sí, todos los paréntesis después de `__attribute__` se necesitan). El 1 en el parámetro de `aligned` se puede reemplazar con otras potencias de dos para especificar otros alineamientos (2 para alineamiento de palabra, 4 para el alineamiento de palabras dobles, etc.). Si el elemento `y` de la estructura se cambia por un tipo `unaligned_int`, *gcc* podría colocar `y` a una distancia de 2. Sin embargo `z` podría continuar a una distancia de 8 ya que las palabras dobles tienen un alineamiento de palabra por omisión. La definición del tipo de `z` tendría que ser cambiada para colocarlos una distancia de 6.

El compilador *gcc* también permite *embalar* una estructura. Esto le dice al compilador que use el mínimo de espacio para la estructura. La Figura 7.3 muestra cómo se podría escribir `S` de esta manera. De esta forma `S` usaría el menor número de bytes posibles, 14.

Los compiladores de Borland y Microsoft, soportan el mismo método de especificar el alineamiento usando la directiva `#pragma`.

```
struct S {
  short int x;     /* entero de 2 bytes */
  int       y;     /* entero de 4 bytes */
  double    z;     /* float de 8 bytes  */
} __attribute__((packed));
```

Figura 7.3: Estructura embalada usando *gcc*

```
#pragma pack(push) /* guarda el estado de alineamiento */
#pragma pack(1)    /* establece el alieamiento de byte  */

struct S {
  short int x;     /* entero de 2 bytes */
  int       y;     /* entero de 4 bytes  */
  double    z;     /* float de 8 bytes   */
};

#pragma pack(pop)  /* restaura el alineamiento original */
```

Figura 7.4: Estructura embalada usando Microsoft o Borland

#pragma pack(1)

La directiva anterior le dice al compilador que embale los elementos de estructuras en los límites de los bytes (sin relleno extra). El uno se puede reemplazar con dos, cuatro, ocho o dieciséis para especificar el alineamiento en una palabra, palabra doble, palabra cuádruple y párrafo respectivamente. Esta directiva tiene efecto hasta que se sobreescriba por otra directiva. Esto puede causar problemas ya que estas directivas se usan a menudo en los archivos de encabezamiento. Si el archivo de encabezamiento se incluye antes que otros archivos de encabezamiento con estructuras, esas estructuras pueden estar diferente que las que tendrían por omisión. Esto puede conducir un error muy difícil de encontrar. Los módulos de un programa podrían desordenar los elementos de las estructuras de otros módulos.

Hay una manera de evitar este problema. Microsoft y Borland tienen una manera de guardar el alineamiento y restaurarlo luego. La Figura 7.4 muestra cómo se haría esto.

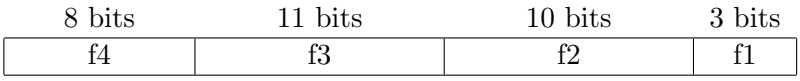
7.1.3. Campos de bits

Los campos de bit le permiten a uno especificar miembros de una estructura que solo usa un número especificado de bits. El tamaño de los bits

```
struct S {
  unsigned f1 : 3;    /* campo de 3 bits  */
  unsigned f2 : 10;   /* campo de 10 bits */
  unsigned f3 : 11;   /* campo de 11 bits */
  unsigned f4 : 8;    /* campo de 8  bits */
};
```

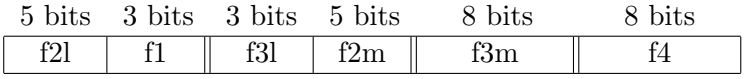
Figura 7.5: Ejemplo campo de bits

no tiene que ser múltiplo de 8. Un campo de bits miembro está definido como un **unsigned int** o un **int** con dos puntos y luego el tamaño en bits. La Figura 7.5 muestra un ejemplo. Esto define una variable de 32 bits que se descompone en las siguientes partes:



El primer campo está asignado a los bits menos significativos de esta palabra doble.[2]

Sin embargo, el formato no es tan simple, si uno mira cómo se almacenan los bits en memoria. La dificultad ocurre cuando los campos de bits se salen de los límites del byte. Ya que los bytes en un procesador little endian se invertirán en memoria. Por ejemplo, los campos de la estructura `S` se verán en memoria.



La etiqueta *f2l* se refiere a los últimos cinco bits (los 5 bits menos significativos) del campo *f2*. La etiqueta *f2m* se refiere a los 5 bits más significativos de *f2*. Las líneas verticales dobles muestran los límites de los bytes. Si uno invierte todos los bytes, las partes de los campos *f2* y *f3* se colocarán en el lugar correcto.

La forma de la memoria física no es importante a menos que los datos se transfieran hacia o fuera de nuestro programa (lo cual es muy poco común con campos de bits). Es común para dispositivos de hardware usar números impares de bits donde los campos de bits son útiles para representarlos.

Un ejemplo es SCSI[3]: Una orden de lectura directa para un dispositivo SCSI se especifica enviando un mensaje de 6 bits al dispositivo con el formato especificado en la Figura 7.6. La dificultad de representar esto usando

[2] Actualmente, la norma ANSI/ISO C le da al compilador alguna flexibilidad de como se ordenan los bits exactamente. Sin embargo los compiladores populares (*gcc*, *Microsoft* y *Borland*) usarán este orden en los campos.

[3] *Small Computer System Interface*, es una norma de la industria para discos duros etc.

Byte \ Bit	7	6	5	4	3	2	1	0
0	Codigo de operación (08h)							
1	Unidad Lógica #			msb de LBA				
2	mitad de la dirección del bloque lógico							
3	lsb de dirección del bloque lógico							
4	Longitud de transferencia							
5	Control							

Figura 7.6: Formato de la orden leer de SCSI

campos de bits es la *dirección del bloque lógico* (LBA) que abarca 3 bytes diferentes en la orden. De la Figura 7.6 uno ve que el dato se almacena en el formato big endian. La Figura 7.7 muestra una definición que intenta trabajar con todos los compiladores. Las dos primeras líneas definen un macro que retorna verdadero si el código se compila con los compiladores de Borland o Microsoft. La parte potencialmente confusa está en las líneas 11 a 14. Primero uno podría sorprenderse de ¿Por qué los bits lba_mid y lba_lsb están definidos separados y no en un solo campo de 16 bits? La razón es que el dato está en big endian. Un campo de 16 bits podría ser almacenado con el orden little endian por el compilador. Ahora, los campos lba_msb y logical_unit parecen invertidos; sin embargo esto no es el caso. Ellos tienen que colocarse en este orden. La Figura 7.8 muestra cómo se colocan los campos en una entidad de 48 bits (los límites del byte están una vez más marcados con líneas dobles). Cuando esto se almacena en memoria en el orden little endian, los bits se colocan en el orden deseado.(Figura 7.6)

Para complicar las cosas más, la definición para SCSI_read_cmd no trabaja correctamente en el compilador de Microsoft. Si se evalúa la expresión **sizeof**(SCSI_read_cmpl), Microsoft C retornará 8 no 6. Esto es porque el compilador de Microsoft usa el tipo de campo de datos para determinar cómo colocar los bits. Ya que los bits están definidos como de tipo **unsigned**, el compilador rellena dos bytes al final de la estructura para tener un múltiplo de palabras dobles. Esto se puede solucionar haciendo todos los campos **unsigned short**.

Ahora el compilador de Microsoft no necesita agregar ningún relleno ya que los 6 bytes son un múltiplo de una palabra[4] Los otros compiladores también trabajarán correctamente con este cambio. La Figura 7.9 muestra otra definición que trabaja con los tres compiladores. Se evitan todos los problemas usando campos de tipo **unsigned char**.

El lector no debería entristecerse si encuentra la división anterior confusa. ¡Es confusa! El autor a menudo encuentra menos confuso evitar los

[4]Mezclar diferentes tipos de campos de bits produce un comportamiento muy confuso. Se invita al lector a experimentar al respecto

```
#define MS_OR_BORLAND (defined(__BORLANDC__) \
                       || defined(_MSC_VER))

#if MS_OR_BORLAND
# pragma pack(push)
# pragma pack(1)
#endif

struct SCSI_read_cmd {
  unsigned opcode : 8;
  unsigned lba_msb : 5;
  unsigned logical_unit : 3;
  unsigned lba_mid : 8;     /* bits de la mitad */
  unsigned lba_lsb : 8;
  unsigned transfer_length : 8;
  unsigned control : 8;
}
#if defined(__GNUC__)
   __attribute__((packed))
#endif
;

#if MS_OR_BORLAND
# pragma pack(pop)
#endif
```

Figura 7.7: Estructura del formato de la orden leer de SCSI

campos de bits y en su lugar usar las operaciones entre bits para examinar y modificar los bits manualmente.

7.1.4. Usando estructuras en ensamblador

Como se discutió antes, acceder a una estructura en ensamblador es muy parecido a acceder a un arreglo. Por ejemplo considere cómo uno podría escribir una rutina en ensamblador que le asignará cero al elemento `y` de una estructura `S`. Asumiendo que el prototipo de la rutina sería:

```
void zero_y( S * s_p );
```

8 bits	8 bits	8 bits	8 bits	3 bits	5 bits	8 bits
control	transfer_length	lba_lsb	lba_mid	logical_unit	lba_msb	opcode

Figura 7.8: Ordenamiento de los campos de `SCSI_read_cmd`

```
struct SCSI_read_cmd {
  unsigned char opcode;
  unsigned char lba_msb : 5;
  unsigned char  logical_unit  : 3;
  unsigned char lba_mid;     /* bits de la mitad*/
  unsigned char lba_lsb;
  unsigned char  transfer_length ;
  unsigned char control;
}
#if defined(__GNUC__)
    __attribute__((packed))
#endif
;
```

Figura 7.9: Estructura alterna del formato de la orden leer de SCSI

La rutina en ensamblador sería:

```
1  %define      y_offset  4
2  _zero_y:
3        enter  0,0
4        mov    eax, [ebp + 8]      ; toma  s_p (apuntador a la estructura)
5                                   ; de la pila
6        mov    dword [eax + y_offset], 0
7        leave
8        ret
```

C le permite a uno pasar estructuras por valor a una función; sin embargo esto la mayoría de las veces es mala idea. Cuando se pasa por valor, todos los datos de la estructura se deben copiar en la pila y luego traídos por la rutina, En lugar de esto es mucho más eficiente pasar un apuntador a una estructura.

C También permite que un tipo de estructura se use como un valor de retorno. Obviamente la estructura no se puede retornar en el registro `EAX`. Cada compilador maneja la situación diferente. Una solución común es que el compilador usa esto para internamente reescribir la función para que tome una apuntador a una estructura como parámetro. El apuntador se usa para

```
#include <stdio.h>

void f( int x )
{
   printf (" %d\n", x);
}

void f( double x )
{
   printf (" %g\n", x);
}
```

Figura 7.10: Dos funciones `f()`

colocar el valor de retorno en una estructura definida fuera de la rutina llamada.

La mayoría de los ensambladores (incluido NASM) tiene un soporte empotrado para definir estructuras en su código de ensamblador. Consulte su documentación para los detalles.

7.2. Ensamblador y C++

El lenguaje de programación C++ es una extensión del lenguaje C. Muchas de las reglas básicas para interfazar C y ensamblador también son válidas para C++. Sin embargo algunas reglas necesitan ser modificadas. También algunas de las extensiones de C++ son fáciles de entender con conocimiento del lenguaje ensamblador. Esta sección asume un conocimiento básico de C++.

7.2.1. Manipulación de la sobrecarga de nombres

C++ permite que se definan diferentes funciones (y funciones miembro de clases) con el mismo nombre. Cuando más de una función comparte el mismo nombre, se dice que la función está *sobrecargada*. Si se definen dos funciones con el mismo nombre en C, el encadenador producirá un error porque él encontrará dos definiciones para el mismo símbolo en los archivos objeto que está encadenando. Por ejemplo considere el código de la Figura 7.10. El código en ensamblador equivalente definiría 2 etiquetas `_f` que obviamente será un error.

C++ usa el mismo proceso de encadenamiento que C, pero evita este error haciendo una *manipulación de nombres* o modificando el símbolo usado para etiquetar la función. De alguna manera C usa la manipulación de

nombres también. Añade un guión bajo a la etiqueta de la función de C cuando se crea la etiqueta para la función. Sin embargo, C manipulará el nombre de ambas funciones en la Figura 7.10 de la misma manera y produce un error. C++ usa un proceso de manipulación más sofisticado que produce dos etiquetas diferentes para las funciones. Por ejemplo, a la primera función de la Figura 7.10 le podría asignar DJGPP la etiqueta `_f__Fi` y a la segunda función, `_f__Fd`. Esto evita errores de encadenamiento.

Desafortunadamente no está normalizado cómo se manejan los nombres en C++ y cada compilador trata los nombres a su manera. Por ejemplo Borland C++ podría usar las etiquetas `@f$qi` y `@f$qd` para las dos funciones de la Figura 7.10. Sin embargo, las reglas no son completamente arbitrarias. El nombre manipulado codifica la *firma* de la función. La firma de una función está definida por el orden y el tipo de sus parámetros. Observe que la función que toma un solo parámetro `int` tiene una *i* al final de la etiqueta manipulada (para DJGPP y Borland) y la que toma un argumento `double` tiene una *d* al final de la etiqueta manipulada. Si tuviera una función llamada `f` con el prototipo.

```
void f( int x, int y, double z);
```

DJGPP podría manipular el nombre para ser `_f__Fiid` y Borland a `@f$qiid`.

El tipo de retorno de una función *no* hace parte de la firma de la función y no se codifica en el nombre manipulado. Este hecho explica una regla de sobre carga en C++. Sólo funciones cuya firma es única se pueden sobrecargar. Como uno puede ver, si dos funciones, con el mismo nombre y firma, están definidas en C++ ellas producirán el mismo nombre manipulado y crearán un error de encadenamiento. Por omisión C++ manipula los nombres de las funciones así no estén sobrecargadas. Cuando se compila un archivo el compilador no tiene manera de saber si una función en particular está sobrecargada o no, así que manipula todos los nombres. De hecho también manipula los nombres de las variables globales codificando el tipo de variable de manera similar a la firma de las funciones. Así uno puede definir una variable global en un archivo como de cierto tipo y entonces intentar usarla en otro archivo como de un tipo erróneo, se producirá un error del encadenador. Esta característica de C++ es conocida como *encadenamiento seguro de tipos*(*typesafe linking*. También se expone a otro tipo de error, prototipos inconsistentes. Esto ocurre cuando la definición de una función en un módulo no concuerda con el prototipo usado por otro módulo. En C esto puede ser un problema muy difícil de depurar. C no atrapa este error. El programa compilará y encadenará, pero tendrá un comportamiento indefinido cuando se invoca el código colocando diferentes tipos de datos en la pila de los que la función espera. En C++, esto producirá un error de encadenado.

Cuando el compilador de C++ está examinando la sintaxis de un llamado

a función, él busca una función donde emparejen los tipos de los argumentos pasados a la función.[5] Si se empareja entonces crea un `CALL` a la función correcta usando las reglas de manipulación del compilador.

Ya que cada compilador usa sus reglas de manipulación de nombres, los archivos objeto de C++ generados por diferentes compiladores no se pueden unir, el código de C++ compilado con compiladores diferentes no es posible unirlo. Este hecho es importante cuando se considera usar una biblioteca de C++ precompilada. Si uno desea escribir una función en ensamblador que será usada con código C++, debe conocer las reglas de manipulación de nombres del compilador (o usar la técnica explicada abajo).

El estudiante astuto puede preguntar si el código de la Figura 7.10 trabajará como se espera. Ya que C++ manipula los nombres de todas las funciones, entonces la función `printf` será manipulada y el compilador no producirá un `CALL` a la etiqueta `_printf`. Esto es un válido. Si el prototipo para `printf` fue colocado simplemente al principio del archivo, esto podría pasar. El prototipo es:

```
int printf( const char *, ...);
```

DJGPP podría manipular esto como `_printf__FPCce` (la `F` es por *función*, `P` por *apuntador*, `C` por *constante*, `c` por *char* y `e` por *elipsis*). Esto podría no llamar la función `printf` de la biblioteca de C. Claro está, debe haber alguna forma para el código de C++ llamar código de C. Esto es muy importante ya que hay una gran cantidad de código de C útil. Además permitirle a uno llamar código de C heredado, C++ también le permite a uno llamar código de ensamblador usando las convenciones normales de llamado.

C++ extiende la palabra clave `extern` para permitir especificar que la función o variable global que modifica usa las convenciones de llamado de C. En la terminología de C++, las variables y funciones globales usan el *encadenamiento de C*. Por ejemplo para declarar que `printf` tenga el encadenamiento de C, use el prototipo:

```
extern "C" int printf( const char *, ... );
```

Esto le dice al compilador que no use las reglas de manipulación de nombres en esta función, y en su lugar usar las reglas de C. Sin embargo, haciendo esto, la función `printf` no se puede sobrecargar. Esto suministra una manera fácil de interfazar C++ y ensamblador, define la función para que use el encadenamiento de C y la convención de llamado de C.

Por conveniencia C++ también permite el encadenamiento de un bloque de funciones y variables globales a ser definidas. El bloque se enmarca por las típicas llaves:

[5] El emparejamiento no tiene que ser exacto, el compilador considerará las conversiones de tipos de los argumentos. Las reglas para este proceso están más allá del alcance de este libro. Consulte un libro de C++ para los detalles.

```
void f( int & x )      // el & denota una referencia
{ x++; }

int main()
{
  int y = 5;
  f(y);                // se pasa la referencia a y, obeserve que
                       // no hay & acá
   printf (" %d\n", y); // imprime 6!
  return 0;
}
```

Figura 7.11: Ejemplo de referencia

```
extern "C" {
  /* encadenamiento tipo C a variables globales y prototipos de funciones */
}
```

Si uno examina los archivos de encabezado de ANSI C que vienen con los compiladores de C/C++ de hoy día, en ellos se encontrá al principio de cada archivo de cabecera:

```
#ifdef __cplusplus
extern "C" {
#endif
```

Y una construcción parecida cerca del final conteniendo el fin de la llave. Los compiladores de C++ definen el macro `__cplusplus` (con *dos* guiones bajos adelante). La porción de código anterior encierra todo el archivo de cabecera en un bloque `extern C` si el archivo de cabecera es compilado como C++, pero no hace nada si es compilado como C (ya que el compilador de C daría un error de sintaxis para `extern C`). Esta misma técnica puede ser usada por cualquier programador para crear un archivo de cabecera para las rutinas de ensamblador que pueden ser usadas con C o C++.

7.2.2. Referencias

Las *referencias* son otra característica nueva C++. Ellas le permiten a uno pasar parámetros a funciones sin usar apuntadores explícitamente. Por ejemplo, considere el código de la Figura 7.11. Los parámetros por referencia son muy simples, ellos realmente son tan solo apuntadores. El compilador solo le oculta esto al programador (tal como los compiladores de Pascal implementan los parámetros `var` como apuntadores). Cuando el compilador genera el ensamblador para el llamado a la función en la línea 7, pasa la

dirección de `y`. Si uno escribió la función `f` en ensamblador, ella se debería comportar como si el prototipo fuera[6]:

```
void f( int * xp);
```

Las referencias son sólo una conveniencia que es especialmente útil para la sobrecarga de operadores. Esta es otra característica de C++ que le permite a uno definir el significado de los operadores comunes en estructuras o tipos de clases. Por ejemplo, un uso común es definir el operador más (`+`) para unir objetos tipo cadena. Así, si `a` y `b` fueran cadenas, `a + b` debería retornar la unión de las cadenas `a` y `b`. C++ llama una función para hacer esto (de hecho, esta expresión podría ser reescrita en la notación de función como `operator +(a,b)`). Por eficiencia uno podría querer pasar la dirección de los objetos cadena en lugar de pasarlos por valor. Sin referencias esto se podría hacer como `operator +(&a,&b)`, pero esto requeriría escribir en la sintaxis del operador como `&a + &b`. Esto sería muy complicado y confuso. Sin embargo, usando referencias, uno puede escribir `a + b` que se ve muy natural.

7.2.3. Funciones inline

Las *funciones inline* son otra característica de C++[7]. Las funciones inline son hechas para reemplazar, los macros basados en el procesador que toman parámetros que son muy propensos a tener errores. Recuerde de C, que escribir un macro que eleva al cuadrado un número podría verse como:

```
#define SQR(x) ((x)*(x))
```

Debido a que el preprocesador no entiende C y hace simples sustituciones, se requieren los paréntesis para calcular la respuesta correcta en la mayoría de los casos. Sin embargo, aún esta versión no dará la respuesta correcta para `SQR(X++)`.

Los macros se usan porque ellos eliminan el trabajo extra de hacer un llamad a función para una función elemental. Como se demostró en el capítulo de subprogramas realizar un llamado a función involucra varios pasos. Por cada función elemental, el tiempo que se toma en llamar la función puede ser mayor que el tiempo necesario en realizar las operaciones en la función. La función inline es una manera mucho más amigable de escribir código que se vea como una función normal, pero que *no* hace el bloque de código para `CALL`. En lugar de ello los llamados a funciones inline son reemplazados por el código que realiza la función. C++ le permite a una función ser inline

[6]Claro está ella podría esperar declarar la función con encadenamiento C para evitar la manipulación de nombres discutida en la sección 7.2.1

[7]Los compiladores de C a menudo soportan esta característica como una extensión de ANSI C

```
inline int inline_f ( int x )
{ return x*x; }

int f( int x )
{ return x*x; }

int main()
{
  int y, x = 5;
  y = f(x);
  y = inline_f (x);
  return 0;
}
```

Figura 7.12: ejemplo inline

colocando la palabra `inline` clave al frente de la definición de la función. Por ejemplo considere las funciones declaradas en la Figura 7.12. El llamado a la función `f` en la línea 10 hace un llamado normal a una función (en ensamblador, asumiendo que `x` está en la dirección `ebp-8` e `y` está en `ebp-4`:

```
1   push   dword [ebp-8]
2   call   _f
3   pop    ecx
4   mov    [ebp-4], eax
```

Sin embargo, el llamado a la función `inline_f` de la línea 11 podría verse como:

```
1   mov    eax, [ebp-8]
2   imul   eax, eax
3   mov    [ebp-4], eax
```

En este caso hay dos ventajas de las funciones inline. Primero la función en línea es rápida. No se colocan los parámetros en la pila, no se crea el marco de la pila ni se destruye, no se hace el salto. Segundo el llamado de las funciones en línea usa menos código. Este último punto es verdad para este ejemplo pero no siempre es cierto.

La principal desventaja de las funciones inline es que el código en línea no se encadena y así el código de una función en línea debe estar disponible

```
class Simple {
public:
  Simple();                  // Construcctor por omisión
  ~Simple();                 // destructor
  int get_data() const;      // funciones miembro
  void set_data( int );
private:
  int data;                  // datos miembro
};

Simple::Simple()
{ data = 0; }

Simple::~Simple()
{ /* cuerpo vacío */ }

int Simple::get_data() const
{ return data; }

void Simple::set_data( int x )
{ data = x; }
```

Figura 7.13: Una clase simple de C++

en *todos* los archivos que la usen. El ejemplo anterior prueba esto. El llamado de una función normal solo requiere el conocimiento de los parámetros, el tipo de valor de retorno, la convención de llamado y el nombre de la etiqueta para la función. Toda esta información está disponible en el prototipo de la función. Sin embargo al usar funciones inline se requiere conocer todo el código de la función. Esto significa que si *cualquier* parte de una función en línea se cambia, *todos* los archivos fuentes que usen la función deben ser recompilados. Recuerde que para las funciones no inline, si el prototipo cambia, a menudo los archivos que usan la función no necesitan ser recompilados. Por todas estas razones, el código de las funciones en línea se colocan normalmente en los archivos de cabecera. Esta práctica es contraria a la regla estricta de C las instrucciones de código ejecutable *nunca* se colocan en los archivos de cabecera.

```
void set_data( Simple * object, int x )
{
  object->data = x;
}
```

Figura 7.14: Versión de C de Simple::set_data()

```
1  _set_data__6Simplei:           ; nombre manipulador
2        push   ebp
3        mov    ebp, esp
4
5        mov    eax, [ebp + 8]   ; eax = apuntador al objeto (this)
6        mov    edx, [ebp + 12]  ; edx = parámetro entero
7        mov    [eax], edx       ; data tiene un desplazamiento de 0
8
9        leave
10       ret
```

Figura 7.15: Salida del compilador para Simple::set_data(int)

7.2.4. Clases

En efecto, C++ usa la palabra clave `this` *para acceder al apuntador de un objeto que está dentro de la función miembro.*

Una clase en C++ describe un tipo de objeto. Un objeto tiene como miembros a datos y funciones[8]. En otras palabras, es una `estructura` con datos y funciones asociadas a ella. Considere la clase definida en la Figura 7.13. Una variable de tipo `Simple` se podría ver tal como una `estructura` normal en C con un solo `int` como miembro. Las funciones *no* son almacenadas en la memoria asignada a la estructura. Sin embargo las funciones miembro son diferentes de las otras funciones . Ellas son pasadas como un parámetro *oculto*. Este parámetro es un apuntador al objeto al cual la función pertenece.

Por ejemplo considere el método `set_data` de la clase `Simple` de la Figura 7.13. Si se escribió en C se vería como una función que fue pasada explícitamente a un apuntador al objeto como muestra la Figura 7.14. La opción `-S` en el compilador *DJGPP* (y en los compiladores *gcc* y Borland también) le dice al compilador que genere un archivo que contenga el lenguaje ensamblador equivalente del código producido. Para *DJGPP* y *gcc* el archivo ensamblador finaliza con una extensión `.s` y desafortunadamente usa la sintaxis del lenguaje ensamblador AT&T que es diferente de la

[8] a menudo llamados en C++ *funciones miembro* o más generalmente *métodos*.

sintaxis de NASM y MASM.[9] (Los compiladores de Borland y Microsoft generan un archivo con extensión `.asm` usando la sintaxis MASM). La Figura 7.15 muestra la salida de *DJGPP* convertida a la sintaxis de MASM y con comentarios añadidos para clarificar el propósito de las instrucciones. En la primera línea, observe que al método `set_data` se le asigna la etiqueta manipulada que codifica el nombre del método, el nombre de la clase y los parámetros. El nombre de la clase se codifica porque otras clases podrían tener un método llamado `set_data` y los dos métodos *deben* tener etiqueta diferente. Los parámetros se codifican tal que la clase pueda sobrecargar el método `set_data` para tomar otros parámetros como es normal en las funciones C++. Sin embargo, tal como antes, cada compilador codificará esta información diferente en la etiqueta manipulada.

Luego en las líneas 2 y 3, aparece el prólogo típico de la función. En la línea 5 se almacena el primer parámetro de la pila en `EAX`. Este *no* es el parámetro x, es el parámetro oculto.[10] que apunta al objeto sobre el cual actúa. La línea 6 almacena el parámetro `x` en `EDX` y la línea 7 almacena `EDX` en la palabra doble a la que apunta `EAX`. Este es el miembro `data` del objeto `Simple` sobre el cual actúa, que es el único dato en la clase, se almacena con un desplazamiento cero en la estructura `Simple`.

Ejemplo

Esta sección usa las ideas del capítulo para crear una clase de C++ que representa un entero sin signo de tamaño arbitrario. Ya que el entero puede ser de cualquier tamaño, será almacenado en un arreglo de enteros sin signo (palabras dobles). Se puede hacer de cualquier tamaño usando memoria dinámica. Las palabras dobles se almacenan en orden inverso.[11] (la palabra doble menos significativa es la palabra doble que está en la posición 0). La Figura 7.16 muestra la definición de la clase `Big_int`.[12]. El tamaño de un `Big_int` se mide por el tamaño del arreglo `sin signo` que es usado para almacenar este dato. El dato miembro `size_` de la clase se le asigna un desplazamiento de cero y el miembro `number_` se le asigna un desplazamiento de 4.

[9] El sistema de compilador *gcc* incluye su propio ensamblador llamado *gas*. El ensamblador gas usa la sintaxis AT&T y así el compilador produce el código en el formato de gas. Hay varias páginas web que discuten las diferencias entre los formatos INTEL y AT&T. Hay también un programa mínimo llamado *a2i* (`http://www.multimania.com/placr/a2i.html`), que convierte del formato AT&T al formato NASM

[10] Como es normal *nada* está oculto en el código ensamblador

[11] ¿Por qué? Porque las operaciones de adición siempre se comenzarán a procesar al inicio del arreglo y se moverá hacia adelante.

[12] Vea los archivos example para el código completo de este ejemplo. El texto solo se referirá a alguna parte del código.

```
class Big_int {
public:
   /*
   * Parámetros:
   *    size            - tamaño del entero expresado como el número de
   *                      unsigned int normales
   *    initial_value   - valor inicial de Big_int como un
   *                      unsigned int normal
   */
  explicit Big_int( size_t     size,
                    unsigned initial_value = 0);
  /*
   * Parámetros
   *    size            - tamaño del entero expresado como el número de
   *                      unsigned int normales
   *    initial_value   - valor inicial de Big_int como la cadena que
   *                      almacena la representación hexadecimal del valor.
   */
  Big_int( size_t          size,
           const char *  initial_value );

  Big_int( const Big_int & big_int_to_copy );
  ~Big_int();

  // retorna el tamaño de Big_int (en términos de unsigned int)
  size_t size() const;

  const Big_int & operator = ( const Big_int & big_int_to_copy );
  friend Big_int operator + ( const Big_int & op1,
                              const Big_int & op2 );
  friend Big_int operator - ( const Big_int & op1,
                              const Big_int & op2);
  friend bool operator == ( const Big_int & op1,
                            const Big_int & op2 );
  friend bool operator < ( const Big_int & op1,
                           const Big_int & op2);
  friend ostream & operator << ( ostream &      os,
                                 const Big_int & op );
private:
  size_t        size_ ;     // tamaño del arreglo sin signo
  unsigned * number_; // apuntador al arreglo sin singno que
                      // almacena el valor
};
```

Figura 7.16: Definición de la clase Big_int

```
// Prototipos para las rutinas en ensamblador
extern "C" {
  int add_big_ints ( Big_int &       res,
                     const Big_int & op1,
                     const Big_int & op2);
  int sub_big_ints ( Big_int &       res,
                     const Big_int & op1,
                     const Big_int & op2);
}

inline Big_int operator + ( const Big_int & op1, const Big_int & op2)
{
  Big_int result (op1.size ());
  int res = add_big_ints( result , op1, op2);
  if (res == 1)
    throw Big_int::Overflow();
  if (res == 2)
    throw Big_int::Size_mismatch();
  return result ;
}

inline Big_int operator - ( const Big_int & op1, const Big_int & op2)
{
  Big_int result (op1.size ());
  int res = sub_big_ints( result , op1, op2);
  if (res == 1)
    throw Big_int::Overflow();
  if (res == 2)
    throw Big_int::Size_mismatch();
  return result ;
}
```

Figura 7.17: Código de la aritmética de la clase Big_int

Para simplificar este ejemplo, sólo se pueden sumar o restar instancias objeto con el mismo tamaño

La clase tiene tres constructores: El primero (línea 9) inicia la instancia de la clase usando un entero sin signo normal; el segundo (línea 18) inicia la instancia usando una cadena que contiene un valor en hexadecimal. El tercer constructor (línea 21) es el *constructor copia*.

Esta discusión se enfoca en cómo trabajan los operadores de la suma y de la resta ya que es allí donde se usa el lenguaje ensamblador. La Figura 7.17 muestra las partes relevantes del archivo de encabezado para estos operadores. Ellas muestran cómo se usan los operadores para llamar las rutinas en ensamblador. Ya que cada compilador usa sus reglas de manipulación de nombres radicalmente diferentes para las funciones de los operadores, las funciones en línea de los operadores se usan para establecer el encadenamiento de rutinas en ensamblador. Esto hace relativamente fácil portar a diferentes compiladores y es tan rápido como los llamados directos. Esta técnica también elimina la necesidad de lanzar una excepción desde ensamblador.

¿Por qué, a pesar de todo, se usa el ensamblador acá? Recuerde que para realizar aritmética de precisión múltiple, el carry se debe mover de una palabra doble a la siguiente para sumar. C++ (y C) no le permiten al programador acceder a la bandera de carry de la CPU. Para realizar la suma con C++ se podría hacerlo recalculando la bandera de carry y condicionalmente sumar esto a la siguiente palabra doble. Es mucho más eficiente escribir el código en ensamblador donde se puede acceder a la bandera de carry y usando la instrucción `ADC` que automáticamente suma la bandera de carry.

Por brevedad, sólo se discutirá acá la rutina de ensamblador `add_big_ints`. A continuación está el código de esta rutina (de `big_math.asm`):

big_math.asm

```
1  segment .text
2          global  add_big_ints, sub_big_ints
3  %define size_offset 0
4  %define number_offset 4
5
6  %define EXIT_OK 0
7  %define EXIT_OVERFLOW 1
8  %define EXIT_SIZE_MISMATCH 2
9
10 ; Parámetros para las rutinas add y sub
11 %define res ebp+8
12 %define op1 ebp+12
13 %define op2 ebp+16
```

```
14
15  add_big_ints:
16          push    ebp
17          mov     ebp, esp
18          push    ebx
19          push    esi
20          push    edi
21          ;
22          ; primero      esi señala a op1
23          ;              edi señala a op2
24          ;              ebx señala a res
25          mov     esi, [op1]
26          mov     edi, [op2]
27          mov     ebx, [res]
28          ;
29          ; Asegurarse que todos los 3 Big_Int tienen el mismo tamaño
30          ;
31          mov     eax, [esi + size_offset]
32          cmp     eax, [edi + size_offset]
33          jne     sizes_not_equal                 ; op1.size_ != op2.size_
34          cmp     eax, [ebx + size_offset]
35          jne     sizes_not_equal                 ; op1.size_ != res.size_
36
37          mov     ecx, eax                        ; ecx = size of Big_int's
38          ;
39          ; ahora, los registros señalan a sus respectivos arreglos
40          ;      esi = op1.number_
41          ;      edi = op2.number_
42          ;      ebx = res.number_
43          ;
44          mov     ebx, [ebx + number_offset]
45          mov     esi, [esi + number_offset]
46          mov     edi, [edi + number_offset]
47
48          clc                                     ; borra bandera de carry
49          xor     edx, edx                        ; edx = 0
50          ;
51          ; bucle de suma
52  add_loop:
53          mov     eax, [edi+4*edx]
54          adc     eax, [esi+4*edx]
55          mov     [ebx + 4*edx], eax
```

```
56          inc     edx                             ; no altera la bandera de carry
57          loop    add_loop
58
59          jc      overflow
60  ok_done:
61          xor     eax, eax                        ; valor de retorno = EXIT_OK
62          jmp     done
63  overflow:
64          mov     eax, EXIT_OVERFLOW
65          jmp     done
66  sizes_not_equal:
67          mov     eax, EXIT_SIZE_MISMATCH
68  done:
69          pop     edi
70          pop     esi
71          pop     ebx
72          leave
73          ret
```

big_math.asm

Las líneas 25 a 27 almacenan los apuntadores a los objetos `Big_int` pasados a la función en registro. Recuerde que las referencias realmente son solo apuntadores. Las líneas 31 a 35 aseguran que los tamaños de los 3 objetos de los arreglos sean iguales (Observe que el desplazamiento de `size_` se añade al apuntador para acceder al dato miembro). Las líneas 44 a 46 ajustan los registros para apuntar al arreglo usado por los objetos respectivos en lugar de los objetos en sí mismos. (Una vez más, el desplazamiento de `number_` se añade al apuntador del objeto).

El bucle en las líneas 52 a 57 suma los enteros almacenados en los arreglos sumando primero la palabra doble menos significativa, luego la siguiente palabra doble y así sucesivamente. La suma debe realizarse en esta secuencia para la aritmética de precisión extendida (vea la Sección 2.1.5). La línea 59 examina el desborde, si hay desborde la bandera de carry se debería fijar con la última suma de la palabra doble más significativa. Ya que las palabras dobles en el arreglo se almacenan en el orden de little endian, el bucle comienza en el inicio del arreglo y se mueve adelante hasta el final.

La Figura 7.18 muestra un corto ejemplo del uso de la clase `Big_int`. Observe que las constantes `Big_int` se deben declarar explícitamente como en la línea 16. Esto es necesario por dos razones. Primero, no hay un constructor de conversión que convierta un unsigned int en un `Big_int`. Segundo, solo se pueden sumar `Big_Int` del mismo tamaño, esto hace la conversión problemática ya que podría ser difícil conocer a qué tamaño convertir. Una implementación más sofisticada de la clase debería permitir sumar objetos de cualquier tamaño. El autor no desea complicar este ejemplo implemen-

```
#include "big_int.hpp"
#include <iostream>
using namespace std;

int main()
{
  try {
    Big_int b(5,"8000000000000a00b");
    Big_int a(5,"80000000000010230");
    Big_int c = a + b;
    cout << a << " + " << b << " = " << c << endl;
    for( int i=0; i < 2; i++ ) {
      c = c + a;
      cout << "c = " << c << endl;
    }
    cout << "c-1 = " << c - Big_int(5,1) << endl;
    Big_int d(5, "12345678");
    cout << "d = " << d << endl;
    cout << "c == d " << (c == d) << endl;
    cout << "c > d " << (c > d) << endl;
  }
  catch( const char * str ) {
    cerr << "Caught: " << str << endl;
  }
  catch( Big_int :: Overflow ) {
    cerr << "Overflow" << endl;
  }
  catch( Big_int :: Size_mismatch ) {
    cerr << "Size mismatch" << endl;
  }
  return 0;
}
```

Figura 7.18: Uso simple de `Big_int`

tandolo acá. (Sin embargo exhorta al lector a hacer esto).

7.2.5. Herencia y polimorfismo

La *herencia* le permite a una clase heredar los datos y métodos de otra. Por ejemplo consider el código de la Figura 7.19. Ella muestra dos clases `A` y `B`, donde la clase `B` hereda de `A`. La salida del programa es:

```
Tamaño de a: 4 Desplazamiento de ad: 0
Tamaño de b: 8 Desplazamiento de ad: 0 Desplazamiento de bd: 4
A::m()
A::m()
```

Observe que los miembros `ad` de ambas clases (`B` hereda de `A`) están a la misma distancia. Esto es importante ya que la función `f` puede ser pasada como un apuntador, a un objeto `A` a cualquier tipo derivado de A. La Figura 7.20 muestra el código asm (editado) para la función (generada por *gcc*).

Observe que en la salida del método `m` de `A` fue llamado por los objetos `a` y `b`. Desde ensamblador, uno puede ver que el llamado a `A::m()` está empotrado en la función. Para la programación realmente orientada a objetos, el método llamado dependería de qué tipo de objeto se pasa a la función. Esto es conocido como `polimorfismo`. C++ tiene esta característica apagada por omisión. Uno usa la palabra clave *virtual* para habilitarla. La Figura 7.21 muestra cómo las dos clases serían cambiadas. Nada del otro código necesita ser cambiado. El polimorfismo puede implementarse de muchas maneras. Desafortunadamente la implementación de `gcc` está en transición al momento de escribir esto y es significativamente más complicada que la implementación inicial. Interesado en simplificar esta discusión, el autor solo cubrirá la implementación del polimorfismo que usan los compiladores de Windows, Microsoft y Borland. Esta implementación no ha cambiado en muchos años y probablemente no cambie en el futuro cercano.

Con estos cambios, la salida del programa cambia:

```
Tamaño de a: 8 Desplazamiento de ad: 4
Tamaño de b: 12 Desplazamiento de ad: 4 Desplazamiento de bd: 8
A::m()
B::m()
```

Ahora el segundo llamado a `f` invoca el método `B::m()` porque es pasado a un objeto `B`. Esto no es lo único que cambia. El tamaño de `A` es ahora 8 (y `B` es 12). También el desplazamiento de `ad` es 4 no es 0. ¿Qué es un desplazamiento de cero? La respuesta a esta pregunta está relacionada en cómo es la implementación del polimorfismo.

```
#include <cstddef>
#include <iostream>
using namespace std;

class A {
public:
  void __cdecl m() { cout << "A::m()" << endl; }
  int ad;
};

class B : public A {
public:
  void __cdecl m() { cout << "B::m()" << endl; }
  int bd;
};

void f( A * p )
{
  p->ad = 5;
  p->m();
}

int main()
{
  A a;
  B b;
  cout << "Tamaño de a: " << sizeof(a)
       << " Desplazamiento de ad: " << offsetof(A,ad) << endl;
  cout << "Tamaño de b: " << sizeof(b)
       << " Desplazamiento de ad: " << offsetof(B,ad)
       << " Desplazamiento de bd: " << offsetof(B,bd) << endl;
  f(&a);
  f(&b);
  return 0;
}
```

Figura 7.19: Herencia simple

```
1  _f__FP1A:                        ; nombre de la función manipulada
2        push   ebp
3        mov    ebp, esp
4        mov    eax, [ebp+8]        ; eax apunta al objeto
5        mov    dword [eax], 5      ; usar desplazamiento 0 para ad
6        mov    eax, [ebp+8]        ; pasar la dirección del objeto a A::m()
7        push   eax
8        call   _m__1A              ; nombre del método manipulado para  A::m()
9        add    esp, 4
10       leave
11       ret
```

Figura 7.20: Código en ensamblador para la herencia simple

```
class A {
public:
  virtual void __cdecl m() { cout << "A::m()" << endl; }
  int ad;
};

class B : public A {
public:
  virtual void __cdecl m() { cout << "B::m()" << endl; }
  int bd;
};
```

Figura 7.21: Herencia polimórfica

```
1  ?f@@YAXPAVA@@@Z:
2        push   ebp
3        mov    ebp, esp
4
5        mov    eax, [ebp+8]
6        mov    dword [eax+4], 5  ; p->ad = 5;
7
8        mov    ecx, [ebp + 8]    ; ecx = p
9        mov    edx, [ecx]        ; edx = apuntador a vtable
10       mov    eax, [ebp + 8]    ; eax = p
11       push   eax               ; empuja apuntador "this"
12       call   dword [edx]       ; llama primera función en vtable
13       add    esp, 4            ; limpia la pila
14
15       pop    ebp
16       ret
```

Figura 7.22: Código en ensamblador para la función `f()`

Una clase C++ que tiene cualquier método virtual se le da un campo oculto extra que es un apuntador a un arreglo de apuntadores a métodos.[13]. Esta tabla se llama a menudo *vtable*. Para las clases `A` y `B` este apuntador se almacena con un desplazamiento 0. Los compiladores de Windows siempre colocan este apuntador al inicio de la clase, en el tope del árbol de herencia. Mirando en el código ensamblador (Figura 7.22) generado para la función `f` (de la Figura 7.19) para el método virtual del programa, uno puede ver que la llamada al método `m` no es a una etiqueta. La línea 9 encuentra la dirección de la vtable de objeto. La dirección del objeto se empuja en la pila en la línea 11. La línea 12 llama al método virtual ramificándose en la primera dirección en la vtable.[14]. Esta llamada no usa una etiqueta, salta a la dirección de código apuntada por `EDX`. Este tipo de llamadas es un ejemplo de *vínculo tardío* . El vínculo tardío retraza la decisión de qué método llamar hasta que el código se esté ejecutando. Esto le permite al código llamar el método apropiado para el objeto. El caso normal (Figura 7.20) fija una llamada a cierto método es llamado *primer vínculo* ya que aquí el método está atado en el tiempo de compilación.

[13]Para las clases sin métodos virtuales los compiladores de C++ siempre hacen la clase compatible con una estructura normal de C con los mismos miembros.

[14]Claro está, este valor ya está en el registro `ECX` él fue colocado en la línea 8 y la línea 10 podía ser quitado y la próxima línea cambiada para empujar `ECX`. El código no es muy eficiente porque se genera sin tener activadas las optimizaciones del compilador.

```
class A {
public:
  virtual void __cdecl m1() { cout << "A::m1()" << endl; }
  virtual void __cdecl m2() { cout << "A::m2()" << endl; }
  int ad;
};

class B : public A {     // B hereda de A  m2()
public:
  virtual void __cdecl m1() { cout << "B::m1()" << endl; }
  int bd;
};
/* imprime la vtable de un objeto dado */
void print_vtable ( A * pa )
{
  // p ve a pa como un arreglo de palabras dobles
  unsigned * p = reinterpret_cast<unsigned *>(pa);
  // vt ve la vtable como un arreglo de apuntadores
  void ** vt = reinterpret_cast<void **>(p[0]);
  cout << hex << "Dirección de la vtable = " << vt << endl;
  for( int i=0; i < 2; i++ )
    cout << "dword " << i << ": " << vt[i] << endl;

  // llamado a funciones virtuales de una manera
  // EXTREMADAMENTE no portable
  void (*m1func_pointer)(A *);    // function pointer variable
  m1func_pointer = reinterpret_cast<void (*)(A*)>(vt[0]);
  m1func_pointer(pa);             // Llama al método m1 a través de un
                                  // apuntador a una función
  void (*m2func_pointer)(A *);    // function pointer variable
  m2func_pointer = reinterpret_cast<void (*)(A*)>(vt[1]);
  m2func_pointer(pa);             // llamado al método m2 a través de
                                  // un apuntador a función
}

int main()
{
  A a;    B b1;   B b2;
  cout << "a: " << endl;   print_vtable (&a);
  cout << "b1: " << endl;  print_vtable (&b);
  cout << "b2: " << endl;  print_vtable (&b2);
  return 0;
}
```

Figura 7.23: Un ejemplo más complicado

Figura 7.24: Representación interna de `b1`

```
a:
vtable address = 004120E8
dword 0: 00401320
dword 1: 00401350
A::m1()
A::m2()
b1:
vtable address = 004120F0
dword 0: 004013A0
dword 1: 00401350
B::m1()
A::m2()
b2:
vtable address = 004120F0
dword 0: 004013A0
dword 1: 00401350
B::m1()
A::m2()
```

Figura 7.25: Salida del programa de la Figura 7.23

El lector atento se sorprenderá por qué los métodos de la clase en la Figura 7.21 están declarados explícitamente para usar la conveción de llamado de usando la plabra clave `__cdecl`. Por omisión, Microsoft usa una convención de llamado diferente a la convención normal de C para los métodos de las clases de C++. Él pasa un apuntador al objeto sobre el que actúa el método en el registro `ECX` en lugar de usar la pila. La pila se sigue usando para los otros parámetros explícitos del método. El modificador `__cdecl` le dice que use la convención de llamado estándar de C. Borland C++ usa la convención de llamado de C por omisión.

Ahora miraremos un ejemplo un poco más complicado (Figura 7.23). En él, las clases `A` y `B` cada una tiene dos métodos `m1` y `m2`. Recuerde que ya que la clase `B` no define su propio método `m2`, hereda el método de la clase `A`. La Figura 7.24 muestra como un objeto `b` se ve en memoria. La Figura 7.25 muestra la salida del programa. Primero observe la dirección de vtable para cada objeto. Las dos direcciones de los objetos `B` son iguales y así, ellos comparten la misma vtable. Una vtable es una propiedad de la clase no un objeto (como un miembro `static`). Ahora mire en las direcciones en las vtables. Mirando la salida en ensamblador uno puede determinar que el

apuntador al método `m1` es un desplazamiento cero (o palabra doble 0) y `m2` es un desplazamiento de 4 (palabra doble 1). Los apuntadores al método `m2` son los mismos vtables de las clases `A` y `B` porque la clase `B` hereda el método `m2` de la clase `A`.

Las líneas 25 a 32 muestran como uno podría llamar a una función virtual leyendo su dirección de la vtable para el objeto.[15]. La dirección del método se almacena en un apuntador a función de tipo C con un apuntador explícito *this*. De la salida en la Figura 7.25, uno puede ver que esto trabaja. Sin embargo, por favor *no* escriba código como este. Es solo para ilustrar cómo los métodos virtuales usan la vtable.

Hay algunas lecciones prácticas para aprender de esto. Un hecho importante es que uno tendría que tener mucho cuidado cuando lee y escribe variables de una clase a un archivo binario. Uno no puede sóo usar una lectura o escritura binaria en un objeto completo ya que esto podría leer o escribir la apuntador vtable al archivo. Este es un apuntador que señala a donde la vtable reside en la memoria del programa y variará de programa a programa. Este mismo problema puede ocurrir con estructuras, pero en C, las estructuras sólo tienen apuntadores en ellas si el programador las coloca explícitamente en ellas. No hay apuntadores obvios definidos en las clases `A` o `B`.

Otra vez, es importante tener en cuenta que cada compilador implementa los métodos virtuales a su manera. En Windows, los objetos de las clases COM (Component Object Model) usan las vtables para implementar interfaces COM[16]. Sólo los compiladores que implementan métodos virtuales como Microsoft pueden crear clases COM. Esta es la razón por la cual Borland usa la misma implementación que Microsoft y una de las principales razones por las cuales *gcc* no se puede usar para crear clases COM.

El código para el método virtual parece exáctamente como un método no virtual. Sólo el código que llama es diferente. Si el compilador puede estar absolutamente seguro de qué método virtual será llamado, puede ignorar la vtabley llamar el método directamente, (usando el primer vínculo).

7.2.6. Otras características de C++

Las otras características de C++ (información en tiempo de ejecución, manejo de excepciones y herencia múltiple) están más allá del alcance de este texto. Si el lector desea avanzar más, un buen punto de partida es *The Annotated C++ Reference Manual* por Ellis and Stroustrup y *The Design and Evolution of C++* por Stroustrup.

---

[15] Recuerde que este código solo trabaja con los compiladores de Borland y MS, no para *gcc*.

[16] Las clases COM también usan la convención de llamado code __stdcall no la normal de C.

Apéndice A

Instrucciones del 80x86

A.1. Instrucciones para enteros

Esta sección lista y describe las acciones y formatos de las instrucciones para enteros de la familia de Intel 80x86

Los formatos usan las siguientes abreviaturas:

R	registro general
R8	registro de 8 bits
R16	registro de 16 bits
R32	registro de 32 bits
SR	registro de segmento
M	memoria
M8	byte
M16	palabra
M32	palabra doble
I	valor inmediato

Las abreviaciones se pueden combinar para las instrucciones con varios operandos. Por ejemplo el formato *R,R* significa que la instrucción toma dos operandos de registro. Muchas de las instrucciones con dos operandos permiten los mismos operandos. La abreviación *O2* se usa para representar los siguientes operandos: *R,R R,M R,I M,R M,I*. Si se puede usar como operando un registro de 8 bits o la memoria, se usa la abreviación *R/M8*

La tabla también muestra cómo afectan las instrucciones varias de las banderas del registro FLAGS. Si la columna está en vacía, el bit correspondiente no se afecta. Si el bit siempre cambia a algún valor en particular, se muestra un 1 o un 0 en la columna. Si el bit cambia a un valor que depende del operando de la instrucción se coloca una *C* en la columna. Finalmente, si el bit es modificado de alguna manera no definida se coloca un *?* en

la columna. Ya que las únicas instrucciones que cambian las banderas de dirección son `CLD` y `STD`, no se listan bajo la columna de FLAGS.

			Banderas					
Nombre	**Descripción**	**Formatos**	**O**	**S**	**Z**	**A**	**P**	**C**
`ADC`	Suma con carry	O2	C	C	C	C	C	C
`ADD`	Suma enteros	O2	C	C	C	C	C	C
`AND`	AND entre bits	O2	0	C	C	?	C	0
`BSWAP`	Byte Swap	R32						
`CALL`	Llamado a rutina	R M I						
`CBW`	Convierta byte a palabra							
`CDQ`	Convierte Dword a Qword							
`CLC`	Borra el Carry							0
`CLD`	borra la bandera de dirección							
`CMC`	Complementa el carry							C
`CMP`	Compara enteros	O2	C	C	C	C	C	C
`CMPSB`	Compara Bytes		C	C	C	C	C	C
`CMPSW`	Compara Words		C	C	C	C	C	C
`CMPSD`	Compara Dwords		C	C	C	C	C	C
`CWD`	Convierte Word a Dword en DX:AX							
`CWDE`	Convierte Word a Dword en EAX							
`DEC`	Decrementa entero	R M	C	C	C	C	C	
`DIV`	División sin signo	R M	?	?	?	?	?	?
`ENTER`	Hace el marco de la pila	I,0						
`IDIV`	División con signo	R M	?	?	?	?	?	?
`IMUL`	Multiplicación con signo	R M R16,R/M16 R32,R/M32 R16,I R32,I R16,R/M16,I R32,R/M32,I	C	?	?	?	?	C
`INC`	Incremento entero	R M	C	C	C	C	C	
`INT`	Genera una interrupción	I						
`JA`	Salta si está sobre	I						

Nombre	Descripción	Formatos	Banderas					
			O	**S**	**Z**	**A**	**P**	**C**
`JAE`	Salta si está sobre o es igual	I						
`JB`	Salta si está bajo	I						
`JBE`	Salta si está bajo o es igual	I						
`JC`	Salta si hay carry	I						
`JCXZ`	Salta si CX = 0	I						
`JE`	Salta si es igual	I						
`JG`	Salta si es mayor	I						
`JGE`	Salta si es mayor o igual	I						
`JL`	Salta si es menor	I						
`JLE`	Salta si es menor o igual	I						
`JMP`	Salto incondicional	R M I						
`JNA`	Salta si no está sobre	I						
`JNAE`	Salta si no está sobre o es igual	I						
`JNB`	Salta si no está bajo	i						
`JNBE`	Salta si no está bajo o es igual	I						
`JNC`	Salta si no hay carry	I						
`JNE`	Salta si no es igual	I						
`JNG`	Salta si no es mayor	I						
`JNGE`	Salta si no es mayor o es igual	I						
`JNL`	Salta si no es menor	I						
`JNLE`	Salta si no es menor o igual	I						
`JNO`	Salta si no hay desborde	I						
`JNS`	Salta si es positivo	I						
`JNZ`	Salta si no es cero	I						
`JO`	Salta si hay desborde	I						
`JPE`	Salta si hay paridad par	I						
`JPO`	Salta si hay paridad impar	I						
`JS`	Salta si hay signo	I						
`JZ`	Salta si hay cero	I						

Nombre	Descripción	Formatos	Banderas					
			O	S	Z	A	P	C
LAHF	Carga FLAGS en AH							
LEA	Carga dirección efectiva	R32,M						
LEAVE	Abandona el marco de la pila							
LODSB	Carga byte							
LODSW	Carga palabra							
LODSD	Carga palabra doble							
LOOP	Bucle	I						
LOOPE/LOOPZ	Bucle si es igual	I						
LOOPNE/LOOPNZ	Bucle si no es igual	I						
MOV	Mueve datos	O2 SR,R/M16 R/M16,SR						
MOVSB	Mueve byte							
MOVSW	Mueve palabra							
MOVSD	Mueve palabra doble							
MOVSX	Mueve con signo	R16,R/M8 R32,R/M8 R32,R/M16						
MOVZX	Mueve sin signo	R16,R/M8 R32,R/M8 R32,R/M16						
MUL	Multiplicación sin signo	R M	C	?	?	?	?	C
NEG	Negación	R M	C	C	C	C	C	C
NOP	No Opera							
NOT	Complemento a 1	R M						
OR	OR entre bits	O2	0	C	C	?	C	0
POP	Saca de la pila	R/M16 R/M32						
POPA	Saca todo de la pila							
POPF	Saca FLAGS		C	C	C	C	C	C
PUSH	Empuja en la pila	R/M16 R/M32 I						
PUSHA	Empuja todo en la pila							
PUSHF	Empuja FLAGS							
RCL	Rota a la izquierda con carry	R/M,I R/M,CL	C					C

Nombre	Descripción	Formatos	Banderas					
			O	S	Z	A	P	C
RCR	Rota a la derecha con carry	R/M,I R/M,CL	C					C
REP	Repite							
REPE/REPZ	Repite si es igual							
REPNE/REPNZ	Repite si no es igual							
RET	Retorno							
ROL	Rota a la izquierda	R/M,I R/M,CL	C					C
ROR	Rota a la derecha	R/M,I R/M,CL	C					C
SAHF	Copia AH en FLAGS			C	C	C	C	C
SAL	Dezplazamiento a la izquierda	R/M,I R/M, CL						C
SBB	Resta con préstamo	O2	C	C	C	C	C	C
SCASB	busca un Byte		C	C	C	C	C	C
SCASW	Busca una palabra		C	C	C	C	C	C
SCASD	Busca una palabra doble		C	C	C	C	C	C
SETA	fija sobre	R/M8						
SETAE	fija sobre o igual	R/M8						
SETB	fija bajo	R/M8						
SETBE	fija bajo o igual	R/M8						
SETC	fija el carry	R/M8						
SETE	fija igual	R/M8						
SETG	fija mayor	R/M8						
SETGE	fija mayor o igual	R/M8						
SETL	fija menor	R/M8						
SETLE	fija menor o igual	R/M8						
SETNA	fija no sobre	R/M8						
SETNAE	fija no sobre o igual	R/M8						
SETNB	fija no bajo	R/M8						
SETNBE	fija no bajo o igual	R/M8						
SETNC	fija no carry	R/M8						
SETNE	fija no igual	R/M8						
SETNG	fija no mayor	R/M8						
SETNGE	fija no mayor o igual	R/M8						
SETNL	fija no menor	R/M8						
SETNLE	fijan no menor o igual	R/M8						
SETNO	fija no desborde	R/M8						
SETNS	fija no signo	R/M8						

Nombre	Descripción	Formatos	Banderas					
			O	**S**	**Z**	**A**	**P**	**C**
SETNZ	fija no cero	R/M8						
SETO	fija desborde	R/M8						
SETPE	fija paridad par	R/M8						
SETPO	fija paridad impar	R/M8						
SETS	fija signo	R/M8						
SETZ	fija cero	R/M8						
SAR	Desplazamiento aritmético a la derecha	R/M,I R/M, CL						C
SHR	Desplazamiento lógico a la derecha	R/M,I R/M, CL						C
SHL	Desplazamiento lógico a la izquierda	R/M,I R/M, CL						C
STC	fija el carry							1
STD	fija la bandera de dirección							
STOSB	almacena byte							
STOSW	almacena palabra							
STOSD	almacena palabra doble							
SUB	resta	O2	C	C	C	C	C	C
TEST	comparación lógica	R/M,R R/M,I	0	C	C	?	C	0
XCHG	Intercambio	R/M,R R,R/M						
XOR	XOR entre bits	O2	0	C	C	?	C	0

A.2. Instrucciones de punto flotante

En esta sección, se describen muchas de las instrucciones del coprocesador matemático del 80x86. La sección de descripción describe brevemente la operación de la instrucción. Para ahorrar espacio, la información sobre si la instrucción saca de la pila no se da en la descripción.

La columna de formato muestra que tipo de operandos se pueden usar con cada instrucción. Se usan las siguientes abreviaturas

STn	Un registro del coprocesador
F	número de precisión simple en memoria
D	número de precisión doble en memoria
E	número de precisión extendida en memoria
I16	palabra entera en memoria
I32	palabra doble entera en memoria
I64	palabra cuádruple entera en memoria

Las instrucciones que requieren un Pentium Pro o posterior están marcadas con un asterisco (*).

Instrucción	Descripción	Formato
`FABS`	`ST0` = |`ST0`|	
`FADD` *src*	`ST0 +=` *src*	STn F D
`FADD` *dest*`, ST0`	*dest* `+= ST0`	STn
`FADDP` *dest*`[,ST0]`	*dest* `+= ST0`	STn
`FCHS`	`ST0` = $-$`ST0`	
`FCOM` *src*	Compara `ST0` y *src*	STn F D
`FCOMP` *src*	Compara `ST0` y *src*	STn F D
`FCOMPP` *src*	Compara `ST0` y `ST1`	
`FCOMI`* *src*	Compara en FLAGS	STn
`FCOMIP`* *src*	Compara en FLAGS	STn
`FDIV` *src*	`ST0 /=` *src*	STn F D
`FDIV` *dest*`, ST0`	*dest* `/= ST0`	STn
`FDIVP` *dest*`[,ST0]`	*dest* `/= ST0`	STn
`FDIVR` *src*	`ST0 =` *src*`/ST0`	STn F D
`FDIVR` *dest*`, ST0`	*dest* `= ST0/`*dest*	STn
`FDIVRP` *dest*`[,ST0]`	*dest* `= ST0/`*dest*	STn
`FFREE` *dest*	Marca como vacío	STn
`FIADD` *src*	`ST0 +=` *src*	I16 I32
`FICOM` *src*	Compara `ST0` y *src*	I16 I32
`FICOMP` *src*	Compara `ST0` y *src*	I16 I32
`FIDIV` *src*	`ST0 /=` *src*	I16 I32
`FIDIVR` *src*	`ST0 =` *src*`/ST0`	I16 I32

Instrucción	Descripción	Formato
`FILD` *src*	Push *src* on Stack	I16 I32 I64
`FIMUL` *src*	`ST0 *=` *src*	I16 I32
`FINIT`	Inicia el coprocesador	
`FIST` *dest*	almacena `ST0`	I16 I32
`FISTP` *dest*	almacena `ST0`	I16 I32 I64
`FISUB` *src*	`ST0 -=` *src*	I16 I32
`FISUBR` *src*	`ST0 =` *src* `- ST0`	I16 I32
`FLD` *src*	empuja *src* en la pila	STn F D E
`FLD1`	empuja 1.0 en la pila	
`FLDCW` *src*	Load Control Word Register	I16
`FLDPI`	empuja π en la pila	
`FLDZ`	empuja 0.0 en la pila	
`FMUL` *src*	`ST0 *=` *src*	STn F D
`FMUL` *dest*`, ST0`	*dest* `*= ST0`	STn
`FMULP` *dest*`[,ST0]`	*dest* `*= ST0`	STn
`FRNDINT`	Redondea `ST0`	
`FSCALE`	$\texttt{ST0} = \texttt{ST0} \times 2^{\lfloor \texttt{ST1} \rfloor}$	
`FSQRT`	$\texttt{ST0} = \sqrt{\texttt{ST0}}$	
`FST` *dest*	almacena `ST0`	STn F D
`FSTP` *dest*	almacena `ST0`	STn F D E
`FSTCW` *dest*	Store Control Word Register	I16
`FSTSW` *dest*	Store Status Word Register	I16 AX
`FSUB` *src*	`ST0 -=` *src*	STn F D
`FSUB` *dest*`, ST0`	*dest* `-= ST0`	STn
`FSUBP` *dest*`[,ST0]`	*dest* `-= ST0`	STn
`FSUBR` *src*	`ST0 =` *src*`-ST0`	STn F D
`FSUBR` *dest*`, ST0`	*dest* `= ST0-`*dest*	STn
`FSUBP` *dest*`[,ST0]`	*dest* `= ST0-`*dest*	STn
`FTST`	Compare `ST0` with 0.0	
`FXCH` *dest*	Intercambia `ST0` y *dest*	STn

Índice alfabético