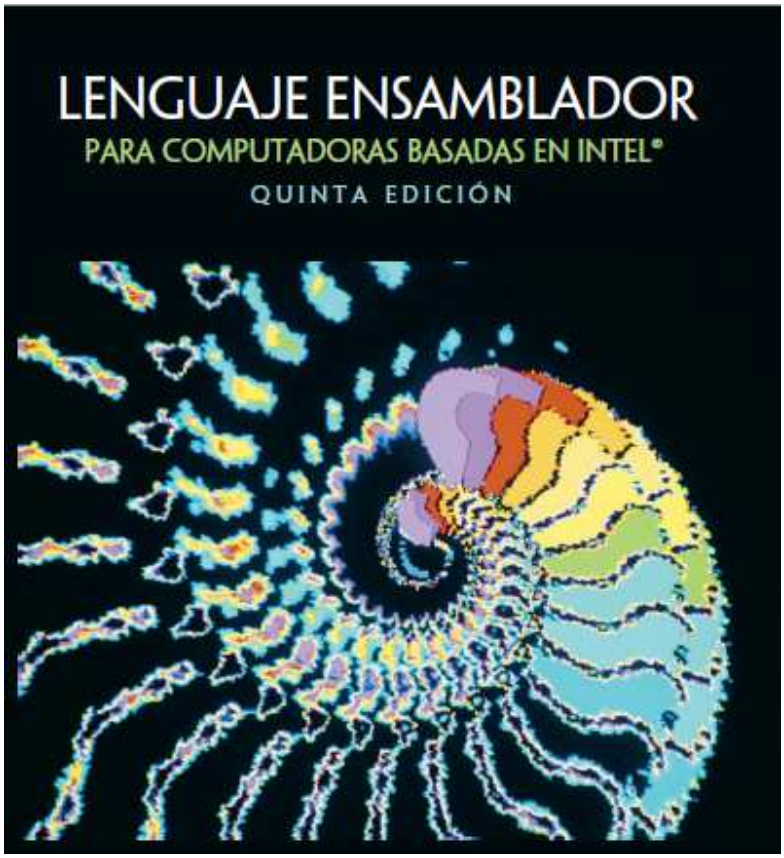
Lenguaje ensamblador para computadoras basadas en Intel



Autor del libro: Kip R. Irvine
5ta edición
Resumen: Victor Hugo Gutiérrez Gutiérrez

Dedicado a todas aquellas personas que luchan por sus sueños, a mi padre Jaime Gutiérrez, y en especial a Bianca Faría quien me alentó cada momento en este proyecto.

Hacer un resumen de un libro con tanta cantidad fundamental de conocimiento es una tarea algo difícil, clasifico y a la vez defino resumen como la ovación de aquellos detalles minúsculos que no afectan en el proceso de a prendimiento del lector, se han eliminado los resúmenes de capítulos (estos mostrados en la parte final de cada capítulo), ejercicios de programación, bibliografías, apéndices, etc.

Índice:

1 Conceptos básicos:

Bienvenidos al lenguaje ensamblador

Ensamblador: Convierte un código de fuente en lenguaje ensamblador a código de máquina.

Enlazador: Combina los archivos individuales creados por un ensamblador en un solo programa ejecutable.

Interprete de microcódigo: Programa dentro del procesador encargado de interpretar las instrucciones de un programa en lenguaje ensamblador cuando estas no coinciden directamente con la arquitectura de la computadora (cuando está en ejecución el programa).

Programas embebidos: Son programas cortos que se almacenan en un espacio reducido de memoria.

Portabilidad: Lenguajes cuyos programas de código de fuente pueden compilarse y ejecutarse en cualquier SO.

Controlador de dispositivo: Son programas que traducen los comandos generales del sistema operativo en referencias específicas a los detalles de hardware que solo el fabricante conoce.

Concepto de máquina virtual

Por lo general una computadora ejecuta programas escritos en su *lenguaje máquina nativo* a este lenguaje lo llamaremos **L0**, para los programadores es muy dificultoso escribir en **L0** (*Lenguaje máquina nativo*), por lo que se crea **L1** un lenguaje más fácil de usar hay dos formas para lograr esto:

Interpretación: De esta manera a medida que se ejecuta el programa en L1 este se va decodificando en lenguaje L0 por un programa escrito en L0 encargado de realizar la interpretación.

Traducción: El programa en L1 se convierte en L0 debido a un programa en L0 que se encarga de traducirlo. Después de esto el programa podría ejecutarse directamente desde el hardware de la computadora.

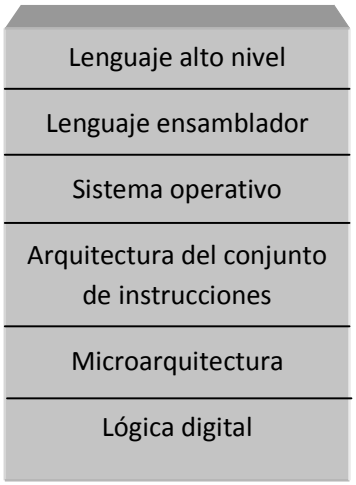
Máquinas Virtuales: en vez de tan solo utilizar lenguajes es más fácil verlo en términos de una computadora ficticia o máquina virtual **VM**, VM0 será la máquina virtual que utiliza L0 y VM1 L1

Una VM no puede ser completamente distinta a la otra debido a que el proceso *interpretación* o *traducción* sería muy largo.

Si el lenguaje que soporta **VM1** no es muy amigable para el programador se podría diseñar (hardware o software) una **VM** hasta que *VMn* fuera fácil de programar.

Máquinas específicas:

Relacionaremos estos conceptos con las computadoras y los lenguajes reales usando nombres como **VM0->Nivel 0**, **VM1->Nivel 1**:



Nivel 0->Lógica digital: Hardware lógico digital

Nivel 1->Micro arquitectura: Los comandos de micro arquitectura específicos son un secreto del propietario y por lo general no se permiten que los usuarios promedio escriban microinstrucciones, podrían requerirse 3 hasta 4 instrucciones en microcódigo para llegar a cabo una instrucción primitiva.

Nivel 2->Arquitectura del conjunto de instrucciones: Primer nivel en donde el usuario puede escribir un programa, este consiste en números binarios este lenguaje es llamado "*lenguaje máquina convencional*" cada instrucción en lenguaje máquina se ejecuta mediante varias microinstrucciones.

Nivel 3-> Sistema Operativo

Nivel 4->Lenguaje ensamblador

Nivel 5->Lenguaje de alto nivel

Representación de datos

Sistema	Base	Dígitos posibles
Binario	2	0 1
Octal	8	0 1 2 3 4 5 6 7
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

Binario: Son números en base 2 en donde cada dígito binario (bit) es un 0 o un 1, los bits se enumeran empezando desde cero en la parte derecha y se incrementan hacia la izquierda. El bit más a la izquierda se conoce como "*bit más significativo*" (**MSB**) y el bit de más a la derecha es el "*bit menos significativo*" (**LSB**)

Números bits MSB Y LBS de un número binario de 16 dígitos:

MSB **LSB**

1 0 1 1 0 0 1 0 1 0 0 1 1 1 0 0

15 0

Traducción de enteros binarios sin signo a decimal:

EJ: **00001001** equivale a **9**

Enumeramos desde LSB (*bit menos significativo*) hasta MSB (bit más significativo) de 0 a "n"

0 0 0 0 1 0 0 1

7 6 5 4 3 2 1 0

*Multiplicamos (1x2^**X**) + (1x2^**X**)*

***X**=equivale a la posición en donde está ubicado el primer bit desde LSB a MSB de 0 a "n"

(1x2^0) + (1x2*^3)=9

Traducción de entero decimal sin signo a binario:

Se debe dividir en forma repetida el número entero decimal sin signo con 2, se debe guardar cada residuo o resto como binario y luego tomar el cociente y volverlo a dividir con 2.

EJ: 37

División	Cociente	Residuo
37/2	18	1
18/2	9	0
9/2	4	1
4/2	2	0
2/2	1	0
1/2	0	1

Se recolecta de manera inversa obteniendo: 100101, Como el almacenamiento en las computadoras Intel siempre consisten en números binarios de longitud 8 bits se debe rellenar con 0 a la izquierda: **00**100101

Suma binarias:

Existen 4 maneras de sumar dígitos binarios:

0+0=0	0+1=1
1+0=1	1+1=10

Se debe comenzar la suma desde LSB hasta MSB

EJ:

0 0 0 0 0 1 0 0

+ 0 0 0 0 0 1 1 1

0 0 0 0 1 0 1 1

Tamaño de almacenamiento de enteros:

La unidad básica de almacenamiento para todos los datos en una computadora basada en IA-32 es un byte que contiene 8 bits. Otros tamaños de almacenamiento son:

-*palabra* (2 bytes)

-*doble palabra* (4 bytes)

-*palabra cuádruple* (8 bytes)

Un Kilobyte es igual a 2^10 o 1024 bytes

Un megabyte es igual a 2^20 o 1,048,576 bytes

Un gigabyte es igual a 2^30 o 1,073,741,824 bytes

Un terabyte es igual a 2^40 o 1,099,511,627,776 bytes

Un petabyte es igual 2^50 o 1,125,899,906,842,624 bytes

Un exabyte es igual a 2^60 o 1,152,921,504,606,846,976 byte

Un zettabyte es igual a 2^70 bytes

Un yottabyte es igual a 2^80 bytes

Enteros hexadecimales:

Decimal:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Hexadecimal:

1 2 3 4 5 6 7 8 9 A B C D E F

Conversión de hexadecimal sin signo decimal:

Ejemplo nº1:

(Hexadecimal) 1234 equivale a 4660 (decimal)

Enumeramos desde 0 a “n” partiendo desde 0 y desde la derecha:

1 2 3 4

3 2 1 0

(4x16^0) + (3x16^1) + (2x16^2) + (1x16^3)= 4660

Ejemplo nº2:

(Hexadecimal) 3BA4 equivale a 15268 (decimal)

Enumeramos desde 0 a “n” partiendo desde 0 y desde la derecha:

3 B A 4

3 2 1 0

(4x16^0) + (10x16^1) + (11x16^2) + (3x16^3)= 15268

Conversión de decimal sin signo a hexadecimal:

Se debe dividir en forma repetida el valor decimal por 16 luego conservar el residuo o resto de cada división como dígito hexadecimal.

EJ:

422 (decimal)

División	Cociente	Residuo
422/16	26	6
26/16	1	A
1/16	0	1

Se recolecta de manera inversa obteniendo 1A6

Enteros con signo:

Los enteros binarios con signos son positivos o negativos. En las computadoras basadas en Intel, el bit más significativo (MSB) indica:

0: positivo

1: negativo

Ejemplo:

1 1 1 1 0 1 1 0->Negativo

0 0 0 0 1 0 1 0->Positivo

Notación de complemento a dos:

Los enteros negativos utilizan la representación de *complemento a dos*, en base al principio matemático que establece que el complemento a dos de un entero es su inverso aditivo (si se suma un número a su inverso aditivo el resultado es cero)

La representación en complemento a dos es útil para los diseñadores de procesadores ya que elimina la necesidad de tener circuitos digitales separados para manejar tanto la suma como la resta. Por ejemplo, si presentamos al procesador la expresión A – B. tan solo necesita convertirla en expresión de suma

A + (-B)

Para formar el complemento de dos de un entero binario se invierten (complementan) sus bits y se les suma 1

EJ: 00000001

Valor Inicial: 00000001

Paso 1: invertir los bits 11111110

Paso2: Sumar 1 al valor de paso 1

Suma: representación a complemento de dos 11111111

11111111 es la representación en complemento a dos de -1, esta operación es reversible por lo que el complemento a dos de 11111111 es 00000001

Complemento a dos de hexadecimal:

Se debe invertir los bits y sumarle uno

Ejemplo:

63AD

0110 1010 0011 1101

Complemento de dos:

1001 0101 1100 0010

+ 1

= 1001 0101 1100 0011

95C3

Conversión de binario con signo a decimal:

- Si el bit más alto (MSB) es 1, el número está almacenado en notación de complemento a dos. Se debe formar nuevamente el complemento a dos una segunda vez para obtener el equivalente positivo, luego se convierte el número a decimal.

  Ejemplo: 11110000 (binario)

  Valor inicial: 11110000

  Paso 1: Invertir los bits: 00001111

  Paso 2: Sumar 1 al valor del paso 1: 00001111+1

  Paso 3: Formar el complemento a dos: 00010000

  Paso 4: Convertir en decimal: 16

  Como el valor era negativo, inferimos que su valor decimal era **-16**

- Si el bit más alto (MSB) es 0, se puede convertir en decimal como si fuera un entero binario sin signo.

Conversión de entero decimal a binario:

Paso 1: Convertir el valor absoluto del entero a decimal binario

Paso 2: Si el entero original era negativo, se debe formar el complemento de dos del número binario del paso anterior

Conversión de hexadecimal con signo a decimal:

Paso 1: Si el entero hexadecimal es negativo, se debe formar el complemento de dos, si no es negativo se debe mantener el valor

Paso 2: El entero del paso anterior se convierte a decimal, si este era negativo se debe agregar el símbolo (-)

*Hexadecimal negativo: el primer valor de izquierda a derecha debe ser mayor o igual a 8 (> o =)

Hexadecimal positivo: el primer valor de izquierda a derecha debe ser menor o igual a 7 (< o =)

Operaciones booleanas

NOT: ¬ o ~ o ‘

AND: ^ o ·

OR: ˅o +

NOT: Puede escribirse en notación matemática como ¬X, en donde X es una variable (o expresión) que contiene un valor verdadero (V) o falso (F)

Tabla NOT:

X	**¬X**
F	V
V	F

Una tabla de verdad puede usar:

0=falso

1=verdadero

AND: La operación AND requiere dos operandos y puede expresarse mediante la notación X^Y

Tabla AND:

X	Y	X^Y
F	F	F
F	V	F
V	F	F
V	V	V

AND solo es verdadero si X e Y son V, en todos los otros casos son falso

OR: La operación OR requiere dos operandos y a menudo se expresa mediante la notación X ᵛ Y

Tabla OR:

X	Y	X ᵛ Y
F	F	F
F	V	V
V	F	V
V	V	V

El resultado es F solo cuando ambas variables (X e Y) son F

Procedencia de los operadores:

Mayor a menor:

-NOT

-AND

-OR

2 Arquitectura del procesador IA-32:

Conceptos generales:

Describiremos la arquitectura de la familia de los procesadores IA-32 de Intel y su sistema computacional anfitrión desde el punto de vista del programador.

El lenguaje ensamblador es una herramienta excelente para aprender el funcionamiento de una computadora. Para lo cual es necesario tener un conocimiento práctico acerca del hardware de la computadora.

Diseño básico de una microcomputadora:

CPU: (*Unidad central de procesamiento*), es en donde se realizan los cálculos y las operaciones lógicas, contiene un número limitado de lugares de almacenamiento, conocidas como *registros* Además contiene un reloj de alta frecuencia, una unidad de control y una unidad de aritmética-lógica.

- Reloj de alta frecuencia: Sincroniza las operaciones internas de la CPU con los demás componentes del sistema.
- Unidad de control (CU): Coordina la secuencia de los pasos involucrados en la ejecución de instrucciones de máquina.
- Unidad aritmética-lógica: (ALU): Realiza operaciones aritméticas como la suma y la resta, y operaciones lógicas como AND,OR y NOT

La CPU se une a al resto de la computadora mediante terminales que se conectan al zócalo (socket) de la CPU en la tarjeta madre de la computadora. La mayoría de las computadoras se conectan al bus de datos, el bus de control y al bus de direcciones.

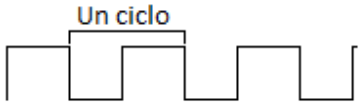
Unidad de almacenamiento de memoria: Es en donde se mantienen las instrucciones y los datos mientras se ejecuta un programa en la computadora. La unidad de almacenamiento recibe solicitudes de datos por parte de la CPU, transfiere datos de la RAM (memoria de acceso aleatorio) a la CPU y transfiere datos de la CPU a la memoria.

Un *bus* es un grupo de cables en paralelos que transfieren datos de una parte de la computadora hacia otra. Por lo general el *bus del sistema de una computadora* consiste en tres buses separados:

- bus de datos: Transfiere instrucciones y datos entre la CPU y la memoria.
- bus de control: Utiliza señales binarias para sincronizar las acciones de todos los dispositivos conectados al bus del sistema.

- bus de direcciones: Almacena direcciones de las instrucciones y datos, cuando la instrucción actual que está en ejecución transfiere datos entre la CPU y la memoria.

Reloj: Unidad básica de tiempo es un ciclo de máquina (o ciclo de reloj). La longitud de un ciclo de reloj es el tiempo requerido para un pulso completo de reloj:



Ciclo de ejecución de instrucciones:

La operación de una sola instrucción de máquina puede dividirse en una secuencia de operaciones individuales, conocidas como *ciclo de ejecución de instrucciones*. Antes de ejecutarse un programa se carga en memoria. El *apuntador de instrucciones* contiene la dirección de la siguiente instrucción. La cola de instrucciones guarda un grupo de instrucciones que están a punto de ejecutarse. Para ejecutar una instrucción de máquina se requieren tres pasos: *búsqueda, decodificación y ejecución*. Cuando la instrucción utiliza un operando en memoria se requieren dos pasos más: *búsqueda de operandos* y *almacenamiento del operando del resultado*.

- Búsqueda: la unidad de control busca la instrucción en la cola de instrucciones e incrementa el apuntador de instrucciones (IP) o *contador del programa*.
- Decodificación: la unidad de control decodifica la función de la instrucción para determinar lo que ésta debe hacer. Los operandos de entrada de la instrucción se pasan a la unidad aritmética-lógica (ALU), y se envían señales a la ALU para indicar la operación que se va a realizar
- Búsqueda de operandos: si la instrucción utiliza un operando de entrada ubicado en memoria, la unidad de control utiliza una operación de *lectura* para obtener el operando y copiarlo en los registros internos. Estos registros no son visibles para los programa de los usuarios.
- Ejecución: la ALU ejecuta la instrucción utilizando los registros con nombre y los registros internos como operandos, y envía el resultado a los registros con nombre y a la memoria. La ALU actualiza las banderas de estado que proporcionan la información acerca del estado del procesador.
- Almacenamiento del operando del resultado: si el operando resultante está en memoria, la unidad de control utiliza una operación de *escritura* para almacenar el dato.

La secuencia de pasos puede expresarse en seudocódigo:

```
Iterar

Obtener la siguiente instrucción

Avanzar el apuntador de instrucciones (IP)

Decodificar la instrucción

Si se necesita un operando en memoria, leer el valor de memoria

Ejecutar la instrucción

Si el resultado es un operando en memoria, escribir el resultado en la
memoria

Continuar el ciclo
```

Canalización de varias etapas:

Cada paso en el ciclo de instrucciones requiere cuando menos un pulso del reloj del sistema, a lo cual se le conoce como *ciclo de reloj*. El procesador no tiene que esperar hasta que se completen todos los pasos antes de empezar la siguiente instrucción, sino que puede ejecutar pasos en paralelo, con la técnica conocida como canalización.

Ejecución de una instrucción no canalizada de seis etapas:

	S1	S2	S3	S4	S5	S6
1	I-1					
2		I-1				
3			I-1			
4				I-1		
5					I-1	
6						I-1
7	I-2					
8		I-2				
9			I-2			
10				I-2		
11					I-2	
12						I-2

Ejecución de una instrucción canalizada de seis etapas:

	S1	S2	S3	S4	S5	S6
1	I-1					
2	I-2	I-1				
3		I-2	I-1			
4			I-2	I-1		
5				I-2	I-1	
6					I-2	I-1
7						I-2

Arquitectura superescalar

Un procesador superescalar o multinúcleo tiene dos o más canalizaciones de ejecución. Lo cual hace posible que haya dos instrucciones en la etapa de ejecución al mismo tiempo.

Ejemplo de ejecución con canalización, utilizando una sola canalización (la etapa S4 requiere dos ciclos de reloj para que la siguiente instrucción pueda entrar al ciclo, por lo que a medida que van entrando más instrucciones se producen ciclos desperdiciados):

	S1	S2	S3	S4	S5	S6
1	I-1					
2	I-2	I-1				
3	I-3	I-2	I-1			
4		I-3	I-2	I-1		
5			I-3	I-1		
6				I-2	I-1	
7				I-2		I-1
8				I-3	I-2	
9				I-3		I-2
10					I-3	
11						I-3

Procesador escalar canalizado de 6 etapas (la etapa S4 requiere dos ciclos):

Las instrucciones con numeración par entran la canalización ***v*** mientras que las instrucciones con numeración impar entran en la canalización ***u***, eliminando los ciclos desperdiciados.

	S1	S2	S3	u	v	S5	S6
1	I-1						
2	I-2	I-1					
3	I-3	I-2	I-1				
4	I-4	I-3	I-2	I-1			
5		I-4	I-3	I-2	I-1		
6			I-4	I-3	I-2	I-1	
7				I-4	I-3	I-2	I-1
8					I-4	I-3	I-2
9						I-4	I-3
10							I-4

Lectura de la memoria

A menudo el rendimiento de un programa depende de la velocidad del acceso a la memoria. La velocidad acceso a la CPU podría ser varios gigahertz, mientras que el acceso a la memoria se realiza a través de un bus del sistema que se ejecuta con una lentitud de 33 MHz La CPU se ve obligada a esperar uno o más ciclos de reloj hasta que se buscan y se obtienen los operandos de la memoria para poder ejecutar las instrucciones. Los ciclos de reloj desperdiciados se conocen como *estados de espera*.

Memoria caché

Como la memoria convencional es mucho más lenta que la CPU, las computadoras utilizan *memoria caché* de alta velocidad para almacenar las instrucciones y datos más recientes. La primera vez que un programa lee un bloque de datos, deja una copia en el caché. Si el programa necesita leer los mismos datos una segunda vez, los busca en el caché. Un *acierto en el caché* indica que los datos se encuentran en el caché; un *fallo en el caché* indica que los datos no se encuentran en el caché y deben leerse de la memoria convencional. En general la memoria caché tiene un efecto notable cuando se trata de mejorar el acceso a los datos en especial cuando es grande.

Como se ejecutan los programas

Proceso de carga y ejecución:

- El sistema operativo (OS) busca el nombre de archivo del programa en el directorio actual, si no puede encontrar el nombre ahí, lo busca en una lista predeterminada de directorios (llamados trayectorias). Si el OS no puede encontrar el nombre del archivo del programa, muestra un mensaje de error.

- Si encuentra el nombre del archivo, el OS obtiene la información básica sobre el archivo del programa del directorio en el disco, incluyendo el tamaño del archivo y su ubicación física en la unidad de disco.

- El OS determina la siguiente ubicación disponible en memoria y carga el archivo del programa. Asigna un bloque de memoria al programa e ingresa información acerca del tamaño y la ubicación del programa en una tabla (lo que algunas veces se conoce como tabla de descriptores). Además, el OS puede ajustar los valores de los apuntadores dentro del programa, para que contengan las direcciones de los datos.

- El OS ejecuta una instrucción de salto que hace que la CPU empiece la ejecución de la primera instrucción de máquina del programa. Al momento en que el programa empieza a ejecutarse, se le denomina proceso. El OS asigna un número de identificación al proceso (ID del proceso), el cual utiliza para llevar el registro del proceso mientras se ejecuta.

- El proceso se ejecuta por sí solo. Es función del OS registrar la ejecución del proceso y responder a las solicitudes de recursos del sistema como memoria, archivos en disco y dispositivos de entrada y salida.

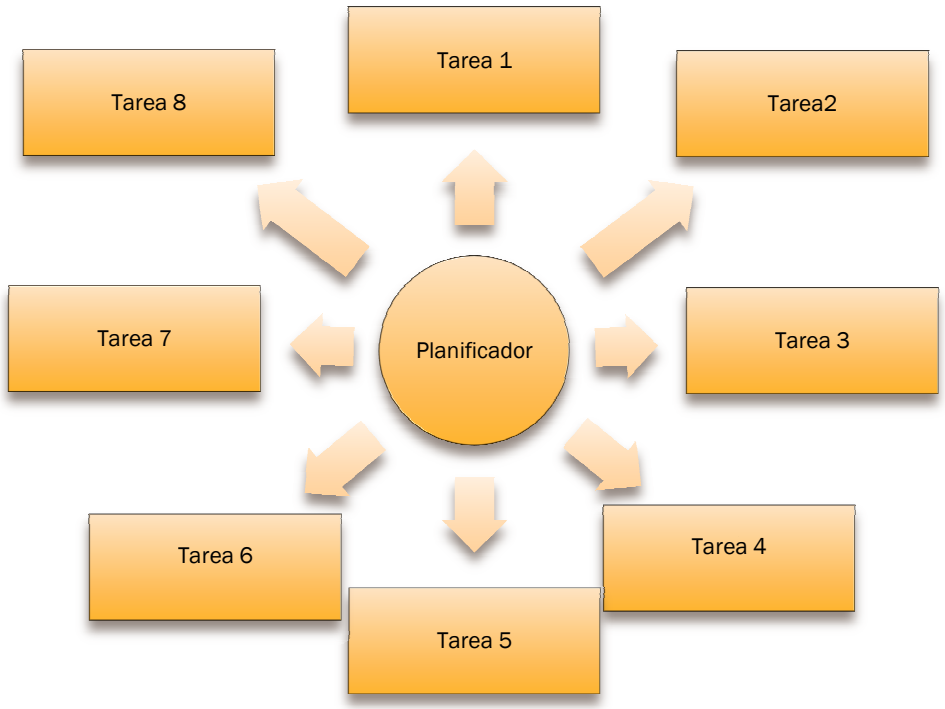
- Cuando termina el proceso, se elimina su manejador y la memoria que se utilizó se libera para que otros programas puedan utilizarla.

Multitarea

Un sistema operativo multitarea puede ejecutar varias tareas al mismo tiempo. Una tarea se define como un programa (proceso) o un hilo (subproceso) de ejecución. En realidad la CPU puede

ejecutar sólo una instrucción a la vez, por lo que un componente del sistema operativo, conocido como *planificador*, asigna una partición de tiempo de la CPU a cada tarea.

Ejemplo de tipo de planificador: *Planificador por turnos (round-robin):*



Un OS multitareas se ejecuta en un procesador con soporte para *conmutación de tareas.* El procesador almacena el estado de cada tarea antes de cambiar a una nueva. El estado de una tarea consiste en los registros del procesador, el contador del programa y las banderas de estado, junto con las referencias a los segmentos de memoria de la tarea. Por lo general, un SO multitarea asigna distintas prioridades a las tareas con la cual reciben particiones de tiempo relativamente más grandes o más pequeñas.

Arquitectura del procesador IA-32

Modo de operación

Los procesadores IA-32 tienen tres modos principales de operación: *modo protegido, modo de direccionamiento real y modo de administración de sistemas*. Existe otro modo llamado *8086 virtual*, que es un caso especial de modo protegido.

- Modo protegido: El modo protegido es el estado nativo del procesador, en la que están disponibles todas las instrucciones y características.

- Modo 8086 virtual: mientras se encuentra en modo protegido, el procesador puede ejecutar en forma directa el software por modo de direccionamiento real, como los programas de MS-DOS, en un entorno multitarea seguro. En otras palabras, si un programa de MS-DOS falla o trata de escribir datos en el área de memoria del sistema, no afectará a los otros programas que se ejecutan al mismo tiempo. Windows XP puede ejecutar varias sesiones separadas en modo 8086 a la vez.

- Modo de direccionamiento real: Este modo implementa el entorno de programación del procesador 8086 de Intel, con unas cuantas características adicionales, como la habilidad de cambiar a otros modos. Este modo está disponible en Windows 98 y puede usarse para ejecutar un programa de MS-DOS que requiera acceso directo a la memoria del sistema y a los dispositivos hardware.

- Modo de administración del sistema: El modo de administración del sistema (SMM) proporciona al sistema operativo un mecanismo para implementar funciones, como la administración de energía y la seguridad del sistema. Por lo general, estas funciones las implementan los fabricantes de computadoras.

Espacio de direcciones

Los procesadores IA-32 pueden acceder a 4G de memoria en modo protegido; este límite se basa en el tamaño de una dirección representada por un número entero binario sin signo, de 32 bits. Los programas en direccionamiento real tienen un rango de memoria de 1 MB. Si el procesador se encuentra en modo protegido y ejecuta varios programas en modo 8086 virtual, cada programa tiene su propia área de memoria de 1 MB.

Registros básicos de ejecución de un programa

Los registros son ubicaciones de almacenamiento de alta velocidad, que se encuentran directamente dentro de la CPU (Unidad central de procesamiento), y están diseñadas para una velocidad de acceso mucho mayor que la de la memoria convencional.

Registros de propósito general de 32 bits:

EAX	EBP

EBX	ESP
ECX	ESI
EDX	EDI

EFLAGS

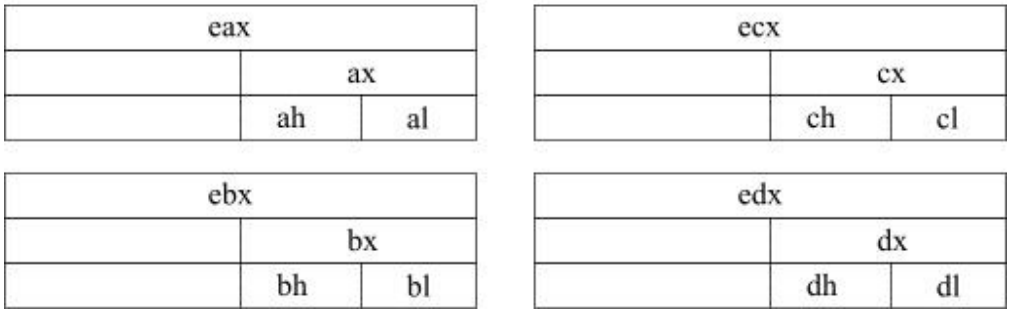
EIP

Registros de segmento de 16 bits

CS	ES
SS	FS
DS	GS

Registros de propósito general: Los registros de propósito general se utilizan principalmente para las operaciones aritméticas y el movimiento de datos.

Cada registro puede direccionarse como un valor individual de 32 bits, o como dos valores de 16 bits.



EAX=32 bits, **AX**=16 bits, **AH** (mitad superior) y **AL** (mitad inferior) =8 bits

El resto de los registros de propósito general solo tiene nombres específicos para sus 16 bits inferiores. Por lo general, los registros de 16 bits que se muestran aquí se utilizan cuando se escriben programas en modo de direccionamiento real:

32 bits	16 bits
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Usos especializados Algunos registros de propósito general tienen usos especializados:

- EAX es el registro que se utilizan de manera automática las instrucciones de multiplicación y división. A menudo se conoce como el registro *acumulador extendido*.
- La CPU utiliza de manera automática a ECX como su contador de ciclo
- ESP direcciona datos en la pila (una estructura de memoria del sistema). Se utiliza raras veces para operaciones aritméticas o de transferencia de datos ordinarias. A menudo se conoce como el registro *apuntador de pila extendido*.
- ESI y EDI son registros que utilizan las instrucciones de transferencia de memoria de alta velocidad. Algunas veces se les conoce como *registros índice de origen extendido* e *índice de destino extendido*.
- EBP es el registro que utilizan los lenguajes de alto nivel para hacer referencia a los parámetros de funciones y las variables locales en la pila. No debe utilizarse para operaciones aritméticas o de transferencia de datos ordinarias, excepto en nivel avanzado de programación. A menudo se le conoce como registro *apuntador de estructura extendido*.

Registros de segmento En el modo de direccionamiento real, los registros de segmento indican las direcciones bases de las áreas pre asignadas de memoria, conocidas como *segmentos*. En el modo protegido, los registros de segmento guardan apuntadores a tablas de descriptores de segmento. Algunos segmentos guardan instrucciones de un programa (código), otros guardan variables (datos), y otros segmento llamado *segmento de pila* guarda las variables de funciones locales y los parámetros de funciones.

Apuntador de instrucciones El registro EIP, o apuntador de instrucciones, contiene la dirección de la siguiente instrucción a ejecutar. Ciertas instrucciones de máquina manipulan a EIP, para que el programa se bifurque hacia una nueva dirección.

Registro EFLAGS El registro EFLAGS (o simplemente Flags) consiste en bits binarios individuales que controlan la operación de la CPU, o que reflejan resultados de alguna operación de la CPU. Algunas instrucciones evalúan y manipulan las banderas individuales del procesador.

(Una bandera se activa cuando es igual a 1; se desactiva cuando es igual a 0)

Banderas de control Las banderas de control controlan la operación de la CPU. Por ejemplo, pueden hacer que la CPU se salga de un ciclo después de ejecutar cada instrucción, generar una interrupción cuando se detecta un desbordamiento aritmético, entrar al modo 8086 virtual y entrar en modo protegido.

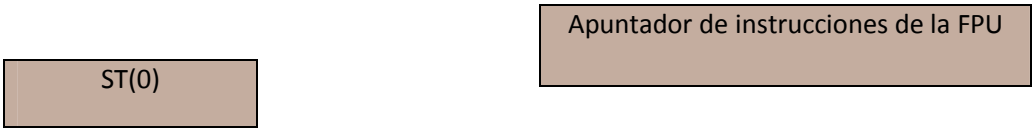
Banderas de estado Las banderas de estado reflejan el resultado de las operaciones aritméticas y lógicas que realiza la CPU. Estas banderas son: Desbordamiento, Signo, Cero, Acarreo Auxiliar, paridad y Acarreo

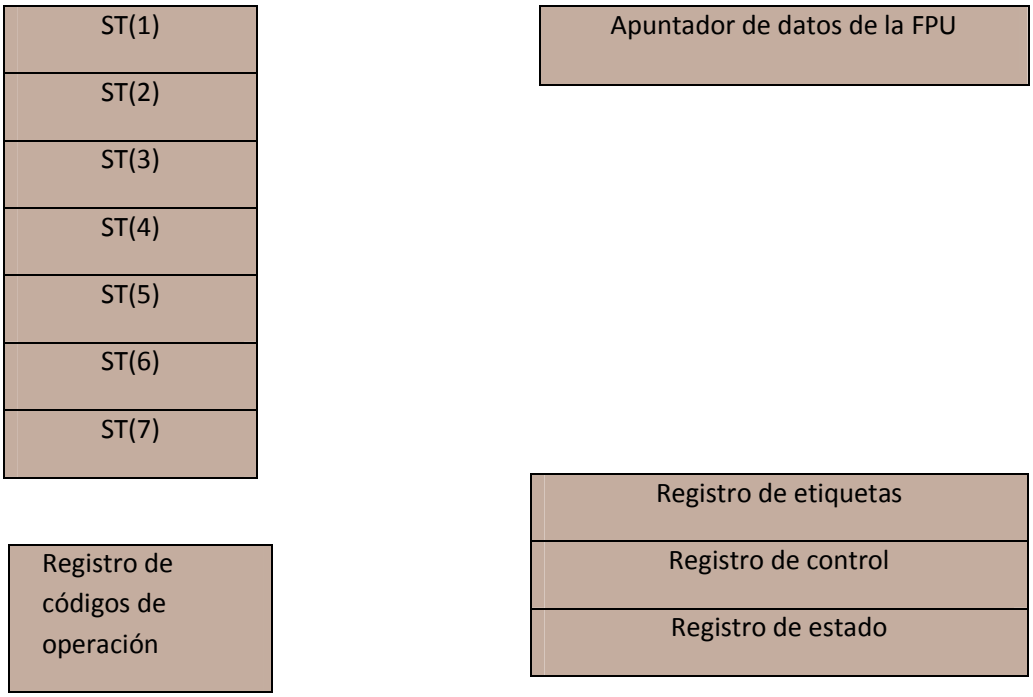
- Bandera **Acarreo** (CF): Se activa cuando el resultado de una operación aritmética *sin signo* es demasiado grande para caer en el destino.
- Bandera **Desbordamiento** (OF): Se activa cuando el resultado de una operación aritmética *con signo* es demasiado grande o pequeño para caer en el destino.
- Bandera **Signo** (SF): Se activa cuando el resultado de una operación aritmética o lógica genera un resultado negativo.
- Bandera **Cero** (ZF): Se activa cuando el resultado de una operación aritmética o lógica genera un resultado de cero.
- Bandera de **Acarreo Auxiliar** (AC): Se activa cuando una operación aritmética produce un acarreo del bit 3 al bit 4, en un operando de 8 bits
- Bandera de **Paridad** (PF) se activa si el byte menos significativo en el resultado contiene un número par de bits que sean 1.

Unidad de punto flotante

La unidad de punto flotante (FPU) de los procesadores IA-32 realiza operaciones aritméticas de punto flotante, de alta velocidad. Hace tiempo se requería un chip coprocesador separado para esto. A partir de intel486 a la fecha, la FPU está integrada en el procesador principal. Hay ocho registros de datos de punto flotante en la FPU, cuyos nombres son ST(0),ST(1),ST(2),ST(3),ST(4),ST(5),ST(6),ST(7).

Registros de datos de 80 bits





Otros registros

Existen otros dos conjuntos de registros que se utilizan para la programación avanzada con multimedia en la serie de procesadores Pentium:

- Ocho registros de 64 bits para utilizarlos con el conjunto de instrucciones MMX
- Ocho registros XMM de 128 bits que se utilizan para las operaciones de una sola instrucción y varios datos (SIMD).

Administración de memoria del procesador IA-32

Los procesadores IA-32 administran la memoria de acuerdo a los modelos básicos de operación. El modo protegido es el más simple y poderoso; los demás se utilizan, por lo general cuando los programas deben acceder directamente al hardware del sistema.

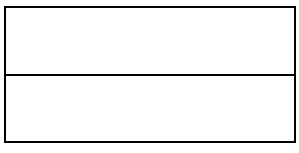
En **modo de direccionamiento real** sólo puede direccionarse 1 MB de memoria, entre el 00000 hasta FFFFF hexadecimal. El procesador sólo puede ejecutar un programa a la vez, pero puede interrumpir en formas momentánea ese programa para procesar las solicitudes (conocidas como *interrupciones*) de los periféricos. Los programas de aplicación pueden leer y modificar cualquier área de RAM (memoria de acceso aleatorio) y pueden leer pero no modificar cualquier área del ROM (memoria de solo lectura). El sistema operativo MS-DOS se ejecuta en modo de direccionamiento real, y Windows 95/98 puede cargarse en este modo.

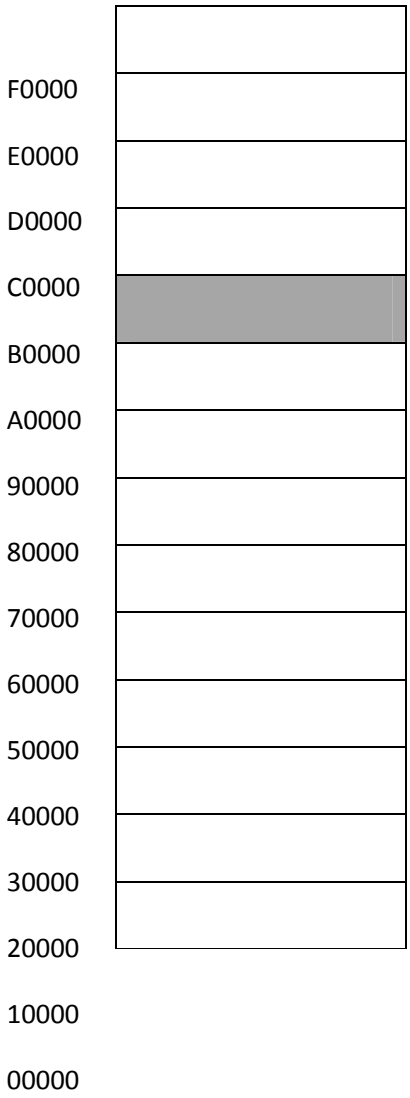
En **modo protegido**, el procesador puede ejecutar varios programas al mismo tiempo. A cada proceso (programa en ejecución) le asigna un total de 4GB de memoria. A cada programa se le puede asignar su propia área reservada de memoria, y los programas no pueden acceder de manera accidental al código y loas datos de los demás programas. MS-Windows y Linux se ejecutan en modo protegido

En **modo 8086 virtual**, la computadora se ejecuta en modo protegido y crea una máquina 8086 virtual con su propio espacio de direcciones 1MB, que simula a una programación 80 x 86 que se ejecuta en modo de direccionamiento real. Algunos programas de MS-DOS que hacen referencias directas al hardware de la computadora no se ejecutarán en este modo bajo Windows NT, 2000 y XP.

Modo de direccionamiento real

En el modo de direccionamiento real, el procesador IA-32 puede acceder 1MB de memoria mediante el uso de direcciones de 20 bits, en el rango de 0 a FFFFF hexadecimal. Los ingenieros de Intel tuvieron que resolver un problema básico: los registros de 16 bits en el procesador 8086 no podían almacenar direcciones de 20 bits, Por ende, idearon un esquema conocido como *memoria segmentada*. Toda la memoria se divide en unidades de 64kilobytes, a los cuales se les llama segmento. Cada segmento empieza en una dirección que tiene un cero en su último dígito hexadecimal. Como el último dígito siempre es cero, se omite al representar los valores de los segmentos. Por ejemplo un valor de segmento C000 hace referencia al segmento en la dirección C0000.





En la dirección 80000 para acceder a un byte en este segmento se suma un desplazamiento de 16 bits (de 0 a FFFFF) a la ubicación base del segmento. Por ejemplo, la dirección 8000:0250 representa un desplazamiento de 250 dentro del segmento que empieza en la dirección 80000. La dirección lineal es 80250h

Cálculo de direcciones lineales de 20 bits Una dirección se refiere a una ubicación individual en la memoria, y cada byte de memoria tiene una dirección distinta. En el modo de direccionamiento real, la *dirección lineal o absoluta* es de 20 bits y varía de 0 a FFFFF hexadecimal. Los programas no pueden utilizar las direcciones lineales directamente, por lo que las direcciones se expresan mediante el uso de dos enteros de 16 bits. Una dirección tipo *segmento-desplazamiento* incluye lo siguiente:

- Un valor de **segmento** de 16 bits, que se coloca en uno de los registros de segmento (CS,DS,ES,SS)
- Un valor de **desplazamiento** de 16 bits

La CPU convierte en forma automática una dirección segmento-desplazamiento en una dirección lineal de 20 bits. Supongamos que la dirección segmento-desplazamiento hexadecimal de una variable es O8F1:0100. La CPU multiplicará el valor del segmento pro 16(10 hexadecimal) y suma el producto al desplazamiento de la variable:

```
08F1 * 10h = 08F10h                    (valor de segmento ajustado)
Valor de segmento ajustado:             0F810
Se suma el desplazamiento:              0100
Dirección lineal:                       09010
```

Un programa ordinario tiene tres segmentos: código datos y pila. Los tres registros de segmento CS, DS y SS contienen las ubicaciones base de los segmentos:

- CS contiene la dirección del segmento de **código** de 16 bits
- DS contiene la dirección del segmento de **datos** de 16 bits
- SS contiene la dirección del segmento de la **pila** de 16 bits
- ES, FS y GS pueden apuntar a segmentos de datos alternativos.

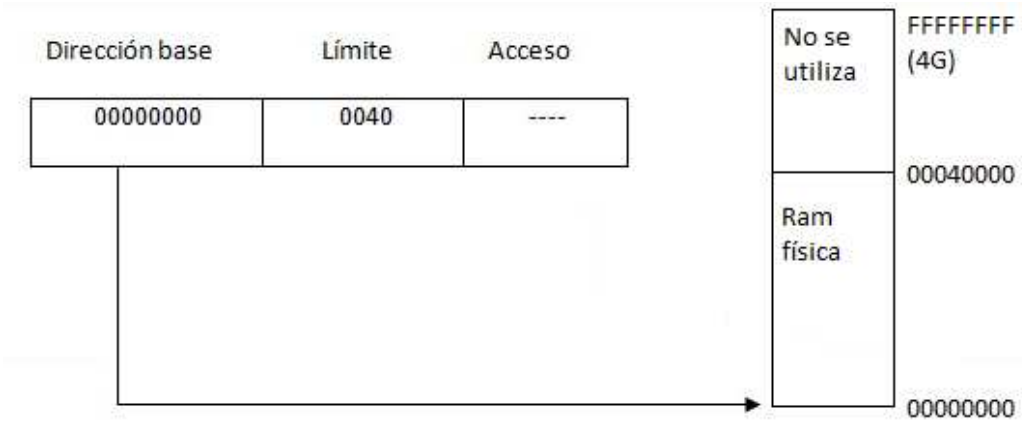
Modo protegido

El modo protegido es el modo “nativo” más poderoso del procesador. Al ejecutarse en modo protegido, un programa puede acceder a 4GB de memoria, con direcciones de 0 hasta FFFFFFFF hexadecimal. En el contexto de Microsoft Assambler, el modelo de memoria **plano** es apropiado para la programación en modo protegido. El modelo plano es fácil de usar, ya que sólo requiere un entero de 32 bits para guardar la dirección de una instrucción o variable. La CPU realiza el cálculo y traducción de las direcciones en segundo plano, todo lo cual es transparente para los programadores de aplicaciones. Los registros de segmento (CS, DS, SS, ES, FS, GS) apuntan a tablas de descriptores de segmentos, que el sistema operativo utiliza para llevar el registro de las ubicaciones de los segmentos individuales de un programa. Un programa ordinario en modo protegido tiene tres segmentos: código, datos y pila, y utiliza los registros de segmento CS.DS y SS:

- CS hace referencia a la tabla de descriptores para el segmento de código.
- DS hace referencia a la tabla de descriptores para el segmento de datos.
- SS hace referencia a la tabla de descriptores para el segmento de pila.

Modelo de segmentación plano

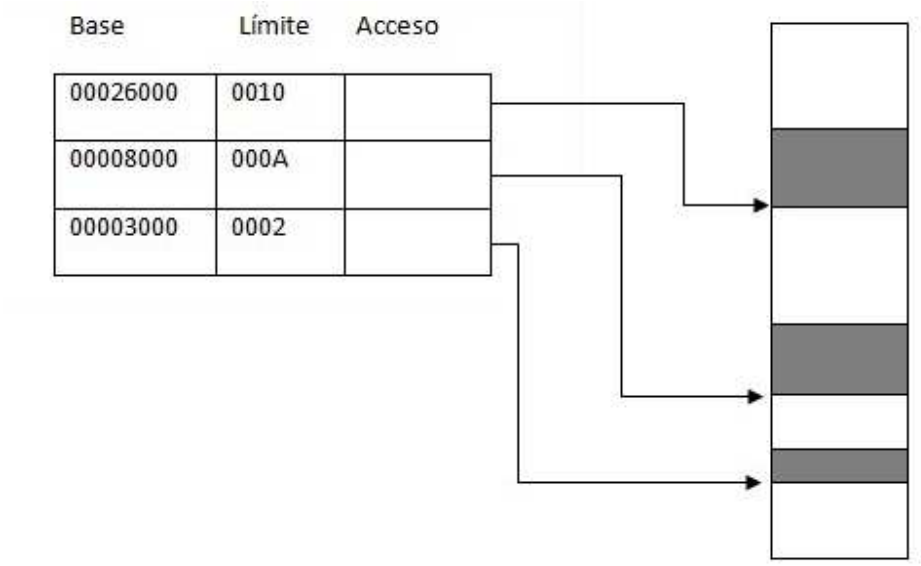
En este modelo, todos los segmentos se asignan al espacio completo de direcciones físicas de 32 bits de la computadora. Se requieren por lo menos dos segmentos, uno para código y otro para datos. Cada segmento se define mediante un *descriptor de segmento*. Un entero de 64 bits que se almacena en una tabla conocida como la *tabla de descriptores globales* (GDT).



Muestra un descriptor de segmento cuyo campo *dirección base* apunta a la primera ubicación disponible en la memoria (00000000). El campo *límite* de segmento puede indicar de manera opcional la cantidad de memoria física en el sistema. En esta figura, el límite del segmento es 0040. El campo *acceso* contiene bits que determinan cómo puede utilizarse el segmento.

Modelo de varios segmentos

En el modelo multisegmentos, cada tarea o programa recibe su propia tabla de descriptores de segmento conocida como tabla *de descriptores locales* (LDT). Cada descriptor apunta a un segmento, que puede ser distinto de los demás segmentos utilizados por otros procesos. Cada segmento tiene su propio espacio de direcciones.



Paginación

Los procesadores IA-32 tienen soporte para la paginación, una característica que permite dividir los segmentos en bloques de 4,096 bytes de memoria, conocidas como páginas. La paginación permite que el total de memoria utilizada por todos los programas que se ejecutan al mismo tiempo sea mucho más grande que la memoria física de la computadora. La colección completa de páginas asignadas por el sistema operativo se llama *memoria virtual*.

Componentes de una microcomputadora IA-32

Tarjeta madre

El corazón de una microcomputadora es una tarjeta madre, un tablero de circuitos planos en el que se colocan a la CPU de la computadora, los procesadores de soporte (juego de chips), la memoria principal, los conectores de entrada-salida, los conectores de fuente de alimentación, y las ranuras de expansión. Los diversos componentes se conectan entre sí mediante un bus, un conjunto de alambres grabados directamente en la tarjeta madre. Hay decenas de tarjetas madre disponibles en el mercado de la PCs, las cuales varían en cuanto su capacidad de expansión, los componentes integrados y la velocidad. Los siguientes componentes se encuentran de manera tradicional en la tarjeta madre de las PCs:

- Un zócalo (socket) para la CPU.
- Ranuras de memoria (SIMM o DIMM) que alojan pequeñas tarjetas de memoria insertables.
- Chips de BIOS.
- RAM de CMOS, con una pequeña batería circular para mantenerla energizada.
- Conectores para los dispositivos de almacenamiento masivo, como discos duros y unidades de CD-ROM.
- Conectores USB para los dispositivos externos.
- Puerto de teclado y ratón.
- Conectores de bus PCI para las tarjetas de sonido, de gráficos, de adquisición de datos, y demás dispositivos de entrada-salida.

Los siguientes componentes son opcionales:

- Procesador de sonido integrado.
- Conectores de dispositivos en serie y en paralelo.
- Adaptador de red integrado.
- Conector de bus AGP para una tarjeta de video de alta velocidad.

Los siguientes son algunos procesadores de soporte importante en un sistema IA-32 ordinario:

- La *Unidad de punto flotante* (FPU) se encarga de los cálculos de punto flotante y de números enteros extendidos.
- El *Generador de reloj* conocido simplemente como *reloj*.
- El *Controlador de interrupciones programables* (PIC).
- El *Temporizador/Contador de intervalos programables*.
- El *Puerto paralelo programable*.

Arquitectura de los buses PCI y PCI Express

El bus **PCI** (*Interconexión de componentes periféricos*) proporciona un puente de conexión entre la CPU y los otros dispositivos del sistema, como discos duros, memoria, controladores de video, tarjetas de sonido y controladores de red. El bus PCI Express, que es más reciente proporciona conexiones seriales de dos vías entre los dispositivos, la memoria y el procesador. Transporta los datos en forma de paquetes, de manera similar a las redes, en "vías" separadas. Es de amplio uso en los controladores gráficos, y puede transferí datos a una velocidad aproximada de 4GB por segundo.

Juego de chips de la tarjeta madre

La mayoría de las tarjetas madres contienen un conjunto integrado de microprocesadores y controladores, al cual se le conoce como *juego de chips* (chipset). El juego de chips determina en gran parte las capacidades de la computadora.

Salida de video

El adaptador de video controla la visualización de texto y gráficos en computadoras IBM-compatibles. Tiene dos componentes: el controlador de video y la memoria de visualización de video. Todos los gráficos y el texto que se muestran en el monitor se escriben en la RAM de visualización de video, para después enviarlo al monitor mediante el controlador de video. El controlador de video es en sí un microprocesador de propósito especial, que libera a la CPU principal del trabajo de controlar el hardware de video.

Memoria

En los sistemas basados en Intel se utilizan varios tipos de memoria: memoria de sólo lectura (ROM), memoria de sólo lectura programable y borrable (EPROM), memoria dinámica de acceso aleatorio (DRAM), RAM estática (SRAM),RAM de video (VRAM) y RAM de metal-óxido semiconductor complementario (CMOS).

Puertos de entrada/salida e interfaces de dispositivos

Bus serial universal (USB) El puerto del Bus serial universal proporciona una conexión inteligente de alta velocidad entre una computadora y los dispositivos con soporte USB. La versión 2.0 de USB soporta velocidades de transferencia de datos de480 Megabits por segundo. Puede conectar unidades de una sola función (ratones, impresoras) o dispositivos compuestos con más de un periférico, que comparten el mismo puerto.

Sistema de entrada/salida

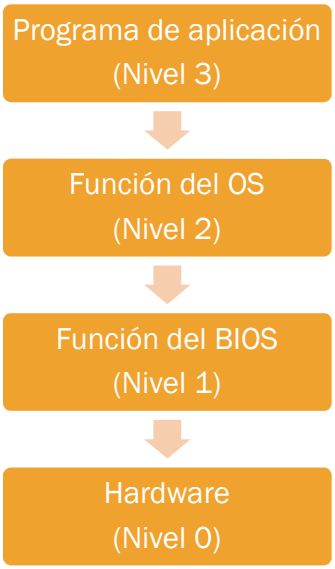
Cómo funciona todo

Los programas de aplicación leen, de manera rutinaria, la entrada que se recibe del teclado y de los archivos en disco, y escriben la salida en la pantalla y en archivos. Las operaciones de E/S no se realizan mediante el acceso directo al hardware, sino que podemos llamar a las funciones que proporciona el sistema operativo. Hay opciones de E/S disponible en distintos niveles de acceso, de manera similar al concepto de máquina

- Funciones de lenguaje de alto nivel (HLL): un lenguaje de programación de alto nivel, como C++ o java, contienen funciones para realizar operaciones de entrada-salida. Estas funciones son portables, ya que se trabajan en una variable de sistemas computacionales distintos, y no dependen de ningún sistema operativo.
- Sistema operativo: los programadores pueden hacer llamadas a las funciones de sistema operativo, desde una biblioteca conocida como API (Interfaz de programación de aplicaciones). El sistema operativo proporciona operaciones de alto nivel, como la escritura de cadenas en archivos, lectura de cadenas del teclado y la asignación de bloques de memoria.
- BIOS: el Sistema básico de entrada-salida es una colección de subrutinas de bajo nivel, que se comunican en forma directa con los dispositivos de hardware. El fabricante de la computadora instala el BIOS, el cual se adapta para ajustarse al hardware de la computadora. Por lo general, los sistemas operativos se comunican con el BIOS.

Controladores de dispositivos

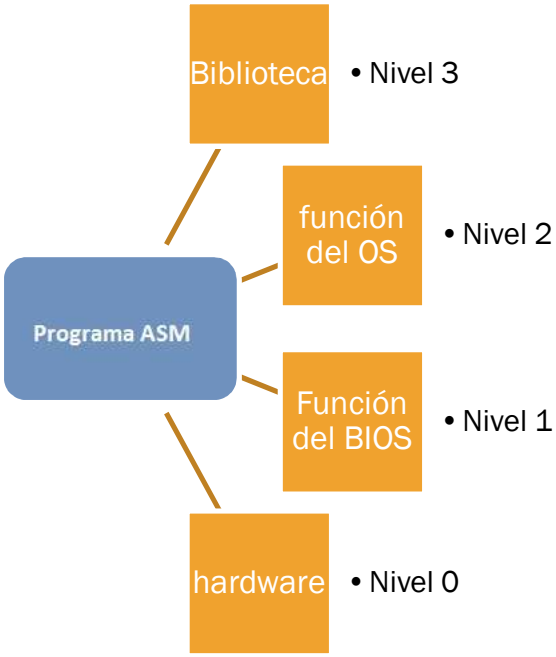
¿Qué ocurre si se instala un nuevo dispositivo en la computadora, que el BIOS desconozca? Cuando se inicia el sistema operativo, carga un programa controlador de dispositivos que contiene funciones para comunicarse con el dispositivo. Un controlador de dispositivos funciona de forma muy parecida al BIOS, pues proporciona funciones de entrada-salida adaptadas a un dispositivo específico, o familia de dispositivos. Un ejemplo de ello es CDROM.SYS, que permite a MS-DOS leer unidades de CD-ROM.



Jerarquía de E/S en perspectiva, mostrando lo que ocurre cuando un programa de aplicación muestra una cadena de caracteres en la pantalla. Se lleva a cabo los siguientes pasos:

1. La instrucción en el programa de aplicación llama a una función de la biblioteca HLL, La cual escribe la cadena en la salida estándar.
2. La función de biblioteca (Nivel 3) llama a una función del sistema operativo, y pasa a un apuntador de cadena.
3. La función del sistema operativo (Nivel 2) utiliza un ciclo para llamar a una subrutina del BIOS, pasarle el código ASCII y el color de carácter. El sistema operativo llama a otra subrutina del BIOS para desplazar el cursor a la siguiente posición en la pantalla.
4. La subrutina de BIOS (Nivel 1) recibe un carácter, lo asigna a una fuente específica del sistema y envía el carácter a un puerto de hardware, conectado a la tarjeta controladora de video.
5. La tarjeta controladora de video (Nivel 0) genera señales de hardware sincronizadas para la pantalla de video. Estas señales controlan el barrido de trama y la visualización de los pixeles.

Programación en varios niveles Los programas en lenguaje ensamblador tienen poder y flexibilidad en el área de la programación de las operaciones de entrada-salida. Pueden elegir uno de los siguientes niveles de acceso:



- Nivel 3: llamar a las funciones de biblioteca para realizar operaciones de E/S de texto genérico, y de E/S basada en archivos.
- Nivel 2: llamar funciones del sistema operativo para realizar operaciones de E/S de texto genérico y de E/S basada en archivos. Si el OS utiliza una interfaz gráfica de usuario, tiene funciones para visualizar los gráficos de una manera independiente de los dispositivo.
- Nivel 1: llamar a las funciones del BIOS para controlar características específicas de cada dispositivo, como el color, los gráficos, el sonido, la entrada del teclado y la E/S de disco bajo nivel.
- Nivel 0: enviar y recibir datos desde puntos de hardware, teniendo un control absoluto sobre cada dispositivo.

3 Fundamentos del lenguaje ensamblador:

Constantes enteras

Una constante entera (o literal) está compuesta de un signo opcional a la izquierda, uno o más dígitos y un carácter opcional de sufijo (llamado raíz)

[{+|-}] dígitos [raíz]

La raíz puede ser una de las siguientes (en mayúsculas o minúsculas)

h	Hexadecimal
q/o	Octal
d	Decimal
b	Binario
r	Real codificado
t	decimal (alternativo)
y	Binario (alternativo)

Si no se da una raíz, se asume que la constante entera es decimal.

Ejemplos:

26 Decimal 42o Octal

26d Decimal 1Ah Hexadecimal

11010011b Binario 0A3h Hexadecimal

42q Octal

Una constante hexadecimal que empieza con una letra debe tener un *cero a la izquierda*, para evitar que el ensamblador lo interprete como un identificador.

Expresiones enteras

Una expresión entera es una expresión matemática que involucra valores enteros y operadores aritméticos. La expresión se debe evaluar como un entero, el cual puede almacenarse en 32 bits (del 0 al FFFFFFFFh)

Operadores aritméticos según su procedencia (mayor a menor)

Operador	Nombre	Nivel de procedencia
()	Paréntesis	1
+,-	Unario positivo, unario negaivo	2
*,/	Multiplicación, División	3
MOD	Modulo	3
+,-	Suma, resta	4

La *procedencia* se refiere al orden implícito de las operaciones cuando una expresión contiene dos o más operadores. El orden de operaciones se muestra para las siguientes expresiones:

4 + 5 * 2 Multiplicación, suma

12 – 1 MOD 5 Módulo, resta

-5 + 2 Unario negativo, suma

(4 + 2) * 6 Suma, multiplicación

Constantes numéricas reales

Las constantes numéricas reales se representan como reales decimales o reales codificados (hexadecimales). Un *real decimal* contiene un signo opcional seguido de un entero, un punto decimal, un entero opcional que expresa una fracción y un exponente opcional:

[Signo] entero.[entero] [exponente]

A continuación se muestra la sintaxis para el signo y el exponente

Signo {+,-}

Exponente E[{+,-}]entero

A continuación mostraremos algunos ejemplos de constantes numéricas reales válidas:

2.

+3.0

-44.2E+05

26.E5

Se requieren, por lo menos, un dígito y un punto decimal.

Reales codificados Un real codificado representa a un número real en hexadecimal, utilizando el formato IEEE de punto flotante, Por ejemplo la representación binaria del número +1.0 decimal es:

0011 1111 1000 0000 0000 0000 0000 0000

El mismo valor se codificará como un real corto en lenguaje ensamblador de la siguiente manera:

3F800000r

Constante tipo carácter

Una constante tipo carácter es un solo carácter encerrado entre comillas sencillas o dobles. MASM almacena el valor en memoria como el código ASCII binario del carácter. Algunos ejemplos:

'A'

"d"

Constante tipo cadena

Una constante tipo cadena es una secuencia de caracteres (inclusive espacios) encerrados entre comillas sencillas o dobles:

'ABC'

'X'

"Buenas noches, Victor"

Se pueden agregar comillas a la cadena, siempre y cuando se utilicen de la siguiente manera:

"Ésta no es una prueba"

'Diga "Buenas noches", Victor'

Palabras reservadas

Las palabras reservadas tienen un significado especial en MASM, y sólo pueden usarse dentro de su contexto correcto. Hay distintos tipos de palabras reservadas:

- Nemónicos de instrucciones.
- Directivas.
- Atributos (BYTE y WORD).
- Operadores.
- Símbolos predefinidos.

Identificadores

Un identificador es un nombre elegido por el programador. Puede servir para identificar a una variable, una constante, un procedimiento o una etiqueta de código. Se debe Considerar lo siguiente a la hora de crear identificadores:

- Pueden contener entre 1 y 247 caracteres.
- No son sensibles a mayúsculas/minúsculas.
- El primer carácter debe ser una letra, guion bajo, @, ? o $. Los siguientes caracteres también pueden ser dígitos.
- Un identificador no puede ser igual que una palabra reservada para el lenguaje ensamblador.

*Para hacer que todas las palabras claves y los identificares sean sensibles a mayúscula/minúscula, se debe agregar el modificador de línea de comandos –Cp cuando se ejecute el ensamblador.

El ensamblador utiliza mucho el símbolo @ como prefijo para los símbolos predefinidos, por lo que es conveniente evitarlo en los identificadores que se creen. Se debe utilizar nombres descriptivos y fáciles de entender para sus identificaciones. He aquí algunos identificadores válidos:

Var1	Cuenta	$primero
_main	Max	archivo_abierto
@@miarchivo	ValorX	_12345

Directivas

Una directiva es un comando incrustado en el código de fuente, que el ensamblador reconoce y actúa en base a ésta. Las directivas no se llevan a cabo en tiempo de ejecución, mientras que las instrucciones sí. Las directivas pueden definir variables, macros y procedimientos. Pueden asignar nombres a los segmentos de memoria y realizar muchas otras tareas de mantenimiento relacionadas con el ensamblador en MASM, las directivas no son sensibles a mayúsculas/minúsculas. MASM reconoce a **.data**, **.DATA** y a **.Data** como equivalentes.

El siguiente ejemplo nos ayudará a mostrar las directivas que no se llevan a cabo en tiempo de ejecución, La directiva DWORD indica al ensamblador que debe reservar espacio en el programa para una variable de tipo doble palabra. La instrucción MOV se lleva a cabo en tiempo de ejecución, copiando el contenido de miVar al registro EAX:

```
miVar DWORD 26 ;directiva DWORD
mov eax, miVar      ;instrucción MOV
```

Cada ensamblador tiene un conjunto distinto de directivas. Por ejemplo TASM (Borland) y NAMS (Netwide Assembler) comparten un subconjunto común de directivas con MASM. Por otro lado, el ensamblador GNU casi no tiene directivas en común con MASM.

Definición de segmentos Una función importante de las directivas de ensamblador es definir las secciones, o segmentos, del programa. La directiva .DATA identifica el área de un programa que contiene variables:

.data

La directiva .CODE identifica el área de un programa que contiene instrucciones:

.code

La directiva .STACK identifica el área de un programa que guarda la pila en tiempo de ejecución, y establece su tamaño:

.stack 100h

Instrucciones

Una instrucción es un enunciado que se vuelve ejecutable cuando se ensambla un programa. El ensamblador traduce las instrucciones en byte de lenguaje máquina, para que la CPU los cargue y los lleve a cabo en tiempo de ejecución. Una instrucción contiene cuatro partes básicas:

- Etiqueta (opcional).
- Nemónico de instrucciones (requerido).
- Operando(s) (Por lo general, son requeridos).
- Comentario (opcional).

Ésta es una sintaxis básica:

[etiqueta:] nemónico operando(s) [;comentario]

Etiqueta

Una etiqueta es un identificador que actúa como marcador de posición para las instrucciones y los datos una etiqueta que se coloca justo antes de una instrucción, representa la dirección de esa instrucción De manera similar, una etiqueta se coloca justo antes de una variable para representar la dirección de esa variable.

Etiquetas de datos Una etiqueta de datos identifica la ubicación de una variable y proporciona una manera conveniente de hacer referencia a la variable dentro del código. El siguiente ejemplo define a una variable llamada cuenta:

```
cuenta DWORD 100
```

El ensamblador asigna una dirección numérica a cada etiqueta. Es posible definir varios elementos de datos después de una etiqueta. En el siguiente ejemplo, la etiqueta arreglo define la ubicación del primer número (1024). Los demás números que se le siguen en la memoria van inmediatamente después:

```
arreglo DWORD 1024, 2048
        DWORD 4096, 8192
```

Etiquetas de código Una etiqueta en el área código de un programa (en donde se encuentran las instrucciones) debe terminar con un carácter de dos puntos (:). En este contexto, las etiquetas se utilizan como destinas de las instrucciones de salto y de ciclos. Por ejemplo, la siguiente instrucción JMP transfiere el control a la ubicación marcada por la etiqueta llamada destino, con lo cual se creará un ciclo:

```
destino:
        mov ax,bx
        ...
        Jmp destino
```

Una etiqueta de código puede compartir la misma línea con una instrucción, o puede estar en una línea por si sola:

```
L1: mov ax,bx
L2:
```

Una etiqueta de datos no puede terminar con un signo de dos puntos. Los nombres de las etiquetas se crean utilizando las reglas para los identificadores. Los nombres de las etiquetas de datos deben ser únicos dentro del mismo archivo de código de fuente; las etiquetas de código sólo deben ser únicas dentro del mismo procedimiento.

Nemónicos de instrucción

Un nemónico de instrucción es una palabra corta que identifica a una instrucción, En Inglés, un nemónico es un dispositivo que ayuda a la memoria. De manera similar, los nemónicos de instrucciones en lenguaje ensamblador como mov, add y sub, proporcionan sugerencias acerca del tipo de operación a realiza

mov	Mueve (asigna) un valor a otro
add	Suma dos valores
Sub	Resta un valor de otro
mul	Multiplica dos valores
jmp	Salta a una nueva ubicación
call	Llama a un procedimiento

Operandos las instrucciones en lenguaje ensamblador pueden tener de cero a tres operandos cada uno de los cuales puede ser un registro, un operando de memoria, una expresión constante o un puerto de E/S

```
stc                 ;activa la bandera de Acarreo

inc eax             ;suma 1 a EAX

mov cuenta,ebx      ;mueve EBX a cuenta
```

En una instrucción con dos operandos, al primero se le llama *destino* y al segundo el *origen*. En general, la instrucción modifica el contenido del operando de destino. Por ejemplo, en una instrucción MOV los datos se copian del origen al destino.

Comentarios

Los comentarios son un medio importante para que el escritor de un programa comunique información acerca de su funcionamiento a la persona que lee el código de fuente. Con frecuencia se incluye la siguiente información en la parte superior del listado de un programa:

- La descripción de propósito del programa.
- Los nombres de las personas que crearon y revisaron el programa.
- Las fechas de creación y revisión del programa.
- Notas técnicas acerca de la puesta en marcha del programa.

Los comentarios pueden especificarse en dos formas:

- Comentarios de una sola línea, que empiezan con un carácter de punto y coma (;). El ensamblador ignora todos los caracteres que van después del punto y coma en la misma línea.
- Comentarios de bloque, que empiezan con directiva COMMENT y un símbolo especificado por el usuario por ejemplo,

```
COMMENT ¡
        Esta línea es un comentario.
        Esta línea también es un comentario.
¡
```

Podemos usar cualquier símbolo

```
COMMENT ¡
COMMENT $
COMMENT &
```

La instrucción NOP (ninguna operación)

La instrucción más segura que podemos escribir se llama NOP (ninguna operación). Ocupa 1 byte de almacenamiento de programa y no hace nada. Algunas veces los compiladores y los ensambladores lo utilizan para alinear el código con los límites de las direcciones pares

```
00000000 66 8B C3mov ax,bx
00000003 90      nop            ; alinea la siguiente instrucción
00000004 8B D1 mov edx,ecx
```

Los procesadores IA-32 están diseñados para cargar código y datos con más rapidez de direcciones pares de doble palabra.

Ejemplo: Suma y resta de enteros

Ahora vamos a presentar un programa corto en lenguaje ensamblador que suma y resta enteros. Los registros se utilizan para almacenar los datos intermedios, y se hace una llamada a una subrutina de biblioteca para mostrar el contenido de los registros en la pantalla. He aquí el código de fuente del programa:

```
TITLE suma y resta (SumaResta.asm)

;Este programa suma y resta enteros de 32 bits.

INCLUDE Irvine32.inc

.code

main proc

mov      eax, 10000h  ;EAX=10000h

add      eax, 40000h  ;EAX=50000h

sub      eax, 20000h  ;EAX=30000h

call     DumpRegs     ;muestra los registros

exit

main ENDP

END main
```

Vamos a analizar el programa, línea por línea:

```
TITLE suma y resta         (SumaResta.asm)
```

La directiva TITLE marca toda la línea como comentario. Puede ponerse lo que sea en esta línea.

```
; Este programa suma y resta enteros de 32 bits.
```

El ensamblador ignora todo el texto que esté a la derecha de un signo punto y coma, así que lo utilizamos para comentarios.

```
INCLUDE Irvine32.inc
```

La directiva INCLUDE copia definiciones necesarias y la información de configuración de un archivo de texto llamado Irvine32.inc, ubicado en el directorio INCLUDE del ensamblador

```
.code
```

La directiva **.code** marca el inicio del *segmento código*, en el que se ubican todas las instrucciones ejecutables de un programa.

```
main PROC
```

La directiva PROC identifica el comienzo de un procedimiento. El nombre que elegimos para el único procedimiento en nuestro programa es **main**.

```
mov eax, 10000h ;EAX=10000h
```

La instrucción MOV mueve (copia) el número entero 10000h al registro EAX. El primer operando (EAX) se llama *operando de destino* y el segundo *operando de origen*.

```
add eax, 40000h ;EAX=50000h
```

La instrucción ADD suma 40000h al registro EAX.

```
sub eax, 20000h ;EAX=30000h
```

La instrucción SU resta 20000h del registro EAX.

```
call DumpRegs   ; muestra los registros
```

La instrucción CALL llama a un procedimiento que muestra los valores actuales del registro de la CPU. Ésta puede ser una forma útil de verificar que un programa esté funcionando en forma apropiada.

```
        exit
main ENDP
```

La instrucción **exit** llamaa (indirectamente) a una función predefinida de MS-Windows que detiene la ejecución del programa. La directiva ENDP marca el final del procedimiento main. Observe que **exit** no es una palabra clave de MASM, sino un comando definido de *Irvine32.inc* que proporciona una manera sencilla de terminar un programa

```
END main
```

La directiva END marca la última línea del programa que se va a ensamblar. Identifica el nombre del procedimiento de arranque del programa (el procedimiento que inicia la ejecución del programa).

Resultado del programa

EAX=00030000 EBX=7FFDF000 ECX=00000101 EDX=FFFFFFFF
ESI=00000000 EDI=00000000 EBP=0012FFF0 ESP=0012FFC4
EIP=00401024 EFL=00000206 CF=0 SF=0 ZF=0 0F=0 AF=0 PF=1

Segmentos Los programas se organizan alrededor de segmentos llamados código, datos y pila. El *segmento de código* contiene todas las instrucciones ejecutables de un programa. Por lo general, el segmento código contiene uno o más procedimientos, en donde uno de ellos se designa como un procedimiento de *arranque*.

En el programa **SumaResta**, el procedimiento de arranque es **main**. Otro segmento, el segmento de pila almacena los parámetros de procedimiento y las variables locales. El segmento de datos almacena variables.

Estilo de codificación Como el lenguaje ensamblador es insensible al uso de mayúsculas/minúsculas no hay una regla de estilo fija en relación con la capitalización de código de fuente. Para su fácil lectura, debemos ser consistentes en el uso de minúsculas y mayúsculas, así como en los nombres que asignaremos a los identificadores. A continuación se muestran algunas metodologías relacionadas con el uso de mayúsculas y minúsculas que se aconseja adoptar:

- Utilizar minúsculas para las palabas clave, mayúsculas y minúsculas para los identificadores y solo mayúsculas para las constantes. Esta metodología sigue el modelo general de C,C++ y Java.
- Utilizar mayúsculas en todo. Esta metodología se utilizó en el software anterior a la década de 1970 cuando muchas terminales de computadoras no tenían soporte para las letras minúsculas. Tiene la ventaja de solucionar los efectos de las impresoras de mala calidad y los defectos en la vista pero es un poco anticuada.
- Usar mayúsculas para las palabras reservadas en el lenguaje ensamblador, incluyendo a los nemónicos de instrucciones y los nombres de los registros. Esta metodología nos ayuda a diferencia entre los identificadores y las palabras reservadas.
- Usar mayúsculas en las directivas y los operadores de lenguaje ensamblador, usar una mezcla de mayúsculas y minúsculas para los identificadores. Y minúsculas para todo lo demás (se empleará esta metodología a excepción de que utilizaremos minúsculas para las directivas).

Versión alternativa de SumaResta

El programa SumaResta utiliza el archivo Irvine32.lib, el cual oculta unos cuantos detalles. Con el tiempo comprenderemos todo lo que hay en ese archivo, pero apenas estamos empezando con el lenguaje ensamblador. Si se quiere ver toda la información completa desde el principio, he aquí una versión de SumaResta que no depende de archivos incluidos. Se utiliza la fuente en negrita para resaltar las partes del programa que son distintas a la versión anterior.

```
TITLE suma y resta (SumaResta.asm)
```

```
;Este programa suma y resta enteros de 32 bits.
.386
.model flat,stdcall
ExitProcess PROTO, dwExitCode:DWORD
DumpRegs PROTO
.code
main proc
mov      eax, 10000h  ;EAX=10000h
add      eax, 40000h  ;EAX=50000h
sub      eax, 20000h  ;EAX=30000h
call     DumpRegs     ;muestra los registros
INVOKE ExitProcess,0
main ENDP
END main
```

Vamos a hablar de las líneas que se modificaron. Como antes mostraremos cada línea de código con su explicación:

```
.386
```

La directiva .386 identifica el tipo de procesador mínimo requerido para este programa (Intel386)

```
.model flat,stdcall
```

La directiva .MODEL instruye el ensamblador para que genere código para un programa en modo protegido, y STDCALL habilita las llamadas a funciones de MS-Windows.

```
ExitProcess PROTO, dwExitCode:DWORD
DumpRegs PROTO
```

Dos directivas PROTO declaran los prototipos de los procedimientos utilizados por este programa: ExitProcess es una función de MS-Windows que detiene la ejecución del programa actual (denominado proceso) y DumpRegs es un procedimiento de la biblioteca de enlace Irvine32 que muestra el contenido de los registros.

```
INVOKE ExitProcess,0
```

El programa termina con una llamada a la función ExitProcess a la cual le asa un código de retorno cero. INVOKE es una directiva del ensamblador, que llama a un procedimiento o a una función.

Plantilla de programa

Los programas en lenguaje ensamblador tienen una estructura simple, con pequeñas variaciones, Al empezar un nuevo programa, es útil empezar con una plantilla de un programa vacío, con todos los elementos básicos en su lugar. Puede evitar escribir en forma redundante si rellena las partes faltantes y guarda el archivo bajo un nuevo nombre. El siguiente programa en modo protegido (plantilla.asm) puede personalizarse fácilmente. Observe que hemos insertado comentarios, marcado los putos en donde se debe agregar código propio:

```
TITLE Plantilla de programa (Plantilla.asm)

; Descripción del programa:

; Autor:

; Fecha de creación:

; Revisiones:

; Fecha de la última modificación

INCLUDE Invirne32.inc

; (aquí se insertan las definiciones de símbolos)

.data

          ; (aquí se insertan las variables)

.code

main PROC

          ;(aquí se insertan las instrucciones ejectuables)

     exit                                  ;sale al sistema operativo

main ENDP

          ;(aquí se insertan los procedimientos adicionales)

END main
```

Uso de comentarios Al principio del programa se han insertado varios campos de comentarios. Es conveniente incluir la descripción del programa, el nombre del autor del mismo, la fecha de creación e información acerca de las modificaciones subsiguientes.

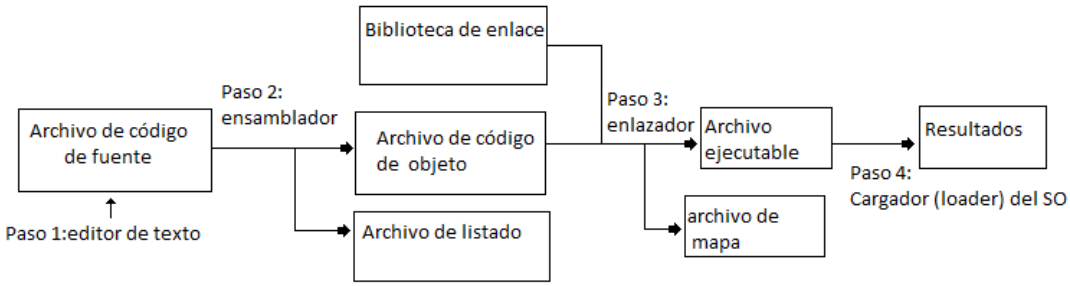
La documentación de este tipo es útil para cualquier persona que lea el listado del programa (incluyendo a uno mismo, meses o años a partir de ahora). Muchos programadores han descubierto que, años después de haber escrito un programa, deben volver a familiarizarse con su propio código para poder modificarlo.

Ensamblado, enlazado y ejecución de programas

En los capítulos anteriores, vimos ejemplos de programas simples en lenguaje máquina, por lo que está claro que un programa de código de fuente en lenguaje ensamblador no puede ejecutarse directamente en su computadora de destino. Debe *traducirse o ensamblarse* en código ejecutable. De hecho, un ensamblador es muy similar a *un compilador*, el tipo de programa que utilizamos para traducir un programa en C++ o Java a código ejecutable.

El ensamblador produce un archivo que contiene lenguaje máquina, al cual se le conoce como *archivo de código objeto*. Este archivo no está listo para ejecutarse. Debe pasarse por otro programa llamado *enlazador*, que a su vez produce un archivo ejecutable, Este archivo está listo para ejecutarse desde MS-DOS/Windows.

Ciclo de ensamblado-enlazado-ejecución



El ensamblador produce dos tipos de archivos:

- Archivo de código de objeto (.OBJ).
- Archivo de listado (.LIST) opcional.

El enlazador produce dos tipos de archivos:

- Archivo ejecutable (.EXE).
- Archivo de mapa (.MAP) opcional.

Archivos creados o actualizados por el enlazador

Archivo de mapa Un archivo de mapa contiene información (en texto simple) acerca de los segmentos de un programa, incluyendo lo siguiente:

- Nombre del módulo, utilizado como el nombre base del archivo EXE que produce el enlazador.
- Etiqueta de fecha y hora del encabezado del archivo del programa (no del sistema del archivo).
- Una lista de de símbolos públicos, que contiene la dirección, el nombre, la dirección plana y el módulo en donde se define cada símbolo.
- La dirección del punto de entrada del programa.

Archivo de base de datos del programa Cuando MASM ensambla un programa con la opción de depuración (-Zi), crea un archivo de base de datos del programa con la extensión de archivo pdb. Durante el paso de enlazado, el enlazador lee y actualiza el archivo pdb. Cuando ejecutamos el programa usando un depurador, éste muestra el código de fuente del programa, sus datos, la pila en tiempo de ejecución y demás información de utilidad.

Definición de datos

Tipos de datos intrínsecos

MASM define tipos de datos intrínsecos, cada uno de los cuales describe un conjunto de valores que pueden asignarse a las variables y expresiones del tipo dato, La característica esencial de cada tipo es su tamaño en bits: 8, 16, 32, 64 y 80. Las demás características (por ejemplo: con signo, apuntador o de punto flotante) son opcionales y su propósito principal es para el beneficio de los programadores que desean que se les recuerde acerca del tipo de datos que guarda la variable. Por ejemplo, es lógico que una variable declarada como DWORD guarde un entero de 32 bits sin signo. De hecho podría guardar un entero de 32 bits con signo, un número real de 32 bits con precisión simple o un apuntador de 32 bits. El ensamblador no es sensible al uso de mayúsculas/minúsculas, por lo que una directiva tal como DWORD podría escribirse como dword, Dword, dWord, etcétera.

Instrucciones de definición de datos

Una *instrucción* o *estatuto de datos* separa espacio de almacenamiento de memoria para una variable, con un nombre opcional. Las instrucciones (o estatutos) de definición de datos crean variables con base en los tipos de datos intrínsecos, una definición de datos tiene la siguiente sintaxis:

```
[nombre] directiva inicializador [,inicializador]….
```

Tipo	Uso
BYTE	Entero de 8 bits sin signo
SBYTE	Entero de 8 bits con signo
WORD	Entero de 16 bits sin signo [también puede ser un apuntador cercano (near en modo de direccionamiento real]
SWORD	Entero de 16 bits con signo
DWORD	Entero de 32 bits sin signo
SDWORD	Entero de 32 bits con signo [también puede ser un apuntador cercano (near) en modo protegido]
FWORD	Entero de 48 bits [apuntador lejano (far) en modo protegido]
QWORD	Entero de 64 bits
TBYTE	Entero de 80 bits (10 bytes)
REAL4	Número real corto IEEE de 32 bits (4 bytes)
REAL8	Número real ´largo IEEE de 64 bits (8 bytes)
REAL10	Número real extendido IEEE de 80 bits (10 bytes)

*Todos los tipos de datos pertenecen a enteros, excepto los últimos tres, en los que la notación IEEE hace referencia a los formatos de números reales estándar, publicados por la sociedad de computadoras del IEEE.

Nombre El nombre opcional que se asigna a una variable debe apegarse a las reglas para los identificadores

Directiva La directiva en una instrucción de definición de datos puede ser BYTE, WORD, DWORD, SBYTE, SWORD o cualquiera de los tipos listados en la tabla anterior. Además, puede ser cualquiera de las directivas de definición de datos heredadas que se muestran a continuación:

Directiva	Uso
DB	Entero de 8 bits
DW	Entero de 16 bits
DD	Entero real de 32 bits
DQ	Entero real de 64 bits
DT	Define 80 bits, o diez bytes

Inicializador En una definición de datos se requiere por lo menos un *inicializador*, aunque sea cero. Los inicializadores adicionales (si los hay) van separados por comas. Para los tipos de datos enteros, inicializadores es una constante o expresión entera, que coincide con el tamaño del tipo de la variable, como BYTE o WORD Si se quiere dejar la variable sin inicializar (que se le asigne un valor aleatorio), se usa el símbolo **?** como inicializador. Todos los inicializadores, sin importar su formato, se convierten en datos binarios mediante el ensamblador. Los inicializadores como 00110010b, 32h y 50d terminan con el mismo valor binario.

Definición de datos BYTE y SBYTE

La directiva BYTE (definir byte) y SBYTE (definir byte con signo) asignan espacio de almacenamiento para uno o más valores con o sin signo. Cada inicializador debe caber en 8 bits de almacenamiento.

```
valor1 BYTE 'A'                    ; constante tipo carácter

valor2 BYTE 0                      ; el byte sin signo más pequeño

valor3 BYTE 255                    ; el byte sin signo más grande

valor4 SBYTE  -128                 ; el byte con signo más pequeño

valor5 SBYTE +127                  ; el byte con signo más grande
```

Un inicializador con signo de interrogación (?) deja la variable sin inicializar, indicando que se le asignará un valor en tiempo de ejecución:

```
valor6 BYTE ?
```

El nombre opcional es una etiqueta que marca el desplazamiento de la variable, desde el inicio de su segmento. Por ejemplo, si **valor1** se encuentra en el desplazamiento 0000 dentro del segmento de datos y consume un byte de almacenamiento, **valor2** se encuentra de manera automática en el desplazamiento 0001:

```
valor1 BYTE 10h

valor2 BYTE 20h
```

La directiva heredada DB puede definir también una variable de 8 bits, con o sin signo:

```
Val1 DB 255                        ; byte sin signo

Val 1 DB -128                      ; byte con signo
```

Múltiples inicializadores

Si se utilizan varios inicializadores e la misma definición de datos, su etiqueta sólo hace referencia al desplazamiento del primer inicializador. El siguiente ejemplo, **lista** se encuentra en el desplazamiento 0000. Si es así, entonces el valor 10 se encuentra en el desplazamiento 0000, 20 en el desplazamiento 0001, 30 en el desplazamiento 0002, y 40 en el desplazamiento 3:

Desplazamiento	valor
0000:	10
0001:	20
0002:	30
0003:	40

No todas las definiciones de datos requieren etiquetas. Por ejemplo para continuar el arreglo de bytes que empezamos con **lista**, podemos definir bytes adicionales en las siguientes líneas:

```
lista BYTE 10,20,30,40
      BYTE 50,60,70,80
      BYTE 81,82,83,84
```

Dentro de una definición de datos individual, sus inicializadores pueden utilizar distintas raíces. Pueden mezclarse libremente las constantes tipo carácter y de cadena. En el siguiente ejemplo lista1 y lista2 tienen el mismo contenido:

```
lista1 BYTE 10, 32, 41h, 00100010b

lista2 BYTE 0Ah, 20h, 'A', 22h
```

Definición de cadenas

Para definir una cadena de caracteres, hay que encerrarlos entre comillas sencillas o dobles. El tipo más común de cadena termina con un byte nulo (que contiene 0). Las cadenas de este tipo, conocidas como *cadenas con terminación nula*, se utilizan en programas en C,C++ y Java:

```
saludo1 BYTE "Buenas tardes",0

saludo2 BYTE "Buenas noches",0
```

Cada carácter utiliza un byte de almacenamiento. Las cadenas son una excepción de la regla que establece que los valores de bytes deben separarse por comas. Sin esta excepción tendríamos que definir a **saludo1** de la siguiente manera:

```
saludo1 BYTE 'b','u','e','n','a','s'…..etc.
```

Lo cual sería muy tedioso. Una cadena puede distribuirse a través de varias líneas, sin tener que suministrar a una etiqueta para cada línea:

```
saludo1 BYTE "Bienvenido al programa de demostración de Cifrado"
        BYTE "creado por Kip Irvine.",0dh,0ah
        BYTE "Si desea modificar este programa, por favor"
        BYTE "envíame una copia.",0dh,0ah,0
```

Los códigos hexadecimales 0Dh y 0Ah se llaman también CR/LF (retorno de carro/avance de línea) o *caracteres de fin de línea*. Cuando se escriben en la salida estándar, desplazan el cursor hacia la columna izquierda de la línea que sigue a la línea actual.

El carácter de continuación de línea (\) concatena dos líneas de código de fuente en una sola instrucción. Debe ser el último carácter en la línea. Las siguientes instrucciones son equivalentes:

```
saludo1 BYTE "Bienvenido al programa de demostración de cifrado"
```

Y

```
saludo1 \
BYTE "Bienvenido al programa de demostración de cifrado"
```

El operador DUP

Este operador asigna almacenamiento para varios elementos de datos, usando una expresión constante como contador. En especial, es útil cuando se asigna un espacio para una cadena o arreglo, y puede utilizarse con datos inicializados o sin inicializar:

```
BYTE 20 DUP(0)                    ; 20 bytes, todos iguales a cero
BYTE 20 DUP(?)                    ; 20 bytes, sin inicializar
BYTE   5 DUP("PILA")              ; 20 bytes: "PILAPILAPILAPILAPILA"
```

Definición de datos WORD y SWORD

La directiva WORD (definir palabra) y SWORD (definir palabra con signo) crean almacenamiento para uno o más enteros de 16 bits:

```
palabra1 WORD 65535               ; el valor sin signo más grande
palabra2 SWORD -32768             ; el valor con signo más pequeño
palabra3 WORD  ?                  ; sin inicializar, sin signo
```

También puede usarse la directiva DW heredada:

```
val1 DW 65535                     ; sin signo
val2 DW -32768                    ; con signo
```

Arreglo de palabras para crear un arreglo de palabras se listan los elementos o se usa el operador DUP. El siguiente arreglo contiene una lista de valores:

miLista WORD 1,2,3,4,5

A continuación se muestra un diagrama de arreglo en memoria, suponiendo que **miLista** empieza en el desplazamiento 0000. Las direcciones se incrementan en 2, ya que cada valor ocupa 2 bytes:

Desplazamiento	valor
0000:	1
0002:	2
0004:	3
0006:	4
0008:	5

El operador DUP proporciona una manera conveniente de inicializar varias palabras:

```
arreglo  WORD 5 DUP(?)                    ; 5 valores, sin inicializar
```

Definición de datos DWORD y SDWORD

Las directivas DWORD (definir doble palabra) y SDWORD (definir doble palabra con signo) asignan almacenamiento para uno o más enteros de 32 bits:

```
val1 DWORD 12345678h                      ; sin signo

val2 SDWORD -2147483648                   ; con signo

val3 DWORD 20 DUP(?)                      ; arreglo sin signo
```

Arreglo de dobles palabras Para crear un arreglo de dobles palabras, se inicializa cada elemento en forma explícita, o se utiliza el operador DUP. He aquí un arreglo que contiene valores sin signo específicos:

```
miLista DWORD 1,2,3,4,5
```

A continuación se muestra un diagrama del arreglo en memoria, suponiendo que **miLista** empieza en el desplazamiento 0000. Los desplazamientos se incrementan por 4:

Desplazamiento	valor
0000:	1
0004:	2
0008:	3
000C:	4
0010:	5

Definición de datos QWORD

La directiva QWORD (define palabra cuádruple) asigna almacenamiento para valores de 64 bits (8 bytes):

```
quad1 QWORD 1234567812345678h
```

También puede usarse la directiva DQ heredada:

```
quad1 DQ 1234567812345678h
```

Definición de datos TBYTE

La directiva TBYTE (define 10 bytes) crea almacenamiento para los enteros de 80 bits. Este tipo de datos se utiliza principalmente para almacenar números decimales codificados en binario. Para manipular estos valores se requieren instrucciones especiales en el conjunto de instrucciones de punto flotante:

```
val1 TBYTE 10000000000123456789Ah
```

También puede utilizarse la directiva DT heredada:

```
val DT 10000000000123456789Ah
```

Definición de datos de números reales

REAL4 define a una variable real de 4 bytes y precisión simple. REAL8 define a una real de 8 bytes y precisión doble, y REAL10 define a una real de 10 bytes y doble precisión extendida. Cada uno requiere de uno o más inicializadores constantes reales:

```
rVal1 REAL4 -1.2

rVal2 REAL8 3.2E-260

rVal3 REAL10 4.6E+4096

rArreglo REAL4 20 DUP (0.0)
```

La siguiente tabla describe a cada uno de los tipos reales estándar. En términos de su número mínimo de dígitos significativos y el rango aproximado:

Tipo de datos	Dígitos significativos	Rango aproximado
Real corto	6	1.18x10^-38 a 3.40x10^38
Real largo	15	2.23x10^-308 a 1.79x10^308
Real de precisión extendida	19	3.37x10^-4932 a 1.18x10^4932

La directiva DD, DQ y DT heredadas pueden definir números reales:

```
rVal1 DD -1.2                          ; real corto
```

```
rVal2 DQ 3.2E-260                    ; real largo

rVal3 DT 4.6E+4096                   ; real de precisión extendida
```

Orden Little Endian

Los procesadores Intel almacenan y recuperan datos de la memoria usando el orden *little endian*. El byte menos significativo se almacena en la primera dirección de memoria asignada para los datos. El resto de los bytes se almacenan en las siguientes posiciones consecutivas de memoria. Como ejemplo, considere la doble palabra 123455678h Si se coloca en el desplazamiento 0000 de la memoria, se almacenaría 78h en el primer byte, 56 en el segundo byte. Y el resto de los bytes estarían en el desplazamiento 0003 y 0004:

0000:	78	
0001:	56	Little endian
0002:	34	
0003:	12	

Hay otros sistemas computaciones que utilizan el orden *big endian* (de mayor a menor), La siguiente figura muestra un ejemplo del número 12345678h almacenado en orden big endian, en el desplazamiento 0:

0000:	12	
0001:	34	Big endian
0002:	56	
0003:	78	

Agregar variables al programa SumaResta

Vamos a utilizar el programa **SumaResta** para agregarle un segmento de datos que contenga varias variables de tipo doble palabra. A este programa modificado le llamaremos **SumaResta2**:

```
TITLE suma y resta, Versión 2      (SumaResta2.asm)
; Este programa suma y resta enteros de 32 bits sin signo
; y almacena la suma en una variable.
INCLUDE Irvine32.inc
.data
val1 dword 10000h
val2 dword 40000h
val3 dword 20000h
valFinal dword ?
.code
main PROC
          mov eax,val1                  ; empieza con 10000h
          add eax,val2                  ; suma 40000h
          sub eax,val3                  ; resta 20000h
          mov valFinal,eax              ; almacena el resultado (30000h)
         call DumpRegs                  ; muestra los registros
         exit
main ENDP
END main
```

¿Cómo funciona? Primero el entero en **val1** se mueve a EAX:

```
mov eax,val1                    ; empieza con 10000h
```

Después, **val2** se suma a EAX:

```
add eax,val2                    ; suma 40000h
```

Luego, se resta **val3** de EAX:

```
sub eax,val3                    ; resta 20000h
```

EAX se copia a **valFinal**:

```
mov valFinal,eax                ; almacena el resultado (30000h)
```

Declaración de datos sin inicializar

La directiva .DATA? declara los datos sin inicializar, Al definir un bloque extenso de datos sin inicializar, la directiva .DATA? reduce el tamaño de un programa compilado. Por ejemplo, el siguiente código se declara de manera eficiente:

```
.data

arregloPequenio DWORD 10 DUP(0)             ; 40 bytes

.data?

arregloGrande DWORD 5000 DUP(?)             ; 20,000 bytes sin inicializar
```

Por otro lado el siguiente código produce un programa compilado que es 20,000 bytes más grande:

```
.data

arregloPequenio DWORD 10 DUP(0)             ;40 bytes

arregloGrande DWORD 5000 DUP(?)             ;20,000 bytes
```

Mezcla de código y datos El ensamblador nos permite cambiar entre código y los datos en nuestros programas. Por ejemplo, tal vez podríamos llegar a necesitar declarar una variable que se utilice sólo dentro de un área localizada de un programa. El siguiente ejemplo inserta una variable llamada **temp** entre dos instrucciones de código:

```
.code

mov eax,ebx

.data

temp DWORD ?

.code

mov temp, eax

…
```

Aunque **temp** parece interrumpir el flujo de instrucciones ejecutables, MASM coloca a **temp** en el segmento de datos, separada del segmento que guarda el código compilado.

Constantes simbólicas

Una *constante simbólica* (o *definición de símbolo*) se crea mediante la asociación de un identificador (un símbolo) con una expresión entera, o con cierto texto. Los símbolos no

reservan almacenamiento. Sólo los utiliza el ensamblador al momento de explorar un programa, y no pueden cambiar en tiempo de ejecución. La siguiente tabla muestra un resumen sobre sus diferencias:

	Símbolo	Variable
¿Usa almacenamiento?	NO	SI
¿Cambia su valor en tiempo de ejecución	NO	SI

Mostraremos cómo utilizar la directiva de signo de igual (=) para crear símbolos que representen expresiones enteras. Utilizaremos las directivas EQU y TEXTEQU para crear símbolos que representen texto arbitrario.

Directiva de signo de igual

La directiva de signo igual asocia el nombre de un símbolo con una expresión entera. La sintaxis es:

nombre = expresión

Por lo común, expresión es un valor entero de 32bits. Cuando se ensambla un programa, todas las coincidencias de *nombre* se sustituyen por *expresión* durante el paso del preprocesador del ensamblador. Por ejemplo, si el ensamblador lee las líneas:

```
CUENTA = 500

mov ax,CUENTA
```

genera y ensambla la siguiente instrucción:

```
mov ax,500
```

¿Para qué utilizar símbolos? Podríamos haber omitido el símbolo CUENTA por completo, y solo codificar la instrucción MOV con el valor literal 500, pero la experiencia nos ha demostrado que es más fácil leer y mantener programas si se utilizan símbolos. Supongamos que CUENTA se utiliza 10 veces en todo un programa. Podría incrementarse después a 600, alterando sólo una línea de código:

CUENTA = 600

Cuando se vuelve a ensamblar el programa que utiliza CUENTA, todas las instancias de CUENTA se sustituyen de manera automática por 600. Sin este símbolo, el programador tendría que buscar y sustituir manualmente todos los números 500 con 600 en el código de fuente del programa. ¿Qué pasaría si una ocurrencia del número 500 no estuviera relacionada con todas las demás? Entonces se producirá un error si se cambiara por 600.

Definiciones del teclado A menudo, los programas definen símbolos para caracteres importantes del teclado. Por ejemplo, 27 es el código ASCII para la tecla Esc:

Tecla_Esc = 27

Después en el mismo programa, una instrucción se describe más a sí misma si utiliza el símbolo en vez de un valor inmediato:

```
mov al,Tecla_Esc                    ; buen estilo
```

en vez de

```
mov al,27                           ; mal estilo
```

Uso del operador DUP el contador utilizado por DUP debe ser una constante simbólica, para simplificar el mantenimiento del programa. En el siguiente ejemplo, si se definió CUENTA, puede utilizarse en la siguiente definición de datos:

```
Arreglo DWORD CUENTA DUP(0)
```

Redefiniciones Un símbolo definido con = puede definirse de nuevo dentro del mismo programa. El siguiente ejemplo muestra cómo el ensamblador evalúa a CUENTA, cuando cambia de valor:

```
CUENTA = 5

mov al,CUENTA              ; AL = 5

CUENTA = 10

mov  al, CUENTA            ; AL = 10

CUENTA = 100

mov al, CUENTA             ; AL = 100
```

El valor cambiante de un símbolo tal como CUENTA no tiene nada que ver con el orden de ejecución de las instrucciones en tiempo de ejecución, sino que el símbolo cambia su valor de acuerdo con el procesamiento secuencial del código de fuente que hace el ensamblador.

Cálculo de los tamaños de los arreglos y cadenas

Al utilizar un arreglo por lo general, es conveniente conocer su tamaño. El siguiente ejemplo utiliza una constante llamada **TamLista** para declarar el tamaño de **lista**:

lista BYTE 10,20,30,40

TamLista = 4

No es conveniente calcular en forma manual los tamaños de los arreglos cuando éstos pueden cambiar de tamaño más adelante en el programa. Si tuviéramos que agregar más bytes a **lista**, habría que corregir el valor de **TamLista**. Una mejor manera de manejar esta situación sería dejar que el ensamblador calculara **TamLista** en forma automática. El operador $ (*contador de ubicación actual*) devuelve el desplazamiento asociado con la instrucción actual del programa. En el siguiente ejemplo, **TamLista** se calcula restando el desplazamiento de **lista**, a partir del contador de ubicación actual ($):

```
lista BYTE 10,20,30,40

TamLista = ($ - lista)
```

TamLista debe ir justo después de **lista**. El siguiente ejemplo produce un valor demasiado grande para **TamLista**, debido a que el almacenamiento utilizado por **var2** afecta a la distancia entre el contador de la ubicación actual y el desplazamiento de **lista**:

```
lista BYTE 10,20,30,40

var2 BYTE 20 DUP (?)

TamLista = ($ - lista)
```

En vez de calcular la longitud de una cadena en forma manual, dejemos que el ensamblador lo haga:

```
miCadena BYTE "Esta es una cadena larga, que contiene"

         BYTE "Cualquier número de caracteres"

miCadena_longitud = ($ - miCadena)
```

Arreglo de palabras y doble palabras Al calcular el número de elementos en un arreglo que contiene palabras de 16 bits (WORD), divida la diferencia en los desplazamientos entre 2:

```
lista WORD 1000h,2000h,3000h,4000h

TamLista= ($ - lista) / 2
```

De manera similar, cada elemento de un arreglo de dobles palabras es de 4 bytes, por lo que su longitud total debe dividirse entre cuatro para producir el número de elementos de arreglo:

```
lista DWORD 10000000h,20000000h,30000000h,40000000h

TamLista = ($ - lista) / 4
```

Directiva EQU

La directiva EQU asocia un nombre simbólico con una expresión entera o con algún texto arbitrario. Existen tres formatos:

nombre EQU *expresión*

nombre EQU *símbolo*

nombre EQU <*texto*>

En el primer formato, *expresión* debe ser una expresión entera válida. En el segundo formato, *símbolo* es el nombre de un símbolo existente, que ya se ha definido con = o EQU. En el tercer formato, puede aparecer cualquier texto dentro de los signos < y >. Cuando el ensamblador encuentra a *nombre* más adelante en el programa, sustituye el valor entero o texto por ese símbolo.

EQU puede ser útil cuando se define un valor que no se evalúa como entero, Por ejemplo, una constante numérica real puede definirse mediante EQU:

```
PI EQU <3.1416>
```

Ejemplo El siguiente ejemplo asocia un símbolo con una cadena de caracteres. Después puede crearse una variable mediante el uso de símbolo:

```
oprimaTecla EQU <”Oprima cualquier tecla para continuar…”,0>

.

.

.data

Indicador BYTE oprimaTecla
```

Ejemplo supongamos que deseamos definir un símbolo que cuente el número de celdas en una matriz entera de 10 por 10. Definiremos los símbolos de dos formas distintas, Primero como una expresión entera y después como una expresión de texto. Después utilizaremos los dos símbolos en definiciones de datos:

```
matriz1 EQU 10 * 10

matriz2 EQU <10 * 10>

.data

M1 WORD matriz1

M2 WORD matriz2
```

El ensamblador produce distintas definiciones de datos para **M1** y **M2**. La expresión entera en **matriz1** se evalúa y se asigna a **M1**. Por otro lado, el texto en **matriz2** se copia directamente en la definición de datos para **M2**:

```
M1 WORD 100

M2 WORD 10 *  10
```

Sin redefinición A diferencia de la directiva =, un símbolo definido con EQU no puede redefinirse en el mismo archivo de código fuente. Esta restricción evita que un símbolo se le asigne sin querer un nuevo valor.

Directiva TEXTEQU

La directiva TEXTEQU, similar a EQU, crea lo que se conoce como *macro de texto*. Hay tres formatos distintos: el primero asigna texto, el segundo asigna el contenido de una macro de texto existente, y el tercero asigna una expresión entera constante:

```
nombre TEXTEQU <texto>

nombre TEXTEQU macrotexto

nombre TEXTEQU %exprConst
```

Por ejemplo, la variable indicador1 utiliza el macro de texto **msjContinuar**:

msjContinuar TEXTEQU <"Desea continuar (S/N)?">

.data

Indicador1 BYTE msjContinuar

Las macros de texto se pueden basar en otras constantes de texto. En el siguiente ejemplo, **cuenta** se establece al valor de una expresión entera en la que se utiliza **tamFila**. Después, el símbolo **mover** se define como **mov**. Por último, **establecerAL**, se crea a partir de **mover** y **cuenta**:

```
tamFila = 5

cuenta TEXTEQU %(tamFila * 2)

mover TEXTEQU <mov>

establecerAL TEXTEQU <mover al, cuenta>
```

Por lo tanto, la instrucción

establecerAL

se ensamblaría como

mov al,10

Un símbolo definido por TEXTEQU puede redefinirse en cualquier momento.

Programación en modo direccionamiento real (opcional)

Los programas diseñados para MS-DOS deben ser aplicaciones de 16 bits que se ejecutan en modo de direccionamiento real. Las aplicaciones en modo direccionamiento real utilizan segmentos de 16 bits y siguen el esquema de direccionamiento segmentado. Si se utiliza un procesador IA-32, aún se puede utilizar los registros de 32 bits de propósito general para los datos.

Cambios básicos

Hay unos cuantos cambios que debemos hacer a los programas de 32 bits que presentamos en este capítulo, para transformarlos en programas en modo direccionamiento real:

- La directiva INCLUDE hace referencia a una biblioteca distinta:
  INCLUDE Irvine16.inc
- Se insertan dos instrucciones adicionales al principio del procedimiento de arranque (main). Estas instrucciones inicializan el registro DS con la ubicación inicial del segmento de datos, identificada por la constante predefinida @data de MASM:
  mov ax, @data
  mov ds,ax
- Los desplazamientos (direcciones) de la etiqueta de datos y de código son de 16 bits.

No se puede mover @data directamente a DS y ES, ya que la instrucción MOV no permite mover una constante directamente a un registro de segmento.

El programa SumaResta2

He aquí un listado del programa SumaResta2.asm, modificado para ejecutarse en modo de direccionamiento real. Las nuevas líneas están marcadas por comentarios:

```
TITLE Suma y resta, Versión 2   (SumaResta2.asm)

; Este programa suma y resta enteros de 32 bits

; y almacena la suma en una variable.

; Modo de direccionamiento real.

INCLUDE Irvine16.inc              ; cambiado *
```

```
.data
val1  dword 10000h
val2 dword 40000h
val3 dword 20000h
valFinal dword ?
.code
main PROC
           mov ax,@data           ; nuevo *
           mov ds,ax              ; nuevo *
           mov eax,val1           ; empieza con 10000h
           add eax,val2           ; suma 40000h
           sub eax,val3           ; resta 20000h
           mov valFinal,eax       ; almacena el resultado (30000h)
          call DumpRegs           ; muestra los registros
         exit
main ENDP
END main
```

4 Transferencia de datos, direccionamiento y aritmética

Tipos de operandos

En este capítulo presentaremos tres tipos de operandos de instrucciones: *inmediatos*, de *registro* y de *memoria*. De los tres el tercero es un poco complicado.

Notación de operandos de instrucciones.

Operando	Descripción
r8	Registro de propósito general de 8 bits: AH, AL, BH, BL, CH, CL, DH, DL
r16	Registro de propósito general de 16 bits: AX, BX, CX, DX, SI, DI, SP, BP
r32	Registro de propósito general de 32 bits: EAX, ECX, EDX, ESI, EDI, ESP, EBP
reg	Cualquier tipo de registro de propósito general
sreg	Registros de segmento de 16 bits: CS, DS, SS, ES, FS, GS
imm	Valor inmediato de 8, 16 o 32 bits
imm8	Valor tipo byte inmediato de 8 bits
imm16	Valor tipo palabra inmediato de 16 bits
imm32	Valor tipo doble palabra inmediato de 32 bits
r/m8	Operando de 8 bits, que puede ser un registro general de 8 bits o un byte de memoria
r/m16	Operando de 16 bits, que puede ser un registro general de 16 bits o una palabra de memoria
r/m32	Operando de 32 bits, que puede ser un registro general de 32 bits o una doble palabra
mem	Cualquier operenado de memoria de 8,16 o 32 bits

Operandos directos de memoria

En el capítulo anterior (capítulo 3) explicamos que los nombres de las variables son referencias a desplazamientos dentro del segmento de datos. Por ejemplo la siguiente declaración indica que se ha asignado al segmento de datos un byte que contiene el número 10h:

```
.data

var1 BYTE 10h
```

El código del programa contiene instrucciones que emplean (buscan) operandos que hacen referencia a direcciones de memoria. Supongamos que **var1** se ubicó en desplazamiento 10400h. Una instrucción en lenguaje ensamblador para mover esta variable al registro AL podría ser:

```
mov  AL,var1
```

MASM ensamblaría esta instrucción como el siguiente código de máquina:

A0 00010400

El primer byte en la instrucción de máquina es el código de operación. La parte restante es la dirección hexadecimal de 32 bits de **var1**. Aunque se podría escribir programas con direcciones numéricas, los nombres simbólicos como **var1** facilitan el proceso de referenciar a la memoria.

Notación alternativa. Algunos programadores prefieren usar la siguiente notación con operandos directos, ya que los corchetes implican una operación que emplean operandos que hacen referencia a una dirección de memoria:

```
mov al,[var1]
```

MASM permite esta notación, por lo que usted puede utilizarla en sus propios programas, si así lo desea. Debido a que muchos programas (incluyendo los de Microsoft) se imprimen sin corchetes, sólo lo utilizaremos en este libro cuando esté involucrada una expresión aritmética:

```
mov al, [var1 + 5]
```

Instrucción MOV

La instrucción MOV copia datos de un operando de origen a un operando de destino. Esta instrucción, conocida como *transferencia de datos*, se utiliza en casi todos los programas. Su formato básico muestra que el primer operando es el destino y el segundo es el origen:

```
MOV destino,origen
```

El contenido del operando de destino cambia, pero el del operando de origen no. El movimiento de datos de derecha a izquierda es similar a la instrucción de asignación en C++ o Java:

```
destino = origen ;
```

(En casi todas las instrucciones en lenguaje ensamblador, el operando izquierdo es el destino y el operando derecho es el origen)

MOV es bastante flexible en el uso de sus operandos, siempre y cuando se observen las siguientes reglas:

- Ambos operandos deben ser del mismo tamaño.
- Ambos operandos no pueden ser operandos de memoria.
- CS, EIP e IP no pueden ser operandos de destino.
- Un valor inmediato no puede moverse en el registro de segmento.

He aquí una lista de variantes generales de MOV, excluyendo los registros de segmento:

```
MOV reg,reg

MOV mem,reg

MOV reg,mem

MOV mem,imm

MOV reg,imm
```

Los programas que se ejecutan en modo protegido no deben modificar directamente los registros de segmento. Las siguientes operaciones están disponibles en modo real, con excepción de CS no puede ser un operando de destino:

```
MOV r/m16,sreg
```

```
MOV sreg,r/m16
```

Memoria a memoria Una sola instrucción MOV no puede usarse para mover datos directamente de una ubicación de memoria a otra. En vez de ello, puede mover el valor del operando de origen a un registro, antes de mover a un operando de memoria:

```
.data
```

```
var1 WORD ?
```

```
var2 WORD ?
```

```
.code
```

```
mov ax,var1
```

```
mov var2,ax
```

Hay que considerar el número mínimo de bytes requeridos por una constante entera, al copiarlo a una variable o registro (capítulo 1).

Extensión con cero y con signo de enteros

Copia valores más pequeños a valores más grandes

Aunque MOV no puede copiar datos directamente de un operando más pequeño a uno más grande, los programadores pueden crear soluciones alternativas. Supongamos que **cuenta** (con signo, 16 bits) debe moverse a ECX (32 bits.). Podemos establecer ECX con cero y mover **cuenta** a CX:

```
.data
```

```
cuenta WORD 1
```

```
. code
```

```
mov ecx,0
```

```
mov cx, cuenta
```

¿Qué ocurre si tratamos el mismo método con un entero con signo igual a -16?

```
.data

valorConSigno SWORD -16      ; FFF0h (-16)

.code

mov ecx,0

mov cx,valConSigno           ; ECX= 0000FFF0 (+65520)
```

El valor ECX (+65520) es completamente distinto de -16. Por otro lado, si llenáramos primero a ECX con FFFFFFFFh y después copiáramos **valorConSigno** a CX, el valor final hubiera sido correcto:

```
mov ecx,0FFFFFFFFh

mov cx,valConSigno            ; ECX = FFFFFFF0h (-16)
```

Esto representa un problema al tratarse de enteros con signo: no queremos tener que comprobar sus valores para ver si son positivos o negativos antes de decidir cómo llenar los operandos de destino. Por fortuna los ingenieros de Intel detectaron este problema al diseñar el procesador Intel386, e introdujeron las instrucciones MOVZX y MOVSX para mejorar enteros sin signo y enteros con signo.

Instrucción MOVZX

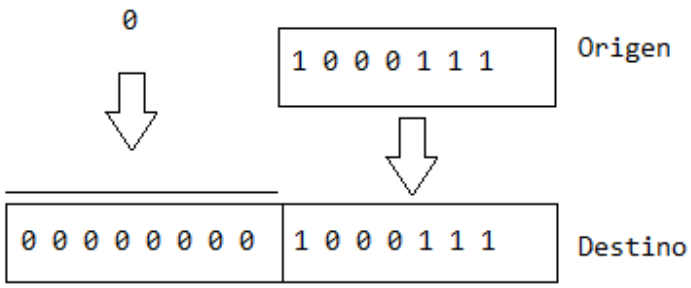
La instrucción MOVZX (*mover con extensión de ceros*) copia el contenido de un operando de origen a un operando de destino, y extiende con ceros el valor hasta 16 o 32 bits. Esta instrucción se utiliza solo con enteros sin signo. Hay tres variantes:

```
MOVZX r32,r/m8

MOVZX r32,r/m16

MOVZX r16,r/m18
```

La siguiente instrucción mueve el número binario 10001111 a AX:

```
movzx ax,10001111b
```

Diagrama de MOVZX ax, 8Fh:



Los siguientes ejemplos utilizan registros para todos los operandos, mostrando todas las variaciones de tamaño:

```
mov     bx,0A69Bh

movzx   eax,bx              ; EAX = 0000A69Bh

movzx   edx,bl              ; EDX = 0000009Bh

movzx   cx,bl               ; CX = 009Bh
```

Los siguientes ejemplos utilizan operandos de memoria para el origen y producen el mismo resultado:

```
.data

byte1 BYTE 9Bh

palab1 WORD 0A69Bh

.code

movzx eax,palab1            ; EAX  = 0000A69Bh

movzx edx,byte1             ; EDX = 0000009Bh

movzx cx,byte1              ; CX = 009Bh
```

Si se quiere ejecutar y probar los ejemplos de este capítulo en modo direccionamiento real, utilice INCLUDE con Irvine16.lib e inserte las siguientes líneas al principio del procedimiento principal (main):

mov ax,@data

mov ds,ax

Instrucción MOVSX

La instrucción MOVSX (mover extensión de signo) copia el contenido de un operando de origen en un operando de destino, y extiende con signo el valor hasta 16 o 32 bits. Esta instrucción sólo se utiliza con enteros con signo. Existen tres variables

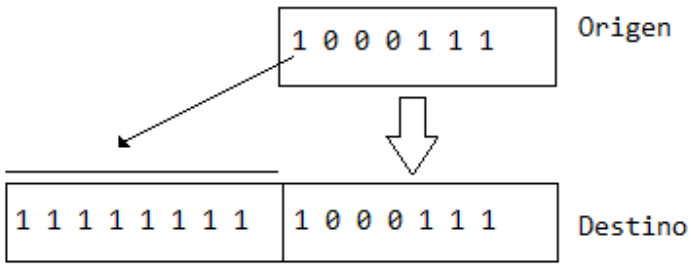
```
MOVSX r32,r/m8

MOVSX r32,r/m16

MOVSX r16,r/m8
```

Para extender con signo un operando, se toma el bit más alto del operando más pequeño y re repite (replica) este bit a lo largo de los bits extendidos en el operando de destino.
Supongamos que el valor 8 bits 10001111b

Se mueve a un destino de 16 bits:

```
movsx ax,10001111b
```

Diagrama de MOVSX ax,10001111b:



He aquí unos cuantos ejemplos más que utilizan una variedad de tamaños de registro:

```
mov    bx,0A69Bh

movzx  eax,bx          ; EAX = FFFFA69Bh

movzx  edx,bl          ; EDX = FFFFFF9Bh
```

```
movzx  cl,bl           ; CX = FF9Bh
```

Instrucciones LAHF y SAHF

La instrucción LAHF (cargar banderas de estados en AH) copia el byte inferior del registro EFLAGS a AH. Se copian las siguientes banderas: Signo, Cero, Acarreo Auxiliar, Paridad y Acarreo. Mediante el uso de esta instrucción, podemos guardar fácilmente una copia de las banderas en una variable, por seguridad:

```
.data

guardarbanderas  BYTE ?

.code

lahf                        ; carga las banderas en AH

mov guardarbanderas, ah     ; las guarda en una variable
```

La Instrucción SAHF (almacenar AH en banderas de estado) copia AH el byte inferior del registro EFLAGS. Por ejemplo, puede obtener los valores de las banderas que se hayan guardado antes en una variable:

```
mov ah, guardarbanderas     ; carga las banderas guardadas a AH

sahf                       ; las copia a los registros flags
```

Instrucción XCHG

La instrucción XCHG (intercambiar datos) intercambia el contenido de dos operandos. Hay tres variantes:

```
XCHG reg,reg

XCHG reg,mem

XCHG mem,reg
```

Las reglas para los operandos en la instrucción XCHG son las mismas que para la instrucción MOV, excepto que XCHG no acepta operandos inmediatos. En las aplicaciones para ordenar arreglos XCHG proporciona una manera simple de intercambiar los elementos de dos arreglos. He aquí unos cuantos ejemplos del uso de XCHG:

```
xchg  ax,bx   ; intercambia registros de 16 bits

xchg ah,al    ; intercambia registros de 8 bits

xchg var1,bx  ; intercambia op mem de 16 bits con BX

xchg eax,ebx  ; intercambia registros de 32 bits
```

Para intercambiar dos operandos de memoria, se debe utilizar un registro como contenedor temporal y combinar MOV con XCHG:

```
mov ax,val1

xchg ax,val2

mov val1,ax
```

Operando de desplazamiento directo

Para crear un operando de desplazamiento directo podemos sumar un desplazamiento al nombre una variable. Esto nos permite acceder a ubicaciones de memoria que tal vez no tengan etiquetas explícitas. Empecemos con un arreglo de bytes llamado **arregloB**:

```
arregloB BYTE 10h,20h,30h,40h,50h
```

Si utilizamos MOV con **arregloB** como operando de origen, moveremos de manera automática el primer byte en el arreglo:

```
mov al,arregloB        ; AL = 10h
```

Podemos acceder al segundo byte en el arreglo, sumando 1 al desplazamiento de **arregloB**:

```
mov al,[arregloB+1]    ; AL = 20h
```

Para acceder al tercer byte, le sumamos 2:

```
mov al,[arregloB+2]    ; AL = 30h
```

Una expresión como **arregloB+1** produce lo que se conoce como *dirección efectiva*, al sumar una constante al desplazamiento de la variable. Al encerrar una dirección efectiva entre corchetes, indicamos que la expresión se utiliza para hacer referencia a una dirección de memoria para obtener su contenido. MASM no requiere los corchetes, por lo las siguientes instrucciones son equivalentes:

```
mov al,[arregloB+2]

mov al,arregloB+2
```

Comprobación de rango MASM no tiene comprobación de rango integrada para las direcciones efectivas. Si ejecutamos la siguiente instrucción, el ensamblador sólo obtiene un byte de memoria fuera del arreglo. El resultado es un traicionero error lógico, por lo que debemos ser muy cuidadosos al comprobar las referencias de arreglos:

```
Mov al,[arregloB+20]    ; AL = ??
```

Arreglo de tipo palabra y doble palabra En un arreglo de palabras de 16 bits, el desplazamiento de cada elemento del arreglo es 2 bytes más adelante del anterior. Ésta es la

razón por la cual sumamos 2 a **arregloW** en el siguiente ejemplo, para llegar al segundo elemento:

```
.data

arregloW WORD 100h,200h,300h

.code

mov ax,arregloW               ; AX = 100h

mov ax,[arregloW+2]           ; AX = 200h
```

De manera similar, el segundo elemento en un arreglo de tipo doble palabra se encuentra en 4 bytes más adelante del primero:

```
.data

arregloD DWORD 10000h,20000h

.code

mov eax,arregloD             ; EAX = 10000h

mov eax,[arregloD+4]         ; EAX = 20000h
```

Programa de ejemplo (movimientos)

El siguiente programa demuestra la mayor parte de los ejemplos de transferencias de datos

```
TITLE ejemplo de transferencia de datos (Moves.asm)

; Ejemplo del capítulo 4. Demostración de MOV y

; XCHG con operando directos y desplazamiento directo

; Última actualización: 06/01/2006

INCLUDE Irvine32.inc

.data

val1 WORD 1000h

val2 WORD 2000h

arregloB BYTE 10h,20h,30h,40h,50h

arregloW WORD 100h,200h,300h
```

```
arregloD DWORD 10000h,20000h
.code
main proc
; MOVZX
       mov   bx,0A69Bh
       movzx eax,bx                          ; EAX = 0000A69Bh
       movzx edx,bl                          ; EDX = 0000009Bh
       movzx cx,bl                           ; CX  = 009Bh
; MOVSX
       mov   bx,0A69Bh
       movzx eax,bx                          ; EAX = FFFFA69Bh
       movzx edx,bl                          ; EDX = FFFFFF9Bh
       mov   bl,7Bh
       mov   cx,bl                           ; CX  = 007Bh
; Intercambio de memoria a memoria:
      mov  ax,val1                           ; AX = 1000h
      xchg ax,val2                           ; AX = 2000h, val2 = 1000h
      mov  val1,ax                           ; val1 = 2000h
; Direccionamiento con desplazamiento directo (arreglo de  bytes):
      mov al, arregloB                       ; AL = 10h
      mov al,[arregloB+1]                    ; AL = 20h
      mov al, [arregloB+2]                   ; AL = 30h
; Direccionamiento con desplazamiento directo (arreglo de palabra):
      mov al,arregloW                        ; AX = 100h
      mov ax,[arregloW+2]                    ; AX = 200h
; Direccionamiento con desplazamiento directo (arreglo de dobles
palabras):
```

```
      mov eax,arregloD                          ; EAX = 10000h
      mov eax,[arregloD+4]                      ; EAX = 20000h
      mov eax,[arregloD+TYPE arregloD]          ; EAX = 20000h
     exit
main ENDP
END main
```

*Este programa no genera resultados en la pantalla

Suma y resta

La aritmética es un tema bastante extenso en el lenguaje, por lo que lo dividiremos en pasos. Aquí nos enfocaremos en la suma y resta de enteros en el capítulo 7 presentaremos la multiplicación y división de enteros. En el capítulo 17 mostraremos cómo realizar operaciones aritméticas de punto flotante, con un conjunto de instrucciones completamente distinto. Vamos a empezar con INC (incremento), DEC (decremento), ADD, SUB y NEG (negación). El tema de cómo se ven afectadas las banderas de estado (Acarreo, Signo, Cero, etc.).

Instrucciones INC y DEC

Las instrucciones INC (incremento) y DEC (decremento) suman 1 y restan 1 de un solo operando, respectivamente, la sintaxis es:

```
INC reg/mem
DEC reg/mem
```

A continuación se muestran algunos ejemplos:

```
.data
miPalabra WORD 1000h
.code
```

```
inc miPalabra              ; 1001h

mov bx,miPalabra

dec bx                     ; 1000h
```

Las banderas Desbordamiento, Signo, Cero, Acarreo auxiliar y Paridad cambian de acuerdo al valor del operando de destino. No afectan a la bandera de Acarreo (lo cual es un poco sorprendente).

Instrucción ADD

La instrucción ADD suma un operando de origen con uno de destino del mismo tamaño. La sintaxis es;

```
ADD dest,origen
```

Origen permanece sin cambio en la operación y la suma se almacena en el operando de destino. El conjunto de posibles operandos es el mismo que para la instrucción MOV. He aquí un ejemplo breve que suma dos enteros de 32 bits:

```
.data

var1 DWORD 10000h

var2 DWORD 20000h

.code

mov eax,var1                    ; EAX = 10000h

add eax,var2                     ; EAX = 30000h
```

Banderas Las banderas Acarreo, cero, Signo, Desbordamiento, Acarreo auxiliar y Paridad cambian de acuerdo con el valor del operando de destino.

Instrucción SUB

La instrucción SUB resta un operando de origen a un operando de destino. El conjunto de posibles operaciones es el mismo que para la instrucción ADD y MOV. La sintaxis es:

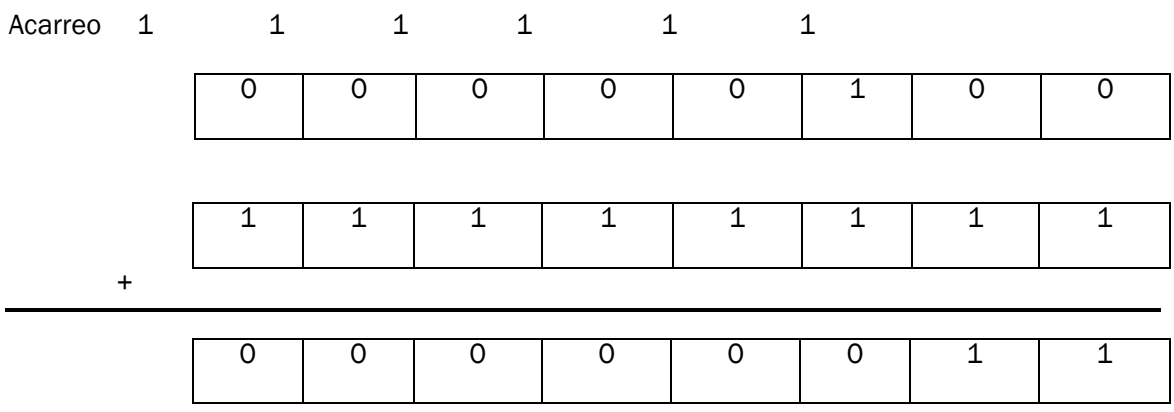
```
SUB dest,origen
```

He aquí un ejemplo breve de código que se resta dos enteros de 32 bits:

```
.data
```

```
var1 DWORD 30000h

var2 DWORD 10000h

.code

mov eax, var1                          ; EAX = 30000h

sub eax,var2                           ; EAX = 20000h
```

Una manera sencilla de realizar una resta sin tener que crear nuevos circuitos digitales es negar y después sumar, por ejemplo 4 -1 puede interpretarse como 4 + (-1). Para los números negativos se utiliza la notación de complementos a dos, por lo que -1 se representa mediante 11111111:



Banderas Las banderas Acarreo, Cero, Signo, Desbordamiento, Acarreo auxiliar y Paridad cambian de acuerdo con el valor del operando de destino.

Instrucción NEG

La instrucción NEG (negación) invierte el signo de un número convirtiéndolo en su complemento a dos. Se permiten los siguientes operandos:

```
NEG reg

NEG mem
```

(Para calcular el complemento de a dos de un número se invierten todos los bit en el operando de destino y se le suma 1)

Banderas Las banderas Acarreo, Cero, Signo, Desbordamiento, Acarreo auxiliar y paridad cambian de acuerdo con el valor de operando de destino.

Implementación de expresiones aritméticas

Armado con las instrucciones ADD, SUB y NEG, tenemos los medios para implementar expresiones, aritméticas que involucran la suma, resta y negación en lenguaje ensamblador. En otras palabras, uno puede simular lo que un compilador en C++ podría hacer al leer una expresión como:

valR = -valX + (valY – valZ);

Se utilizan las siguientes variables de 32 bits:

```
valR SDWORD ?

valX SDWORD 26

valY SDWORD 30

valZ SDWORD 40
```

Al traducir una expresión, se evalúa cada término por separado y se combinan los términos al final. Primero, negamos una copia de **valX**:

```
; primer término: -valX

mov eax,valX

neg eax                    ; EAX = -26
```

Después **valY** se copia a un registro y se resta **valZ**:

```
mov ebx,valY

sub ebx,valZ               ; EBX = -10
```

```
Por último, se suman los dos términos (en EAX y EBX):

add eax,ebx
```

```
mov valR,eax              ; valR = -36
```

Banderas afectadas por la suma y la resta

Al ejecutar instrucciones aritméticas, a menudo es conveniente saber algo acerca del resultado. ¿Es negativo, positivo o cero? ¿Es demasiado grande o demasiado pequeño para caber en el operador de destino? Las respuestas a tales preguntas nos pueden ayudar a detectar errores de cálculo que de otra manera podrían ocasionar un comportamiento errático. Utilizamos los valores de las banderas de estado de la CPU para comprobar el resultado de las operaciones aritméticas. También utilizamos los valores de banderas de estado para activar instrucciones de bifurcación condicional, las herramientas básicas de la lógica de programación. He aquí un breve vistazo a las banderas de estado. Más adelante las veremos con detalle:

- La bandera de Acarreo indica un desbordamiento de enteros sin signo. Por ejemplo, si una instrucción tiene un operando de destino de 8 bits, pero genera un resultado mayor que el 11111111 binario, se activa la bandera Acarreo.
- La bandera de Desbordamiento indica un desbordamiento de enteros con signo. Por ejemplo, si una instrucción tiene un operando de destino de 16 bits, pero genera un resultado negativo menor que el número -32768, se activa la bandera de Desbordamiento.
- La bandera Cero indica que una operación produjo cero como resultado. Por ejemplo, si se resta un operando de otro de igual valor, se activa la bandera Cero.
- La bandera Signo indica que una operación produjo un resultado negativo. Si se activa el bit más significativo de operando de destino, se activa la bandera Signo.
- La bandera Paridad cuenta el número de bits que son 1 en el byte menos significativo del operando de destino.
- La bandera de Acarreo auxiliar se activa cuando un bit 1 se acarrea hacia fuera de la posición 3 en el byte menos significativo del operando destino.

Operaciones sin signo: Cero, Acarreo y Acarreo auxiliar

La bandera Cero se activa cuando el resultado de una operación aritmética es cero. Los siguientes ejemplos muestran el estado del registro de destino y de la bandera Cero, después de ejecutar las instrucciones SUB, INC y DEC

```
mov ecx,1

sub  ecx,1           ; ECX = 0, ZF = 1

mov eax,0FFFFFFFFh

inc eax              ; EAX = 0, ZF = 1

inc eax              ; EAX = 1, ZF = 0

dec eax              ; EAX = 0, ZF = 1
```

La suma y la banderas Acarreo La operación de bandera Acarreo es más fácil de explicar si consideramos la suma y resta por separado. Cuando se suman dos enteros sin signo, la bandera Acarreo es una copia del acarreo que sale del MSB (bit más significativo) del operando de destino. Por intuición, podemos decir que CF = 1 cuando la suma excede al tamaño de almacenamiento de su operando de destino. En el siguiente ejemplo, ADD activa la bandera de Acarreo, debido a que la suma (100h) es demasiado grande para AL:

```
mov al,0FFh

add al,1              ; AL = 00, CF = 1
```

La siguiente figura muestra lo que ocurre a nivel de bits cuando se suma 1 a 0FFh. El acarreo que sale de la posición del bit más alto de AL (MSB) se copia en la bandera de Acarreo:

(Bit de Acarreo) 1← 1 1 1 1 1 1 1 1

1	1	1	1	1	1	1	1

0	0	0	0	0	0	0	0

+

Por otro lado, si se suma 1 a 00FFh en AX, la suma cabe fácilmente en 16 bits y la bandera Acarreo se borra:

```
mov ax,00FFh

add ax,1                              ; AX = 0100h,  CF = 0
```

Pero si se suma 1 a FFFFh en el registro AX, se genera un Acarreo hacia fuera de la posición del bit superior de AX:

```
mov ax,0FFFFh

add ax,1                              ; AX = 0000, CF = 1
```

La resta y la bandera de Acarreo Una operación de resta activa la bandera de Acarreo cuando se resta un entero sin signo más grande de uno más pequeño. Es más fácil considerar

el efecto de la resta sobre la bandera Acarreo desde un punto de vista relacionado con el hardware. Vamos a suponer por un momento que la CPU puede negar un entero positivo sin signo, formando su complemento a dos:

1. El operando de origen se niega y se suma al destino.
2. El acarreo que sale de MSB se invierte y se copia a la bandera Acarreo.

Vamos a restar 2 de 1, como operandos de 8 bits. Después de negar el 2, sumamos los enteros:

0 <-

0	0	0	0	0	0	0	1

\+

1	1	1	1	1	1	1	1

_____

0	0	0	0	0	0	1	1

La suma (255) no es válida. El acarreo que sale del bit 7 se invierte y se coloca en la bandera Acarreo, por lo que CF = 1. He aquí el correspondiente código en ensamblador:

```
mov al,1

sub al,2                ; AL = FFh, CF = 1
```

Las instrucciones INC y DEC no afectan a la bandera Acarreo. Si se aplica NEG a un operando distinto de cero, siempre se activa la bandera Acarreo.

Acarreo auxiliar La bandera de Acarreo auxiliar (AC) indica un acarreo o préstamo (borrow) en el bit 3 del operando de destino. Se utiliza principalmente en la aritmética con números decimales codificados en binario (BCD), pero puede usarse en otros contextos. Supongamos que sumamos 1 a 0Fh. La suma (10h) contiene un 1 en la posición del bit 4 que se acarreó de la posición del bit 3:

```
mov al ,0Fh

add al,1                ; AC = 1
```

He aquí la aritmética:

```
   0 0 0 0 1 1 1 1

+  0 0 0 0 0 0 0 1

--------------------

   0 0 0 1 0 0 0 0
```

Paridad La bandera paridad (PF) se activa cuando el byte menos significativo del destino tiene un número par de bits que son 1. Las siguientes instrucciones ADD y SUB alteran la paridad de AL:

```
mov al,10001100b

add al,00000010b      ; AL = 10001110, PF = 1

sub al,10000000b      ; AL = 00001110, PF = 0
```

Después de la instrucción ADD, AL contiene el número binario 10001110 (cuatro bits 0 y cuatro bits 1), y PF = 1. Después de SUB, AL contiene un número impar de bits 1, por lo que PF = 0.

Operaciones con signo: banderas Signo y Desbordamiento

Bandera signo La bandera signo se activa cuando el resultado de una operación aritmética con signo es negativo. El siguiente ejemplo resta un entero más grande (5) de un entero más pequeño (4):

```
mov eax,4

sub eax,5             ;EAX = -1, SF = 1
```

Desde el punto de vista mecánico, la bandera Signo es una copia del bit superior del operando de destino. El siguiente ejemplo muestra los valores hexadecimales de BL cuando se genera un resultado negativo:

```
mov bl,1              ; BL = 01h
```

```
sub bl,2              ; BL = FFh (-1)
```

Bandera desbordamiento La bandera de Desbordamiento se active cuando el resultado de una operación aritmética con signo provoca que el operando de destino tenga un desbordamiento por exceso (overflow) o por defecto (underflow). Por ejemplo, del capítulo 1 sabemos que el valor de byte entero con signo más grande posible es +127 si se le suma uno se produce desbordamiento por exceso (overflow):

```
mov al,+127

add al,1              ; OF = 1
```

De manera similar el valor de byte entero con signo más pequeño posible -128. Si le restamos 1, se produce un desbordamiento por defecto (underflow). El valor del operando de destino no almacena un resultado aritmético válido, por lo que se activa la bandera Desbordamiento:

```
mov al, -128

sub al,1              ; OF =1
```

La prueba de suma Hay una manera muy sencilla de saber si ha ocurrido un desbordamiento con signo cuando se suman dos operandos. El desbordamiento ocurre cuando:

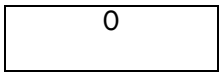
- Dos operandos positivos generan una suma negativa.
- Dos operandos negativos generan una suma positiva.

El desbordamiento nunca se produce cuando los signos de los dos operandos de la suma son distintos.

Cómo el hardware detecta el desbordamiento La CPU utiliza un interesante mecanismo para determinar el estado de la bandera Desbordamiento, Después de una operación de suma o resta. Al bit que se acarrea hacia fuera del MSB (Bit más significativo) de un operando se le aplica un OR exclusivo con el bit que se acarrea hacia el MSB. El valor resultante se coloca en la bandera Desbordamiento. Por ejemplo, al sumar los enteros binarios de 8 bits 10000000 y 11111110 no se produce ningún acarreo hacia el bit 7 (MSB), pero sí hay un acarreo del bit 7 a la bandera Acarreo:

*No hay acarreo del bit 6 al 7

	7 6 5 4 3 2 1 0
	1 0 0 0 0 0 0 0
+	1 1 1 1 1 1 1 0
	0 1 1 1 1 1 1 1



En otras palabras la operación 1 XOR 0 produce OF = 1.

Instrucción NEG La instrucción NEG produce un resultado inválido si el operando de destino no puede almacenarse en forma correcta. Por ejemplo, si movemos -128 a AL y tratamos de negarlo, el valor correcto (+128) no cabrá en AL, La bandera Desbordamiento se activa, indicando que AL contiene un valor inválido:

```
mov al,-128                ; AL = 100000000b

neg al                     ; AL = 100000000b, OF = 1
```

Por otro lado, si +127 se niega, el resultado es válido y la bandera Desbordamiento se borra:

```
mov al,+127                ; AL = 011111111b

neg al                     ; AL = 100000001b, OF = 0
```

Programa de ejemplo (SumaResta3)

El siguiente programa implementa varias expresiones aritméticas, usando instrucciones ADD, SUB, INC, DEC y NEG, y muestra cómo se ven afectadas ciertas banderas de estado:

```
TITLE suma y resta (AddSub3.asm)

; Ejemplo del capítulo 4. Demostración de instrucciones

; ADD, SUB, INC, DEC y NEG, y la manera en

; que afectan a las banderas de estado de la CPU.

; última actualización: 06/01/2006

INCLUDE Irvine32.inc

.data

valR SDWORD ?

valX SDWORD 26

valY SDWORD 30

valZ SDWORD 40

.code
```

```
main PROC
          ; INC y DEC
          mov ax,1000h
          inc ax                        ; 1001h
          dec ax                        ; 1000h
          ; Expresión: ValR = -valX + (valY – valZ)
          mov eax,valX
          neg eax                       ; -26
          mov ebx,valY
          sub ebx,valZ                  ; -10
          add eax,ebx
         mov valR,eax                   ; -36
         ; Ejemplo de la bandera Cero:
         mov cx,1
         sub cx,1                       ; ZF = 1
         mov ax,0FFFFh
         inc ax                         ; ZF = 1
         ; Ejemplo de la bandera Signo:
         mov cx,0
         sub cx,1                       ; SF = 1
         mov ax, 7FFFh
         add ax,2                       ; SF = 1
         ; EJemplo de bandera Acarreo
         mov al,0FFh
         add al,1                       ; CF = 1, AL = 00
         ; Ejemplo de la bandera Desbordamiento:
```

```
        mov al,+127
        add al,1                    ; 0F = 1
        mov al,-128
        sub al,1                    ; 0F = 1
        exit
main ENDP
END main
```

Operadores y directivas relacionadas con los datos

Los operadores y las directivas no son instrucciones ejecutables; en vez de eso, el ensamblador las interpreta. Podemos usar varias directivas de MASM para obtener información acerca de las direcciones y características de tamaño de los datos:

- El operador OFFSET devuelve la distancia de una variable, a partir del inicio de su segmento circulante.
- El operador PTR nos permite redefinir el tamaño predeterminado de una variable.
- El operador TYPE devuelve el tamaño (en bytes) de un operando, o cada elemento en un arreglo.
- El operador LENGTHOF devuelve el número de elementos en un arreglo.
- El operador SIZEOF devuelve el número de bytes utilizados por un inicializador de arreglos.

Además la directiva LABEL proporciona una manera de redefinir la misma variable con distintos atributos de tamaño. Los operadores y las directivas en este capítulo representan sólo un pequeño subconjunto de operadores que soporta MASM.

> MASM sigue ofreciendo soporte para la directiva heredadas LENGTH (en vez de LENGTHOF) y SIZE (en vez de SIZEOF).

Operador OFFSET

El operador OFFSET devuelve el desplazamiento de una etiqueta de datos. El desplazamiento representa la distancia (en bytes) de la etiqueta, a partir del inicio del segmento de datos). En modo protegido, los desplazamientos son de 32 bits. En modo direccionamiento real, los desplazamientos son de 16 bits.

Ejemplo de OFFSET

En el siguiente ejemplo, declaramos tres tipos distintos de variables:

```
.data

valB  BYTE  ?

valW  WORD  ?

valD  DWORD ?

valD2 DWORD ?
```

Si **valB** se encontrara en el desplazamiento 00404000 (hexadecimal), el operador OFFSET devolvería los siguientes valores:

```
mov esi,OFFSET valB             ; ESI = 00404000

mov esi,OFFSET valW             ; ESI = 00404001

mov esi,OFFSET valD             ; ESI = 00404003

mov esi,OFFSET valD2            ; ESI = 00404007
```

OFFSET también puede aplicarse a un operando de desplazamiento directo. Supongamos que **miArreglo** contiene cinco palabras de 16 bits. La siguiente instrucción MOV obtiene el desplazamiento de **miArreglo**, le suma 4 y mueve la suma a ESI:

```
.data

miArreglo WORD 1,2,3,4,5

.code

mov esi,OFFSET miArreglo + 4
```

Directiva ALIGN

La directiva ALIGN alinea una variable a un límite definido por byte, palabra, doble palabra o párrafo. La sintaxis es:

```
ALIGN límite
```

Límite puede ser 1, 2, 4 o 16. Un valor de 1 alinea a la siguiente variable en un límite de 1 byte (el valor predeterminado). Si el límite es de 2, la siguiente variable se alinea en una dirección

con numeración par. Si el límite es 4, la siguiente dirección es un múltiplo de 4. Si el límite es 16, la siguiente dirección es un múltiplo de 16, un límite de párrafo. El ensamblador puede insertar uno o más bytes vacios antes de la variable para corregir la alineación. ¿Por qué molestarse en alinear los datos? Porque la CPU puede procesar los datos almacenados en direcciones con numeración par más rápido que las direcciones con numeración impar.

En la siguiente revisión de un ejemplo de la sección 4, **valB** se encuentra de manera arbitraria en el desplazamiento 00404000. Al insertar la directiva ALIGN 2 antes de **valW**, se le asigna un desplazamiento con numeración par:

```
valB BYTE ?    ; 00404000

ALIGN 2

valW WORD ?    ; 00404002

valB2 BYTE ?   ; 00404004

ALIGN 4

valD DWORD ?    ; 00404008

valD2 DWORD ?   ; 0040400C
```

Observe que **valD** hubiera estado en el desplazamiento 00404005, pero la directiva ALIGN 4 lo cambió al desplazamiento 00404008.

Operador PTR

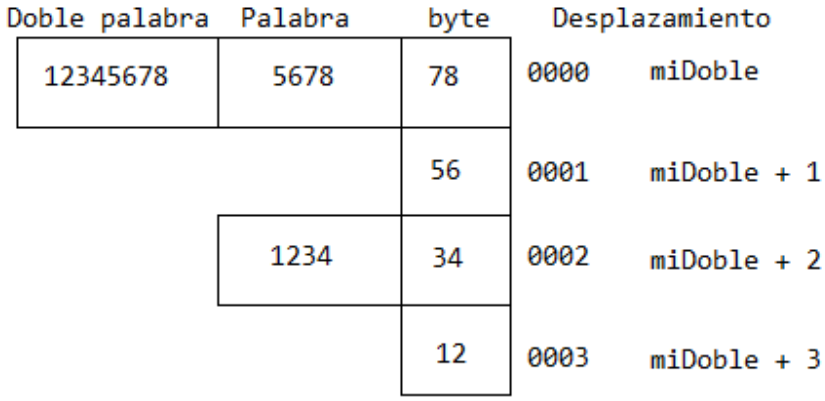
Podemos utilizar el operador PTR para redefinir el tamaño declarado de un operando. Esto sólo es necesario cuando tratamos de acceder a la variable mediante un atributo de tamaño distinto al que utilizamos para declarar la variable. Por ejemplo, supongamos que deseamos mover hacia AX los 16 bits inferiores de una variable de una variable tipo doble palabra llamado **miDoble**. El ensamblador no permitirá el siguiente movimiento ya que los tamaños de los operandos no coinciden:

```
.data

miDoble DWORD 12345678h

.code

mov ax,miDoble              ; error
```

Pero el operador WORD PTR hace posible el movimiento de la palabra de menor orden(5678h) a AX:

mov ax,WORD PTR miDoble

¿Por qué no se movió el número 1234h a AX? Intel utiliza el formato de almacenamiento *Little endian*, en el cual el byte de menor orden se almacena en la dirección inicial de la variable. En la siguiente figura, se muestra la distribución de la memoria **miDoble** de tres formas: primero como doble palabra, después como dos palabras (5678h), y por último como cuatro bytes (78h, 56h, 34h, 12h):



La CPU puede acceder a la memoria en cualquiera de estas tres formas, independientemente de la manera en que se defina una variable. Por ejemplo, si **miDoble** empieza en el desplazamiento 0000, el valor de 16 bits almacenado en esa dirección es 5678h. También podríamos obtener 1234h, la palabra en la ubicación **miDoble+2**, usando la siguiente instrucción:

```
mov ax,WORD PTR [miDoble+2]       ; 1234h
```

De manera similar podríamos usar el operador BYTE PTR para mover un byte individual de miDoble a BL:

```
mov bl,BYTE PTR miDoble           ; 78h
```

Observe que PTR debe usarse en combinación con uno de los tipos de datos estándar del ensamblador: BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD o TERABYTE.

Mover valores más pequeños a destinos más grandes En algunas ocasiones será necesario mover dos valores más pequeños de memoria hacia un operando de destino más grande, En el siguiente ejemplo, la primera palabra se copia a la mitad inferior de EAX y la segunda palabra se copia a la mitad superior. El operador DWORD PTR hace esto posible:

```
.data

listaPalabras WORD 5678h, 1234h

.code

mov eax,DWORD PTR listaPalabras   ; EAX = 12345678h
```

Operador TYPE

El operador TYPE devuelve el tamaño en bytes de un solo elemento de una variable. Por ejemplo, el tipo (TYPE) de un byte es igual a 1, el tipo de una palabra es igual a 2, el tipo de una doble palabra es 4 y el tipo de una palabra cuádruple es 8. He aquí ejemplos de cada uno:

```
.data

var1 BYTE ?

var2 WORD ?

var3 DWORD ?

var4 QWORD ?
```

Expresión	Valor
TYPE var1	1
TYPE var2	2
TYPE var3	3
TYPE var4	4

Operador LENGTHOF

El operador LENGTHOF cuenta el número de elementos de un arreglo, definido por los valores que aparecen en la misma línea que su etiqueta. Utilizaremos los siguientes datos por ejemplo:

```
.data

byte1 BYTE 10,20,30

arreglo1 WORD 30 DUP(?),0,0

arreglo2 WORD 5 DUP(3 DUP(?))

arreglo3 DWORD 1,2,3,4

cadDigitos BYTE “12345678”,0
```

Cuando se utilizan operadores DUP anidados en la definición de un arreglo, LENGTHOF devuelve el producto de los dos contadores. La siguiente tabla presenta los valores devueltos por cada expresión LENGTHOF:

Expresión	Valor
LENGTHOF byte1	3
LENGTHOF arreglo1	30 + 2
LENGTHOF arreglo2	5 * 3
LENGTHOF arreglo3	4
LENGTHOF cadDigitos	9

Si se declara un arreglo que abarca varias líneas de programa, LENGTHOF sólo relaciona los datos de la primera línea como parte del arreglo, en el siguiente ejemplo, LENGTHOF miArreglo devuelve el valor 5

```
miArreglo BYTE 10,20,30,40,50
          BYTE 60,70,80,90,100
```

De manera alternativa, se puede terminar la primera línea como una coma y continuar la lista de inicializadores en la siguiente línea. En el siguiente ejemplo, LENGTHOF miArreglo devuelve el valor 10

```
miArreglo BYTE 10,20,30,40,50,
          BYTE 60,70,80,90,100
```

Operador SIZEOF

El operador SIZEOF devuelve el valor que equivale multiplicar LENGTHOF por TYPE. Por ejemplo, **arregloInt** tiene los valores TYPE = 2 y LENGTHOF 32. Por lo tanto, SIZEOF **arregloInt** es igual a 64:

```
.data
arregloInt WORD 32 DUP(0)
.code
mov eax,SIZEOF arregloInt     ; EAX = 64
```

Directiva LABEL

La directiva LABEL nos permite insertar una etiqueta y proporcionarle un atributo de tamaño sin asignar espacio de almacenamiento. Con LABEL puede usarse todos los atributos de tamaño estándar, como BYTE, WORD, QWORD o TBYTE. Un uso común de LABEL es para proporcionar un nombre y atributo de tamaño alternativos para la variable que se declara a continuación en el segmento de datos. En el siguiente ejemplo, declaramos una etiqueta llamada **val16** justo antes de **val32**, y le damos un atributo WORD:

```
.data

val16 LABEL WORD

val32 DWORD 12345678h

.code

mov ax,val16                    ; AX = 5678h

mov dx,[val16+2]                ; DX = 1234h
```

val16 es un alias para la misma la misma ubicación de almacenamiento llamada **val32**. La directiva LABEL en sí no asigna espacio de almacenamiento.

Algunas veces es necesario construir un entero más grande a partir de dos enteros más pequeños. En el siguiente ejemplo, se carga un valor de 32 bits en EAX, a partir de dos variables de 16 bits:

```
.data

ValorLargo LABEL DWORD

val1 WORD 5678h

val2 WORD 1234h

.code

mov eax,ValorLargo       ; EAX = 12345678h
```

Direccionamiento indirecto

El direccionamiento indirecto no es práctico para el procesamiento de arreglos. Muy raras veces es necesario proporcionar una etiqueta única para cada elemento de arreglo. No es conveniente utilizar desplazamientos constantes para direccionar más de unos cuantos elementos de arreglo. La única forma práctica de manejar un arreglo es utilizar un registro como apuntador (conocido como *direccionamiento indirecto*) y manipular el valor de ese registro. Cuando un operando utiliza el direccionamiento indirecto, se llama *operando indirecto*.

Operandos indirectos

Modo protegido Un operando indirecto puede ser cualquier registro de propósito general de 32 bits (EAX, EBX, ECX, EDX, ESI, EDI y ESP) encerrado entre corchetes. Se asume que el registro debe contener el desplazamiento de ciertos datos. En el siguiente ejemplo, ESI contiene el desplazamiento de **val1**. La instrucción MOV utiliza el operando indirecto como origen, el desplazamiento en ESI se emplea para poder hacer referencia a la dirección de memoria y se mueve un byte a AL:

```
.data

val1 BYTE 10h

.code

mov esi,OFFSET val1

mov al,[esi]                          ; AL = 10h
```

Si el operando de destino utiliza el direccionamiento indirecto, se coloca un nuevo valor en memoriam en la ubicación a la que apunta el registro:

```
mov [esi],bl
```

Modo de direccionamiento real En el modo, un registro de 16 bits almacena el desplazamiento de una variable. Si el registro se utiliza como un operando indirecto, sólo puede ser SI, DI, BX o BP. Por lo general, evitamos usar el registro BP, ya que direcciona a la pila en vez del segmento de datos. En el siguiente ejemplo, SI hace referencia a **val1**:

```
.data

val1 BYTE 10h

.code

main PROC

          inicio

          mov si,OFFSET val1

          mov al,[si]                    ; al = 10h
```

Error de protección general En modo protegido, si la dirección efectiva apunta a una a un área fuera del segmento de datos de nuestro programa, la CPU ejecuta un *error de protección general* (GP). Esto ocurre incluso aunque una instrucción no modifique la memoria, Por ejemplo, si ESI no se inicializara, la siguiente instrucción probablemente generará un error de protección general:

```
mov ax,[esi]
```

Siempre hay que inicializar los registros antes de usarlos como operandos indirectos. Lo mismo se aplica a la programación en lenguaje de alto nivel con subíndices y apuntadores. Los errores de protección general no ocurren en el modo de direccionamiento real, el cual hace que los operandos indirectos sin inicializar sean difíciles de detectar.

Uso de PTR con operandos indirectos Tal vez el tamaño de un operando no esté claro desde el contexto de una instrucción. La siguiente instrucción hace que el ensamblador genere un error de tipo "el operando debe tener un tamaño":

```
inc [esi]
```

El ensamblador no sabe si ESI apunta a un byte, una palabra, una doble palabra o cualquier otro tamaño.

El operador PTR aclara el tamaño del operando:

```
inc BYTE PTR [esi]
```

Arreglos

Los operandos indirectos son útiles al manera arreglos, ya que el valor de un operando indirecto puede modificarse en tiempo de ejecución. De manera similar al subíndice de un arreglo, los operandos indirectos pueden apuntar a distintos elementos del arreglo. En el siguiente ejemplo, **arregloB** contiene 3 bytes. Podemos incrementar ESI y hacer que apunte a cada byte, en orden:

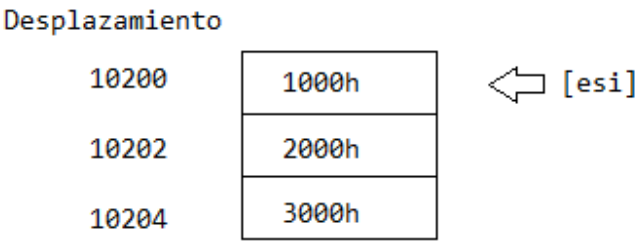
```
.data

arregloB BYTE 10h,20h,30h

.code

mov esi,OFFSET arregloB

mov al,[esi]                        ; AL = 10h

inc esi

mov al,[esi]                        ; AL = 20h

inc esi

mov al,[esi]                        ; AL = 30h
```

Si utilizamos un arreglo de enteros de 16 bits, sumamos 2 a ESI para direccionar cada elemento subsiguiente del arreglo:

```
.data
```

```
arregloW WORD 1000h,2000h,3000h

.code

mov esi,OFFSET arregloW

mov ax,[esi]                              ; AX = 1000h

add esi,2

mov ax,[esi]                              ; AX = 2000h

add esi,2

mov ax,[esi]                              ; AX = 3000h
```

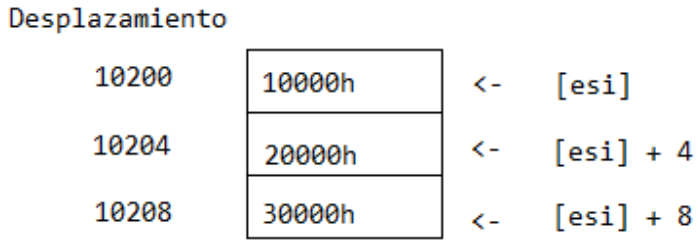
Supongamos que **arregloW** se encuentra en el desplazamiento 10200h. La siguiente instrucción muestra a ESI en relación con los datos del arreglo:



Ejemplo: suma de enteros de 32 bits El siguiente extracto de un programa suma tres dobles palabras. Debe sumarse un desplazamiento de 4 para cada valor subsiguiente del arreglo, ya que las dobles palabras son de 4 bytes:

```
.data

arregloD DWORD 10000h, 20000h, 30000h

.code

mov esi,OFFSET arregloD

mov eax,[esi]                                ; primer número

add esi,4

add eax,[esi]                                ; segundo número

add esi,4

add eax,[esi]                                ; tercer número
```

Si **arregloD** se encuentra en el desplazamiento 10200h, la siguiente ilustración muestra a ESI en relación con los datos del arreglo:



Operandos indexados

Un *operando indexado* suma una constante a un registro para generar una dirección efectiva. Puede usarse cualquiera de los registros de propósito general de 32 bits como registro índice. MASM permite varias formas de notación (los corchetes son parte de la notación):

```
constante[reg]

[constante + reg]
```

La primera forma de notación combina el nombre de una variable con un registro. El nombre de la variable es una constante que representa el desplazamiento de la variable. He aquí ejemplos de muestran ambas formas de notación:

arregloB[esi]	[arregloB + esi]
arregloD[ebx]	[arregloD + ebx]

Los operandos indexados son adecuados para el procesamiento de arreglos. El registro índice debe inicializarse con cero antes de acceder al primer elemento de arreglo:

```
.data

arregloB BYTE 10h,20h,30h

.code

mov esi,0

mov al,[arreglo+ esi]        ; AL = 10h
```

La última instrucción suma ESI al desplazamiento de **arregloB**. La dirección generada por la expresión [**arregloB + ESI**] se emplea para hacer referencia a memoria y el byte en memoria se copia a AL.

Suma de desplazamientos El segundo tipo de direccionamiento indexado combina a un registro con un desplazamiento constante. El registro índice almacena la dirección base de un arreglo o estructura, y la constante identifica los desplazamientos de varios elementos del arreglo. El siguiente ejemplo muestra cómo hacer esto con un arreglo de palabras de 16 bits:

```
.data

arregloW WORD 1000h,2000h,3000h

.code

mov esi,OFFSET arregloW

mov ax,[esi]                              ; AX = 1000h

mov ax,[esi+2]                            ; AX = 2000h

mov ax,[esi+4]                            ; AX = 3000h
```

Uso de registros de 16 bits Es común utilizar registros de 16 bits con operandos indexados en el modo de direccionamiento real. En este caso, estamos limitados a utilizar SI,DI,BX o BP:

```
mov al,arregloB[si]

mov ax,arregloW[di]

mov eax,arregloD[bx]
```

Al igual que n el caso de los operandos indirectos, se debe evitar utilizar BP, excepto cuando se direccionen datos en la pila.

Factores de escala en operandos indexados

Los operandos indexados deben tener en cuenta el tamaño de cada elemento de arreglo al calcular los desplazamientos. Por ejemplo, si usamos un arreglo de dobles palabras, multiplicamos el subíndice (3) por 4(el tamaño de una doble palabra) para generar el desplazamiento del elemento de arreglo que contiene 400h:

```
.data
```

```
arregloD DWORD 100h, 200h, 300h, 400h

.code

mov esi,3* TYPE arregloD                ; desplazamiento de arregloD[3]

mov eax,arregloD[esi]                   ; EAX = 400h
```

Los diseñadores de Intel querían facilitar una operación común a los escritores de compiladores, por lo que proporcionó una forma de calcular los desplazamientos, usando un *factor de escala*. Este factor de escala es el tamaño del componente del arreglo (palabra = 2, doble palabra = 4, o palabra cuádruple = 8). Vamos a revisar el ejemplo anterior, asignando a ESI el subíndice (3) del arreglo y multiplicando ESI por el factor de escala (4) para dobles palabras:

```
.data

arregloD DWORD 1,2,3,4

.code

mov esi,3                               ; subíndice

mov eax,arregloD[esi*4]                 ; EAX = 400h
```

El operador TYPE puede hacer el indexado más flexible, en caso de que el arregloD se redefina como otro tipo en el futuro:

```
mov esi,3                               ; subíndice

mov eax,arregloD[esi*TYPE arregloD]     ; EAX = 400h
```

Apuntadores

A una variable que contiene la dirección de otra variable se le conoce como *apuntador*. Los apuntadores son una estupenda herramienta para manipular arreglos y estructuras de datos, y hacen posible la asignación dinámica de memoria. Los programas basados en Intel utilizan dos tipos básicos de apuntadores, cercanos (NEAR) y lejanos (FAR). Sus tamaños se ven afectados por el modo actua del procesador (real de 16 bits o protegido de 32 bits), como se muestra a continuación:

	Modo de 16 bits	Modo de 32 bits
Apuntador NEAR	Desplazamiento de 16 bits, a partir del inicio del segmento de datos	Desplazamiento de 32 bits, a partir del inicio del segmento de datos
Apuntador FAR	Dirección de desplazamiento de segmento de 32 bits	Dirección de desplazamiento de selección de segmento de

		40 bits

En este resumen, los programas en modo protegido utilizan apuntadores cercanos (near), por lo que se almacenan en variables tipo doble palabra. He aquí dos ejemplos: **apuntB** contiene el desplazamiento y **apuntW** contiene el desplazamiento de **arregloW**:

```
arregloB BYTE  10h,20h,30h,40h

arregloW WORD  1000h,2000h,3000h

apuntB   DWORD arregloB

apuntW   DWORD arregloW
```

También se puede utilizar el operador OFFSET para que la relación sea más clara:

```
apuntB DWORD OFFSET arregloB

apuntW DWORD OFFSET arregloW
```

Los lenguajes de alto nivel ocultan de manera intencional los detalles físicos acerca de los apuntadores, ya que sus implementaciones varían entre las distintas arquitecturas de las máquinas. En el lenguaje ensamblador, como estamos tratando una sola implementación, examinamos y utilizamos los apuntadores en el nivel físico. Este enfoque nos permite eliminar algunos de los misterios que rodean a los apuntadores.

Uso del operador TYPEDEF

El operador TYPEDEF nos permite crear un tipo definido por el usuario, que tiene todas las características de un tipo integrado a la hora de definir las variables. TYPEDEF es ideal para crear variables tipo apuntador. Por ejemplo, la siguiente declaración crea un nuevo tipo de datos PBYTE, que es un apuntador a bytes:

```
PBYTE TYPEDEF PTR BYTE
```

Esta declaración por lo general, se coloca cerca del principio de un programa, antes del segmento de datos, Así. Podrían definirse variables mediante el uso de PBYTE:

```
.data

arregloB BYTE 10,20,30,40

apunt1 PBYTE ?               ; sin inicializar

apunt2 PBYTE arregloB        ; apunta a un arreglo
```

Programa de ejemplo: apuntadores El siguiente programa (apuntadores.asm) utiliza TYPEDEF para crear tres tipos de apuntadores (PBYTE, PWORD, PDWORD). Crea varios apuntadores, asigna varios desplazamientos de arreglo y emplea apuntadores para hacer la referencia a memoria:

```
TITLE Apuntador              (Apuntadores.asm)

; Demostración de los apuntadores y TYPEDEF.

; Última actualización: 06/01/2006

INCLUDE Irvine32.inc

; Crea tipos definidos por los usuarios.

PBYTE TYPEDEF PTR BYTE                      ; apuntador a bytes

PWORD TYPEDEF PTR WORD                      ; apuntador a palabras

PDWORD TYPEDEF PTR DWORD                    ; apuntador a dobles palabras

.data

arregloB BYTE 10h,20h,30h

arregloW WORD 1,2,3

arregloD DWORD 4,5,6

; Crea algunas variables tipo apuntador

apunt1 PBYTE arregloW

apunt2 PWORD arregloW

apunt3 PDWORD arregloD

.code

main PROC

; Usa los apuntadores para accede a los datos.

     mov esi,apunt1

     mov al,[esi]                           ; 10h

     mov esi,apunt2

     mov ax,[esi]                           ; 1
```

```
    mov esi,apunt3
    mov eax,[esi]                          ; 4
main ENDP
END main
```

Instrucciones JMP y LOOP

De manera predeterminada, la CPU carga y ejecuta los programas en forma secuencial. Pero la instrucción actual podría ser *condicional*, lo cual significa que transfiere el control a una nueva ubicación en el programa, con base en los valores de las banderas de estado de la CPU (Cero, Signo, Acarreo, etc.). Los programas en lenguaje ensamblador utilizan instrucciones condicionales para implementar instrucciones de alto nivel, tales como las instrucciones IF y los ciclos. Cada una de las instrucciones condicionales implica una posible transferencia de control (salto) hacia una dirección de memoria distinta. Una *transferencia de control*, o *bifurcación*, es una manera de alterar el orden en el que se ejecutan las instrucciones. Hay dos tipos básicos de transferencias:

- Transferencia incondicional: en todos los casos el programa se transfiere (bifurca) hacia una nueva ubicación; se carga una nueva dirección de memoria en el apuntador de instrucciones, lo cual provoca que la ejecución continúe en la nueva dirección. La instrucción JMP es un buen ejemplo.
- Transferencia condicional: el programa se bifurca si se cumple cierta condición. Puede combinarse una alta variedad de instrucciones de transferencia condicional para crear estructuras lógicas condicionales. La CPU interpreta las condiciones de verdadero/falso de acuerdo con el contenido de los registros ECX y Flags.

Instrucción JMP

La instrucción JMP es una transferencia incondicional hacia un destino, la cual se identifica mediante una etiqueta de código que el ensamblador traduce en un desplazamiento. La sintaxis es:

JMP *destino*

Cuando la CPU ejecuta una transferencia incondicional, el desplazamiento de destino (a partir del inicio del segmento código) se mueve hacia el apuntador de instrucciones, lo cual provoca que la ejecución continúe en la nueva ubicación. Bajo circunstancias normales, sólo se puede saltar a una etiqueta dentro del procedimiento actual.

Creación de un ciclo La instrucción JMP proporciona una manera sencilla de crear un ciclo saltando a una etiqueta en la parte superior del ciclo

```
superior:
        .
        .
        jmp superior       ; repite el ciclo infinito
```

JMP es incondicional, por lo que el ciclo continuará infinitamente, a menos que se encuentre otra forma de salir del ciclo.

Instrucción LOOP

La instrucción LOOP repite un bloque de instrucciones, un número específico de veces. ECX se utiliza de manera automática como contador, y se decrementa cada vez que se repite el ciclo. Su sintaxis es:

LOOP *destino*

Para la ejecución de la instrucción LOOP se requieren dos pasos: primero, se resta 1 a ECX. Después, ECX se compara con cero. Si no es igual a cero, se utiliza un salto hacia la etiqueta identificada por *destino*. En caso contrario, si ECX es igual a cero, no se realiza ningún salto y el control pasa a la instrucción que sigue después del ciclo.

> En el modo direccionamiento real. CX es el contador de ciclo predeterminado para la instrucción LOOP. Por otro lado, la instrucción LOOPD utiliza a ECX como el contador de ciclo, y la instrucción LOOPW utiliza a CX como el contador de ciclo.

En el siguiente ejemplo, sumamos 1 a AX cada vez que se repite el ciclo. Cuando termina el ciclo, AX = 5 y ECX = 0:

```
      mov ax,0
      mov ecx,5
L1:
      inc ax
      loop L1
```

Un error común de programación es inicializar de manera inadvertida a ECX con cero antes de empezar un ciclo. Si esto ocurre, la instrucción LOOP decrementa ECX para que quede en FFFFFFFFh, ¡y el ciclo se repite 4,294,967,296 veces! Si CX es el contador del ciclo (modo direccionamiento real), se repite 65,535 veces.

El destino del ciclo debe estar a una distancia entre -128 y +127 bytes del contador de la ubicación actual. Las instrucciones de máquina tienen un tamaño aproximado de 3 bytes, por lo que un ciclo podría contener, en promedio, un máximo de 42 instrucciones. A continuación se muestra un ejemplo de un mensaje de error generado por MASM, debido a que la etiqueta de destino de una instrucción LOOP estaba demasiado alejada:

```
error A2075: jump destination too far: by 15 byte(s)
```

Si modificamos a ECX dentro del ciclo, tal vez la instrucción LOOP no funcione en forma apropiada. En el siguiente ejemplo, ECX se incrementa dentro del ciclo. Nunca llega a cero, por lo que el ciclo nunca se detiene:

```
superior:
        .
        .
        inc ecx
        loop superior
```

Si se agotan los registros y se necesita a ECX para otro fin, se puede guardar su contenido en una variable al principio del ciclo y restaurarlo justo antes de la instrucción LOOP:

```
.data
cuenta DWORD ?
.code
        mov ecx,100                     ; establece la cuenta del ciclo
superior:
        mov cuenta,ecx                  ; guarda la cuenta
        .
        mov ecx,20                      ; modifica ECX
        .
        mov ecx,cuenta                  ; restaura la cuenta del ciclo
        loop superior
```

Ciclos anidados Al crear un ciclo dentro del otro, hay que tener cierta consideración especial con el contador de ciclo exterior en ECX. Se puede guardar en una variable:

```
.data
```

```
cuenta DWORD ?

.code

          mov ecx,100           ; establece la cuenta del ciclo exterior

L1:

     mov cuenta,ecx             ; guarda la cuenta del ciclo exterior

     mov ecx,20

L2:

     .

     .

     loop L2                    ; repite el ciclo anterior

     mov ecx,cuenta             ; restaura la cuenta del ciclo exterior

     loop L1                    ; repite el ciclo exterior
```

Como regla general, hay que evitar anidar ciclos más de dos niveles. De no ser así, la administración de los contadores del los ciclos se vuelve demasiado problemática. Si el algoritmo que se utiliza requiere de ciclos con más de 2 niveles de anidación, se puede mover algunos de los ciclos internos hacia las subrutinas.

Suma de un arreglo de enteros

No hay una tarea más común cuando se empieza a programar que el cálculo de la suma de los elementos en un arreglo. En lenguaje ensamblador, deben seguirse estos pasos:

1. Asignar la dirección de arreglo a un registro que servirá como operando indexado.
2. Establecer ECX con el número de elementos en el arreglo (en modo de 16 bits se utiliza CX).
3. Asignar cero al registro que acumula la suma.
4. Crear una etiqueta para marcar el inicio del ciclo.
5. En el cuerpo del ciclo, usar direccionamiento indirecto para sumar un elemento individual del arreglo con el registro que almacena la suma.
6. Establecer el registro índice para que apunte el siguiente elemento del arreglo.
7. Usar una instrucción LOOP para repetir el ciclo desde la etiqueta inicial.

Los pasos del 1 al 3 pueden realizarse en cualquier orden. He aquí un programa corto que realiza el trabajo:

```
TITLE suma de un arreglo (SumaArreglo.asm)

; Este programa suma un arreglo de palabras.
```

```
; Última actualización: 06/01/2006
INCLUDE Irvine32.inc
.data
arregloint WORD 100h,200h,300h,400h
.code
 main PROC
          mov     edi, OFFSET arregloint     ; dirección del arregloint
          mov     ecx, LENGTHOF arregloint   ; contador de ciclos
          mov     ax,0                       ; pone el acumulador en cero
L1:
          add     ax,[edi]                   ; agrega un entero
          add     edi, TYPE arregloint       ; apunta al siguiente entero
          loop    L1                         ; repite hasta que ECX = 0
          exit
main ENDP
END main
```

Copia de una cadena

A menudo, los programas tienen que copiar bloques extensos de datos, de una ubicación a otra. Los datos pueden ser arreglos o cadenas, pero pueden contener cualquier tipo de objetos. Vamos a ver cómo se puede hacer esto en lenguaje ensamblador, mediante un ciclo que copia una cadena. El direccionamiento indexado funciona bien para este tipo de operación, ya que el mismo registro índice hace referencia a ambas cadenas. La cadena de destino debe tener suficiente espacio disponible para recibir los caracteres copiados, incluyendo el byte nulo al final:

```
TITLE Copia de una cadena    (CopiaCad.asm)
; Este programa copia una cadena.
```

```
; Última actualización: 06/01/2006
INCLUDE Irvine32.inc
.data
origen BYTE “Esta es la cadena de origen”,0
destino BYTE SIZEOF origen DUP(0)
.code
main PROC
     mov esi,0                ; registro índice
     mov ecx, SIZEOF origen   ; contador del ciclo
L1:
     mov al,origen[esi]       ; obtiene un carácter del origen
     mov destino[esi],al      ; lo almacena en el destino
     inc esi                  ; se mueve al siguiente carácter
     loop L1                  ; repite el proceso para toda la cadena
     exit
main ENDP
END main
```

*La instrucción MOV no puede tener dos operandos de memoria, por lo que cada carácter se mueve de la cadena de origen a AL, y después de AL a la cadena de destino.

5 Procedimientos

Enlace con una biblioteca externa

Si se invierte tiempo, se podría escribir código detallado para las operaciones de entrada-salida en lenguaje ensamblador. Es algo muy parecido al proceso de ensamblar el motor de un automóvil cada vez que queramos dar un paseo. Un trabajo interesante, pero que consume mucho tiempo. En el capítulo 11 veremos cómo se manejan las operaciones de entrada-salida en modo protegido de Windows. Es muy divertido, además de que se nos abrirá un mundo nuevo cuando veamos las herramientas disponibles. Sin embargo, por ahora las operaciones de entrada-salida deben ser sencillas mientras se aprende los fundamentos del lenguaje ensamblador. En este capítulo veremos cómo llamar a los procedimientos de las bibliotecas de enlace del libro, llamadas **Irvine32.lib** e **Irvine16.lib**. El código fuente completo de la biblioteca está disponible en el sitio Web del libro y se actualiza con regularidad.

La biblioteca Irvine32 es para programas escritos en modo protegido de 32 bits. Contiene procedimientos para enlazarse con la API de MS-Windows al generar operaciones de entrada-salida. La biblioteca Irvine16 es para programas escritos en modo de direccionamiento real de 16 bits. Contiene procedimientos que ejecutan interrupciones de MS-DOS cuando se generan operaciones de entrada-salida.

Antecedentes

Una *biblioteca de enlace* es un archivo que contiene procedimientos (subrutinas) ensamblados en código máquina. Una biblioteca de enlace empieza como uno o más archivos de código de fuente, los cuales se ensamblan en archivos de código objeto. Estos archivos se insertan en un archivo con formato especial, reconocido por la herramienta enlazador. Supongamos que un programa muestra una cadena en la ventana de la consola, llamando a un procedimiento de nombre **EscribirCadena**. El código de fuente del programa debe contener una directiva PROTO que identifique el procedimiento EscribirCadena:

```
EscribirCadena PROTO
```

Después, una instrucción CALL ejecuta a **EscribirCadena**:

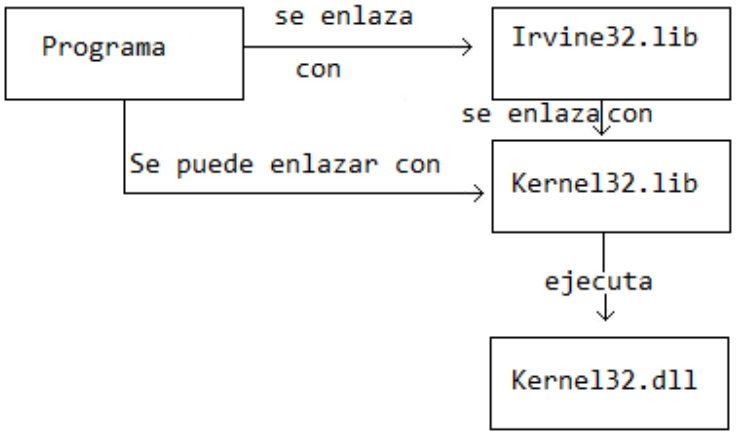
```
Call EscribirCadena
```

Cuando el programa se ensambla, el ensamblador deja la dirección de destina de la instrucción CALL en blanco, sabiendo que el enlazador la llenará. El enlazador busca **EscribirCadena** en la biblioteca de enlace y copia las instrucciones de máquina apropiadas de la biblioteca en el archivo ejecutable del programa. Además, inserta la dirección **EscribirCadena** en la instrucción CALL. Si un procedimiento que llamamos no se encuentra en la biblioteca de enlace, el enlazador genera un mensaje de error y no genera un archivo ejectuable.

Opciones de comandos del enlazador La herramienta enlazador combina el archivo de código objeto de un programa con uno o más archivos de código objeto y bibliotecas de enlace. Por ejemplo, el siguiente comando enlaza hola.obj con la biblioteca **irvine32.lib** y **kernel32.lib**:

```
link hola.obj irvine32.lib kernel32.lib
```

Como enlazar programas de32 bits Vamos a ver con más detalle todo lo relacionado con el proceso de enlazar programas de 32 bits. El archivo **Kernel32.lib**, que forma parte *del Kit de desarrollo de software* de plataforma Microsoft Windows, contiene información de enlace para las funciones del sistema que se encuentra en un archivo llamado **kernel32.dll**. Este archivo es una parte fundamental de MS-Windows, y se le conoce como *biblioteca de vínculos dinámicos*. Contiene funciones ejecutables que se encargan de las operaciones de entrada-salida basadas en caracteres. La siguiente figura muestra como **kernel32.lib** constituye un puente para **kernel32.dll**:



*En los capítulos 1 a 10, nuestros programas se enlazan con Irvine32.lib. El capítulo 11 muestra cómo enlazar programas directamente con kernel32.lib.

La biblioteca de enlace del libro

Generalidades

La siguiente tabla contiene una lista de procedimientos de uso más común en las bibliotecas Irvine32 e Irvine16 que se incluyen en el sitio Web del libro. Aunque la biblioteca Irvine16 es para programas que se ejecutan en modo 16 bits (modo de direccionamiento real), utiliza registros de 32 bits. La mayoría de los procedimientos documentados en esta sección existen en ambas bibliotecas. Los procedimientos que se encuentran sólo en la biblioteca Irvine32 están marcados con un * al final de sus descripciones.

Ventana de consola La *ventana* de *consola* (o *ventana* de *comandos*) es una ventana sólo de texto, que MS-Windows crea cuando se muestra el símbolo de sistema. Para mostrarla, se debe hacer clic en inicio->Ejecutar y escribir **cmd** (para Windows 2000 y Windows XP) o **command** (para Windows 95 y 98). En Windows 2000 y Windows XP se puede cambiar el tamaño del búfer de la ventana de consola, haciendo clic con el botón derecho del mouse en el menú de sistema, en la esquina superior izquierda de la ventana. También se pueden seleccionar varios tamaños y colores de fuentes. En Windows 95 y 98, podemos establecer el número de filas a uno de varios valores predeterminados. En todas las versiones de Windows y MS-DOS, la ventana de consola tiene una configuración predeterminada de 250 filas por 80 columnas. Podemos cambiar el número de líneas mediante el comando **mode**. Si escribimos lo siguiente en el símbolo del sistema, la ventana de consola se establecerá a 40 columnas por 30 líneas

```
mode con cols=40 lines=30
```

Procedimientos en la biblioteca de enlaces.

Procedimiento	Descripción
CloseFile	Cierra un archivo en disco que se había abierto anteriormente*
Clrscr	Borra la ventana de consola y posiciona el cursor en la esquina superior izquierda
CreateOutputFile	Crea un nuevo archivo en el disco, para escribir en modo de salida*
Crlf	Escribe una secuencia de fin de línea en la ventana de consola
Delay	Detiene la ejecución del programa durante un intervalo especificado en *n* milisegundos
DumpMem	Escribe un bloque de memoria a la ventana de consola en hexadecimal
DumpRegs	Muestra los registros EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EFLAGS y EIP en hexadecimal. También muestra la bandera de estado más comunes de la CPU
GetCommandTail	Copia argumentos de línea de comandos del programa (*llamados cola de comandos*) en un arreglo de bytes
GetMaxXY	Obtiene el número de columnas y filas en el búfer de la ventana de consola
GetMseconds	Devuelve el número de milisegundos transcurridos desde medianoche
GetTextColor	Devuelve los colores del texto y del fondo de la ventana de consola*
Gotoxy	Posiciona el cursor en una fila y columna específicas en la ventana de consola
IsDigit	Activa la bandera Cero si el registro AL contiene código ASCII para un dígito decimal (0-9)
MsgBox	Muestra un cuadro de mensaje contextual*

MsgBoxAsk	Muestra una pregunta tipo si/no en un cuadro de mensaje contextual*
OpenInputFile	Abre un archivo existente en disco para entrada*
ParseDecimal32	Convierte una cadena de enteros decimales sin signo a un número binario de 32 bits
ParseInterger32	Convierte una cadena de enteros decimales con singo en un número binario de 32 bits
Random32	Genera un entero seudoaleatorio de 32 bits en el rango 0 a FFFFFFFFh
Randomize	Siembra el generador de números aleatorios con un valor único
RandomRange	Genera un entero seudoaleatorio dentro de un rango específico
ReadChar	Espera que se escriba un carácter desde el teclado y devuelve ese carácter
ReadDec	Lee un entero decimal sin signo de 32 bits del teclado; para terminarlo se oprime Intro
ReadFromFile	Lee un archivo en disco de entrada y lo coloca en un búfer*
ReadHex	Lee un entero hexadecimal de 32 bits desde el teclado; para terminarlo se oprime Intro
ReadInt	Lee un entero decimal con signo de 32 bits desde el teclado; para terminarlo se oprime intro
ReadKey	Lee un carácter del búfer de entrada del teclado, sin esperar la entrada
ReadString	Lee una cadena del teclado, la cual se termina oprimiendo Intro
SetTextColor	Establece los colores de texto y de fondo de toda la salida de texto subsiguiente a la consola
StrLength	Devuelve la longitud de una cadena
WaitMsg	Muestra un mensaje y espera a que se oprima una tecla
WriteBin	Escribe un entero sin signo de 32 bits a la ventana de consola, en formato ASCII binario
WriteBinB	Escribe un entero a la ventana de consola en formato byte, palabra o doble palabra
WriteChar	Escribe un solo carácter a la ventana consola
WriteDec	Escribe un entero sin signo de 32 bits a la ventana de consola, en formato decimal
WriteHex	Escribe un entero de 32 bits a la ventana de consola, en formato hexadecimal
WriteHexB	Escribe un entero tipo byte, palabra o doble palabra a la ventana de consola, en formato hexadecimal
WriteInt	Escribe un entero con signo de 32 bits a la ventana de consola, en formato decimal

WriteString	Escribe una cadena de terminación nula a la ventana de consola
WriteToFile	Escribe un búfer a un archivo de salida*
WriteWindowsMsg	Muestra una cadena que contiene el error más reciente generado por MS-Windows*

Redirección de la entrada-salida estándar

Ambas bibliotecas Irvine32 e Irvine16 escriben su salida a la ventana de consola, pero la biblioteca Irvine16 tiene una característica adicional*: la redirección de la entrada-salido estándar*. Su salida puede redirigirse al símbolo del sistema de DOS o Windows, para escribir a un archivo en disco en vez de hacerlo en la ventana de consola. He aquí cómo funciona: supongamos que un programa llamado *ejemplo.exe* escribe a la salida estándar; entonces, podemos usar el siguiente comando (en el símbolo DOS) para redirigir su salida a un archivo llamado *salida.txt*

```
ejemplo > salida.txt
```

De manera similar, si el mismo programa lee la entrada desde el teclado (entrada estándar), podemos decirle que lea su entrada desde un archivo llamado *entrada.txt*:

```
ejemplo < entrada.txt
```

Podemos redirigir tan la entrada como la salida con un solo comando:

```
ejemplo < entrada.txt > salida.txt
```

Podemos enviar la salida estándar desde *prog1.exe* a la salida estándar de *prog2.exe*, utilizando el símbolo de canalización (|):

```
prog1 | prog2
```

Podemos enviar la salida desde *prog1.exe* a la entrada estándar de *prog2.exe*, y enviar la salida de *prog2*.exe a un archivo llamado *salida.txt*:

```
prog1 | prog2 > salida.txt
```

Prog1.exe puede leer la entrada desde *entrada.txt* y enviar su salida a *prog2.exe*, que a su vez puede enviar su salida a *salida.txt*:

```
prog1 < entrada.txt | prog2 > salida.txt
```

Los nombres de archivo entrada.txt y salida.txt son completamente arbitrarios, por lo que podemos elegir los nombres que deseamos.

Descripciones de los procedimientos individuales

CloseFile (Irvine32 *solamente*) El procedimiento CloseFile cierra un archivo que estaba abierto previamente. El archivo se identifica mediante el *manejador* entero de 32 bits, el cual se pasa en EAX. Si el archivo se cierra con éxito, el valor devuelto en EAX será distinto de cero. He aquí una llamada de ejemplo

```
mov eax,manejadorArchivo

call CloseFile
```

Clrscr El procedimiento Clrscr borra la ventana de consola Por lo general, este procedimiento se le llama al principio y al final de un programa. Si lo llamamos otras veces, tal vez sea necesario detener la ejecución del programa llamando WaitMsg. Esto permite al usuario ver la información que ya se encuentra en la pantalla, antes de que se borre. He aquí una llamada de ejemplo:

```
call WaitMsg            : “Oprima cualquier tecla…”

call Clrscr
```

Crlf El procedimiento Crlf desplaza el cursor al principio de la siguiente línea en la ventana de consola. Escribe una cadena que contiene los valores 0Dh y 0Ah. He aquí una llamada del ejemplo:

```
call Crlf
```

CreateOutputFile El procedimiento CreateOutputFile crea un archivo en disco y lo abre en modo de salida. Pasa el desplazamiento del nombre de un archivo en EDX. Cuando el procedimiento regresa, si el archivo se creó con éxito, EAX contiene un manejador de archivo válido (entero de 32 bits). En caso contrario, EAX es igual a INVALID_HANDLE_VALUE (una constante predefinida). He aquí un llamado de ejemplo:

```
.data

Nombre del archivo BYTE “nuevoarchivo.txt”,0

manejador DWORD ?

.code

mov edx,OFFSET nombre del archivo

call CreateOutpuFile

cmp eax,INVALID_HANDLE_VALUE

je error_archivo                        ; muestra el mensaje de error

mov manejador,eax                       ; guarda el manejador del archivo
```

Nota: El código de ejemplo anterior compara el valor en EAX con una constante predefinida. Si son iguales la instrucción JE salta a una etiqueta llamada **error_archivo**. En el capítulo 6 hablaremos sobre las instrucciones CMP y JE. Proporcionamos este código de manejo de errores para su futura referencia.

Delay El procedimiento Delay detiene la ejecución del programa durante cierto número de milisegundos Antes de llamar a Delay, asigne a EAX el intervalo deseado, He aquí una llamade de ejemplo:

```
mov eax,1000    ; 1 segundo

call Delay
```

(La versión Irvine16.lib no funciona bajo Windows NT, 2000 o XP).

DumpMem El procedimiento DumpMem escribe un rango de memoria a la ventana de consola en hexadecimal. Se le pasa la dirección inicial en ESI, el número de unidades en ECX y el tamaño de la unidad en EBX (1 = byte, 2 = palabra, 4 = doble palabra). La siguiente llamada de ejemplo muestra un arreglo de 11 dobles palabras en hexadecimal:

```
.data

arreglo DWORD 1,2,3,4,5,6,7,8,9,0,0Ah,0Bh

.code

main PROC

      mov esi,OFFSET arreglo                  ; desplazamiento inicial

      mov ecx,  LENGTHOF arreglo              ; número de unidades

      mov ebx, TYPE arreglo                   ; format de doble palabra

      call DumpMem
```

Se producirá la siguiente salida:

```
00000001    00000002    00000003    00000004    00000005    00000006

00000007    00000008    00000009    00000000    0000000A    0000000B
```

DumpRegs El procedimiento DumpRegs muestra los registros EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP y EFL (EFLAGS) en hexadecimal. También muestra los valores de banderas Acarreo, Signo, Cero, Desbordamiento, Acarreo auxiliar y Paridad. He aquí una llamada de ejemplo:

```
call DumpRegs
```

Resultado de ejemplo:

```
EAX=00000613    EBX=00000000    ECX=000000FF    EDX=00000000
ESI=00000000    EDI=00000100    EBP=0000091E    ESP=000000F6
EIP=00401026    EFL=00000286    CF=0  SF=1  ZF=0  OF=0  AF=0  PF=1
```

El valor mostrado de EIP es el desplazamiento de la instrucción que va después de la llamada a DumpRegs. Este procedimiento puede ser útil al depurar programas, ya que muestra una instantánea de la CPU. No tiene parámetros de entrada ni valor de retorno.

GetCommandTail El procedimiento GetCommanTail copia la línea de comandos del programa en una cadena con terminación nula. Si la línea de comandos se encontró vacía, se activa la bandera de Acarreo en caso contrario, se borra. Este procedimiento es útil ya que permite al usuario de un programa pasar información a la línea de comandos. Supongamos que un programa llamado **Cifrar** lee un archivo de entrada llamado **Archivo1.txt** y produce un archivo de salida llamado **Archivo2.txt**. El usuario puede pasar ambos nombres en una línea de comandos cuando ejecuta el programa:

```
Cifrar archivo1.txt a archivo2.txt
```

Al iniciar, el programa Cifrar se puede llamar a GetCommandTail y obtener los dos nombres de archivo. Al llamar a GetCommandTail, EDX debe contener el desplazamiento de un arreglo de por lo menos 129 bytes. He aquí una llamada de ejemplo:

```
.data
colaComandos BYTE 129 DUP(0)     ; vacía el búfer
.code
mov edx,OFFSET colocaComandos
call GetCommandTail              ; llena el búfer
```

GetMaxXY (*Irvine 32 solamente*) El procedimiento GetMaxXY devuelve el tamaño del búfer de la ventana de consola. Si el búfer es más grande que el tamaño visible de la ventana, aparecerá barras de desplazamiento de manera automática. GetMaxXY no tiene parámetros de entrada. Cuando regresa, el registro DL contiene el número de columnas del búfer y DH contiene el número de filas. El posible rango de cada valor no puede ser mayor de 255, lo cual podría ser menor que el tamaño actual del búfer de la ventana. He aquí una llamada de ejemplo:

```
.data
filas BYTE ?
cols  BYTE ?
.code
```

```
call GetMaxXY

mov filas,dh

mov cols,dl
```

GetMseconds El procedimiento GetMseconds devuelve el número de segundos transcurridos desde medianoche en el registro EAX. Podemos usarlo para medir el tiempo entre un evento y otro. No se requieren parámetros de entrada. El siguiente ejemplo llama a GetMseconds y almacena su valor de retorno. Después de ejecutar el ciclo, llamamos a GetMseconds una segunda vez y restamos los dos valores de tiempo. La diferencia es la duración aproximada del ciclo:

```
.data

TiempoInicio DWORD ?

.code

call GetMseconds

mov tiempoInicio,eax

L1:

; (cuerpo del ciclo)

loop L1

call GetMseconds

sub eax,tiempoInicio    ; EAX = tiempo del ciclo, en milisegundos
```

GetTextColor El procedimiento GetTextColor devuelve los colores de texto y de fondo de la ventana de consola (*Irvine 32 solamente*). No tiene parámetros de entrada. Devuelve el color de fondo en los cuatro bits superiores de AL y el color de texto en cuatro bits inferiores. He aquí una llamada de ejemplo:

```
.data

color BYTE ?

.code

call GetTextColor

mov color,AL
```

Gotoxy El procedimiento Gotoxy posiciona el cursor en una fila y columna especificadas en la pantalla. De manera predeterminada, el rango de coordenadas X de la ventana de consola es

de 0 a 79, y el rango de coordenadas Y es de 0 a 24. Al llamar a Gotoxy, se debe pasar la coordenada Y (fila) en DH y la coordenada X (columna) en DL, He aquí una llamada de ejemplo:

```
mov dh,10    ; fila 10

mov dl,20    ; columna 20

call Gotxy   ;posiciona el cursor
```

En caso de que el usuario haya cambiado el tamaño de la ventana de consola, podemos llamar a GetMaxXY para encontrar el número actual de filas y columnas.

IsDigit El procedimiento IsDigit determina si el carácter en AL, es un dígito decimal válido. Al llamarlo se pasa un carácter ASCII en AL. El procedimiento activa la bandera Cero si AL contiene un dígito decimal válido, en caso contrario, la bandera Cero se borra. He aquí una llamada de ejemplo:

```
mov AL,uncaracter

call IsDigit

jz digito_encontrado
```

La instrucción JZ, que veremos en la sección 6, salta a una etiqueta cuando se activa la bandera Cero.

MsgBox (*Irvine32 solamente*) El procedimiento MsgBox muestra un cuadro de mensaje contextual gráfico con una leyenda opcional. Se le pasa el desplazamiento de una cadena en EDX, la cual aparece dentro del cuadro. De manera opcional, se le pasa el desplazamiento de una cadena en EBX para el título del cuadro. Para dejar el título en blanco. EBX se establece en cero. He aquí una llamada de ejemplo:

```
.data

leyenda db “Titulo del cuadro dialogo”,0

MsgHola BYTE “Este es un cuadro de mensaje contextual.”,0dh,0ah

                BYTE “Haga clic en Aceptar para continuar…”,0

.code

mov ebx,OFFSET leyenda

mov edx,OFFSET Msjhola

call MsgBox
```

MsgBoxAsk (*Irvine 32 solamente*) El procedimiento MsgBoxAsk muestra un cuadro de mensaje contextual gráfico con botones Sí y No. Se le pasa el desplazamiento de una cadena

de pregunta en EDX, la cual aparece dentro del cuadro. De manera opcional, se le pasa el desplazamiento de una cadena en EBX para el título del cuadro. Para dejar el título en blanco, EBX se establece con cero. MsgBoxAsk devuelve un entero en EAX que nos indica qué botón seleccionó el usuario IDYES (igual a 6) o IDNO (igual a 7). He aquí una llamada de ejemplo:

```
.data

leyenda   BYTE “Encuesta terminada”,0

pregunta  BYTE “Gracias pos completar la encuesta.”

          BYTE 0dh,0ah

          BYTE “Desea recibir los resultados?”,0

resultado BYTE “Los resultados se enviarán vía correo
electrónico.”,0dh,0ah,0

.code

mov ebx,OFFSET leyenda

mov edx,OFFSET pregunta

call MsgBoxAsk

;(comprobar el valor de retorno en EAX)
```

OpenInputFile (*Irvine 32 solamente*) El procedimiento OpenInputFile abre un archivo existente en modo de entrada. Se le pasa el desplazamiento de un nombre de archivo en EDX. Al regresar, si el archivo se abrió con éxito, EAX contiene un manejador de archivo válido. En caso contrario, EAX es igual a INVALID_HANDLE_VALUE (una constante predefinida). He aquí una llamada de ejemplo:

```
.data

nombrearchivo BYTE “miarchivo.txt”,0

manejador     DWORD ?

.code

mov edx,OFFSET nombrearchivo

call OpenInputFile

cmp eax,INVALID_HANDLE_VALUE

je archivo_error                        ; muestra el mensaje de error

mov manejador,eax                       ; guarda el manejador del archivo
```

Nota: el código de ejemplo anterior compara el valor en EAX con una constante predefinida. Si son iguales, la instrucción JE salta a una etiqueta llamada archivo_error. En el capítulo 6 hablaremos sobre instrucciones CMP y JE. Proporcionamos este código de manejo de errores para una referencia a futuro.

ParseDecimal32 El procedimiento ParseDecimal32 convierte una cadena de enteros decimales sin signo en un número binario de 32 bits. Todos los dígitos válidos que ocurren antes de un carácter no numérico se convierten: los espacios en blanco a la izquierda se ignoran. Se le pasa el desplazamiento de una cadena en EDX y la longitud de la cadena en ECX: el valor binario se devuelve en EAX. He aquí una llamada de ejemplo:

```
.data

búfer BYTE “8193”

tamBufer = ($ - búfer)

.code

mov edx,OFFSET buffer

mov ecx,tamBufer

call ParseDecimal32        ; devuelve EAX
```

Se debe consultar la descripción del procedimiento ReadDec para ver los detalles acerca de cómo se ve afectada la bandera e Acarreo.

PerserInteger32 El procedimiento ParseInteger32 convierte una cadena de enteros decimales con signo en un número binario de 32 bits. Todos los datos válidos ocurren antes de un carácter no numérico se convierten. Todos los dígitos válidos que ocurren antes de un carácter no numérico se convierten: Los espacios en blanco a la izquierda se ignoran. Se le pasa el desplazamiento de una cadena en EDX y la longitud de la cadena en ECX: el valor binario se devuelve en EAX. He aquí una llamada de ejemplo:

```
.data

búfer BYTE “-8193”

tambufer = ($ - búfer)

.code

mov edx,OFFSET búfer

mov ecx,tamBufer

call parseInteger32         ; devuelve EAX
```

La cadena puede contener un signo positivo o negativo opcional a la izquierda, seguido sólo de dígitos decimales. La bandera de Desbordamiento se activa y se muestra un mensaje en la consola si el valor no puede representarse como entero con signo de 32 bits (rango de -2,147,483,648 a +2,147,483,647).

Random32 El procedimiento Random32 genera y devuelve un entero aleatorio de 32 bits en EAX. Si se llama repetidas veces, Random32 genera una secuencia aleatoria simulada, en la que cada número se conoce como *entero seudoaleatorio*. Los números se crean utilizando una función simple, que tiene una entrada conocida como *semilla*. Esta función utiliza la semilla en una fórmula que genera el valor aleatorio.

Los valores aleatorios subsiguientes se generan utilizando cada valor aleatorio generado anteriormente como sus semillas. De este punto en adelante, el término *aleatorio* significará seudoaleatorio . He aquí una llamada de ejemplo:

```
.data

ValAleatorio DWORD ?

.code

call Random32

mov valAleatorio,eax
```

El procedimiento Random32 también está disponible en la biblioteca Irvine16, y devuelve su valor en EAX.

Randomize El procedimiento Randomize inicializa el valor de la semilla inicial de los procedimientos Random32 y RandomRange. La semilla es igual a la hora del día, con una precisión de 1/100 de un segundo. Cada vez que se ejecute un programa que llama a Random32 y a RandomRange, la secuencia generada será distinta y cualquier secuencia de números aleatorios también será única. Solo necesitamos llamar a Randomiza una vez al principio de un programa. En el siguiente ejemplo producimos 10 enteros aleatorios:

```
call Randomize

mov ecx,10

L1: call Random32

   ; aquí se utiliza o se muestra el valor aleatorio en EAX…

   loop L1
```

RandomRange El procedimiento RandomRange produce un entero aleatorio dentro de un rango de 0 a *n* – 1, en donde *n* es un parámetro de entrada que se pasa en el registro EAX. El entero aleatorio se devuelve en EAX. El siguiente ejemplo genera un entero aleatorio individual entre 0 y 4999, y lo coloca en EAX:

```
.data valAleat DWORD ?

.code

mov eax,5000

call RandomRange

mov valAleat,eax
```

ReadChar El procedimiento ReadChar lee un solo carácter del teclado y devuelve ese carácter en el registro AL. El carácter no se imprime en la ventana de consola. He aquí una llamada del ejemplo:

```
.data

car BYTE ?

.code

call ReadChar

mov car,al
```

Si el usuario oprime una tecla extendida como una tecla de función, la flecha de cursor, Insert o Supr, el procedimiento establece AL en cero y AH contiene un código de exploración de teclado.

ReadDec El procedimiento RecDect lee un entero decimal sin signo de 32 bits del teclado y devuelve el valor en EAX. Los espacios a la izquierda se ignoran. El valor de retorno se calcula en base a todos los dígitos válidos que aparezcan, hasta encontrarse un carácter que no sea dígito. Por ejemplo, si el usuario escribe 123ABC, el valor devuelto en EAX es 123. He aquí una llamada de ejemplo:

```
.data

valEntero DWORD ?

.code

call ReadDec

mov valEntero,eax
```

ReadDec afecta a la bandera de Acarreo en lo siguiente:

- Si el entero está en blanco, EAX = 0 y CF = 1.
- Si el entero sólo contiene espacios EAX = 0 Y CF = 1.
- Si el entero es mayor que 2^32-1, EAX = 0 y CF = 1.
- En caso contrario, EAX = el entero convertido y CF = 0.

ReadFromFile (*Irvine 32 solamente*) El procedimiento ReadFromFile lee un archivo de entrada y lo coloca en un búfer. Recibe un manejador de archivo abierto en EAX, el desplazamiento de un búfer en EDX y el número máximo de bytes a leer en ECX. Cuando el procedimiento regresa, si CF = 0, EAX contiene la cuenta del número de bytes que se leyeron del archivo. Si CF = 1, EAX contiene el código de error del sistema, que explica lo que salió mal. (Podemos llamar a WriteWindowsMsg para obtener una representación de texto del mensaje). He aquí una llamada de ejemplo:

```
.data

TAM_BÚFER = 5000

.data

búfer BYTE TAM_BÚFER DUP (?)

bytesLeidos DWORD ?

.code

mov edx,OFFSET buffer          ; apunta al buffer

mov ecx,TAM_BÚFER              ; máximo de bytes a leer

call readFromFile              ; lee el archivo

jc muestra_mensaje_error       ; ocurrió un error

mov bytesLeidos,eax            ; cuenta los bytes que se leyeron
```

ReadHex El procedimiento ReadHex lee un entero hexadecimal de 32 bits desde el teclado. Y devuelve el valor en EAX. No se realiza una comprobación de errores para caracteres inválidos. Se pueden utilizar letras tanto mayúsculas como minúsculas para los dígitos A-F. Puede introducirse un máximo de ocho dígitos (los caracteres adicionales se ignoran). Los espacios a la izquierda se ignoran. He aquí una llamada de ejemplo:

```
.data

valHex DWORD ?

.code

call ReadHex

mov valHex,eax
```

ReadInt El procedimiento ReadInt lee un entero con signo de 32 bits desde el teclado, y devuelve el valor en EAX. El usuario puede escribir un signo positivo o negativo opcional a la izquierda, y el resto del número sólo puede consistir de dígitos. ReadInt Activa la bandera Desbordamiento y muestra un mensaje de error si el valor introducido no puede representarse

como entero con signo de 32 bits (rango: -2,147,483,648 a +2,147,483,647). El valor de retorno se calcula a partir de todos los dígitos válidos encontrados, hasta que se encuentra un carácter que no sea dígito. Por ejemplo, si el usuario escribe +123ABC, el valor devuelto es +123. He aquí una llamada de ejemplo:

```
.data

valEntero SWORD ?

.code

call ReadInt

mov ValEntero,eax
```

ReadKey El procedimiento ReadKey realiza una comprobación del teclado sin espera. Si no se encuentra una tecla, se activa la bandera Cero. Si se encuentra una tecla, se borra la bandera Cero y AL contiene ya sea cero o un código ASCII. Si AL contiene cero, tal vez el usuario oprimió una tecla especial (tecla de función, flecha de cursor, etc.). El registro AH contiene un código de exploración virtual, DX contiene un código de tecla virtual y EBX contiene los bits de bandera del teclado. Las mitades superiores de EAX Y EDX se sobrescriben. *En el capítulo 11 veremos con más detalle el funcionamiento de ReadKey*. He aquí una llamada de ejemplo, cuando se utiliza la biblioteca Irvine32 y el usuario oprime una tecla alfanumérica estándar:

```
.data

car BYTE ?

.code

L1: mov eax,10     ; crea un retraso de 10 ms

call Delay

call ReadKey       ; comprueba la tecla

JZ L1              ; repite si no hay tecla

mov car,AL         ; guarda el carácter
```

*Se ha agregado un retraso de 10 milisegundos al ciclo, para dar tiempo a que MS-Windows procese los mensajes de eventos. En caso contrario, podrían perderse algunos tecleos. Si se utiliza la biblioteca Irvine16, se puede omitir el retraso:

```
.data

car BYTE ?

.code
```

```
L1: call ReadKey      ; comprueba la tecla
      jz L1           ; repite si no hay tecla
      mov car,AL      ; guarda el carácter
```

ReadString El procedimiento ReadString lee una cadena del teclado, y se detiene cuando el usuario oprime Intro. Recibe el desplazamiento de un búfer en EDX y establece ECX al máximo número de caracteres que puede introducir un usuario, más 1 (para guardar espacio para el byte de terminación nulo). Procedimiento devuelve la cuenta del número de caracteres escritos por el usuario en EAX. He aquí una llamada de ejemplo:

```
.data
búfer BYTE 21 DUP(0)       ; búfer de entrada
cuentaBytes DWORD ?        ; guarda el contador
.code
mov edx,OFFSET búfer       ; apunta al búfer
mov ecx,SIZEOF búfer       ; especifica el máximo de caracteres
call ReadString            ; recibe la cadena de entrada
mov cuentaBytes,eax        ; números de caracteres
```

ReadString inserta en forma automática un terminador nulo en memoria, al final de la cadena. A continuación se muestra un vaciado hexadecimal y ASCII de los primeros 8 bytes de **búfer**, después que el usuario introduce la cadena "ABCDEFG":

41 42 43 44 45 46 47 00	ABCDEFG

La variable **cuentaBytes** es igual a 7.

SetTextColor El procedimiento SetTextColor (*biblioteca Irvine32 solamente*) establece los colores de texto y de fondo para la salida de texto. Al llamar a SetTextColor, hay que asignar un atributo de color a AX. Pueden utilizarse las siguientes constantes de colores predefinidas, tanto para el texto como para el fondo:

Black = 0	Red = 4	Gray = 8	LightRed = 12
Blue = 1	Magenta = 5	lightBlue = 9	LightMagenta = 13
Green = 2	Brown = 6	lightGreen = 10	Yellow = 14

Cyan = 3	lightGray = 7	LightCyan = 11	White = 15

Las constantes de color se definen en los archivos de inclusión llamados *Irvine32.inc* e *Irvine16.inc*

Multiplicamos el color de fondo por 16 y le sumamos al color de texto. Por ejemplo, la siguiente constante indica caracteres amarillos en un fondo azul:

```
yellow + (blue = 16)
```

La siguiente instrucción establece el color blanco, en un fondo azul:

```
mov eax,White + (blue * 16)     ; blanco sobre azul
```

En la sección 15 se encontrará una explicación detallada de los atributos de video. La versión de SetTextColor en la biblioteca Irvine16 borra la ventana de consola con los colores seleccionados.

StrLength El procedimiento StrLength devuelve la longitud de una cadena con terminación nula. Recibe el desplazamiento de la cadena en EDX. El procedimiento devuelve la longitud de la cadena en EAX. He aquí una llamada de ejemplo:

```
.data

búfer BYTE “abcde”,0

longBufer DWORD ?

.code

mov edx,OFFSET buffer          ; apunta a la cadena

call StrLength                 ; EAX = 5

mov longBufer,eax              ; guarda la longitud
```

WaitMsg El procedimiento WaitMsg muestra el mensaje “Press any key to continue…” y espera a que el usuario oprima una tecla. Este procedimiento es útil cuando deseamos detener la visualización de la pantalla antes de que se desplacen los datos y desaparezcan. No tiene parámetro de entrada. He aquí una llamada de ejemplo:

```
call WaitMsg
```

WriteBin el procedimiento WriteBin escribe un entero en la ventana de la consola, en formato ASCII binario. El entero se pasa en EAX. Los bits binarios se muestran en grupos de cuatro, para facilitar su legibilidad. He aquí una llamada de ejemplo:

```
mov eax,12346AF9h
```

```
call WriteBin

; muestra: “0001 0010 0011 0100 0110 1010 1111 1001”
```

WriteBinB El procedimiento WriteBinB escribe un entero de 32 bits en la ventana de consola, en formato ASCII binario. El valor pasa en el registro EAX y deja que EBX indique el tamaño de visualización en bytes (1,2 o 4). Los bits se muestran en grupos de a cuatro, para facilitar su legibilidad. He aquí una llamada de ejemplo:

```
mov eax,00001234h

mov ebx,TYPE WORD              ; 2 bytes

call WriteBinB                 ; muestra 0001 0010 0010 0100
```

WriteChar El procedimiento WriteChar escribe un solo carácter en la ventana de consola. Pasa el carácter (o su código ASCII) en AL. He aquí una llamada de ejemplo:

```
mov al,’A’

call WriteChar                  ; muestra “A”
```

WriteDec El procedimiento WriteDec escribe un entero sin signo de 32 bits en la ventana de la consola, en formato decimal sin ceros a la izquierda. El entero se pasa en EAX. He aquí una llamada de ejemplo:

```
mov eax,295

call WriteDec                   ; muestra “295”
```

WriteHex El procedimiento WriteHex escribe un entero sin signo de 32 bits en la ventana de consola. En formato hexadecimal de 8 dígitos. Si es necesario, pueden insertarse ceros a la izquierda. El entero se pasa en EAX. He aquí un llamado de ejemplo:

```
mov eax,7FFFh

call WriteHex                   ; muestra: “00007FFF”
```

WriteHexB El procedimiento WriteHexB escribe un entero sin signo de 32 bits en la ventana de consola en formato hexadecimal. Si es necesario, se insertan ceros a la izquierda. El entero se pasa en EAX y deja que EBX indique el formato de visualización en bytes (1, 2 o 4). He aquí una llamada de ejemplo:

```
mov eax,7FFFh

mov ebx,TYPE WORD               ; 2 bytes

call WriteHexB                  ; muestra: “7FFF”
```

WriteInt El procedimiento WriteInt escribe un entero con signo de 32 bits en la ventana de consola, en formato decimal con un signo a la izquierda y sin ceros a la izquierda. El entero se pasa en EAX. He aquí una llamada de ejemplo:

```
mov eax,216543

call WriteInt                    ; muestra: "+216543"
```

WriteString El procedimiento WriteString escribe una cadena con terminación nula en la ventana de consola. El desplazamiento de la cadena se pasa en EDX. He aquí una llamada de ejemplo:

```
.data

indicador BYTE "Escriba su nombre: ",0

.code

mov edc,OFFSET indicador

call WriteString
```

WriteToFile (*Irvine 32 solamente*) El procedimiento WriteToFile escribe el contenido de un búfer en un archivo de salida. Se le pasa un manejador de archivos válidos en EAX, el desplazamiento del búfer en EDX, y el número de bytes a escribir en ECX. Cuando el procedimiento regresa, EAX contiene una cuenta del número de bytes escritos. He aquí una llamada de ejemplo:

```
TAM_BÚFER = 5000

.data

manejadorArchivo DWORD ?

búfer BYTE TAM_BÚFER DUP(?)

bytesEscritos DWORD ?

.code

mov eax,manejadorArchivo

mov edx,OFFSET búfer

mov ecx,TAM_BÚFER

call WriteToFile

mov byteEscritos,eax          ; guarda el valor de retorno
```

WriteWindowsMsg (*Irvine 32 solamente*) El procedimiento WriteWindosMsg muestra una cadena que contiene el error más reciente generado por MS-Windows. Su mayor utilidad es cuando un programa no puede crear o abrir un archivo. El siguiente ejemplo trata de abrir un archivo para entrada, y al encontrarse con un error llama a WriteWindowsMsg para mostrar ese error:

```
mov edx,OFFSET nombrearchivo

call OpenInputFile

.IF eax == INVALID_HANDLE_VALUE

call WriteWindowsMsg

.ENDIF
```

La siguiente cadena se escribe en la ventana de la consola:

Error 2: El sistema no puede hallar el archivo especificado.

Operaciones de la pila

Si colocamos 10 panqueques, uno encima del otro en el siguiente diagrama, al resultado se le puede llamar *pila*. Por lo general, no sacamos un panqueque de en medio de la pila; quitamos un panqueque de la parte superior de la pila para colocarlo en nuestro plato. Pueden agregarse más panqueques en la parte superior de la pila, pero no abajo ni en medio

Pila de panqueques:

4 (Superior)
3
2
1(Inferior)

Los panqueques tienen algo en común con los programas de computadora. A una pila también se le conoce como estructura UEPS (*Última en entrar, primera en salir*), en infles LIFO (Last-in,

First-Out), ya que el último valor que se coloca en la pila siempre es el primero que se saca (UEPS es un término contable muy conocido)

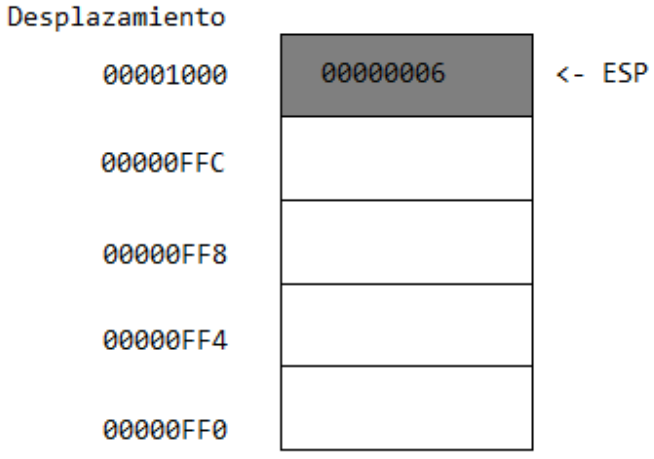
Una *estructura de datos tipo pila* sigue el mismo principio: se agregan nuevos valores a la parte superior de la pila y los valores existentes se quitan de la parte superior. En general, las pilas son estructuras útiles para una variedad de aplicaciones de programación, y pueden implementarse con facilidad mediante el uso de métodos de programación orientada a objetos.

En este capítulo nos concentraremos en lo que se conoce como la *pila en tiempo de ejecución*. El hardware en la CPU la soporta directamente, y es una parte esencial del mecanismo para llamar a los procedimientos y regresar de ellos. La mayor parte del tiempo, la llamaremos simplemente *pila*.

Pila en tiempo de ejecución

La pila *en tiempo de ejecución* es un arreglo de memoria que la CPU administra directamente, mediante el uso de dos registros: SS y ESP. En modo protegido, el registro SS guarda el apuntador a un descriptor de segmento y no se modifica en los programas de usuario. El registro ESP guarda un desplazamiento de 32 bits hacia alguna ubicación en la pila. Muy raras veces es necesario manipular el registro ESP en forma directa; en vez de ello, se modifica de manera indirecta mediante las instrucciones tales como CALL, RET. PUSH y POP.

El registro apuntador de pila (ESP) apunta al último entero que se va a agregar (o *meter*) a la pila. Para demostrar esto, vamos a empezar con una pila que contiene un solo valor. En la siguiente ilustración, el registro ESP (apuntador de pila extendido) contiene el número decimal 00001000, el desplazamiento del valor más reciente se metió a la pila (00000006):



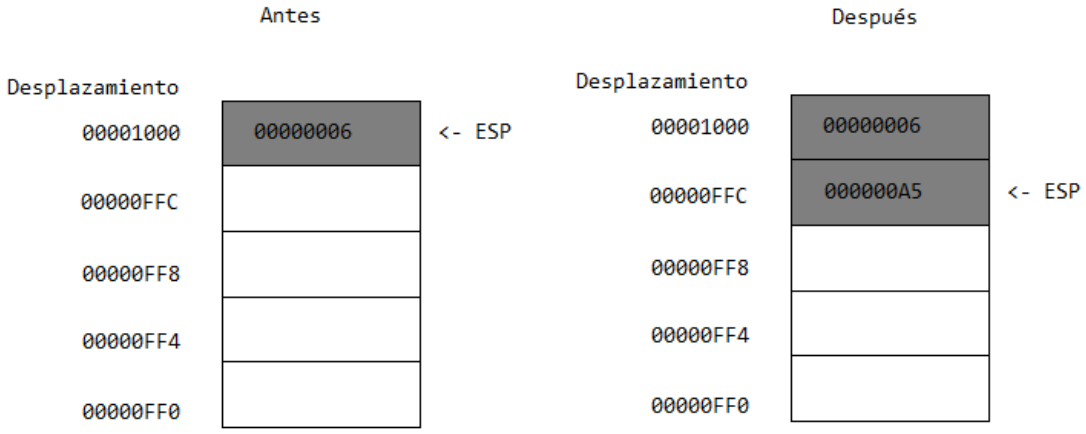
Cada posición de la en esta figura contiene 32 bits, que es el caso cuando el programa se ejecuta en modo protegido. En el modo direccionamiento real de 16 bits, el registro SP apunta al valor más reciente que se metió a la pila, y las entradas de la pila son, por lo general, de 16 bits de longitud.

La pila en tiempo de ejecución que describimos aquí no es la misma que el *tipo de datos abstracto* (ADT) *pila* del que hablamos en el curso sobre estructuras de datos. La pila en tiempo de ejecución trabaja a nivel del sistema para manejar las llamadas a las subrutinas. El ADT pila es una expresión de programación que, por lo general, se escribe en un lenguaje de programación de alto nivel como C++ o java. Se utiliza cuando se implementan algoritmos que dependen de operaciones tipo "último en entrar, primero en salir".

Operación Push (meter)

Una operación *push* de 32 bits decrementa el apuntador de la pila por 4 y copia un valor a la ubicación en la pila a la que apunta el apuntador. En la siguiente figura meteremos el valor 000000A5 en la pila. La figura muestra el orden de la pila opuesto al de la pila de panqueques que vimos antes. La pila en tiempo de ejecución siempre crece hacia abajo en la memoria, siguiendo el principio de "ultimo en entrar, primero en salir". Antes de la operación push, ESP = 00001000h; después de push, ESP = 00000FFCh.

Meter enteros en la pila.

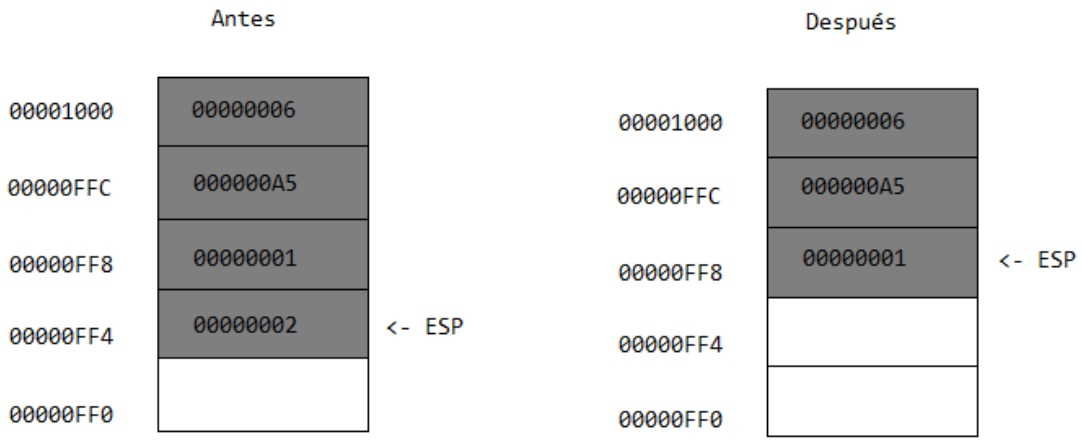


La misma pila luego de meter dos enteros más (00000001 y 00000002).

Operación POP (sacar)

Una operación *pop* elimina un valor de la pila y lo copia a un registro o ubicación de memoria. Después de sacar el valor de la pila, el apuntador de la pila se incrementa para l siguiente ubicación más alta en la pila.

Sacar un valor de la pila en tiempo de ejecución (Antes y después de sacar el valor 00000002).



El área de la pila debajo de ESP está lógicamente vacía, y se sobrescribirá la próxima vez que el programe actual ejecute cualquier instrucción para meter un valor en la pila.

Aplicaciones de las pilas

Hay varios usos importantes para las pilas en tiempo de ejecución en los programas:

- Una pila es un área de almacenamiento temporal conveniente para los registros, cuando se utilizan para más de un propósito. Después de modificarlos, pueden restaurarse a sus valores originales.
- Cuando se ejecuta la instrucción CALL, la CPU almacena la dirección de retorno del procedimiento actual en la pila.
- Al llamar a un procedimiento, es común que se le pasen valores de entrada llamadas *argumentos*, los cuales se meten en la pila.
- La pila proporciona un área de almacenamiento temporal para las variables locales, dentro de los procedimientos.

Instrucciones PUSH y POP

Instrucción PUSH

La instrucción PUSH primero decrementa a ESP y después copia un operando de origen de 16 o 32 bits en la pila. Un operando de 16 bits hace que ESP se decremente por 2. Un operando de 32 bits hace que ESP se decremente por 4. Hay tres formatos para la instrucción:

PUSH r/m16

PUSH r/m32

PUSH imm32

> Si el programa llama a los procedimientos de la biblioteca Irvine32, siempre se debe meter valores de 32 bits; si no es así, las funciones de consola Win32 utilizadas por la biblioteca no funcionarán correctamente. Si el programa llama a procedimientos de la biblioteca Irvine16 (modo direccionamiento real), se pueden meter valores de 16 o 32 bits.

Instrucción POP

La instrucción POP primero copia el contenido del entero de la pila al que apunta ESP, en un operando de destino de 16 o 32 bits, y después incremente ESP. Si el operando es de 16 bits, ESP se incrementa por 2; si el operando es de 32 bits, ESP se incrementa por 4;

POP r/m16

POP r/m32

Instrucciones PUSHFD y POPFD

La instrucción PUSHFD mete el registro EFLAGS de 32 bits en la pila, y POPFD saca el valor de la pila y lo mete en EFLAGS:

pushfd

popfd

> Los programas de 16 bits utilizan la instrucción PUSHF para meter el registro FLAGS de 16 bits en la pila, y POPF para sacar un valor de la pila y meterlo en FLAGS.

La instrucción MOV no puede usarse para copiar las banderas a una variable, por lo que PUSHFD puede ser la mejor forma de guardar las banderas. Hay veces que es útil realizar una copida de respaldo de las banderas, para poder restaurarlas más adelante a los valores que tenían. A menudo, encerramos un bloque de código dentro de PUSHFD y POPFD:

```
pushfd                          ;  Guarda banderas

;

;  Aquí va cualquier secuencia de instrucciones…

popfd                           ; Restaura las banderas
```

Al utilizar instrucciones de este tipo, hay que asegurarnos de que la ruta de ejecución del programa no ignore a la instrucción POPFD. Cuando se modifica un programa después de cierto tempo, puede ser difícil recordar conde se encuentran toda las instrucciones que meten y sacan del stack. ¡Es imprescindible tener una documentación precisa!

Una manera menos propensa a errores de guardar y restaurar las banderas es meterlas en la pila, he inmediatamente después de sacarlas y colocarlas en una variable:

```
.data

guardabanderas DWORD ¿

.code

pushfd                          ; mete las banderas en la pila

pop guardabanderas              ; las copia en una variable
```

Las siguientes instrucciones restauran las banderas desde la misma variable:

```
push guardabanderas             ; mete los valores guardados de las banderas

popfd                           ; las copia a las banderas
```

PUSHAD, PUSHA, POPAD y POPA

La Instrucción PUSHAD mete todos los registros de propósito general de 32 bits en la pila, en el siguiente orden: EAX, ECX, EDX, EBX, ESP (Su valor antes de ejecutar PUSHAD), EBP, ESO y EDI. La instrucción POPAD saca los mismos registros de la pila, en orden inverso. De manera similar, la instrucción PUSHA que se introdujo con el procesador 8026, mete los registro de propósito general de 16 bits (AX, CX, BX, SP, BP, SI, DI) en la pila, en el orden listado. La instrucción POPA saca los mismos registros en orden inverso.

Si se escribe un procedimiento para modificar varios registros de 32 bits, se debe usar PUSHAD al principio del procedimiento y POPAD al final para guardar y restaurar los registros. El siguiente fragmento de código es un ejemplo:

```
MiSub PROC
      pushad        ; guarda los registros de propósito general
      .
      .
      mov eax,...
      mov edx,…
      mov ecx,…
      .
      .
      popad                    ; restaura los registros de propósito
general
       ret
MiSub ENDP
```

Debemos recalcar una importante excepción al ejemplo anterior: los procedimientos que devuelven resultados en uno o más registros no deben utilizar PUSHA y PUSHAD. Supongamos que el siguiente procedimiento **LeerValor** devuelve un entero en EAX; la llamada a POPAD sobrescribe el valor de retorno de EAX:

```
LeerValor PROC
          pushad           ; guarda los registros de propósito general
          .
          .
          mov eax, valor_retorno
          .
          .
          popad            ; sobrescribe EAX
          ret
```

```
LeerValor ENDP
```

Ejemplo: Invertir una cadena

El programa InvCad.asm itera a través de una cadena y mete cada uno de sus caracteres en la pila. Después saca las letras de la pila (en orden inverso) y las almacena de vuelta en la misma variable de cadena. Como la pila es una estructura UEOS (*último en entrar, primero en salir*), se invierten las letras en la cadena:

```
TITLE Invertir una cadena           (InvCad.asm)

; Este programa invierte una cadena

; última actualización: 06/01/2006

INCLUDE Irvine32.inc

.data

unNombre BYTE “Abraham Lincoln”,0

tamanioNombre = ($ - unNombre) -1

.code

main PROC

; Mete el nombre en la pila.

     mov ecx,tamanioNombre

     mov esi,0

L1: movzx eax,unNombre[esi]     ; obtiene el carácter

     push eax                   ; lo mete en la pila

     inc esi

     loop L1

; Saca el nombre de la pila, en orden inverso

; y lo almacena en el arreglo unNombre.

     mov ecx,tamanioNombre

     mov esi,0

L2: pop eax
```

```
     mov unNombre[ESI],al
     inc esi
     loop L2
; Muestra el nombre
     mov edx,OFFSET unNombre
     call Writestring
     call Crlf
     exit
main ENDP
END main
```

Definición y uso de los procedimientos

Si ya se ha estudiado un lenguaje de programación de alto nivel, entonces se sabe lo útil que puede ser dividir los programas en *subrutinas*. Por lo general, un problema complicado se divide en tareas separadas para poder comprenderlo, implementarlo y probarlo con efectividad. En el lenguaje ensamblador, por lo regular, usamos el término de *procedimiento* para indicar una subrutina. En otros lenguajes, a las subrutinas se les llama métodos o funciones.

En términos de programación orientada a objetos, las funciones o métodos de una clase individual son apenas equivalentes a la colección de procedimientos y datos encapsulados en un módulo en lenguaje ensamblador. Este lenguaje se creó mucho antes de la programación orientada a objetos, por lo que tiene la estructura formal que se encuentra en los lenguajes orientados a objetos. Los programadores de ensamblador deben imponer su propia estructura formal en los programas.

Directiva PROC

Definición de un procedimiento

De manera informal, podemos definir un *procedimiento* como un bloque de instrucciones con nombre, que termina en una instrucción de retorno. Para declarar un procedimiento se utiliza las directivas PROC y ENDP. Se le debe asignar un nombre (un identificador válido). Cada uno de los programas que hemos escritos hasta ahora contiene un procedimiento llamado **main**, por ejemplo,

```
main PROC

.

.

main ENDP
```

Al crear un procedimiento distinto al procedimiento de inicio de un programa, se debe terminar con una instrucción RET, la cual obliga a la CPU a regresar a la ubicación desde la que se llamó al procedimiento:

```
ejemplo PROC

      .

      .

      ret

ejemplo END
```

El procedimiento de inicio (**main**) es un caso especial, ya que termina con la instrucción **exit**. Al utilizar la instrucción INCLUDE *Irvine32.inc*, **exit** es un alías para una llamada a **ExitProcess**, un procedimiento del sistema que termina el programa:

```
INVOKE ExitProcess,0
```

En el capítulo 8 presentaremos la directiva INVOKE, que puede llamar a un procedimiento y pasarle argumentos.

Si se utiliza la instrucción INCLUDE Irvine 16.inc, **exit** se traduce a la directiva de ensamblador **.EXIT**. Esta directiva hace que el ensamblador genera las siguientes dos instrucciones:

```
 mov ah,4C00H        ; llama a la función 4CH de MS-DOS
int 21h              ; termina el programa
```

Ejemplo: suma de tres enteros

Vamos a crear un procedimiento llamado **SumaDE**, para calcular la suma de tres enteros de 32 bits. Asumiremos que se asignan enteros relevantes a EAX, EBX y ECX antes de llamar al procedimiento. Este procedimiento devuelve la suma en EAX

```
SumaDe PROC

       add eax, ebx

       add eax,ecx
```

```
        ret

SumaDe ENDP
```

Documentación de los procedimientos

Un buen hábito que cultivar es el de agregar una documentación clara y legible a nuestros programas. A continuación se muestran algunas sugerencias para la información que se puede colocar al principio de cada procedimiento:

- Una descripción de todas las tareas que realiza el procedimiento.
- Una lista de los parámetros de entrada y su uso, etiquetados mediante una palabra como **Recibe**. Si alguno de los parámetros de entrada tiene requerimientos específicos para los valores de entrada, se deben mostrar aquí.
- Una descripción de los valores devueltos por el procedimiento, etiquetados mediante una palabra como **Devuelve**.
- Una lista de los requerimientos especiales, conocidos como *condiciones previas* que deben satisfacerse para poder llamar al procedimiento. Éstos pueden etiquetarse mediante la
  Palabra **Requiere**. Por ejemplo, para un procedimiento que dibuja una línea de gráficos, una condición previa útil sería que el adaptador de video ya se debe encontrar en modo gráficos.

Las etiquetas descriptivas que hemos elegido, como Recibe, Devuelve y Requiere, no son las únicas; a menudo se utilizan otros nombres útiles.

Con estas ideas en mente, vamos a agregar la documentación apropiada al procedimiento **SumaDe**:

```
SumaDe PROC

; Calcula y devuelve la suma de tres enteros de 32 bots-

; Recibe: EAX, EBX, ECX, los tres enteros. Pueden ser

;         con o sin signo.

; Devuelve: EAX = suma

; -----------------------------------------------------------------

        add eax,ebx

        add eax,ecx

SumaDe ENDP
```

Por lo general, las funciones escritas en lenguaje de alto nivel, como C y C++, devuelven valores de 8 bits en AL, valores de 16 bits en AX, y valores de 32 bits en EAX.

Instrucciones CALL y RET

La instrucción CALL llama a un procedimiento, para lo cual dirige al procesador para que empiece la ejecución en una nueva ubicación de memoria. El procedimiento utiliza una instrucción RET (retorno del procedimiento) para regresar al procesador al punto en el programa en el que se llamó al procedimiento. Hablando en el sentido mecánico, la instrucción CALL mete su dirección de retorno en la pila y copia la dirección del procedimiento al que se llamó en el apuntador de instrucciones. Cuando el procedimiento está listo para regresar, su instrucción RET saca la dirección de retorno de la pila y la coloca en el apuntador de instrucciones. En modo de 32 bits, la CPU ejecuta la instrucción en la memoria a la que apunta el registro EIP (registro apuntador de instrucciones). En modo de 16 bits, IP apunta a la instrucción.

Ejemplo de llamada y retorno

Supongamos que en **main**, una instrucción CALL se encuentra en el desplazamiento 00000020. Por lo general esta instrucción requiere de cinco bytes de código máquina, por lo que la siguiente instrucción (MOV en este caso) se encuentra en el desplazamiento 00000025:

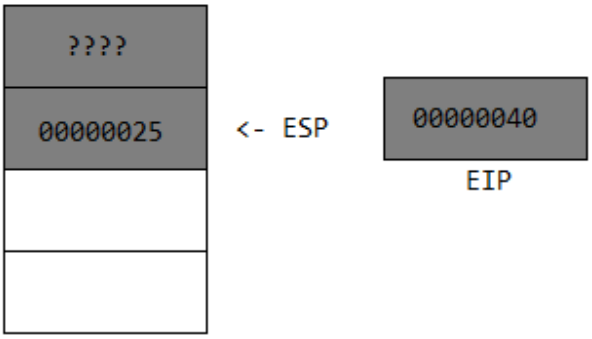
```
    main PROC

00000020 call MiSub

00000025 mov  eax,ebx
```

Ahora, supongamos que la primera instrucción ejecutable en **MiSub** se encuentra en el desplazamiento 00000040:

```
        MiSub PROC

00000040 mov eax,edx

            .

            .

         ret

MiSub ENDP
```

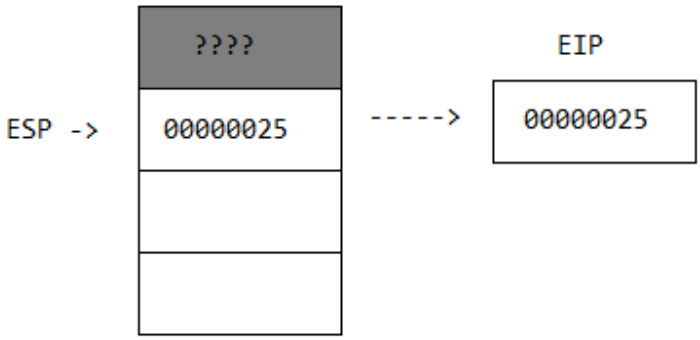
Cuando se ejecuta la instrucción CALL, la dirección que sigue después de la llamada (00000025) se mete en la pila, y la dirección **MiSub** se carga en EIP.

Ejecución de una instrucción CALL.

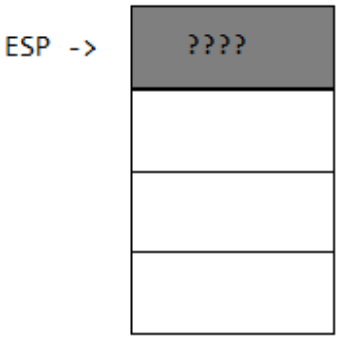


Todas las instrucciones en **MiSub** se ejecutan hasta la instrucción RET. Cuando se ejecuta la instrucción RET, el valor en la pila al que apunta ESP se saca y se coloca en EIP.

Ejecución de la instrucción RET.

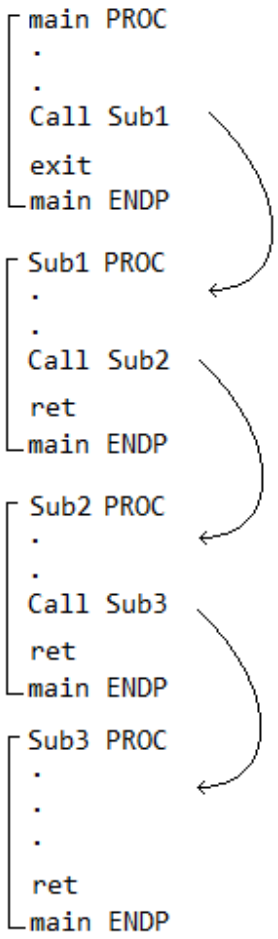


Luego, ESP se decrementa, de manera que apunta al valor anterior en la pila.



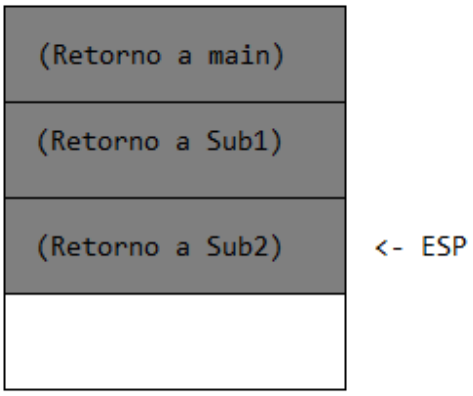
Llamadas a procedimientos anidadas

Una *llamada a procedimiento anidada* ocurre cuando un procedimiento al cual se llamó hace una llamada a otro procedimiento antes de que el primer procedimiento regrese. Supongamos que **main** llama a un procedimiento llamado **Sub1**. Mientras **Sub1** se ejecuta, hace una llamada al procedimiento **sub2**. Mientras **Sub2** se ejecuta, hace una llamada a **Sub3**. El proceso se muestra a continuación:

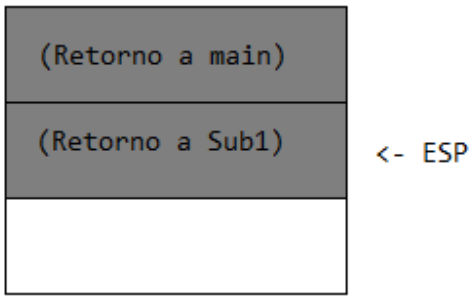


Cuando se ejecuta la instrucción RET al final de **sub3**, saca el valor que hay en la pila[ESP] y lo coloca en el apuntador de instrucciones. Esto hace que la ejecución continúe en la instrucción

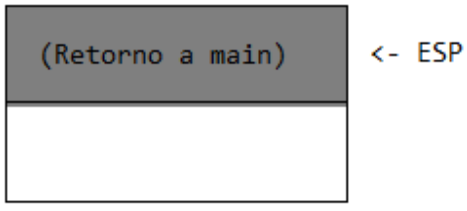
que va después de la instrucción que llama a sub3. El siguiente diagrama muestra la pila, justo antes de que se ejecute el retorno de **sub3**:



Después del retorno, ESP apunta a la siguiente entrada más alta en la pila. Cuando la instrucción RET al final de **Sub2** está a punto de ejecutarse, la pila aparece como se muestra a continuación:



Por último, cuando **Sub1** regresa, se saca pila[ESP] y se coloca en el apuntador de instrucciones, y la ejecución se reanuda en **main**:



Es evidente que la pila en sí demuestra ser un dispositivo útil para recordar información, incluyendo las llamadas a procedimientos anidadas. En general, las estructuras tipo pila se utilizan en situaciones en las los programas deben volver a trazar sus oasis en un orden específico.

Paso de argumentos tipo registro a los procedimientos

Se escribimos un procedimiento que realice alguna operación estándar, como calcular la suma de un arreglo de enteros, no es conveniente incluir referencias a nombres de variables específicos dentro del procedimiento. Si se hace, el procedimiento solo podrá usarse con un arreglo. Un mejor método es pasar el desplazamiento de un arreglo al procedimiento, y pasarle un entero que especifique el número de elementos del arreglo. A estos valores les llamamos *argumentos* (o *parámetros de entrada*). En lenguaje ensamblador, es común pasar argumentos dentro de los registros de propósito general.

En la sección anterior creamos un procedimiento simple llamado **SumaDe**, el cual sumaba los enteros en los registros EAX, EBX y ECX En **main**, antes de llamar a **SumaDe**, asignamos valores a EAX, EBX y ECX:

```
.data
laSuma DWORD ?
.code
main PROC
     mov eax,10000h            ; argumento
     mov ebx,20000h            ; argumento
     mov ecx,30000h            ; argument
     call SumaDe               ; EAX = (EAX + EBX + ECX)
     mov laSuma,eax            ; guarda la suma
```

Después de la instrucción CALL, tenemos la opción de copiar la suma en EAX a una variable.

Ejemplo: Suma de un arreglo de enteros

Un tipo bastante común de ciclo, que probablemente ya se ha codificado en C++ o en Java, es el que calcula la suma de un arreglo de enteros. Esto es muy fácil de llevarse a cabo en el lenguaje ensamblador, y puede codificarse de tal forma que se ejecute lo más rápido posible. Por ejemplo, podremos usar registros en vez de variables dentro de un ciclo.

Vamos a crear un procedimiento llamado **SumaArreglo**, el cual recibe dos parámetros de un programa, este procedimiento calcula y devuelve la suma del arreglo en EAX

```
;-----------------------------------------------------------------

SumaArreglo PROC
```

```
; Calcula la suma de un arreglo de enteros de 32 bits.
; Recibe   ESI = el desplazamiento del arreglo
;          ECX = número de elementos en el arreglo
; Devuelve EAX = suma de los elementos del arreglo
; ----------------------------------------------------------------
push esi                 ; guarda ESI, ECX
push ecx
mov eax,0                ; establece la suma en cero
L1: add eax,[ESI]        ; agrega cada entero a la suma
add esi, TYPE DWORD      ; apunta al siguiente entero
loop L1                  ; repite para el tamaño del arreglo
pop ecx                  ; restaura a ECX, ESI
pop esi
ret                      ; la suma está en EAX
SumaArreglo ENDP
```

Nada en este procedimiento es específico para el nombre o el tamaño de un cierto arreglo. Podría utilizarse en cualquier programa que necesite sumar un arreglo de enteros de 32 bits. Siempre que sea posible, debemos crear procedimientos que sean flexibles y adaptables.

Llamada a SumaArreglo A continuación se muestra un ejemplo de una llamada a **SumaArreglo**, en donde se le pasa la dirección de **arreglo** en ESI y la cuenta del arreglo en ECX, Después de la llamada copiamos la suma en EAX a una variable:

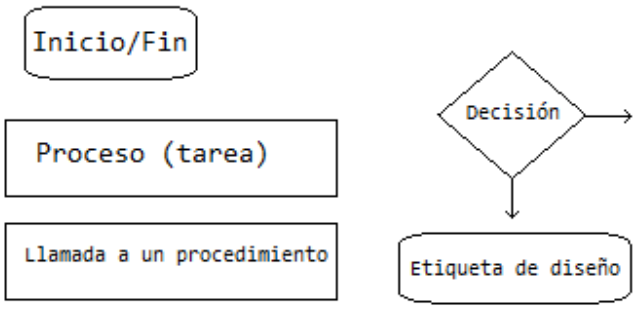
```
.data
arreglo DWORD 10000h,20000h,30000h,40000h,50000h
laSuma DWORD ?
.code
main PROC
     mov esi,OFFSET arreglo                ; ESI apunta al arreglo
     mov ecx, ECX LENGTHOF arreglo         ; ECX = cuenta del arreglo
```

```
call SumaArreglo                    ; calcula la suma
mov laSuma,eax                      ; se devuelve en EAX
```

Diagrama de flujo

Un *diagrama de flujo* es una manera bien establecida de diagramar la lógica de un programa. Cada figura en un diagrama de flujo representa un solo paso lógico, y las líneas con flechas que conectan a las figuras muestran el orden de los pasos lógicos.

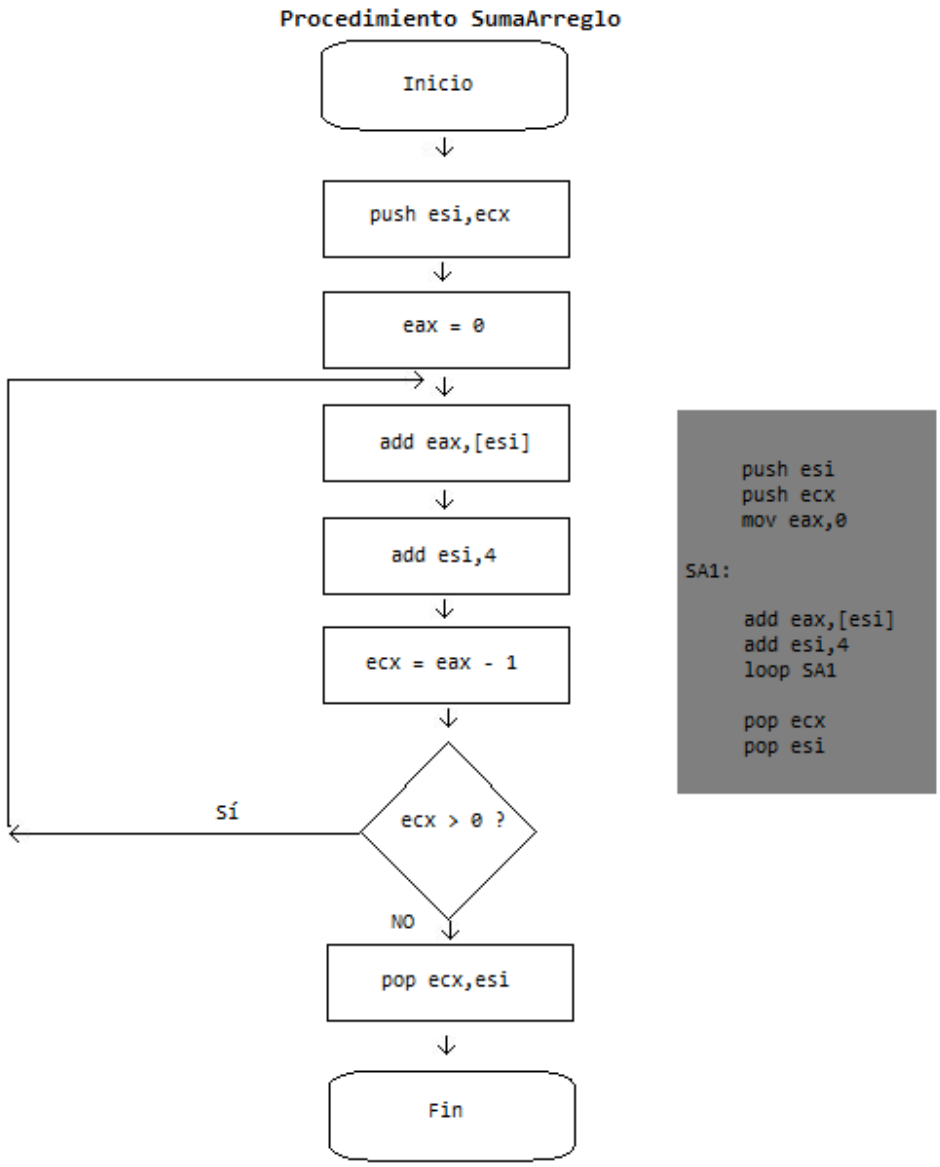
Figuras básicas de un diagrama de flujo



*Figuras más comunes de los diagramas de flujo. La misma figura se utiliza para los conectores inicio/fin, así como para las etiquetas que son los destinos de las instrucciones de salto.

Las notaciones de texto como *sí* y *no* se agregan a un lado del símbolo de *decisión*, para mostrar las direcciones de las bifurcaciones. No hay una posición requerida para cada flecha conectada a un símbolo de decisión. Cada símbolo de proceso puede contener una o más instrucciones estrechamente relacionadas. Las instrucciones no necesitan tener una sintaxis correcta. Por ejemplo, podríamos sumar 1 a CX usando cualquiera de los siguientes símbolos de proceso

Vamos A utilizar el procedimiento **SumaArreglo** de la sección anterior para diseñar un diagrama de flujo simple.



Utiliza un símbolo de decisión para la instrucción LOOP, ya que ésta debe determinar si va a transferir o no el control a una etiqueta (con base en el valor de CX). El fragmento de código muestra el listado del procedimiento original.

Almacenamiento y restauración de registros

En el ejemplo **SumaArreglo**, ECX y ESI se metieron en la pila al principio del procedimiento y se sacaron al final. Esta acción es común en la mayoría de los procedimientos que modifican registros. Siempre hay que guardar y restaurar los registros que modifican un procedimiento, para que el programa que hace la llamada pueda estar seguro de que no se sobrescribirá ninguno de sus propios valores de registro. La excepción a esta regla pertenece a los registros que se utilizan como valores de retorno, por lo general, EAX. No se deben meter y sacar.

Operador USES

El operador USES junto con la directiva PROC, nos permite presentar los nombres de todos los registros que se modifican dentro de un procedimiento. USES indica al ensamblador que debe hacer dos cosas: primero, debe generar la instrucción PUSH que se guarden los registros en la pila, al principio del procedimiento. Después, debe generar instrucciones POP para restaurar los valores de los registros al final del procedimiento. El operador USES va inmediatamente después de PROC, y va seguido de una lista de registros en la misma línea separados por espacio o tabuladores (no por comas).

El procedimiento **SumaArreglo** utiliza instrucciones PUSH y POP para almacenar y restaurar los registros ESI y ECX. El operador USES puede hacer lo mismo, con más facilidad:

```
SumaArreglo PROC USES esi ecx
        mov eax,0               ; establece la suma a cero
L1:
        add eax,[esi]           ; agrega cada entero a suma
        add esi,4               ; apunta a l siguiente entero
        loop L1                 ; repite para el tamaño del arreglo
        ret                     ; la suma está en EAX
SumaArreglo ENDP
```

El código correspondiente que genera el ensamblador nos muestra el efecto de USES:

```
SumaArreglo PROC
push esi
push ecx
mov eax,0                   ; establece la suma a cero
L1:
add eax,[esi]               ; agrega cada entero a la suma
add esi,4                   ; apunta al siguiente entero
loop L1                     ; repite para el tamaño del arreglo
pop ecx
pop esi
```

```
ret

SumaArreglo ENDP
```

Tip de depuración: al utilizar el depurador de Microsoft Visual Studio, Se puede verlas instrucciones de máquinas ocultas que generan las directivas y operadores avanzados de MASM, seleccionamos la opción Desensamblado en el menú *Depurar* | *ventanas*. Esta ventana mostrará el código de fuente de los programas, junto con las instrucciones máquinas ocultas que genera el ensamblador.

Excepción Hay una importante excepción a nuestra regla existente acerca de guardar los registros que se aplica cuando un procedimiento devuelve un valor en un registro (Por lo general, EAX). En este caso, el registro de retorno debería meterse y sacarse de la pila. Por ejemplo, en el procedimiento **SumaDe**, si metemos y sacamos el registro EAX, se pierde el valor de retorno del procedimiento:

```
SumaDe PROC             ; suma de tres enteros

push eax                ; guarda EAX

add eax,ebx             ; calcula la suma

add eax,ecx             ; de EAX, EBX, ECX

pop eax                 ; se perdió la suma!

ret

SumaDe ENDP
```

Diseño de programas mediante el uso de procedimientos

Cualquier aplicación de programación poco común tiende a involucrar una variedad de tareas distintas. Podríamos codificar todas las tareas en un solo procedimiento, pero el programa sería difícil de leer y mantener. Es mejor dedicar un solo procedimiento para cada tarea.

Al crear un programa, se debe crear un conjunto de especificaciones en las que se mencione con exactitud lo que se supone que debe hacer el programa. Las especificaciones deben ser el resultado de un cuidadoso análisis del problema que tratamos de resolver. Después se diseña el programa de acuerdo con las especificaciones. Un método de diseño estándar es dividir un problema general en tareas discretas; a este proceso se le conoce como *descomposición funcional*, o *diseño arriba-abajo*. Este método depende de ciertos principios básicos:

- Un problema extenso puede dividirse con más facilidad en pequeñas tareas.
- Un programa es más fácil de mantener si cada procedimiento se prueba por separado.
- Un diseño de arriba-abajo nos permite ver cuántos procedimientos están relacionados entre sí.

- Cuando se está seguro del diseño en general, es más fácil concentrarse en los detalles, escribiendo el código que implemente cada procedimiento.

En la siguiente sección demostraremos el método de diseño de arriba-abajo para un programa que recibe enteros como entrada y calcula su suma. Aunque el programa es simple, el mismo método puede aplicarse a programas de casi cualquier tamaño.

Programa para sumar enteros (diseño)

A continuación se muestran las especificaciones para un programa simple, al que llamaremos Suma de enteros:

> Crear un programa que pida al usuario tres enteros de 32 bits, los almacene en un arreglo, calcule la suma del arreglo y muestre la suma en la pantalla.

El siguiente seudocódigo muestra cómo podríamos dividir las especificaciones en tareas:

```
Programa de suma de enteros

Pedir al usuario tres enteros

Calcular la suma del arreglo

Mostrar la suma
```

En nuestra preparación para escribir un programa, vamos a asignar un nombre de procedimiento a cada tarea

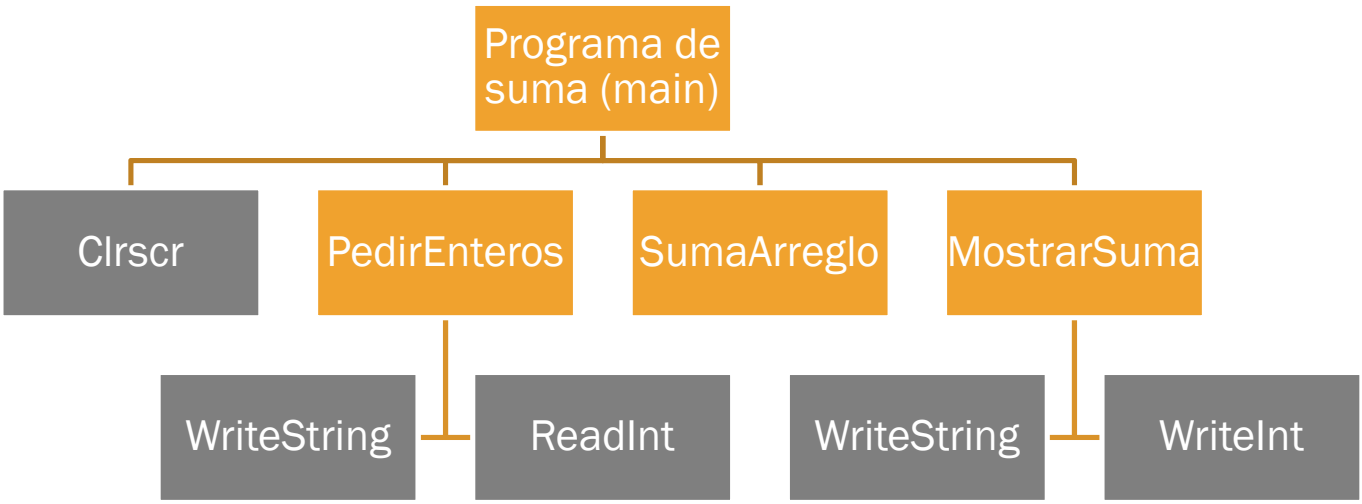
```
Main

     PedirEntero

     SumaArreglo

     MostrarSuma
```

En el lenguaje ensamblador, las tareas de entrada-salida requieren, por lo general, la implementación de código detallado. Para reducir parte de este detalle, podemos llamar a procedimientos que borren la pantalla, muestren una cadena, recibe un entero como entrada, y muestre un entero en pantalla:

```
Main

Clrscr                    ; borra la pantalla

PedirEnteros

   WriteString            ; muestra una cadena
```

```
   ReadInt                  ; recibe un entero como entrada
SumaArreglo                 ; suma los enteros
MostrarSuma
   WriteString              ; muestra una cadena
   WriteInt                 ; muestra un entero
```

Diagrama de estructura El diagrama a continuación, conocido como *diagrama de estructura*, describe la estructura del programa. Los procedimientos de la biblioteca de enlace están sombreados.



Programa maestro Versión mínima del programa. Este programa sólo contiene procedimientos vacíos (o casi vacíos). El programa se ensambla y se ejecuta, pero en realidad no hace nada útil. Un programa maestro nos proporciona la oportunidad de asignar todas las llamadas a los procedimientos, estudiar dependencia entre ellos y posiblemente mejorar el diseño estructura antes de codificar los detalles. Se debe usar comentarios en cada procedimiento para explicar su propósito y de los requerimientos de los parámetros.

Ejemplo de Programa maestro:

```
TITLE Programa de suma de enteros (Suma1.asm)
; Este programa pide al usuario tres enteros,
; los guarda en un arreglo, calcula la suma del
```

```
; arreglo y muestra la suma.
INCLUDE Irvine32.inc
.data
.code
main PROC
; Procedimiento principal de control del programa.
; Llama a: Clrscr, PedirEnteros, SumaArreglo, MostrarSuma
     exit
main ENDP
; -----------------------------------------------------------
PedirEnteros PROC
;
; Pide tres enteros al usuario, y los inserta
; en un arreglo.
; Recibe: ESI apunta a un arreglo de entero tipo doble palabra, ECX =
tamaño del arrego.
; Devuelve: el arreglo contiene los valores
; que introdujo el usuario
; Llama a : ReadInt, WriteString
; -----------------------------------------------------------
             ret
PedirEnteros ENDP
; -----------------------------------------------------------
SumaArreglo PROC
;
; Calcula la suma de un arreglo de enteros de 32 bits.
; Recibe: ESI apunta al arreglo, ECX = tamaño del arreglo
```

```
; Devuelve: EAX = suma de los elementos del arreglo
;-----------------------------------------------------------
            ret
SumaArreglo ENDP
; -----------------------------------------------------------
MostrarSuma PROC
;
; Muestra la suma en la pantalla
; Recibe: EAX = la suma
; Llama a: WriteString, WriteInt
;-----------------------------------------------------------
            ret
MostrarSuma ENDP
END main
```

*Una etiqueta de código seguida de dos puntos es local para su procedimiento circundante. Una etiqueta seguida de :: es global, lo que hace accesible desde cualquier instrucción en el mismo archivo código de fuente.

6-Procesamiento condicional

Instrucciones booleanas y de comparación

Empezaremos nuestro estudio del procesamiento condicional trabajando a nivel binario, con cuatro operaciones básicas del álgebra booleana: AND, OR, XOR y NOT. Estas operaciones se utilizan en el diseño del hardware y software computacional.

El conjunto de instrucciones IA-32 contiene las instrucciones AND, OR, XOR, NOT, TEST y BT*op*, las cuales implementan de manera directa las operaciones booleanas entre bytes, palabras y dobles palabras.

Operación	Descripción
AND	Operación AND booleana entre un operando de origen y un operando de destino
OR	Operación OR booleana entre un operando de origen y un operando de destino

XOR	Operación OR exclusivo booleana entre un operando de origen y un operando de destino
NOT	Operación NOT booleana sobre un operando de destino
TEST	Operación AND booleana implícita entre un operando de origen y uno de destino que, activan las banderas de la CPU en forma apropiada
BT,BTC,BTR,BTS	Copia el bit *n* del operando de origen a la bandera de Acarreo, y complementa/restablece/activa el mismo bit en el operando de destino

Las banderas de la CPU

Las instrucciones booleanas afectan a las banderas Cero, Acarreo, Signo, Desbordamiento y paridad. He aquí un breve repaso de su significado:

- La bandera Cero se activa cuando el resultado de una operación es igual a cero.
- La bandera de Acarreo se activa cuando una instrucción genera un resultado demasiado grande (o muy pequeño) para el operando de destino, cuando se le considera un entero sin signo.
- La bandera Signo es una copia del bit superior del operando de destino, indicando que es negativo si está activa y positivo si está borrada. (Se asume que Cero es positivo).
- La bandera de Desbordamiento se activa cuando una instrucción genera un resultado inválido con signo.
- La bandera de Paridad se activa cuan una instrucción genera un número par de bits 1 en el byte inferior del operando de destino.

Instrucción AND

La instrucción AND realiza una operación AND booleana (a nivel de bits) entre cada par de bits coincidentes en dos operandos, y se coloca el resultado en el operando de destino:

AND *destino,origen*

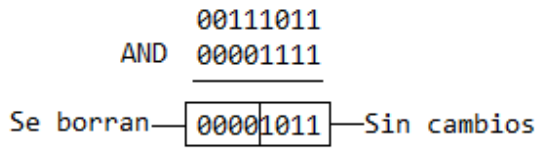
Se permite las siguientes combinaciones de operandos:

```
AND reg,reg

AND reg,mem

AND reg,imm

AND mem,reg

AND mem,imm
```

Los operandos pueden ser de 8, 16 o 32 bits, y deben tener el mismo tamaño. Para cada bit coincidente en los dos operandos, se aplica la siguiente regla; si ambos bits son iguales a 1, el bit de resultado es 1; en caso contrario, es 0. La siguiente tabla de verdad del capítulo 1 nombra a los bits de entrada como *x* y *y*. La tercera columna muestra el valor de la expresión **x^y**:

X	Y	X^Y
0	0	0
0	1	0
1	0	0
1	1	1

A menudo, la instrucción AND se utiliza para borrar los bits seleccionados y preservar otros. En el siguiente ejemplo, los cuatro bits superiores se borran y los cuatro bits inferiores permanecen sin cambios:



Las siguientes instrucciones llevan a cabo esta operación:

```
mov al,00111011b

and al,00001111b
```

Los cuatro bits inferiores podrían contener información útil, mientras que no nos importan los cuatro bits superiores. Es útil pensar en esta técnica como una extracción de bits, ya que los cuatro bits inferiores se “sacan” de AL.

Banderas La instrucción AND siempre borra las banderas Desbordamiento y Acarreo. Modifica las banderas Signo, Cero y Paridad de acuerdo con el valor del operando de destino.

Conversión de caracteres a mayúsculas

La instrucción AND proporciona una manera sencilla de traducir una letra de minúscula a mayúsculas. Si comparamos los códigos ASCII de la letra **A** y de la letra **a**, queda claro que sólo el bit 5 es diferente:

```
01100001 = 61h (‘a’)

01000001 = 41h (‘A’)
```

El resto de los caracteres alfabéticos tienen la misma relación. Si aplicamos la operación AND a cualquier carácter con el número 11011111 binario, todos los bits quedan sin cambio excepto el bit 5, que se borra. En el siguiente ejemplo, todos los caracteres en un arreglo se convierten en mayúsculas:

```
.data

arreglo BYTE 50 DUP(?)

.code

    mov ecx,LENGTHOF arreglo

    mov esi,OFFSET arreglo

L1: and BYTE PTR [esi],11011111b  ; borra el bit 5

    inc esi

loop L1
```

Instrucción OR

La instrucción OR realiza una operación booleana entre cada par de bits coincidentes en dos operandos, y coloca el resultado en el operando de destino:

```
OR destino, origen
```

La instrucción OR utiliza las mismas combinaciones de operandos que la instrucción AND:

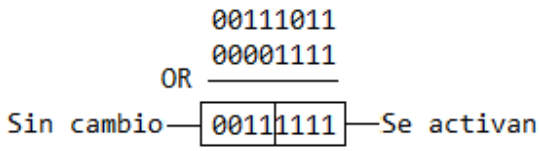
```
AND reg,reg

AND reg,mem

AND reg,imm

AND mem,reg

AND mem,imm
```

Los operandos pueden ser de 8, 16 o 32 bits, y deben tener el mismo tamaño. Para cada bit coincidente en los dos operandos, el bit de salida es 1 cuando por lo menos uno de los bits de entrada es 1. La siguiente tabla de verdad (capítulo 1) describe la expresión booleana **XvY**:

X	Y	XvY
0	0	0
0	1	1

1	0	1
1	1	1

A menudo, la instrucción OR se utiliza para activar los bits seleccionados y preservar los demás. En la siguiente figura, se aplica un OR entre 3Bh y 0Fh. Los cuatro bits inferiores del resultado se activan y los cuatro bits superiores permanecen sin cambio:



La instrucción OR puede usarse para convertir un byte que contenga un entero entre 0 y 9, en un dígito ASCII. para hacer esto, debemos activar los bits 4 y 5, por ejemplo, AL = 05h, podemos aplicarle un OR con 30h para convertirlo en el código ASCII para el dígito 5 (35h):

```
    00000101  05h
OR  00110000  30h
    00110101  35h.'5'
```

Las instrucciones en lenguaje máquina para hacer esto son:

```
mov dl,5       ; valor binario

or  dl,30      ; lo convierte a ASCII
```

Banderas La instrucción OR siempre borra las banderas Desbordamiento y Acarreo. Modifica las banderas Signo, Cero y Paridad de acuerdo con el valor del operando de destino. Por ejemplo, se puede aplicar OR a un número con sí mismo (o cero) para obtener cierta información de su valor:

```
or al,al
```

Los valores de las banderas Cero y Signo indican lo siguiente acerca del contenido de AL:

Bandera Cero	Bandera Signo	El valor en AL es...
Cero	Cero	Mayor que cero
Activa	Cero	Igual a cero
Cero	Activa	Menos que cero

Instrucción XOR

La instrucción XOR realiza una operación booleana OR exclusivo entre cada par de bits coincidentes en dos operandos, y almacena el resultado en el operando de destino:

XOR *destino,origen*

La instrucción XOR utiliza las mismas combinaciones y tamaños de operandos que las instrucciones AND y OR. Para cada bit coincidente en los dos operandos, se aplica lo siguiente: Si ambos bits son iguales (ambos 0 o ambos 1), el resultado es 0: en cualquier otro caso, el resultado es 1. La siguiente tabla de verdad prescribe la expresión booleana **X** xor **Y**:

X	**Y**	**X** xor **Y**
0	0	0
0	1	1
1	0	1
1	1	0

Un bit al que se le aplica OR exclusivo con 0 retiene su valor, y un bit al que se le aplica OR exclusivo con 1 cambia el valor opuesto (se complementa). XOR se invierte a sí mismo cuando se aplica dos veces el mismo operando. La siguiente tabla de verdad muestra que, cuando se aplica OR exclusivo al bit x con el bit y dos veces a su valor original:

X	**Y**	**X** xor **Y**	(**X** xor **Y**) xor **Y**
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Como veremos esta propiedad es “reversible” de XOR lo convierte en una herramienta ideal para una forma simple de cifrado simétrico.

Banderas La instrucción XOR siempre borra las banderas Desbordamiento y Acarreo. Modifica las banderas Signo, Cero y Paridad de acuerdo con el valor de operando de destino.

Comprobación de la bandera Paridad La bandera paridad indica si *el bit más inferior* del resultado de una operación a nivel de bits o aritmética tiene un número par o impar de bits que sean 1. La bandera se activa cuando la paridad es par y se borra cuando la paridad es

impar. Una manera de comprobar la paridad de un número sin cambiar su valor es aplicar un OR exclusivo con puros ceros:

```
mov al,10110101b            ; 5 bits = paridad impar

xor al,0                    ; la bandera de Paridad se borra (PO)

mov al,11001100b            ; 4 bits = paridad par

xor al,0                    ; La bandera Paridad se activa (PE)
```

A menudo, los depuradores utilizan PE para indicar paridad par o PO para indicar paridad impar.

Paridad de 16 bits Para comprobar la paridad de un registro de 16 bits, se aplica un OR exclusivo entre los bytes superior e inferior:

```
mov ax,64C1h                ; 0110 0100 1100 0001

xor ah,al                   ; la bandera Paridad se activa (PE)
```

Imaginemos que los bits activos (iguales a 1) en cada registro son miembros de un conjunto de 8 bits. La instrucción XOR pone en ceros todos los bits que pertenecen a la intersección de los conjuntos. XOR también forma la unión entre los bits restantes. La paridad de esta unión será la misma que la paridad del entero completo de 16 bits.

¿Qué hay acerca de los valores de 32 bits? Si numeramos los bytes de B0 a B3, podemos calcular la paridad como B0 XOR B1 XOR B2 XOR B3.

Instrucción NOT

La Instrucción NOT cambia el valor de todos los bits en un operando. Al resultado se le llama *complemento a uno*. Se permite los siguientes tipos de operandos:

```
NOT reg

NOT mem
```

Por ejemplo, el complemento a uno de F0h es 0Fh:

```
mov al,11110000b

not al                          ; AL = 00001111b
```

Banderas Ninguna bandera se ve afectada por la instrucción NOT.

Instrucción TEST

La instrucción TEST realiza una operación AND implícita entre cada par de bits coincidentes en dos operandos, y activa las banderas de manera acorde. La única diferencia entre TEST y AND es que TEST no modifica el operando de destino. La instrucción TEST permite las mismas combinaciones de operandos que la instrucción AND. En especial, TEST es valiosa para averiguar si los bits individuales en un operando están activos.

Ejemplo: prueba de varios bits La instrucción TEST puede comprobar varios bits a la vez. Supongamos que deseamos saber si el bit 0 o el bit 3 están activos en el registro AL. Podemos usar la siguiente instrucción para averiguarlo:

```
test al,00001001b               ; prueba los bits 0 y 3
```

(Al valor 00001001 en este ejemplo se llama *máscara de bits*). De los siguientes conjuntos de datos de ejemplo. Podemos inferir que la bandera Cero se activa sólo cuando todos los bits de prueba están en cero:

```
00100101 <- valor de entrada

00001001 <-valor de prueba

00000001 <-resultado: ZF = 0


00100100 <-valor de entrada

00001001 <- valor de prueba

00000000 <-resultado: ZF = 1
```

Banderas La instrucción TEST siempre borra las banderas Desbordamiento y Acarreo. Modifica las banderas Signo, Cero y Paridad de la misma forma que la instrucción AND.

Instrucción CMP

La instrucción CMP (comparar) resta implícitamente un operando de origen de un operando de destino. Ninguno de los operandos se modifica:

`CMP` *`destino,origen`*

CMP utiliza las mismas combinaciones de operandos que la instrucción AND.

Banderas La instrucción CMP cambia las banderas Desbordamiento, Signo, Cero, Acarreo, Acarreo auxiliar y Paridad de acuerdo con el valor que el operando de destina debería tener si se hubiera usado la instrucción SUB. Cuando se comparan dos operandos sin signo, las banderas Cero y Acarreo indican las siguientes relaciones entre los operandos:

Resultados de CMP	ZF	CF

Destino < origen	0	1
Destino > origen	0	0
Destino = origen	1	0

Cuando se comparan dos operandos con Signo, las banderas Signo, Cero y Desbordamiento indican las siguientes relaciones entre los operandos:

Resultados de CMP	Banderas
Destino < origen	SF ≠OF
Destino > origen	SF = OF
Destino = origen	ZF = 1

CMP es una valiosa herramienta para crear estructuras lógicas condicionales. Cuando se le coloca después de CMP una instrucción de salto condicional, el resultado es el equivalente en lenguaje ensamblador de una instrucción IF.

Ejemplos Analicemos tres fragmentos de código que muestran cómo se ven afectadas las banderas por la instrucción CMP. Cuando AX es igual a 5 y se compara con 10, la bandera de Acarreo se activa debido a que para restar 10 de 5 se requiere pedir prestado:

```
mov ax,5

cmp ax,10                    ; ZF = 0 y CF = 1
```

Al comparar 1000 con 1000 se activa la bandera Cero, ya que al restar el origen del destino se produce un cero:

```
mov eax,1000

mov cx,1000

cmp cx,ax                    ; ZF = 1 y CF = 0
```

Al comparar 105 con 0 se borran las banderas Cero y Acarreo, ya que 105 es mayor que 0:

```
mov si,105

cmp si,0                     ; ZF = 0 y CF = 0
```

Como establecer y borrar banderas individuales de la CPU

¿Cómo podemos activar o borrar fácilmente las banderas Cero, Signo, Acarreo y Desbordamiento? Hay varias formas, la mayoría requiere modificar el operando de destino. Para activar la bandera Cero, se aplica una operación TEST o AND a un operando con cero; para borrar la bandera Cero, se aplica un OR a un operando con 1:

```
test al,0                    ; activa la bandera Cero

and al,0                     ; activa la bandera Cero

or al,1                      ; borra la bandera Cero
```

TEST no modifica el operando, mientras que AND sí. Para activar la bandera Signo, se debe aplicar un OR al bit más alto de un operando con 1. Parra borrar la bandera Signo, se debe aplicar un AND al bit más alto con 0:

```
or al,80h                    ; activa la bandera Signo

and al,7Fh                   ; borra la bandera signo
```

Para activar la bandera Acarreo, se debe usar la instrucción STC; para borrar la bandera Acarreo, CLC:

```
stc                          ; activa la bandera Acarreo

clc                          ; borra la bandera Acarreo
```

Para activar la bandera Desbordamiento, se debe sumar dos valores de byte positivos que produzcan una suma negativa. Para borrar la bandera Desbordamiento, se debe aplicar un OR a un operando con 0:

```
mov al,7Fh                   ; al = +127

inc al                       ; al = 80h (-128), OF = 1

or eax,0                     ; borra la bandera Desbordamiento
```

Saltos condicionales

Estructuras condicionales

No hay estructuras lógicas de alto nivel en el conjunto de instrucciones IA-32, pero podemos implementar cualquier estructura lógica, sin importar qué tan compleja sea, mediante una combinación de comparaciones y saltos. Se requieren dos pasos para ejecutar una instrucción condicional: En primer lugar, un operando como CMP, AND o SUB modifica las banderas de la CPU. En segunda lugar, una instrucción de salto condicional prueba las banderas y provoca una bifurcación a una nueva dirección. Veamos un par de ejemplos:

Ejemplo 1 La instrucción CMP en el siguiente ejemplo compara a AL con Cero. La instrucción JZ (salta si es Cero) salta a la etiqueta **L1** si la instrucción CMP activó la bandera Cero:

```
cmp al,0

jz L1       ; salta si ZF = 1

.
```

```
.
L2:
```

Instrucción Jcond

Una instrucción de salto condicional se bifurca hacia una etiqueta de destino cuando una bandera de condición es verdadera. Si la bandera de condición es falsa, se ejecuta la instrucción que sigue justo después del salto condicional. La sintaxis es la siguiente

```
Jcond destino
```

cond se refiere a una bandera de condición que identifica el estado de una o más banderas. Por ejemplo:

Jc	Salta si hay acarreo (si la bandera Acarreo está activa)
Jnc	Salta si no hay acarreo (si la bandera Acarreo no está activa)
Jz	Salta si es cero (si la bandera Cero está activa)
Jnz	Salta si no es cero (si la bandera Cero no está activa)

Las banderas casi siempre se activan mediante las instrucciones aritméticas, de comparación y booleanas. Las instrucciones de salto condicional evalúan los estados de las banderas, y las utilizan para determinar si se deben realizar saltos o no.

Limitaciones De manera predeterminada, MASM requiere que el *destino* del salto sea una etiqueta dentro del procedimiento actual (en el capítulo 5 mencionamos esto con JMP). Para lidiar esta restricción, podemos declarar una etiqueta global (seguida por ::):

```
jc MiEtiqueta
.
.
MiEtiqueta::
```

En general, debemos evitar saltar fuera del procedimiento actual, ya que esto dificulta la depuración de un programa.

Uso de la instrucción CMP Supongamos que deseamos saltar a la ubicación L1 cuando AX es igual a 5. En el siguiente ejemplo, supongamos que AX es igual a 5: entonces la instrucción CMP activa la bandera Cero y la instrucción JE salta debido a que la bandera Cero está activa:

```
cmp ax,5
```

```
je L1                    ; salta si es igual
```

Si AX no fuera igual a 5, CMP borraría la bandera Cero y la instrucción JE no saltaría. En el siguiente ejemplo, el salto se realiza debido a que AX es menor que 6:

```
mov ax,5

cmp ax,66

jl L1                    ; salta si es menor
```

En el siguiente ejemplo, el salto se realiza ya que AX es mayor que 4:

```
mov ax,5

cmp ax,4

jg L1                    ; salta si es mayor
```

Tipos de instrucciones de saltos condicionales

El conjunto de instrucciones IA-32 tiene un sorprendente número de instrucciones de salto condicional. Estas instrucciones soportan un rango completo de instrucciones condicionales, para comparar enteros con o sin signo y comprobar las banderas de la CPU. Las instrucciones de salto condicional pueden dividirse en cuatro grupos:

- Con base en valores específicos de las banderas.
- Con base en la igualdad entre los operandos o el valor de (E)CX.
- Con base en las comparaciones de operandos sin signo.
- Con base en las comparaciones de operandos con signo.

La siguiente tabla muestra un listado de saltos con base en los valores específicos de las banderas de la CPU: Cero, Acarreo, Desbordamiento, Paridad y Signo.

Salto con base en los valores específicos de las banderas

Nemónico	Descripción	Banderas/Registros
JZ	Salta si es Cero	ZF = 1
JNZ	Salta si no es Cero	ZF = 0
JC	Salta si hay acarreo	CF = 1
JNC	Salta si no hay acarreo	CF = 0
JO	Salta si hay desbordamiento	OF = 1
JNO	Salta si no hay desbordamiento	OF = 0

JS	Salta si tiene signo	SF = 1
JNS	Salta si no tiene signo	SF = 0
JP	Salta si hay paridad (par)	PF = 1
JNP	Salta si no hay paridad (impar)	PF = 0

Los nombres de los operandos reflejan su orden para los operadores relacionados en álgebra. Por ejemplo en la expresión X < Y, X puede llamarse *opIzq* y Y puede llamarse *opDer*.

Comparación de igualdad

La siguiente tabla lista las instrucciones de salto con base en la evaluación de la igualdad de los dos operandos, o de los valores de CX y ECX. En la tabla, las notaciones *opIzq* y *opDer* se refieren a los operandos izquierdo (destino) y derecho (origen) en una instrucción CMP:

```
CMP opIzq,opDer
```

Saltos con base en la evaluación de la igualdad.

Nemónico	Descripción
JE	Salta si es igual (*opIzq* = *opDer*)
JNE	Salta si no es igual (*opIzq* distinto *opDer*)
JCXZ	Salta si CX = 0
JECXZ	Salta si ECX = 0

La instrucción JE es equivalente a JZ (salta si es Cero) y JNE es equivalente a JNZ (salta si no es Cero). Veamos algunos ejemplos:

Ejemplo 1:

```
mov edx,0A523h

cmp edx,0A523h

jne L5              ; no se realiza el salto

je L1               ; se realiza el salto
```

Ejemplo 2:

```
mov bx,1234h
```

```
sub bx,1234h

jne L5              ; no se realiza el salto

je L1               ; se realiza el salto
```

Ejemplo 3:

```
mov cx,0FFFFh

inc cx

jcxz L2             ; se realiza el salto
```

Ejemplo 4:

```
xor ecx,ecx

jecxz L2            ; se realiza el salto
```

Comparaciones sin signo

En la siguiente tabla se muestran los saltos que se basan específicamente en comparaciones de enteros sin signo. Este tipo de salto es útil cuando se comparan valores si signo. Por ejemplo, como enteros de 16 dígitos sin signo, 7FFFh es menor que 8000h. (Como enteros de 16 bits *con signo*, 7FFFh sería mayor que 8000h).

Saltos con base en comparaciones sin signo.

Nemónicos	Descripción
JA	Salta si es mayor (si *oplzq* > *opDer*)
JNBE	Salta si no es menor o igual (igual que JA)
JAE	Salta si es mayor o igual (si *oplzq* > o = *opDer*)
JNB	Salta si no es menor (Igual que JAE)
JB	Salta si es menor (si oplzq < opDer)
JNAE	Salta si no es mayor o igual (igual que JB)
JBE	Salta su es menor o giaul (si *oplzq* < o = *opDer*)
JNA	Salta si no es mayor (igual que JBE)

Comparaciones con signo

La siguiente tabla muestra una lista de saltos con base en las comparaciones con signo. Por ejemplo, el valor con signo de 1 byte 80h (-128d) es menor que 7Fh (+127d). El siguiente ejemplo muestra cómo difieren JA y JG en su comparación de 80h y 7Fh:

```
mov al,7Fh              ; (7Fh o +127)

cmp al,80h              ; (80h o -128)

ja esMayor              ; (no salta, ya que 7F no es > 80h)

jg esMayor              ; salta ya que +127 > -128
```

La instrucción JA no salta, ya que el número 7Fh sin signo es menor que el número 80h sin signo. Por otro lado, la instrucción JG salta ya que +127 (7Fh) es mayor que -128 (80h).

Saltos con base en comparaciones con signo.

Nemónico	Descripción
JG	Salta si es mayor (si *oplzq* > *opDer*)
JNLE	Salta si no es menor o igual (igual que JG)
JGE	Salta si es mayor o igual (si *oplzq* > o = *opDer*)
JNL	Salta si no es menor (igual que JGE)
JL	Salta si es menor (si *oplzq* < *opDer*)
JNGE	Salta si no es mayor o igual (igual que JL)
JLE	Salta si es menor o igual (si *oplzq* < o = *opDer*)
JNG	Salta si no es mayor (igual que JLE)

Veamos algunos ejemplos de comparaciones con signo.

Ejemplo 1:

```
mov edx,-1

cmp edx,0

jnl L5                ; no se realiza el salto

jnle L5               ; no se realiza el salto
```

```
jl L1                ; se realiza el salto
```

Ejemplo 2:

```
mov bx,+34

cmp bx,-35

jng L5               ; no se realiza el salto

jnge L5              ; no se realiza el salto

jnl L5               ; se realiza el salto
```

Ejemplo 3:

```
mov ecx,0

cmp ecx,0

jg L5                ; no se realiza el salto

jnl L1               ; se realiza el salto
```

Ejemplo 4:

```
mov ecx,0

cmp ecx,0

jl L5                ; no se realiza el salto

jng L1               ; se realiza el salto
```

Rangos de las instrucciones de salto condicional

En el modo real de 16 bits, los saltos condicionales utilizan un solo byte con signo, conocido como *desplazamiento relativo*, para localizar el destino del salto. El destino está limitado a un rango de -128 a +127 bytes, a partir del contador de la ubicación actual. El contador de ubicación es la dirección de la instrucción que sigue de la instrucción actual, ya que la CPU incrementa el apuntador de instrucciones antes de ejecutar la instrucción actual. En las instrucciones LOOP, LOOPZ y LOOPNZ se aplica la misma limitación de rango.

El siguiente ejemplo lista los bytes generados por el ensamblador para una instrucción JZ, cuando se compila en modo real de 16 bits. La instrucción JZ en el desplazamiento 0000 se codifica como **74 03**. El código de operación es 74 y el desplazamiento relativo es 03. (NOP

representa a la instrucción sin operación). La instrucción que va después de JZ es 0002, por lo que la CPU suma 3 al 2, produciendo 5 (el desplazamiento de la etiqueta L2):

```
Desplazamiento Codificación Código fuente ASM

-----------------------------------------

0000           74 03        jz L2

0002           90           nop

0003           90           nop

0004           90           nop

0005                        L2:
```

De manera similar, el siguiente ejemplo muestra un salto hacia atrás (desplazamiento negativo). El desplazamiento después del salto 0005, por lo que se suma 0FBh (-5) al 5, produciendo el desplazamiento 0000 (el desplazamiento de la etiqueta L1):

```
0000                        L1:

0000           90           nop

0001           90           nop

0002           90           nop

0003           74 FB        jz L1

0005
```

Saltos más largos en modo de 16 bits Si un salto en un programa en modo de 16 bits excede el rango permitido de desplazamiento de byte con signo, MASM genera un error *llamado fuera de rango de salto relativo*. Suponiendo que las instrucciones tengan una longitud de promedio de 3 bytes, podemos colocar alrededor de 40 instrucciones dentro de un ciclo antes de encontrarnos con un error. Para sortear este error, hay que saltar a una instrucción de salto incondicional (que tiene un rango de 16 bits):

```
     jz  L2

     jmp L3

L2:  jmp destinoLejano

L3:
```

Saltos en modo de 32 bits En modo de 32 bits, MASM genera un desplazamiento relativo de 32 bits con signo para los saltos a destinos que se encuentran fuera del rango de 1 byte. En el

siguiente ejemplo, la etiqueta L1 se encuentra 189 bytes (BDh) adelante del contador de ubicación, por lo que el campo de la dirección de destino es de 32 bits:

```
00000000 0F 84 000000BD jz L1
```

Los códigos de operación para los saltos de 32 bits son de 2 bytes, como en los números 0Fh, 84h que utilizamos en nuestro ejemplo.

Aplicación de saltos condicionales

Prueba de los bits de estado

A menudo, las instrucciones como AND, OR, NOT, CMP y TEST van seguidas las instrucciones de saltos condicionales que alteran el flujo del programa. Por lo general, los saltos condicionales evalúan los valores de las banderas de estado de la CPU. Por ejemplo, supongamos que un operando de la memoria de 8 bits llamado **estado** contiene información de estado acerca de un dispositivo conectado a la computadora. Las siguientes instrucciones saltan a una etiqueta si se activa el bit 5, lo cual indica que el dispositivo está desconectado:

```
mov  al,estado

test al,00100000b       ; evalúa el bit 5

jnz  EquipoDesconectado
```

Las siguientes instrucciones saltan a una etiqueta si alguno de los bits 0, 1 o 4 están activos:

```
mov  al,estado

test al,00010011b       ; evalúa los bits 0,1,4

jnz  ByteDatosEntrada
```

Para saltar a una etiqueta si están activos los bits 2, 3 y 7 se requieren las instrucciones AND y CMP:

```
mov al,estado

and al,10001100b        ; preserva los bits 2,3,7

cmp al,10001100b        ; ¿todo los bits activos?

je ReiniciarMaquina     ; sí: salta a la etiqueta
```

El mayor de dos enteros El siguiente código compara los enteros sin signo en AX y BX, y mueve el mayor de los dos a DX:

```
mov dx,ax               ; asume que AX es mayor

cmp ax,bx               ; si AX >= BX salta a L1
```

```
jae L1                  ; de lo contrario, mueve BX a DX
mov dx,bx
L1:
```

El menor de tres enteros Las siguientes instrucciones comparan los valores sin signo en las variables V1, V2 y V3, y mueven el menor de los tres a AX:

```
.data
V1 WORD ?
V2 WORD ?
V3 WORD ?
.code
mov ax,V1            ; asume que V1 es el menor
cmp ax,V2            ; si AX<= V2 entonces
jbe L1               ; salta a L1
mov ax,V2            ; de lo contario, mueve V2 a AX
L1: cmp ax,V3        ; si AX <= V3 entonces
jbe L2               ; salta a L2
mov ax,V3            ; de lo contrario, mueve V3 a AX
L2:
```

Instrucciones de prueba de bits (opcional)

Las instrucciones BT, BTC, BTR y BTS se llaman conjunto instrucciones de *prueba de bits*. Son interesantes, ya que ejecutan varios pasos dentro de una sola instrucción atómica. Esto tiene implicaciones para los subprogramas con subprocesamiento múltiple, en los que a menudo es muy importante poder evaluar, borrar, activar y complementar los bits de banderas (llamadas *semáforos*) sin peligro de interrupciones por parte de otro subproceso del programa.

Instrucción BT

La instrucción BT (prueba de bit) selecciona el bit *n* como el primer operando y lo copia en la bandera de Acarreo:

```
BT baseBit,n
```

El primer operando, llamado *baseBit*, no se cambia. BT permite los siguientes tipos de operandos:

```
BT r/m16,r16

BT r/m32,r32

BT r/m16,imm8

BT r/m32,imm8
```

En el siguiente ejemplo, la bandera Acarreo se le asigna el valor del bit 7 en la variable llamada **semáforo**:

```
.data

semáforo WORD 10001000b

.code

BT semáforo,7                ; CF = 1
```

Antes de que se introdujera la instrucción BT en el conjunto de instrucciones Intel, hubiéramos tenido que copiar la variable a un registro y desplazar el bit 7 hacia la bandera de Acarreo:

```
mov ax,semáforo

shr ax,8                     ; CF = 1
```

(Aquí, la instrucción SHR desplaza todos los bits en AX ocho posiciones a la derecha. Esto hace que el bit 7 se desplace hacia la bandera de Acarreo)

Instrucción BTC

La instrucción BTC (prueba y complemento de bit) selecciona el bit n en el primer operando, lo copia a la bandera Acarreo y lo complementa (cambia su valor):

```
BTC baseBit,n
```

BTC permite los mismos tipos de operandos que BT. En el siguiente ejemplo, a la bandera Acarreo se le asigna el valor del bit 6 en **semáforo**, y se complementa el mismo bit:

```
.data

semáforo WORD 10001000b

.code

BTC semáforo,6               ; CF = 0, semáforo=11001000b
```

Instrucción BTR

La instrucción BTR (prueba y restablecimiento de bit) selecciona el bit *n* en el primer operando, lo copia en la bandera Acarreo y restablece (borra) el bit *n*:

```
BTR baseBit,n
```

BTR permite los mismos tipos de operandos BT y BTC. En el siguiente ejemplo, a la bandera Acarreo se le asigna el valor del bit 7 en el semáforo y se borra el mismo bit:

```
.data

semáforo WORD 10001000b

.code

BTR semáforo,7                ; CF = 1; semáforo=00001000b
```

Instrucción BTS

La instrucción BTS (prueba de activación de bit) selecciona el bit n en el primer operando, lo copia a la bandera de Acarreo y lo activa:

```
BTS baseBit,n
```

BTS permite los mismos tipos de operandos que BT. En el siguiente ejemplo, a la bandera Acarreo se le asigna el valor del bit 6 en semáforo y después se activa el mismo bit:

```
.data

semáforo WORD 10001000b

.code

BTS semáforo,6                 ; CF = 0, semáforo=11001000b
```

Instrucciones de saltos condicionales

Instrucciones LOOPZ y LOOPE

La instrucción LOOPZ (salta si es cero) permite que un ciclo continúe mientras esté activa la bandera Cero y el valor sin signo de ECX sea mayor que cero. La etiqueta de destino debe estar a una distancia entre -128 y +127 bytes de la ubicación de la siguiente instrucción. La sintaxis es:

LOOPZ *destino*

La instrucción LOOPE (itera si es igual) es equivalente a LOOPZ, ya que comparten el mismo código de operación. Realizan las siguientes tareas:

```
ECX = ECX – 1

Si ECX > 0 y ZF = 1, saltar al destino
```

En caso contrario, no se produce ningún salto y el control pasa a la siguiente instrucción. LOOPZ y LOOPE no afectan a ninguna bandera de estado.

> Un programa que se ejecuta en modo direccionamiento real utiliza a CX como el contador de ciclo predeterminado en la instrucción LOOPZ. Si se desea forzar a que ECX sea el contador de ciclo, se debe usar la instrucción LOOPZD

Instrucciones LOOPNZ y LOOPNE

La instrucción LOOPNZ (salta si no es cero) es la contraparte de LOOPZ. El ciclo continúa mientras el valor sin signo de ECX sea mayor que cero, y la bandera Cero esté en cero. La sintaxis es:

LOOPNZ *destino*

La instrucción LOOPNE (salta si no es igual) es equivalente a LOOPNZ, ya que comparten el mismo código de operación. Estas instrucciones realizan las siguientes tareas:

```
ECX = ECX – 1
Si ECX > 0 y ZF = 0, salta al destino
```

En caso contrario, no ocurre nada y el control pasa a la siguiente instrucción.

Ejemplo El siguiente extracto de código (de Loopnz.asm) explora cada número en un arreglo hasta encontrar un número no negativo (cuando el bit de signo no está en cero):

```
.data
arreglo SWORD -3,-6,-1,-10,10,30,40,4
centinela SWORD 0
.code
     mov esi,OFFSET arreglo
     mov ecx,LENGTHOF arreglo
L1:  test WORD PTR [esi],8000h          ; prueba el bit de signo
     pushfd                             ; mete las banderas en la pila
     add esi,TYPE arreglo
     popfd                              ; saca las banderas de la pila
     loopnz L1                          ; continúa el ciclo
     jnz salir                          ; no se encontró un valor
```

```
    sub esi,TYPE arreglo                    ; ESI apunta al valor
salir:
```

Si se encuentra un valor no negativo. ESI se queda apuntando hacia él. Si el ciclo no se encuentra un número positivo, se detiene cuando ECX es igual a cero. En ese caso, la instrucción JNZ salta a la etiqueta **salir**, y ESI apunta al valor centinela (0) que está justo después del arreglo.

Estructuras condicionales

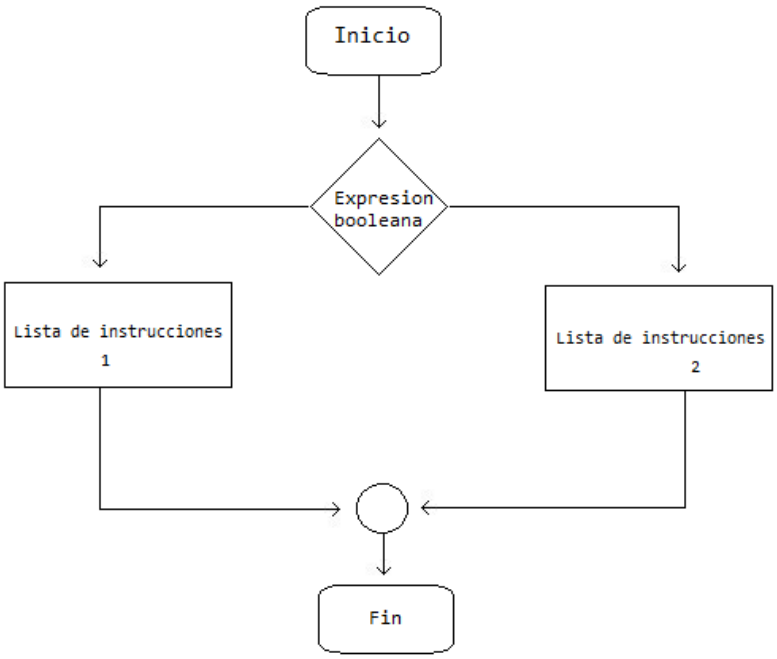
En esta sección examinaremos unas cuantas de las estructuras condicionales más comunes que se utilizan en los lenguajes de programación de alto nivel. Veremos cómo puede traducirse fácilmente cada estructura a lenguaje ensamblador. Consideremos que una *estructura condicional* es una o más expresiones condicionales que activan una elección entre dos bifurcaciones lógicas distintas. Cada bifurcación hace que se ejecute una secuencia distinta de instrucciones.

Instrucciones IF con estructura de bloque

En la mayoría de los lenguajes de alto nivel, una estructura IF implica que una expresión booleana deber ir seguida de dos listas de instrucciones: una que se realiza cuando la expresión es verdadera, y otra que se realiza cuando la expresión es falsa:

```
if(expresión)
   lista-instrucciones-1
else
   lista-instrucciones-2
```

La porción **else** de la instrucción es opcional. El diagrama de flujo a continuación muestra las dos rutas de bifurcación en una estructura IF condicional, las cuales se etiquetan como *verdadero* y *falso*.



Ejemplo 1 Usando la sintaxis Java/C++, se ejecutan dos instrucciones de asignación si **op1** es igual a **op2**:

```
if(op1 == op2)
{
    X = 1;
    Y = 2;
}
```

La única manera de traducir esta instrucción IF en lenguaje ensamblador es utilizar la instrucción CMP, seguida de uno o más saltos condicionales. Como **op1** y **op2** son operandos de memoria, hay que mover uno de ellos a un registro antes de ejecutar la instrucción CMP. El siguiente código implementa la instrucción IF de la manera más eficiente posible, invirtiendo la condición de igualdad y usando la instrucción JNE:

```
    mov eax,op1
    cmp eax,op2          ; ¿op1 == op2?
    jne L1               ; no: salta la siguiente instrucción
    mov X,1              ; sí: asigna X y Y
```

```
     mov Y,2
L1:
```

Si implementamos el operador == usando JE, el código resultante será menos compacto (seis instrucciones, en vez de cinco):

```
     mov eax,op1
     cmp eax,op2          ; ¿op1 == op2?
     je L1                ; sí: salta a L1
     jmp L2               ; no: salta las asignaciones
L1:  mov X,1              ; sí: asigna X y Y
     mov Y,2
L2:
```

Ejemplo 2 En el sistema de archivos FAT32 que utiliza MS-Windows, el tamaño de un clúster de disco depende de la capacidad general del mismo. En el siguiente seudocódigo, establecemos el tamaño del clúster a 4,096 si el tamaño del disco (en la variable llamada gigabytes) es menor que 8GB. En caso contrario, establecemos el tamaño del clúster a 8,192:

```
  tamCluster = 8192;
si gigabytes < 8
  tamCluster = 4096;
```

He aquí una buena manera de implementar la misma instrucción en lenguaje ensamblador:

```
mov tamCluster,8912     ; supone un clúster más grande
cmp gigabytes,8         ; ¿es mayor que 8GB?
jae siguiente
mov tamCluster,4096     ; cambia a un clúster más pequeño
siguiente:
```

En el capítulo 14 hablaremos sobre los clúster de disco.

Ejemplo 3 La siguiente instrucción IF-ELSE en seudocódigo tiene bifurcaciones alternativas:

```
if op1 > op2 then
    call Rutina1
```

```
else
    call Rutina2
end if
```

En la siguiente traducción de lenguaje ensamblador, asumimos que op1 y op2 son variables de tipo doble palabra con signo. El operador > se implementa usando JNG, el complemento de JG:

```
mov eax,op1
cmp eax,op2    ; ¿op1 > op2?
jng A1         ; no: llama a Rutina2
call Rutina1   ; si: llama a Rutina1
jmp A2
A1: call Rutina2
A2:
```

Uso de la prueba de la caja blanca

Las instrucciones condicionales complejas en lenguaje ensamblador tienen varias rutas de ejecución, lo cual dificulta su depuración mediante inspección (analizando el código). A menudo, los buenos programadores implementan una técnica conocida como *prueba de la caja blanca*, la cual verifica las entradas y correspondientes salidas de una subrutina. La prueba de la caja blanca requiere que se tenga una copia del código fuente. Podemos asignar una variedad de valores a las variables de entrada. Para cada combinación de entrada, se realiza un rastreo manual por el código de fuente, y se verifica la ruta de ejecución y las salidas producidas por la subrutina. Veamos cómo se hace esto. Supongamos que deseamos traducir la siguiente instrucción IF anidada en lenguaje ensamblador:

```
if op1 == op2 then
   if X > Y then
      call Rutina1
   else
      call Rutina2
   end if
else
```

```
 call Rutina3

end if
```

He aquí una posible traducción a lenguaje ensamblador, en la que se agregaron números de línea por cuestión de referencia. Se invierte la condición inicial (op1 == op2) y de inmediato se realiza un salto a la porción correspondiente al ELSE. Todo lo que queda por traducir es la instrucción IF-ELSE interior:

```
1: mov eax,op1

2: cmp eax,op2                    ; ¿op1 == op2?

3: jne L2                         ; no: llama a la Rutina3

; procesa la instrucción IF-ELSE interior

4: mov eax,X

5: cmp eax,Y                       ; ¿X > Y?

6: jg L1                           ; si: llama Rutina1

7: call Rutina2                    ; no: llama a Rutina2

8: jmp L3 ; y termina

9: L1: call Rutina1                ; llama a Rutina1

10: jmp L3                         ; y termina

11: L2: call Rutina3

12: L3:
```

La siguiente tabla muestra los resultados de aplicar la prueba de la caja blanca al código de ejemplo: Se asignaron valores de prueba a op1, op2, X y Y, y se verifican las rutas de ejecución resultantes.

Prueba de la instrucción IF anidada.

Op1	Op2	X	Y	Secuencia de ejecución de líneas	Llama a
10	20	30	40	1,2,3,11,12	Rutina3
10	20	40	30	1,2,3,11,12	Rutina3
10	10	30	40	1,2,3,4,5,6,7,8,12	Rutina2
10	10	40	30	1,2,3,4,5,6,9,10,12	Rutina1

Expresiones compuestas

Operador AND lógico

El lenguaje ensamblador implementa con facilidad las expresiones booleanas compuestas que contienen operadores AND. Consideremos el siguiente seudocódigo, en el que se asume que las variables son enteros sin signo:

```
if (al > bl) AND (bl > cl)
{
     X = 1
}
```

Evaluación de corto circuito La siguiente es una implementación directa que utiliza la evaluación de *corto circuito*, en la que la segunda expresión no se evalúa si la primera expresión es falsa:

```
     cmp al,bl                  ; primera expresión…
     ja L1
     jmp siguiente
L1:  cmp bl,cl                  ; segunda expresión…
     ja L2
     jmp siguiente
L2:  mov X,1                    ; ambas verdaderas: se establece X en 1
siguiente:
```

Podemos optimizar el código a cinco instrucciones, si cambiamos la instrucción JA inicial por JBE:

```
      cmp al,bl                 ; primera expresión…
      jbe siguiente             ; termina si es falso
      cmp bl,cl                 ; segunda expresión
      jbe siguiente             ; termina si es falso
      mov X,1                   ; ambas son verdaderas
```

```
siguiente:
```

El 29% de reducción en el tamaño del código (de siete instrucciones a cinco) se obtiene al dejar que la CPU pase a la segunda instrucción CMP si JBE no se ejecuta. Los compiladores de lenguaje de alto nivel para Java/C++ utilizan la evaluación de corto circuito, tal vez por razones de eficiencia.

Evaluación sin corto circuito Algunos lenguajes (como BASIC) no realizan la evaluación de corto circuito. Es difícil implementar una expresión compuesta de ese tipo en el lenguaje ensamblador, ya que se necesita una bandera o valor booleano para almacenar los resultados de la primera expresión:

```
.data

temp BYTE ?

.code

     mov temp,0                    ; borra la bandera temp

     cmp al,bl                     ; ¿AL > BL?

     jna L1                        ; no

     mov temp,1                    ; si: establece bandera = verdadero

L1:  cmp bl,cl                     ; ¿BL > CL?

     jna siguiente                 ; no: la expression es falsa

     and temp,1                    ; sí: se aplica AND a bandera y 1

     jz siguiente                  ; evalúa la bandera

     mov X,1

siguiente:
```

Para codificar este ejemplo de la manera más eficiente posible, necesitamos aprovechar la forma en que la instrucción AND afecta a la bandera Cero. Un compilador de BASIC ordinario no podría hacerlo tan bien. Las ocho instrucciones resultantes son aún 60% más grandes que las cinco instrucciones que utilizamos en la evaluación optimizada de corto circuito de la misma expresión.

Operador OR lógico

Cuando ocurren varias expresiones en una expresión compuesta que utiliza el operador lógico OR, la expresión es automáticamente verdadera, tan pronto como cualquiera de las expresiones sean verdaderas. Vamos a utilizar el siguiente seudocódigo como ejemplo:

```
if (al > bl) OR (bl > cl)
   X = 1
```

En la siguiente implementación, el código se bifurca a L1 si la primera expresión es verdadera, en caso contrario pasa a la segunda instrucción CMP. La segunda expresión invierte el operador > y utiliza JBE en su defecto:

```
     cmp al,bl              ; 1: compara AL con BL
     ja L1                  ; si es verdadero, omite la segunda expresión
     cmp bl,cl              ; 2: compara BL con CL
     jbe siguiente          ; falso: omite la siguiente instrucción
L1:  mov X,1
siguiente:
```

Para una expresión compuesta dada, hay por lo menos varias formas en que ésta puede implementarse en lenguaje ensamblador:

Ciclos WHILE

La estructura WHILE evalúa una condición antes de ejecutar un bloque de instrucciones. Mientras que la condición del ciclo permanezca siendo verdadera, se repiten las instrucciones. El siguiente ciclo está escrito en C++:

```
while (val1 < val2)
{
     val1++;
     val2--;
}
```

Al codificar esta estructura en lenguaje ensamblador, es conveniente invertir la condición del ciclo y saltar a **finwhile** cuando la condición se vuelve verdadera. Suponiendo que val1 y val2 son variables, debemos mover una de ellas a un registro al principio, y restaurar la variable al final:

```
     mov eax,val1        ; copia la variable a EAX
@@while:
     cmp eax,val         ; if not (val1 < val2)
     jnl finwhile        ; sale del ciclo
```

```
     inc eax             ; val1++;
     dec val2            ; val2--;
     jmp @@while2        ; repite el ciclo
finwhile:
     mov val1,eax
```

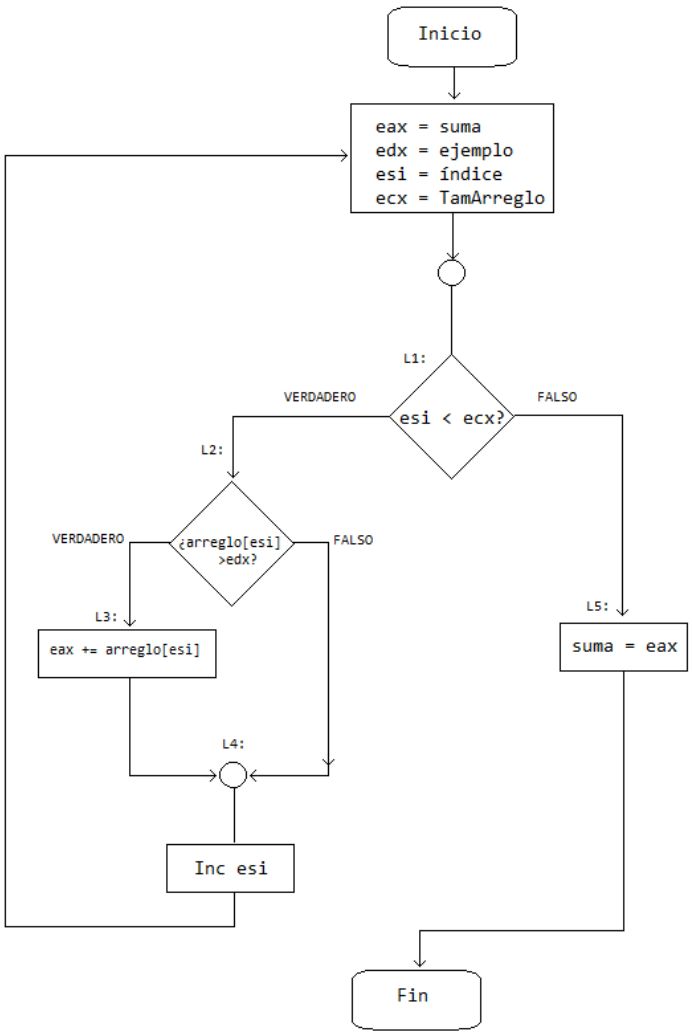
EAX es un proxy (sustituto) para **val1** dentro del ciclo. Las referencias a val1 deben ser a través de EAX. utiliza JNL, lo cual implica que **val1** y **val2** son enteros con signo.

Ejemplo: instrucción IF anidada en un ciclo

Los lenguajes estructurados de alto nivel son muy adecuados para representar estructuras de control anidadas. En el siguiente ejemplo C++, una instrucción IF está anidada dentro de un ciclo WHILE, Esta instrucción calcula la suma de todos los elementos del arreglo que sean mayores que el valor en **ejemplo**:

```
int arreglo [] = {10,60,20,33,72,89,45,65,72,18}
int ejemplo = 50;
inr TamArreglo = sizeof arreglo / sizeof ejemplo;
int indice = 0;
int suma = 0;
while (índice < TamArreglo)
{
   if(arreglo[índice]> ejemplo)
   {
    suma +=arreglo[índice];
    indice++;
   }
}
```

Antes de codificar este ciclo en lenguaje ensamblador, utilizaremos el siguiente diagrama de flujo para describir la lógica.



Para simplificar la traducción y agilizar la ejecución reduciendo el número de accesos a memoria, hemos sustituido los registros por variables. EDX = ejemplo, EAX = suma, ESI = índice y ECX = TamArreglo (una constante). Se agregaron nombres de etiquetas a las figuras.

Código en ensamblador La manera más sencilla de generar código ensamblador de un diagrama de flujo es implementar el código para cada figura. Observemos la correlación directa entre las etiquetas del diagrama de flujo y las etiquetas que se utilizan en el siguiente código de fuente

```
.data

suma DWORD 0

ejemplo DWORD 50

arreglo DWORD 10,60,20,33,72,89,45,65,2,18
```

```
TamArreglo = ($ - arreglo) / TYPE arreglo

.code

main PROC

     mov eax,0                      ; suma

     mov edx,ejemplo

     mov esi,0                      ; índice

     mov ecx,TamArreglo

L1:  cmp esi,ecx

     jl L2

     jmp L5

L2:  cmp arreglo[esi*],edx

     jg   L3

     jmp  L4

L3:  add eax,arreglo[esi*4]

L4:  inc esi

     jmp L1

L5: mov suma,eax
```

Selección controlada por tablas

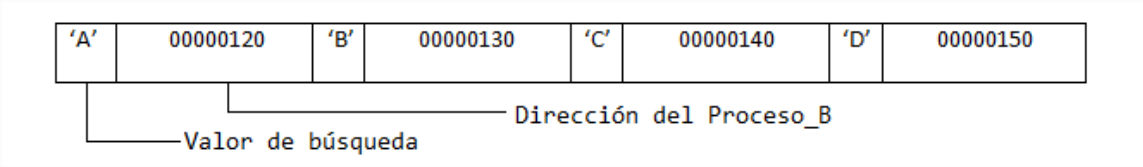
La *selección controlada por tablas* es una forma de utilizar una búsqueda en tablas para sustituir una estructura de selección de varias vías. Para usarla, debemos crear una tabla que contenga valores de búsqueda y los desplazamientos de etiquetas o procedimientos, y utilizar un ciclo para buscar en la tabla. Esto funciona mejor cuando se realiza una gran cantidad de comparaciones.

Por ejemplo, el siguiente código es parte de una tabla que contiene valores de búsqueda de un solo carácter y direcciones de procedimientos:

```
.data

TablaCasos BYTE ‘A’         ; valor de búsqueda

     DWORD Proceso_A        ; dirección de procedimiento
```

```
BYTE ‘B’

DWORD Proceso_B

(etc.)
```

Supongamos que Proceso_A, Proceso_B, Proceso_C y Proceso_D se encuentran en la direcciónes 120h, 130h, 140h y 150h, respectivamente. La tabla se ordenaría en memoria como se muestra en la siguiente figura:



Programa de ejemplo En el siguiente programa de ejemplo, el usuario introduce un carácter desde el teclado. Mediante el uso de un ciclo, el carácter se compara con cada entrada en la tabla.

La primera coincidencia encontrada en la tabla produce una llamada al desplazamiento del procedimiento almacenado inmediatamente después del valor de búsqueda. Cada procedimiento carga a EDX con el desplazamiento de una cadena distinta, la cual se muestra durante el ciclo:

```
TITLE Tabla de desplazamiento de procedimientos (TablaProc.asm)

; Este programa contiene una tabla con desplazamientos de procedimientos.

; Utiliza la tabla para ejecutar llamadas indirectas a procedimientos

; Última actualización: 06/01/2006

INCLUDE Irvine32.inc

.data

TablaCasos BYTE ‘A’                  ; valor de búsqueda

           DWORD Proceso_A           ; dirección del procedimiento

TamanioEntrada = ($ - TablaCasos)

           BYTE ‘B’

           DWORD Proceso_B

           BYTE ‘C’

           DWORD Proceso_C
```

```
          BYTE 'D'
          DWORD Proceso_D
NumeroDeEntradas = ($ - TablaCasos)/ TamanioEntrada
indicador BYTE "Oprima A,B,C, o D mayúscula:",0
msjA BYTE "Proceso_A",0
msjB BYTE "Proceso_B",0
msjC BYTE "Proceso_C",0
msjD BYTE "Proceso_D",0
.code
main PROC
     mov edx,OFFSET indicador        ; pide la entrada al usuario
     call WriteString
     call ReadChar                   ; lee un character y lo coloca en AL
     mov ebx,OFFSET TablaCasos       ; apunta EBX a la tabla
     mov ecx,NumeroDeEntradas        ; contador de ciclo
L1:
     cmp al,[ebx]                    ; ¿se encontró coincidencia?
     jne L2                          ; no: continúa
     call NEAR PTR [ebx+1]           ; si: llama al procedimiento
```

Esta instrucción CALL llama al procedimiento cuya dirección se encuentra almacenada en la ubicación de memoria a la que EBX + 1 hace referencia. Una llamada indirecta tal como ésta requiere el operador NEAR PTR.

```
     call WriteString                ; muestra un mensaje
     call Crlf
     jmp L3
L2:
     add ebx,TamanioEntrada          ; apunta a la siguiente entrada
```

```
     loop L1
L3:
     exit
main ENDP
```

Cada uno de los siguientes procedimientos mueve un desplazamiento de cadena distinto a EDX:

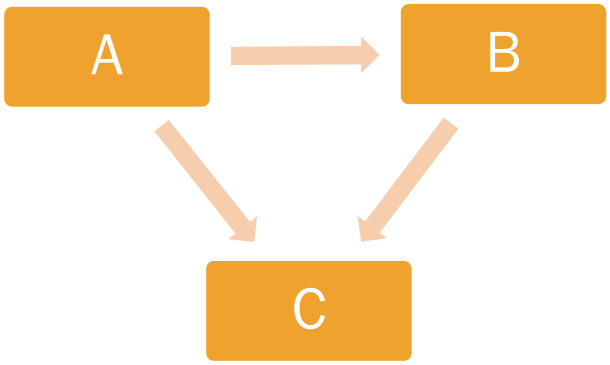
```
Proceso_A PROC
    mov edx,OFFSET msjA
    ret
Proceso_A ENDP
Proceso_B PROC
    mov edx,OFFSET msjB
    ret
Proceso_B ENDP
Proceso_C PROC
    mov edc,OFFSET msjC
    ret
Proceso_C ENDP
Proceso_D PROC
    mov edx,OFFSET msjD
    ret
Proceso_D ENDP
END main
```

El método de selección controlado por una tabla implica cierta sobrecarga inicial, pero puede reducir la cantidad de código que se necesita escribir. Una tabla puede manejar un gran número de comparaciones, y puede modificarse con más facilidad que una larga serie de instrucciones de comparación, de salto y CALL. Inclusive una tabla puede reconfigurarse en tiempo de ejecución.

Aplicación: máquinas de estado finito

Una *máquina de estado finito* (FSM) es una máquina o programa que cambia de estado con base en cierta entrada. Es bastante sencillo utilizar un gráfico para representar una FSM, la cual contiene cuadros (o círculos) llamados *nodos* y líneas con flechas entre los círculos, llamadas *flancos* (o arcos).

En la siguiente figura se muestra un ejemplo simple. Cada nodo representa un estado del programa, y cada flanco representa una transición de estado a otro. Un nodo se designa como el *estado inicial*, que se muestra en nuestro diagrama con una flecha entrante. Los estados restantes pueden etiquetarse con números o letra. Uno o más estados se designan como *estados terminales*, los cuales se muestran en un borde grueso alrededor del cuadro. Un estado terminal representa a un estado en el que el programa podría detenerse sin producir un error. Una máquina de estado finito es una instancia específica de un tipo más general de estructura conocido como grafo dirigido (o diágrafo). Éste es un conjunto de nodos conectados por flancos que tienen direcciones específicas



.

Los grafos dirigidos tienen muchas aplicaciones útiles en ciencias computacionales, relacionadas con las estructuras dinámicas de datos y las técnicas avanzadas de búsqueda.

Validación de una cadena de entrada

A menudo, los programas que leen los flujos de entrada deben validar su entrada realizando cierta forma de comprobación de errores. Por ejemplo, un compilador de lenguaje de programación puede usar una máquina de estado finito para explorar los programas fuente y convertir las palabras y símbolos en *tokens*, que son objetos como palabras claves, operadores aritméticos e identificadores.

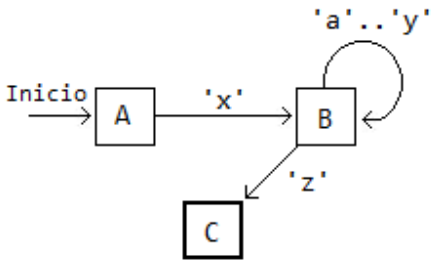
Al utilizar una máquina de estado finito para comprobar la validez de una cadena de entrada, por lo general, se lee la entrada carácter por carácter. Cada carácter se representa mediante un flanco (transición) en el diagrama. Una máquina de estado finito detecta las secuencias de entrada ilegales en una de dos formas:

- El siguiente carácter de entrada no corresponde con ninguna transición del estado actual
- Se llega al fin de la entrada y el estado actual no es un estado terminal

Ejemplo de cadena de caracteres Comprobemos la validez de una cadena de entrada, de acuerdo con las siguientes dos reglas:

- La cadena debe empezar con la letra ‘x’ y terminar con la letra ‘z’
- Entre los caracteres primero y último, puede haber cero o más letras dentro del rango [‘a’..’y’].

El diagrama de la FSM en la siguiente figura describe esta sintaxis. Cada transición se identifica mediante un tipo específico de entrada. Por ejemplo, la transición de estado A al estado B sólo puede lograrse si la letra **x** se lee del flujo de entrada. Una transición del estado B a sí mismo se realiza mediante la introducción de cualquier letra del alfabeto, excepto **z**. una transición de estado B al estado C sólo ocurre cuando la letra **z** se lee del flujo de entrada.



Si se llega al fin del flujo de entrada mientras el programa está en el estado A o B, se produce una condición de error debido a que solo el estado C se marca como estado terminal. La FSM reconoce las siguientes cadenas de entrada:

xabcdefgz

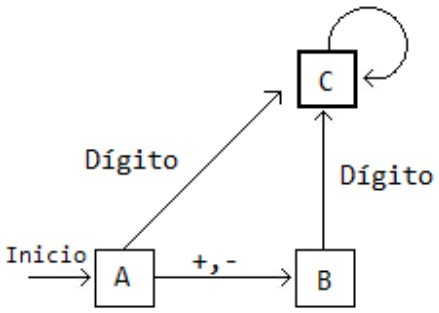
xz

xyyqqrrstuvz

Validación de un entero con signo

En la siguiente figura se muestra una máquina de estado finito para analizar un entero con signo. La entrada consiste de un signo a la izquierda opcional, seguido de una secuencia de dígitos. No hay número máximo establecido de dígitos implicados para el diagrama.

FSM de un entero decimal con signo.



Las máquinas de estado finito se traducen con mucha facilidad a código en lenguaje ensamblador. Cada estado en el diagrama (A, B, C....) se representa mediante una etiqueta. En cada etiqueta se realizan las siguientes acciones:

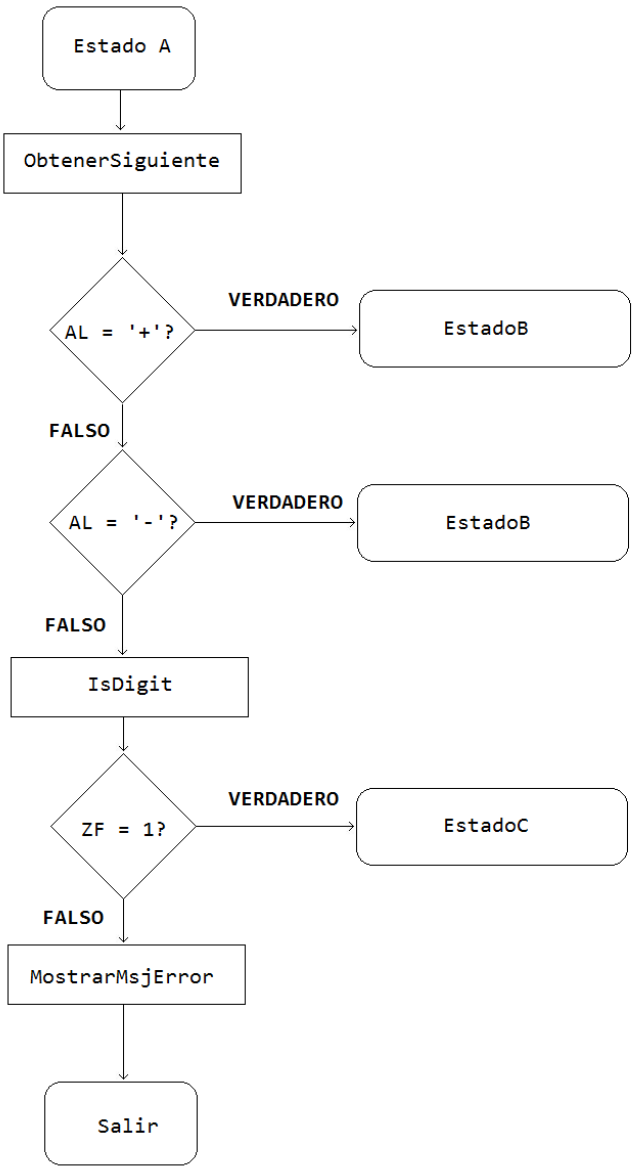
- Una llamada a un procedimiento de entrada que lee el siguiente carácter de la entrada.
- Si el estado es terminal, ay que comprobar para ver si el usuario oprimió la tecla Intro para terminar la entrada.
- Una o más instrucciones verifican cada posible transición que nos lleve hacia fuera de ese estado. Cada comprobación va seguida de una instrucción de salto condicional.

Por ejemplo, en el estado A el siguiente código lee el siguiente carácter de entrada y comprueba una posible transición al estado B:

```
EstadoA:
    call Obtensiguiente     ; lee el siguiente carácter y lo coloca en AL
    cmp al,'+'              ; ¿signo + a la izquierda?
    je EstadoB              ; ir al estadoB
    cmp al,'-'              ; ¿ signo – a la izquierda?
    je EstadoB              ; ir al estadoB
    call IsDigit            ; ZF = 1 si Al contiene un dígito
    jz EstadoC              ; ir al estado C
    call MostrarMsjError    ; se encontró entrada inválida
    jmp Salir
```

Además en el estado A llamamos a **IsDigit**, un procedimiento de la biblioteca de enlace que activa la bandera Cero cuando se lee un dígito numérico de la entrada. Esto hace posible buscar una transición al estado C. Si no se cumple esto, el programa muestra un mensaje de error y termina. El diagrama de flujo de la siguiente figura representa el código adjunto a la etiqueta **EstadoA**.

Diagrama de flujo de la FSM de un entero con signo.



Directivas de decisión

MASM cuenta con directivas de decisión (.IF, .ELSE, .ELSEIF, .ENDIF) que nos facilitan la codificación de la lógica de bifurcación de varias vías. Estas directivas hacen que el ensamblador genere instrucciones CMP y de salto condicional en segundo plano, que se pueden ver en el archivo de listado de salida (*nombreprog.lst*).

He aquí la sintaxis:

```
.IF condición1
     instrucciones
[.ELSEIF condición2
     instrucciones]
[.ELSE
     instrucciones]
.ENDIF
```

Los corchetes muestran que .ELSEIF y .ELSE son opcionales, mientras que .IF y .ENDIF son requeridos. Una *condición* es una expresión booleana que involucra a los mismos operadores que se utilizan en C++ y Java (como <, >, == y !=). La expresión se evalúa en tiempo de ejecución. A continuación se muestran ejemplos de condiciones válidas, usando registros de 32 bits y variables:

eax > 10000h

val1 <= 100

val2 == eax

val3 != ebx

A continuación se muestran ejemplos de condiciones compuestas:

(eax > 0) && (eax > 10000h)

(val1 <= 100) || (val2 <> 100)

Operadores relacionales y lógicos en tiempo de ejecución.

Operador	Descripción
expr1 == *expr2*	Devuelve verdadero cuando *expr1* es igual a *expr2*
expr1 != *expr2*	Devuelve verdadero cuando *expr1* no es igual a *expr2*
expr1 > *expr2*	Devuelve verdadero cuando *expr1* es mayor que *expr2*
expr1 >= *expr2*	Devuelve verdadero cuando *expr1* es mayor o igual que *expr2*
expr1 < *expr2*	Devuelve verdadero cuando *expr1* es menor que *expr2*
expr1 <= *expr2*	Devuelve verdadero cuando *expr1* es menor o igual que *expr2*
!*expr*	Devuelve verdadero cuando *expr* es falsa

expr1 && *expr2*	Realiza un AND lógico entre *expr1* y *expr2*
expr1 || *expr2*	Realiza un OR lógico entre *expr1* y *expr2*
expr1 & *expr2*	Realiza un AND a nivel de bits entre *expr1* y *expr2*
CARRY?	Devuelve verdadero su se activa la bandera Acarreo
OVERFLOW?	Devuelve verdadero si se activa la bandera Desbordamiento
PARITY?	Devuelve verdadero si se activa la bandera Paridad
SIGN?	Devuelve verdadero si se activa la bandera Signo
ZERO?	Devuelve verdadero si se activa la bandera Cero

El uso de las directivas de decisión es controversial, ya que su simplicidad aparente puede engañarnos. Antes de usarlas, debemos asegurarnos de comprender por completo las instrucciones de bifurcación condicional. Además, cuando se ensamble un programa que contiene directivas de decisión, se debe inspeccionar el archivo de listado para asegurarse de que el código generado por MASM es el que nosotros deseamos.

Generación de código ASM Al utilizar directivas de alto nivel tales como .IF y .ELSE, el ensamblador desempeña la función de escritor de código. Por ejemplo, vamos a escribir una directiva .IF que compara a EAX con la variable **val1**:

```
mov eax,6

.IF eax > val1

  mov resultado,1

.ENDIF
```

Se asume que **val1** y **resultado** son enteros sin signo de 32 bits. Cuando el ensamblador lee las líneas anteriores, las expande en las siguientes instrucciones en el lenguaje ensamblador:

```
mov eax,6

cmp eax,val1

jbe @C0001              ; salta en comparación sin signo

mov resultado,1

@C0001:
```

El nombre de etiqueta @C0001 lo crea el ensamblador. Esto hace de una manera que garantice que todas las etiquetas dentro del mismo procedimiento serán únicas.

Comparaciones con signo y sin signo

Al utilizar la directiva .IF para comparar valores, debemos estar conscientes de la forma en que MASM genera los saltos condicionales. Si la comparación involucra a una variable sin signo, se inserta una instrucción de salto condicional sin signo en el código generado. Ésta es una repetición de un ejemplo anterior que compara a EAX con **val1**, una doble palabra sin signo:

```
.data

val1 DWORD 5

resultado DWORD ?

.code

    mov eax,6

    .IF eax > val1

    mov resultado,1

.ENDIF
```

El ensamblador expande esto mediante el uso de la instrucción JBE (salto sin signo):

```
    mov eax,6

    cmp eax,val1

    jbe @C0001          ; salta en comparación sin signo

@C0001:
```

Comparación de un entero con signo Vamos a probar una comparación similar con val2, una doble palabra con signo:

```
.data

val2 SDWORD -1

.code

    mov eax,6

    .IF eax > val2

    mov resultado,1

.ENDIF
```

Ahora el ensamblador genera código mediante la instrucción JLE, el salto basado en comparaciones con signo:

```
    mov eax,6

    cmp eax,val2

    jle @C0001          ; salta en comparación con signo

    mov resultado,1

@C0001:
```

Comparaciones de registros La pregunta que podríamos hacer entonces es, ¿qué ocurre cuando se comparan dos registros? Es evidente que el ensamblador no puede determinar si los valores son con o sin signo:

```
mov eax,6

mov ebx,val2

.IF eax > ebx

  mov resultado,1

.ENDIF
```

Resulta que el ensamblador utiliza de manera predeterminada una comparación sin signo, por lo que la directiva .IF que compara dos registros se implementa usando la instrucción JBE.

Expresiones compuestas

Muchas expresiones booleanas compuestas utilizan los operadores OR y AND lógicos. Al utilizar la directiva .IF, el símbolo || es el operador OR lógico:

```
.IF expresión || oexpresión2

     instrucciones

.ENDIF
```

De manera similar, el símbolo && es el operador AND lógico:

```
.IF expresión1 && expresión2

     instrucciones

.ENDIF
```

En el siguiente programa de ejemplo utilizaremos el operador OR lógico.

Ejemplo: establecerPosiciónCursor

El procedimiento **EstablecerPosiciónCursor**, que se muestra en el siguiente ejemplo, realiza comprobaciones de rango en sus dos parámetros de entrada, DH y DL. La coordenada Y (DH) debe estar entre 0 y 24. La coordenada X (DL) debe estar entre 0 y 79. Si cualquiera de las dos se encuentra fuera del rango, se muestra un mensaje de error:

```
EstablecerPosicionCursor PROC

; Establece la posición del cursor.

; Recibe: DL = coordenada X, DH = coordenada Y

; Comprueba los rangos de DL y DH.

; Regresa: nada

;-----------------------------------------------

.data

MsjCoordXIncorr BYTE “Coordenada X fuera de rango!”,0Dh,0AH,0

MsjCoordYIncorr BYTE “Coordenada Y fuera de rango!”,0Dh,0AH,0

.code

.IF (DL < 0) || (DL > 79)

  mov edx,OFFSET MsjCoordXIncorr

  call WriteString

  jmp salir

.ENDIF

.IF (DH < 0) || (DH > 24)

  mov edx,OFFSET MsjCoordYIncorr

  call WriteString

  jmp salir

.ENDIF

call Gotoxy

salir:
```

```
ret

EstablecerPosicionCursor ENDP
```

Ejemplo de inscripción universitaria

Supongamos que un estudiante universitario desea inscribirse en ciertos cursos. Utilizaremos dos criterios para determinar si el estudiante puede registrarse o no: El primero es el promedio de calificaciones de la persona, con base en una escala de 0 a 400, en donde 400 es la mayor calificación posible. El segundo es el número de créditos que desea tomar la persona. Puede utilizarse una estructura de bifurcación de varias vías, en la que se utilicen las directivas .IF, .ELSEIF y .ENDIF. A continuación se muestra un ejemplo:

```
.data

VERDADERO = 1

FALSO = 0

promedioCalif WORD 275 ; valor de prueba

créditos  WORD 12 ; valor de prueba

SePuedeRegistrar BYTE ?

.code

    mov SePuedeRegistrar,FALSO

    .IF promedioCalif > 350

      mov SePuedeRegistrar,VERDADERO

    .ELSEIF (promedioCalif > 250) && (créditos <= 16)

      mov SePuedeRegistrar,VERDADERO

    .ELSEIF (créditos <= 12)

      mov SePuedeRegistrar,VERDADERO

.ENDIF
```

Directiva .REPEAT y .WHILE

La directiva .REPEAT y .WHILE ofrecen alternativas para escribir nuestros propios ciclos, con instrucciones CMP y de salto condicional. Permiten las expresiones condicionales (ver tabla: Operadores relacionales y lógicos en tiempo de ejecución). La directiva .REPEAT ejecuta el cuerpo del ciclo antes de evaluar la condición en tiempo de ejecución que va después de la directiva .UNTIL:

```
.REPEAT
     instrucciones
.UNTIL condición
```

La directiva .WHILE evalúa la condición antes de ejecutar el ciclo:

```
.WHILE condición
     instrucciones
.ENDW
```

Ejemplos: las siguientes instrucciones muestran los valores de 1 al 10, usando la directiva .WHILE:

```
mov eax,0
.WHILE eax < 10
     inc eax
     call WriteDec
     call Crlf
.ENDW
```

Las siguientes instrucciones muestran los valores del 1 al 10, usando la directiva .REPEAT:

```
mov eax,0
.REPEAT
     inc eax
     call WriteDec
     call Crlf
.UNTIL eax == 10
```

Ejemplo: ciclo que contiene una instrucción IF

Anteriormente en este capítulo, mostramos como escribir código en el lenguaje ensamblador para una instrucción IF anidada dentro de un ciclo WHILE. He aquí el seudocódigo:

```
while(op1 < op2)
{
```

```
     op1++;
     if(op1 == op3)
      X = 2;
    else
      X = 3
}
```

A continuación se muestra una implementación del seudocódigo, utilizando las directivas .WHILE e .IF, Como **op1**, **op2** y **op3** son variables, se mueven en los registros para evitar tener dos operandos de memoria en cualquier instrucción:

```
.data
X DWORD 0
op1 DWORD 2 ; datos de prueba
op2 DWORD 4 ; datos de prueba
op3 DWORD 5 ; datos de prueba
.code
    mov eax,op1
    mov ebx,op2
    mov ecx,op3
   .WHILE eax < ebx
     inc eax
     .IF eax == ecx
        mov X,2
     .ELSE
       mov X,3
  .ENDIF
.ENDW
```

7 Aritmética de enteros

Instrucciones de desplazamiento y rotación

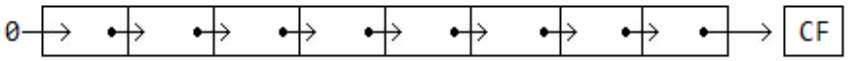
Junto con las instrucciones a nivel de bits que presentamos en el capítulo 6, las instrucciones de desplazamiento son las más características del lenguaje ensamblador. *Desplazar* significa mover bits a la derecha y a la izquierda dentro de un operando. Intel proporciona un conjunto bastante completo de instrucciones en esta área, todas las cuales afectan a las banderas Desbordamiento y Acarreo.

Instrucciones de desplazamiento y rotación

SHL	Desplazamiento a la izquierda
SHR	Desplazamiento a la derecha
SAL	Desplazamiento aritmético a la izquierda
SAR	Desplazamiento aritmético a la derecha
ROL	Rotación a la izquierda
ROR	Rotación a la derecha
RCL	Rotación con acarreo a la izquierda
RCR	Rotación con acarreo a la derecha
SHLD	Desplazamiento de doble precisión a la izquierda
SHRD	Desplazamiento de doble precisión a la derecha

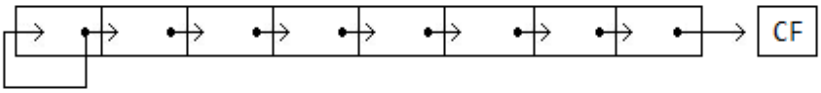
Desplazamientos lógicos y desplazamientos aritméticos

Existen dos formas de desplazar los bit de un operando. La primera, el *desplazamiento lógico*, llena la nueva posición de bit creada con cero. El siguiente diagrama, un byte se desplaza en forma lógica una posición a la derecha. (El bit se le asigna un 0):

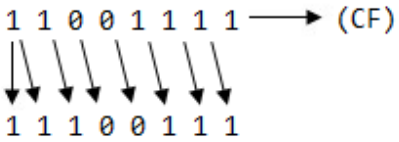


Supongamos que ejecutamos un solo desplazamiento lógico a la derecha en el valor binario 11001111, lo cual produce 01100111. El bit inferior se desplaza hacia la bandera Acarreo

Hay otro tipo de desplazamiento, conocido como desplazamiento aritmético. La nueva posición del bit creada se llena con una copia del bit del signo del número original:

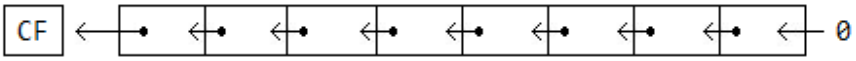


Por ejemplo, el número binario 11001111 tiene un 1 en el bit del signo. Cuando se desplaza aritméticamente 1 bit a la derecha, se convierte en 11100111:

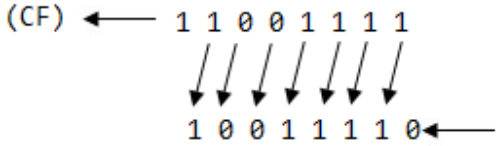


Instrucción SHL

La instrucción SHL (desplazamiento a la izquierda) realiza un desplazamiento lógico a la izquierda en el operando de destino, llenando el bit inferior con 0. El bit superior se mueve hacia la bandera de Acarreo, y el bit que estaba en la bandera Acarreo se pierde:



El número binario 11001111 desplazado 1 bit a la izquierda se convierte en 10011110:



El primer operando de SHL es el destino, y el segundo es la cuenta de desplazamiento

SHL *destino*, *cuenta*

A continuación presentamos los tipos de operandos que permite esta instrucción:

SHL *reg*,*imm8*

SHL *mem*,*imm8*

SHL *Reg*, *CL*

SHL *mem*, *CL*

Los procesadores Intel 8086/8088 requieren que *imm8* sea igual a 1. Del procesador Intel 80286 en adelante, *imm8* pude ser cualquier entero entre 0 y 255. En cualquier procesador Intel, CL puede contener una cuenta de desplazamiento. Los formatos que se muestran aquí también se aplican a las instrucciones SHR, SAL, SAR, ROR, ROL, RCR Y RCL.

Ejemplos En las siguientes instrucciones, BL se desplaza una vez a la izquierda. El bit superior se copia a la bandera Acarreo y la posición del bit inferior se asigna un cero:

```
mov bl,8Fh              ; BL = 10001111b

shl bl,1                ; CF,BL = 1,0001110b
```

Múltiples desplazamientos Cuando un valor se desplaza varias veces, la bandera Acarreo contiene el último bit que se desplazó hacia afuera del bit más significativo (MSB). En el siguiente ejemplo, el bit 7 no termina en la bandera Acarreo, ya que lo sustituye el bit 6 (un cero):

```
mov al,10000000b

shl al,2                 ; CF = 0
```

Lo mismo ocurre cuando se realizan desplazamientos a la derecha.

Multiplicación rápida SHL puede realizar multiplicaciones de alta velocidad, por potencias de 2. Si se desplaza cualquier operando a la izquierda por *n* bits, esto equivale a multiplicar el operando por 2^*n*. Por ejemplo, si se desplaza el número 5 a la izquierda por 1 bit, se produce el producto de 5*2:

```
mov dl,5

shl dl,1
```

Antes: `0 0 0 0 0 1 0 1 = 5`

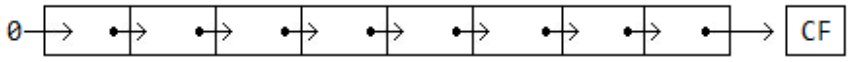
Después: `0 0 0 0 1 0 1 0 = 10`

Si el número 10 decimal se desplaza 2 bits a la izquierda, el resultado es el mismo que si se multiplica 10 por 2^2:

```
mov dl,10

shl dl,2                ; (10 * 4) = 40
```

Instrucción SHR

La instrucción SHR (desplazamiento a la derecha) realiza un desplazamiento lógico a la derecha en el operando de destino, sustituyendo el bit superior con un 0. El bit inferior se copia a la bandera Acarreo, y el bit que estaba en la bandera Acarreo se pierde:



SHR utiliza los mismos formatos de instrucciones que SHL. En el siguiente ejemplo, el 0 del bit inferior en AL se copia a la bandera Acarreo, y el bit superior en AL se borra:

```
mov al,0D0h             ; AL = 11010000b

shr al,1                ; AL = 01101000b, CF = 0
```

Múltiples desplazamientos En una operación con varios desplazamientos , el último bit que se desplaza hacia fuera de la posición 0 termina en la bandera Acarreo:

```
mov al,00000010b

shr al,2                ; AL = 00000000b, CF = 1
```

División rápida Si se desplaza un entero sin signo a la derecha por *n* bits, esto equivale divir el operando entre 2^*n*. Por ejemplo, vamos a dividir 32 entre 2^1, lo cual produce 16:

```
mov dl,32

shr dl,1
```

Antes: `0 0 1 0 0 0 0 0 = 32`

Después: `0 0 0 1 0 0 0 0 = 16`

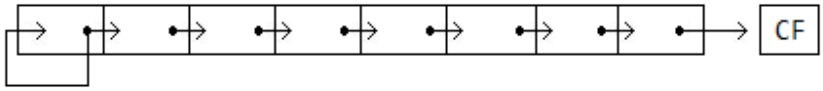
En el siguiente ejemplo, 64 se divide entre 2^3:

```
mov al,01000000b        ; AL = 64

shr al,3                ; divide entre 8, AL = 00001000b
```

La división de números con signo mediante los desplazamientos se realiza con la instrucción SAR, ya que preserva el bit de signo del número.

Instrucción SAL y SAR

SAL (desplazamiento aritmético a la izquierda) es idéntica a la instrucción SHL. La instrucción SAR (desplazamiento aritmético a la derecha) realiza un desplazamiento aritmético a la derecha en su operando de destino:



Los operandos para SAL y SAR son idénticos a los operandos para SHL y SHR. El desplazamiento puede repetirse, con base en el contador en el segundo operando:

SAR *destino,cuenta*

El Siguiente ejemplo muestra como SAR duplica el bit de signa. AL es negativo antes y después que se desplaza a la derecha:

```
mov al,0F0h             ; AL = 11110000b (-16)

sar al,1                ; AL = 11111000b (-8), CF = 0
```

División con signo Podemos dividir un operando con signo entre potencia de 2, mediante el uso de la instrucción SAR. En el siguiente ejemplo, -128 se divide entre 2^3. El cociente es -16:

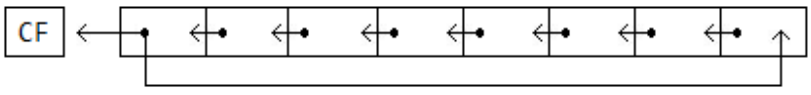
```
mov dl,-128             ; DL = 10000000b

sar dl,3                ; DL = 11110000b
```

Extensión de AX con signo hacia EAX Supongamos que AX contiene un entero con signo y deseamos extender su signo hacia EAX. El primero se desplaza EAX 16 bits a la izquierda, y luego se desplaza aritméticamente 16 bits a la derecha:

```
mov eax,-128            ; EAX = ????FF80h

shl eax,16              ; EAX = FF800000h

sar eax,16              ; EAX = FFFFFF80h
```

Instrucción ROL

La instrucción ROL (rotación a la izquierda) desplaza cada bit a la izquierda. El bit superior se copia a la bandera Acarreo y a la posición del bit inferior. El formato de instrucción es el mismo que para SHL:



En la rotación no se pierden bits. Un bit que se rota hacia un extremo de un número aparece en el otro extremo. Observemos en el siguiente ejemplo cómo el bit superior se copia tanto a la bandera Acarreo como a la posición del bit 0:

```
mov al,40h              ; AL = 01000000b

rol al,1                ; AL = 10000000b, CF = 0

rol al,1                ; AL = 00000001b, CF = 1

rol al,1                ; AL = 00000010b, CF = 0
```

Rotaciones múltiples Cuando se utiliza una cuenta de rotaciones mayor que 1, la bandera Acarreo contiene el último bit que se rotó hacia fuera de la posición del bit más significativo:

```
mov al,00100000b

rol al,3                ; CF = 1, AL = 00000001b
```

Intercambio de grupos de bits Se puede usar ROL para intercambiar las mitades superior (bits 4-7) e inferior (bits 0-3) de un byte. Por ejemplo, si se rota el número 26h cuatro bits en cualquier dirección, se convierte en 62h:

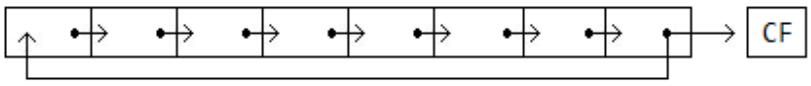
```
mov al,26h

rol al,4                ; AL = 62h
```

Al rotar un entero de varios bytes por 4 bits, el efecto es que se rota cada dígito hexadecimal una posición a la derecha o a la izquierda. Aquí, por ejemplo rotamos en forma repetida el número 6A4Bh 4 bits a la izquierda, para volver a quedarnos con el valor original:

```
mov ax,6A4Bh

rol ax,4                ; AX = A4B6h

rol ax,4                ; AX = 4B6Ah

rol ax,4                ; AX = B6A4h

rol ax,4                ; AX = 6A4Bh
```

Instrucción ROR

La instrucción ROR (rotación a la derecha) desplaza cada bit a la derecha y copia al bit inferior en la bandera Acarreo y en la posición del bit superior. El formato de la instrucción es el mismo que para SHL:



En los siguientes ejemplos, observe cómo se copia el bit inferior tanto como en la bandera Acarreo como en la posición del bit superior del resultado:

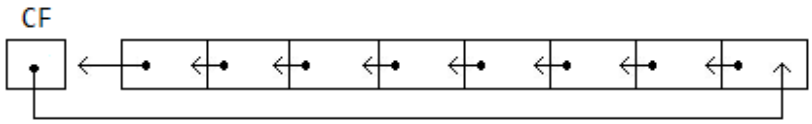
```
mov al,01h              ; AL = 00000001b

ror al,1                ; AL = 10000000, CF = 1

ror al,1                ; AL = 01000000, CF = 0
```

Múltiples rotaciones Al utilizar una cuenta de rotaciones mayor que 1, la bandera Acarreo contiene el último bit que se rotó hacia fuera de la posición del bit más significativo:

```
mov al,00000100b

ror al,3                ; AL =10000000, CF = 1
```

Instrucciones RCL y RCR

La instrucción RCL (rotación a la izquierda con acarreo) desplaza cada bit a la izquierda, copia la bandera Acarreo al bit menos significativo (LSB) y copia el bit más significativo (MSB) a la bandera Acarreo:



Si imaginamos la bandera Acarreo como un bit adicional que se agrega al extremo superior del operando RCL se ve como una operación de rotación a la izquierda. En el siguiente ejemplo, la instrucción CLC borra la bandera Acarreo. La primera instrucción RCL mueve el bit superior de BL hacia la bandera Acarreo y desplaza los otros bits a la izquierda. La segunda instrucción RCL mueve la bandea Acarreo hacia la posición del bit inferior y desplaza los otros bits a la izquierda:

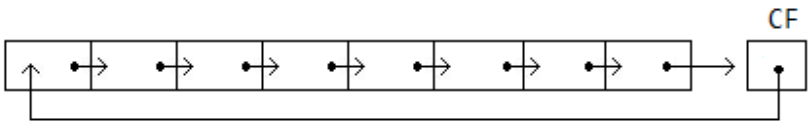
```
clc                     ; CF = 0

mov bl,88h              ; CF,BL = 0 10001000b

rcl bl,1                ; CF, BL = 1 00010000b

rcl bl,1                ; CF,BL = 0 00100001b
```

Recuperación de un bit de la bandera Acarreo RCL puede recuperar un bit que se haya desplazado previamente a la bandera Acarreo. El siguiente ejemplo comprueba el bit inferior de **valprueba**, desplazando su bit inferior hacia la bandera Acarreo. Si el bit inferior de

valprueba es 1, se realiza un salto; si el bit inferior es 0, RCL restaura el número a su valor original:

```
.data

valrpueba BYTE 01101010b

.code

shr valprueba,1          ; desplaza LSB hacia la bandera Acarreo

jc salir                 ; termina si se activa la bandera Acarreo

rcl valprueba,1          ; en caso contrario, restaura el número
```

Instrucción RCR La instrucción RCR (Rotación a la derecha con acarreo) desplaza cada bit a la derecha, copia la bandera Acarreo al bit más significativo, y copia el bit menos significativo a la bandera Acarreo:



Como en el caso de RCL, es útil visualizar el entero en esta figura como un valor de 9 bits, con la bandera Acarreo a la derecha del bit menos significativo.

En el siguiente ejemplo, STC activa la bandera Acarreo antes de rotarla hacia el MSB y antes de rotar el LSB hacia la bandera Acarreo:

```
stc                      ; CF = 1

mov ah,10h               ; AH, = CF = 00010000 1

rcr ah,1                 ; AH, = CF = 1001000 0
```

Desbordamiento con signo

La bandera Desbordamiento se activa cuando al desplazar o rotar un entero con signo por una posición de bit se genera un valor fuera del rango de enteros con signo para el operando. Dicho de otra forma, se invierte el signo del número. En el siguiente ejemplo, un entero positivo (+127) se vuelve negativo (-2) cuando se rota a la izquierda:

```
mov al,+127              ; AL = 01111111b

rol al,1                 ; OF 1, AL = 11111111b
```

De manera similar, cuando el número -128 se desplaza una posición a la derecha, se activa la bandera Desbordamiento. El resultado en AL (+64) tiene el signo opuesto:

```
mov al,-18                ; AL = 10000000b

shr al,1                  ; OF = 1, AL = 01000000b
```

El valor de la bandera Desbordamiento es indefinido cuando la cuenta de desplazamiento o rotaciones es mayor que 1.

Instrucciones SHLD/SHRD

Las instrucciones SHLD y SHRD se introdujeron con el Intel386. La instrucción SHLD (desplazamiento doble a la izquierda) desplaza un operando de destino de cierto número de bits a la izquierda. Las posiciones de bits que se abren debido al desplazamiento se llenan con los bits más significativos del operando de origen.

Este operando no se ve afectado, pero las banderas Signo, Cero, Auxiliar, Paridad y Acarreo sí:

SHLD *destino*, *origen*, *cuenta*

La instrucción SHRD (desplazamiento doble a la derecha) desplaza un operando de destino cierto número de bits a la derecha. Las posiciones de bits que se abren debido al desplazamiento se llenan con los bits menos significativos del operando de origen:

SHRD *destino*, *origen*, *cuenta*

Los siguientes formatos de instrucciones se aplican tanto a SHLD como a SHRD. El operando de *destino* puede ser un registro u operando de memoria, mientras que el operando de *origen* debe ser un registro. El operando *cuenta* puede ser el registro CL o un operando inmediato de 8 bits:

SHLD *reg16*,*reg16*,*CL/imm8*

SHLD *mem16*,*reg16*,*CL/imm8*

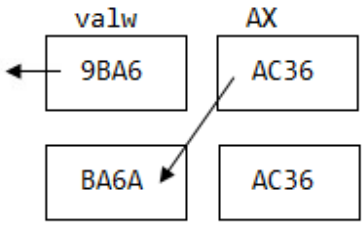
SHLD *reg32*,*reg32*,*CL/imm8*

SHLD *mem32*,*reg32*,*CL/imm8*

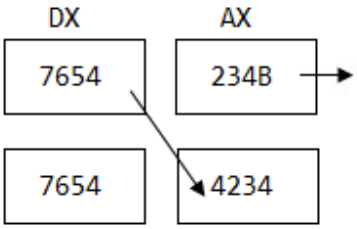
Ejemplo 1 Las siguientes instrucciones desplazan **valw** 4 bits a la izquierda, e insertan los 4 bits superiores de AX en las 4 posiciones de bits inferiores de **valw**:

```
.data

valw WORD 9BA6h

.code

mov ax,0AC36h

shld valw,ax,4            ; valw = BA6Ah
```

El movimiento de datos se muestra en la siguiente figura:



Ejemplo 2 En el siguiente ejemplo, AX se desplaza 4 bits a la derecha, y los 4 bits inferiores de DX se desplazan hacia las 4 posiciones superiores de AX:




```
mov ax,234Bh

mov dx,7654h

shrd ax,dx,4              ; AX = 4234h
```

SHLD y SHRD pueden usarse para manipular imágenes de mapas de bits, cuando varios grupos de bits deben desplazarse a la izquierda y a la derecha para reposicionar imágenes en la pantalla. Otra aplicación potencial es el cifrado de datos, en donde el algoritmo de cifrado requiere desplazamiento de bits. Por último, las dos instrucciones pueden usarse al realizar operaciones rápidas de multiplicación y división con enteros muy grandes.

Aplicaciones de desplazamiento y rotación

Desplazamiento de varias dobles palabras

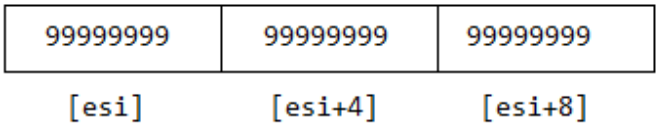
Para desplazar un entero con precisión extendida, hay que dividirlo en un arreglo de bytes, palabras o dobles palabras. Una manera común de almacenar el número en memoria es con el valor de menor orden en la dirección más baja (a la cual se le conoce como orden *Little-endian*). Los siguientes pasos nos mostrarán cómo desplazar un arreglo de este tipo a la derecha, usando un arreglo de dobles palabras como ejemplo:

```
TamArreglo = 3

.data

arreglo DWORD TamArreglo DUP(?)
```

1. Establecer ESI con el desplazamiento del arreglo.

2. Desplazar la doble palabra de orden superior en **[ESI+8]** a la derecha, copiando en forma automática su bit inferior a la bandera Acarreo.
3. Desplazar el valor en **[ESI+4]** a la derecha. Su bit superior se llena de manera automática de la bandera Acarreo, y su bit inferior se copia en la nueva bandera Acarreo.
4. Desplazar la doble palabra de orden inferior en **[ESI+0]** a la derecha. Su bit superior se llena de la bandera Acarreo y su bit inferior se copia a la nueva bandera Acarreo.

La siguiente figura muestra el contenido del arreglo y las referencias indirectas:



El programa llamado DespMulti.asm implementa el siguiente código. Utilizando RCR en este ejemplo, pero podríamos usar la instrucción SHRD en su lugar:

```
.data

TamArreglo = 3

arreglo DWORD TamArreglo dup(99999999h) ; 1001 1001…

.code

mov esi,0

shr arreglo[esi+8],1       ; doble palabra más alta

rcr arreglo[esi+4],1       ; doble palabra media, incluye bandera Acarreo

rcr arreglo[esi],1         ; doble palabra baja, incluye la bandera
Acarreo
```

La salida del programa muestra los números binarios, antes y después del desplazamiento:

```
1001 1001 1001 1001 1001 1001 1001 1001 1001 1001…(etc.)

0100 1100 1100 1100 1100 1100 1100 1100 1100 1100…(etc.)
```

Multiplicación binaria

Las instrucciones de multiplicación binaria de la familia IA-32 (MUL e IMUL) se consideran lentas, en relación con otras instrucciones de máquina. A menudo, los programadores de ensamblador buscan mejores formas de realizar la multiplicación binaria, y es evidente que el desplazamiento de bits es superior. La instrucción SHL realiza la multiplicación sin signo con

eficiencia, cuando el multiplicador es una potencia de 2. Al desplazar un entero sin signo *n* bits a la izquierda, se multiplica por 2^*n*. Cualquier otro multiplicador puede expresarse como la suma de potencias de 2. Por ejemplo, para multiplicar el valor de EAX sin signo por 36, podemos escribir el 36 como 2^5 + 2^2 y utilizar la propiedad distributiva de la multiplicación:

EAX * 36 = EAX * (32 + 4)

= (EAX * 32) + (EAX * 4)

La siguiente figura muestra la multiplicación 123 * 36, que produce 4428, el producto:

```
    01111011  123
x   00100100  36
  01111011    123 SHL 2
+ 01111011    123 SHL 5
0001000101001100  4428
```

Los bits 2 y 5 se activan en el multiplicador (36) y también son contadores de desplazamiento requeridos. El siguiente código implementa esta multiplicación, mediante registros de 32 bits:

```
.code

mov eax,123

mov ebx,eax

shl eax,5           ; multiplica por 2^5

shl ebx,2           ; multiplica por 2^2

add eax,ebx         ; suma los productos
```

Visualización de bits binarios

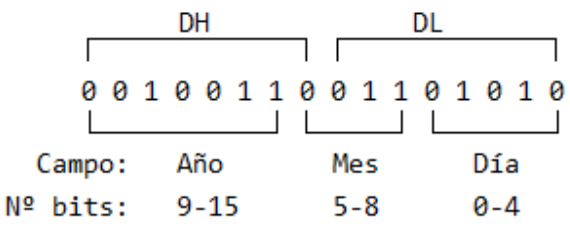
Una tarea común de programación es convertir un entero binario en cadena ASCII binaria, para poder visualizarla en pantalla, la instrucción SHL es útil para esto, ya que copia el bit más alto de un operando a la bandera Acarreo, cada vez que el operando se desplaza a la izquierda. El siguiente procedimiento BinAAsc es una implementación simple:

```
;-----------------------------------------------------------------

BinAAsc PROC

;

; Convierte un entero binario de 32 bits a ASCII binario.

; Recibe: EAX = entero binario, ESI apunta al búfer
```

```
; Devuelve: búfer lleno de dígitos ASCII binarios
;-----------------------------------------------------------------
push ecx
push esi
mov ecx,32                      ; número de bits en EAX
L1: shl eax,1          ; desplaza el bit superior hacia la bandera Acarreo
mov BYTE PTR [esi],'0'          ; elije 0 como dígito predeterminado
jnc L2                          ; si no hay Acarreo, salta a L2
mov BYTE PTR [esi],'1'          ; en caso contrario, mueve 1 al búfer
L2: inc esi                     ; siguiente posición del búfer
loop L1                         ; desplaza otro bit a la izquierda
pop esi
pop ecx
ret
BinAASC ENDP
```

Aislamiento de campos de datos de archivos de MS-DOS

Con frecuencia, un byte o palabra contiene más de un campo, por lo que es necesario extraer secuencias de bits llamadas *cadenas de bits*. Por ejemplo, en modo direccionamiento real, la función 57h de DOS devuelve la estampa de fecha de un archivo en DX. (La estampa de fecha muestra la fecha de última modificación que se realizo a ese archivo). Los bits del 0 al 4 representan un número de día entre 1 y 31, los bits de 5 al 8 son el número de mes, y los bits del 9 al 15 guardan el número del año. Supongamos que la fecha de la última modificación de un archivo es marzo 10,1999. La estampa de fecha del archivo aparecería como se muestra a continuación en el registro DX (el número de año es relativo a 1980):



Para extraer un solo campo, hay que desplazar sus bits hacia la parte inferior de un registro y borrar las posiciones de los bits irrelevantes. El siguiente ejemplo de código extrae el número del día. Sacando una copia de DL y enmascarando los bits que no pertenecen al campo:

```
mov al, dl                  ; saca una copia de DL

and  al,00011111b          ; borra los bits 5-7

mov dia, al                 ; guarda en día
```

Para extraer el número del mes, desplazamos los bits del 5 al 8 en la parte baja de AL, antes de enmascarar todos los demás bits. Después, AL se copia a una variable:

```
mov ax,dx                   ; saca una copia de DX

shr ax,5                    ; desplaza 5 bits a la derecha

and al,00001111b            ; borra los bits 4-7

mov mes,al                  ; guarda en mes
```

El número de año (bit del 9 al 15) se encuentra completamente dentro del registro DH. Lo copiamos a AL y lo desplazamos 1 bit a la derecha:

```
mov al,dh                   ; saca una copia de DH

shr al,1                    ; desplaza una posición a la derecha

mov ah,0                    ; borra AH y lo deja en ceros

add eax,1980                ; el año es relativo a 1980

mov anio,ax                 ; guarda en el año
```

Instrucciones de multiplicación y división

Las instrucciones IMUL e MUL realizan operaciones de multiplicación de enteros con y sin signo, respectivamente. La instrucción DIV realiza la división de enteros sin signo, e IDIV realiza la división de enteros con signo.

Instrucción MUL

La instrucción MUL (multiplicación sin signo) viene en tres versiones: la primera multiplica un operando de 8 bits por AL, la segunda multiplica un operando de 16 bits por AX y la tercera multiplica un operando de 32 bits por EAX. El multiplicador y el multiplicativo son del mismo tamaño, y el producto es el doble de su tamaño. Los tres formatos aceptan operandos de registro y de memoria, pero no operandos inmediatos:

MUL r/m8

MUL r/m16

MUL r/m32

El operando individual es el multiplicador. La siguiente tabla muestra el multiplicando predeterminado y el producto, dependiendo del tamaño del multiplicador. Como el operando de destino es del doble del tamaño del multiplicando y del multiplicador, no puede ocurrir desbordamiento. MUL activa las banderas Acarreo y Desbordamiento si la mitad superior del producto no es igual a cero. Por lo general, la bandera Acarreo se utiliza para aritmética sin signo, por lo que aquí nos enfocamos en eso. Por ejemplo, cuando AX se multiplica por un operando de 16 bits, el producto se almacena en DX:AX La bandera Acarreo se activa si DX no es igual a cero.

Operandos de MUL.

Multiplicando	Multiplicador	Producto
AL	r/m8	AX
AX	r/m16	DX:AX
EAX	r/m32	EDX:EAX

Una buena razón para comprobar la bandera Acarreo después de ejecutar MUL, es para saber si la mitad superior del producto puede ignorarse sin riesgos.

Ejemplos de MUL

Las siguientes instrucciones multiplican AL por BL, almacenando el producto en AX. La bandera Acarreo se borra (CF = 0) debido a que AH (la mitad superior del producto) es igual a cero:

```
mov al,5h

mov bl,10h

mul bl                    ; AX = 50h, CF = 0
```

Las siguientes instrucciones multiplican el valor de 16 bits 2000h por 100h. CF = 1, ya que la parte superior del producto en DX no es igual a cero:

```
.data

val1 WORD 2000h

val2 WORD 0100h

.code

mov ax,val1               ; AX = 2000h
```

```
mul val2                    ; DX:AX = 00200000h, CF = 1
```

Las siguientes instrucciones multiplican 12345h por 1000h, y el producto es de 64 bits. CF = 0, ya que EDX es igual a cero:

```
mov eax,12345h

mov ebx,1000h

mul ebx                     ; EDX:EAX = 0000000012345000h, CF = 0
```

Instrucción IMUL

La instrucción IMUL (multiplicación con signo) realiza la multiplicación de enteros con signo, preservando el signo del producto. El conjunto de instrucciones IA-32 soporta tres formatos para esta instrucción: un operando, dos operandos y tres operandos. En el formato de un operando, el multiplicador y el multiplicando son del mismo tamaño y el producto es del doble de su tamaño. (Los procesadores 8086/8088 sólo soportan el formato de un operando).

Formatos de un operando Los formatos de un operando almacenan el producto en el acumulador (AX, DX:AX o EDX:EAX):

IMUL r/m8 ; AX = AL * r/m byte

IMUL r/m16 ; DX:AX = AX * r/m palabra

IMUL r/m32 ; EDX:EAX = EAX * r/m doble palabra

Como en el caso de MUL, el tamaño de almacenamiento del producto hace que el desbordamiento sea imposible en la instrucción IMUL de un operando. Las banderas Acarreo y Desbordamiento se activan si la mitad superior del producto no es una extensión del signo de la mitad inferior. Se puede utilizar esta información para decidir si se debe ignorar o no la mitad superior del producto.

Formato de dos operandos La versión de dos operandos de esta instrucción almacena el producto en el primer operando. El primer operando debe ser un registro. El segundo operando puede ser un registro, operando de memoria o valor inmediato. A continuación se muestran los formatos de 16 bits:

IMUL r16,r/m16

IMUL r16,imm8

IMUL r16,imm16

A continuación se presentan los formatos de 32 bits, para demostrar que el multiplicador puede ser un registro de 32 bits, un operando de memoria de 32 bits o un valor inmediato (de 8 o 32 bits):

IMUL r32,r/m32

IMUL r32,imm8

IMUL r32,imm32

Los formatos de dos operandos truncan el producto a la longitud del destino. Si se pierden dígitos significativos, se activan las banderas Desbordamiento y Acarreo. Se debe verificar una de estas banderas antes de realizar una operación IMUL con dos operandos.

Formato de tres operandos Los formatos de tres operandos almacenan el producto en el primer operando. Un registro de 16 bits u operando de memoria puede multiplicarse por un valor inmediato de 8 o de 16 bits:

IMUL r16,r/m16,imm8

IMUL r16,r/m16,imm16

Un registro de 32 bits u operando de memoria puede multiplicarse por un valor inmediato de 8 o 32 bits:

IMUL r32,r/m32,imm8

IMUL r32,r/m32,imm32

Si se pierden dígitos significativos, se activan las banderas Desbordamiento y Acarreo. Se debe revisar una de estas banderas después de realizar una operación IMUL con tres operandos.

Multiplicación sin signo Los formatos de IMUL de dos y tres operandos también pueden utilizarse para la multiplicación sin signo, ya que la mitad inferior del producto es igual para los números con y sin signo. Hay una pequeña desventaja al hacer esto: las banderas Acarreo y Desbordamiento no indican si la mitad superior del producto es Cero.

Ejemplos de IMUL

Las siguientes instrucciones multiplican 48 por 4, produciendo +192 en AX. En el producto, AH no es una extensión del signo AL, por lo que ocurre un desbordamiento con signo:

```
mov al,48

mov bl,4

imul bl             ; AX = 00C0h, OF = 1
```

Las siguientes instrucciones multiplican -4 por 4, produciendo -16 en AX. AH es una extensión del signo de AL en el producto, por lo que la bandera Desbordamiento se borra:

```
mov al,-4

mov bl,4

imul bl             ; AX =FFF0h, OF = 0
```

Las siguientes instrucciones multiplicando 48 por 8, produciendo -192 en DX:AX es una extensión del signo de AX, por lo que no hay desbordamiento con signo:

```
mov ax,48

mov bx,4

imul bx             ; DX:AX = 000000C0h, OF = 0
```

Las siguientes instrucciones realizan una multiplicación de 32 bits con signo (482424 * -423), produciendo -2,040,308,352 en EDX:EAX. EDX es una extensión de signo de EAX, por lo que la bandera de Desbordamiento se borra:

```
mov eax,+482424

mov ebx,4

imul ebx            ; EDX:EAX = FFFFFFF86635D80h, OF = 0
```

Las siguientes instrucciones demuestran los formatos de dos operandos:

```
.data

word1 SWORD 4

dword1 SWORD 4

.code

mov ax,-16          ; AX = -16

mov bx,2            ; BX = 2

imul bx,eax         ; BX = -32

imul bx,2           ; BX = -64

imul bx,word1       ; BX = -256

mov eax,-16         ; EAX = -16

mov ebx,2           ; EBX = 2

imul ebx,eax        ; EBX = -32

imul ebx,2          ; EBX = -64

imul ebx,dword1     ; EBX = -256
```

Las siguientes instrucciones de dos operandos demuestra un desbordamiento con signo, ya que -64000 no puede ajustarse dentro de un operando de destino de 16 bits:

```
mov ax,-32000

imul ax,2                   ; OF = 1
```

Las siguientes instrucciones demuestran los operandos de tres formatos, incluyendo un ejemplo de desbordamiento con signo:

```
.data

word1 SWORD 4

dword1 SDWORD 4

.code

imul bx,word1,-16           ; BX = -64

imul ebx,dword, -16         ; EBX = -64

mul ebx,dword1,-2000000000 ; OF = 1
```

Evaluación del rendimiento de las operaciones de multiplicación

Ahora que hemos visto como se realiza la multiplicación mediante el desplazamiento de bits y las instrucciones MUL e IMUL estándar, es interesante comparar su rendimiento relativo. Los siguientes procedimientos multiplican EAX por 36, usando los dos métodos:

```
mult_por_desplazamiento PROC

;

; Multiplica EAX por 36 usando SHL, CUENTA_CICLO veces.

    mov ecx,CUENTA_CICLO

L1:push eax              ; guarda el valor original de EAX

    mov ebx,eax

    shl eax,5

    shl ebx,2

    add eax,ebx

    pop eax              ; restaura EAX

    loop L1

    ret

mult_por_desplazamiento ENDP
```

```
mult_por_MUL PROC
;
; Multiplica EAX por 36 usando MUL, CUENTA_CICLO veces.
    mov ecx,CUENTA_CICLO
L1:push eax              ; guarda EAX original
    mov ebx,36
    mul ebx
    pop eax              ; restaura EAX
    loop L1
    ret
mult_por_MUL ENDP
```

Vamos a llamar a **mult_por_desplazamiento** un gran número de veces y vamos a registrar el tiempo de ejecución:

```
.data
CUENTA_CICLO = 0FFFFFFFFh
.data
ValInt DWORD 5
tiempoInicial DWORD ?
.code
call GetMseconds ; obtiene tiempo inicial
mov tiempoInicial,eax
mov eax,ValInt
call mult_por_desplazamiento ; multiplica ahora
call GetMseconds ; obtiene tiempo final
sub eax,tiempoInicial
call WriteDec ; muestra el tiempo transcurrido
```

Suponiendo que llamamos a **mult_por_MUL** de la misma forma, los tiempos resultantes en un Pentium 4 de 4 GHz son evidentes: El método SHL se ejecuta en 6.078 segundos y el método con MUL se ejecuta en 20.718 segundos. En otras palabras, ¡el uso de la instrucción MUL hace que el cálculo sea un 241 por ciento más lento!

Instrucción DIV

La instrucción DIV (división sin signo) realiza la división de enteros sin signo de 8 bits, 16 bits y 32 bits. El registro individual u operando de memoria es el divisor. Los formatos son:

DIV r/m8

DIV r/m16

DIV r/m32

La siguiente tabla muestra la relación entre el dividendo, el divisor, el cociente y el residuo:

Dividendo	Divisor	Cociente	Residuo
AX	r/m8	AL	AH
DX:AX	r/m16	DX	AX
EDX:EAX	r/m32	EDX	EAX

Ejemplos de DIV

Las siguientes instrucciones realizan una división sin signo de 8 bits (83h/2), produciendo un cociente de 41h y un residuo de 1:

```
mov ax,0083h          ; dividendo
mov bl,2              ; divisor
div bl                ; AL =  41h, AH = 01h
```

Las siguientes instrucciones realizan una división sin signo de 16 bits (8003h/100h), produciendo un cociente de 80h y un residuo de 3, DX contiene la parte superior del dividendo, por lo que se debe borrar antes de ejecutar la instrucción DIV:

```
mov dx,0              ; borra dividendo, superior
mov ax,8003h          ; dividendo, inferior
mov cx,100h           ; divisor
```

```
div cx                  ; AX = 0080h, DX = 0003h
```

La siguiente instrucción realiza una división sin signo de 32 bits, usando un operando de memoria como divisor:

```
.data

dividendo QWORD 0000000800300020h

divisor DWORD 00000100h

.code

mov edx,DWORD PTR dividendo+4          ; doble palabra superior

mov eax,DWORD PTR dividendo            ; doble palabra inferior

div divisor                            ; EAX = 08003000h, EDX = 00000020h
```

División de enteros sin signo

La división de enteros con signo es casi idéntica a la división sin signo, con una importante diferencia: el dividendo implicado debe tener una extensión completa del signo antes de realizar la división. Primero veremos las instrucciones para la extensión de signo. Después las aplicaremos a la instrucción de división de enteros con signo, IDIV.

Instrucciones para la extensión del signo (CBW, CWD, CDQ)

A menudo, se debe extender el signo de los dividendos de las instrucciones de división de enteros con signo para poder realizar la división. Intel proporciona tres instrucciones útiles de extensión de signo: CBW, CWD y CDQ. La instrucción CBW (convertir byte a palabra) extiende el bit de signo de AL hacia AH, preservando el signo del número. En el siguiente ejemplo, 9Bh (en AL) y FF9Bh (en AX) son ambos iguales a -101:

```
.data

valByte SBYTE -101        ; 98h

.code

mov al,valByte            ; AL = 98h

cbw                       ; AX = FF9Bh
```

La instrucción CWD (convertir palabra a doble palabra) extiende el bit de signo de AX hacia DX:

```
.data

valWord SWORD -101        ; FF9Bh

.code
```

```
mov ax,valWord          ; AX = FF9Bh

cwd                     ; DX:AX = FFFFFF9Bh
```

La instrucción CDQ (Convertir doble palabra a palabra cuádruple) extiende el bit de signo de EAX hacia EDX:

```
.data

valDword SDWORD -101    ; FFFFFF9Bh

.code

mov eax,valDword

cdq                     ; EDX:EAX = FFFFFFFFFFFFFF9Bh
```

La instrucción IDIV

La instrucción IDIV (división con signo) realiza una división de enteros con signo, usando los mismos operandos que DIV. Antes de ejecutar la división de ocho bits, se debe extender por completo el signo del dividendo (AX). El residuo siempre tiene el mismo signo que el dividendo.

Ejemplo 1 Las siguientes instrucciones dividen -48 entre 5. Después de ejecutar IDIV, el cociente en AL es -9 y el residuo en AH es -3:

```
.data

valByte SBYTE -48

.code

mov al,valByte          ; dividendo

cbw                     ; extiende AL hacia AH

mov bl,+5               ; divisor

idiv bl                 ; AL = -9, AH = -3
```

Ejemplo 2 La división de 16 bits requiere que se extienda AX hacia DX. El siguiente ejemplo divide -5000 entre 256:

```
.data

valWord SWORD -5000

.code

mov ax,valWord
```

```
cwd                     ; se extiende AX hacia DX

mov bx,+256             ; divisor

idiv bx                 ; cociente AX = -19, res DX = -136
```

Ejemplo 3 La división de 32 bits requiere que se extienda el signo de EAX hacia EDX. El siguiente ejemplo divide -5000 entre 256:

```
.data

valDword SDWORD + 5000

.code

mov eax,valDword        ; dividendo inferior

cdq                     ; extiende EAX hacia EDX

mov ebx,-256            ; divisor

idiv ebx                ; cociente EAX = -195, res EDX = +80
```

Todos los valores de las banderas de estado aritméticas quedan indefinidos después de ejecutar DIV e IDIV.

Desbordamiento en la división

Si el operando de una división produce un cociente que no cabe en el operando de destino, se produce una condición de *desbordamiento en la división*. Eso produce una interrupción de la CPU, y el programa actual se detiene. Por ejemplo, las siguientes instrucciones generan un desbordamiento en la división debido a que el cociente (100h) no cabe en el registro AL:

```
mov ax,1000h

mov bl,10h

div bl
```

Cuando este código se ejecuta en MS-Windows, muestra el cuadro de diálogo de error resultante, producido por MS-Windows:

Al escribir instrucciones que intenten dividir entre cero, aparece una ventana de cuadro de diálogo similar:

```
mov ax,dividendo

mov bl,0

div bl
```

Es recomendable utilizar un divisor de 32 bits para reducir la probabilidad de una condición de desbordamiento en la división. Por ejemplo,

```
mov eax,1000h

cdq

mov ebx,10h

div ebx                    ; EAX = 00000100h
```

Para evitar la división entre cero, se evalúa el divisor antes de la división:

```
mov ax,dividendo

mov bl,divisor

cmp bl,0                   ; verifica el divisor

je NoDivisionCero          ; ¿Cero? muestra error

div bl                     ; no es cero: continúa
```

```
.

.

NoDivisionCero            ; (muestra “Intento de dividir entre cero”)
```

Implementación de expresiones aritméticas

En el capítulo 4 se demostró cómo implementar expresiones aritméticas mediante la suma y la resta. Ahora podemos incluir la multiplicación y la división. Al principio, la implementación de esas expresiones parece ser una actividad más adecuada para los escritores de compiladores, pero hay mucho que obtener a través del estudio práctico. Podemos aprender de qué manera los compiladores optimizan el código. Además, se puede implementar una mejor comprobación de error que un compilador ordinario, comprobando el tamaño del producto resultante de las operaciones de multiplicación. La mayoría de los compiladores de lenguajes de alto nivel ignoran los 32 bits superiores de producto, cuando multiplican dos operandos de 32 bits. Sin embargo, en lenguaje ensamblador, podemos usar las banderas Acarreo y Desbordamiento para saber cuando el producto no cabe en 32 bits. En el capítulo 7 se explicó el uso de estas banderas.

Hay dos formas sencillas de ver el código ensamblador generado por un compilador de C++: abrir una ventana de desensamblado mientras se depura un programa en C++, o generar un archivo de listado en lenguaje ensamblador. Por ejemplo, en Microsoft Visual C++ el interruptor de línea de comandos /FA genera un archivo de listado en lenguaje ensamblador.

Ejemplo 1 Implementemos la siguiente instrucción de C++ en el lenguaje ensamblador, usando enteros de 32 bits sin signo:

```
var4 = (var1 + var2) * var3;
```

Éste es un problema simple, ya que podemos trabajar de izquierda a derecha (primero la suma, después la multiplicación). Después de la segunda instrucción, EAX contiene la suma de **var1** y **var2**. En la tercera instrucción, EAX se multiplica por **var3** y el producto se almacena en EAX:

```
mov eax,var1

add eax,var2

mul var3            ; EAX = EAX * var3

jc muyGrande        ; ¿desbordamiento con signo?

mov var4,eax

jmp siguiente

muyGrande:          ; muestra un mensaje de error
```

Si la instrucción MUL genera un producto mayor que 32 bits, la instrucción JC, nos envía a una etiqueta que maneja el error.

Ejemplo 2 implementemos la siguiente instrucción en C++, usando enteros de 32 bits sin signo:

```
var4 = (var1 * 5) / (var2 – 3);
```

En este ejemplo, hay dos subexpresiones dentro de los paréntesis. El lado izquierdo puede asignarse a EDX:EAX, por lo que no es necesario comprobar si hay desbordamiento. El lado derecho se asigna EBX, y la división final completa la expresión:

```
mov eax,var1          ; lado izquierdo

mov ebx,5

mul ebx               ; EDX:EAX = producto

mov ebx,var2          ; lado derecho

sub ebx,3

div ebx               ; división final

mov var4,eax
```

Ejemplo 3 Implementemos la siguiente instrucción en C++, usando enteros de 32 bits con signo:

```
var4 = (var1 * -5) / (-var2 % var3);
```

Este ejemplo es un poco más engañoso que los anteriores. Podemos empezar con la expresión del lado derecho y almacenar su valor en EBX. Como los operandos tienen signo, es importante extender el signo del dividendo hacia EDX y utilizar la instrucción IDIV:

```
mov eax,var2          ; empieza lado derecho

neg eax

cdq                   ; dividendo con signo extendido

idiv var3             ; EDX = residuo

mov ebx,edx           ; EBX = lado derecho
```

A continuación, calculamos la expresión del lado izquierdo, almacenando el producto en EDX:EAX:

```
mov eax,-5            ; empieza lado izquierdo

imul var1             ; EDX:EAX = lado izquierdo
```

Por último, el lado izquierdo (EDX:EAX) se divide entre el lado derecho (EBX):

```
idiv ebx              ; división final

mov var4,eax          ; cociente
```

Suma y resta extendidas

El proceso de *Suma y resta con precisión extendida* se refiere a la suma y resta de números que tengan un tamaño casi ilimitado. Supongamos que nos piden escribir un programa en C++ para sumar dos enteros de 1024 bits ¡La solución no será fácil! Pero en el lenguaje ensamblador, las instrucciones ADC (suma con acarreo) y SBB (resta con préstamo) se adaptan muy bien a este tipo de problemas.

Instrucción ADC

La instrucción ADC (suma con acarreo) suma un operando de origen y el contenido de la bandera Acarreo a un operando de destino. Los formatos de las instrucciones son iguales que para la instrucción ADD:

ADC reg,reg

ADC mem,reg

ADX reg,mem

ADC mem,imm

ADC reg,imm

Por ejemplo, las siguientes instrucciones suman dos enteros de 8 bits (FFh + FFh), produciendo una suma en DL:AL, que es 01FEh:

```
mov dl,0

mov al,0FFh

add al,0FFh                    ; AL = FE

adc dl,0                       ; DL = 01
```

De manera similar, las siguientes instrucciones suman dos enteros de 32 bits (FFFFFFFFh + FFFFFFFh), produciendo una suma de 64 bits en EDX:EAX: 00000001FFFFFFFEh

```
mov edx,0

mov eax,0FFFFFFFFh

add eax,0FFFFFFFFh

adc edx,0
```

Ejemplo de suma extendida

El siguiente procedimiento **Suma_Extendida** suma dos enteros extendidos del mismo tamaño. Utiliza un ciclo para sumar cada par de dobles palabras, guarda la bandera Acarreo e incluye el acarreo con cada par subsiguiente de dobles palabras:

```
pushad
clc                             ; borra la bandera Acarreo
L1: mov eax,[esi]               ; obtiene el primer entero
adc eax,[EDI]                   ; suma el segundo entero
pushfd                          ; guarda la bandera Acarreo
mov [ebx],eax                   ; almacena la suma parcial
add esi,4                       ; avanza los 3 apuntadores
add edi,4
add ebx,4
popfd                           ; restaura la bandera Acarreo
loop L1                         ; repite el ciclo
mov dword ptr [ebx],0           ; borra doble palabra superior de suma
adc dword ptr [ebx],0           ; suma cualquier acarreo restante
popad
ret
Suma_Extendida ENDP
```

El siguiente extracto de SumaEx.asm llama a **Suma_Extendida** y le pasa dos enteros de 64 bits. Debemos tener cuidado de asignar una doble palabra extra para la suma:

```
.data
op1 QWORD 0A2B2A40674981234h
op2 QWORD 08010870000234502h
suma DWORD 3 dup(0FFFFFFFFh)   ; = 0000000122C32B0674BB5736
.code
main PROC
```

```
mov esi,OFFSET op1              ; primer operando
mov edi,OFFSET op2              ; segundo operando
mov ebx,OFFSET suma             ; operando de suma
mov ecx,2                       ; número de dobles palabras
call Suma_Extendida
; Muestra la suma
mov eax,suma + 8                ; muestra doble palabra de orden superior
call WriteHex
mov eax,suma + 4                ; muestra doble palabra intermedia
call WriteHex
mov eax,suma                    ; muestra doble palabra de orden inferior
call Crlf
exit
main ENDP
```

El programa produce el siguiente resultado. La suma produce un acarreo:

0000000122C32B0674BB5736

Instrucción SBB

La instrucción SBB (resta con préstamo) resta un operando de origen y el valor de la bandera Acarreo a un operando de destino. Los posibles operandos son los mismos que para la instrucción ADC.

El siguiente código de ejemplo realiza una resta de 64 bits. Establece EDX:EAX a 0000000100000000h

Y resta uno a este valor. Los 32 bits inferiores se restan primero, con lo que se activa la bandera Acarreo. Después se restan los 32 bits superiores, incluyendo la bandera Acarreo:

```
mov edx,1                       ; mitad superior
mov eax,0                       ; mitad inferior
sub eax,1                       ; resta 1
sbb edx,0                       ; resta la mitad superior
```

La diferencia de 64 bits en EDX:EAX es 00000000FFFFFFFFh.

Aritmética ASCII y con decimales desempaquetados

La aritmética de enteros que hemos mostrado hasta ahora, sólo ha tratado con valores binarios. La CPU calcula en binario, pero también puede realizar aritmética con cadenas ASCII de decimales. El usuario puede introducir en forma conveniente estas cadenas y se pueden mostrar en la ventana de la consola, sin necesidad de que se conviertan a binario. Supongamos que un programa debe recibir como entrada dos números del usuario, para sumarlos. A continuación se muestra un ejemplo del resultado, en donde el usuario introdujo los números 3402 y 1256:

```
Escriba el primer número: 3402

Escriba el segundo número: 1256

La suma es: 4658
```

Tenemos dos opciones al calcular y mostrar la suma:

1. Convertir ambos operandos a binario, sumar los valores binarios y convertir la suma de binario a cadenas de dígitos ASCII.
2. Sumar las cadenas de dígitos directamente, mediante una suma sucesiva de cada par de dígitos ASCII (2 + 6, 0 + 5, 4 + 2. 3 + 1). La suma es una cadena de dígitos ASCII, por lo que puede mostrarse directamente en pantalla.

La segunda opción requiere del uso de instrucciones especializadas para ajustar la suma después de sumar cada par de dígitos ASCII. A continuación se muestran las cuatro instrucciones que tatan con la suma, resta, multiplicación y división ASCII:

AAA	(Ajuste ASCII después de la suma)
AAS	(Ajuste ASCII después de la resta)
AAM	(Ajuste ASCII después de la multiplicación)
AAD	(Ajuste ASCII antes de la división)

ASCII decimal y decimal desempaquetado Los 4 bits superiores de un entero decimal desempaquetado siempre son ceros, mientras que los mismos bits en un número decimal ASCII son iguales a 0011b. En cualquier caso, ambos tipos de enteros almacenan un dígito por byte. El siguiente ejemplo muestra cómo se almacenaría 3402 en ambos formatos:

Formato ASCII:

33	34	30	32

Desempaquetado:

03	04	00	02

(Todos los valores son en hexadecimal)

Aunque la aritmética ASCII se ejecuta con más lentitud que la aritmética binaria, tiene dos ventajas:

- No es necesaria la conversión desde el formato de cadena antes de realizar las operaciones aritméticas.
- El uso de un punto decimal supuesto permite las operaciones sobre números reales, sin peligro de errores de redondeo, que se producen con los números de punto flotante.

La suma y resta ASCII permiten que los operandos estén en formato ASCII o en formato decimal desempaquetado. Sólo pueden usarse números decimales desempaquetados para la multiplicación y división.

Instrucción AAA

La instrucción AAA (ajuste ASCII después de la suma) ajusta el resultado binario de una instrucción ADD o ADC. Suponiendo que AX contiene un valor binario que se produce al sumar dos dígitos ASCII, AAA convierte AX en dos dígitos decimales, desempaquetados. Una vez en formato desempaquetado, AH y AL pueden convertirse fácilmente en ASCII, si se les aplica un OR con 30h.

El siguiente ejemplo muestra cómo sumar los dígitos ASCII 8 y 2 de manera correcta, usando la instrucción AAA. Hay que dejar AH en cero antes de realizar la suma, ya que de lo contrario el resultado devuelto por AAA se verá influenciado. La última instrucción convierte a AH y AL en dígitos ASCII:

```
mov ah,0

mov al,'8'                    ; AX = 0038h

add al,2                      ; AX = 006A

aaa                           ; AX = 100 (resultado del ajuste ASCII)

or ax,3030h                   ; AX = 3130h = '10' (se convierte a ASCII)
```

Suma de varios bytes mediante el uso de AAA

Vamos a ver un procedimiento que suma valores decimales ASCII con puntos decimales implícitos. La implementación es un poco más compleja de lo que uno podría imaginar, ya que el acarreo de la suma de cada dígito debe propagarse a la siguiente posición más alta. En el siguiente seudocódigo, el nombre *acc* se refiere a un registro acumulador de 8 bits:

```
esi (índice) = longitud de primer_numero – 1
```

```
edi (índice) = longitud de primer_numero

ecx = longitud de primer_numero

establece valor de acarreo en 0

Itera

acc = primer_numero[esi]

suma acarreo anterior a acc

guarda acarreo en acarreo1

acc += segundo_numero[esi]

OR entre acarreo y acarreo1

suma[edi] acc

dec edi

hasta que ecx == 0

Almacena el último dígito de acarreo en la suma
```

El dígito de acarreo siempre debe convertirse en ASCII. Al sumar el dígito de acarreo con el primer operando, hay que ajustar el resultado con AAA. He aquí el listado:

```
.data

DECIMAL_OFFSET = 5

.data

decimal_one BYTE "100123456789765"       ; 1001234567.89765

decimal_two BYTE "900402076502015"       ; 9004020765.02015

sum BYTE (SIZEOF decimal_one + 1) DUP(0),0


.code

main PROC

; Empieza en la posición del último dígito.

mov esi,SIZEOF decimal_one - 1

mov edi,SIZEOF decimal_one
```

```
mov ecx,SIZEOF decimal_one
mov bh,0                        ; establece el valor de acarreo a cero
L1:mov ah,0                     ; borra AH antes de la suma
mov al,decimal_one[esi]         ; obtiene el primer dígito
add al,bh                       ; suma el acarreo anterior
aaa                             ; ajusta la suma (AH = acarreo)
mov bh,ah                       ; guarda el acarreo en acarreo1
or bh,30h                       ; lo convierte en ASCII
add al,decimal_two[esi]         ; suma el Segundo dígito
aaa                             ; ajusta la suma (AH = acarreo)
or bh,ah                        ; OR al acarreo con acarreo1
or bh,30h                       ; lo convierte en ASCII
or al,30h                       ; convierte a AL de vuelta en ASCII
mov sum[edi],al                 ; lo guarda en suma
dec esi                         ; retrocede un dígito
dec edi
loop L1
mov sum[edi],bh                 ; guarda el último dígito
; Muestra la suma como una cadena.
mov edx,OFFSET sum
call WriteString
call Crlf
exit
main ENDP
END main
```

Instrucción AAS

La instrucción AAS (ajuste ASCII después de la resta) va después de una instrucción SUB o SBB que resta un valor decimal desempaquetado de otro, y almacena el resultado en AL. Hace que el resultado en AL sea consistente con la representación de dígitos ASCII. El ajuste es necesario sólo cuando la resta genera un resultado negativo. Por ejemplo, las siguientes instrucciones restan al 9 del 8 ASCII:

```
.data

val1 BYTE ‘8’

val2 BYTE ‘9’

.code

mov ah,0

mov ah,val1                    ; AX = 0038h

sub al,val2                    ; AX = 00FFh

aas                            ; AX = FF09h

pushf                          ; guarda la bandera Acarreo

or al,30h                      ; AX = FF39h

popf                           ; restaura la bandera Acarreo
```

Después de la instrucción SUB, AX es igual a 0FFh. La instrucción ASS convierte a AL en 09h y resta 1 de AH, con lo que lo establece en FFh y activa la bandera Acarreo.

Instrucción AAM

La instrucción AAM (ajuste ASCII después de la multiplicación) convierte el producto binario producido por MUL a decimal empaquetado. La multiplicación sólo puede usar decimales desempaquetados. En el siguiente ejemplo, multiplicamos 5 por 6 y ajustamos el resultado en AX. Después del ajuste, AX = 300h, la representación de 30 en decimal desempaquetado:

```
.data

valAsc BYTE 05h,06h

.code

mov bl,ValAsc                  ; primer operando

mov al,[ValASC+1]              ; segundo operando

mul bl                         ; AX = 001Eh

mul bl                         ; AX = 0300h
```

```
aam
```

Instrucción AAD

La instrucción ADD (ajuste ASCII antes de la división) convierte un dividendo decimal desempaquetado en AX a binario, en preparación para ejecutar la instrucción DIV. El siguiente ejemplo convierte el número 0307h desempaquetado a binario, y después lo divide entre 5. DIV produce un cociente de 07h en AL y un residuo de 02h en AH:

```
.data

cociente BYTE ?

residuo BYTE ?

.code

mov ax,0307h                    ; dividendo

aad

mov bl,5                        ; divisor

div bl                          ; AX = 0207

mov cociente,al

mov residuo,ah
```

Aritmética con decimales empaquetados

Los enteros decimales empaquetados almacenan dos dígitos decimales por byte. Cada dígito se representa mediante cuatro bits. Si hay un número impar de dígitos, el nibble más alto se llena con un cero. Los tamaños de almacenamiento pueden variar:

```
bcd1 QWORD 2345673928737285h    ; 2,345,673,928,737,285 decimal

bcd2 DWORD 12345678h            ; 12,345,678 decimal

bcd3 DWORD 08723654h            ; 8,723,654 decimal

bcd4 WORD 9345h                 ; 9,345 decimal

bcd5 BYTE 34h                   ; 34 decimal
```

El almacenamiento de decimales empaquetados tiene por lo menos dos puntos fuertes:

- Los números pueden tener casi cualquier número de dígitos significativos. Esto hace posible realizar cálculos con mucha precisión.
- La conversión de números decimales empaquetados a ASCII (y viceversa) res relativamente simple.

Dos instrucciones, DAA (ajuste decimal después de suma) y DAS (ajuste decimal después de resta), ajustan el resultado de una operación de suma o resta con decimales empaquetados. Por desgracia, no existen instrucciones así para la multiplicación y la división. En esos casos, el número debe desempaquetarse, multiplicarse o dividirse, y luego volver a empaquetarse.

Instrucción DAA

La instrucción DAA (ajuste decimal después de la suma) convierte una suma binaria producida por ADD o ADC en AL, a formato decimal empaquetado. Por ejemplo, las siguientes instrucciones suman los decimales empaquetados 35 y 48. La suma binaria (7Dh) se ajusta a 83h, la suma decimal empaquetada de 35 y 48.

```
mov al,35h

add al,48h                    ; AL = 7Dh

daa                           ; AL = 83 (resultado ajustado)
```

La lógica interna de DAA se documenta en el Manual de referencia del conjunto de instrucciones IA-32.

Ejemplo El siguiente programa suma dos enteros decimales empaquetados de 16 bits y almacena la suma en una doble palabra empaquetada. La suma requiere que la variable suma contenga espacio para un dígito más que los operandos.

```
TITLE Ejemplos con decimales empaquetados (SumaEmpaquetado.asm)

; Demuestra la suma de decimales  empaquetados.

; Última actualización: 06/01/2006

INCLUDE Irvine32.inc

.data

empaquetado_1 WORD 4536h

empaquetado_2 WORD 7207h

suma DWORD ?

.code

main PROC

; Inicializa suma e índice.

  mov suma,0

  mov esi,0
```

```
; Suma los bytes inferiores.
  mov al,BYTE PTR empaquetado_1[esi]
  add al,BYTE PTR empaquetado_2[esi]
  daa
  mov BYTE PTR suma[esi],al
; Suma los bytes superiores, incluye el acarreo.
  inc esi
  mov al,BYTE PTR empaquetado_1[esi]
  adc al,BYTE PTR empaquetado_2[esi]
  daa
  mov BYTE PTR suma[esi],al
; Suma el acarreo final, si hay.
  inc esi
  mov al,0
  adc al,0
  mov BYTE PTR suma[esi],al
; Muestra la suma en hexadecimal.
  mov eax,suma
  call WriteHex
  call Crlf
  exit
main ENDP
END main
```

Sin necesidad de decirlo, el programa contiene código repetitivo que sugiere el uso de un ciclo.

Instrucción DAS

La instrucción DAS (ajuste decimal después de la resta) convierte el resultado binario de una instrucción SUB o SBB en AL, a formato decimal empaquetado. Por ejemplo, las siguientes instrucciones restan los decimales desempaquetados 48 y 85, y ajustan el resultado:

```
mov bl,48h

mov al,85h

sub al,bl                       ; AL = 3Dh

das                             ; AL = 37h (resultado ajustado)
```

La lógica interna de DAS se documenta en el manual de referencia del conjunto de instrucciones IA-32

8 Procedimientos avanzados

Marcos de pila

Un marco de pila (o registro de activación) es el área de la pila que se aparta para los argumentos que se pasan, la dirección de retorno de las subrutinas, las variables locales y los registros almacenados. El marco de pila se crea mediante los siguientes pasos secuenciales:

- Los argumentos que se pasan, en caso de haber, se meten en la pila.
- Se hace la llamada a la subrutina, lo cual provoca que la dirección de retorno de la misma se meta en la pila.
- En cuanto la subrutina se empieza a ejecutar, EBP se coloca en la pila.
- EBP se hace igual a ESP. De este punto en adelante, EBP actúa como una referencia base para todos los parámetros de la subrutina.
- Si hay variables locales, ESP se decrementa para reservar espacio para las variables en la pila.
- Si hay que guardar algún registro, se mete en la pila.

Parámetros de pila

Existen dos tipos básicos de parámetros de subrutinas: *los parámetros de registro y los parámetros de pila*. En esta sección veremos cómo declarar y utilizar parámetros de pila.

> Los valores que un programa pasa a una subrutina que llama se conocen como *argumentos*. Cuando la subrutina a la que se llamó recibe los valores, se les llama *parámetros*.

La subrutina que se llamó accede a los argumentos que se meten en la pila al momento de su llamada. Los parámetros de registro están optimizados para la velocidad de ejecución de un programa. Por desgracia, tienden a crear amontonamiento de código en los programas que hacen llamadas. A menudo hay que guardar el contenido existente de los resultados antes de

que se puedan cargar con los valores de los argumentos. Tal el caso de cuando se hace una llamada a **DumpMem**, por ejemplo:

```
pushad

mov esi, OFFSET arreglo        ; desplazamiento (OFFSET) inicial

mov ecx,LENGTHOF               ; tamaño, en unidades

mov ebx,TYPE arreglo           ; formato de doble palabra

call DumpMem

popad
```

Los parámetros de pila ofrecen un método más flexible. Justo antes de la llamada a la subrutina, los argumentos se meten en la pila. Por ejemplo, si **DumpMem** utilizara parámetros de pila, lo llamaríamos usando el siguiente código:

```
push TYPE arreglo

push LENGTHOF arreglo

push OFFSET arreglo

call DumpMEM
```

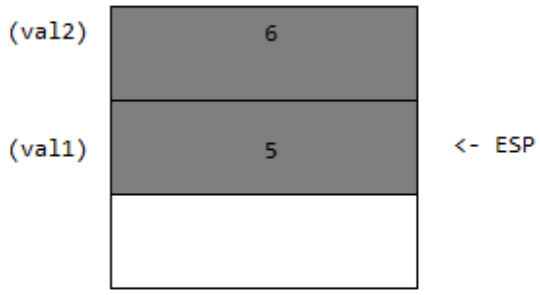
Durante las llamadas a subrutinas, se meten en la pila dos tipos generales de argumentos:

- Argumentos de valor (los valores de las variables y constantes).
- Argumentos de referencia (las direcciones de las variables).

Paso por valor Cuando un argumento se pasa por *valor*, se mete una copia del valor en la pila, supongamos que llamamos a una subrutina de nombre **SumarDos** y le pasamos dos enteros de 32 bits:

```
.data

val1 DWORD 5

val2 DWORD 6

.code

push val2

push val1

call SumarDos
```

A continuación se muestra una imagen de la pila justo antes de la instrucción CALL:



Una llamada a una función equivalente en C++ sería:

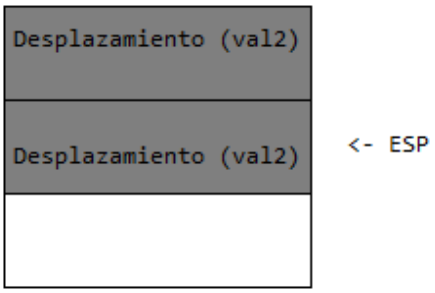
```
int suma = SumarDos (val1, val2);
```

Observamos que los argumentos se meten en la pila en orden inverso, lo cual es la norma para los lenguajes C y C++.

Paso por referencia Un argumento que se pasa por referencia consiste en la dirección (desplazamiento) de un objeto. Las siguientes instrucciones llaman a **Intercambiar** y le pasan los dos argumentos por referencia:

```
push OFFSET val2

push OFFSET val1

call Intercambiar
```

A continuación se muestra una imagen de la pila, justo antes de la llamada Intercambiar:



La llamada a función equivalente en C/C++ pasaría las direcciones de los argumentos val1 y val2:

Intercambiar(&val1, &val2);

Paso de arreglos Hay una importante excepción a la regla que acabamos de presentar, relacionada con el paso por valor. Al pasar un arreglo, los programas de lenguaje de alto nivel siempre lo pasan por referencia. No es práctico pasar una extensa cantidad de datos por valor, ya que habría que meter los datos directamente en la pila. Hacer esto reduciría la velocidad de ejecución del programa y ocuparía el valioso espacio de la pila. Por ejemplo, las siguientes instrucciones pasan el desplazamiento de **arreglo** a una subrutina llama **LlenarArreglo**:

```
.data

arreglo DWORD 50 DUP(?)

.code

push OFFSET arreglo

call LlenarArreglo
```

Acceso a los parámetros de pila (C/C++)

Los programas de C y C++ tienen forma estándar de inicializar y acceder a los parámetros durante las llamadas a funciones. Empiezan con un *prólogo* que consiste en instrucciones que guardan el registro EBP, y lo establecen en la parte superior de la pila. De manera opcional, pueden meter ciertos registros en la pila, cuyos valores se restauren cuando la función regrese. El final de la función consiste en un *epílogo*, en el cual se restaura un registro EBP y la función RET regresa de la función, y borra los parámetros de la pila.

Ejemplo: *SumarDos* La siguiente función **SumarDos**, escrita en C, recibe dos teneros que se pasan por valor y devuelve su suma:

```
int SumaDos (int x, int y)

{

   return x + y;

}
```

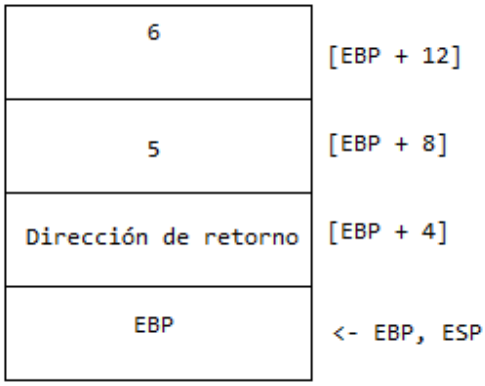
Vamos a crear una implementación equivalente en lenguaje ensamblador. En su prólogo, **SumarDos** mete a EBP en la pila para preservar su valor existente:

```
SumarDos PROC

         Push ebp
```

A continuación, EBP se establece con el mismo valor que ESP, de manera que EBP pueda ser el apuntador base para el marco de la pila de SumarDos:

```
SumarDos PROC

         Push ebp

         mov ebp,esp
```

La siguiente figura muestra el contenido del marco de pila después de ejecutar las dos instrucciones anteriores. Cada entrada en la pila es una doble palabra:



SumarDos podría meter registros adicionales en la pila, sin alterar los desplazamientos de los parámetros de pila de EBP. ESP cambiará su valor, pero EBP no.

Acceso a los parámetros de pila Las funciones en C y C++ utilizan el direccionamiento base-desplazamiento para acceder a los parámetros de pila. EBP es el registro base y el desplazamiento es una constante. Por lo general, los valores de 32 bits se devuelven en EAX. La siguiente implementación de SumarDos suma los parámetros y devuelve su suma en EAX:

```
SumarDos PROC

         push ebp

         mov ebp, esp          ; base del marco de pila

         mov eax,[ebp + 12]    ; segundo parámetro

         add eax,[ebp + 8]     ; primer parámetro

         pop ebp

         ret

SumarDos ENDP
```

Limpieza de la pila

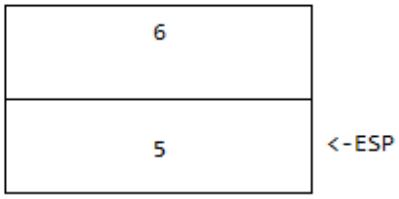
Debe haber una manera para que los parámetros se eliminen de la pila, al momento en que se regrese una subrutina. De no ser así se produce una fuga de memoria, y la pila se vuelve corrupta. Por ejemplo, supongamos que las siguientes instrucciones en **main** llaman a **SumarDos**:

push 5

push 6

call SumarDos

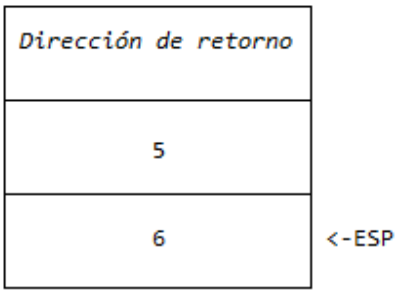
He aquí una imagen de la pila, después de regresar de la llamada:



Dentro de main, podemos ignorar el problema y esperar que el programa termine en forma normal. Si llamamos a SumarDos dentro de un ciclo, la pila podría desbordarse debido a que cada llamada consume 8 bytes de memoria. Se produce un problema más serio si llamamos a **Ejemplo1** desde **main**, que a su vez llama a **SumarDos**:

```
main PROC
     call Ejemplo1
     exit
main ENDP
Ejemplo1 PROC
     push 5
     push 6
     call SumarDos
     ret                        ; la pila está corrupta!
Ejemplo1 ENDP
```

Cuando la instrucción RET en el ejemplo1 está a punto de ejecutarse, ESP apunta al entero 5, en vez de apuntar a la dirección de retorno que nos regresa a **main**. Evidentemente el programa se bifurca a la ubicación 5 y falla:



Una solución simple para este problema es sumar un valor a ESP que haga que apunte a la dirección de retorno. En el ejemplo actual, podemos colocar una instrucción ADD después de CALL:

```
Ejemplo1 PROC
   push 5
   push 6
   call SumarDos
   add esp,8                    ; elimina argumentos de la pila
   ret
Ejemplo1 ENDP
```

Ésta es la acción que toman los programas en C y C++.

Convención de llamadas de STDCALL Otra forma común de manejar el problema de la limpieza de la pila es utilizar una conversión llamada STDCALL. Podemos suministrar un parámetro entero a la instrucción RET dentro de SumarSub para corregir a ESP. El entero debe ser igual al número de bytes del espacio que consumen en la pila los parámetros de la subrutina:

```
SumarDos PROC
    push ebp
    mov ebp,esp                 ; base del marco de pila
    mov eax,{ebp+12}            ; segundo parámetro
    add eax,[ebp+8]             ; primer parámetro
    pop ebp
```

```
    ret 8                       ; limpia la pila
SumarDos ENDP
```

Entonces la pregunta simplemente sería, ¿quién será el responsable de limpiar la pila? ¿El código que llama a una subrutina, o la misma subrutina? Existen concesiones. Por una parte, STDCALL reduce la cantidad de código que se genera para las llamadas a las subrutinas (por una instrucción) y asegura que los procedimientos que hacen las llamadas nunca olviden limpiar la pila. Por otra parte, la convención de llamadas de C permite que las subrutinas declaren un número variable de parámetros. El procedimiento que hace la llamada decide cuántos argumentos va a pasar. Un ejemplo es la función **printf**, cuyo número de argumentos depende del número de especificadores de formato en el argumento de cadena inicial:

```
int x = 5;

float y = 3.2;

char z =’Z’;

printf(“Imprimiendo valores: %d, %f, %c, x, y, z);
```

Un compilador de C mete los argumentos en la pila en orden inverso, seguidos de un argumento de cuenta, que indíca el número de argumentos actuales. La función obtiene la cuenta de argumentos y accede a éstos, uno por uno. La implementación de la función no tiene una manera conveniente de codificar una constante en la instrucción RET para limpia la pila, por lo que la responsabilidad se deja al proceso que hace la llamada.

De aquí en adelante, asumiremos que se utiliza STDCALL en todos los ejemplos de procedimientos, a menos que se indique explícitamente lo contrario. También nos referimos a las subrutinas como procedimientos, ya que nuestros ejemplos están escritos en lenguaje ensamblador.

Paso de argumentos de 8 y 16 bits a la pila

Al pasar a la pila argumentos de los procedimientos en modo protegido, es mejor meter operandos de 32 bits. Aunque se pueden meter operandos de 16 bits, esto impide que ESP se alinee en un límite de doble palabra, Puede ocurrir un fallo de página y se puede degradar el rendimiento en tiempo de ejecución. Por eso debemos expandirlos a 32 bits antes de meterlos en la pila.

El siguiente procedimiento Mayúscula recibe un argumento tipo carácter y devuelve su equivalente en mayúscula en AL:

```
Mayuscula PROC
    push ebp
    mov ebp,esp
```

```
    mov al,[ebp+8]              ; AL = carácter
    cmp al,'a'                  ; ¿es menor que 'a'?
    jb L1                       ; si: no hace nada
    cmp al,'z'                  ; ¿es mayor que 'z'?
    ja L1                       ; si: no hace nada
    sub al,32                   ; no: lo convierte
L1: pop ebp
    ret 4                       ; limpia la pila
Mayuscula ENDP
```

Si pasamos una literal tipo carácter a Mayúsculas, la instrucción PUSH expande en forma automática el carácter a 32 bits:

```
push 'x'
call Mayuscula
```

Para pasar una variable tipo carácter se requiere más cuidado, ya que la instrucción PUSH no permite operandos de 8 bits:

```
.data
valCar BYTE 'x'
.code
    push ValCar                 ; error de sintaxis!
    call Mayuscula
```

En su lugar, utilizamos MOVZX para expandir hacia EAX:

```
   movzx eax,valCar             ; mueve con extensión
    push eax
    call Mayuscula
```

Ejemplo de argumento de 16 bits Supongamos que queremos pasar dos enteros de 16 bits al procedimiento SumarDos que vimos antes. El procedimiento espera valores de 32 bits, por lo que la siguiente llamada producirá un error:

```
.data
```

```
palabra1 WORD 1234h
palabra2 WORD 4111h
.code
    push palabra1
    push palabra2
    call SumarDos                ; ¡Error!
```

En su lugar, podemos extender con ceros cada argumento, antes de meterlo en la pila. El siguiente código llama a SumarDos en forma correcta:

```
    movzx eax,palabra1
    push eax
    movzx eax,palabra2
    push eax
    call SumarDos                ; la suma está en EAX
```

> El procedimiento que llama a otro debe asegurarse de que los argumentos que pase sean consistentes con los parámetros que espera el procedimiento al que llamó. En el caso de los parámetros de pila, el orden y el tamaño de los parámetros son importantes.

Paso de argumentos multipalabras

Al pasar enteros multipalabras a los procedimientos mediante el uso de la pila, tal vez sea conveniente meter la parte de mayor orden primero, para pasar después a la parte de menor orden. Al hacer esto, el entero se coloca en la pila en orden *Little endian* (el byte de menor orden en la dirección más baja). El siguiente procedimiento **EscribirHex64** recibe un entero de 64 bits en la pila y lo muestra en hexadecimal:

```
EscribirHex64 PROC
push ebp
mov ebp,esp
mov eax,[ebp+12]             ; doble palabra superior
call WriteHex
mov eax,[ebp+8]              ; doble palabra inferior
call WriteHex
```

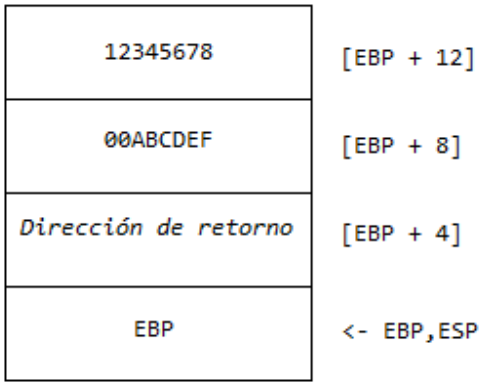
```
pop ebp

ret 8

EscribirHex64 ENDP
```

La llamada a EscribirHex64 mete la mitad superior de **valLong**, seguida de la mitad inferior:

```
.data

valLong DQ 1234567800ABCDEFh

.code

      push DWORD PTR valLong + 4    ; doble palabra superior

      push DWORD PTR valLong        ; doble palabra inferior

      call EscribirHex64
```

La siguiente figura muestra una imagen del marco de la pila, después de meter EBP dentro de EscribirHex64.

Marco de la pila después de meter EBP.



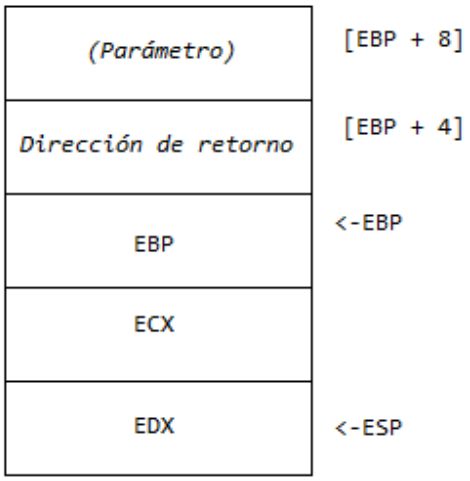
Como guardar y restaurar registros

A menudo, las subrutinas guardan el contenido actual de los registros en la pila antes de modificarlos, para poder restaurar los valores originales justo antes de regresar. Lo ideal es que los registros en cuestión se metan a la pila justo antes de establecer EBP a ESP, y justo después de reservar espacio para las variables locales. Esto nos ayuda a evitar cambiar los desplazamientos de los parámetros existentes en la pila. Por ejemplo, supongamos que el siguiente procedimiento **MiSub** tiene un parámetro en la pila. Mete a ECX y a EDX después de asignar a EBP la base del marco de pila, y carga el parámetro de la pila en EAX:

```
MiSub PROC
        push ebp                ; guarda apuntador base
        mov ebp,esp             ; base del marco de pila
        push ecx
        push edx                ; guarda EDX
        push eax,[ebp+8]        ; obtiene el parámetro de la pila
        .
        .
        pop edx                 ; restaura los registros guardados
        pop ecx
        pop ebp                 ; restaura apuntador base
        ret                     ; limpia la pila
MiSub ENDP
```

Después de inicializarse, el contenido de EBP permanece fijo a lo largo de la subrutina. Si se meten ECX y EDX, no se ve afectado el desplazamiento desde EBP de los parámetros que ya se encuentran en la pila, ya que ésta crece debajo de EBP.

Marco de pila para el procedimiento MiSub.



Como el operador USES afecta la pila

El operador USES (capítulo 5) lista los nombres de los registros que se van a guardar al principio de un procedimiento, y que se van a restaurar cuando el procedimiento termine. MASM genera en forma automática las instrucciones PUSH y POP apropiadas para cada registro con nombre. **Precaución: los procedimientos que utilizan parámetros de pila explícitos deben evitar el operando USES**. Veamos un ejemplo que muestra por qué. El siguiente procedimiento MiSub1 emplea el operador USES para guardar y restaurar a ECX y EDX:

```
Misub1 PROC USES ecx edx

       ret

MiSub1 ENDP
```

El siguiente código lo genera MASM cuando ensambla a **MiSub1**:

```
       push ecx

       push edx

       pop edx

       pop ecx

       ret
```

Supongamos que combinamos USES con un parámetro de pila, como en el siguiente procedimiento MiSub2.

Se espera que su parámetro se encuentre en la pila, EBP+8:

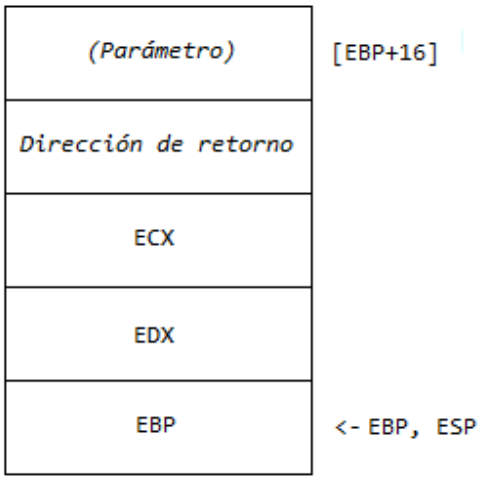
```
MiSub2 PROC USES ecx edx

       push ebp                ; guarda apuntador base

       push ebp,esp            ; base del marco de pila

       mov eax,[ebp+8]         ; obtiene el parámetro de pila

       pop ebp                 ; restaura apuntador base

       ret 4                   ; limpia la pila

MiSub2 ENDP
```

He aquí el código correspondiente, generado por MASM para **MiSub2**:

```
       push ecx

       push edx

       push ebp
```

```
mov ebp,esp
mov eax,dword ptr [ebp+8]   ; ¡ubicación incorrecta!
pop ebp
pop edx
pop ecx
ret 4
```

Se produce un error, ya que MASM inserto las instrucciones PUSH para ECX y EDX al principio del procedimiento, alterando el desplazamiento del parámetro de pila. la siguiente figura muestra cómo debe hacerse referencia ahora al parámetro de la pila como [EBP + 16]. USES modifica la pila antes de guardar EBP, lo cual va contra del código prólogo estándar para las subrutinas. más adelante veremos la directiva PROC que tiene una sintaxis de alto nivel para declarar parámetros de pila. En ese contexto, el operador USES no causa problemas.



Variables locales

En los programas de lenguaje de alto nivel, las variables que se crean, usan y destruyen dentro de una sola subrutina se conocen como *variables locales*. Una variable local tiene distintas ventajas, en comparación las variables que se declaran fuera de las subrutinas:

- Sólo las instrucciones dentro de la subrutina que encierra a una variable local pueden ver o modificar esa variable. Esta característica evita los errores en los programas ocasionados por modificar variables desde muchas ubicaciones distintas en el código fuente de un programa.
- El espacio de almacenamiento que utiliza las variables se libera cuando termina la subrutina.

- Una variable local puede tener un mismo nombre que una variable local en otra subrutina, sin crear un conflicto de nombres, Esta característica es útil en los programas extensos, cuando es probable que dos variables tengan el mismo nombre.
- Las variables locales son esenciales al escribir subrutinas recursivas, así como subrutinas ejecutadas por varios subprocesos de ejecución.

Las variables locales se crean en la pila en tiempo de ejecución, por lo general, debajo del apuntador base (EBP). Aunque no pueden recibir valores predeterminado en tiempo de ensamblado, pueden inicializarse en tiempo de ejecución. Podemos crear variables locales en lenguaje ensamblador mediante el uso de las mismas técnicas que en C y C++.

Ejemplo La siguiente función en C++ declara las variables locales X y Y:

```
void MiSub()

{

   int X = 10;

   int Y = 20;

}
```

Podemos usar el programa en C++ compilado como guía, mostrando cómo el compilador de C++ asigna las variables locales. Cada entrada en la pila tiene una longitud predeterminada de 32 bits, por lo que el tamaño de almacenamiento de cada variable se redondea hacia arriba, a un múltiplo de 4. Se reserva una total de 8 bytes para las dos variables locales:

Variable	Bytes	Desplazamiento de la pila
X	4	EBP – 4
Y	4	EBP - 8

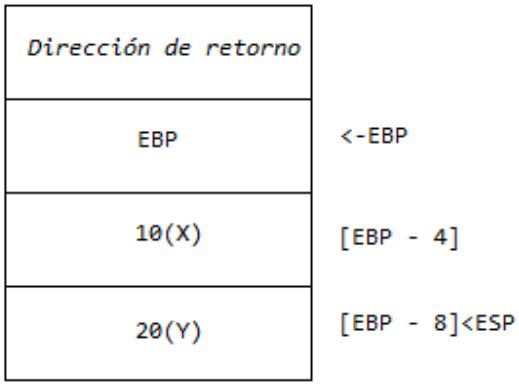
El siguiente desensamblado (mostrado por un depurador) de la función MiSub muestra la forma en que un programa en C++ crea variables locales, asigna valores y elimina las variables de la pola. Utiliza la convención de llamadas de C:

```
MiSub PROC

    push ebp

    mov ebp,esp

    sub esp,8                  ; crea las variables

    mov DWORD PTR [ebp-4],10   ; X
```

```
    mov DWORD PTR [ebp-8],20   ; Y
    mov esp,ebp                ; elimina las variables de la pila
    pop ebp
    ret
MiSub ENDP
```

La siguiente figura muestra el marco de la pila de la función, después de inicializar las variables locales.

Marco de pila, después de crear las variables locales.



Antes de terminar, la función restablece el apuntador de la pila, asignándole el valor de EBP, El efecto es liberar las variables locales de la pila:

```
mov esp,ebp                    ; elimina las variables locales de la pila
```

Si se omitiera este paso, la instrucción POP EBP asignaría 20 a EBP y la instrucción RET se bifurcaría a la ubicación de memoria 10, haciendo que el programa se detenga con una excepción del procesador. Tal es el caso de la siguiente versión de MiSub:

```
MiSub PROC
    push ebp
    mov ebp,esp
    sub esp,8                  ; crea las variables
    mov DWORD PTR [ebp-4],10   ; X
    mov DWORD PTR [ebp-8],20   ; Y
    pop ebp
```

```
    ret
MiSub ENDP
```

Símbolos de las variables locales Con el fin de facilitar la legibilidad de los programas podemos definir un símbolo para el desplazamiento de cada variable local y utilizarlo en nuestro código:

```
X_local EQU DWORD PTR [ebp-4]
Y_local EQU DWORD PTR [ebp-8]
Misub PROC
    push ebp
    mov ebp,esp
    sub esp,8                  ; reserva espacio para las variables
locales
   mov X_local,10              ; X
   mov Y_local,20              ; Y
   mov esp,ebp                 ; elimina las variables de la pila
   pop ebp
   ret
MiSub ENDP
```

Acceso a los parámetros por referencia

Por lo general, las subrutinas acceden a los parámetros por referencia mediante el uso de direccionamiento base-desplazamiento (de EBP). Como cada parámetro por referencia es un apuntador, por lo general, se carga en un registro para usarlo como operando indirecto. Por ejemplo, supongamos que un apuntador a un arreglo se encuentra en la dirección de pila [ebp + 12]. La siguiente instrucción copia el apuntador a ESI:

```
mov esi,[ebp+12]               ; apunta al arreglo
```

Ejemplo: *LlenarArreglo* El procedimiento **LlenarArreglo**, que vamos a ver a continuación, llena un arreglo con una secuencia seudoaleatoria de enteros de 16 bits. Recibe dos argumentos: un apuntador al arreglo y la longitud del arreglo. El primero se pasa por referencia y el segundo se pasa por valor. He aquí una llamada de ejemplo:

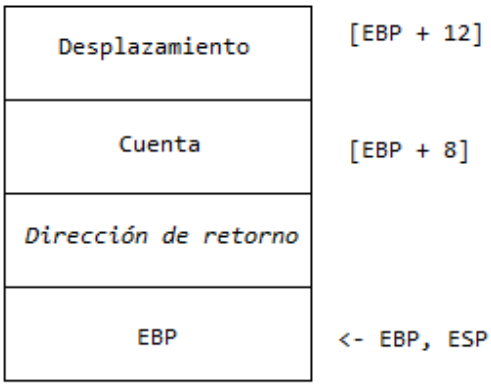
```
.data
```

```
cuenta = 100

arreglo DWORD cuenta DUP (?)

.code

push OFFSET arreglo

push CUENTA

call LlenarArreglo
```

Dentro de **LlenarArreglo**, el siguiente código de prólogo inicializa el apuntador de pila (EBP):

```
LlenarArreglo PROC

   push ebp

   mov ebp, esp
```

Ahora el marco de pila contiene el desplazamiento del arreglo, la cuenta, la dirección de retorno y el registro EBP que se guardo:



LlenarArreglo guarda los registros de propósito general, obtiene los parámetros y llena el arreglo:

```
LlenarArreglo PROC

    push ebp

    mov ebp,esp

    pushad                      ; guarda los registros

    mov esi,[ebp+12]            ; desplazamiento del arreglo

    mov ecx,[ebp+8]             ; tamaño del arreglo
```

```
    cmp ecx,0                   ; ECX==0 ?
    je L2                       ; sí: omite el ciclo
L1:
    mov eax,10000h              ; obtiene un valor al azar entre 0 y FFFFh
    call RandomRange            ; de la biblioteca de enlace
    mov [esi],ax                ; inserta un valor en el arreglo
    add esi, TYPE WORD          ; mueve al siguiente elemento
    loop L1
L2: popad                       ; restaura los registros
     pop ebp
     ret 8                      ; limpia la pila
LlenarArreglo ENDP
```

Instrucción LEA

La instrucción LEA devuelve el desplazamiento de un operando indirecto. Como los operandos indirectos contienen uno o más registros, su desplazamientos se calculan en tiempo de ejecución. Para mostrar cómo puede utilizarse LEA, veamos el siguiente programa en C++, que declara un arreglo local de caracteres y hace referencia a **miCadena** al asignar valores.

```
Void crearArreglo()
{
    Char micadena[30];
    For( int i=0; i < 30; i++)
      miCadena[i] = '*' ;
```

El código equivalente en lenguaje ensamblador asigna espacio para miCadena en la pila, y asigna la dirección a ESI, un operando indirecto. Aunque el arreglo sólo es de 30 bytes, ESP se decrementa por 32 para mantenerlo alineado en un límite de doble palabra. Observemos cómo se utiliza LEA para asignar la dirección del arreglo a ESI:

```
crearArreglo PROC
      push ebp
      mov ebp,esp
```

```
      sub esp,32              ; miCadena está en EBP-32
      lea esi,[ebp-32]        ; carga dirección de miCadena
      mov ecx, 30             ; contador de ciclo
L1: mov BYTE PTR [esi],'*'    ; llena una posición
      Inc esi                 ; mueve a la siguiente posición
      Loop L1                 ; continúa hasta que ECX = 0
      add esp,32              ; elimina el arreglo (restaura ESP)
      pop ebp
      ret
crearArreglo ENDP
```

No es posible utilizar OFFSET para obtener la dirección de un parámetro de la pila ya que OFFSET sóli funciona con las direcciones que se conocen en tiempo de compilación. La siguiente instrucción no se ensamblaría:

```
Mov esi,OFFSET [ebp-30]       ; error
```

Instrucciones ENTER y LEAVE

La instrucción ENTER crea una manera automática un marco de pila para un procedimiento al que se llamó. Reserva espacio en la pila para variables locales y guarda a EBP en la pila. En específico, realiza tres acciones:

- Mete a EBP en la pila (*push ebp*).
- Establece EBP a la base del marco de pila (*mov ebp, esp*).
- Reserva espacio para las variables locales (sub esp, numbytes).

ENTER tiene dos operandos: El primero es una constante que especifica el número de bytes de espacio a reservar en la pila para las variables locales, y el segundo especifica el nivel de anidamiento léxico del procedimiento:

ENTER *numbytes, nivelanidamiento*

Ambos operandos son valores inmediatos. *Numbytes* siempre se redondea hacia arriba a un múltiplo de 4, para mantener ESP en un límite de doble palabra. Nivelanidamiento determina el número de apuntadores al marco de la pila que se copian en el marco de pila actual, provenientes del marco de pila del procedimiento al que se llamó.

Ejemplo 1 El siguiente ejemplo declara un procedimiento sin variables locales:

```
   MiSub PROC
```

```
Enter 0,0
```

Esto es equivalente a las siguientes instrucciones:

```
MiSub PROC
    push ebp
    mov ebp,esp
```

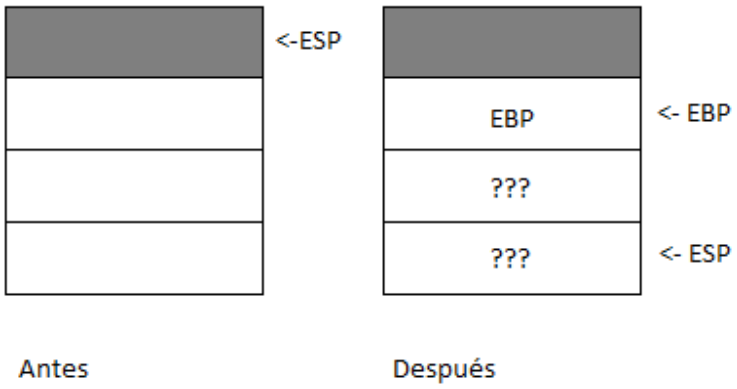
Ejemplo 2 La instrucción ENTER reserva 8 bytes de almacenamiento en la pila par a las variables locales:

```
MiSub PROC
    enter 8,0
```

Esto es equivalente a las siguientes instrucciones:

```
MiSub PROC
    push ebp
    mov ebp,esp
    sub esp,8
```

La siguiente figura muestra antes y después de ejecutar ENTER.



Si se utiliza la instrucción ENTER, es imprescindible que se utilice también la instrucción LEAVE al final del mismo procedimiento. Si no es así, tal vez no se libere el espacio de almacenamiento que haya creado para las variables locales. Esto hará que la instrucción RET saque la dirección de retorno incorrecta de la pila.

Instrucción LEAVE La instrucción LEAVE termina el marco de pila para un procedimiento. Invierte la acción de la instrucción ENTER anterior, restaurando ESP y EBP a los valores que tenían asignados cuando se hizo la llamada al procedimiento. Si utilizamos el ejemplo del procedimiento **MiSub** de nuevo, podemos escribir lo siguiente:

```
MiSub PROC
      enter 8,0
      .
      .
      leave
      ret
MiSub ENDP
```

El siguiente conjunto equivalente de instrucciones invierte y descarta 8 bytes del espacio para las variables locales:

```
MiSub PROC
      push ebp
      mov ebp,esp
      sub esp,8
      .
      .
      mov esp, ebp
      pop ebp
      ret
MiSub ENDP
```

Directiva LOCAL

Podemos suponer que Microsoft creó la directiva LOCAL como un sustituto de alto nivel para la instrucción ENTER. LOCAL declara sólo una o más variables locales por nombre, y les asigna atributos de tamaño. Por otra parte, ENTER sólo reserva un solo bloque sin nombre de espacio en la pila para las variables locales. Si se utiliza, LOCAL debe aparecer en la línea que va justo después de la directiva PROC. Su sintaxis es:

```
LOCAL listavars
```

listavars es una lista de definiciones de variables, separadas por comas, que de manera opcional abarcan varias líneas. Cada definición de variable toma la siguiente forma:

```
etiqueta:tipo
```

La etiqueta puede ser cualquier identificador válido, y el tipo puede ser cualquier tipo estándar (WORD, DWORD, etc.) o un tipo definido por el usuario.

Ejemplos El procedimiento **MiSub** contiene una variable local llamada var1, de tipo BYTE.

```
   MiSub PROC

      LOCAL var1:BYTE
```

El procedimiento **OrdenaBurbuja** contiene una variable tipo doble palabra llamada **temp** y una variable llamada **IntercambiaBandera** de tipo BYTE:

```
OrdenaBurbuja PROC

      LOCAL temp:DWORD, IntercambiaBandera:BYTE
```

El procedimiento **Mezclar** contiene una variable local PTR WORD llamada **pArreglo**, la cual es un apuntador a un entero de 16 bits:

```
Mezclar PROC

      LOCAL pArreglo:PTR WORD
```

La variable local **ArregloTemp** es un arreglo de 10 dobles palabras:

```
      LOCAL ArregloTemp[10]:DWORD
```

Generación de código en MASM

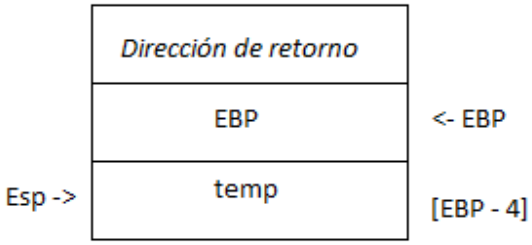
Es conveniente analizar el código generado por MASM cuando se utiliza la directiva LOCAL; para esto se puede ver un desamblado. El siguiente procedimiento **Ejemplo1** tiene una sola variable local de tipo doble palabra:

```
Ejemplo1 PROC

        LOCAL temp:DWORD

        mov eax,temp

        ret

Ejemplo1 ENDP
```

MASM genera el siguiente código para ejemplo1, mostrando cómo se decrementa ESP por 4, para así dejar espacio a la variable tipo doble palabra:

```
push ebp

mov ebp,esp

add esp,0FFFFFFFCh              ; suma -4 a ESP

mov eax,[ebp-4]

leave

ret
```

He aquí un diagrama del marco de pila del Ejemplo1:



Variables locales que no son de doble palabra

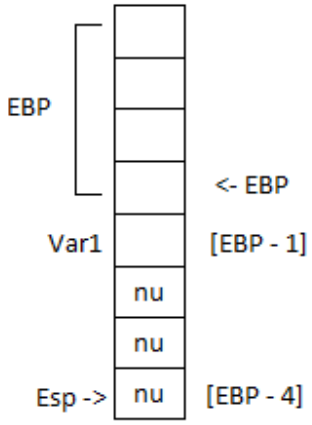
La directiva LOCAL tiene un comportamiento interesante cuando se declaran variables locales de distintos tamaños. A cada una se les asigna un espacio de acuerdo a su tamaño: una variable de 8 bits se asigna el siguiente byte disponible, una variable de 16 bits se le asigna a la siguiente dirección par (alineada por palabra), y a una variable de 32 bits se le asigna el siguiente límite alineado por doble palabra.

Veamos unos cuantos ejemplos. En primer lugar, el procedimiento **Ejemplo1** contiene una variable local llamada **var1** de tipo BYTE:

```
Ejemplo1 PROC

   Local var1:BYTE

   mov al,var1                  ; [EBP – 1]

   ret

Ejemplo1 ENDP
```

Como el desplazamiento predeterminado de la pila es de 32 bits, uno podría esperar que **var1** se encontrara en EBP – 4. En su lugar, como se muestra en la siguiente figura. MASM decrementa a ESP por 4 y coloca a **var1** en EBP – 1, dejando los tres bytes debajo de esta ubicación sin usar (están marcados con la letra un).

Creación de variables locales (Procedimiento Ejemplo1).



El procedimiento **Ejemplo2** contiene una doble palabra, seguida de un byte:

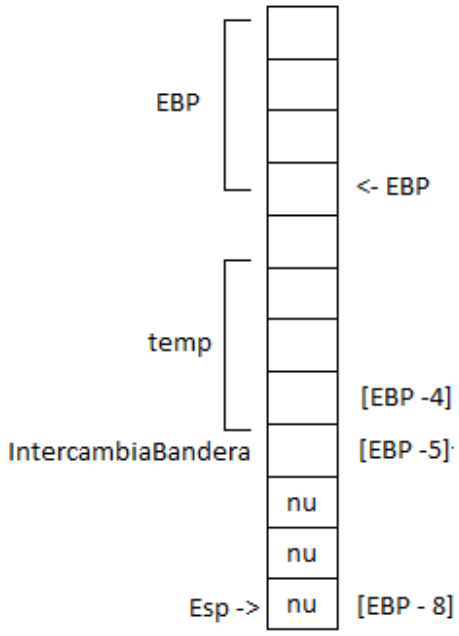
```
   Ejemplo2 PROC

      LOCAL temp:DWORD, IntercambiaBandera:BYTE

      .

      .

      Ret

Ejemplo2 ENDP
```

El siguiente código lo genera MASM para el Ejemplo2. La instrucción ADD suma -8 a ESP, creando una abertura en la pila entre ESP y EBP, para las dos variables locales:

```
push ebp

mov ebp,esp

add esp,0FFFFFFF8h              ; suma -8 a ESP

mov eax,[ebp-4]                 ; temp

mov eax,[ebp-5]                 ; IntercambiaBandera
```

Aunque **IntercambiaBandera** es sólo un byte, ESP se redondea hacia abajo, a la siguiente ubicación tipo doble palabra de la pila. En la siguiente figura se muestra una vista detallada de la pila, en forma de byte individuales, para indicar la ubicación exacta de IntercambiaBandera y el espacio sin usar debajo de ésta (etiquetado con *un*).

Creación de espacio en el ejemplo 2 para variables locales.



Reservación de espacio extra en la pila. Si se planea crear arreglos más grandes que unos cuantos cientos de bytes como variables locales, debemos asegurarnos de reservar el espacio adecuado para la pila en tiempo de ejecución, usando la directiva STACK:

```
.STACK 4096
```

Si las llamadas a procedimientos están anidadas, la pila en tiempo de ejecución debe ser lo bastante grande como para guardar la suma de todas las variables locales activas en cualquier punto de la ejecución del programa, supongamos que **Sub1** llama a **Sub2** y **Sub2** llama a **Sub3**. Cada uno de estos procedimientos podría tener una variable local tipo arreglo:

```
Sub1 PROC

    LOCAL arreglo[50]:DWORD    ; 200 bytes

.

.

Sub2 PROC

    LOCAL arreglo2[80]:DWORD   ; 160 bytes

.

.

Sub3 PROC

    LOCAL arreglo3[300]:BYTE   ; 300 bytes
```

Cuando el programa entra a **Sub3**, la pila en tiempo de ejecución guarda las variables locales de **sub1**, **sub2** y **sub3**. La pila requeriría 660 bytes utilizados por las variables locales, más las dos direcciones de retorno de los procedimientos (8bytes), y cualquier registro que puediera haber metido a la pila dentro de los procedimientos.

Recursividad

Una subrutina recursiva es una que se llama a si misma, ya sea en forma directa o indirecta. *La recursividad*, que es la práctica de llamar funciones recursivas, puede ser una poderosa herramienta al trabajar con estructuras de datos que tienen patrones repetitivos. Algunos ejemplos son las listas enlazadas y varios tipos de gráficos conectados, en los que un programa debe volver a trazar su ruta.

Recursividad sin fin El tipo más obvio de recursividad ocurre cuando una subrutina se llama a sí misma. Por ejemplo, el siguiente programa tiene un procedimiento llamado **sinfin**, el cual se llama a sí mismo repetidas veces sin detenerse:

```
TITLE Recursividad sin fin     (SinFin.asm)

INCLUDE Irvine32.inc

.data

cadSinFin BYTE “Esta recursividad nunca termina”,0

.code

main PROC

      call sinfín

      exit

main ENDP

SinFin PROC

      mov EDX,offset cadSinFin

      call WriteString

      call sinfín

      ret                      ; nunca llega a esta línea

SinFin ENDP

END main
```

Desde luego que este ejemplo no tiene ningún valor práctico. Cada vez que el procedimiento se llama a sí mismo, utiliza 4 bytes de espacio en la pila cuando la instrucción CALL mete la dirección de retorno. La instrucción RET nunca se ejecuta.

Calculo recursivo de una suma

Las subrutinas recursivas útiles siempre contienen una condición de terminación. Cuando esta condición de terminación se vuelve verdadera, la pila se limpia cuando el programa ejecuta todas las instrucciones RET pendientes. Para ilustrar esto, vamos a considerar el procedimiento recursivo llamado **CalcSuma**, que suma los enteros de 1 a *n*, en donde *n* es un parámetro de entrada que se pasa en ECX. CalcSuma devuelve la suma en EAX:

```
TITLE Suma de enteros          (Csuma.asm)

INCLUDE Irvine32.inc

.code

main PROC

      mov ecx,10               ; cuenta = 10

      mov eax,0                ; guarda la suma

      call CalcSuma            ; calcula la suma

L1:   call WriteDec            ; muestra EAX

      call Crlf                ; nueva línea

     exit

main ENDP

;-----------------------------------

CalcSuma PROC

; Calcula la suma de una lista de enteros

; Recibe: ECX = Cuenta

; Devuelve: EAX = suma

; ----------------------------------------

      cmp ecx,0                ; comprueba el valor del contador

      jz L2                    ; termina si es cero

      add eax,ecx              ; en caso contrario, le agrega a la suma
```

```
      dec ecx                 ; decrementa el contador

      call CalcSuma           ; llamada recursiva

L2: ret

calcSuma ENDP

END Main
```

Las primeras dos líneas de **CalcSuma** comprueban el contador y salen del procedimiento cuando ECX = 0. El código pasa por alto las subsiguientes llamadas recursivas. Cuando se llega a la instrucción RET por primera vez, regresa a la llamada anterior a CalcSuma, la cual regresa a su llamada anterior y así sucesivamente. La siguiente tabla muestra las direcciones de retorno (como etiquetas) que se meten en la pila mediante la instrucción CALL, junto con los valores concurrentes de ECX (contador) y EAX (suma).

Incluso hasta un simple procedimiento recursivo hace uso de la pila. Como mínimo, se utilizan cuatro bytes de espacio de la pila cada vez que se realiza una llamada al procedimiento, ya que la dirección de retorno debe guardarse en la pila.

Se metió en la pila	Valor en ECX	Valor en EAX
L1	5	0
L2	4	5
L2	3	9
L2	2	12
L2	1	14
L2	0	15

Cálculo de una factorial

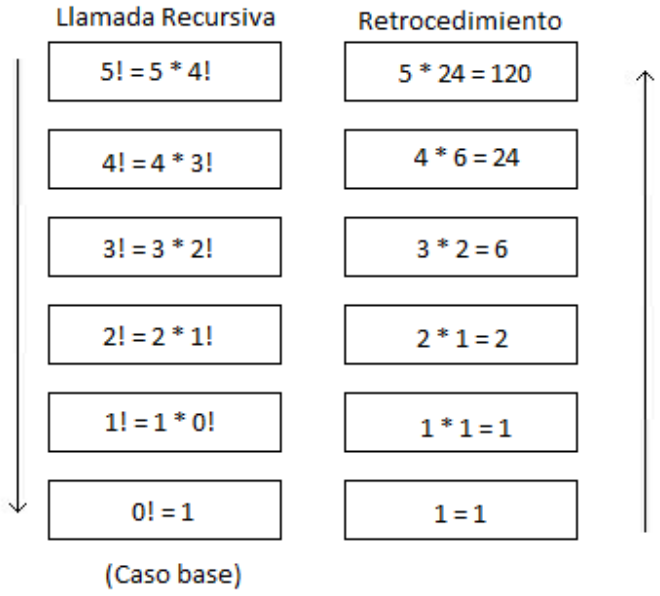
A menudo, las llamadas recursivas almacenan los datos temporales en el parámetro de pila. Cuando las llamadas recursivas se limpian, los datos en la pila pueden ser útiles. El siguiente ejemplo que vamos a analizar calcula la factorial de un entero *n*. El algoritmo *factorial* calcula *n!*, en donde *n* es un entero sin signo. La primera vez que se llama a la función **factorial**, el parámetro *n* es el número inicial, que se muestra programado a continuación en sintaxis de C/C++/Java:

```
int function factorial(int n)

{
```

```
    if(n == 0)
       return 1;
    else
       return n * factorial(n-1);
}
```

Dado cualquier número *n*, asumimos que podemos calcular la factorial de *n -1*. Si es así, podemos continuar reduciendo *n* hasta que sea igual a cero. Por definición, 0! es igual a 1. En el proceso de retroceder hasta la expresión original *n!*, acumulamos el producto de cada multiplicación. Por ejemplo, calcular 5!, el algoritmo recursivo desciende a lo largo de la columna izquierda de la siguiente figura y retrocede a lo largo de la columna derecha.

Llamadas recursivas a la función Factorial.



Programa de ejemplo El siguiente programa en lenguaje ensamblador contiene un procedimiento llamado **Factorial**, el cual utiliza la recursividad para calcular un factorial. Pasamos *n* (un entero sin signo entre el 0 y 12) en la pila al procedimiento **Factorial**, y se devuelve un valor en EAX. Como se utiliza un registro de 32 bits, el mayor factorial que puede guardar es 12! (479,001,600).

```
TITLE Cálculo de un factorial (Fact.asm)

INCLUDE Irvine32.inc

.code
```

```
main PROC
      push 12                  ; calcula el factorial de 12
      call Factorial           ; calcula factorial (EAX)
RetornoMain:
      call WriteDec            ; lo muestra
      call Crlf
      exit
main ENDP
;--------------------------------
Factorial PROC
; Calcula un factorial.
; Recibe [EBP+8] = n, el número a calcular
; Devuelve: eax = el factorial de n
;--------------------------------
      push ebp
      mov ebp,esp
      mov eax,[ebp+8]          ; obtiene n
      cmp eax,0                ; n > 0?
      ja L1                    ; sí: continúa
      mov eax,1                ; no: regresa a 1
      jmp L2
L1: dec eax
      push eax                 ; Factorial (n-1)
      call Factorial
; Las instrucciones aquí en adelante se ejecutan cuando
; regresa cada una de las llamadas recursivas.
```

```
RetornoFact:
      mov ebx,[ebp+8]          ; obtiene n
      mul ebx                  ; EDX: EAX = EAX * EBX
L2: pop ebp                    ; devuelve EAX
      ret 4                    ; limpia la pila
Factorial ENDP
END main
```

Cuando se hace la llamada a **Factorial**, el desplazamiento de la siguiente instrucción después de la llamada se mete en la pila. Desde **main**, éste es el desplazamiento de la etiqueta **RetornoMain**; desde **Factorial**, es el desplazamiento de la etiqueta **RetornoFact**.

Directiva .MODEL

MASM utiliza la directiva .MODEL para determinar varias características importantes de un programa: su tipo de modelo de memoria, el esquema de nomenclatura de procedimientos, y la convención para el paso de parámetros. Los últimos dos son en especial importantes cuando los programas escritos en otro lenguaje de programación llaman al lenguaje ensamblador. La sintaxis de la directiva .MODEL es:

```
.MODEL modelomemoria [,opcionesmodelo]
```

Modelo de memoria El campo *modelomemoria* puede ser uno de los modelos descritos en la siguiente tabla, Todos los modos, con la excepción del plano, se utilizan cuando se programa en modo direccionamiento real de 16 bits.

Modelo	Descripción
Tiny (diminuto)	Un solo segmento, contiene código y datos. Este modelo lo utilizan los programas que tienen la extensión .com en su nombre de archivo.
Small (pequeño)	Un segmento de código y un segmento de datos. Todo el código y los datos son cercanos, de manera predeterminada.
Medium (mediano)	Varios segmentos de código y un solo segmento de datos.
Compact (compacto)	Un segmento de código y varios segmentos de datos.
Large (grande)	Varios segmentos de código y de datos.
Huge (enorme)	Igual que el modelo grande (large), pero los elementos individuales de datos pueden ser más grandes que un solo

	segmento.
Flat (plano)	Modo protegido. Utiliza desplazamientos de 32 bits para el código y los datos. Todos los datos y el código (incluyendo los recursos del sistema) se encuentran en un solo segmento de 32 bits.

Todos los programas en modo de direccionamiento real que hemos mostrado hasta ahora, utilizan el modelo de memoria pequeño, ya que mantiene todo el código dentro de un solo segmento, y todos los datos (incluyendo la pila) dentro de un solo segmento. Como resultado, sólo tenemos que manipular los desplazamientos de código y datos, y los segmentos nunca cambian.

Los programas en modo protegido utilizan el modelo de memoria plana, en el cual los desplazamientos son de 32 bits, y el código y los datos pueden ser hasta de 4 GB. Por ejemplo Irvine32.inc contiene la siguiente directiva . MODEL:

```
.MODEL flat,STDCALL
```

Opciones de modelo El campo *opcionesmodelo* en la directiva .MODEL puede contener un especificador de lenguaje y una distancia de pila. El *especificador de lenguaje* determina las convenciones de llamadas y de nomenclatura para los procedimientos y símbolos públicos. La *distancia de pila* puede ser NEARSTACK (el valor predeterminado) o FARSTACK.

Especificadores de lenguaje

Vamos a ver más de cerca los especificadores de lenguaje utilizados en la directiva .MODEL. Las opciones son C, BASIC, FORTRAN, PASCAL, SYSCALL y STDCALL. Los especificadores C, BASIC, FORTRAN y PASCAL permiten a los programadores de lenguaje ensamblador crear procedimientos que sean compatibles con estos lenguajes. Los especificadores SYSCALL y STDCALL son variaciones de los otros especificadores de lenguaje. En este resumen, demostraremos los especificadores C y STDCALL. Cada uno de ellos se muestra a continuación, con el modelo plano de memoria:

```
.model flat, C

.model flat, STDCALL
```

STDCALL Se utiliza en la mayoría de nuestros programas de ejemplo. Es el especificador de lenguaje que se utiliza al llamar a las funciones de MS Windows. Más adelante utilizaremos el especificador de lenguaje C, al enlazar el código de lenguaje ensamblador con los programas en C y C++.

STDCALL

El especificador de lenguaje STDCALL, hace que los argumentos de una rutina se metan en la pila en orden inverso (del último al primero). Supongamos que escribimos la siguiente llamada a una función en lenguaje de alto nivel:

```
SumarDos( 5, 6);
```

El siguiente código en lenguaje ensamblador es equivalente

```
push 6

push 5

call SumarDos
```

Otra consideración importante es la forma en que los argumentos se sacan de la pila, después de las llamadas a procedimientos. STDCALL requiere que se proporcione un operando constante en la instrucción RET. Este operando indica el valor que se suma a ESP después de que RET saca la dirección de retorno de la pila:

```
SumarDos Proc
        push ebp
        mov ebp,esp
        mov eax,[ebp + 12]      ; primer parámetro
        mov eax,[ebp + 8]       ; segundo parámetro
        pop ebp
        ret 8
SumarDos ENDP
```

Al sumar 8 al apuntador de la pila, lo restablecemos al valor que tenía antes de meter los argumentos en la pila mediante el programa que hace la llamada.

Por último STDCALL modifica los nombres de los procedimientos exportados (public), y los almacena en el siguiente formato:

```
_nombre@nn
```

Se agrega un guión bajo a la izquierda del nombre del procedimiento y se coloca un entero después del signo @, indicando el número de bytes utilizados por los parámetros del procedimiento (se redondean hacia arriba, a un múltiplo de 4). Por ejemplo, supongamos que el procedimiento **SumarDos** tiene dos parámetros de tipo doble palabra. El nombre que pasa el ensamblador al enlazador es **_SumarDos@8**.

La herramienta LINK32.EXE es sensible al uso de mayúsculas y minúsculas, por lo que _MISUB@8 es distinto a _MiSub@8. Para ver todos los nombres de los procedimientos dentro de un archivo .OBJ, podemos usar la herramienta DUMPBIN que se proporciona en Visual Studio.

Especificador C

El especificador de lenguaje C requiere que los argumentos del procedimiento se metan en la pila, del último al primero, de igual forma que STDCALL. Con respecto a la eliminación de argumentos de la pila después de la llamada a un procedimiento, el especificador de lenguaje C deja esa responsabilidad al que hace la llamada. En el programa que hace la llamada, se suma una constante en ESP, con lo cual se restablece el valor que tenía antes de meter los argumentos:

```
push 6                          ; segundo argumento

push 5                          ; primer argumento

call SumarDos

add esp,8                       ; límpia la pila
```

El especificador de lenguaje C adjunta un carácter de guión bajo a la izquierda de los nombre del procedimientos externos. Por ejemplo:

_SumarDos

INVOKE, ADDR, PROC Y PROTO

Las directivas INVOKE, ADDR, PROC y PROTO son herramientas poderosas para definir y llamar a los procedimientos. En muchos sentidos, se aproximan a la conveniencia que ofrecen los lenguajes de programación de alto nivel. Desde un punto de vista pedagógico, su uso es controversial ya que enmascaran la estructura subyacente de la pila en tiempo de ejecución.

Hay una situación en la que el uso de directivas de procedimientos avanzados conlleva a una mejor programación: cuando el programa ejecuta llamadas a procedimientos a través de los límites de los módulos. En tales casos la directiva PROTO ayuda al ensamblador a validar las llamadas a los procedimientos, para lo cual se comparan las listas de los argumentos con las declaraciones de los procedimientos. Esta característica alienta a los programadores de lenguaje ensamblador a aprovechar la conveniencia que ofrecen las directivas avanzadas de MASM.

Directiva INVOKE

La directiva INVOKE mete argumentos en la pila (en el orden especificado por el especificador de lenguaje de la directiva MODEL) y llama a un procedimiento. INVOKE es un reemplazado conveniente para la instrucción CALL, ya que nos permite pasar varios argumentos en una sola línea de código. He aquí la sintaxis general:

```
INVOKE nombreProcedimiento [, listaArgumentos]
```

ListaArgumentos es una lista opcional delimitada por comas, de los argumentos que se pasan al procedimiento. Por ejemplo, mediante el uso de la instrucción CALL podríamos llamar a DumpMem después de ejecutar varias instrucciones PUSH

```
push TYPE arreglo

push LENGTHOF arreglo

push OFFSET arreglo

call DumpMem
```

La instrucción equivalente mediante el uso de INVOKE se reduce a una sola línea, en la cual los argumentos se listan en orden inverso (asumiendo que STDCALL está en efecto)

```
INVOKE DumpMem, OFFSET arreglo, LENGTHOF arreglo, TYPE arreglo
```

INVOKE permite casi cualquier número de argumentos, y los argumentos individuales pueden aparecer en líneas de código separadas. La siguiente instrucción INVOKE incluye comentarios útiles:

```
INVOKE DumpMem,
       OFFSET arreglo          ; muestra un bloque de memoria
       LEGTHOF arreglo         ; apunta al arreglo
       TYPE arreglo            ; tamaño de los componentes del arreglo
```

Los tipos de argumentos se listan en la siguiente tabla:

Tipo	Ejemplos
Valor inmediato	10,3000h, OFFSET milista, TYPE arreglo
Expresión entera	(10 * 20), CUENTA
Variable	miLista, arreglo, miPalabra, miDoblePalabra
Expresión de dirección	[miLista+2],[ebx+esi]
Registro	eax, bl, edi
ADDR nombre	ADDR miLista
OFFSET	OFFSET miLista

Sobre escritura de EAX y EDX Si se pasan argumentos menores de 32 bits a un procedimiento, INVOKE ocasiona con frecuencia que el ensamblador sobre escriba el contenido de EAX y EDX al ampliar los argumentos antes de meterlos en la pila. Podemos evitar este comportamiento pasando siempre argumentos de 32 bits a INVOKE, o podemos guardar y restaurar EAX y EDX antes y después de la llamada al procedimiento.

Operador ADDR

El operador ADDR puede utilizarse para pasar un argumento tipo apuntador al llamar a un procedimiento usando INVOKE. Por ejemplo, la siguiente instrucción INVOKE pasa la dirección de **miArreglo** al procedimiento **LlenarArreglo**:

```
INVOKE LlenarArreglo, ADDR miArreglo
```

El argumento que se pasa a ADDR debe ser una constante en tiempo de ensamblado. La siguiente expresión es un error:

```
INVOKE miSub, ADDR [ebp+12]    ; error
```

El operador ADDR sólo puede usarse en conjunto con INVOKE. La siguiente expresión es un error:

```
mov esi, ADDR miArreglo        ; error
```

ADDR pasa un apuntador cercano o lejano, dependiendo de lo que se llame mediante el modelo de memoria del programa. En los programas en modo protegido, ADDR y OFFSET pasan desplazamientos de 32 bits.

Ejemplo La siguiente directiva INVOKE llama a **Intercambiar** y le pasa las direcciones de los primeros dos elementos en un arreglo de dobles palabras:

```
.data

Arreglo DWORD 20 DUP(?)

.code

…

INVOKE Intercambiar,

       ADDR Arreglo,

       ADDR [Arreglo+4]
```

He aquí el código correspondiente que genera el ensamblador, suponiendo que STDCALL está en efecto:

```
push OFFSET Arreglo
```

```
push OFFSET Arreglo+4

call Intercambiar
```

Directiva PROC

Sintaxis de la directiva PROC

La directiva PROC tiene la siguiente sintaxis básica:

```
etiqueta PROC [atributos] [USES listaregs], lista_Parámetros
```

Etiqueta es una etiqueta definida por el usuario, que sigue las reglas para los identificadores que explicamos. *Atributo*s se refiere a cualquiera de los siguientes:

```
[distancia] [tipoleng] [visibildad] [prólogo]
```

La siguiente tabla describe cada uno de los atributos.

Atributo	Descripción
Distancia	NEAR (cercana) o FAR (lejana). Indica el tipo de instrucción RET (RET o RETF) que genera el ensamblador.
Tipoleng	Especifica la convención de llamada (convención de paso de parámetros) tal como C, PASCAL o STDCALL. Ignora el lenguaje especificado en directiva .MODEL
Visibilidad	Indica la visibilidad del procedimiento para con los otros módulos. Las operaciones son PRIVATE, PUBLIC (predeterminada) y EXPORT. Si la visibilidad es EXPORT, el enlazador coloca el nombre del procedimiento en la tabla de exportación para los ejecutables segmentados. EXPORT también habilita la visibilidad PUBLIC.
Prólogo	Especifica los argumentos que afectan la generación del código prólogo y del epílogo.

Lista de parámetros

La directiva PROC nos permite declarar un procedimiento con una lista separada por comas de parámetros con nombre. El código de implementación se puede referir a los parámetros por nombre, en vez de hacerlo por los desplazamientos calculados de la pila como [ebp+8]:

```
etiqueta PROC [atributos] [USES listaregs],

parámetro_1,

parámetro_2,
```

.

.

parámetro_n

Podemos observar las comas requeridas antes del primer parámetro, estas pueden pasarse por alto de manera inadvertida. La lista de parámetros puede aparecer en la misma línea:

etiqueta PROC [atributos], parámetro_1, parámetro_2, . . ., parámetro_n

Un parámetro individual tiene la siguiente sintaxis:

nombreParam: tipo

NombreParam es un nombre arbitrario que se asigna al parámetro. Su alcance está limitado al procedimiento actual (lo que se conoce como alcance *local*). El mismo nombre de parámetro puede usarse en más de un procedimiento, pero no puede ser el nombre de una variable global o etiqueta de código. *Tipo* puede ser uno de los siguientes: BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD o TBYTE.

También puede ser un *tipo calificado*, como un apuntador a un tipo existente. A continuación se muestran ejemplos de tipos calificados:

PTR BYTE	PTR SBYTE
PTR WORD	PTR SWORD
PTR DWORD	PTER SDWORD
PTR QWORD	PTR TBYTE

Aunque es posible agregar atributos NEAR y FAR a estas expresiones, sólo son relevantes en aplicaciones más específicas. También pueden crearse tipos calificados mediante el uso de las directivas TYPEDEF y STRUCT, que se explicará más adelante.

Ejemplo 1 El procedimiento SumarDos recibe dos valores tipo doble palabra y devuelve su suma en EAX:

```
SumarDos PROC,
        val1:DWORD,
        val2:DWORD
        mov eax,val1
        add eax,val2
```

```
        ret
SumarDos ENDP
```

El lenguaje ensamblador que genera MASM al ensamblar SumarDos muestra cómo se traducen los nombres de los parámetros a desplazamientos desde EBP. Se genera un operando constante para la instrucción RET, ya que STDCALL está en efecto:

```
SumarDos PROC
        push ebp
        mov ebp,esp
        mov eax,dword ptr [ebp+8]
        add eax,dword ptr [ebp+0ch]
        leave
        ret 8
SumarDos ENDP
```

Ejemplo 2 El procedimiento LlenarArreglo recibe un apuntador a un arreglo de bytes:

```
LlenarArreglo PROC,
   pArreglo: PTR BYTE
   . . .
LlenarArreglo ENDP
```

Ejemplo 3 El procedimiento Intercambiar recibe dos apuntadores a dobles palabras:

```
Intercambiar PROC,
    pValX:PTR DWORD,
    pValY:PTR DWORD
    . . .
Intercambiar ENDP
```

Ejemplo 4 El procedimiento Leer_Archivo recibe un apuntador a byte llamado pBufer. Tiene una variable local tipo doble palabra llamado **manejadorArchivo**, y guarda dos registros en la pila (EAX y EBX):

```
Leer_Archivo PROC USES eax ebx,
   pBufer:PTR BYTE
   LOCAL  manejadorArchivo:DWORD
   mov esi,pBufer
   mov manejadorArchivo,eax
   .
   .
   ret
Leer_Archivo ENDP
```

El código que genera MASM para Leer_Archivo muestra como se reserva el espacio en la pila para la variable local (manejadorArchivo) antes de meter EAX y EBX (especificadas en la cláusula USES):

```
Leer_Archivo PROC
   push ebp
   mov ebp,esp
   add esp,0FFFFFFFCh          ; crea manejadorArchivo
   push eax                    ; guarda EAX
   push ebx                    ; guarda EBX
   mov esi,dword ptr [ebp+8]   ; pBufer
   mov dword ptr [ebp-4],eax   ; manejadorArchivo
   leave
   ret 4
Leer_Archivo ENDP
```

Instrucción RET modificada por PROC Cuando PROC se utiliza con uno o más parámetros y STDCALL es el protocolo estándar, MASM genera el siguiente código de entrada y salida, asumiendo que PROC tiene n parámetros:

```
push ebp
mov ebp,esp
```

```
.
.
leave
ret (n*4)
```

La constante que aparece en la instrucción RET es el número de parámetros multiplicado por 4 (ya que cada parámetro es una doble palabra). STDCALL es la convención de llamada utilizada para todas las llamadas a funciones de la API de Windows.

Especificación del protocolo de paso de parámetros

Un programa podría llamar a los procedimientos de la biblioteca Irvine32 y, a su vez, contener procedimientos que puedan llamarse desde programas en C++. Para proveer esta flexibilidad, el campo *atributos* de la directiva PROC nos permite especificar la convención de lenguaje para el paso de parámetros. Este campo redefine la convención de lenguaje predeterminado que se especifica en la directiva .MODEL. El siguiente ejemplo declara un procedimiento con la convención de llamadas de C:

```
Ejemplo1 PROC C,
   parm1:DWORD, parm2:DWORD
```

Si ejecutamos el Ejemplo 1 usando INVOKE, el ensamblador genera código consistente con la convención de llamadas de C. De manera similar, si declaramos Ejemplo usando STDCALL, INVOKE genera código consistente con esa convención de lenguaje:

```
Ejemplo1 PROC STDCALL,
   parm1:DWORD, parm2:DWORD
```

Directiva PROTO

La directiva PROTO crea un prototipo para un procedimiento existente. Un *prototipo* declara el nombre y la lista de parámetros de un procedimiento. Nos permite llamar a un procedimiento antes de definirlo y verificar que el número y tipos de argumentos concuerden con la definición del procedimiento. Los lenguajes C y C++ utilizan prototipos de funciones para validar las llamadas a funciones en tiempo de compilación.

MASM requiere un prototipo para cada procedimiento llamado por INVOKE. PROTO debe aparecer primero, es decir antes que INVOKE. En otras palabras, el orden estándar de estas directivas es:

```
MiSub PROTO                    ; prototipo de procedimiento
INVOKE MiSub                   ; llamada a procedimiento
```

```
MiSub PROC                    ; implementación de procedimiento
  .
  .
MiSub ENDP
```

Es posible un escenario alternativo: La implementación del procedimiento puede aparecer en el programa, antes de la ubicación de la instrucción INVOKE para ese procedimiento. En ese caso, PROC actúa como su propio prototipo:

```
MiSub PROC                    ; definición del procedimiento
  .
  .
MiSub ENDP

INVOKE MiSub                  ; llamada a procedimiento
```

Suponiendo que ya se ha escrito un procedimiento específico, se puede crear un prototipo con facilidad copiando la instrucción PROC y haciendo los siguientes cambios:

- Cambiamos la palabra PROC a PROTO
- Eliminamos el operador USES si lo hay, junto con su lista de registros.

Por ejemplo, supongamos que ya hemos creado el procedimiento **SumaArreglo**.

```
SumaArreglo PROC USES esi ecx,
ptrArreglo:PTR DWORD,         ; apunta al arreglo
tamArreglo:DWORD              ; tamaño del arreglo
  .
  .
SumaArreglo ENDP
```

Ésta es la declaración PROTO correspondiente:

```
SumarArreglo PROTO,
   ptrArreglo:PTR DWORD,      ; apunta al arreglo
   tamArreglo:DWORD           ; tamaño del arreglo
```

La directiva PROTO nos permite redefinir el protocolo de paso de parámetros predeterminado en la directiva .MODEL. Se debe ser consistente con la declaración PROC del procedimiento:

```
Ejemplo1 PROTO C,
   parm1:DWORD, parm2:DWORD
```

Comprobación de argumentos en tiempo de ensamblado

La directiva PROTO ayuda al ensamblador a comparar una lista de argumentos en una llamada al procedimiento, con la definición del mismo. La calidad de la comprobación de errores no es tan buena como la de C y C++. En vez de ello, MASM comprueba el número correcto de parámetros y, hasta cierto punto, relaciona los tipos de los argumentos con los tipos de los parámetros. Por ejemplo, supongamos que el prototipo para **Sub1** se declara así:

```
Sub1 PROTO, p1:BYTE, p2:WORD, p3:PTR BYTE
```

Vamos a definir las siguientes variables:

```
.data
byte_1   BYTE 10h
palab_1  WORD 2000h
palab_2   DWORD 3000h
dipalab_1 DWORD 12345678h
```

La siguiente llamada a Sub1 es válida:

```
   INVOKE SUB,1 byte_1, palab_1, ADDR byte_1
```

El código que genera MASM para esta instrucción INVOKE muestra que los argumentos se meten en la pila en orden inverso:

```
  push 404000h                    ; apuntador a byte_1
  sub esp,2                       ; rellena la pila con 2 bytes
  push word ptr ds:[00404001]     ; valor de palab_1
  mov al,byte ptr ds:[00404000h]  ; valor de byte_1
  push eax
  call 00401071
```

EAX se sobrescribe y la instrucción **sub esp,2** rellena la subsiguiente entrada de la pila hasta 32 bits.

Errores detectados por MASM Si un argumento excede el tamaño de un parámetro declarado, MASM genera un error:

```
INVOKE Sub1, palab_1, palab_2, ADDR byte_1    ; error en arg 1
```

MASM genera errores si invocamos a Sub1 usando muy pocos o demasiados argumentos:

```
INVOKE Sub1, byte_1, palab_2    ; error: muy pocos argumentos

INVOKE Sub1, byte_1, palab_2, ADDR byte_1, palab_2     ; error:
demasiados argumentos
```

Errores que MASM no detecta Si el tipo de un argumento es más pequeño que un parámetro declarado MASM no detecta un error:

```
INVOKE Sub1, byte_1, byte_1, ADDR byte_1
```

En vez de ello, MASM expande el argumento más pequeño hasta el tamaño del parámetro declarado. En el siguiente código generado por nuestro ejemplo de INVOKE, el segundo argumento (byte_1) se expande hasta EAX, antes de meterlo en la pila:

```
push 40400h                 ; dirección de byte_1

mov al,byte ptr ds:[00404000h]  ; valor de byte_1

movzx eax,al                ; lo expande hasta EAX

push eax                    ; lo mete en la pila

mov al,byte ptr ds:[00404000h]  ; valor de byte_1

push eax                    ; lo mete en la pila

call 00401071               ; llama a Sub1
```

Si se pasa una doble palabra cuando se espera un apuntador, no se detecta un error. Por lo general, este tipo de error produce un error en tiempo de ejecución, cuando la subrutina trata de utilizar el parámetro de pila como apuntador:

```
INVOKE Sub1, byte_1, palab_2, dpalab_1      ; no se detecta un error
```

Ejemplo: SumaArreglo

Vamos a revisar el procedimiento SumaArreglo, que calcula la suma de un arreglo de dobles palabras. En un principio, pasábamos los argumentos en registros; ahora podemos usar la directiva PROC para declarar los parámetros de pila:

```
SumarArreglo PROC USES esi ecx,
```

```
    ptrArreglo: PTR DWORD,     ; apunta al arreglo
    tamArreglo:DWORD           ; tamaño del arreglo
    mov esi,ptrArreglo         ; dirección del arreglo
    mov ecx,tamArreglo         ; tamaño del arreglo
    mov eax,0                  ; establece la suma a cero
    cmp ecx,0                  ; ¿longitud = cero?
    je L2                      ; si: termina
L1: add eax,[esi]              ; agrega cada entero a la suma
    add esi,4                  ; apunta al siguiente entero
    loop L1                    ; repite para el tamaño del arreglo
L2: ret                        ; la suma está en eax
SumaArreglo ENDP
```

La instrucción INVOKE llama SumaArreglo y le pasa la dirección de un arreglo, junto con el número de elementos que contiene:

```
.data
arreglo DWORD 10000h,20000h,30000h,40000h,50000h
laSuma DWORD ?
.code
main PROC
   INVOKE SumaArreglo,
     ADDR arreglo,             ; dirección del arreglo
     LENGTHOF arreglo          ; número de elementos
   mov laSuma,eax              ; almacena la suma
```

Clasificaciones de parámetros

Por lo general, los parámetros de los procedimientos se clasifican de acuerdo con la dirección de la transferencia de datos entre el programa que hace la llamada y el procedimiento al cual se llamo:

- ***Entrada***: un parámetro de entrada son los datos que el programa que hizo la llamada pasa al procedimiento. El procedimiento al que se llamó no debe modificar la variable correspondiente al parámetro y, aún si lo hace, la modificación queda confinada al propio procedimiento.
- ***Salida***: un parámetro de salida se crea cuando un programa que llama pasa la dirección de la variable a un procedimiento, el cual utiliza la dirección para localizar y asignar datos a la variable. Por ejemplo, la Biblioteca de consola Win32 tiene una función llamada **ReadConsole**, la cual lee una cadena de caracteres desde el teclado. El programa que hace la llamada a un procedimiento pasa un apuntador a un buffer de cadena, en el que ReadConsole almacena el texto escrito por el usuario:
  .data
  búfer BYTE 80 DUP(?)
  manejadorEntrada DWORD ?
  .code
  INVOKE ReadConsole, manejadorEntrada, ADDR búfer, (etc.)

1. ***Entrada-Salida***: un parámetro de entrada-salida es idéntico a un parámetro de salida, con una excepción: el procedimiento que se llamó espera una variable a la que hace referencia el parámetro contenga datos. También se espera que el procedimiento modifique la variable a través del apuntador.

Ejemplo: Intercambio de dos enteros

El siguiente programa intercambia el contenido de dos enteros de 32 bits. El procedimiento intercambiar tiene dos parámetros de entrada-salida llamados **pValX** y **pValY**, los cuales contienen las direcciones de los datos que se van a intercambiar:

```
TITLE Ejemplo del procedimiento Intercambiar (Intercambiar.asm)

INCLUDE Irvine32.inc

Intercambiar PROTO, pValX:PTR DWORD, pValY:PTR DWORD

.data

Arreglo DWORD 10000h,20000h

.code

main PROC

   ; Muestra el arreglo antes del intercambio:

   mov esi,OFFSET Arreglo

   mov ecx,2                    ; cuenta = 2

   mov ebx, TYPE Arreglo
```

```
   call DumpMem                ; vacía los valores del arreglo
   INVOKE Intercambiar, ADDR Arreglo, ADDR [Arreglo+4]
   ; Muestra el arreglo después del intercambio:
   call DumpMem
   exit
main ENDP
;-----------------------------------------
Intercambiar PROC USES eax esi edi,
   pValX:PTR DWORD,            ; apuntador al primer entero
   pValY:PTR DWORD             ; apuntador al segundo entero
;
; Intercambia los valores de dos enteros de 32 bits
; Devuelve: nada
;-----------------------------------------
   mov esi,pValX               ; obtiene los apuntadores
   mov edi,pValY
   mov eax,[esi]               ; obtiene el primer entero
   xchg eax,[edi]              ; lo intercambia con el segundo
   mov [esi],eax               ; sustituye el primer entero
   ret                         ; PROC genera RET 8
Intercambiar ENDP
END main
```

Los dos parámetros en el procedimiento Intercambiar, **pValX** y **pValY**, son parámetros de entrada-salida. Sus valores existentes *entran* al procedimiento, y sus nuevos valores también *salen* del procedimiento. Como estamos usando PROC son parámetros, el ensamblador cambia la instrucción RET al final de intercambiar a **RET 8** (suponiendo que STDCALL es la convención de llamadas).

Creación de programas con varios módulos

Los archivos de código extenso son difíciles de manipular y lentos para ensamblarse. Podemos dividir un solo archivo en varios archivos de inclusión, pero la modificación a cualquiera de los archivos de código fuente de todas maneras requeriría que se vuelvan a ensamblar todos los archivos. Un mejor método es dividir un programa en *módulos* (unidades ensambladas). Cada módulo se ensambla de manera independiente, por lo que un cambio en el código de fuente de un módulo sólo requiere que se vuelva a ensamblar ese módulo individual. El enlazador combina todos los módulos ensamblados (archivos OBJ) en un solo archivo ejecutable con bastante rapidez. Para enlazar grandes cantidades de módulos objeto se requiere mucho menos tiempo que ensamblar el mismo número de archivos de código de fuente.

Hay dos métodos generales para crear varios módulos: El primero es el tradicional, en el que se utiliza la directiva EXTERN, la cual es más o menos portable a través de distintos ensambladores del 80x86. El segundo método es utilizar la directiva avanzada INVOKE y PROTO de Microsoft, que simplifica las llamadas a procedimientos y ocultan ciertos detalles de bajo nivel.

Ocultar y exportar nombres de procedimientos

De manera predeterminada, MASM hace todos los procedimientos públicos, con lo cual se pueden llamar desde cualquier otro módulo dentro del mismo programa. Podemos redefinir este comportamiento mediante el calificador PRIVATE:

```
miSub PROC PRIVATE
```

Al hacer los procedimientos privados, utilizamos el principio de *encapsulamiento* para ocultar los procedimientos dentro de módulos y evitar conflictos de nombres potenciales, cuando los procedimientos en distintos módulos tienen los mismos nombres.

Directiva OPTION PROC:PRIVATE Otra manera de ocultar procedimientos dentro de un módulo de código fuente es colocar la directiva OPTION PROC:PRIVATE en la parte superior del archivo. Todos los procedimientos se volverán privados de manera predeterminada. Después, podemos usar la directiva PUBLIC para identificar cualquier procedimiento que se desee exportar:

```
OPTION PROC:PRIVATE

PUBLIC MiSub
```

La directiva PUBLIC recibe una lista de nombres delimitados por comas:

```
PUBLIC sub1,sub2,sub3
```

De manera alternativa, podemos designar procedimientos individuales como públicos:

```
miSub PROC PUBLIC

.

miSub ENDP
```

Si utilizamos OPTION PROC:PRIVATE en el módulo de arranque de nuestro programa, debemos asegurarnos de asignar nuestro procedimiento de arranque (main) como PUBLIC, o el cargador del sistema operativo no podrá encontrarlo. Por ejemplo,

```
main PROC PUBLIC
```

Llamadas a procedimientos externos

La directiva EXTERN, que se utiliza al llamar a un procedimiento que está fuera del módulo, identifica el nombre y el tamaño del marco de pila de ese procedimiento. El siguiente ejemplo llama a **sub1**, que se encuentra en un módulo externo:

```
INCLUDE Irvine32.inc

EXTERN Sub1@0:PROC

.code

main PROC

   call sub1@0

   exit

main ENDP

END main
```

Cuando el ensamblador descubre que falta un procedimiento en un archivo de código fuente (identificado mediante una instrucción CALL), su comportamiento predeterminado es generar un mensaje de error. En vez de ello EXTERN indica al ensamblador que debe crear una dirección en blanco para el procedimiento. El enlazador resuelve la dirección faltante al crear el archivo ejecutable del programa.

El sufijo **@n** al final del nombre un procedimiento identifica el espacio total de la pila utilizando los parámetros declarados. Si se utiliza la directiva PROC básica sin parámetros declarados, el sufijo en cada nombre de procedimiento en EXTERN será **@0**. Si se declara un procedimiento usando la directiva PROC, debemos agregar 4 bytes por cada parámetro. Supongamos que declaramos a **SumarDos** con dos parámetros tipo doble palabra:

```
SumarDos PROC,

   val1:DWORD

   val2:DWORD

   …

SumarDos ENDP
```

La correspondiente directiva EXTERN es **EXTERN SumarDos@8:PROC**. Si se planea llamar a SumarDos mediante INVOKE, se debe usar la directiva PROTO en vez de EXTERN:

```
SumarDos PROTO,
   val1:DWORD,
   val2:DWORD
```

Uso de variables y símbolos a través de los límites de los módulos

Exportación de variables y símbolos

De manera predeterminada, las variables y los símbolos son privados para los módulos en los que se definen. Podemos utilizar la directiva PUBLIC para exportar nombres específicos, como en el siguiente ejemplo:

```
PUBLIC cuenta, SIM1

SIM1 = 10

.data

cuenta DWORD 0
```

Acceso a las variables y símbolos externos

Podemos utilizar la directiva EXTERN para acceder a las variables y símbolos definidos en módulos externos:

```
EXTERN nombre : tipo
```

Para los símbolos (definidos con EQU y =), *tipo* debe ser ABS. Para variables, *tipo* puede ser un atributo de definición de datos como BYTE, WORD, DWORD o SDWORD, incluyendo PTR. He aquí algunos ejemplos:

```
EXTERN uno:WORD, dos:SDWORD, tres:PTR BYTE, cuatro:ABS
```

Uso de un archivo INCLUDE con EXTERNDEF

MASM cuenta con la útil directiva llamada EXTERNDEF, la cual sustituye tanto a PUBLIC como a EXTERN. Puede colocarse en un archivo de texto y copiarse en cada uno de los módulos del programa, mediante el uso de la directiva INCLUDE. Por ejemplo, vamos a definir un archivo llamado *vars.inc* que contiene la siguiente declaración:

```
; vars.inc

EXTERNDEF cuenta:DWORD, SIM:ABS
```

Ahora vamos a crear un archivo de código de fuente llamado *sub1.asm*, que contenga a **cuenta** y **SIM1**, y una instrucción INCLUDE para copiar vars.inc al flujo de compilación.

```
TITLE sub1.asm

.386

.model flat,STDCALL

INCLUDE vars.inc

SIM1 = 10

.data

cuenta DWORD 0

END
```

Como éste no es el módulo de arranque del programa, omitimos una etiqueta de punto de entrada al programa en la directiva END, y no necesitamos declarar una pila en tiempo de ejecución.

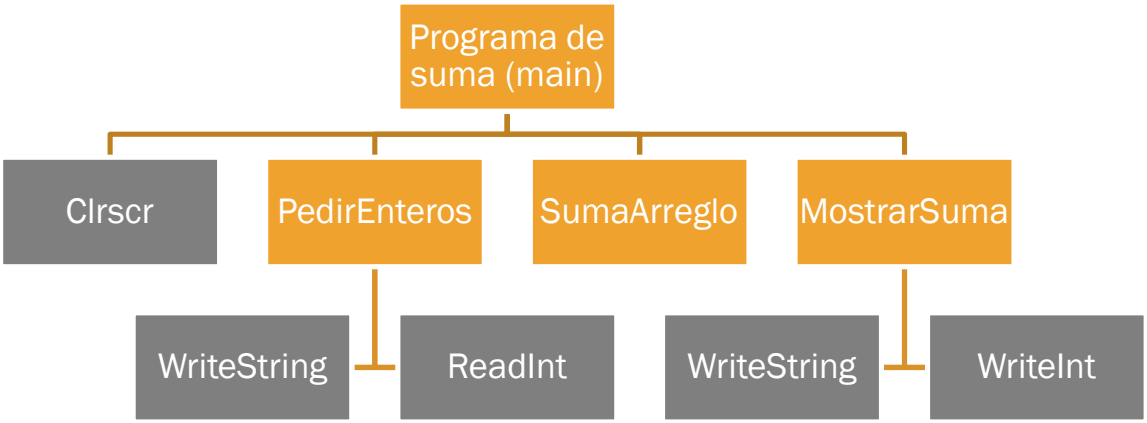
A continuación, vamos a crear un módulo llamado main.asm que incluya a vars.inc y haga referencia a cuenta y SIM1:

```
TITLE main.asm

INCLUDE Irvine32.inc

INCLUDE Vars.inc

.code

main PROC

   mov cuenta,2000h

   mov eax,SIM1

   exit

main ENDP

END main
```

Este módulo contiene una pila en tiempo de ejecución, que se declara con la directiva .STACK dentro de Irvine32.inc, también define el punto de entrada del programa en la directiva END.

Ejemplo: programa SumaArreglo

El programa *SumaArreglo*, es un programa que puede separarse fácilmente en módulos. Para un breve repaso del diseño de programa, vamos a analizar el diagrama de estructura.



Los rectángulos sombreados se refieren a los procedimientos en la biblioteca Irvine. El procedimiento main llama a **PedirEntero**, que a su vez llama a WriteString y **ReadInt**. Por lo general, es más sencillo llevar la cuenta de los diversos archivos en un programa con varios módulos si creamos un directorio separado en el disco para los archivos.

9 Cadenas y Arreglos

Introducción

Si aprendemos a procesar cadenas y arreglos con eficiencia, podremos dominar el área más común de optimización de código. Los estudios han demostrado que la mayoría de los programas invierten 90% de su tiempo ejecutando el 10% de su código. Sin duda, el 10% ocurre con frecuencia en los ciclos, y éstos se requieren al procesar cadenas y arreglos. En este capítulo mostraremos las técnicas para procesar cadenas y arreglos, con el objetivo de escribir código eficiente.

Empezaremos con las instrucciones primitivas de cadena optimizadas de Intel diseñadas para mover, comparar, cargar y almacenar bloques de datos. A continuación presentaremos varios procedimientos para el manejo de cadenas en la biblioteca Irvine32. Sus implementaciones son bastante similares al código que podríamos ver en una implementación de la biblioteca de cadenas estándar de C.

Instrucciones primitivas de cadenas

El conjunto de instrucciones IA-32 tiene cinco grupos de instrucciones para procesar arreglos de bytes, palabras y dobles palabras. Aunque se llaman *primitivas de cadenas*, no se limitan a los arreglos de cadenas. Cada instrucción en la siguiente tabla utiliza en forma implícita a ESI, EDI o ambos registros para direccionar la memoria. Las referencias al acumulador implican el uso de AL, AX o EAX, dependiendo del tamaño de los datos de la instrucción. Las primitivas de cadenas se ejecutan con eficiencia, ya que se repiten e incrementan los índices de los arreglos de manera automática.

Instrucciones primitivas de cadenas.

Instrucción	Descripción
MOVSB, MOVSW, MOVSD	**Mover datos de cadena**: copia los datos de la memoria direccionada por ESI a la memoria direccionada por EDI
CMPSB, CMPSW, COMPSD	**Comparar cadenas:** compara el contenido de dos ubicaciones de memoria direccionadas por ESI y EDI
SCASB, SCASW, SCASD	**Explorar cadena:** compara el acumulador (AL, AX o EAX) con el contenido de la memoria direccionada por EDI
STOSB, STOSW, STOSD	**Almacenar datos de cadena:** almacena el contenido del acumulador en la memoria direccionada por EDI
LODSB, LODSW, LODSD	**Cargar acumulador desde cadena:** carga la memoria direccionada por ESI al acumulador

En los programas en modo protegido, ESI es de manera automática un desplazamiento en el segmento direccionado por DS: y EDI es de manera automática un desplazamiento en el segmento diseccionado por ES, DS y ES siempre se establecen en el mismo valor, y no se pueden cambiar. Por otro lado, en el modo direccionamiento real, los programadores de ASM manipulan con frecuencia a ES y DS.

En el modo direccionamiento real, las primitivas de cadenas usan los registros SI y DI para direccionar la memoria. SI es un desplazamiento desde DS, y DI es un desplazamiento desde ES. Por lo general, ES se establece al mismo valor de segmento que DS, al principio de main:

```
main PROC
   mov ax,@data   ; obtiene direccionamiento del seg de datos
   mov ds,ax           ; inicializa DS
   mov es,ax           ; inicializa ES
```

Uso de un prefijo de repetición Por sí sola, una instrucción de primitiva de cadena sólo procesa un solo valor de memoria o un par de valores. Si agregamos un *prefijo de repetición*, la instrucción se repite usando a ECX como contador. El prefijo de repetición nos permite procesar un arreglo completo mediante una sola instrucción. Se utilizan los siguientes prefijos de repetición:

REP	Repite mientras que ECX > 0
REPZ, REPE	Repite mientras la bandera Cero esté en uno y ECX > 0
REPNZ, REPNE	Repite mientras la bandera Cero esté en cero y ECX > 0

Ejemplo: copiar una cadena El siguiente ejemplo, MOVSB se mueve 10 bytes a partir de **cadena1**, hacia **cadena2**. El prefijo de repetición primero evalúa ECX > 0 antes de ejecutar la instrucción MOVSB.

Si ECX = 0, la instrucción se ignora y el control pasa a la siguiente línea en el programa. Si ECX > 0, ECX se decrementa y la instrucción se repite:

```
cld                          ; borra la bandera Dirección

mov esi,OFFSET cadena1       ; ESI apunta al origen

mov edi,OFFSET cadena2       ; EDI apunta al destino

mov ecx,10                   ; establece el contador a 10

rep movsb                    ; se mueve 10 bytes
```

ESI y EDI se incrementan de manera automática cuando MOVSB se repite. Este comportamiento se controla mediante la bandera Dirección de la CPU

Bandera Dirección La instrucciones primitivas de cadenas incrementan o decrementan a ESI y EDI, según el estado de la bandera Dirección (tabla siguiente). Esta bandera puede modificarse en forma explícita usando las instrucciones CLD y STD:

```
CLD                          ; borra la bandera Dirección (dirección de avance)

STD                          ; activa la bandera de dirección (dirección de
retroceso)
```

Si olvidamos activar la bandera Dirección antes de una instrucción primitiva de cadena, podemos tener grandes problemas. El código resultante se ejecuta de manera inconsistente, según el estado arbitrario de la bandera Dirección.

Uso de la bandera Dirección en instrucciones primitivas de cadena.

Valor de la bandera Dirección	Efecto sobre ESI y EDI	Secuencia de direcciones
Cero	Se incrementa	Bajo-alto
Uno	Se decrementa	Alto-bajo

MOVSB, MOVSW y MOVSD

La instrucción MOVSB, MOVSW y MOVSD copian datos de la ubicación de memoria a la que apunta ESI, hasta la ubicación de memoria a la que apuntan EDI. Los dos registros se incrementan o decrementan en forma automática (según el valor de la bandera Dirección):

MOVSB	Mueve (copia) bytes
MOVSW	Mueve (copia) palabras
MOVSD	Mueve (copia) dobles palabras

Podemos utilizar un prefijo de repetición con MOVSB, MOVSW y MOVSD. La bandera Dirección determina si ESI y EDI se van a incrementar o decrementar. El tamaño del incremento o decremento se muestra en la siguiente tabla:

Instrucción	Valor que se agrega o resta a ESI y EDI
MOVSB	1
MOVSW	2
MOVSD	4

Ejemplo: copiar arreglo de dobles palabras Supongamos que queremos copiar 20 enteros tipo doble palabra, de **origen** a **destino**. Una vez que se copia el arreglo, ESI y EDI apuntan una posición (4 bytes) más lejos del final de cada arreglo:

```
.data

origen DWORD 20 DUP(0FFFFFFFFh)

destino DWORD 20 DUP(?)
```

```
.code
cld                             ; dirección = avance
mov ecx, LENGTHOF origen        ; establece contador REP
mov esi, OFFSET origen          ; ESI apunta al origen
mov edi, OFFSET destino         ; EDI apunta al destino
rep movsd                       ; copia dobles palabras
```

CMPSB, CMPSW y CMPSD

Las instrucciones CMPSB, CMPSW y CMPSD comparan un operando de memoria al que apunta ESI, con un operando de memoria al que apunta EDI:

CMPSB	Compara bytes
CMPSW	Compara palabras
CMPSD	Compara dobles palabras

Podemos usar un prefijo de repetición con CMPSB, CMPSW y CMPSD. La bandera Dirección determina el incremento o decremento de ESI y EDI.

Forma explícita de CMPS: en otra forma de la instrucción de comparación de cadenas llamada forma explícita, se suministran dos operandos indirectos. El operando PTR aclara los tamaños de los operandos. Por ejemplo:

```
cmps DWORD PTR [esi],[edi]
```

Pero CMPS es engañoso, ya que el ensamblador nos permite suministrar operandos erróneos:

```
cmps DWORD PTR [eax],[ebx]
```

Sin importar qué operandos se utilicen, CMPS compara el contenido de la memoria a la que apunta ESI con la memoria que apunta EDI. Observamos que el orden de los operandos CMPS es opuesta a la instrucción CMPS, más conocida:

```
CMP destino, origen
```

```
CMPS origen, destino
```

He aquí otra forma de recordar la diferencia: CMP implica restar el *origen* del *destino*, CMPS implica restar el destino del origen. Se sugiere evitar el uso de CMPS y utilizar las versiones específicas (CMPSB, CMPSW, CMPSD).

Ejemplo: comparación de dobles palabras Supongamos que deseamos comparar un par de dobles palabras mediante el uso de CMPSD. En el siguiente ejemplo, **origen** tiene un valor menor que **destino**. Cuando se ejecuta JA, el salto condicional no se lleva a cabo; en vez de ello se ejecuta la instrucción JMP:

```
.data

origen DWORD 1234h

destino DWORD 5678h

.code

mov esi OFFSET origen

mov edi OFFSET destino

cmpsd                          ; compara dobles palabras

ja L1                          ; salta si origen > destino

jmp L2                         ; salta, ya que origen <= destino
```

Para comparar varias dobles palabras, debemos borrar la bandera Dirección, inicializamos ECX como contador y utilizamos un prefijo repetido con CMPSD:

```
mov esi, OFFSET origen

mov edi, OFFSET destino

cld

mov ecx,LENGTHOF origen        ; contador de repetición

repe cmpsd                     ; repite mientras sea igual
```

El prefijo REPE repite la comparación e incrementa a ESI y EDI de manera automática hasta que ECX sea igual a cero, o hasta que un par de dobles palabras sea distinto.

Ejemplo: comparación de dos cadenas

Por lo general, las cadenas se comparan relacionando caracteres individuales en secuencia, empezando al principio de las cadenas. Por ejemplo, los primeros tres caracteres de “AABC” y “AABB” son idénticos. En la cuarta posición, el código ASCII para “C” (en la primera cadena) es mayor que el código ASCII para “B” (en la segunda cadena). Por ende, la primera cadena se considera mayor que la segunda. De manera similar, si las cadenas “AAB” y “AABB” se comparan, la segunda cadena tiene un valor más grande. Los primeros caracteres son idénticos, pero existe un carácter adicional en la segunda cadena.

El siguiente programa utiliza COMPSB para comparar dos cadenas de igual longitud. El prefijo REPE hace que CMPSB, continúe incrementando a ESI y EDI, y que compare los caracteres uno a uno hasta encontrar una diferencia entre las dos cadenas:

```
TITLE Comparación de cadenas (Cmpsb.asm)

; Este programa utiliza a CMPSB para comprobar dos cadenas

; de la misma longitud.

INCLUDE Irvine32.inc

.data

origen BYTE “MARTIN  “

dest BYTE “MARTINEZ”

cad1 BYTE “La cadena de origen es más chica”,0dh,0ah,0

cad2 BYTE “La cadena de origen no es más chica”,0dh,0ah,0

.code

main PROC

   cld                          ; dirección = avance

   mov esi,OFFSET origen

   mov edi,OFFSET dest

   mov ecx,LENGTHOF origen

   repe cmpsb

   jb origen_mas_chico

   mov edx,OFFSET cad2

   jmp listo

origen_mas_chico:

   mov edx,OFFSET cad1

listo:

   call WriteString

   exit
```

```
main ENDP

END main
```

Al utilizar los datos de prueba proporcionados, aparece el mensaje “La cadena de origen es mas chica”. ESI y EDI quedan apuntando a una posición más lejos del punto en el que se encontró que las dos cadenas diferían. Si las cadenas hubieran sido idénticas, ESI y EDI hubieran apuntado una posición más lejos del final de sus respectivas cadenas.

La comparación de dos cadenas con CMPSB sólo funciona cuando son de la misma longitud. Esto explica por qué era necesario en el ejemplo anterior rellenar “MARTIN” con espacios a la derecha, para que fuera de la misma longitud que “MARTINEZ”. El proceso de rellenar cadenas con espacios impone una restricción para el manejo de las cadenas.

SCASB, SCASW, SCASD

Las instrucciones SCASB, SCASW y SCASD comparan un valor en AL/AX/EAX con un byte, palabra o doble palabra, respectivamente, la cual está direccionada por EDI. Las instrucciones son útiles cuando se busca un valor individual en una cadena o arreglo. Si se combinan con el prefijo REPE (o REPZ), la cadena o arreglo se explora mientras ECX > 0, y el valor en AL/AX/EAX coincida con cada valor subsiguiente en memoria. El prefijo REPNE explora hasta que AL/AX/EAX coincida con un valor en memoria, o cuando ECX = 0.

Explorar en busca de un carácter que coincida En el siguiente ejemplo buscamos la letra F en la cadena **alfa**. Si se encuentra la letra, EDI apunta una posición más allá del carácter que coincidió. Si no se encuentra la letra JNZ termina el programa:

```
.data

alfa BYTE “ABCDEFGH”,0

.code

mov edi, OFFSET alfa          ; EDI apunta a la cadena

mov al,’F’                    ; busca la letra F

mov ecx, LENGTHOF alfa        ; establece la cuenta de búsqueda

cld                           ; dirección = avance

repne scasb                   ; repite mientras no sea igual

jnz salir                     ; termina si no se encontró la letra

dec edi                       ; se encontró: retrocede EDI
```

JNZ se agregó después del ciclo para evaluar la posibilidad de que el ciclo se detuviera debido a ECX = 0, y que no se encontrara el carácter en AL.

STOSB, STOSW y STOSD

Las instrucciones STOSB, STOSW y STOSD almacenan el contenido de AL/AX/EAX, respectivamente, en el desplazamiento al que apunta EDI. EDI se incrementa o decrementa con base en el estado de la bandera Dirección. Cuando se utiliza con el prefijo REP, estas instrucciones son útiles para rellenar todos los elementos de una cadena o arreglo con un solo valor. Por ejemplo, el siguiente código inicializa cada byte en **cadena1** con 0FFh:

```
.data

Cuenta = 100

cadena1 BYTE Cuenta DUP(?)

.code

mov al, 0FFh                    ; valor a guardar

mov edi,OFFSET cadena1          ; EDI apunta al destino

mov ecx,Cuenta                  cuenta de caracteres

cld                             ; dirección = avance

rep stosb                       ; llena con el contenido de AL
```

LODSB, LODSW y LODSD

Las instrucciones LODSB, LODSW y LODSD cargan un byte o palabra de memoria en ESI, hacia AL/AX/EAX, respectivamente. ESI se incrementa o decrementa según el estado de la bandera Dirección. El prefijo REP se utiliza raras veces con LODS, ya que cada nuevo valor que se carga en el acumulador sobrescribe su contenido anterior. En vez de ello, LODS se utiliza para cargar un solo valor. En el siguiente ejemplo, LODSB sustituye a las dos instrucciones siguientes (suponiendo que la bandera Dirección está en cero):

```
mov al,[esi]                    ; mueve byte hacia AL

inc esi                         ; apunta al siguiente byte
```

Ejemplo de multiplicación de arreglos El siguiente programa multiplica cada elemento de un arreglo de dobles palabras por un valor constante. LODSD y STOSD trabajan en conjunto:

```
TITLE Multiplicación de un arreglo (Mult.asm)

; este programa multiplica cada elemento de un arreglo

; de enteros de 32 bits por un valor constante.

INCLUDE Irvine32.inc
```

```
.data
arreglo DWORD 1,2,3,4,5,6,7,8,9,10 ; datos de prueba
multiplicador DWORD 10         ; datos de prueba
.code
main PROC
   cld
   mov esi,OFFSET arreglo      ; índice de origen
   mov edi,esi                 ; índice de destino
   mov ecx, LENGTHOF arreglo   ; contador de ciclo
L1: lodsd                      ; copia [ESI] hacia EAX
   mul multiplicador           ; multiplica por un valor
   stosd                       ; guarda EAX en [EDI]
   loop L1
   exit
main ENDP
END main
```

Arreglos bidimensionales

Ordenamiento de fila y columnas

Desde la perspectiva de un programador de lenguaje ensamblador, un arreglo bidimensional es una abstracción de alto nivel de un arreglo unidimensional. Los lenguajes de alto nivel seleccionan uno de dos métodos para ordenar las filas y columnas en memoria: *orden por filas (row-major)* y *orden por columnas (column-major)*. Cuando se utiliza el orden por filas (el más común), la primera fila aparece al principio del bloque de memoria. El último elemento en la primera fila va seguido en la memoria por el primer elemento de la segunda fila. Cuando se utiliza el orden por columnas, los elementos en la primera columna aparecen al principio del bloque de memoria. El último elemento en la primera columna va seguido en la memoria por el primer elemento de la segunda columna.

Orden por filas y por columnas

Arreglo lógico:

10	20	30	40	50
60	70	80	90	A0
B0	C0	D0	E0	F0

Orden por filas

10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0

Orden por columnas

10	60	B0	20	70	C0	30	80	D0	40	90	E0	50	A0	F0

Si se implementa un arreglo bidimensional en el lenguaje ensamblador, podemos elegir cualquiera de los dos métodos de ordenamiento.

El conjunto de instrucciones IA-32 incluye dos tipos de operandos (base-índice y base-índice-desplazamiento), ambos adecuados para las aplicaciones con arreglos. Examinaremos amos y mostraremos ejemplos de cómo podemos utilizarlos de forma efectiva.

Operandos base-índice

Un operando base-índice suma los valores de dos registros (llamados base e índice), produciendo una dirección de desplazamiento:

```
[base + índice]
```

Los corchetes son necesarios. En modo de 32 bits, cualquier registro de propósito general de 32 bits puede ser registro base o índice. En modo de 16 bits, el registro debe ser BX o BP. (Se recomienda evitar usar BP o EBP, excepto al direccionar la pila). El registro índice de ser SI o DI. He aquí un ejemplo de varias combinaciones en modo de 32 bits:

```
.data

arreglo WORD 1000h,2000h,3000h

.code

mov ebx,OFFSET arreglo

mov esi,2

mov ax,[ebx+esi]                ; AX = 2000h
```

```
mov edi,OFFSET arreglo

mov ecx,4

mov ax,[edi+ecx]               ; AX = 3000h


mov ebp,OFFSET arreglo

mov esi,0

mov eax,[ebp+esi]              ; AX = 1000h
```

Arreglo bidimensional Al acceder a un arreglo bidimensional en orden por filas, el desplazamiento de fila se guarda en el registro base y el desplazamiento de columnas está en el registro índice. Por ejemplo, la siguiente tabla tiene tres filas y cinco columnas:

```
tablaB BYTE   10h,  20h,  30h,  40h, 50h

TamFila = ($ - tablaB)

       BYTE   60h,  70h,  80h,  90h, 0Ah

       BYTE   0B0h, 0C0h, 0D0h, 0E0  0F0h
```

La tabla está en orden por filas y el valor TamFila constante lo calcula el ensamblador como el número de bytes en cada fila de la tabla. Supongamos que deseamos localizar una entrada específica en la tabla, usando coordenadas de fila y de columna. Suponiendo que las coordenadas tienen base 0, la entrada en la fila 1, columna 2 contiene 80h. Establecemos EBX al desplazamiento de la tabla, sumamos (TamFila * índice_fila) para calcular el desplazamiento de la fila y establecemos ESI al índice de columna:

```
índice_fila = 1

índice_columna = 2

mov ebx,OFFSET tablaB          ; desplazamiento de la tabla

add ebx,TamFila * índice_fila  ; desplazamiento de la fila

mov esi,índice_columna

mov al,[ebx + esi]             ; AL = 80h
```

Cálculo de la suma de una fila

El direccionamiento base índice simplifica muchas veces tareas asociadas con los arreglos bidimensionales. Por ejemplo, podríamos sumar los elementos en una fila que pertenezcan a

una matriz de enteros. El siguiente procedimiento llama calc_suma_fila, calcula la suma de una fila seleccionada, en una matriz de enteros de 8 bits:

```
calc_suma_fila PROC uses ebx ecx edx esi

; Calcula la suma de una fila en una matriz de bytes.

; Recibe: EBX = desplazamiento de la tabla, EAX = índice de fila,

;                              ECX = tamaño de la fila en bytes.

; Devuelve: EAX guarda la suma.

;-------------------------------------------------

   mul ecx                     : índice de fila * tamaño de fila

   add ebx,eax                 ; desplazamiento de fila

   mov eax,0                   ; acumulador

   mov esi,0                   ; índice de columna

L1: movzx edx, BYTE PTR[ebx + esi] ; obtiene un byte

   add eax,edx                 ; lo suma al acumulador

   inc esi                     ; siguiente byte de la fila

   loop L1

ret

calc_suma_fila ENDP
```

Se requirió el uso de BYTE PTR para aclarar el tamaño del operando de la instrucción MOVZX.

Factores de escala

Si se escribe código para un arreglo de valores tipo WORD, debemos multiplicar el operando por un factor de escala de 2. El siguiente ejemplo localiza el valor en la fila 1, columna2.

```
tablaW BYTE   10h,  20h,  30h,  40h, 50h

TamFilaW = ($ - tablaB)

       BYTE   60h,  70h,  80h,  90h, 0Ah

       BYTE   0B0h, 0C0h, 0D0h, 0E0  0F0h

índice_fila = 1
```

```
índice_columna = 2

mov ebx,OFFSET tablaW              ; desplazamiento de la tabla

add ebx,TamFilaW * índice_fila     ; desplazamiento de la fila

mov esi,índice_columna

mov ax,[ebx + esi * TYPE tablaW]   ; AX = 80h
```

El factor de escala que se utiliza en este ejemplo (TYPE tablaW) es igual a 2, de manera similar, hay que utilizar un factor de escala de 4 si el arreglo contiene dobles palabras.

Operandos base-índice-desplazamiento

Un operando base-índice-desplazamiento combina un desplazamiento, un registro base, un registro índice y un factor de escala opcional para producir una dirección efectiva. He aquí los formatos:

```
[base + índice + desplazamiento]

desplazamiento + índice]
```

Desplazamiento puede ser el nombre de la variable o expresión constante. En modo de 32 bits, puede utilizarse cualquier registro de propósito general de 32 bits para la base y el índice. En modo de 16 bits, el operando base debe ser BX o BP y el operando índice debe ser SI o DI. Los operandos base-índice-desplazamiento se adaptan muy bien al procedimiento de arreglos bidimensionales. El desplazamiento puede ser el nombre de un arreglo, el operando base puede guardar el desplazamiento de la fila y el operando índice puede guardar el desplazamiento de la columna.

Ejemplo con arreglo de dobles palabras El siguiente arreglo bidimensional guarda tres filas de cinco dobles palabras:

```
tablaD DWORD   10h,  20h,  30h,  40h, 50h

TamFilaW = ($ - tablaB)

       DWORD  60h,  70h,  80h,  90h, 0Ah

       DWORD 0B0h, 0C0h, 0D0h, 0E0  0F0h
```

TamFila es igual a 20 (14h). Suponiendo que las coordenadas estén con base cero, la entrada en la fila 1, columna 2 contiene 80h. Para acceder a esta entrada, establecemos EBX al índice de fila y ESI al índice de columna:

```
   mov ebx,TamFila             ; índice de fila

   mov esi,2                   ; índice de columna
```

```
mov eax,tablaD[ebx 6 esi * TYPE tablaD]
```

Búsqueda y ordenamiento de arreglos de enteros

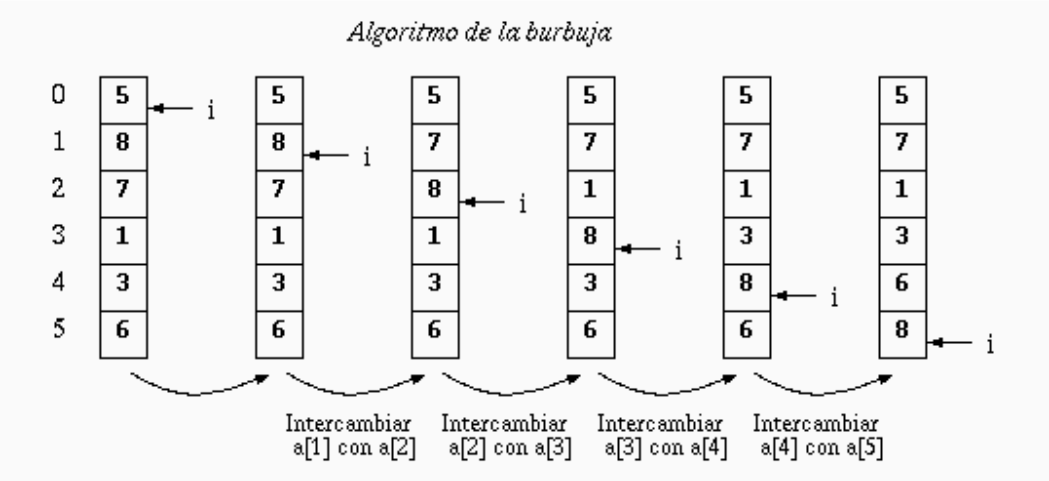
Los científicos de la computación han invertido una gran cantidad de tiempo y energía para encontrar mejores formas de buscar y ordenar conjuntos de datos masivos. Se ha demostrado que es más útil elegir el mejor algoritmo para una aplicación específica que comprar una computadora más rápida. La mayoría de los estudiantes aprenden los métodos de búsqueda y ordenamiento usando el lenguaje de alto nivel, como C++ y Java. El lenguaje ensamblador presenta una perspectiva diferente para el estudio de los algoritmos, ya que nos permite ver los detalles de implementación de bajo nivel. Es interesante observar que uno de los autores de algoritmos más notables del siglo veinte, Donald Knuth, utilizó lenguaje ensamblador para sus ejemplos de programas publicados.

La búsqueda y ordenamiento nos dan la oportunidad de probar los modos de direccionamiento. En especial, el direccionamiento indexado por la base resulta ser útil, ya que podemos apuntar un registro (como EBX) a la base de un arreglo y utilizar otro registro (como ESI) para indexar hacia cualquier otra ubicación del arreglo.

Ordenamiento de burbuja

El ordenamiento de burbuja compara pares de valores de arreglos empezando en las posiciones 0 y 1. Si los valores comparados están en orden inverso, se intercambian. La siguiente figura muestra el progreso de una ejecución (pasada) a través de un arreglo de enteros.

Primera ejecución a través de un arreglo (Ordenamiento de burbuja).



Después de una ejecución, el arreglo aún no está ordenado, pero el valor más grande se encuentra ahora en la posición del índice más alto. El ciclo externo empieza otra ejecución a través del arreglo. Después de *n – 1* ejecuciones, se garantiza que el arreglo quedará ordenado.

El ordenamiento de burbuja funciona bien con arreglos pequeños, pero se vuelve demasiado ineficiente para los arreglos más grandes. Es un algoritmo O(n^2), lo que significa que el tiempo de ordenamiento se incrementa en forma cuadrática, en relación con el número de elementos del arreglo (n). Supongamos, por ejemplo, que se requieren 0.1 segundos para ordenar 1000 elementos. A medida que el número de elementos se incrementa por un factor de 10, el tiempo requerido para ordenar se incrementa por un factor de 10^2 (100). La siguiente tabla muestra los tiempos de ordenamiento para varios tamaños de arreglos, suponiendo que pueden ordenarse 1000 elementos en 0.1 segundos:

Tamaño del arreglo	Tiempo (en segundos)
1.000	0.1
10.000	10.0
100.000	1000
1.000.000	100.000 (27,78 horas)

El ordenamiento de burbuja no será un buen método de ordenamiento para un arreglo de 1 millón de enteros, ya que ¡se requerirían más de 27 horas para terminar! Pero está bien para unos cuantos cientos de enteros.

Seudocódigo Es útil crear una versión simplificada del ordenamiento de burbuja, usando seudocódigo que sea similar al lenguaje ensamblador. Utilizando **N** para representar el tamaño del arreglo, **cx1** para representar el contador de ciclo externo y **cx2** para representar el contador de ciclo interno:

```
cx1 = N -1

while (cx1 > 0)

{

 esi = addr(arreglo)

 cx2 = cx1

while ( cx2 > 0)

 {

  if(arreglo[esi] < arreglo[esi+4]

   intercambiar( arreglo[esi], arreglo[esi+4] )
```

```
 add esi,4
 dec cx2
 }
dec cx1
}
```

Lenguaje ensamblador Del seudocódigo podemos generar con facilidad una implementación que coincida en lenguaje ensamblador, colocándola en un procedimiento con parámetros y variables locales:

```
;-----------------------------------------------------
 OrdenBurbuja PROC USES eax ecx esi,
  pArreglo: PTR DWORD          ; apuntador a un arreglo
  Cuenta: DWORD                ; tamaño del arreglo
;-----------------------------------------------------
 mov ecx, Cuenta
 dec ecx                       ; decrementa la cuenta en 1
L1: push ecx                   ; guarda cuenta del ciclo externo
 mov esi, pArreglo             ; apunta al primer valor
L2: mov eax,[esi]              ; obtiene el valor del arreglo
 cmp [esi+4],eax               ; compara un par de valores
 jge L3
 xchg eax,[esi+4]              ; intercambia el par
 mov [esi],eax
L3: add esi,4                  ; mueve ambos apuntadores hacia adelante
 loop L2                       ; ciclo interno
 pop ecx                       ; obtiene cuenta del ciclo externo
 loop L1                       ; en cualquier otro caso, repite el ciclo
externo
```

```
L4: ret

OrdenBurbuja ENDP
```

Búsqueda binaria

Las búsquedas en los arreglos son alguna de las operaciones más comunes en la programación ordinaria. Para un arreglo pequeño (1000 elementos o menos), es más fácil realizar una *búsqueda secuencial*, en la que se empieza al principio del arreglo y se examina cada elemento en secuencia, hasta encontrar uno que coincida. Para un arreglo de *n* elementos, una búsqueda secuencial requiere un promedio de *n/2* comparaciones. Si se busca en un arreglo pequeño, el tiempo de ejecución es mínimo. Por otro lado, para buscar en un arreglo de 1 millón de elementos se requiere una cantidad más considerable de tiempo de procesamiento.

El algoritmo de *búsqueda binaria* es en especial eficiente cuando se busca un elemento individual en un arreglo grande. Tiene una condición previa importante: los elementos de arreglo deben ordenarse en forma ascendente o descendente. He aquí una descripción informal del algoritmo:

Antes de comenzar la búsqueda, pedir al usuario que introduzca un entero, al cual llamaremos *valBusqueda*

1. El rango del arreglo en el que se va a buscar se indica mediante los subíndices llamados *primero* y *último*. Si *primero* > *último*, terminar la búsqueda, indicando que no se pudo encontrar una coincidencia.
2. Calcular el punto medio del arreglo, entre subíndices *primero* y *último*.
3. Comparar *valBusqueda* con el entero que está en el punto medio del arreglo:
    - Si los valores son iguales, regresar del procedimiento con el punto medio EAX. Este valor de retorno indica que encontró una coincidencia en el arreglo.
    - Por otro lado, si *valBusqueda* es mayor que el número en el punto medio, restablecer *primero* a una posición más alta que el punto medio.
    - O, si valBusqueda es menor que el punto medio, restablece *último* a una posición más baja que el punto medio.
4. Regresar paso1.

La búsqueda binaria es extraordinariamente eficiente, ya que utiliza una estrategia llamada *dividir y vencer*. El rango de valores se divide a la mitad con cada iteración del ciclo. En general, se describe como un algoritmo O(log n), lo que significa que, a medida que se incrementa el número de elementos del rango por un factor *n*, el tiempo de búsqueda promedio se incrementa en tan solo por un factor de log *n*. Como el número real de comparaciones puede variar, la siguiente tabla registra el número máximo de comparaciones requeridas para varios tamaños de arreglos:

Tamaño del arreglo (n)	Comparaciones máximas
64	6

1,024	10
65,536	17
1,048,576	21
4,294,967,296	33

El número máximo de comparaciones se calcula como log2(n+1), redondeando al siguiente entero más grande.

10 Estructuras y macros

Estructuras

Una *estructura* es una plantilla o patrón que se proporciona a un grupo de variables relacionadas en forma lógica. A las variables en una estructura se les llama *campos*. Las instrucciones de un programa pueden acceder a la estructura como una sola entidad, o pueden acceder a los campos individuales. Con frecuencia, las estructuras contienen campos de tipos distintos. Una unión también agrupa a varios identificadores, pero éstos traslapan la misma área en memoria.

La estructura proporciona una forma fácil de agrupar datos y pasarlos de un procedimiento a otro. Supongamos que los parámetros de entrada para un procedimiento consisten en 20 unidades distintas de datos relacionados con una unidad del disco. No sería práctico llamar al procedimiento y pasar los argumentos requeridos en forma correcta. En vez de ello, podríamos colocar todos los datos de entrada en una estructura y pasar la dirección de esa estructura al procedimiento. se utilizaría un mínimo espacio en la pila (una dirección) y el procedimiento llamado podría modificar el contenido de la estructura.

En el lenguaje ensamblador, las estructuras son en esencia las mismas que en C y C++. Con un pequeño esfuerzo de traducción, podemos tomar cualquier estructura de la biblioteca API de MS Windows y hacerla que funcione en lenguaje ensamblador. la mayoría de los depuradores pueden mostrar los campos individuales de las estructuras.

Estructura COORD la estructura COORD que se define en la API de Windows identifica a las coordenadas de pantalla X y Y. El campo X tiene un desplazamiento de cero, relativo al principio de la estructura y el desplazamiento del campo Y es 2;

```
COORD STRCUT

 X WORD ?                    ; desplazamiento 01

 Y WORD ?                    ; desplazamiento 02
```

```
COORD ENDS
```

Para usar una estructura se requieren tres pasos secuenciales:

1. Definir la estructura.
2. Declarar una o más variables del tipo de la estructura, a las cuales se les llama *variables de estructura*.
3. Escribir instrucciones en tiempo de ejecución para acceder a los campos de la estructura.

Definición de estructuras

Una estructura se define mediante el uso de las directivas STRUCT y ENDS. Dentro de la estructura, se definen campos usando la misma sintaxis para las variables ordinarias. Las estructuras pueden contener casi cualquier número de campos:

```
nombre STRUCT

   declaración de campos

nombre ENDS
```

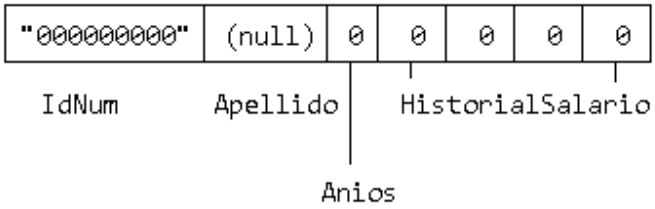
Inicializadores de campo Cuando los campos de una estructura tienen inicializadores, los valores se asignan cuando se las crean variables de la estructura. Podemos utilizar varios tipos de inicializadores de campos:

- **Indefinido**: el operador ? deja el contenido del campo indefinido.
- **Literales de cadena**: las cadenas de caracteres se encierran entre comillas.
- **Enteros**: las constantes y expresiones enteras.
- **Arreglos**: el operador DUP puede inicializar elementos de arreglos.

La siguiente estructura llamada **Empleado** describe la información de un empleado, con campos como número de identificación (ID), apellido, años de servicio y un arreglo de valores del historial de su salario. La siguiente definición debe aparecer antes de la declaración de variables **Empleado**:

```
Empleado STRUCT

 NumID BYTE “000000000”

 Apellido BYTE 30 DUP(0)

 Anios WORD 0

 HistorialSalario DWORD 0,0,0,0

Empleado ENDS
```

Ésta es una representación lineal de la distribución de memoria de la estructura:



Alineación de los campos de una estructura

Para el mejor rendimiento de E/S de memoria, los miembros de una estructura deben alinearse a direcciones que coincidan con sus tipos de datos. De no ser así, la CPU requerirá más tiempo para acceder a los miembros. Por ejemplo, un miembro tipo doble palabra debe alinearse en un límite de doble palabra. La siguiente tala presenta las alineaciones que usan los compiladores C y C++ de Microsoft, y las funciones de la API Win32. En lenguaje ensamblador, la directiva ALIGN establece la alineación de la dirección del siguiente campo o variable:

```
ALIGN tipodedatos
```

El siguiente ejemplo, alinea **miVar** a un límite de doble palabra:

```
.data

ALIGN DWORD

miVar DWORD ?
```

Vamos a definir de manera correcta la estructura Empleada, usando ALIGN para colocar **Anios** en un límite WORD y a **HistorialSalario** en límite DWORD. Los tamaños de los campos aparecen como comentarios.

```
Empleado STRUCT

numID BYTE “000000000”          ; 9

Apellido BYTE 30 DUP(0)         ; 30

ALIGN WORD                      ; se agregó 1 byte

Anios WORD 0                    ; 2

ALIGN DWORD                     ; se agregó 2 bytes

HistorialSalario DWORD 0,0,0,0 ; 16

Empleado ENDS                   ; 60 en total
```

Alineación de los miembros de una estructura.

Tipo de miembro	Alineación
BYTE, SBYTE	Se alinea en un límite de 8 bits (byte)
WORD,SWORD	Se alinea en un límite de 16 bits (palabra)
DWORD, SDWORD	Se alinea en un límite de 32 bits (doble palabra)
QWORD	Se alinea en un límite de 64 bits (palabra cuádruple)
REAL4	Se alinea en un límite de 32 bits (doble palabra)
REAL8	Se alinea en un límite de 64 bits (palabra cuádruple)
Estructura	El requerimiento de alineación más grande de cualquier miembro
Unión	El requerimiento de alineación del primer miembro

Declaración de variables de estructura

Las variables de una estructura pueden declararse e inicializarse de manera opcional con valores específicos. Ésta es la sintaxis, en la que ya se ha definido *tipoEstructura* mediante la directiva STRUCT:

```
identificador tipoEstructura < lista-inicializadores >
```

El *identificador* sigue las mismas reglas que los demás nombres de variables en MASM. La *lista-inicializadores* es opcional, pero si se utiliza, es una lista separada por comas de constantes en tiempo de ensamblado que coinciden con los tipos de datos de los campos específicos de una estructura:

```
inicializador [, inicializador] …
```

Los signos de mayor y menor que (<>) vacíos hacen que la estructura contenga los valores predeterminados de los campos, provenientes de la definición de la estructura. De manera alternativa, podemos insertar nuevos valores en los campos seleccionados. Los valores se insertan en los campos de la estructura, en orden de izquierda a derecha, coincidiendo con el orden de los campos en la declaración de la estructura. A continuación se muestran ejemplos de ambos métodos, usando las estructuras **COORD** y **Empleado**:

```
.data

punto1 COORD <5,10>            ; X = 5, Y = 10

punto2 COORD <20>              ; X = 20, Y = ?

punto2 COORD <>                ; X = ?, y = ?

trabajador Empleado <>         ; (inicializadores predeterminados)
```

Es posible redefinir sólo los inicializadores de los campos, predeterminados. La siguiente declaración redefine sólo el campo **NumID** de la estructura **Empleado**, asignando los valores predeterminados a los campos restantes:

```
persona1 Empleado <”555223333”>
```

Una forma de notación alternativa utiliza llaves {...} en vez de los signos <>:

```
persona2 Empleado {“555223333”}
```

Cuando el inicializador para un campo de cadena es más corto que el campo, el resto de las posiciones se rellenan con espacios. No se inserta de manera automática un byte nulo al final de un campo de cadena. Podemos omitir campos de la estructura insertando comas como marcadores de posición. Por ejemplo, la siguiente instrucción omite el campo **NumID** e inicializa el campo **Apellido**:

```
Persona3 Empleado <,”dJones”>
```

Para un campo de arreglo, debemos utilizar el operador DUP para inicializar algunos o todos los elementos del arreglo. Si el inicializador es más corto que el campo, el resto de las posiciones se rellena con ceros. En el siguiente ejemplo, inicializamos los primeros dos valores de **HistorialSalario** y establecemos el resto a cero:

```
Persona4 Empleado <,,,2 DUP(20000)>
```

Arreglo de estructuras Usamos el operador DUP para crear un arreglo de estructuras. En el siguiente ejemplo, los campos X y Y de cada elemento en **TodosLosPuntos** se inicializan con ceros:

```
NumPuntos = 3

TodosLosPuntos COORD NumPuntos DUP (<0,0>)
```

Alineación de variables de estructura

Para un mejor rendimiento del procesador, se deben alinear las variables de estructura en un límite de memoria iguales al miembro más grande de la estructura. La estructura Empleado contiene campos DWORD, por lo que la siguiente definición utiliza esa alineación:

```
.data

ALIGN DWORD

persona Empleado <>
```

Referencia a variables de estructura

Las referencias a las variables de estructura y los nombres de estructura pueden hacerse utilizando operadores TYPE y SIZEOF. Por ejemplo, vamos a regresar a la estructura **Empleado** que vimos anteriormente:

```
Empleado STRUCT
   NumId BYTE “000000000”      ; 9
   Apellido BYTE 30 DUP(0)     ; 30
   ALIGN WORD                  ; se agregó 1 byte
   Anios WORD 0                ; 2
   ALIGN DWORD                 ; se agregaron 2 bytes
   HistorialSalario DWORD      ; 16
   Empleados ENDS              ; 60 en total
```

Dada la siguiente definición de datos:

```
.data
Trabajador Empleado <>
```

Cada una de las siguientes expresiones devuelve el mismo valor:

```
TYPE Empleado                  ; 60
SIZEOF Empleado                ; 60
SIZEOF trabajador              ; 60
```

Referencia a los miembros

Las referencias a los miembros de estructura con nombre requieren una variable de estructura como calificador. Las siguientes expresiones constantes pueden generarse en tiempo de ensamblado, usando la estructura **Empleado**:

```
TYPE Empleado.HistorialSalario     ; 4
LENGTHOF Empleado.HistorialSalario ; 4
SIZEOF Empleado.HistorialSalario   ; 16
TYPE Empleado.Anios                ; 2
```

Las siguientes son referencias en tiempo de ejecución a **trabajador**, un Empleado:

```
.data
trabajador Empleado <>
.code
```

```
mov dx,trabajador.Anios

mov trabajador.HistorialSalario,20000    ; primer salario

mov [trabajador.HistorialSalario+4],3000 ; segundo salario
```

Uso del operador OFFSET Podemos utilizar el operador OFFSET para obtener la dirección de un campo dentro de una variable de estructura:

```
mov edx,OFFSET trabajador.Apellido
```

Operandos indirectos e indexados

Los operandos indirectos permiten el uso de un registro (como ESI) para direccionar los miembros de la estructura. El direccionamiento indirecto proporciona flexibilidad, en especial al pasar la dirección de una estructura a un procedimiento, o al utilizar un arreglo de estructuras. Cuando se hace referencia a los operandos indirectos se requiere el operador PTR:

```
mov esi, OFFSET trabajador

mov ax,(Empleado PTR [esi]).Anios
```

La siguiente instrucción no se ensambla, ya que **Anios** por sí sola no identifica la estructura a la que pertenece:

```
mov ax,[esi]. Anios            ; inválida
```

Operandos indexados Podemos utilizar operandos indexados para acceder a los arreglos de estructuras. Supongamos que **departamento** es un arreglo de cinco objetos Empleado. Las siguientes instrucciones acceden al campo **Anio** del empleado en la posición de índice 1:

```
.data

departamento Empleado 5 DUP(<>)

.code

mov esi, TYPE Empleado         ; índice = 1

mov departamento[esi].Anios, 4
```

Iteración a través de un arreglo Puede utilizarse un ciclo con el direccionamiento indirecto o directo para manipular un arreglo de estructuras. El siguiente programa (TodosLosPuntos.asm) asigna coordenadas al arreglo **TodosLosPuntos**:

```
TITLE Iterar a través de un arreglo (TodosLosPuntos.asm)

INCLUDE Irvine32.inc
```

```
NumPuntos = 3
.data
ALIGN WORD
TodosLosPuntos COORD NumPuntos DUP(<0,0>)
.code
main PROC
      mov edi,0                ; índice del arreglo
      mov ecx,NumPuntos        ; contador de ciclo
      mov ax,1                 ; valores X, Y iniciales
L1: mov (COORD PTR TodosLosPuntos[edi]).X,ax
      mov (COORD PTR TodosLosPuntos[edi]).Y,ax
      add edi,TYPE COORD
      inc ax
      loop L1
     exit
main ENDP
END main
```

Ejemplo: mostrar la hora del sistema

MS Windows cuenta con funciones de consola que establecen la posición del cursor en la pantalla y obtienen la hora del sistema. Para usar estas funciones, hay que crear instancias de dos estructuras predefinidas:

```
COORDS y SYSTEMTIME:
COORD STRUCT
  X WORD ?
  Y WORD ?
COORDS ENDS
SYSTEMTIME STRUCT
```

```
  wAnio WORD ?
  wMes WORD ?
  wDiaDeLaSemana WORD ?
  wDia WORD ?
  wHora WORD ?
  wMinuto WORD ?
  wSegundo WORD ?
  wMilisengundos WORD ?
SYSTEMTIME ENDS
```

Para obtener la hora del sistema (ajustada para la zona horaria local), debemos llamar a la función **GetLocalTime** de MS Windows y pasarle la dirección de estructura SYSTEMTIME:

```
.data
horaSys SYSTEMTIME <>
.code
INVOKE GetLocalTime, ADDR horaSys
```

Ahora obtenemos los valores apropiados de la estructura SYSTEMTIME:

```
movzx eax,horaSys.wAnio
call WriteDec
```

Cuando un programa Win32 produce resultados en pantalla, llama a la función **GetStdHandle** de MS Windows para obtener el manejador de salida de consola estándar (un entero):

```
.data
manejadorConsola DWORD ?
.code
INVOKE GetStdHandle, STD_OUTPUT_HANDLE
mov manejadorConsola,eax
```

Para establecer la posición del cursor, debemos llamar la función **SetConsoleCursorPosition** de MS Windows, pasándole el manejador de salida de consola y una variable de estructura COORD que contenga caracteres de las coordenadas X,Y:

```
.data
posXY COORD <10,5>
.code
INVOKE  SetConsoleCursorPosition, manejadorConsola, posXY
```

Listado del programa El siguiente programa (*MostrarHora.asm*) obtiene la hora del sistema y los muestra en una ubicación de pantalla seleccionada. Sólo se ejecuta en modo protegido:

```
TITLE Mostrar la hora          (MostrarHora.ASM)
INCLUDE Irvine32.inc
.data
horaSys SYSTEMTIME <>
posXY COORD <10,5>
manejadorConsola DWORD ?
cadDosPuntos BYTE ":",0

.code
main PROC
; Obtiene el manejador de salida estándar para la consola Win32.
   INVOKE GetStdHandle, STD_OUTPUT_HANDLE
   mov manejadorConsola,eax
; Establece la posición del cursor y obtiene la zona horaria local.
   INVOKE SetConsoleCursorPosition, manejadorConsola, posXY
   INVOKE GetLocalTime, ADDR horaSys
; Muestra la hora del sistema (hh:mm:ss).
   movzx eax,horaSys.wHour     ; horas
   call WriteDec
   movzx eax,horaSys.wMinuto   ; minutos
   call WriteDec
```

```
   call WriteString

   movzx eax,horaSys.wSecond   ; segundos

   call WriteDec

   call Crlf

   call WaitMsg

   exit

main ENDP

END main
```

Este programa utiliza definiciones de SmallWin.inc (Irvine32.inc las incluye de manera automática)

Estructuras que contienen estructuras

Las estructuras pueden contener instancias de otras estructuras. Por ejemplo, un rectángulo puede definirse en términos de sus esquinas superior izquierda e inferior derecha, ambas estructuras COORD:

```
Rectangulo STRUCT

  SupIzq COORD <>

  InfDer COORD <>

Rectangulo ENDS
```

Las variables de rectángulo pueden declararse sin redefiniciones, o redefiniendo campos individuales de COORD. A continuación se muestran formas alternativas:

```
rect1 Rectangulo <>

rect2 Rectangulo { }

rect3 Rectangulo { {10,10}, {50, 20} }

rect4 Rectangulo < <10,10>, <50,20> >
```

A continuación se muestra una referencia directa a un campo de la estructura:

```
mov rect1.SupIzq.x, 20
```

Podemos acceder al campo de una estructura mediante un operando indirecto. El siguiente ejemplo mueve 10 a la coordenada Y de la esquina superior izquierda de la estructura a la que apunta ESI:

```
mov esi, OFFSET rect1

mov (Rectangulo PTR [esi].SupIzq).Y, 10
```

El operador OFFSET puede regresar apuntadores a campos individuales de una estructura, incluyendo los campos anidados:

```
mov edi,OFFSET rect2.InfDer

mov (COORD PTR [edi]).X,50

mov edi,OFFSET rect2.InfDer.X

mov WORD PTR [edi], 50
```

Declaración y uso de uniones

Mientras que cada campo en una estructura tiene un desplazamiento relativo al primer byte de la estructura, todos los campos en una *unión* comienzan en el mismo desplazamiento. El tamaño de almacenamiento de una unión es igual a la longitud de su campo más grande. Cuando no forma parte de una estructura, una unión se declara mediante las directivas UNION y ENDS:

```
nombreunión UNION

   campos-de-la-estructura

nombreunión ENDS
```

Si la unión se anida dentro de una estructura, la sintaxis es un poco distinta:

```
nombreestruct STRUCT

   campos-de-la-estructura

   UNION nombreunión

      campos-de-la-unión

   ENDS

nombreestruct ENDS
```

Las declaraciones de los campos en una unión siguen las mismas reglas que para las estructuras, con la excepción de que cada campo sólo puede tener un inicializador. Por ejemplo, la unión **Entero** tiene tres atributos de tamaño distinto para los mismos datos, e inicializa todos los campos con cero:

```
Entero UNION

   D DWORD 0
```

```
   W WORD 0

   B BYTE 0

Entero ENDS
```

Debemos ser consistentes SI utilizamos los inicializadores, debemos tener valores consistentes. Supongamos que Entero se declara con distintos inicializadores:

```
Entero UNION

   D WORD 1

   W WORD 5

   B BYTE 8

Entero ENDS
```

Entonces declararíamos una variable Entero llamada **miEnt**, usando los inicializadores predeterminados:

```
.data

miEnt Entero <>
```

Los valores de miEnt.D, miEnt.W y miEnt.B serían todos iguales a 1. El ensamblador ignoraría inicializadores declarados para los campos W y B.

Estructura que contiene una unión Podemos probar una unión dentro de una estructura, usando el nombre de la unión en una declaración, como hemos hecho a continuación para el campo **IDArchivo** dentro de la estructura InfoArchivo.

```
InfoArchivo STRUCT

   IDArchivo Entero<>

   NombreArchivo BYTE 64 DUP(?)

InfoArchivo ENDS
```

o podemos declarar una unión directamente dentro de una estructura, como hemos hecho a continuación para el campo **IDArchivo**:

```
InfoArchivo STRUCT

UNION IDArchivo

 D DWORD ?

  W WORD ?
```

```
  B BYTE ?
ENDS
NombreArchivo BYTE 64 DUP(?)
InfoArchivo ENDS
```

Declaración y uso de variables de unión Una variable de unión se declara y se inicializa en forma muy parecida a una variable de estructura, con una importante diferencia: no se permite más de un inicializador. A continuación se muestran ejemplos de variable tipo Entero:

```
val1 Entero <12345678h>
val2 Entero <100h>
val3 Entero <>
```

Para utilizar una variable de unión en una instrucción ejecutable, debemos suministrar el nombre de uno de los campos variantes. En el siguiente ejemplo, asignamos valores de los registros a los campos de la unión **Entero**. Observamos la flexibilidad que tenemos al poder usar distintos tamaños de operandos:

```
mov val3.B, al
mov val3.W, ax
mov val3.D, eax
```

Las uniones también pueden contener estructuras. La siguiente estructura llamada INPUT_RECORD la utilizan algunas funciones de entrada de consola de MS Windows. Contiene una unión llamada **Event**, la cual selecciona uno de varios tipos de estructura predefinidas. El campo **EventType** indica el tipo de registro que aparece en la unión. Cada estructura tiene una distribución y tamaño distinto, pero solo se utiliza una a la vez:

```
INPUT_RECORD STRUCT
 EventType WORD ?
 ALIGN DWORD
 UNION Event
   KEY_EVENT_RECORD <>
   MOUSE_EVENT_RECORD <>
   WINDOWS_BUFFER_SIZE_RECORD <>
   MENU_EVENT_RECORD <>
```

```
   FOCUS_EVENT_RECORD <>
ENDS
INPUT_RECORS ENDS
```

A menudo, la API Win32 incluye la palabra RECORD al nombra las estructuras. Ésta es la definición de una estructura KEY_EVENT_RECORD:

```
KEY_EVENT_RECORD STRUCT
  bKeyDown DWORD ?
  wRepeatCount WORD?
  wVirtualKeyCode WORD ?
  wVirtualScanCode WORD ?
  UNION uChar
   UnicodeChar WORD ?
   AsciiChar BYTE ?
ENDS
dwControlKeyState DWORD ?
KEY_EVENT_RECORD ENDS
```

El resto de las definiciones STRUCT de INPUT_RECORD pueden ser encontradas en SmallWin.inc.

Macros

Generalidades

Un *macro procedimiento* es un bloque con nombre de instrucciones en lenguaje ensamblador. Una vez definido, puede invocarse (llamarse) todas las veces que se desee en un programa. Al *invocar* a un macro procedimiento, se inserta una copia de su código directamente en el programa, en la ubicación en la que se invocó. Se acostumbra utilizar el término *llamar* a un macro procedimiento, aunque técnicamente no hay una instrucción CALL implicada.

El término *macro procedimiento* se utiliza en el manual de Microsoft Assembler para identificar a los macros que no devuelven un valor. También hay *macro funciones* que devuelven un valor. Entre programadores, por lo general se entiende que la palabra *macro* significa lo mismo que *macro procedimiento*. De aquí en adelante, utilizaremos la forma más corta.

mediante una directiva INCLUDE. Los macros se expanden durante el paso del

preprocesamiento del ensamblador. EN este paso, el preprocesador lee la definición de un macro y explora el resto del código fuente en el programa. En cada punto en el que se hace una llamada al macro, el ensamblador inserta una copia del código de fuente en el programa. El ensamblador debe encontrar la definición de un macro antes de tratar de ensamblar cualquier llamada del macro. Si un programa define a un macro pero nunca lo llama, el código de ese macro no aparece en el programa compilado.

En el siguiente ejemplo, un macro llamado **ImprimirX** contiene una sola instrucción que llama al procedimiento **WriteChar** de Irvine32 o Irvine16. Por lo general, esta definición se coloca justo antes del segmento de datos:

```
ImprimirX MACRO
   mov al,'X'
   call WriteChar
ENDM
```

A continuación, en el segmento de código llamamos al macro:

```
.code
ImprimirX
```

Cuando el preprocesador explora este programa y descubre la llamada a **ImprimirX**, sustituye la llamada al macro con las siguientes instrucciones:

```
mov al,'X'
call WriteChar
```

Se ha llevado a cabo la sustitución de texto. Si bien el macro no es muy flexible, pronto mostraremos cómo pasar argumentos a los macros, para que sean mucho más útiles

Definición de macros

Un macro se define usando las directivas MACRO y END. La sintaxis es:

```
nombremacro MACRO parámetro-1, parámetro-2…
   lista-instrucciones
ENDM
```

No hay una regla fija en relación con la sangría, pero se recomienda aplicar sangría a las instrucciones entre *nombremacro* y ENDM. También podría ser conveniente colocar prefijos en los nombres de los macros con la letra m, para crear nombres reconocibles como **mColocarCar**, **mEscribirCadena** y **mGotoxy**. Las instrucciones entre la directiva MACRO y ENDM

no se ensamblan sino hasta que se llama al macro. Puede haber cualquier número de parámetros en la definición del macro, separados por comas.

Parámetros Los parámetros de los macros son receptáculos para los argumentos de texto que se pasan al procedimiento que hace la llamada. De hecho, los argumentos pueden ser enteros, nombres de variables u otros valores, pero el preprocesador los trata como texto. Los parámetros no tienen tipo, por lo que el preprocesador no comprueba los tipos de argumentos para ver si son correctos. Si ocurre un conflicto de tipos, el ensamblador lo atrapa después de que se expande el macro.

Ejemplo: mColocarCar El siguiente macro llama **mColocarCar** recibe un solo parámetro de entrada llamado **car** y lo muestra en consola, mediante una llamada a **WriteChar** de la biblioteca de enlace del libro:

```
mColocarCar MACRO car

   push eax

   mov al,car

   call WriteChar

   pop eax

ENDM
```

Invocación de macros

Para llamar (invocar) a un macro se inserta su nombre en el programa, posiblemente seguido de los argumentos del macro. La sintaxis para llamar a un macro es

```
nombremacro argumento-1, argumento-2, …
```

Nombremacro debe ser el nombre de un macro definido antes de este punto en el código de fuente. Cada argumento es un valor de texto que sustituye a un parámetro en el macro. El orden de los argumentos debe corresponder al orden de los parámetros, pero el número de argumentos no tiene que coincidir con el número de parámetros. Si se pasan demasiados argumentos, el ensamblador genera una advertencia. Si se pasan muy pocos argumentos a un macro, los parámetros faltantes se dejan en blanco.

Invocación de mColocarCar En la parte anterior definimos al marco **mColocarCar**. Al invocarlo, podemos pasarle cualquier carácter o código ASCII. La siguiente instrucción invoca a mColocarCar y le pasa la letra A:

```
mColocarCar ‘A’
```

El preprocesador del ensamblador expande la instrucción en el siguiente código, mostrando en el archivo de listado:

```
1 push eax

1 mov al,’A’

1 call WriteChar

1 pop eax
```

El 1 en la columna izquierda indica el nivel de expansión del macro, que se incrementa cuando se hacen llamadas a otros macros dentro de un macro. El siguiente ciclo muestra las primeras 20 letras del alfabeto:

```
mov al,’A’

mov ecx,20

L1:

mColocarCar al                  ; llamada al macro

inc al

loop L1
```

El preprocesador expande nuestro ciclo en el siguiente código (visible en el archivo de listado de código fuente). La llamada al macro se muestra justo antes de su expansión:

```
   mov al,’A’

   mov ecx,20

L1:

   mColocarCar al               ; llamada al macro

1 push eax

1 mov al,al

1 call WriteChar

1 pop eax

 inc al

 loop L1
```

En general, los macros se ejecutan con más rapidez que los procedimientos, ya que éstos tienen la sobrecarga adicional de las instrucciones CALL y RET. Sin embargo, existe la desventaja en cuanto al uso de los macros: el uso repetido de macros extensos tiende a incrementar el tamaño de un programa, ya que cada llamada a un macro inserta una nueva copia de las instrucciones del macro en el programa.

Características adicionales de los macros

Parámetros requeridos

Mediante el uso del calificador REQ, podemos especificar que un parámetro de un macro es requerido. Si el macro se llama sin un argumento que coincida con el parámetro requerido, el ensamblador muestra un mensaje de error. Si un macro tiene varios parámetros requeridos, cada uno debe incluir el calificador REQ. En el siguiente macro **mColocarCar**, se requiere el parámetro **car**:

```
mColocarCar MACRO car:REQ
   push eax
   mov al,car
   call WriteChar
   pop eax
ENDM
```

Comentarios en los macros

Las líneas de comentarios ordinarias que aparecen en la definición de macro, aparecen cada vez que ésta se expande. Si deseamos tener comentarios en el macro que no aparezcan en las expansiones, debemos empezarlos con doble signo de punto y coma (;;):

```
mColocarCar MACRO car:REQ      ;; recordatorio: car debe ser de 8 bits
   push eax
   mov al,car
   call WriteChar
   pop eax
ENDM
```

Directiva ECHO

La directiva ECHO muestra un mensaje en la consola, a medida que éste se ensambla. En la siguiente versión de **mColocarCar**, aparece el mensaje “Expandiendo el macro mColocarCar” en la consola durante el ensamblado:

```
 mColocarCar MACRO car:REQ
   ECHO Expandiendo el macro mColocarCar
```

```
   push eax
   mov al,car
   call WriteChar
   pop eax
ENDM
```

Directiva LOCAL

A menudo, las definiciones de los macros contienen etiquetas y hacen autorreferencias a esas etiquetas en su código. Por ejemplo, el siguiente macro **crearCadena** declara una variable llamada **cadena** y la inicializa con un arreglo de caracteres:

```
crearCadena MACRO texto
   .data
   cadena BYTE texto,0
ENDM
```

Supongamos que invocamos el macro dos veces:

CrearCadena “Hola”

CrearCadena “Adios”

Se produce un error, ya que el ensamblador no permite redefinir la etiqueta **cadena**:

```
 crearCadena “Hola”
1 .data
1 cadena BYTE “Hola”,0
 CrearCadena “Adios”
1 .data
1 cadena BYTE “Adios”,0        ; ¡error!
```

Uso de LOCAL para evitar los problemas ocasionados por las redefiniciones de etiquetas, podemos aplicar la directiva LOCAL a las etiquetas dentro de la definición de un macro. Cuando una etiqueta se marca como LOCAL, el preprocesador convierte el nombre de esa etiqueta en un identificador único, cada vez que se expande el macro. He aquí una versión de **crearCadena** en la que se utiliza a LOCAL:

```
crearCadena MACRO texto
```

```
   LOCAL cadena

   .data

   cadena BYTE texto,0

ENDM
```

Si invocamos al macro dos veces como antes, el código que genera el preprocesador sustituye cada ocurrencia de **cadena** con un identificador único:

```
crearCadena "Hola"

1 .data

1 ??0000 BYTE "Hola",0

 crearCadena "Adios"

1 .data

1 ??0001 BYTE "Adios",0
```

Los nombres de las etiquetas producidos por el ensamblador toman la forma *??nnnn*, en donde *nnnn* es un entero único. La directiva LOCAL también debe usarse para las etiquetas de código en los macros.

Macros que contienen código y datos

A menudo, los macros contienen tanto código como datos. Por ejemplo, el siguiente macro **mEscribir** (mWrite) muestra una cadena literal en la consola:

```
mWrite MACRO texto

   LOCAL cadena                ;; datos locales

   cadena BYTE texto,0         ;; define la cadena

   .code

   push edx

   mov edx,OFFSET cadena

   call WriteString

   pop edx

ENDM
```

Las siguientes instrucciones invocan al macro dos veces, y le pasan distintas literales de cadena:

```
mWrite “Por favor escriba su nombre de pila”

mWrite “Por favor escriba su apellido”
```

La expansión que hace el ensamblador de las dos instrucciones muestra que a cada cadena se les asigna una etiqueta única y las instrucciones mov se ajustan de manera acorde:

```
   mWrite “Por favor escriba su nombre de pila”
1 .data
1 ??0000 BYTE “Por favor escriba su nombre de pila”,0
1 .code
1 push edx
1 mov edx, OFFSET ??0000
1 call WriteString
1 pop edx
   mWrite “Por favor escriba su apellido”
1 .data
1 ??0001 BYTE “Por favor escriba su apellido”,0
1 .code
1 push edx
1 mov edx, OFFSET ??0001
1 call WriteString
1 pop edx
```

Macros anidados

A un macro que se invoca desde otro macro se le conoce como *macro anidado*. Cuando el preprocesador del ensamblador encuentra una llamada a un macro anidado, lo expande en ese lugar. Los parámetros que se pasan a un macro que encierra a otro macro, se pasan directamente a sus macros anidados.

Ejemplo mWriteLn El siguiente macro **mWriteLn** escribe una literal de cadena en la consola y la adjunta a un fin de línea. Invoca **mWrite** y llama al procedimiento **Crlf**:

```
mWriteLn MACRO texto

   mWrite texto

   call Crlf

ENDM
```

El parámetro texto se pasa directamente a **mWrite**, Supongamos que la siguiente instrucción invoca a mWriteLn:

```
mWriteLn "Mi programa de ejemplo de macros"
```

En la expansión de código resultante, el nivel de anidamiento (2) enseguida de las instrucciones indica que se ha invocado a un macro anidado:

```
mWriteLn "Mi programa de ejemplo de macros"

2 .data

2 ??0002 BYTE "Mi programa de ejemplo de macros",0

2 .code

2 push edx

2 mov edx,OFFSET ??0002

2 call WriteString

2 pop edx

1 call Crlf
```

Directivas de ensamblado condicional

Podemos utilizar una variable de directivas de ensamblado condicional distintas en conjunto con los macros, para que éstos sean más flexibles. La sintaxis general para las directivas de ensamblado condicional es:

```
IF condición

  instrucciones

[Else

  instrucciones]

ENDIF
```

Las directivas constantes que se muestran no deben confundirse con las directivas en tiempo de ejecución, como .IF y .ENDIF. Estas últimas evalúan expresiones con base en los valores en tiempo de ejecución almacenados en registros y variables.

La siguiente tabla lista las directivas de ensamblado condicional más comunes. Cuando las descripciones indican que una directiva *permite el ensamblado*, significa que todas las instrucciones subsiguientes se ensamblan hasta la siguiente directiva ELSE o ENDIF. Hay que enfatizar que las directivas que se listan en la tabla se evalúan en tiempo de ensamblado, no en tiempo de ejecución.

Directiva	Descripción
IF *expresión*	Permite el ensamblado si el valor de *expresión* es verdadero (distinto de cero). Los posibles operadores relacionados son LT, GT, EQ, NE, LE y GE
IFB <*argumento*>	Permite el ensamblado si *argumento* está en blanco. El nombre del argumento debe ir encerrado entre los signos <>
IFNB <*argumento*>	Permite el ensamblado si *argumento* no está en blanco. El nombre del argumento debe ir encerrado entre signos <>
IFIDN <*arg1*>,<*arg2*>	Permite el ensamblado si los dos argumentos son iguales. Usa una comparación sensible a mayúsculas y minúsculas
IFIDNI <*arg1*>,<*arg2*>	Permite el ensamblado si los dos argumentos son iguales. Usa una comparación insensible a mayúsculas y minúsculas
IFDIF <*arg1*>,<*arg2*>	Permite el ensamblado si los dos argumentos no son iguales. Usa una comparación sensible a mayúsculas y minúsculas
IFDIFI <*arg1*>,<*arg2*>	Permite el ensamblado si los dos argumentos no son iguales. Usa una comparación insensible a mayúsculas y minúsculas
IFDEF *nombre*	Permite el ensamblado si *nombre* está definido
IFNDEF *nombre*	Permite el ensamblado si *nombre* no está definido
ENDIF	Termina un bloque que se empezó con una de las directivas de ensamblado condicional
ELSE	Termina el ensamblado de las instrucciones anteriores si la condición es Verdadera. Si la condición es falsa. ELSE ensambla las instrucciones hasta la siguiente instrucción ENDIF
ELSEIF *expresión*	Ensambla todas las instrucciones hasta ENDIF si la condición especificada por una directiva condicional anterior es falsa y el valor de la expresión actual es verdadero

EXITM	Sale del macro inmediatamente, evitando que se expandan todas las instrucciones siguientes del macro

Comprobación de argumentos faltantes

Un macro puede comprobar si alguno de sus argumentos está en blanco. A menudo, si un macro recibe un argumento en blanco, se producen instrucciones inválidas cuando el preprocesador expande el macro. Por ejemplo, si invocamos el macro **mWriteString** sin pasarle un argumento, el macro se expande con una instrucción inválida al mover el desplazamiento de la cadena a EDX. A continuación se muestran instrucciones generadas por el ensamblador, que detecta el operando faltante y genera un mensaje de error:

```
mWriteString

1 push edx

1 mov edx,OFFSET

Macro2.asm(18) : error A2081: missing operand after unary operator

1 call WriteString

1 pop edx
```

Para evitar los errores ocasionados por operandos faltantes, podemos usar la directiva IFB (*si está en blanco*), la cual devuelve verdadero si un argumento del macro está en blanco. O también podemos usar el operador IFNB (*si no está en blanco*), el cual devuelve verdadero si el argumento de un macro no está en blanco. Vamos a crear una versión alternativa de **mWriteString** que muestra un mensaje de error durante el ensamblado:

```
mWriteString MACRO cadena

IFB <cadena>

ECHO ----------------------------------------

ECHO * Error: falta un parámetro en mWriteString

ECHO * (no se genero código)

ECHO ----------------------------------------

EXITM

ENDIF

push edx
```

```
mov edx,OFFSET cadena

call WriteString

pop edx

ENDM
```

Como vimos la directiva ECHO escribe un mensaje en la consola mientras se ensambla un programa. La directiva EXITM indica al preprocesador que debe salir del macro sin expandir más instrucciones de la misma.

Inicializadores de argumentos predeterminados

Los macros pueden tener inicializadores de argumentos predeterminados. Si falta un argumento del macro cuando se hace la llamada a ésta, se utiliza el argumento predeterminado. La sintaxis es:

```
nombreParam := <argumento>
```

(Los espacios antes y después de los operandos son opcionales). Por ejemplo, el macro **mWriteLn** puede proporcionar una cadena que contenga un solo espacio como argumento predeterminado. Si se llama sin argumentos, de todas formas imprime un espacio, seguido de un fin de línea:

```
mWriteLn MACRO texto:=<” “>

   mWrite texto

   call Crlf

ENDM
```

El ensamblador genera un error si se utiliza una cadena nula (“”) como argumento predeterminado, por lo que debemos insertar cuando menos un espacio entre comillas.

Expresiones booleanas

El ensamblador nos permite utilizar los siguientes operadores relacionales en expresiones booleanas constantes que contenga a IF y otras directivas condicionales:

```
LT Menor que

GT Mayor que

EQ Igual que

NE Distinto que

LE Menor o igual que
```

```
                                GE mayor o igual que
```

Directivas IF, ELSE y ENDIF

La directiva IF debe ir seguida de una expresión booleana constante. La expresión puede contener constantes enteras, simbólicas o argumentos de macro constantes, pero no puede contener registro ni nombres de variables. Un formato de sintaxis utiliza solo a IF y ENDIF:

```
IF expresión
   lista-instrucciones
ENDIF
```

Otro formato utiliza a IF, ELSE y ENDIF:

```
IF expresión
   lista-instrucciones
ELSE
   lista-instrucciones
ENDIF
```

Ejemplo: el macro mGotoxyConst El macro **mGtoxy** utiliza los operadores LT y GT para realizar la comprobación de rangos en los argumentos que se pasan al macro. Los argumentos X y Y deben ser constantes. Otro símbolo constante llamado ERRS cuenta el número de errores encontrados. Dependiendo del valor de X, podemos establecer ERRS a 1. Dependiendo del valor de Y, podemos sumar 1 a ERRS. Por último, si ERRS es mayor que cero, la directiva EXITM termina el macro:

```
;-----------------------------------------------
mGotoxyConst MACRO X:REQ, Y:REQ
; Establece la posición del cursor en la columna X, fila Y.
; Requiere que las coordenadas X y Y sean expresiones constantes
; en los rangos 0 <= X < 80 y 0 <= Y < 24
;-----------------------------------------------
LOCAL ERRS                      ;; constante local
ERRS = 0
IF (X LT 0) OR (X GT 79)
```

```
   ECHO Advertencia: El primer argumento para mGotoxy (X) está fuera de
rango.

   ECHO ********************************************

   ERRS = 1

ENDIF

IF ( Y LT 0) OR (Y GT 24)

   ECHO Advertencia: El segundo argumento para mGotoxy (Y) está fuera de
rango.

   ECHO *********************************************

   ERRS = ERRS + 1

ENDIF

IF ERRS GT 0                  ;; sí se encontraron errores,

   EXITM                      ;; sale del macro

ENDIF

push edx

mov dh,Y

mov dl,X

call Gotoxy

pop edx

ENDM
```

Las directivas IFIDN e IFIDNI

La directiva IFIDNI realiza una comparación insensible a mayúsculas y minúscula entre dos símbolos (incluyendo los nombres de los parámetros del macro) y devuelve verdadero si son iguales. La directiva IFIDN realiza una comparación sensible a mayúsculas y minúsculas. Esta última es útil cuando deseamos asegurarnos de que el procedimiento que llamó al macro no haya utilizado un argumento de registro que pueda estar en conflicto con el uso de los registros dentro del macro. La sintaxis para IFIDNI es

```
IFIDNI <símbolo>, <símbolo>

   instrucciones
```

```
ENDIF
```

La sintaxis para IFIDN es idéntica. Por ejemplo, en el siguiente macro **mReadBuf**, el segundo argumento no puede ser EDX ya que se sobrescribirá cuando el desplazamiento de **Búfer** se mueva a EDX. La siguiente versión revisada del macro muestra un mensaje de advertencia si no se cumple con este requerimiento:

```
;-----------------------------------------

mReadBuf MACRO apuntBufer, maxCars

;

; Lee de la entrada estándar hacia un búfer.

; Recibe: desplazamiento del búfer, cuenta del número máximo

;         de caracteres que pueden introducirse. El segundo argumento no
;         puede ser edx ni EDX

IFIDNI <maxCars> , <EDX>

   ECHO Advertencia: EDX no puede ser el segundo argumento para mReadBuf.

   ECHO ********************************************

   EXITM

ENDIF

push ecx

push edx

mov edx,apuntBufer

mov ecx,maxCars

call ReadString

pop edx

pop ecx

ENDM
```

La siguiente instrucción hace que el macro genere un mensaje de advertencia ya que EDX es el segundo argumento:

```
mReadBuf OFFSET búfer,edx
```

Operadores especiales

Hay cuatro operadores en ensamblador que hacen a los macros más flexibles.

&	Operador de sustitución
<>	Operador de texto-literal
!	Operador de carácter-literal
%	Operador de expansión

Operador de sustitución

El operador de *sustitución (&)* resuelve las referencias ambiguas a los nombres de los parámetros dentro de un macro. El macro **mShowRegister** muestra el nombre y el contenido en hexadecimal de un registro de 32 bits. A continuación se muestra una llamada de ejemplo:

```
.code

mShowRegister ECX
```

He aquí un ejemplo de los resultados generados por la llamada mShowRegister:

```
ECX =
00000101
```

Podría definirse una variable de cadena que contenga el nombre del registro dentro del macro:

```
mShowRegister MACRO nombreReg

.data

cadTemp BYTE "nombreReg=",0
```

Pero el preprocesador asumiría que **nombreReg** es parte de una literal de cadena, y no lo sustituiría con el valor del argumento que recibe el macro. En vez de ello, si agregamos el operador &, el preprocesador se ve obligado a insertar el argumento del macro (como ECX) en la literal de cadena. El siguiente ejemplo muestra cómo definir **cadTemp**:

```
mShowRegister MACRO nombreReg

.data

cadTemp BYTE "&nombreReg=",0
```

Operador de expansión (%)

El operador de *expansión (%)* expande macros de texto o convierte expresiones constantes en sus presentaciones de texto. Hace esto de varias formas. Cuando se utiliza con TEXTEQU, el operador % evalúa una expresión constante y convierte el resultado en un entero. En el siguiente ejemplo, el operador % evalúa la expresión (5 + cuenta) y devuelve el entero 15 (como texto):

```
cuenta = 10

valSuma TEXTEQU %(5 + cuenta)  ; = "15"
```

Si un macro requiere un argumento entero constante, el operador % nos da la flexibilidad de pasar una expresión entera. La expresión se evalúa a su valor entero, el cual se pasa a continuación del macro. Por ejemplo, al invocar **mGotoxyConst**, las siguientes expresiones se evalúan como 50 y 7:

```
mGotoxyConst %(5 * 10), %(3 +4)
```

El preprocesador produce las siguientes instrucciones:

```
1 push edx

1 mov dh,7

1 mov dl,50

1 call Gotoxy

1 pop edx
```

% al principio de la línea cuando el operador expresión (%) es el primer carácter en una línea de código fuente, instruye al preprocesador para que expanda todos los macros de texto y las macrofunciones que encuentre en la misma línea. por ejemplo, supongamos que deseamos mostrar el tamaño de un arreglo en la pantalla durante el ensamblado. Los siguientes intentos no producirán el resultado deseado:

```
.data

arreglo DWORD 1,2,3,4,5,6,7,8

.code

ECHO El arreglo contiene (SIZEOF arreglo) bytes

ECHO El arreglo contiene %(SIZEOF arreglo) bytes
```

El resultado en pantalla sería inútil:

```
 El arreglo contiene (SIZEOF arreglo) bytes

El arreglo contiene %(SIZEOF arreglo) bytes
```

Si en vez de ello utilizamos TEXTEQU para crear un macro que contenga (SIZEOF arreglo), ésta puede expandirse en la siguiente línea:

cadTemp TEXTEQU %(SIZEOF arreglo)

% ECHO El arreglo contiene CadTemp bytes

Se produce el siguiente resultado

```
El arreglo contiene 32 bytes
```

Visualización de un número de línea El siguiente macro **Mul32** multiplica sus primeros dos argumentos entre sí, y devuelve el producto en el tercer argumento. Sus parámetros pueden ser registros, operandos de memoria y operandos inmediatos (excepto el producto):

```
MUL32 MACRO op1, op2, producto
   IFIDNI <op2>,<EAX>
      NUMLINEA TEXTEQU %(@LINE)
      ECHO -------------------------------------------------------
%     ECHO * Error en la línea NUMLINEA: EAX no puede ser el segundo
      ECHO * argumento cuando se invoca al macro MUL32.
      ECHO -------------------------------------------------------
EXITM
ENDIF
push eax
mov eax,op1
mul op2
mov producto,eax
pop eax
ENDM
```

Mul32 comprueba un importante requerimiento: EAX no puede ser el segundo argumento. Lo interesante sobre este macro es que muestra el número de línea desde el cual se llamó, para facilitar el rastreo y la corrección del problema. Primero se define el macro de texto NUMLINEA.

Ésta hace referencia a @LINE, un operador predefinido del ensamblador que devuelve el número de línea de código actual:

```
NUMLINEA TEXTEQU %(@LINE)
```

A continuación, el operador de expansión(%) en la primera columna de la línea que contiene la instrucción ECHO hace que NUMLINEA se expanda:

```
% ECHO * Error en línea NUMLINEA: EAX no puede ser el segundo
```

Operador de texto-literal (<>)

El operador de *texto-literal (<>)* agrupa uno o más caracteres y símbolos en una sola literal de texto. Evita que el preprocesador interprete los miembros de la lista como argumentos separados. Este operador es muy útil cuando una cadena contiene caracteres especiales, como comas, signos de porcentaje (%), signos &, y signos de punto y coma (;), que de otra manera se interpretarían como delimitadores u otros operadores. Por ejemplo, el macro **mWrite** recibe una literal de cadena como su único argumento. Si le pasamos la siguiente cadena, al preprocesador la interpretará como tres argumentos de macro separados:

```
mWrite "Línea tres", 0dh, 0ah
```

El texto después de la primera coma se descartaría, ya que el macro sólo espera un argumento. Por otro lado si encerramos la cadena con el operador de texto-literal, el preprocesador considera que todo el texto entre los signos < > es sólo un argumento de macro:

```
mWrite <"Linea tres", 0dh, 0ah>
```

Operador de carácter-literal (!)

El operador de *carácter-literal* se inventó por la misma razón que el operador de texto-litera: obliga al preprocesador a tratar un operador predefinido como un carácter ordinario. En la siguiente definición TEXTEQU, el operador ! evita que el símbolo > sea un delimitador de texto:

```
ValorYIncorrecto TEXTEQU <advertencia: La coordenada Y es !> 24>
```

Ejemplo de mensaje de advertencia El siguiente ejemplo muestra cómo funcionan los operadores %, & y ! en conjunto. Vamos a suponer que definimos el símbolo **ValorYIncorrecto**. Podemos crear un macro llamado **MostrarAdvertencia** que reciba un argumento de texto, lo encierre entre comillas y pase la literal al macro **mWrite**. Observamos el uso del operador de sustitución (&):

```
MostrarAdvertencia MACRO mensaje

   mWrite "&mensaje"

ENDM
```

A continuación invocamos a **MostrarAdvertencia** y le pasamos la expresión %ValorYIncorrecto. El operador % evalúa (desreferencia) a **ValorYIncorrecto** y produce una cadena equivalente:

```
.code

MostrarAdvertencia %ValorYIncorrecto
```

Como era de esperarse, el programa se ejecuta y muestra el mensaje de advertencia:

```
Advertencia: La coordenada Y es > 24
```

Macrofunciones

Una macrofunción es similar a un macroprocedimiento, en cuanto a que se asigna un nombre a una lista de instrucciones en lenguaje ensamblador. La diferencia es que siempre devuelve una constante (entero o cadena) a través de la directiva EXITM. En el siguiente ejemplo, la macro **EstaDefinido** devuelve verdadero (-1) si se ha definido un símbolos dado; en caso contrario, devuelve falso (0):

```
EstaDefinido MACRO símbolo

   IFDEF símbolo

      EXIT <-1>                ;; Verdadero

   ELSE

      EXITM <0>                ;; Falso

ENDM
```

La directiva EXITM (salir de macro) detiene el resto de la expansión de la macro.

Llamada a una microfunción Al llamar a una microfunción, su lista de argumentos debe ir encerrada entre paréntesis. Por ejemplo, podemos llamar a la macro **EstaDefinido** y pasarle **RealMode**, el nombre de un símbolo que puede o no estar definido:

```
IF EstaDefinido( RealMode)

   mov ax,@data

   mov ds,ax

ENDIF
```

Si el ensamblador ya ha encontrado una definición de **RealMode** antes de este punto en el proceso de ensamblado, ensambla las dos instrucciones:

```
mov ax,@data

mov ds,ax
```

Puede colocarse la misma directiva IF dentro de una macro llamada **Inicio**:

```
Inicio MACRO
   IF EstaDefinido (RealMode)
      mov ax,@data
      mov ds,ax
   ENDIF
ENDM
```

Un macro como **EstaDefinido** puede ser útil cuando se diseñan programas para varios módulos de memoria. Por ejemplo, podemos usarla para determinar qué archivo de inclusión debemos usar:

```
IF EstaDefinido (RealMode)
   INCLUDE Irvine16.inc
ELSE
   INCLUDE Irvine32.inc
ENDIF
```

Definición de símbolo RealMode Todo lo que resta es encontrar una manera de definir el símbolo **RealMode**. Una forma es colocar la siguiente línea al principio de un programa:

```
RealMode = 1
```

De manera alternativa, la línea de comandos del ensamblador tiene una opción para definir símbolos, usando el modificador –D. El siguiente comando ML define el símbolo RealMode y le asigna un valor de 1:

```
ML –c –DRealMode=1 miProg.asm
```

El comando ML correspondiente para los programas en modo protegido no define el símbolo de RealMode:

```
ML –c miProg.asm
```

Definición de bloques de repetición

MASM cuenta con una variedad de directivas de iteración para generar bloques repetidos de instrucciones: WHILE, REPEAT, FOR y FORC. A diferencia de la instrucción LOOP, estas directivas funcionan sólo en tiempo de ensamblado, usando valores constantes como condiciones y contadores de ciclo:

- La directiva WHILE repite un bloque de instrucciones con base en una expresión booleana.
- la directiva REPEAT repite un bloque de instrucciones con base en el valor de un contador.
- La directiva FOR repite un bloque de instrucciones iterando a través de una lista de símbolos.
- Directiva FORC repite un bloque de instrucciones iterando a través de una cadena de caracteres.

Directiva WHILE

La directiva WHILE repite un bloque de instrucciones, siempre y cuando una expresión constante específica sea verdadera. La sintaxis es

```
WHILE expresiónConst
   instrucciones
ENDW
```

El siguiente código muestra cómo generar números de Fibonacci entre 1 y F0000000h como una serie de constantes en tiempo de ensamblado:

```
.data
val1 = 1
val2 = 1
DWORD val1                    ; primeros dos valores
DWORD val2
val3 = val1 + val2
WHILE val3 LT 0F0000000h
   DWORD val3
   val1 = val2
   val2 = val3
   val3 = val1 + val2
ENDM
```

Los valores generados por este código pueden verse en un archivo de listado (.LST)

Directiva REPEAT

La directiva REPEAT repite un bloque de instrucciones un número fijo de veces en tiempo de ensamblado. La sintaxis es:

```
REPEAT expresiónConst
   instrucciones
ENDM
```

ExpresiónConst, una expresión entera constante sin signo, determina el número de repeticiones.

REPEAT puede usarse en forma similar a DUP para crear un arreglo. En el siguiente ejemplo, la estructura IndicadoresClima contiene una cadena llamada ubicación, seguida por un arreglo de indicadores de lluvia y humedad:

```
SEMANAS_POR_ANIO = 52
IndicadoresClima STRUCT
   ubicacion BYTE 50 DUP(0)
   REPEAT SEMANAS_POR_ANIO
      LOCAL lluvia, humedad
      lluvia DWORD ?
      humedad DWORD ?
   ENDM
IndicadoresClima ENDS
```

Se utilizó la directiva LOCAL para evitar los errores ocasionados por la redefinición de lluvia y humedad, cuando se repita el ciclo en tiempo de ensamblado.

Directiva FOR

La directiva FOR repite un bloque de instrucciones al iterar a través de una lista de símbolos delimitados por comas. Cada símbolo en la lista produce una iteración de ciclo. La sintaxis es:

```
FOR parámetro, <arg1,arg2,arg3,…>
   Instrucciones
ENDM
```

En la primera iteración del ciclo, *parámetro* toma el valor de *arg1*; en la segunda iteración, *parámetro* toma el valor de *arg2*; y así sucesivamente hasta el último argumento en la lista.

Ejemplo de inscripción de estudiantes Vamos a crear un escenario de inscripción de estudiantes, en el que tendremos una estructura llamada CURSO, la cual contiene el número del curso y su número de créditos. Una estructura SEMESTRE contiene un arreglo de seis cursos y un contador llamado **NumCursos**:

```
CURSO STRUCT

   Numero BYTE 9 DUP(?)

   Creditos BYTE ?

CURSO ENDS

; Un semestre contiene un arreglo de cursos.

SEMESTRE STRUCT

   Curso CURSO 6 DUP(<>)

   NumCursos WORD ?

SEMESTRE ENDS
```

Podemos usar un ciclo FOR para definir cuatro objetos SEMESTRE, cada uno de los cuales tiene un nombre distinto, seleccionado de la lista de símbolos entre los signos <>:

```
.data

FOR nomSem,<Otonio1999, Primavera2000, Verano2000, Otonio2000>

   nomSem SEMESTRE <>

ENDM
```

Si inspeccionamos el archivo de listado, encontraremos las siguientes variables:

```
.data

Otonio1999 SEMESTRE <>

Primavera2000 SEMESTRE <>

Verano2000 SEMESTRE <>

Otonio2000 SEMESTRE <>
```

Directiva FORC

La directiva FORC repite un bloque de instrucciones a iterar a través de una cadena de caracteres. Cada carácter en la cadena provoca una iteración del ciclo. La sintaxis es

```
FORC parámetro, <cadena>

   instrucciones

ENDM
```

En la primera iteración de ciclo, *parámetro* es igual al primer carácter en la cadena; en la segunda iteración, *parámetro* es igual al segundo carácter en la cadena; y así sucesivamente, hasta el final de la cadena. El siguiente ejemplo crea una tabla de búsqueda de caracteres que consiste en varios caracteres no alfabéticos. Observamos que <y> debe ir precedidos por el operador de carácter-literal (!) para evitar que violen la sintaxis de la directiva PROC:

```
Delimitadores LABEL BYTE

FORC code,<@#$%^&*!<!>>

   BYTE “&code”

ENDM
```

Se genera la siguiente tabla de datos, que podemos ver en el archivo de listado:

```
00000000 401 BYTE “@”

00000001 231 BYTE “#”

00000002 241 BYTE “$”

…

…

00000006 2A1 BYTE “*”

00000007 3C1 BYTE “<”

00000008 3E1 BYTE “>”
```

11 Programación en MS-Windows

Antecedentes

Cuando se inicia una aplicación Windows, crea una ventana de consola o una ventana gráfica. Hemos estado usando la siguiente opción con el comando LINK en el archivo de procesamiento por lotes *make.bat*, el cual indica al enlazador que debe crear una aplicación basada en consola:

```
/SUBSYSTEM:CONSOLE
```

Un programa de consola se ve y se comporta como una ventana de MS-DOS, con algunas mejoras que veremos adelante. La consola tiene un solo búfer de entrada y uno o más búferes de pantalla:

- El *búfer de entrada* contiene una cola de *registros de entrada*, cada uno de los cuales contiene datos acerca de un evento de entrada. Algunos ejemplos de evento de entrada son: entrada del teclado, clics del ratón y cuando el usuario cambia el tamaño de la ventana de consola.
- Un *búfer de pantalla* es un arreglo bidimensional de datos de caracteres y colores, que afectan la apariencia del texto en la ventana de consola.

Información de referencia de la API Win32

Funciones A lo largo de esta sección, sólo podemos presentar una variedad de funciones de la API Win32 y proporcionar unos cuantos ejemplos. Existen muchos detalles que no podemos cubrir aquí. Para averiguar más podemos visitar el sitio Web de Microsoft MSDN (www.msdn.microsoft.com).

Constantes A menudo, cuando se lea la documentación para las funciones de la API Win32, encontraremos nombres constantes como TIME_ZONE_ID_UNKNOWN. En algunos casos, la constante ya estará definida en SmallWin.inc. Pero su no se encuentra ahí, podemos buscar en el sitio Web del libro. Por ejemplo, un archivo de encabezado llamado WinNT.h define a TIME_ZONE_ID_UNKNOWN junto con las constantes relacionadas:

```
#define TIME_ZONE_ID_UNKNOWN 0

#define TIME_ZONE_ID_STANDARD 1

#define TIME_ZONE_ID_DAYLIGHT 2
```

Usando esta información, podemos agregar lo siguiente a SmallWin.h o a nuestro propio archivo de inclusión:

```
TIME_ZONE_ID_UNKNOWN = 0

TIME_ZONE_ID_STANDARD = 1

TIME_ZONE_ID_DAYLIGHT = 2
```

Conjunto de caracteres y funciones de la API de Windows

Al llamar a las funciones en la API Win32, se utilizan dos tipos de conjuntos de caracteres; el conjunto de caracteres ASCII/ANSI de 8 bits y el conjunto Unicode 16 bits (disponible en Windows NT, 2000 y XP). Las funciones de Win32 que tienen que ver con texto, por lo general, se proporcionan en dos versiones, una que termina con la letra A (para caracteres ANSI 8 bits) y la otra que termina en W (para los conjuntos de caracteres *extensos*, incluyendo Unicode). Una de estas funciones es WriteConsole:

- WriteConsoleA.

- WriteConsoleW.

Windows 95 o 98 no soporta los nombres de las funciones que terminan en W. Por otro lado, en Windows NT, 2000 y XP, Unicode es el conjunto de caracteres nativo. Por ejemplo, si llamamos a una función como **WriteConsoleA**, el sistema operativo convierte los caracteres de ANSI a Unicode y llama a **WriteConsoleW**.

En la documentación de Biblioteca de Microsoft MSDN para funciones como WriteConsole, la letra A o W a la derecha se omite del nombre. En el archivo de inclusión para los programas de este libro, redefinimos los nombres de las funciones como **WriteConsoleA**:

```
WriteConsole EQU <WriteConsoleA>
```

Esta definición hace que sea posible llamar a WriteConsole usando su nombre genérico.

Acceso de alto y bajo nivel

Hay dos niveles de acceso a la consola, que permiten concesiones entre simplicidad y control completo:

- Las funciones de consola de alto nivel leen un flujo de caracteres del búfer de entrada de la consola. Escriben datos tipo carácter al búfer de pantalla de la consola. Tanto la entrada como la salida puede redirigirse para leer o escribir en/desde archivos de texto.
- Las funciones de consola de bajo nivel obtienen información detallada acerca de los eventos de teclado y de ratón, y las interacciones del usuario con la ventana de consola (arrastrar, cambiar de tamaño, etcétera). Estas funciones también permiten un control detallado del tamaño y posición de la ventana, así como los colores de texto.

Tipos de datos de Windows

Las funciones Win32 se documentan usando las declaraciones de funciones para programadores de C/C++. En estas declaraciones, los tipos de todos los parámetros de las funciones se basan en los tipos estándar de C o en uno de los tipos predefinidos de MS-Windows (a continuación se muestra una lista parcial). Es importante diferenciar los valores de datos de los apuntadores a los valores. El nombre de un tipo que empieza con la letra LP es un *apuntador largo (long)* a algún objeto.

Tipo de MS-Windows	Tipo de MASM	Descripción
BOOL, BOOLEAN	DWORD	Un valor booleano (TRUE o FALSE)
BYTE	BYTE	Un entero de 8 bits sin signo
CHAR	BYTE	Un carácter ANSI de Windows de 8 bits

COLORREF	DWORD	Un valor de 32 bits que se usa como valor de color
DWORD	DWORD	Un entero de 32 bits sin signo
HANDLE	DWORD	Manejador de un objeto
HFILE	DWORD	Manejador de un archivo abierto mediante OpenFile
INT	SDWORD	Un entero de 32 bits con signo
LONG	SDWORD	Un entero de 32 bits con signo
LPARAM	DWORD	Parámetro de mensaje, usado por los procedimientos de ventana y las funciones de devolución de llamada (callback)
LPCSTR	PTR BYTE	Un apuntador de 32 bits a una cadena constante con terminación nula de caracteres Windows (ANSI) de 8 bits
LPCVOID	DWORD	Apuntador a una constante de cualquier tipo
LPSTR	PTR BYTE	Un apuntador de 32 bits a una cadena con terminación nula de caracteres Windows (ANSI) de 8 bits
LPCTSTR	PTR WORD	Un apuntador de 32 bits a una cadena de caracteres constantes, que es portable para Unicode y los conjuntos de caracteres de doble byte
LPTSTR	PTR WORD	Un apuntador de 32 bits a una cadena de caracteres que es portable para Unicode y los conjuntos de caracteres de doble byte
LPVOID	DWORD	Un apuntador de 32 bits a un tipo no especificado
LRESULT	DWORD	Un valor de 32 bits devuelto de un procedimiento de ventana o función de devolución de llamada (callback)
SIZE_T	DWORD	El número máximo de bytes a los que puede apuntar un apuntador
UINT	DWORD	Un entero de 32 bits sin signo
WNDPROC	DWORD	Un apuntador de 32 bits a un procedimiento de ventana
WORD	WORD	Un entero de 16 bits sin signo
WPARAM	DWORD	Un valor de 32 bits que pasa como parámetro a un procedimiento de ventana o función de devolución de llamada (callback)

Archivo de inclusión SmallWin.inc

SmallWin.inc, es un archivo de inclusión que contiene definiciones de constantes, asociaciones de texto y prototipos de funciones para la programación con la API Win32. Se incluye de manera automática en los programas mediante Irvine32.inc, que hemos estado usando a lo largo de este resumen. El archivo se encuentra. El archivo se encuentra en la carpeta en donde se instaló los programas de ejemplo de este resumen. Encontraremos la mayoría de las constantes en Windows.h, un archivo de encabezado que se utiliza para programar en C y C++. A pesar de su nombre, SmallWin.inc es bastante extenso, por lo que veremos solo una sintaxis:

```
DO_NOT_SHARE = 0

NULL = 0

TRUE = 1

FALSE = 0

; manejadores de consola Win32

STD_INPUT_HANDLE EQU -10

STD_OUTPUT HANDLE EQU -11

STD_ERROR_HANDLE EQU -12
```

El tipo HANDLE, un alías para DWORD, ayuda a nuestros prototipos de función a ser más consistente con la documentación de Microsoft Win32:

```
HANDLE TEXTEQU <DWORD>
```

SmallWin.inc también incluye las definiciones de las estructuras utilizadas en las llamadas a Win32. Una de estas muestra a continuación:

```
COORD STRUCT

   X WORD ?

   Y WORD ?

COORDS ENDS
```

Por último, SmallWin.inc contiene los prototipos de función para todas las funciones de Win32 que se documentan en este capítulo.

Manejadores de consola

Casi todas las funciones de la consola de Win32 requieren recibir un manejador como primer argumento. Un *manejador* es un entero de 32 bits sin signo que identifica en forma única a un

objeto, como un mapa de bits, una pluma de dibujo o cualquier otro dispositivo de entrada/salida:

```
STD_INPUT_HANDLE                entrada estándar

STD_OUTPUT_HANDLE               salida estándar

STD_ERROR_HANDLE                salida de error estándar
```

Los últimos dos manejadores se utilizan al escribir en el buffer de pantalla activo de la consola.

La función **GetStdHandle** devuelve un manejador para un flujo de la consola: entrada, salida o salida de error. Necesitamos un manejador para poder realizar operaciones de entrada/salida en un programa basado en la consola. He aquí el prototipo de función:

```
GetStdHandle PROTO,

   nStdHandle:HANDLE            ; tipo de manejador
```

nStdHandle puede ser STD_INPUT_HANDLE, STD_OUTPUT_HANDLE o STD_ERROR_HANDLE.

La función devuelve el manejador en EAX, que debe copiarse a una variable por protección. He aquí una llamada de ejemplo:

```
.data

manejadorEntrada HANDLE ?

.code

   INVOKE GetStdHandle, STD_INPUT_HANDLE

   mov manejadorEntrada, eax
```

Funciones de la consola Win32

La siguiente tabla contiene una referencia rápida al conjunto completo de funciones de consola Win32. Se puede encontrar una descripción completa de cada función en la biblioteca MSDN, en www.msdn.microsoft.com

Tip: La función de la API Win32 no preservan EAX, EBX, ECX y EDX, por lo que debemos meter y sacar estos registros por nuestra cuenta.

Funciones de consola Win32.

Función	Descripción
AllocConsole	Asigna una nueva consola para el proceso que hace la llamada

CreateConsoleScreenBuffer	Crea un búfer de pantalla de consola
ExitProcess	Termina un proceso y todos sus subprocesos
FillConsoleOutputAttribute	Establece los atributos de texto y color de fondo para un número específico de celdas de caracteres
FillConsoleOutputCharacter	Escribe un carácter en el buffer de pantalla, un número especificado de veces
FlushConsoleInputBuffer	Vacía el buffer de entrada de la consola
FreeConsole	Desconecta el proceso que hizo la llamada de la consola
GenerateConsoleCtrlEvent	Envía una señal especificada a un grupo de proceso de control que comparte la consola asociada con el proceso que hizo la llamada
GetConsoleCP	Obtiene la página de código de entrada utilizada por la consola asociada con el proceso que hizo la llamada
GetConsoleCursorInfo	Obtiene información acerca del tamaño y la visibilidad del cursor para el búfer de pantalla de consola especificado.
GetConsoleMode	Obtiene el modo de entrada actual del búfer de entrada de una consola, o el modo de salida actual de un buffer de pantalla de consola
GetConsoleOutputCP	Obtiene la página de código de salida que utiliza la consola asociada con el proceso que hizo la llamada
GetConsoleScreenBufferInfo	Obtiene información acerca del búfer de pantalla de consola especificado
GetConsoleTitle	Obtiene la cadena de la barra de título para la ventana de consola actual
GetConsoleWindow	Obtiene el manejador de ventana utilizado por la consola asociada con el proceso que hizo la llamada
GetLargestConsoleWindowSize	Obtiene el tamaño de la ventana de consola más grande posible
GetNumberOfConsoleInputEvents	Obtiene el número de registros de entrada no leídos en el búfer de entrada de la consola
GetNumberOfConsoleMouseButtons	Obtiene el número de botones en el ratón, utilizados por la consola actual
GetStdHandle	Obtiene un manejador para la entrada estándar, la salida

	estándar o el dispositivo de error estándar
HandlerRoutine	Una función definida por la aplicación, que se utiliza con la función SetConsoleCtrlHandler
PeekConsoleInput	Lee datos del búfer de entrada de consola especificado, sin eliminarlos del búfer
ReadConsole	Lee la entrada de caracteres del búfer de entrada de la consola y la elimina del búfer
ReadConsoleInput	Lee datos de un búfer de entrada de consola y los elimina del búfer
ReadConsoleOutput	Lee los datos de los caracteres y atributos de color de un bloque rectangular de celdas de caracteres en un búfer de pantalla de consola
ReadConsoleScreenBuffer	Mueve un bloque de datos en un búfer de pantalla
SetConsoleActiveScreenBuffer	Establece el búfer de código de entrada utilizada por la consola asociada con el proceso que hizo la llamada
SetConsoleCtrlHandler	Agrega o elimina una función HandlerRoutine definida por una aplicación, de la lista de funciones manejadoras para el proceso que hizo la llamada
SetConsoleCursorInfo	Establece el tamaño y visibilidad del cursor para el búfer de pantalla de consola especificado
SetConsoleCursorPosition	Establece la posición del cursor en el búfer de pantalla de consola especificado
SetConsoleMode	Establece el modo de entrada del búfer de entrada de una consola, o el modo de salida de un búfer de pantalla de consola
SetConsoleModeOutputCP	Establece la página de código de salida utilizada por la consola asociada con el proceso que hizo la llamada
SetConsoleScreenBufferSize	Modifica el tamaño del buffer de pantalla de consola especificado
SetConsoleTextAttribute	Establece los atributos de color de texto y de fondo de los caracteres que se escriben en el búfer de pantalla
SetConsoleTitle	Establece la cadena de la barra de título para la ventana de consola actual
SetConsoleWindowInfo	Establece el tamaño y posición actuales de la ventana de un

	buffer de pantalla de consola
SetStdHandle	Establece el manejador para la entrada estándar, la salida estándar o el dispositivo de error estándar
WriteConsole	Escribe una cadena de caracteres en un búfer de pantalla de consola, empezando en la posición actual del cursor
WriteConsoleInput	Escribe datos directamente en el búfer de entrada de consola
WriteConsoleOutput	Escribe los datos de los caracteres y atributos de color en un bloque rectangular especificado de celdas de caracteres de un búfer de pantalla de consola
WriteConsoleOutputAttribute	Copia un número de atributos de color de texto y de fondo en las celdas consecutivas de un búfer de pantalla de consola
WriteConsoleOutputCharacter	Copia un número de caracteres en las celdas consecutivas en un búfer de pantalla de consola

Visualización de un cuadro de mensaje

Una de las maneras más sencillas de generar resultados en una aplicación Win32 es llamar a la función **MessageBoxA**:

```
MessageBoxA PROTO,

   hWnd:DWORD,                ; manejador para la ventana (puede ser
nulo)

   lpText:PTR BYTE,           ; cadena, dentro del cuadrado

   lpCaption: PTR BYTE,       ; cadena, titulo del cuadrado de diálogo

   uType:DWORD                ; contenido y comportamiento
```

En las aplicaciones basadas en consola, podemos establecer *hWnd* a NULL, indicando que el cuadro de mensaje no tiene propietario. El parámetro *lpText* es un apuntador a la cadena con terminación nula que deseamos colocar en el cuadro de mensaje. El parámetro *lpCaption* apunta a una cadena con terminación nula para el título del cuadro del diálogo. El parámetro *uType* especifica el contenido especifica el contenido y comportamiento del cuadro de diálogo.

Contenido y comportamiento El parámetro *uType* guarda un entero con asignación de bits que contiene tres tipos de opciones: los botones a visualizar, los íconos y la opción del botón predeterminada. Hay varias combinaciones de botones posibles:

- MB_OK

- MB_OKCANCEL
- MB_YESNO
- MB_YESNOCANCEL
- MB_RETRYCANCEL
- MB_ABORTRETRYIGNORE
- MB_CANCELTRYCONTINUE

Botón predeterminado Podemos elegir que botón se seleccionará de manera automática si el usuario presiona Intro. Las opciones son MB_DEFBUTTON1 (la predeterminada), MB_DEFBUTTON2, MB_DEFBUTTON3 y DEFBUTTON4. Los botones están numerados a partir de la izquierda, comenzando con 1.

Iconos Hay cuatro opciones de íconos disponibles. Algunas veces más de una constante produce el mismo ícono:

- Signo de alto: MB_ICONSTOP, MB_ICONHAND o MB_ICONERROR
- Signo de interrogación (?): MB_ICONQUESTION
- Símbolo de información (i): MB_ICONINFORMATION, MB_ICONASTERISK
- Signo de exclamación (!): MB_ICONEXCLAMATION, MB_ICONWARNING

Valor de retorno Si MessageBoxA falla, devuelve cero. En cualquier otro caso, devuelve un entero que especifica que botón oprimió el usuario al cerrar el cuadrado. Las opciones son IDABORT, IDCANCEL, IDCONTINUE, IDIGNORE, IDNO, IDOK, IDRETRY, IDTRYAGAIN e IDYES. Todas están definidas en SmallWin.inc

SmallWin.inc redefine a **MessageBoxA** como **MessageBox**, el cual parece un nombre más amigable para el usuario.

Programa de demostración

```
TITLE Demostración de MessageBoxA  (MessageBox.asm)

INCLUDE Irvine32.inc

.data

leyenda BYTE “Intento de dividir entre cero”,0

advertenciaMsj BYTE “Por favor revise su denominador.”,0

leyendaP BYTE “Pregunta”,0

PreguntaMsj BYTE “Deseas saber mi nombre?”,0

mostrarMiNombre BYTE “Mi nombre es MASM”,0dh,0ah,0

leyendaC BYTE “Informacion”,0
```

```
infoMsj BYTE “Su archivo se elimino.”,0dh,0ah
        BYTE “Notificar al admin del sistema, o restaurar copia de
respaldo?”,0
.code
main PROC
; Muestra un mensaje de advertencia.
   INVOKE MessageBox, NULL, ADDR AdvertenciaMsj, ADDR leyendaA,
          MB_OK + MB_ICONEXCLAMATION
; Hace una pregunta, evalúa la respuesta.
   INVOKE MessageBox, NULL, ADDR PreguntaMsj,
          ADDR leyendaP, MB_YESNO + MB_ICONQUESTION
   cmp eax,IDYES               ; ¿hizo clic en el botón SÍ?
   jne L2
; escribe el nombre en la ventana de consola.
   mov edx,OFFSET mostrarMiNombre
   call WriteString
L2:
; Conjunto de botones más complejo. Confunde al usuario.
   INVOKE MessageBox, NULL, ADDR infoMsj,
          ADDR leyendaC, MB_YESNOCANCEL + MB_ICONEXCLAMATION \ +
MB_DEFBUTTON2
   exit
main ENDP
END main
```

Si deseamos que nuestra ventana de cuadro de mensaje flote por encima de todas las demás ventanas en nuestro escritorio, podemos agregar la opción MB_SYSTEMMODAL a los valores del último argumento (el parámetro *uType*).

Entrada de consola

Para estos momentos, ya hemos usado los procedimientos **ReadString** y **ReadChar** de la biblioteca de enlace unas cuantas veces. Estos procedimientos se diseñaron para ser simples y directos, de manera que podamos concentrarnos en otras cuestiones. Ambos procedimientos son envolturas de ReadConsole, una función de Win32. Un procedimiento de *envoltura* oculta algunos de los detalles de otro procedimiento.

Búfer de entrada de consola La consola Win32 tiene un búfer de entrada que contiene un arreglo de registros de eventos de entrada. Cada evento de entrada, como una pulsación de tecla, movimiento del ratón o clic del botón del ratón, crea un registro de entrada en el buffer de entrada de la consola. Las funciones de entrada de alto nivel como ReadConsole filtran y procesan los datos de entrada, y devuelven sólo un flujo de caracteres.

Función ReadConsole

La función **ReadConsole** proporciona una manera conveniente de leer la entrada de texto y colocarla en un búfer. He aquí el prototipo:

```
ReadConsole PROTO,
  hConsoleInput:HANDLE,              ; manejador de entrada
  lpBuffer:PTR BYTE                  ; apuntador al búfer
  nNumberOfCharsToRead:DWORD,        ; número de caracteres a leer
  lpNumberOfCharsRead:PTR DWORD,     ; apuntador al núm. de cars. leer
  lpReserved:DWORD                   ; (no se utiliza)
```

hConsoleInput es un manejador de entrada de consola válido devuelto por la función **GetStdHandle**. El parámetro *lpBuffer* es el desplazamiento de un arreglo de caracteres. *nNumberOfCharsToRead* es un entero de 32 bits que especifica el máximo número de caracteres a leer. *lpNumberOfCharsRead* es un apuntador a una doble palabra que permite que la función llene, al regresar, una cuenta del número de caracteres colocados en el búfer. El último parámetro no se utiliza, por lo que se pasa el valor de cero.

Al llamar a ReadConsole, hay que incluir dos bytes extras en el búfer de entrada para los caracteres de fin de línea. Si deseamos que el búfer de entrada contenga una cadena con terminación nula, debemos sustituir el byte que contiene 0Dh con un byte nulo. Esto es lo que hace exactamente el procedimiento **ReadString** de Irvine32.lib.

Programa de ejemplo Para leer los caracteres introducidos por el usuario, se hace una llamada a **GetStdHandle** para obtener el manejador de entrada estándar de la consola y se hace una llamada a **ReadConsole**, usando el mismo manejador de entrada. El siguiente programa con ReadConsole demuestra esta técnica. Observamos que las llamadas a la API Win32 son compatibles con la biblioteca Irvine32, por lo que podemos llamar a DumpRegs al mismo tiempo que llamamos a las funciones Win32:

```
TITLE Lee de la consola (ReadConsole.asm)
```

```
INCLUDE Irvine32.inc

TamBuf = 80

.data

bufer BYTE TamBuf DUP(?),0,0

manejadorEntStd HANDLE ?

bytesLeidos DWORD ?

.code

main PROC

   ; obtiene el manejador para la entrada estándar

   INVOKE GetStdHandle, STD_INPUT_HANDLE

   mov manejadorEntStd,eax

   ; Espera la entrada del usuario

   INVOKE ReadConsole, manejadorEntStd, ADDR bufer,

          TamBuf  – 2, ADDR bytesLeidos,0

   ; Muestra el bufer

   mov esi,OFFSET bufer

   mov ecx,bytesLeidos

   mov ebx,TYPE bufer

   call DumpMem

   exit

main ENDP

END main
```

Si el usuario escribe “abcdefg”, el programa genera los siguientes resultados. Se insertan nueve bytes en el búfer: “abcdefg” más 0Dh y 0Ah.

Comprobación de errores

Si una función API de Windows devuelve un valor de error (como NULL), podemos llamar a la función **GetLastError** de la API para obtener más información acerca del error. Devuelve un código de error entero de 32 bits en EAX:

```
.data

idMensaje DWORD?

.code

call GetLastError

mov idMensaje,eax
```

MS-Windows tiene muchos códigos de error, por lo que tal vez sea conveniente obtener una cadena de memoria que explique el error. Para ello, hay que llamar a la función FormatMessage:

```
FormatMessage PROTO,           ; da formato a un mensaje

   dwFlags:DWORD,              ; opciones de formato

   lpSource:DWORD,             ; ubicación de definición de mensaje

   dwMsgID:DWORD,              ; identificador de mensaje

   dwLenguajeID:DWORD,         ; identificador de lenguaje

   lpBuffer:PTR,               ; apuntador de búfer que recibe la cadena

   nSize:DWORD,                ; tamaño del búfer

   va_list:DWORD,              ; apuntador a la lista de argumentos
```

Sus parámetros son algo complicados, así que tendremos que leer la documentación del SDK para obtener el panorama completo. A continuación se muestra un breve listado de los valores que nos parecen más útiles, todos son parámetros de entrada excepto *lpBuffer*, un parámetro de salida:

- *dwFlag*s, entero tipo doble palabra que guarda opciones de formato, incluyendo cómo interpretar el parámetro lpSource. Especifica cómo manejar las interrupciones de línea, así como la anchura máxima de una línea de salida con formato. Los valores recomendados son FORMAT_MESSAGE_ALLOCATE_BUFFER y FORMAT_MESSAGE_FROM_SYSTEM.
- *lpSource*, un apuntador a la ubicación de la definición del mensaje. Dada la configuración de dwFlags que recomendamos, hay que establecer lpSource a NULL (0).
- *dwMsgID*, el entero tipo doble palabra devuelto por la llamada GetLastError.
- *dwLenguajeID*, un identificador de lenguaje, Si lo establecemos a cero, el mensaje será neutral al lenguaje, o corresponderá a la configuración regional predeterminada del usuario.
- *lpBuffer (parámetro de salida)*, un apuntador a un buffer que recibe la cadena de mensaje con terminación nulo. Como utilizamos la opción FORMAT_MESSAGE_ALLOCATE_BUFFER, el búfer se asigna de manera automática.

- *nSize*, que puede usarse para especificar un búfer para guardar la cadena de mensaje. Podemos establecer este parámetro a 0 si usamos las opciones par dwFlags antes sugeridas.
- *va_list*, un apuntador a un arreglo de valores que pueden insertarse en un mensaje con formato. Como no damos formato a los mensajes de error, este parámetro puede ser NULL (0).

A continuación se muestra una llamada de ejemplo a FormatMessage:

```
.data

idMensaje DWORD ?

pMsjError DWORD ?

.code

call GetLastError

mov idMensaje,eax

INVOKE FormatMessage, FORMAT_MESSAGE_ALLOCATE_BUFER + \

    FORMAT_MESSAGE_FROM_SYSTEM, NULL, idMensaje, 0,

    ADDR pMsjError, 0, NULL
```

Después de llamar a FormatMessage, hay que llamar a LocalFree para liberar el espacio de almacenamiento por FormatMessage:

```
INVOKE LocalFree, pMsjError
```

Entrada de un solo carácter

La entrada de un solo carácter en modo de consola es un poco engañosa. MS-Windows proporciona un controlador de dispositivo para el teclado que está instalado automáticamente. Cuando se oprime una tecla, se transmite un *código de exploración* de 8 bits al puerto del teclado de la computadora. Cuando se suelta la tecla, se transmite un segundo código de exploración. MS-Windows utiliza un programa controlador de dispositivo para traducir el código de exploración en un *código de tecla virtual* de 16 bits, un valor independiente del dispositivo que define MS-Windows, y sirve para identificar el propósito de la tecla. MS-Windows crea un mensaje que contiene el código de exploración, el código de tecla virtual y además información relacionada. El mensaje se coloca en la cola de mensajes de MS-Windows, y en un momento dado encuentra su camino hacia el subproceso del programa que se encuentra en ejecución(al cual identificamos mediante el manejador de entrada de consola). Si deseamos aprender más acerca del proceso de introducción de datos mediante el teclado, podemos leer el tema *About Keyboard Input* (*Acerca de la entrada del teclado*) en la documentación del SDK de la plataforma.

Procedimientos de teclado de irvine32 La biblioteca Irvine32 tiene dos procedimientos relacionados:

- **ReadChar** espera que se teclee un carácter ASCII en el teclado y devuelve su carácter en AL.
- El procedimiento **ReadKey** realiza una comprobación sin espera del teclado. Si no hay tecla esperando en el búfer de entrada de consola, se activa la bandera Cero. Si se encuentra una tecla, la bandera Cero se borra y AL contiene cero o un código ASCII. Las mitades superiores EAX y EDX se sobrescriben.

En ReadKey, si AL contiene cero, el usuario puede haber oprimido una tecla especial (tecla función, flecha del cursor, etcétera.). El registro AH contiene el código de exploración del teclado, el cual se puede relacionar con la tabla a continuación. DX contiene el código de tecla virtual y EBX contiene información acerca de los estados de las teclas de control de teclado.

TECLAS DE FUNCIÓN

Tecla	Normal	Con mayúscula	Con Ctrl	Con Alt
F1	3B	54	5E	68
F2	3C	55	5F	69
F3	3D	56	60	6A
F4	3E	57	61	6B
F5	3F	58	62	6C
F6	40	59	63	6D
F7	41	5ª	64	6E
F8	42	5B	65	6F
F9	43	5C	66	70
F10	44	5D	67	71
F11	85	87	89	8B
F12	86	88	8A	8C

Tecla	Sola	Con tecla Ctrl
Inicio	47	77

Fin	4F	75
AvPág	49	84
Repág	51	76
ImprPant	37	72
Flecha izquierda	4B	73
Flecha derecha	4D	74
Flecha arriba	48	8D
Flecha abajo	50	91
Ins	52	92
Supr	53	93
Retroceso tab	0F	94
+ gris	4E	90
- gris	4A	8E

En la siguiente tabla se podrá ver una lista de valores de las teclas de control. Después de llamar a ReadKey, podemos usar la instrucción TEST para comprobar los diversos valores de las teclas. La implementación de ReadKey es bastante extensa.

Valores de estado de las teclas de control del teclado

Valor	Significado
CAPSLOCK_ON	La luz CAPS LOCK está encendida
ENHANCED_KEY	La tecla es mejorada
LEFT_ALT_PRESSED	Se oprimió la tecla ALT izquierda
LEFT_CTRL_PRESSED	Se oprimió la tecla CTRL izquierda
NUMLOCK_ON	La luz NUM LOCK está encendida
RIGHT_ALT_PRESSED	Se oprimió la tecla ALT derecha
RIGHT_CTRL_PRESSED	Se oprimió la tecla CTRL derecha
SCROLLOCK_ON	La luz SCROLL LOCK está encendida
SHIFT_PRESSED	Se oprimió la tecla MAYÚS

Programa de prueba ReadKey El siguiente programa prueba el procedimiento ReadKey, para lo cual espera a que el usuario oprima una tecla y después informa si la tecla Bloq Mayúscula está oprimida o no. Como mencionamos anteriormente, se debe incluir un factor de retraso al llamar a ReadKey, para dar tiempo a que MS-Windows procese su ciclo de mensajes:

```
TITLE prueba de ReadKey (PruebaReadKey.asm)

INCLUDE Irvine32.inc

INCLUDE Macros.inc

.code

main PROC

L1: mov eax,10

   call Delay                  ; retraso para el procesamiento de
mensajes

   ReadKey                     ; espera una pulsación de tecla

   jz L1

   test ebx,CAPSLOCK_ON

   jz L2

   mWrite <"Bloq Mayus esta ACTIVADA",0dh,0ah>

   jmp L3

L2: mWrite <"Bloq Mayus esta DESACTIVADA",0dh,0ah>

L3: exit

main ENDP

END main
```

Obtención del estado del teclado

Podemos probar el estado de las teclas individuales del teclado, para averiguar cuales están oprimidas en ese momento. Para ello hay que llamar a la función **GetKeyState** de la API:

```
GetKeyState PROTO, nTeclaVirt:DWORD
```

Esta función recibe un valor de tecla virtual, como los que se muestra en la siguiente tabla. Nuestro programa debe probar el valor devuelto en EAX. Según lo indica la misma tabla.

Prueba de las teclas con GetKeyState.

Tecla	Símbolo de tecla virtual	Bit que se evalúa en EAX
Bloq Núm	VK_NUMLOCK	0
Bloq Despl	VK_SCROLL	0
Mayús Izq	VK_LSHIFT	15
Mayús Der	VK_RSHIFT	15
Ctrl Izq	VK_LCONTROL	15
Ctrl Der	VK_RCONTROL	15
Menú Izq	VK_LMENU	15
Menú Der	VK_RMENU	15

El siguiente programa de ejemplo demuestra la función GetKeyState al comprobar los estados de las teclas Bloq Núm y Depl Izq:

```
TITLE Teclas alternantes del teclado     (Teclado.asm)

INCLUDE Irvine32.inc

INCLUDE Macros.inc

; GetKeyState activa el bit 0 en EAX si una tecla

; alternante está activada (Bloq Mayús, Bloq Núm, Bloq Despl.

; Activa el bit 15 en EAX si otra de las teclas especificadas

; está oprimida.

.code

main PROC

   INVOKE GetKeyState, VK_NUMLOCK

   test al,1

   .IF !Zero?

      mWrite <”La tecla Bloq Num esta ACTIVADA”,0dh,0ah>

   .ENDIF
```

```
   INVOKE GetKeyState, VK_LSHIFT
   test al,80h
   .IF !Zero?
      mWrite <"La tecla Mayus Izq esta oprimida en este momento",0dh,0ah>
   .ENDIF
   exit
main ENDP
END main
```

Salida de consola

En los primeros capítulos tratamos de hacer la salida de la consola lo más sencilla posible. El procedimiento **WriteString** en la biblioteca de enlace Irvine32 sólo requería un argumento, el desplazamiento de una cadena en EDX. WriteString es en realidad es una envoltura a una llamada más detallada a una función Win32 llamada **WriteConsole**.

No obstante, aprenderemos a realizar llamadas directas a las funciones Win32 como **WriteConsole** y **WriteConsoleOutputCharacter**. Las llamadas directas implican un conocimiento más detallado, pero también nos ofrecen más flexibilidad que los procedimientos de la biblioteca Irvine32.

Estructuras de datos

Varias funciones de la consola Win32 utilizan estructuras de datos predefinidas, Incluyendo COORD y SMALL_RECT. la estructura COORD guarda las coordenadas de una celda de caracteres en el búfer de pantalla de consola. El origen del sistema de coordenadas (0,0) está en la celda superior izquierda:

```
COORD STRUCT
   X WORD ?
   Y WORD ?
COORD ENDS
```

La estructura SMALL_RECT guarda las esquinas superior izquierda e inferior derecha de un rectángulo. Especifica las celdas de caracteres de búfer de pantalla en la ventana de consola:

```
SMALL_RECT SRTRUC
   Left WORD ?
   Top WORD ?
```

```
   Right WORD ?

   Botton WORD ?

SMALL_RECT ENDS
```

Función WriteConsole

La función **WriteConsole** escribe una cadena en la ventana de consola, en la posición actual del cursor, y deja el cursor justo después del último carácter escrito. Actúa en base a los caracteres de control ASCII estándar como *tabulador*, *retorno de carro* y *avance de página*. La cadena no debe tener terminación nula. He aquí el prototipo de la función:

```
WriteConsole PROTO,

   hConsoleOutput:HANDLE,

   lpBuffer:PTR BYTE,

   nNumberOfCharsToWrite:DWORD,

   lpNumberOfCharsWritten:PTR DWORD,

   lpReserved:DWORD
```

hConsoleOutput es el manejador del flujo de salida de la consola; *lpBuffer* es un apuntador al arreglo de caracteres que se desea escribir; *nNumberOfCharsToWrite* guarda la longitud del arreglo; *lpNumberOfCharsWritten* apunta a un entero al que se asigna el número de bytes que se escriben cuando la función regresa. El último parámetro no se utiliza, por lo cual se establece en cero.

Programa de ejemplo 1: Consola1

El siguiente programa, *Consola1.asm*, demuestra las funciones **GetStdHandle**, **ExitProcess** y **WriteConsole**, al escribir una cadena en la ventana de consola:

```
TITLE Ejemplo de consola Win32 #1  (consola.asm)

; Este programa llama a las siguientes funciones de Consola Win32:

; GetStdHandle, ExitProcess, WriteConsole

INCLUDE Irvine32.inc

.data

fin1 EQU <0dh,0ah>             ; secuencia de fin de línea

mensaje LABEL BYTE

   BYTE "Este programa es una simple demostración de"
```

```
   BYTE “la salida en modo de consola, usa las funciones”
   BYTE “GetStdHandle y WriteConsole.”,fin1
tamanioMensaje DWORD ($-mensaje)
manejadorConsola HANDLE 0      ; manejador para el dispositivo de salida
estándar
bytesEscritos DWORD ?          ; número de bytes escritos
.code
main PROC
   ; Obtiene el manejador de salida de consola:
      INVOKE GetStdHandle, STD_OUTPUT_HANDLE
      mov manejadorConsola,eax

   ; Escribe una cadena en la consola:
      INVOKE WriteConsole,
         manejadorConsola,     ; manejador de salida de consola
         ADDR mensaje,         ; apuntador a la cadena
         tamanioMensaje,       ; longitud de la cadena
         ADDR bytesEscritos    ; devuelve el núm de bytes escritos
         0                     ; no se utiliza
      INVOKE ExitProcess,0
main ENDP
END main
```

El programa produce el siguiente resultado:

```
Este programa es una simple demostración de la salida en modo consola, usa las funciones
GetStdHandle y WriteConsole.
```

Función WriteConsoleOutputCharacter

La función **WriteConsoleOutputCharacter** copia un arreglo de caracteres a celdas consecutivas del búfer de pantalla de consola, empezando en una ubicación especificada. He aquí el prototipo:

```
WriteConsoleOutputCharacter PROTO,
   hConsoleOutput:HANDLE,      ; manejador de salida de consola
   lpCharacter:PTR BYTE,       ; apuntador al búfer
   nLength:DWORD,              ; tamaño del búfer
   dwWriteCoord:COORD,         ; coordenadas de la primera celda
   lpNumberOfCharsWritten: PTR DWORD     ; cuenta de salida
```

Si el texto llega al final de una línea, pasa a la siguiente. Los valores de los atributos en el búfer de pantalla no cambian. Si la función no puede escribir los caracteres, devuelve cero. Los códigos de control ASCII como *tabulador*, *retorno de carro* y *avance de línea* se ignoran.

Lectura y escritura de archivos

Función CreateFile

La función **CreateFile** crea un nuevo archivo o abre uno existente. Si tiene éxito, devuelve en manejador para el archivo abierto; en caso contrario, devuelve una constante especial llamada INVALID_HANDLE_VALUE. He aquí el prototipo:

```
CreateFile PROTO,                   ; crea un nuevo archivo
  lpFileName:PTR BYTE,              ; apuntador al nombre del archivo
  dwDesiredAccess:DWORD,            ; modo de acceso
  dwShareMode:DWORD,                ; modo de compartición
  lpSecurityAttributes:DWORD,       ; apuntador a los archivos de seguridad
  dwCreationDisposition:DWORD,      ; opciones de creación de archivo
  dwFlagsAndAttributes:DWORD,       ; atributos del archivo
  hTemplateFile:DWORD               ; manejador al archivo de plantilla
```

Los parámetros se describen en la siguiente tabla. El valor de retorno es cero si la función falla.

Parámetros de CreateFile

Parámetro	Descripción
lpFileName	Apunta a una cadena con terminación nula, que contiene un nombre de archivo parcial o completamente calificado (unidad: \ruta\nombrearchivo)
dwDesiredAccess	Especifica la forma en que se accederá al archivo (lectura o escritura)
dwShareMode	Controla la capacidad para que varios programas accedan al archivo, mientras está abierto
lpSecurityAttributes	Apunta a una estructura de seguridad que controla los derechos de seguridad
dwCreationDisposition	Especifica la que acción realizar cuando un archivo existe o no
dwFlagsAndAttributes	Guarda las banderas de bits que especifican los atributos de un archivo, como archivo, cifrado, oculto, normal, de sistema y temporal
hTemplateFile	Contiene un manejador opcional para un archivo de plantilla que suministra los atributos de archivo y los atributos extendidos para el archivo que se va a crear; cuando no se utiliza este parámetro, se establece a cero

dwDesiredAccess El parámetro dwDesiredAccess nos permite especificar el acceso de lectura, de escritura, de lectura/escritura, o el acceso de consulta de un dispositivo para el archivo. Podemos seleccionar uno de los valores de la siguiente tabla, o de un conjunto extenso de valores específicos de bandera que no se presentan aquí (buscar *CreateFile* en la documentación SDK de la plataforma).

Opciones del parámetro dwDesiredAccess.

Valor	Significado
0	Especifica el acceso de consulta de dispositivos para el objeto. Una aplicación puede consultar los atributos de un dispositivo sin acceder a este, o puede comprobar la existencia de un archivo
GENERIC_READ	Especifica el acceso de lectura al objeto. Los datos puede leerse del archivo, y el apuntador del archivo se puede mover. Se combina con GENERIC_WRITE para acceso de lectura/escritura
GENERIC_WRITE	Especifica el acceso de escritura para el objeto. Los datos pueden escribirse en el archivo, y el apuntador del archivo puede moverse. Se combina con GENERIC_READ para acceso de lectura/escritura

CreationDisposition El parámetro *dwCreationDisposition* especifica la acción a realizar en los archivos que existen, y la acción a realizar cuando los archivos no existen. Podemos seleccionar uno de los valores mostrados en la siguiente tabla.

Valor	Significado
CREATE_NEW	Crea un nuevo archivo. Requiere establecer el parámetro dwDesiredAccess a GENERIC_WRITE. La función falla si el archivo ya existe
CREATE_ALWAYS	Crea un archivo. Si ya existe, la función sobrescribe el archivo, borra los atributos existentes y combina los atributos del archivo y las banderas especificadas por el parámetro *attributes* con la constante predefinida FILE_ATTRIBUTE_ARCHIVE. Requiere establecer el parámetro de dwDesiredAccess a GENERIC_WRITE
OPEN_EXISTING	Abre el archivo. La función falla si el archivo no existe. Podemos usarlo para leer desde y/o escribir en el archivo
OPEN_ALWAYS	Abre el archivo si es que existe. Si no, la función crea el archivo como si *CreationDisposition* fuera CREATE_NEW
TRUNCATE_EXISTING	Abre el archivo. Una vez abierto, se trunca a un tamaño de cero. Requiere establecer el parámetro *dwDesiredAccess* a GENERIC_WRITE. Esta función falla si el archivo no existe

La siguiente tabla presenta los valores que se utilizan con más frecuencia y que están permitidos en el parámetro *dwFlagsAndAttributes*. Cualquier combinación de los atributos es aceptable, con la excepción de que todos los demás atributos redefinen a FILE_ATTRIBUTE_NORMAL. Los valores se asignan en potencias de 2, por lo que podemos usar el operador OR en tiempo de ensamblado, o el operador + para combinarlos en un solo argumento:

```
FILE_ATTRIBUTE_HIDDEN OR FILE_ATTRIBUTE_READONLY
```

```
FILE_ATTRIBYTE_HIDDEN + FILE_ATTRIBUTE_READONLY
```

Valores seleccionados de FlagsAndAttributes.

Atributo	Significado
FILE_ATTRIBUTE_ARCHIVE	El archivo debe archivarse. Las aplicaciones utilizan este atributa para marcar los archivos para respaldo o eliminación

FILE_ATTRIBUTE_HIDDEN	El archivo está oculto. No debe incluirse en un listado ordinario de directorios.
FILE_ATTRIBUTE_NORMAL	El archivo no tiene atributos establecidos. Este atributo es válido solo si se utiliza solo
FILE_ATTRIBUTE_READONLY	El archivo es de sólo lectura. Las aplicaciones pueden leer el archivo, pero no pueden escribir en él ni eliminarlo
FILE_ATTRIBUTE_TEMPORARY	El archivo se está utilizando para almacenamiento temporal

Ejemplos Los siguientes ejemplos son sólo fines ilustrativos, para mostrar cómo se pueden crear y abrir archivos. Podemos consultar la documentación de Microsoft MSDN sobre **CreateFile** para aprender acerca de las muchas opciones disponibles:

- Abrir un archivo existente para lectura(entrada):

```
INVOKE CreateFile,
   ADDR nombrearchivo,   ; apuntador al nombre de archivo
   GENERIC_READ,         ; lee del archivo
   DO_NOT_SHARE,         ; modo de compartición
   NULL,                 ; apuntador a los atributos de seguridad
   OPEN_EXISTING,        ; abre un archivo existente
   FILE_ATTRIBUTE_NORMAL,    ; atributo normal de archivo
   0                     ; no se utiliza
```

- Abrir un archivo existente para escritura (salida). Una vez abierto el archivo, podríamos escribir sobre datos existentes o adjuntarle nuevos datos, desplazando el puntero del archivo hasta el final (SetFilePointer):

```
INVOKE CreateFile,
   ADDR nombrearchivo,
   GENERIC_WRITE,        ; escribe en el archivo
   NULL,
   OPEN_EXISTING,        ; el archivo debe existir
   FILE_ATTRIBUTE_NORMAL,
   0
```

- Crear un nuevo archivo con atributos normales, borrando cualquier archivo existente que tenga el mismo nombre:

```
INVOKE CreateFile,
   ADDR nombrearchivo,
   GENERIC_WRITE         ; escribe en el archivo
   DO_NOT_SHARE,
   NULL,
   CREATE_ALWAYS,        ; sobrescribe el archivo existente
```

```
   FILE_ATTRIBUTE_NORMAL,
   0
```

- Crear un nuevo archivo si éste no existe ya; en caso contrario, abrir el archivo existente en modo de salida:

```
INVOKE CreateFile,
   ADDR nombrearchivo,
   GENERIC_WRITE         ; escribe en el archivo
   DO_NOT_SHARE,
   NULL,
   CREATE_NEW            ; no borra el archivo existente
   FILE_ATTRIBUTE_NORMAL.
   0
```

La constantes llamadas DO_NOT_SHARE y NULL se definen en el archivo de inclusión SmallWin.inc.

Función CloseHandle

La función **CloseHandle** cierra un manejador de objetos abiertos. Su prototipo es:

```
CloseHandle PROTO,

   hObject:HANDLE              ; manejador para el objeto
```

Podemos usar CloseHandle para cerrar un manejador de un archivo que se encuentre abierto. El valor de retorno es cero si la función falla.

Función ReadFile

La función **ReadFile** lee texto de un archivo de entrada. He aquí el prototipo:

```
ReadFile PROTO,

   hFile:HANDLE,               ; manejador de entrada

   lpBuffer:PTR BYTE,          ; apuntador al búfer

   nNumberOfBytesToRead:DWORD, ; número de bytes a leer

   lpNumberOfBytesRead:PTR DWORD,  ; bytes leídos

   lpOverlapped:PTR DWORD      ; apuntador a info asíncrona
```

El parámetro *hFile* es un manejador de archivo abierto devuelto por **CreateFile**; *lpBuffer* apunta a un búfer que recibe los datos leídos del archivo; *nNumberOfBytesToRead* especifica el número máximo de bytes a leer del archivo; *lpNumberOfBytesRead* apunta a un entero que indica el número de bytes que se leyeron al momento en que regresó la función; *lpOverlapped*

debe establecerse en NULL (0) para la lectura síncrona (la que utilizamos). El valor de retorno es cero si la función falla.

Si se llama más de una vez en el mismo manejador de archivo abierto, ReadFile recuerda dónde terminó de leer la última vez y lee de ese punto en adelante. En otras palabras, mantiene un apuntador interno a la posición actual del archivo. ReadFile también puede ejecutarse en modo asíncrono, lo cual significa que el programa que hace la llamada no espera a que termine la operación de lectura.

Función WriteFile

La función **WriteFile** escribe datos en un archivo, usando un manejador de salida. El manejador puede ser el manejador de búfer de pantalla, o uno asignado a un archivo de texto. La función empieza a escribir datos al archivo, en la posición indicada por el apuntador de posición interna del archivo. Una vez que se completa la operación de escritura, el apuntador de posición del archivo se ajusta según el número de bytes que se escribieron. He aquí el prototipo de función:

```
WriteFile PROTO,

   hFile:HANDLE,               ; manejador de salida

   lpBuffer:PTR BYTE,          ; apuntador al búfer

   nNumberOfBytesToWrite:DWORD,    ; tamaño del búfer

   lpNumberOfBytesWritten:PTR DWORD,     ; número de bytes escritos

   lpOverlapped:PTR DWORD,     ; apuntador a info asíncrona
```

hFile es un manejador a un archivo que se abrió anteriormente; *lpBuffer* apunta a un búfer que guarda los datos escritos en el archivo; *nNumberOfBytesToWrite* especifica cuántos bytes se van a escribir en el archivo; *lpNumberOfBytesWritten* apunta a un entero que especifica el número de bytes que se escribieron después de la ejecución de la función; *lpOverlapped* debe establecerse en NULL para la operación síncrona. El valor de retorno es cero si la función falla.

Función SetFilePointer

La función **SetFilePointer** mueve el apuntador de posición de un archivo abierto. Esta función puede utilizarse para adjuntar datos a un archivo, o para realizar el procesamiento de registros de acceso aleatorio:

```
SetFilePointer PROTO,

   hFile:HANDLE,               ; manejador de archivo

   lDistanceToMove:SDWORD,     ; bytes para mover el apuntador

   lpDistanceToMoveHigh:PTR SDWORD,; apuntador de bytes a mover, sup
```

```
   dwMoveMethod:DWORD          ; punto inicial
```

El valor de retorno es cero si la función falla. *dwMoveMethod* especifica el punto inicial para mover el apuntador de archivos, el cual se selecciona de tres símbolos predefinidos: FILE_BEGIN, FILE_CURRENT y FILE_END. La distancia en sí es un valor entero de 64 bits con signo, dividido en dos partes:

- lpDistanceToMove: los 32 bits inferiores.
- pDistanceToMoveHigh: un apuntador a una variable que contiene los 32 bits superiores.

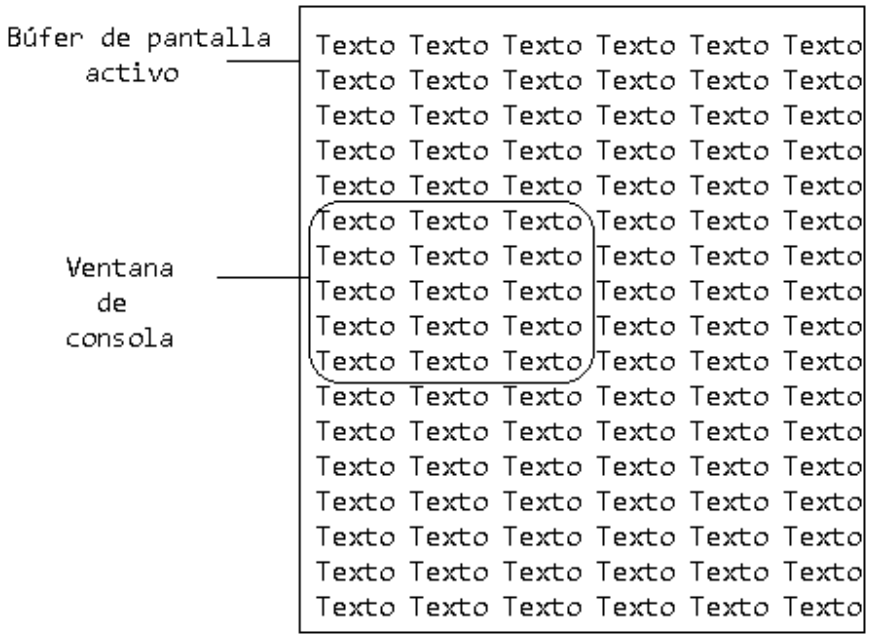
Si *lpDistanceToMoveHigh* es nulo, sólo se utiliza el valor en lpDistanceToMove para mover el apuntador del archivo. Por ejemplo, el siguiente código se prepara para adjuntar datos al final de un archivo:

```
INVOKE SetFilePointer,
   manejadorArchivo,           ; manejador del archivo
   0,                          ; distancia inferior
   0,                          ; distancia superior
   FILE_END                    ; método para mover
```

Manipulación de ventanas de consola

La API Win32 proporciona un control considerable sobre la ventana de consola y su búfer. La siguiente imagen muestra que búfer de pantalla puede ser mayor que el número de líneas que se muestran en un momento dado en la ventana de consola. Esta ventana actúa como “mira”, mostrando parte del búfer.

El búfer de pantalla y la ventana de consola.



Varias funciones afectan a la ventana de consola y su posición relativa al búfer de pantalla:

- **SetConsoleWindowInfo** establece el tamaño y la posición de la ventana de consola, relativa al búfer de pantalla.
- **GetConsoleScreenBufferInfo** devuelve (entre otras cosas) las coordenadas rectangulares de la ventana de consola relativa al búfer de pantalla.
- **SetConsoleCursorPosition** establece la posición del cursor en cualquier ubicación dentro del búfer de pantalla; si esa área no es visible, la ventana de consola se desplaza para que el cursor sea visible.
- **ScrollConsoleScreenBuffer** mueve parte o todo el texto dentro del búfer de pantalla, el cual puede afectar al texto mostrado en la ventana de consola.

SetConsoleTitle

La función **SetConsoleTitle** nos permite cambiar el título de la ventana de consola. A continuación se muestra un ejemplo:

```
.data

cadTitulo BYTE “Titulo de la consola”,0

.code

INVOKE SetConsoleTitle, ADDR cadTitulo
```

GetConsoleScreenBufferInfo

La función **GetConsoleScreenBufferInfo** devuelve la información acerca del estado actual de la ventana de consola. Tiene dos parámetros: un manejador para la pantalla de consola, y un apuntador a una estructura que la función llena:

```
GetConsoleScreenBufferInfo PROTO

   hConsoleOutput:HANDLE,

   lpConsoleScreenBufferInfo:PTR CONSOLE_SCREEN_BUFFER_INFO
```

A continuación se muestra la estructura CONSOLE_SCREEN_BUFFER_INFO:

```
CONSOLE_SCREEN_BUFFER_INFO STRUCT

   dwSize COORD <>

   dwCursorPosition COORD <>

   wAttributes WORD ?

   srWindow SMALL_RECT <>

   dwMaximumWindowSize COORD <>

CONSOLE_SCREEN_BUFFER_INFO ENDS
```

dwSize devuelve el tamaño del búfer de pantalla, en columnas y filas de caracteres. *dwCursorPosition* devuelve la ubicación del cursor. Ambos campos son coordenadas COORD. *wAttributes* devuelve los colores de texto y de fondo de los caracteres que funciones como **WriteConsole** y **WriteFile** escriben en consola. *srWindow* devuelve las coordenadas de la ventana de consola relativa al búfer de pantalla. *dwMaximumWindowSize* devuelve el tamaño máximo de la ventana de consola, con base en el tamaño actual del búfer de pantalla, de la fuente y de la pantalla de video. A continuación se muestra una llamada de ejemplo a la función:

```
.data

infoConsola CONSOLE_SCREEN_BUFFER_INFO <>

manejadorSalida HANDLE ?

.code

INVOKE GetConsoleScreenBufferInfo, manejadorSalida, ADDR infoConsola
```

Función SetConsoleWindowInfo

La función **SetConsoleWindowInfo** nos permite establecer el tamaño y posición de la ventana de consola, relativa a su búfer de pantalla. He aquí su prototipo de función:

```
SetConsoleWindowInfo PROTO,
```

```
   hConsoleOutPut:HANDLE,          ; manejador de búfer de pantalla
   bAbsolute:DWORD,                ; tipo de coordenada
   lpConsoleWindow:PTR SMALL_RECT  ; apuntador a rectángulo de ventana
```

bAbsolute indica la forma en que se van a utilizar las coordenadas en la estructura a la que apunta *lpConsoleWindow*. Si *bAbsolute* es verdadera, las coordenadas especifican las nuevas esquinas superior izquierda e inferior derecha de la ventana de consola. Si *bAbsolute* es falsa, las coordenadas se agregan a las coordenadas de ventana actuales.

El siguiente programa *Despl.asm* escribe 50 líneas de texto en el búfer de pantalla. Después cambia de tamaño y posición la ventana de consola, logrando desplazar el texto hacia atrás. Utiliza la función **SetConsoleWindowInfo**:

```
TITLE Desplazamiento de la ventana de consola (Despl.asm)
INCLUDE Irvine32.inc
.data
mensaje BYTE ": Esta línea de texto se escribió"
        BYTE "en el búfer de pantalla",0dh,0ah
tamMensaje DWORD ($-mensaje)
manejadorSalida HANDLE 0       ; manejador de salida estándar
bytesEscritos DWORD ?          ; número de bytes escritos
numLinea DWORD 0
rectVentana SMALL_RECT <0,0,60,11> ; izquierda, arriba, derecha, abajo
.code
main PROC
   INVOKE GetStdHandle, STD_OUTPUT_HANDLE
   mov manejadorSalida,eax
.REPEAT
   mov eax,numLinea
   call WriteDec               ; muestra cada número de línea
   INVOKE WriteConsole,
```

```
    manejadorSalida            ; manejador de salida de consola
    ADDR mensaje,              ; apuntador a la cadena
    tamMensaje,                ; longitud de la cadena
    ADDR bytesEscritos,        ; devuelve el número de bytes escritos
    0                          ; no se utiliza
   inc numLinea                ; siguiente número de línea
.UNTIL numLinea > 50
; cambia de tamaño y reposiciona la ventana de consola relativa al
; búfer de pantalla.
   INVOKE SetConsoleWindowInfo,
    manejadorSalida,
    TRUE,
    ADDR rectVentana
   call ReadChar               ; espera una tecla
   call Clrscr                 ; borra el búfer de pantalla
   call ReadChar               ; espera una segunda tecla
   INVOKE ExitProcess,0
main ENDP
END main
```

Función SetConsoleScreenBufferSize

La función SetConsoleScreenBufferSize nos permite establecer el tamaño del búfer de pantalla a X columnas por Y filas. He aquí el prototipo:

```
SetConsoleScreenBufferSize PROTO,
   hConsoleOutput:HANDLE,      ; manejador para el búfer de pantalla
   dwSize:COORD                ; nuevo tamaño de búfer de pantalla
```

Control del cursor

La API Win32 proporciona funciones para establecer el tamaño del cursor, la visibilidad y la ubicación de la pantalla. Una estructura de datos importantes, relacionada con estas funciones, es CONSOLE_CURSOR_INFO, la cual contiene información acerca del tamaño y la visibilidad del cursor de la consola:

```
CONSOLE_CURSOR_INFO STRUCT

   dwSize DWORD ?

   bVisible DWORD ?

CONSOLE_CURSOR_INFO ENDS
```

dwSize es el porcentaje (de 1 a 100) de la celda de caracteres que llena el cursor. bVisible es igual a TRUE (1) si el cursor está visible.

Función GetConsoleCursorInfo

La función **GetConsoleCursorInfo** devuelve el tamaño y la visibilidad del cursor de la consola. Recibe un apuntador a una estructura CONSOLE_CURSOR_INFO

```
GetConsoleCursorInfo PROTO,

   hConsoleOutPut:HANDLE,

   lpConsoleCursorInfo:PTR CONSOLE_CURSOR_INFO
```

De manera predeterminada, el tamaño del cursor es 25, lo cual indica que éste llena un 25% la celda de caracteres.

Función SetConsoleCursorInfo

La función SetConsoleCursorInfo establece el tamaño y la visibilidad del cursor. Recibe un apuntador a una estructura CONSOLE_CURSOR_INFO:

```
SetConsoleCursorInfo PROTO,

   hConsoleOutPut:HANDLE,

   lpConsoleCursorInfo:PTR CONSOLE_CURSOR_INFO
```

SetConsoleCursorPosition

La función **SetConsoleCursorPosition** establece la posición X, Y del cursor. Recibe una estructura COORD y el manejador de salida de consola:

```
SetConsoleCursorPosition PROTO,

   hConsoleOutput:DWORD,       ; manejador de modo de entrada

   dwCursorPosition:COORD      ; coordenadas X, Y de la pantalla
```

Control de color de texto

Hay dos formas de controla el color del texto en una ventana de consola. Podemos cambiar de texto actual llamando a **SetConsoleTextAttribute**, el cual afecta toda la salida subsiguiente de texto en la consola. De manera alternativa, podemos establecer los atributos de celdas específicas, llamando a **WriteConsoleOutputAttribute**. La función GetConsoleScreenBufferInfo devuelve los colores actuales de la pantalla, junto con otra información relacionada con la consola.

Función SetConsoleTextAttribute

La función **SetConsoleTextAttribute** nos permite establecer los colores de texto y de fondo para toda la salida subsiguiente de texto en la ventana de consola. He aquí su prototipo:

```
SetConsoleTextAttribute PROTO,
   hConsoleOutput:HANDLE,      ; manejador de salida de consola
   wAttributes:WORD            ; atributo de color
```

El valor de color se almacena en el byte de menor orden del parámetro *wAttributes*, Los colores se crean usando el mismo método que para el BIOS de VIDEO, el cual se mostrará en este resumen.

Función WriteConsoleOutputAttribute

La función **WriteConsoleOutputAttribute** copia un arreglo de valores de atributos a celdas consecutivas del búfer de pantalla, empezando en una ubicación especificada. He aquí el prototipo:

```
WriteConsoleOutputAttribute PROTO,
   hConsoleOutput:DWORD,       ; manejador de salida
   lpAttribute:PTR WORD,       ; atributos de escritura
   nLength:DWORD               ; número de celdas
   dwWriteCoord:COORD,         ; coordenadas de la primera celda
   lpNumberOfAttrsWritten:PTR DWORD; cuenta de salida
```

lpAttribute apunta a un arreglo de atributos, en los que el byte de menor orden de cada uno de ellos contiene el color, *nLength* es la longitud del arreglo; *dwWriteCoord* es la celda inicial de la pantalla que recibe los atributos; y *lpNumberOfAttrsWritten* apunta a una variable que guarda el número de celdas escritas.

Ejemplo: programa EscribirColores

Para demostrar el uso de los colores y los atributos, el programa *EscribirColores.asm* crea un arreglo de caracteres y un arreglo de atributos, uno para cada carácter. Llama a **WriteConsoleOutputAttribute** para copiar los atributos en el búfer de pantalla y a **WriteConsoleOutputCharacter** para copiar los caracteres a las mismas celdas del búfer de pantalla:

```
TITLE Escritura de colores de texto(EscribirColores.asm)

INCLUDE Irvine32.inc

.data

manejadorSalida HANDLE ?

celdasEscritas DWORD ?

posXY COORD <10,2>

; Arreglo de códigos de caracteres:

bufer BYTE 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

      BYTE 16,17,18,19,29

TamBuf DWORD ($ - bufer)

; Arreglo de atributos:

atributos WORD 0Fh,0Eh,0Dh,0Ch,0Bh,0Ah,9,8,7,6

          WORD 5,4,3,2,1,0F0h,0E0h,0D0h,0C0h,0B0h

.code

main PROC

; Obtiene el manejador de salida estándar de la consola:

   INVOKE GetStdHandle, STD_OUTPUT_HANDLE

      mov manejadorSalida,eax

; Establece los colores de (10,2) a (30,2):

   INVOKE WriteConsoleOutputAttribute,

      manejadorSalida, ADDR atributos,

      TamBuf, posXY, ADDR celdasEscritas

; Escribe los códigos de los caracteres del 1 al 20:
```

```
   INVOKE WriteConsoleOutputCharacter,
      manejadorSalida, ADDR bufer, TamBuf,
      posXY, ADDR celdasEscritas
INVOKE ExitProcess,0           ; terminar el programa
main ENDP
END main
```

Funciones de hora y fecha

La API Win32 proporciona una selección bastante extensa de funciones de hora y fecha. Para empezar podemos obtener y establecer la fecha y hora actuales. Sólo veremos un pequeño subconjunto de las funciones, pero podemos consultar la documentación del SDK de la plataforma las funciones Win32 que se presentan en la siguiente tabla:

Funciones de fecha y hora de Win32

Función	Descripción
CompareFileTime	Compara dos horas de archivos de 64 bits
DosDateTimeToFileTime	Convierte valores de fecha y hora de MS-DOS en una hora de archivo de 64 bits
FileTimeToDosDateTime	Convierte la hora de un archivo de 64 bits en valores de fecha y hora de MS-DOS
FileTimeToLocalFileTime	Convierte la hora de un archivo UTC (hora *coordinada universal*) en una hora de archivo local
FileTimeToSystemTime	Convierte la hora de un archivo de 64 bits en formato de hora del sistema
GetFileTime	Obtiene la fecha y hora de creación de un archivo, su último acceso y su última modificación
GetLocalTime	Obtiene la fecha y hora actuales
GetSystemTime	Obtiene la fecha y hora actuales del sistema, en formato UTC
GetSystemTimeAdjustment	Determina si el sistema está aplicando ajustes de hora

	periódico al reloj de hora del día
GetSystemTimeAsFileTime	Obtiene la fecha y hora actuales del sistema, en formato UTC
GetTickCount	Obtiene el número de milisegundos transcurridos desde que se inició el sistema
GetTimeZoneInformation	Obtiene los parámetros de zona horaria actuales
LocalFileTimeToFileTime	Convierte una hora de archivo local en una hora de archivo basada en UTC
SetFileTime	Establece la fecha y hora de creación de un archivo, su último acceso o su última modificación
SetLocalTime	Establece la hora y fecha local actuales
SetSystemTime	Establece la hora y fecha actuales del sistema
SetSystemTimeAjustment	Habilita o deshabilita los ajustes periódicos de hora para el reloj de la hora del día del sistema
SetTimeZoneInformation	Establece los parámetros actuales de la zona horaria
SystemTimeToFileTime	Convierte una hora del sistema en una hora de archivo
SystemTimeToTzSpecificLocalTime	Convierte una hora UTC en una hora local correspondiente a la zona horaria

Estructura SYSTEMTIME La estructura SYSTEMTIME es utilizada por las funciones de la API de Windows relacionadas con la fecha y hora:

```
SYSTEM STRUCT
   wYear WORD ?              ; año (4 dígitos)
   wMonth WORD ?             ; mes (1 – 12)
   wDayOfWeek WORD ?         ; día de la semana (0 – 6)
   wDay WORD ?               ;día (1 – 31)
   wHour WORD ?              ; horas (0- 23)
   wMinute WORD ?            ; minutos (0 – 59)
```

```
   wSecond WORD ?              ; segundos (0- 59)

   wMiliseconds WORD ?         ; milisegundos (0-999)

SYSTEMTIME ENDS
```

El valor *wDayOfWeek* empieza con Domingo = 0, Lunes = 1, y así sucesivamente. El valor en *wMiliseconds* no es exacto, ya que el sistema puede actualizar la hora en forma periódica, sincronizándose con una fuente.

GetLocalTime y SetLocalTime

La función **GetLocalTime** devuelve la fecha y hora actuales del día, de acuerdo con el reloj del sistema. La hora se ajusta según la zona horaria local. Al llamar a esta función se le debe pasar un apuntador a una estructura SYSTEMTIME

```
GetLocalTime PROTO,

   lpSystemTime: PTR SYSTEMTIME
```

La siguiente es una llamada de ejemplo a la función GetLocalTime:

```
.data

horaSist SYSTEMTIME <>

.code

INVOKE GetLocalTime, ADDR horaSist
```

La función **SetLocalTime** establece la fecha y hora local actuales. Al llamarla, hay que pasarle un apuntador a una estructura SYSTEMTIME que contiene la fecha y hora deseadas:

```
SetLocalTime PROTO,

   lpSystemTime:PTR SYSTEMTIME
```

Si la función se ejecuta con éxito, devuelve un entero distinto de cero; si falla, devuelve cero.

Función GetTickCount

La función **GetTickCount** devuelve el número de milisegundos transcurridos desde que se inició el sistema:

```
GetTickCount PROTO             ; valor de retorno en EAX
```

Como el valor devuelto es una doble palabra, la hora se revertirá a cero si el sistema se ejecuta en forma continúa durante 49.7 días. Podemos usar esta función para verificar el

tiempo transcurrido en un ciclo, y salir del ciclo cuando se haya llegado a un cierto límite de tiempo.

El siguiente programa *Cronometro.asm* mide el tiempo transcurrido entre dos llamadas GetTickCount. Verifica que la cuenta del cronómetro no se haya regresado a cero (más de 49.7 días). Podría utilizarse un código similar en una variedad de programas:

```
TITLE Calcula el tiempo transcurrido     (Cronometro.asm)

; Demuestra un temporizador de cronómetro simple, usando

; la función GetTickCount de Win32.

INCLUDE Irvine32.inc

INCLUDE macros.inc

.data

tiempoInicial DWORD ?

.code

main PROC

   INVOKE GetTickCount         ; obtiene cuenta de tics inicial

   mov tiempoInicial,eax       ; la guarda

   ; Crea un ciclo de cálculo inútil.

   mov ecx,10000100h

L1: imul ebx

   imul ebx

   imul ebx

   loop L1

   INVOKE GetTickCount         ; obtiene nueva cuenta de tics

   cmp eax,tiempoInicial       ; ¿menor que el inicial?

   jb error                    ; se regreso a cero

   sub eax,tiempoInicial       ; obtiene milisegundos transcurridos

   call WriteDec               ; los muestra
```

```
   mWrite<”milisegundos transcurridos”,0dh,0ah>
   jmp terminar
error:
   mWrite ”Error: GetTickCount inválido—el sistema ha”
   mWrite <”estado activo por más de 49.7 días”,0dh,0ah>
terminar:
   exit
main ENDP
END main
```

Función Sleep

Algunas veces los programas necesitan detenerse o retrasarse durante periodos breves. Aunque podríamos construir un ciclo de cálculo o un ciclo para mantener al procesador ocupado, el tiempo de ejecución de ese ciclo cariaría de un procesador a otro. Además, el ciclo ocupado tendría atado al procesador en forma innecesaria, reduciendo la velocidad de ejecución de otros programas a la vez. La función **Sleep** de Win32 suspende el subproceso actual en ejecución durante un número especificado de milisegundos:

```
Sleep PROTO,
   dwMiliseconds:DWORD
```

(Debido a que nuestros programas en lenguaje ensamblador tienen un solo subproceso, vamos a suponer que un subproceso es lo mismo que un programa). Un subproceso no ocupa tiempo del procesador mientras está dormido.

Procedimiento **GetDateTime**

El procedimiento **GetDateTime** en la biblioteca Irvine32 devuelve el número de intervalos de 100 nanosegundos que han transcurrido desde enero 1, 1601. Esto podría parecer algo extraño, si tomamos cuenta que las computadoras eran completamente desconocidas en esa época. En cualquier caso, Microsoft utiliza este valor para llevar la cuenta de las fechas y horas de los archivos. El SDK de Win32 recomienda los siguientes pasos cuando deseamos preparar un valor de fecha/hora del sistema para operaciones aritméticas de fecha:

1. Llamar a una función **GetLocalTime** para que llame una estructura SYSTEMTIME.
2. Convertir la estructura SYSTEMTIME en una estructura FILETIME, llamando la función **SystemTimeToFileTime**.
3. Copiar la estructura FILETIME resultante a una palabra cuádruple de 64 bits.

Una estructura FILETIME divide a una palabra cuádruple de 64 bits en dos dobles palabras:

```
FILETIME STRUCT
   loDateTime DWORD ?
   hiDateTime DWORD ?
FILETIME ENDS
```

El siguiente procedimiento **GetDateTime** recibe un apuntador a una variable tipo palabra cuádruple de 64 bits. Almacena la fecha y hora actuales en la variable, en formato FILETIME de Win32:

```
;---------------------------------
GetDateTime PROC,
   pDateTime:PTR QWORD
   LOCAL sysTime:SYSTEMTIME, filTime:FILETIME
; Obtiene y almacena la fecha/hora locales actuales como un
; entero de 64 bits (en el formato FILETIME de Win32).
;---------------------------------
; Obtiene la hora local del sistema.
   INVOKE GetLocalTime,
      ADDR sysTime
; convierte la estructura SYSTEMTIME a FILETIME.
   INVOKE SystemTimeToFileTime,
      ADDR sysTime,
      ADDR filTime
; Copia la estructura FILETIME en un entero de 64 bits.
   mov esi,pDateTime
   mov eax,flTime.loDateTime
   mov DWORD PTR [esi],eax
```

```
    mov eax,flTime.hiDateTime
    mov DWORD PTR [esi+4],eax
    ret
GetDateTime ENDP
```

Como una estructura SYSTEMTIME es un entero de 64 bits, podemos utilizar las técnicas aritméticas de precisión extendida mostradas anteriormente, para realizar operaciones aritméticas con fechas.

Escritura de una aplicación gráfica de Windows

En esta sección mostraremos cómo escribir una aplicación gráfica simple para Microsoft Windows. El programa crea y muestra una ventana principal, presenta cuadros de mensajes y responde a los eventos del ratón. La información que se proporciona a continuación sólo es una introducción; se requeriría cuando menos todo un capítulo completo para describir el funcionam9ento de incluso la aplicación MS Windows más simple.

La siguiente tabla presenta las diversas bibliotecas y archivos de inclusión que utilizamos para crear este programa.

Nombre de archivo	Descripción
WinApp.asm	Código de fuente del programa
GraphWin.inc	Archivo de inclusión que contiene estructuras, constantes y prototipos de funciones que utiliza el programa.
Kernel32.lib	La misma biblioteca de la API de MS Windows que utilizamos anteriormente.
User32.lib	Funciones adicionales de la API de MS Windows

/SUBSYSTEM:WINDOWS sustituye a /SUBSYSTEM:CONSOLE, que utilizamos anteriormente. El programa llama a las funciones de dos bibliotecas estándar de MS Windows: Kernel32.lib y user32.lib

Ventana principal: El programa muestra una ventana principal que llena la pantalla

Estructuras necesarias

La estructura **POINT** especifica las coordenadas X y Y de un punto en la pantalla, medido en pixeles. Por ejemplo, puede usarse para localizar objetos gráficos, ventanas y clics del ratón:

```
POINT STRUCT

   ptX DWORD ?

   ptY DWORD ?

POINTS ENDS
```

La estructura **RECT** define los límites de un rectángulo. El miembro **left** contiene la coordenada X del lado izquierdo del rectángulo. El miembro **top** contiene la coordenada Y de la parte superior del rectángulo. En los miembros **righ**t y **bottom** se almacenan valores similares:

```
REC STRUCT

   left DWORD ?

   top DWORD ?

   right DWORD ?

   bottom DWORD ?

RECT ENDS
```

La estructura **MSGStruct** define los datos necesarios para un mensaje de MS Windows:

```
MSGStruct STRUCT

   msgWnd DWORD ?

   msgMessage DWORD ?

   msgWparam DWORD ?

   msgLparam DWORD ?

   msgTime DWORD ?

   msgPt POINT <>

MSGStruct ENDS
```

La estructura **WINDCLASS** define una clase de ventana. Cada ventana en un programa debe pertenecer a una clase, y cada programa debe definir una clase de ventana para su ventana principal. Esta clase se registra con el sistema operativo antes de poder mostrar la ventana principal:

```
WNDCLASS STRCUT
```

```
    style DWORD ?               ; opciones de estilo de ventana
    lpfnWndProc DWORD ?         ;apuntador a la función WinProc
    cbClsExtra DWORD ?          ; memoria compartida
    cbWndExtra DWORD?           ; número de bytes adicionales
    hInstance DWORD ?           ; manejador para el programa actual
    hIcon DWORD ?               ; manejador para el ícono
    hCursor DWORD ?             ; manejador para el cursor
    hbrBackground DWORD ?       ; manejador para la brocha de fondo
    lpszMenuName DWORD ?        ; apuntador al nombre del menú
    lpszClassName DWORD ?       ; apuntador al nombre de WinClass
WNDCLASS ENDS
```

He aquí un breve repaso de los parámetros:

- *Style* es un conglomerado de distintas opciones de estilo, como WS_CAPTION y WS_BORDER, que controlan la apariencia y el comportamiento de la ventana.
- *lpfnWndProc* es un apuntador a una función (en nuestro programa) que recibe y procesa los mensajes de eventos activados por el usuario.
- *cbClsExtra* se refiere a la memoria compartida que utilizan todas las ventanas pertenecientes a la clase. Puede ser nulo.
- *cbWndExtra* especifica el número de bytes extras para asignar la siguiente instancia de ventana.
- *hInstance* guarda un manejador para la instancia actual del programa.
- *hIcon* y *hCursor* guardan manejadores a recursos tipo icono y cursor para el programa actual.
- *hbrBackground* guarda una brocha de fondo (color).
- *lpszMenuName* apunta a un nombre de menú.
- *lpszClassName* apunta a una cadena con terminación nula, que contiene el nombre de la clase de ventana.

La función MessageBox

La manera más fácil para que un programa muestre texto es colocarlo en un cuadro de mensaje que aparezca y espera a que el usuario haga clic en un botón. La función **MessageBox** de la biblioteca Win32 de la API muestra un cuadro de mensaje simple. He aquí su prototipo:

```
MessageBox PROTO,
```

```
    hWnd:DWORD,

    lpText:PTR BYTE,

    lpCaption:PTR BYTE,

    uType:DWORD
```

hWnd es un manejador para la ventana actual. *lpText* apunta a una cadena con terminación nula que aparecerá dentro del cuadrado. *lpCaption* apunta a una cadena con terminación nula que aparecerá en la barra de leyenda del cuadrado. *style* es un entero que describe tanto el icono (opcional) como los botones (requeridos) del cuadro de diálogo. Los botones se identifican mediante constantes como MB_OK y MB_YESNO. Los iconos también se identifican mediante constantes como MB_ICONQUESTION. Al mostrar un cuadrado de mensaje, podemos sumar las constantes para el icono y los botones:

```
INVOKE MessageBox, hWnd, ADDR TextoPregunta,

       ADDR TituloPregunta, MB_OK + MB_ICONQUESTION
```

El procedimiento WinMain

Toda aplicación Windows necesita un procedimiento de inicio que por, lo general, se llama **WinMain**, y es responsable de las siguientes tareas:

- Obtener un manejador para el programa actual.
- Cargar el ícono y cursor del ratón del programa.
- Registrar la clase de ventana principal del programa e identificar el procedimiento que procesará los mensajes de eventos para la ventana.
- Crear la ventana principal.
- Mostrar y actualizar la ventana principal.
- Empezar un ciclo que reciba y despache los mensajes. El ciclo continúa hasta que el usuario cierra la ventana de la aplicación.

WinMain contiene un ciclo de procesamiento de mensajes que llama a **GetMessage** para obtener el siguiente mensaje, disponible de la cola de mensajes del programa. Si GetMessage recibe un mensaje WM_QUIT, devuelve cero, indicando a WinMain que es tiempo de detener el programa. Para todos los demás mensajes, WinMain los pasa a la función **DispatchMessage**, la cual los reenvía al procedimiento WinProc del programa.

El procedimiento WinProc

El procedimiento **WinProc** recibe y procesa todos los mensajes de eventos relacionados con una ventana. La mayoría de los eventos los indica el usuario, haciendo clic y arrastrando el ratón, oprimiendo teclas, etcétera. El trabajo de este procedimiento es decodificar cada mensaje, y si éste se reconoce, llevar a cabo las tareas orientadas a la aplicación, relacionadas con el mensaje. He aquí la declaración:

```
WinProc PROC,
   hWnd:DWORD,                ; manejador para la ventana
   localMsg:DWORD,            ; ID del mensaje
   wParam:DWORD,              ; parámetro 1 (varía)
   lParam:DWORD               ; parámetro 2 (varía)
```

El contenido de los parámetros tercero y cuarto variará, dependiendo del ID del mensaje específico. Por ejemplo, cuando se hace clic con el ratón. *lParam* contiene las coordenadas X y Y del punto en el que se hizo clic. En el siguiente programa de ejemplo, el procedimiento **WinProc** maneja tres mensajes específicos:

- WM_LBUTTONDOWN, que genera cuando el usuario oprime el botón izquierdo del ratón.
- WM_CREATE, que indica que se acaba de crear la ventana principal.
- WM_CLOSE, que indica que la ventana principal de la aplicación está a punto de cerrar.

Por ejemplo, las siguientes líneas (del procedimiento) manejan el mensaje WM_LBUTTONDOWN, llamando a MessageBox para mostrar un mensaje contextual al usuario:

```
.IF eax == WM_LBUTTONDOWN
   INVOKE MessageBox, hWnd, ADDR TextoContextual,
      ADDR TituloContextual, MB_OK
   jmp TerminarWinProc
```

Cualquier otro mensaje que no deseamos manejar se pasa a **DefWindowsProc**, el manejador de mensajes predeterminado para MS Windows.

El Procedimiento ErrorHandler

El procedimiento **ErrorHandler**, que es opcional, se llama si el sistema reporta un error durante el registro y la creación de la ventana principal del programa. Por ejemplo, la función **RegisterClass** Devuelve un valor distinto de cero si la ventana principal del programa se registró con éxito. Pero si devuelve cero, llamamos a ErrorHandler (para mostrar un mensaje) y terminamos el programa:

```
INVOKE RegisterClass, ADDR ventPrinc
.IF eax == 0
   call ErrorHandler
   jmp Terminar_Programa
```

```
.ENDIF
```

El procedimiento **ErrorHandler** tiene varias tareas importantes que realizar:

- Llamar a **GetLastError** para obtener el número de error del sistema.
- Llamar a **FormatMessage** para obtener la cadena de mensaje de error apropiada con formato del sistema.
- Llamar a **MessageBox** para mostrar un cuadro de mensaje contextual que contenga la cadena de mensaje de error.
- Llamar a **LocalFree** para liberar memoria utilizada por la cadena de mensaje de error.

Listado de un programa

La mayoría es código que sería idéntico en cualquier aplicación MS Windows:

```
TITLE Aplicación Windows                    (WinApp.asm)


; Este programa muestra una ventana de aplicación con

; tamaño ajustable y varios cuadros de mensajes contextuales.

; Gracias a Tom Joyce por crear un prototipo

; a partir del cual se derivó este mensaje.


INCLUDE Irvine32.inc

INCLUDE GraphWin.inc


;==================== DATOS =======================

.data


TituloMsjCargaApp BYTE "Se cargo la aplicación",0

TextoMsjCargaApp  BYTE "Esta ventana aparece cuando se recibe"

                  BYTE "el mensaje WM_CREATE",0


TituloContextual  BYTE "Ventana contextual",0
```

```
TextoContextual   BYTE "Esta ventana se activo mediante un"
                  BYTE "mensaje WM_LBUTTONDOWN",0

TituloBienvenida  BYTE "Ventana principal activa",0
TextoBienvenida   BYTE "Esta ventana se muestra justo despues"
                  BYTE "de llamar a CreateWindow y UpdateWindow.",0

MsjCerrar         BYTE "Se recibió el mensaje WM_CLOSE",0

TituloError       BYTE "Error",0
NombreVentana     BYTE "ASM Windows App",0
NombreClase       BYTE "ASMWin",0

; Define la estructura de la clase de ventana de la aplicación.
VentPrinc WNDCLASS <NULL,WinProc,NULL,NULL,NULL,NULL,NULL, \
                                COLOR_WINDOW,NULL,NombreClase>

msg                                 MSGStruct <>
winRect   RECT <>
hMainWnd  DWORD ?
hInstance DWORD ?

;=================== CÓDIGO =========================
.code
WinMain PROC
; Obtiene un manejador para el proceso actual.
```

```
   INVOKE GetModuleHandle, NULL
   mov hInstance, eax
   mov VentPrinc.hInstance, eax

; Carga el icono y el cursor del programa.
INVOKE LoadIcon, NULL, IDI_APPLICATION
mov VentPrinc.hIcon, eax
INVOKE LoadCursor, NULL, IDC_ARROW
mov VentPrinc.hCursor, eax

; Registra la clase de ventana.
INVOKE RegisterClass, ADDR VentPrinc
.IF eax == 0
   call ErrorHandler
   jmp Terminar_Programa
.ENDIF

; Crea la ventana principal de la aplicación.
; Devuelve un manejador para la ventana principal en EAX.
INVOKE CreateWindowEx, 0, ADDR NombreClase,
   ADDR NombreVentana,MAIN_WINDOW_STYLE,
   CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,
   CW_USEDEFAULT,NULL,NULL,hInstance,NULL
   mov hMainWnd,eax

; Si CreateWindowEx falló, muestra un mensaje y termina.
```

```
.IF eax == 0
   call ErrorHandler
   jmp  Terminar_Programa
.ENDIF

; Muestra y dibuja la ventana.
INVOKE ShowWindow, hMainWnd, SW_SHOW
INVOKE UpdateWindow, hMainWnd

; Muestra un mensaje de bienvenida.
INVOKE MessageBox, hMainWnd, ADDR TextoBienvenida,
   ADDR TituloBienvenida, MB_OK

; Empieza el ciclo de manejo de mensajes del programa.
Ciclo_Mensajes:
   ; Obtiene el siguiente mensaje de la cola.
   INVOKE GetMessage, ADDR msg, NULL,NULL,NULL

   ; Termina si no hay más mensajes.
.IF eax == 0
   jmp Terminar_Programa
.ENDIF

   ; Releva el mensaje al WinProc del programa.
   INVOKE DispatchMessage, ADDR msg
    jmp Ciclo_Mensajes
```

```
Terminar_Programa:
   INVOKE ExitProcess,0
WinMain ENDP
```

En el ciclo anterior, la estructura msg se pasa a la función **GetMessage**. Esta función llena la estructura, que a su vez se pasa a la función **DispatchMessage** de MS Windows.

```
;-----------------------------------------------------
WinProc PROC,
   hWnd:DWORD, localMsg:DWORD, wParam:DWORD, lParam:DWORD
; El manejador de mensajes de la aplicación, que maneja
; mensajes específicos de la aplicación. Todos los demás mensajes
; se pasan al manejador de mensajes predeterminado de Windows.
;-----------------------------------------------------
mov eax, localMsg

.IF eax == WM_LBUTTONDOWN          ; ¿botón del ratón?
   INVOKE MessageBox, hWnd, ADDR TextoContextual,
   ADDR TituloContextual, MB_OK
jmp TerminarWinProc
.ELSEIF eax == WM_CREATE           ; ¿crear ventana?
INVOKE MessageBox, hWnd, ADDR TextoMsjCargaApp,
      ADDR TituloMsjCargaApp, MB_OK
jmp TerminarWinProc
.ELSEIF eax == WM_CLOSE            ; ¿cerrar ventana?
```

```
 INVOKE MessageBox, hWnd, ADDR MsjCerrar,
   ADDR NombreVentana, MB_OK
 INVOKE PostQuitMessage,0
 jmp TerminarWinProc
.ELSE                              ; otro mensaje?
 INVOKE DefWindowProc, hWnd, localMsg, wParam, lParam
 jmp TerminarWinProc
.ENDIF

TerminarWinProc:
 ret
WinProc ENDP

;-----------------------------------------------------
ErrorHandler PROC
; Muestra el mensaje de error apropiado del sistema.
;-----------------------------------------------------
.data
pMsjError  DWORD ?                 ; apuntador al mensaje de error
IDmensaje  DWORD ?
.code
 INVOKE GetLastError           ; Devuelve ID de mensaje en EAX
 mov IDmensaje,eax

; Obtiene la cadena de mensaje correspondiente.
INVOKE FormatMessage, FORMAT_MESSAGE_ALLOCATE_BUFFER + \
```

```
FORMAT_MESSAGE_FROM_SYSTEM,NULL,messageID,NULL,

ADDR pErrorMsg,NULL,NULL


; Muestra el mensaje de error.

INVOKE MessageBox,NULL, pMsjError, ADDR TituloError,

   MB_ICONERROR+MB_OK


; Libera la cadena de mensaje de error.

INVOKE LocalFree, pMsjError

 ret

ErrorHandler ENDP

END WinMain
```

Asignación dinámica de memoria

La asignación dinámica de memoria, también conocida como *asignación del montón*, *heap* en inglés, es una herramienta que tienen los lenguajes de programación para reservar memoria cuando se crean objetos, arreglos y otras estructuras. Por ejemplo, en Java una instrucción como la siguiente ocasiona que se reserve memoria para un objeto String:

```
String cad = new String(“abcde”);
```

De manera similar, en C++ puede ser conveniente asignar espacio para un arreglo de enteros, usando un atributo de tamaño de una variable

```
int tamanio;

cin >> tamanio                  // el usuario introduce el tamaño

int arreglo[] = new int[tamanio];
```

C, C++ y Java tienen administradores integrados del montón de datos en tiempo de ejecución, que se encargan de las peticiones mediante programación para la asignación y liberación de espacio de almacenamiento. Por lo general, los administradores de montón de datos o asignación dinámica de memoria, asignan un bloque extenso de memoria del sistema operativo cuando el sistema inicia. Crean una *lista libre* de apuntadores a bloques de

almacenamiento. Cuando se recibe una petición de asignación, el administrador del montón de datos marca un bloque de memoria del tamaño apropiado como reservado, y devuelve un apuntador al bloque. Más adelante, cuando se recibe una petición de eliminación para el mismo bloque, el montón de datos libera el bloque y lo devuelve a la lista libre. Cada vez que se recibe una nueva petición de asignación, el administrador del montón de datos explora la lista libre, buscando el primer bloque disponible que sea bastante grande como para otorgar la petición.

Los programas en lenguaje ensamblador pueden realizar la asignación dinámica de dos maneras. En primer lugar, pueden implementar sus propios administradores del montón de datos, que atiendan las peticiones para los objetos más pequeños. En esta sección mostraremos como implementar el primer método. El programa de ejemplo es una aplicación de 32 bits en modo protegido.

Podemos solicitar varios bloques de memoria de diversos tamaños a MS Windows, usando varias funciones de la API de Windows que se presentan a continuación:

Función	Descripción
GetProcessHeap	Devuelve en EAX un manejador de entero de 32 bits al área del montón de datos existentes del programa. Si la función tiene éxito, devuelve un manejador para el montón de datos en EAX. Si falla, el valor de retorno en EAX es NULL
HeapAlloc	Asigna un bloque de memoria de un montón de datos. Si tiene éxito, el valor de retorno en EAX contiene la dirección del bloque de memoria, si falla, el valor devuelto en EAX es NULL
HeapCreate	Crea un nuevo montón de datos y lo pone a disposición del programa que hizo la llamada. Si la función tiene éxito, devuelve en EAX un manejador para el montón de datos recién creado. Si falla, el valor de retorno en EAX es NULL
HeapDestroy	Destruye el objeto de montón de datos especificado e invalida su manejador. Si la función tiene éxito, el valor del retorno en EAX es distinto de cero
HeapFree	Libera un bloque de memoria que estaba antes asignado a un montón de datos, identificado por su dirección y su manejador. Si el bloque se libera con éxito, el valor de retorno es distinto de cero
HeapReAlloc	Reasigna y cambia de tamaño un bloque de memoria de un montón de pila. Si la función tiene éxito, el valor de retorno es un apuntador a un bloque de memoria reasignado. Si la función falla y no se especifica HEAP_GENERATE_EXCEPTIONS, el valor de retorno es NULL
HeapSize	Devuelve el tamaño de un bloque de memoria que estaba antes asignado mediante una llamada a HeapAlloc o HeapReAlloc. Si la función

	tiene éxito, EAX contiene el tamaño del bloque de memoria asignado en bytes, si la función falla, el valor de retorno es SIZE_T – 1 (SIZE_T es igual al número máximo de bytes al que puede apuntar un apuntador)

Todas estas funciones sobrescriben los registros de propósito general, por lo que tal vez sea conveniente crear procedimientos de envoltura para meter y sacar registros importantes.

GetProcessHeap GetProcessHeap es suficiente si nos conformamos con utilizar el montón de datos predeterminado que posee el programa actual. No tiene parámetros, y el valor de retorno en EAX es el manejador del montón de datos:

```
GetProcessHeap PROTO
```

Llamada de ejemplo:

```
.data

hMonton HANDLE ?

.code

INVOKE GetProcessHeap

.IF eax == NULL

   jmp = terminar

.ELSE

   mov hMonton,eax              ; el manejador está bien

.ENDIF
```

HeapCreate HeapCreate nos permite crear un nuevo montón de datos privado para el programa actual:

```
HeapCreate PROTO,

  flOptions:DWORD,              ; opciones de asignación del montón de datos

  dwInitialSize:DWORD,          ; tamaño inicial del montón, en bytes

  dwMaxinumSize:DWORD           ; tamaño máximo del montón, en bytes
```

Hay que establecer *flOptions* a NULL. Se establece también *dwInitialSize* al tamaño inicial del montón, en bytes. El valor se redondea hasta el siguiente límite de página. Cuando las llamadas a HeapAlloc exceden en el tamaño inicial del montón de datos, éste crece hasta el valor que se especifica en el parámetro *dwMaxinumSize* (redondeando al siguiente límite de

página). Después de llamar a esta función, un valor de retorno nulo en EAX indica que el montón de datos no se creó. He aquí una llamada a HeapCreate:

```
HEAP_STRAT = 2000000          ; 2 MB

HEAP_MAX = 400000000          ; 400 MB

.data

hMonton = HANDLE ?            ; manejador para el montón

.code

INVOKE HeapCreate, 0, HEAP_STRAT, HEAP_MAX

.IF eax == NULL               ; no se creó el montón

   call WriteWindowsMsg       ; muestra el mensaje de error

   jmp terminar

.ELSE

   mov hMonton,eax            ; el manejador está bien

.ENDIF
```

HeapDestroy HeapDestroy destruye un montón privado existente (uno creado por HeapCreate). Recibe un manejador para el montón

```
HeapDestroy PROTO,

   hHandle:DWORD              ; manejador del montón
```

Si no se puede destruir el montón de datos, EAX es igual a NULL. A continuación se muestra una llamada de ejemplo, usando el procedimiento WriteWindowsMsg descrito anteriormente:

```
.data

hHandle HANDLE ?              ; manejador para el montón

.code

INVOKE HeapDestroy,hMonton

.IF eax == NULL

   call WriteWindowsMsg       ; muestra el mensaje de error

.ENDIF
```

HeapAlloc HeapAlloc asigna un bloque de memoria de un montón de datos existentes:

```
HeapAlloc PROTO,
   hHandle:HANDLE,        ; manejador para el bloque del montón privado
   dwFlags:DWORD,         ; banderas de control de asignación de montón
   dwBytes:DWORD          ; número de bytes a asignar
```

Recibe los siguientes argumentos:

- *hHeap*, un manejador de 32 bits para un montón inicializado con GetProcessHeap o HeapCreate.
- *dwFlag*s, una doble palabra que contiene uno más valores de bandera. Podemos establecer de manera opcional a HEAP_ZERO_MEMORY, la cual establece todo el bloque de memoria a cero.
- *dwBytes*, una doble palabra que indica el tamaño del montón, en bytes.

Si HeapAlloc falla, el valor devuelto en EAX es NULL. Las siguientes instrucciones asignan un arreglo de 1000 bytes del montón identificado por **hMonton**, y establecen sus valores a cero:

```
.data
hMonton HANDLE ?               ; manejador del montón
pArreglo DWORD ?               ; apuntador al arreglo
.code
INVOKE HeapAlloc, hMonton, HEAP_ZERO_MEMORY, 1000
.IF eax == NULL
   mWrite “HeapAlloc fallo”
   jmp terminar
.ELSE
   mov pArreglo,eax
.ENDIF
```

HeapFree La función HeapFree libera un bloque de memoria que estaba asignado a un montón de datos, identificado por su dirección y manejador:

```
HeapFree PROTO,
```

```
   hHeap:HANDLE,

   dwFlags:DWORD,

   lpMem:DWORD
```

El primer argumento es un manejador para el montón de datos que contiene el bloque de memoria, por lo general, el segundo argumento es cero: el tercer argumento es un apuntador al bloque de memoria que se va a liberar. Si el bloque se libera con éxito, el valor de retorno es distinto de cero. Si el bloque no puede liberarse, la función devuelve cero. He aquí un llamado de ejemplo:

```
INVOKE HeapFree, hHeap, 0, pArray
```

Manejo de errores Si encontramos un error al llamar a HeapCreate, HeapDestroy o GetProcessHeap, podemos obtener los detalles llamando a la función **GetLastError** de la API. O podemos llamar a la función **WriteWindowsMsg** de la biblioteca Irvine32. A continuación se muestra un ejemplo de llamada a HeapCreate:

```
INVOKE HeapCreate, 0, HEAP_START, HEAP_MAX

.IF eax == NULL                ; ¿falló?

   call WriteWindowsMsg        ; muestra el mensaje de error

.ELSE

   mov hMonton,eax             ; éxito

.ENDIF
```

Por otro lado la función HeapAlloc no establece un código de error cuando el sistema falla, por lo que no podemos llamar a GetLastError o WriteWindowsMsg

Administración de memoria en la familia IA-32

Cuando salió MS Windows 3.0 por primera vez al mercado, hubo mucho interés entre los programadores en cuanto al cambio del modo de direccionamiento real al modo protegido. ¡Cualquiera que haya escrito programas para Windows 2.x recordará lo difícil que era permanecer dentro de los 640K en el modo de direccionamiento real! Con el modo protegido de Windows (y poco después, el modo virtual), parecían abrirse posibilidades completamente nuevas. No debemos olvidar que fue el procesador Intel386 (el primero de la familia IA-32) el que hizo todo esto posible. Lo que ahora damos por hecho fue una evolución gradual, desde el inestable Windows 3.0 hasta las sofisticadas (y estables) versiones de Windows y Linux que se ofrecen hoy en día.

En esta sección, nos enfocaremos en dos de los aspectos principales de la administración de memoria:

- Traducir las direcciones lógicas en direcciones lineales.
- Traducir las direcciones lineales en direcciones físicas (paginación).

Revisemos brevemente algunos de los términos de administración de memoria de la familia IA-32 que presentamos anteriormente, empezando con los siguientes:

- *Multitarea*: permite la ejecución de varios programas (o tareas) al mismo tiempo. El procesador divide su tiempo entre todos los programas en ejecución.
- *Segmentos*: son áreas de memoria con tamaño variable, que un programa utiliza para contener código o datos.
- *Segmentación*: proporciona una manera de aislar los segmentos de memoria, unos de otros. Esto permite que varios programas se ejecuten al mismo tiempo, sin interferir unos con otros.
- *Descriptor de segmento*: es un valor de 64 bits que identifica y describe a un solo segmento de memoria: Contiene información acerca de la dirección base del segmento, sus derechos de acceso, el límite de tamaño, tipo y uso.

Ahora vamos a agregar dos nuevos términos a la lista:

- Un *selector de segmento*, es un valor de 16 bits que se almacena en un registro de segmento (CS, DS, SS, ES, FS, GS).
- Una *dirección lógica* es una combinación de un selector de segmento y un desplazamiento de 32 bits.

A lo largo de este resumen hemos ignorado los registros de segmento, ya que los programas de usuario nunca los modifican de forma directa. Nos hemos enfocado sólo en el desplazamiento de datos de 32 bits. Sin embargo, desde el punto de vista del programador, los registros de segmentos son importantes ya que contienen referencias indirectas a segmentos de memoria.

Direcciones lineales

Traducción de direcciones lógicas a direcciones lineales

Un sistema operativo multitarea permite que varios programas (tareas) se ejecuten en memoria al mismo tiempo. Cada programa. Cada programa tiene su propia área de datos. Supongamos que cada uno de tres programas tiene una variable de 200h; ¿Cómo podrían las tres variables estar separadas sin compartirse? La respuesta a esto es que el procesador IA-32 utiliza un proceso de uno o dos pasos para convertir el desplazamiento de cada variable en una ubicación única de memoria.

El primer paso combina un valor de segmento con el desplazamiento de una variable para crear una *dirección lineal*. Esta dirección lineal podría ser la dirección física de la variable. Pero los sistemas operativos como MS Windows y Linux emplean una característica de la familia IA-32, conocida como *paginación*, para permitir que los programas utilicen más memoria lineal de la que haya físicamente disponible en la computadora. Deben usar un segundo paso llamado *traducción de página* para convertir una dirección lineal en una dirección física.

Primero veamos la forma en que el procesador utiliza un segmento y un desplazamiento para determinar la dirección lineal de una variable. Cada selector de segmento apunta a un descriptor de segmento (un una tabla de descriptores), el cual contiene la dirección base de un segmento de memoria. El desplazamiento de 32 bits de la dirección lógica se suma a la dirección base del segmento, con lo cual genera una *dirección lineal* de 32 bits.

Dirección lineal Una *dirección lineal* es un entero de 32 bits que varía entre 0 y FFFFFFFFh, el cual se refiere a una ubicación de memoria. La dirección lineal también puede ser la dirección física de los datos de destino, si una característica conocida como *paginación* está deshabilitada.

Paginación

La *paginación* es una importante característica del procesador IA-32, que hace posible que una computadora ejecute una combinación de programas que de otra manera no cabrían en memoria. Para ello, el procesador carga al inicio sólo una parte de un programa de memoria, mientras mantiene las partes restantes en disco. La memoria utilizada por el programa se divide en pequeñas unidades llamadas *páginas*, por lo general, de 4KB cada una. A medida que se ejecuta cada programa, el procesador descarga en forma selectiva las páginas inactivas de la memoria y carga otras páginas que se requieren de inmediato

El sistema operativo mantiene un *directorio de páginas* y conjuntos de *tablas de páginas* para llevar la cuenta de las páginas utilizadas por todos los programas que se encuentra actualmente en memoria. Cuando un programa intenta acceder a una dirección en alguna parte del espacio de direcciones lineales, el procesador convierte en forma automática la dirección lineal en una dirección física. A esta conversión se le conoce como *traducción de páginas*. Si la página solicitada no se encuentra actualmente en memoria, el procesador interrumpe al programa y genera un *fallo de página*. El sistema operativo copia la página requerida del disco duro a la memoria antes de que el programa pueda continuar. Desde el punto de vista de un programa de aplicación, los fallos de pagina y la traducción de páginas ocurren en forma automática.

Tablas de descriptores

Podemos encontrar a los descriptores de segmento en dos tipos de tablas: *tablas de descriptores globales* (GDT) y *tablas de descriptores locales* (LDT).

Tabla de descriptores globales (GDT) Cuando el sistema operativo cambia al procesador al modo protegido durante el arranque, se crea una sola tabla de descriptores globales. Su dirección base se guarda en el GDTR (*registro de tabla de descriptores globales*). La tabla contiene entradas (*llamadas descriptores de segmento*) que apuntan a segmentos. El sistema operativo tiene la opción de almacenar los segmentos que utilizan todos los programas en la GDT.

Tablas de descriptores locales (LDT) En un sistema operativo multitarea, a cada tarea o programa, por lo general, se le asigna su propia tabla de descriptores de segmento, conocida como *tabla de descriptores locales* (LDT). El registro LDTR contiene la dirección de la LDT del

programa. Cada descriptor de segmento contiene la dirección base de un segmento dentro del espacio de direcciones lineales. Por lo general, este segmento es distinto de los demás.

Detalles acerca de los descriptores de segmento

Además de la dirección base del segmento, el descriptor de segmento contiene campos de mapas de bits que especifican el límite del segmento y su tipo. Un ejemplo de un tipo de segmento de sólo lectura es el segmento de código. Si un programa trata de modificar un segmento de sólo lectura, se genera un fallo de página. Los descriptores de segmento pueden contener niveles de protección para evitar que los datos del sistema operativo estén accesibles a los programas de aplicación. A continuación se muestran descripciones de campos de selectores individuales:

Dirección base: un entero de 32 bits que define la ubicación inicial del segmento en el espacio de direcciones lineales de 4 GB.

Nivel de privilegio: a cada segmento se le puede asignar un nivel de privilegio entre 0 y 3, en donde 0 es el que tiene más privilegio, por lo general, para el código núcleo del sistema operativo. Si un programa con un nivel de privilegio con numeración alta trata de acceder a un segmento que tenga un nivel de privilegio con numeración más baja, se genera un fallo del procesador.

Tipo de segmento: indica el tipo de segmento y especifica el tipo de acceso que puede realizarse en el segmento, y la dirección en la que puede crecer (hacia arriba o hacia abajo). Los segmentos de datos (incluyendo la Pila) pueden ser de sólo lectura o de lectura/escritura, y pueden crecer hacia arriba o hacia abajo. Los segmentos de código puede ser de sólo ejecución o de ejecución/sólo lectura.

Bandera de segmento presente: este bit indica si el segmento se encuentra presente en la memoria física.

Bandera de Granularidad: determina la interpretación del campo de límite del segmento. Si el bit es cero, el límite del segmento se interpreta en unidades de bytes. Si el bit está activo, el límite de segmento se interpreta en unidades de 4096 bytes.

Límite de segmento: es un entero de 20 bits que especifica el tamaño del segmento. Se interpreta en una de las siguientes formas, dependiendo de la bandera Granularidad:

- El número de bytes en el segmento, varía de 1 a 1MB.
- El número de unidades de 4096 bytes, lo cual permite que el tamaño del segmento varíe de 4KB a 4GB.

Traducción de páginas

Cuando está habilitada la paginación, el procesador debe traducir una dirección lineal de 32 bits en una dirección física de 32 bits. Se utilizan tres estructuras en el proceso:

- Directorio de página: un arreglo hasta de 1024 entradas de directorio de página de 32 bits.
- Tabla de páginas: un arreglo de hasta 1024 entradas de tabla de página de 32 bits.
- Página: un espacio de direcciones de 4KB o 4MB.

Para simplificar la siguiente discusión, vamos a suponer que se utilizan páginas de 4KB:

Una dirección lineal se divide en tres campos: un apuntador a una entrada de directorio de página, un apuntador a una entrada de tabla de página y un desplazamiento a un marco de página. El registro de control (CR3) contiene la dirección inicial del directorio de páginas. El procesador lleva a cabo los siguientes pasos al traducir una dirección lineal a una dirección física.

1. La *dirección lineal* hace referencia a una ubicación en el espacio de direcciones lineales.
2. El campo *directorio* de 10 bits en la dirección lineal es un índice a una entrada del directorio de páginas. Esta entrada contiene la dirección base de una tabla de páginas.
3. El campo *tabla* de 10 bits en la dirección lineal es un índice a la tabla de páginas, el cual se identifica mediante la entrada en el directorio de páginas. La entrada de la tabla de páginas en esa posición contiene la ubicación base de una *página* en la memoria física.
4. El campo desplazamiento de 12 bits en la dirección lineal se suma a la dirección base de la página, generando la dirección física exacta del operando.

El sistema operativo tiene la opción de utilizar un solo directorio de páginas para todos los programas y tareas en ejecución, o un directorio de páginas por tarea, o una combinación de ambos.

12 Interfaz con leguajes de alto nivel

Se ha excluido este capítulo

13 Programación en MS-DOS de 16 bits

MS-DOS y la IBM-PC

PC-DOS de IBM fue el primer sistema operativo que implementó el modo de direccionamiento real en la Computadora Personal IBM, usando el procesador Intel 8088. Más tarde, evolucionó para convertirse en Microsoft MS-DOS. De ahí que tenga sentido utilizar MS-DOS como el entorno para explicar la programación en modo direccionamiento real. A este modo también se le conoce como *modo de 16 bits*, ya que las direcciones se construyen a partir de valores de 16 bits.

En este capítulo conoceremos la organización básica de la memoria de MS-DOS, cómo activar las llamadas a funciones de MS-DOS (conocidas como *interrupciones*) y como realizar operaciones básicas de entrada/salida en el nivel del sistema operativo. Todos los programas en este capítulo se ejecutarán en modo direccionamiento real, debido a que utilizan

instrucciones INT. Las interrupciones se diseñaron en un principio para ejecutarse bajo MS-DOS en modo direccionamiento real. Es posible llamar a las interrupciones en modo protegido, pero estas técnicas no serán explicadas.

Los programas en modo de direccionamiento real tienen las siguientes características:

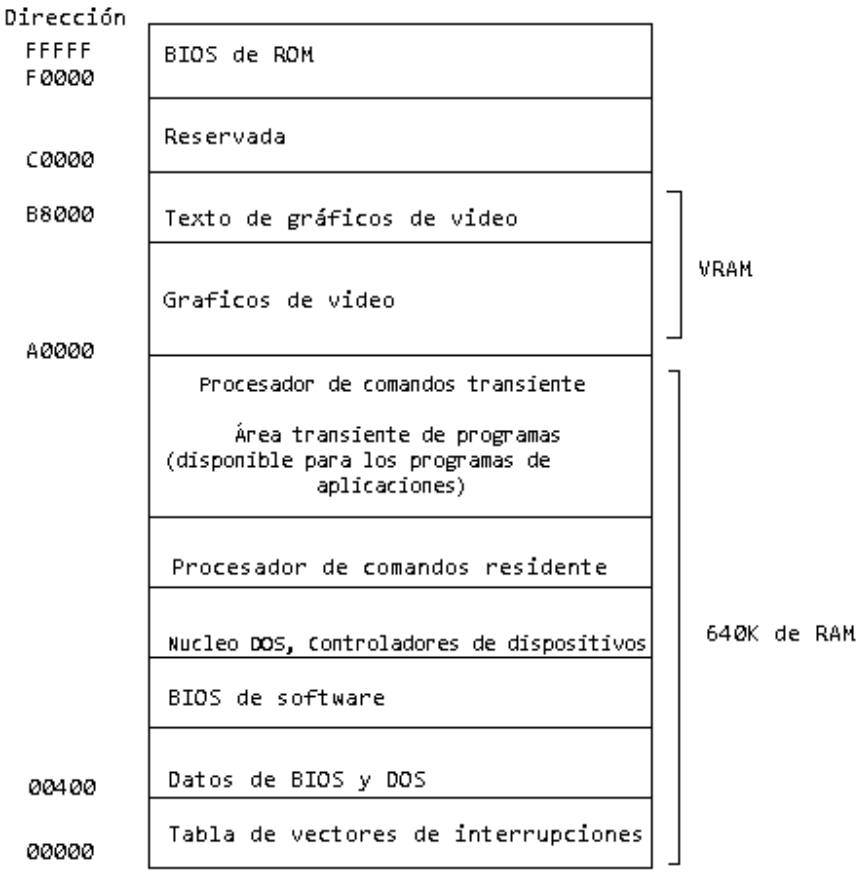
- Solo pueden direccionar 1 megabytes de memoria.
- Solo puede ejecutarse un programa a la vez (una sola tarea) en una sola sesión.
- No es posible la protección de límites de memoria, por lo que cualquier programa de aplicación puede sobrescribir la memoria que utiliza el sistema operativo.
- Los desplazamientos son de 16 bits.

Cuando apareció por primera vez, la IBM-PC fue muy atractiva, ya que era económica y ejecutaba Lotus 1-2-3, el programa de hojas electrónicas de cálculo que era útil para que las empresas adoptaran la PC. A los aficionados de computadoras les fascino la PC, ya que era una herramienta idea para aprender su funcionamiento. Hay que recalcar que Digital Research CP/M, el sistema operativo de 8 bits más popular antes de PC-DOS, sólo era capaz de direccionar 64K de RAM. Desde este punto de vista, los 640K de PC-DOS parecían un regalo del cielo.

Debido a las evidentes limitaciones de memoria y velocidad de los primeros microprocesadores Intel, la IBM-PC era una computadora de un solo usuario. No había protección integrada contra la corrupción de memoria que podrían provocar los programas de aplicaciones. En contraste, los sistemas de minicomputadores disponibles en esa época podían manejar varios usuarios y evitaban que los programas de aplicaciones sobrescribieran los datos uno de otros. Con el tiempo aparecieron sistemas operativos más robustos para la PC, con lo cual se convirtió en una alternativa viable para los sistemas de microcomputadoras, en especial cuando se conectaban varias PCs en red.

Organización de la memoria

En el modo direccionamiento real, tanto el sistema operativo como los programas de aplicaciones utilizan los 640k inferiores de memoria. Después de éstos sigue la memoria de video y la memoria reservada para los controladores de hardware. Por último, las ubicaciones F0000 a FFFFF están reservadas para la ROM (memoria de sólo lectura) del sistema. La siguiente figura muestra un mapa simple de memoria:



Dentro del área de memoria del sistema operativo, los 1024 bytes inferiores de memoria (direcciones 00000 a 003FF) contienen una tabla de direcciones de 32 bits, llamada *tabla de vectores de interrupción*. Estas direcciones, llamadas *vectores de interrupción*, las utiliza la CPU al procesar las interrupciones de hardware y software.

Justo encima de la tabla de vectores se encuentra el *área de datos del BIOS* y de *MS-DOS*. Después sigue el *BIOS de software*., que incluye los procedimientos para administrar la mayoría de los dispositivos de E/S, incluyendo el teclado, el disco duro, la pantalla de video, los puertos seriales y el puerto de impresora. Los procedimientos del BIOS se cargan desde un archivo oculto del sistema en un disco de sistema (arranque) de MS-DOS. El núcleo de MS-DOS es una colección de procedimientos (*llamados servicios*) que también se cargan desde un archivo en el disco del sistema.

Junto con el núcleo de MS-DOS se encuentran los búfer de archivo y los controladores de dispositivos instalables. En la siguiente parte más alta de la memoria, la parte residente del *procesador* de *comandos* se carga desde un archivo ejecutable llamado *command.com*. El procesador de comandos interpreta los comandos que se escriben en el símbolo de MS-DOS para cargar y ejecutar los programas almacenados en disco. Una segunda parte del procesador de comandos ocupa la memoria más alta, justo debajo de la ubicación A0000.

Los programas de aplicaciones se pueden cargar en memoria, en la primera dirección encima de la parte residente del procesador de comandos, y pueden usar toda la memoria hasta la dirección 9FFFFF. Si el programa actual en ejecución sobrescribe el área transiente del procesador de comandos, éste se vuelve a cargar del disco de arranque cuando el programa termina.

Memoria de video El área de la memoria de video (VRAM) en una IBM-PC empieza en la ubicación A0000, la cual se utiliza cuando el adaptador de video cambia al modo gráfico. Cuando el video se encuentra en modo de texto a color, la ubicación de memoria B8000 almacena todo el texto que se muestra en pantalla. La pantalla representa un mapa en la memoria, de manera que cada fila y columna en la pantalla corresponde a una palabra de 16 bits en la memoria. Cuando se copia un carácter a la memoria de video, aparece de inmediato en la pantalla.

BIOS de ROM El *BIOS* de *ROM*, que se encuentra en las ubicaciones de memoria F0000 a FFFFF, es una parte importante del sistema operativo de la computadora. Contiene software de diagnóstico y configuración del sistema, así como procedimientos de entrada-salida de bajo nivel que utilizan los programas de aplicaciones. El BIOS se almacena en un chip de memoria estática en la tarjeta del sistema. La mayoría de los sistemas siguen una especificación estandarizada del BIOS, modelada a partir del BIOS original del IBM, y utilizan el área de datos del BIOS de 00400 a 004FF.

Redirección de entrada-salida

A lo largo de este capítulo haremos referencia al *dispositivo de entrada estándar* y al *dispositivo de salida estándar*. Ambos se conocen en conjunto como la *consola*, en la que se utiliza el teclado para la entrada y la pantalla de video para la salida.

Al ejecutar programas desde el símbolo de sistema, podemos redirigir la entrada estándar de manera que se lea de un archivo o puerto de hardware, en vez de hacerlo del teclado. La salida estándar puede redirigirse a un archivo, impresora y otro dispositivo de E/S. Sin esta capacidad, los programas tendrían que revisarse considerablemente antes de poder modificar su entrada-salida. Por ejemplo, el sistema operativo tiene un programa llamado *sort.exe* que ordena un archivo de entrada. El siguiente comando ordena un archivo *miarchivo.txt* y muestra la salida

```
sort < miarchivo.txt
```

El siguiente comando ordena miarchivo.txt y envía la salida a archsalida.txt

```
sort < miarchivo.txt > archsalida.txt
```

Podemos utilizar el símbolo de canalización (|) para copiar la salida del comando DIR a la entrada del programa *sort.exe*. El siguiente comando ordena el directorio actual del disco y muestra la salida en la pantalla

```
dir | sort
```

El siguiente comando envía la salida del programa sort a la impresora predeterminada (sin conexión en red)(se identifica por PRN):

```
dir | sort > prn
```

En la siguiente tabla se muestra el conjunto completo de nombres de dispositivos:

Nombre de dispositivo	Descripción
CON	Consola (pantalla de video o teclado)
LPT1 o PRN	Primera impresora en paralelo
LPT2, LPT3	Puertos paralelos 2 y 3
COM1,COM2	Puertos seriales 1 y 2
NUL	Dispositivo inexistente o tonto

Interrupciones de software

Una *interrupción de software* es una llamada a un procedimiento del sistema operativo. La mayoría de estos procedimientos, llamados *manejadores de interrupciones*, proporcionan la capacidad de entrada-salida a los programas de aplicaciones. Se utilizan para las siguientes tareas:

- Mostrar caracteres y cadenas.
- Leer caracteres y cadenas de teclado.
- Mostrar texto a color.
- Abrir y cerrar archivos.
- Leer datos de archivos.
- Escribir datos en archivos.
- Establecer y obtener la hora y fecha del sistema.

Instrucción INT

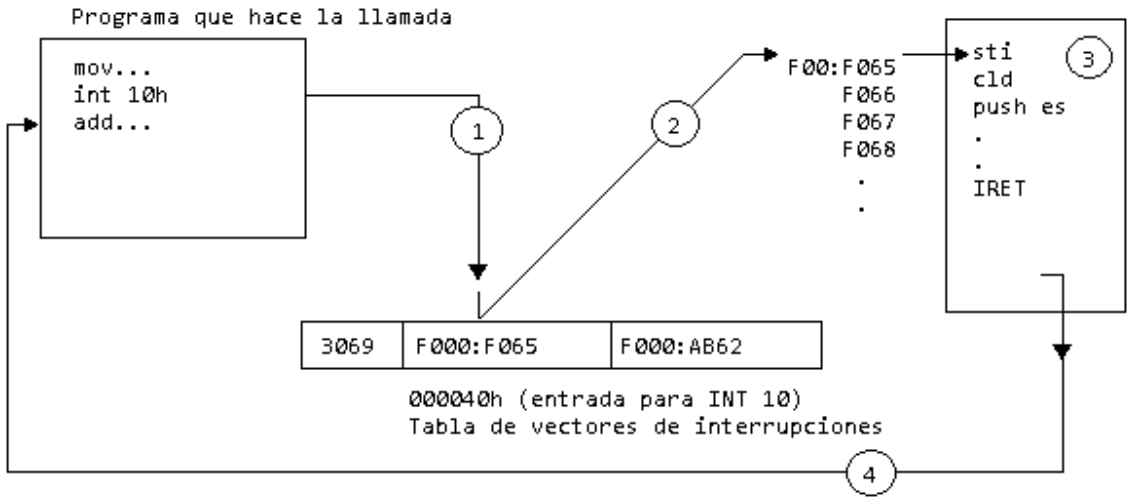
La instrucción INT (*llamada a un procedimiento de interrupción*) llama a una subrutina del sistema que también se conoce como manejador *de interrupciones*. Antes de que se ejecute una instrucción INT, deben insertarse uno o más parámetros en los registros. Por lo menos, debemos mover al registro AH un número que identifique al procedimiento específico. Dependiendo de la función, tal vez haya que pasar otros valores a la interrupción en los registros. La sintaxis es:

```
INT número
```

en donde *número* es un entero en el rango de 0 a FF hexadecimal.

Vectorización de interrupciones

La CPU procesa las instrucciones INT mediante el uso de la tabla de vectores de interrupciones que, como mencionamos antes, es una tabla de direcciones que se encuentra en los 1024 bytes inferiores de memoria. Cada entrada en esta tabla es una dirección de segmento-desplazamiento de 32 bits, que apunta a un manejador de interrupciones. Las direcciones reales en esta tabla varían de un equipo a otro. La siguiente figura ilustra los pasos que realiza la CPU cuando un programa invoca a la instrucción INT:



- **Paso 1**: el operador de instrucción INT se multiplica por 4 para localizar la entrada en la tabla con el vector de interrupción correspondiente.
- **Paso 2**: la CPU mete las banderas y una dirección de retorno de segmento/desplazamiento de 32 bits en la pila, deshabilita las interrupciones de hardware y ejecuta una llamada lejana a la dirección almacenada en la ubicación (10h*4) en la tabla de vectores de interrupción (F000:F065).
- **Paso 3**: el manejador de interrupciones en F000:F065 se ejecuta hasta llegar a una instrucción IRET.
- **Paso 4:** la instrucción IRET (retorno de interrupción) saca las banderas y la dirección de retorno de la pila, lo cual provoca que el procesador continúe la ejecución justo después de la instrucción INT 10h, en el programa que hizo la llamada.

Interrupciones comunes

Las interrupciones de software llaman a *rutinas de servicios de interrupciones* (ISRs), que se encuentran en el BIOS o en DOS. Algunas interrupciones de uso frecuente son:

- **INT 10h** (Servicios de video). Procedimientos que muestran rutinas que controlan la posición del cursor, escriben texto a color, desplazan la pantalla y muestran gráficos de video.
- **INT 16h** (Servicios de teclado). Procedimientos que leen el teclado y comprueban su estado.

- **INT 17h** (Servicios de impresora). Procedimientos que inicializan, imprimen y devuelven el estado de la impresora.
- **INT 1Ah** (Hora del día). Procedimiento que obtiene el número de pulsaciones del reloj desde que se encendió el equipo, o establece el contador a un nuevo valor.
- **INT 1Ch** (Interrupción de temporizador del usuario). Un procedimiento vació que se ejecuta 18.2 veces por segundo.
- **INT 21h** (Servicios de MS-DOS), procedimientos que proporcionan entrada-salida, manejo de archivos y administración de memoria. También se conocen como *llamadas a funciones de MS-DOS*.

Codificación para los programas de 16 bits

Los programas diseñados para MS-DOS deben ser aplicaciones de 16 bits que se ejecutan en modo de direccionamiento real. Las aplicaciones en modo de direccionamiento real utilizan segmentos de 16 bits y siguen el esquema de direccionamiento segmento descrito al principio del resumen. Si se utiliza un procesador de 32 bits, podemos usar registros de propósito general de 32 bits para datos, incluso en modo de direccionamiento real. He aquí un resumen de las características de codificación en los programas de 16 bits:

- La directiva .MODEL especifica el modelo de memoria que utilizará el programa. Se recomienda el modelo pequeño (Small), el cual mantiene el código en un segmento y la pila más los datos en otro:
  ```
  .MODEL small
  ```
- La directiva STACK asigna una pequeña cantidad de espacio en la pila local para nuestro programa. Por lo general, muy pocas veces se necesitan más de 256 bytes de espacio en la pila. La siguiente instrucción es bastante generosa, con 512 bytes:
  ```
  .STACK 200h
  ```
- De manera opcional, podemos habilitar el uso de registros de 32 bits. Esto puede hacerse con la directiva .386:
  ```
  .386
  ```
- Se requieren dos instrucciones al principio de main si el programa hace referencia a variables. Estas instrucciones inicializan el registro DS con la ubicación inicial del segmento de datos, identificado por la constante predefinida @data de MASM.
  ```
  mov ax,@data
  mov ds,ax
  ```
- Todo programa debe incluir una instrucción para terminar el programa y regresar al sistema operativo. Una forma de hacerlo es usando la directiva .EXIT:
  ```
  .EXIT
  ```
  De manera alternativa, podemos llamar a la función 4Ch de INT21h:
  ```
  mov ah,4ch                ; termina el proceso
  int 21h                   ; interrupciones de MS-DOS
  ```
- Podemos asignar valores a los registros de segmento mediante la instrucción MOV, pero sólo cuando se asigne la dirección de un segmento del programa.
- Los programas en modo direccionamiento real sólo pueden acceder a los puertos de hardware, los vectores de interrupción y la memoria del sistema cuando se ejecutan

bajo MS-DOS, Windows 95, 98 y Milenium. Este tipo de acceso no está permitido en Windows NT, 2000 o XP.

- Cuando se utiliza el modelo **Small** de memoria, los desplazamientos (direcciones) de los datos y las etiquetas de código son de 16 bits. La biblioteca Irvine16 utiliza el modelo Small de memoria, en el que todo el código cabe en un segmento de 16 bits, y los datos y la pila del programa caben en otro segmento de 16 bits.
- En el modo de direccionamiento real, las entradas en la pila son de 16 bits de manera predeterminada. De cualquier forma podemos colocar un valor de 32 bits en la pila (utiliza dos entradas).

Podemos simplificar la codificación de los programas de 16 bits sin incluimos el archivo Irvine32.inc. Este archivo inserta las siguientes instrucciones en el flujo de ensamblado, las cuales definen el modo de memoria y la convención de llamadas, asignan espacio en la pila, habilitan los registros de 32 bits y redefinen la directiva .EXIT con **exit**:

```
.MODEL small, stdcall

.STACK 200h

.386

exit EQU <.EXIT>
```

Llamadas a funciones de MS-DOS (INT 21h)

MS-DOS proporciona muchas funciones fáciles de usar para mostrar texto en la consola. Todas forman parte de un grupo que, por lo general se conoce como *llamadas a funciones INT 21h de MS-DOS*. Esta interrupción soporta por lo menos 200 funciones distintas, las cuales se identifican mediante un número de función que se coloca en el registro AH. Una fuente excelente, aunque un poco obsoleta, es el libro de Ray Duncan, *Advanced MS-DOS programming*, 2º Edición, Microsoft Press, 1988. Una lista más extensa y actualizada, conocida como Ralf Brown's Interrupt List, se encuentra en línea .

Para cada función INT 21h que describimos en este capítulo, presentamos los parámetros de entrada y valores de retorno necesarios, agregamos notas acerca de su uso e incluiremos un corto ejemplo de código para llamarla.

Varias de las funciones requieren que la dirección de 32 bits de un parámetro de entrada se almacene en los registros DS:DX. DS, es el registro del segmento de datos, por lo general, se establece con el área de datos del programa. Si por alguna razón no es el caso, hay que usar el operador SEG para establecer DS con el segmento que contiene los datos que se pasan a INT 21h. Las siguientes instrucciones se encargan de esto:

```
.data

buferEnt BYTE 80 DUP(?)

.code
```

```
mov ax, SEG buferEnt

mov ds,ax

mov dx,OFFSET buferEnt
```

Función 4Ch de INT 21h: terminar proceso La función 4Ch de INT 21h termina el programa actual (conocido como proceso). En los programas en modo direccionamiento real que presentamos, nos hemos basado en la definición de un macro en la biblioteca Irvine16 llamada **exit**. Ésta se define así:

exit TEXTEQUI <.EXIT>

En otras palabras, **exit** es un alías o sustituto para .EXIT (la directiva de MASM que termina un programa). Se creó el símbolo **exit** con el fin de poder usar un solo comando para terminar los programas de 16 y 32 bits. En programas de 16 bits, el código generado por **.EXIT** es:

```
mov ah,4Ch                    ; termina el proceso

int 21h
```

Si proporcionamos un argumento de código de retorno opcional a la macro .EXIT, el ensamblador genera una instrucción adicional que mueve el código de retorno a AL:

```
.EXIT 0                       ; llama a la macro
```

Código generado:

```
mov ah,4Ch                    ; termina el proceso

mov al,0                      : código de retorno

int 21h
```

El proceso que hace la llamada recibe el valor en AL, conocido como el *código de retorno del proceso* (incluyendo un archivo por lotes), para indicar el estado de retorno del programa. Por convención, un código de retorno de cero se considera que se completó con éxito. Podemos usar otros códigos de retorno entre 1 y 255 para indicar resultados adicionales que tengan un significado específico para cada programa. Por ejemplo, ML.EXE, Microsoft Assambler, devuelve 0 si un programa se ensambla en forma adecuada, y un valor distinto de cero su no es así.

Funciones de salida selectas

En esta sección presentaremos algunas de las funciones más comunes de INT 21h para escribir caracteres y texto. Ninguna de estas funciones altera los colores actuales predeterminados de la pantalla, por lo que los resultados sólo estarán a colores si se estableció previamente el color de la pantalla por otros medios. Por ejemplo, podemos llamar a las funciones del BIOS de video.

Filtrado de caracteres de control Todas las funciones en esta sección *filtran*, o interpretan caracteres ASCII de control. Por ejemplo, si escribimos un carácter de retroceso en la salida estándar, el cursor se mueve una columna a la izquierda. La siguiente tabla contiene una lista de caracteres de controlo que es probable que nos encontremos.

Caracteres ASCII de control

Código ASCII	Descripción
08h	Retroceso (mueve una columna a la izquierda)
09h	Tabulación horizontal (avanza *n* columnas hacia adelante)
0Ah	Avance de línea (se mueve la siguiente línea de salida)
0Ch	Avance de página (se mueve a la siguiente página de impresión)
0Dh	Retorno de carro (se mueve a la columna de salida que está más a la izquierda)
1Bh	Carácter de escape

Las siguientes tablas describen las características importantes de las funciones 2, 5, 6, 9 y 40h de INT 21h. La función 2 de INT 21h escribe un solo carácter en la salida estándar; la función 5 de INT 21h escribe un solo carácter en la impresora; la función 6 de INT21 escribe un solo carácter sin filtro a la salida estándar; la función 9 de INT 21h escribe una cadena (que se termina con un carácter $) a la salida estándar; y La función 40h de INT 21h escribe un arreglo de bytes en un archivo o dispositivo.

Función 2 de INT 21h

Descripción	Escribe un solo carácter en la salida estándar y avanza el cursor una columna hacia adelante
Recibe	AH = 2 DL = valor de carácter
Devuelve	Nada
Llamada de ejemplo	`mov ah,2`

	mov dl,'A' int 21h

Función 5 de INT 21h

Descripción	Escribe un solo carácter en la impresora
Recibe	AH = 5 DL = valor de carácter
Devuelve	Nada

Llamada de ejemplo	mov ah,5 ; selecciona la salida de la impresora
	
	mov dl,"2" ; carácter a imprimir
	
	int 21h ; llamada a MS-DOS

Notas	MS-DOS espera hasta que la impresora esté lista para imprimir el carácter. Podemos terminar la espera oprimiendo las teclas Ctrl-Enter. La salida predeterminada es al puerto de impresora LP1

Función 6 de INT 21h

Descripción	Escribe un carácter en la salida estándar
Recibe	AH = 6 DL = valor del carácter
Devuelve	Si ZF = 0, AL contiene el código ASCII del carácter

Llamada de ejemplo	mov ah,6
	
	mov dl,"A"

	`int 21h`
Notas	A diferencia de otras funciones de INT 21, ésta no filtra (interpreta) los caracteres ASCII de control

Función 9 de INT 32h

Descripción	Escribe una cadena con terminación $ en la salida estándar
Recibe	AH = 9 DS:DX = segmento/desplazamiento de la cadena
Devuelve	Nada

Llamada de ejemplo	`.data`
	`cadena BYTE “esta es una cadena$”`
	`.code`
	`mov ah,9`
	`mov dx,OFFSET cadena`
	`int 21h`

Notas	La cadena debe terminar con un carácter de signo de dólares ($)

Función 40h de INT 21h

Descripción	Escribe un arreglo de bytes en un archivo o dispositivo
Recibe	AH = 40h BX = manejador de dispositivo o archivo (consola = 1) CX = número de bytes a escribir DS:DX = dirección de arreglo

Devuelve	AX = número de bytes escritos

Llamada de ejemplos	.data
	mensaje “hola, programador”
	.code
	mov ah,40h
	mov bx,1
	mov cx,LENGTHOF mensaje
	mov dx,OFFSET mensaje
	int 21h

Ejemplo de programa: Hola programador

A continuación se muestra un programa de ejemplo, para imprimir una cadena en la pantalla usando una llamada a una función de MS-DOS

```
TITLE Programa Hola programador (Hola.asm)

.MODEL small

.STACK 100h

.386

.data

mensaje BYTE “Hola, programador!”,0dh,0ah

.code

main PROC

   mov as,@data                ; inicializa DS

   mov ds,ax

   mov ah,40                   ; escribe en el archivo/dispositivo

   mov  bx,1                   ; manejador de salida

   mov cx,SIZEOF mensaje       ; números de bytes

   mov dx,OFFSET mensaje       ; dirección del búfer
```

```
   int 21h
   .EXIT
main ENDP
END main
```

Versión alternativa Otra forma de escribir Hola.asm es mediante el uso de la directiva predefinida .STARTUP (la cual inicializa el registro DS). Para ello, hay que eliminar la etiqueta que está seguida de la directiva END:

```
TITLE programa Hola programador (Hola2.asm)
.MODEL small
.STACK 100h
.386
.data
mensaje “Hola, Programador!”, 0dh, 0ah
.code
main PROC
   .STARTUP
   mov ah,40h                  ; escribe en el archivo/dispositivo
   mov bx,1                    ; manejador de salida
   mov cx,SIZEOF mensaje       ; número de bytes
   mov dx,OFFSET mensaje       ; dirección del bufer
   int 21h
   .EXIT
main ENDP
END
```

Funciones de entrada selectas

En esta sección, describiremos unas cuantas de las funciones de MS-DOS que se utilizan con más frecuencia para leer de la entrada estándar.

Función 1 de INT 21h

Descripción	Lee un solo carácter de la entrada estándar
Recibe	AH = 1
Devuelve	AL = carácter (código ASCII)

Llamada de ejemplo	`mov ah,1`
	`int 21h`
	`mov car,al`

Notas	Si no hay un carácter presente en el búfer de entrada, el programa espera. Esta función envía (echo) el carácter de salida estándar

La función 6 de INT 21h lee un carácter de la entrada estándar, si éste está esperando en el búfer de entrada. Si el búfer está vacío, la función regresa con la bandera Cero activa y no realiza ninguna otra acción:

Función 5 de INT 21h

Descripción	Lee un carácter de la entrada estándar sin esperar
Recibe	AH = 6 DL = FFh
Devuelve	Si ZF = 0, Al contiene el código ASCII del carácter

Llamada de ejemplo	mov ah,6
	mov dl,0FFh
	int 21h
	mov car,al
	saltar:

Notas	La interrupción sólo devuelve un carácter si hay uno esperando en el búfer de entrada. No envía (echo) el carácter a la salida estándar y no filtra los caracteres de

	control

La función 0Ah de INT 21h lee una cadena en búfer de la entrada estándar, la cual se termina con la tecla Intro. Al llamar a esta función. hay que pasarle un apuntador a una estructura de entrada que tenga el siguiente formato (**cuenta** puede ser entre 0 y 128):

```
cuenta = 80

TECLADO STRUCT

   maxEntrada  BYTE cuenta     ; caracteres máximos a introducir

   cuentaEntrada BYTE ?        ; cuenta actual de entrada

   búfer BYTE cuenta DUP(?)    ; guarda los caracteres de entrada

TECLADO ENDS
```

El campo *maxEntrada* especifica el número máximo de caracteres que puede introducir el usuario, incluyendo la tecla Intro. Podemos utilizar la tecla de retroceso para borrar caracteres y retroceder el cursor. El usuario termina la entrada oprimiendo la tecla Intro u oprimiendo Ctrl-Enter. Todas las teclas que no sean ASCII, como AvPág y F1, se filtran y no se almacenan en el búfer, una vez que la función regresa, el campo *cuentaEntrada* indica cuantos caracteres se introdujeron, sin contar la tecla Intro. La siguiente tabla describe la función 0Ah:

Instrucción 0Ah de INT 21h

Descripción	Lee un arreglo de caracteres en búfer de la entrada estándar
Recibe	AH = 0Ah DS:DX = dirección de la estructura de entrada del teclado
Devuelve	La estructura se inicializa con los caracteres de entrada

Llamada de ejemplo	`.data`
	
	`datosTecl TECLADO <>`
	
	`.code`

	mov ah,0Ah mov dx,OFFSET datosTecl int 21h

La función 0Bh de INT 21h obtiene el estado del búfer de entrada estándar:

Función 0B de INT 21h

Descripción	Obtiene el estado del búfer de entrada estándar
Recibe	AH = 0Bh
Devuelve	Si hay un carácter esperando, AL = 0FFh; en cualquier otro caso, AL = 0

Llamada de ejemplo	mov ah,0Bh
	int 21h
	cmp al,0
	je saltar
	; (introduce un carácter)
	saltar:

Notas	No elimina el carácter

Ejemplo: programa de cifrado de cadenas

La función 6 de INT 21h tiene la habilidad única de leer caracteres de la entrada estándar sin detener el programa o filtrar los caracteres de control. A esto se puede dar un buen uso, si ejecutamos un programa desde el símbolo de sistema y redirigimos la entrada. Es decir, la entrada provendrá de un archivo de texto, en vez del teclado.

El siguiente programa (*Cifrar.asm*) lee cada carácter de la entrada estándar, usa la instrucción XOR para alterar el carácter y escribe el carácter alterado en la salida estándar:

```
TITLE Programa de cifrado      (Cifrar.asm)

; Este programa usa llamadas a funciones de MS-DOS
```

```
; para leer y cifrar un archivo. Se ejecuta desde el
; símbolo del sistema, usando la redirección:
; Cifrar <archent.txt> archsal.txt
; la función 6 también se usa para salida, para evitar
; filtrar los caracteres ASCII de control.
INCLUDE Irvine16.inc
VALXOR = 239                    ; cualquier valor entre 0 – 255
.code
main PROC
   mov ax,@data
   mov ds,ax
L1:
   mov ah,6                     ; dirige la entrada de consola
   mov dl,0FFh                  ; no espera al carácter
   int 21h                      ; AL = carácter
   jz L2                        ; termina si ZF = 1 (EOF)
   xor al,VALORX                ; escribe en la salida
   mov ah,6
   mov dl,al
   int 21h
   jmp L1                       ; repite el ciclo
L2: exit
main ENDP
END main
```

La elección de 239 como valor de cifrado es completamente arbitraria. Podemos usar cualquier valor entre 0 y 255 en este contexto, aunque si se utiliza 0 no produciríamos ningún cifrado. Desde luego que el cifrado es débil, pero podría ser suficiente como para desalentar al

usuario promedio al tratar de manejarlo. Cuando se ejecuta este programa en el símbolo de sistema, debemos indicar el archivo de entrada (y salida, si lo hay). A continuación se muestran dos ejemplos:

cifrar < archent.txt	Entrada de archivo (archent.txt), salida a la consola
cifrar < archent.txt > archsal.txt	Entrada desde el archivo (archent.txt), salida a un archivo (archsal.txt)

Función 3Fh de int 21h

La función 3Fh de INT 21h, como se muestra en la siguiente tabla, lee un arreglo de bytes de un archivo o dispositivo. Puede usarse para la entrada de teclado cuando el manejador del dispositivo en BX es igual a cero:

Función 3F de INT 21h

Descripción	Lee un arreglo de bytes en un archivo o dispositivo
Recibe	AH = 3Fh BX = manejador de dispositivo/archivo (0 = teclado) CX = máxime de bytes a leer DS:DX = dirección de búfer de entrada
Devuelve	AX = número de bytes que se leyeron

Llamada de ejemplo	`.data`
	`buferEntrada BYTE 127 dup(0)`
	`bytesLeidos WORD ?`
	`.code`
	`mov ah,3Fh`
	`mov bx,0`
	`mov cx,127`
	`mov dx,OFFSET buferEntrada`
	`int 21h`

	`mov bytesLeidos,ax`
Notas	Si se lee desde el teclado, la entrada termina cuando se oprime la tecla Intro, y se adjuntan los caracteres 0Dh, 0Ah al búfer de entrada

Si el usuario introduce más caracteres de los que solicita la llamada a la función, los caracteres en exceso permanecen en el búfer de entrada de MS-DOS. Si la función se llama más adelante en el programa, la ejecución tal vez no se detenga y espere la entrada del usuario, ya que el búfer todavía contiene datos (incluyendo los caracteres 0Dh, 0Ah que marcan el final de la línea). Esto puede ocurrir incluso entre instancias separadas de ejecución del programa. Para estar completamente seguro de que nuestro programa funciona en la forma separada, debemos vaciar el búfer de entrada, un carácter a la vez, después de llamar a la función 3Fh. El siguiente código se encarga de ello:

```
:-----------------------------------

VaciarBufer PROC

;

;  vacía el búfer de la entrada estándar.

; Recibe: nada. Devuelve: nada

;-------------------------------------

.data

unByte BYTE ?

.code

   pusha

L1:

   mov ah,3Fh                  ; lee archivo/dispositivo

   mov bx,0                    ; manejador del teclado

   mov cx,1                    ; un byte

   mov dx,OFFSET unByte        ; lo guarda aquí

   int 21h                     ; llama a MS-DOS

   cmp unByte,0Ah              ; ¿llegó al fin de línea?
```

```
    jne L1                      ; no: lee otro
    popa
    ret
VaciarBufer ENDP
```

Funciones de fecha/hora

Muchas aplicaciones populares de software muestra la fecha y hora actuales. Otras obtienen la fecha y hora, y las utilizan en su lógica interna. Por ejemplo, un programa de agenda puede usar la fecha actual para verificar que un usuario no esté programando por accidente una cita en el pasado.

Como se muestra en las siguientes series de tablas, la función 2Ah de INT 21h obtiene la fecha del sistema, y la función 2B de INT 21h, establece la fecha del sistema. La función 2C de INT 21h obtiene la hora del sistema, y la función 2D de INT 21h la establece.

Función 2Ah de INT 21h

Descripción	Obtiene la fecha del sistema
Recibe	AH = 2Ah
Devuelve	CX = año DH,DL = mes, día AL = día de la semana (Domingo = 0, Lunes = 1, etc.)

Llamada de ejemplo	`mov ah, 2Ah`
	`int 21h`
	`mov anio, cx`
	`mov mes, dh`
	`mov dia, dl`
	`mov diaSemana, al`

Función 2Bh de INT 21h

Descripción	Establece la fecha del sistema
Recibe	AH = 2Bh CX = año DH = mes DL = día
Devuelve	Si el cambio tuvo éxito, AL = 0; en caso contrario, AL = FFh

Llamada de ejemplo	`mov ah,2Bh`
	`mov cx, anio`
	`mov dh, mes`
	`mov dl, dia`
	`int 21h`
	`cmp al,0`
	jne fallo

Notas	Es probable que no funcione si se ejecuta en Windows NT, 2000 o XP con un perfil de usuario restringido.

Función 2Ch de INT 21h

Descripción	Obtiene la hora del sistema
Recibe	AH = 2Ch
Devuelve	CH = horas (0-23) CL = minutos (0-59) DH = segundos (0-59) DL = centésimas de segundos (por lo general, no son precisas)
Llamada de ejemplo	`mov ah, 2Ch` `int 21h`

	mov horas, ch
	mov minutos, cl
	mov segundos, dh

Función 2D de INT 21h

Descripción	Establece la hora del sistema
Recibe	AH = 2Dh CH = horas (0-23) CL = minutos (0-59) DH = segundos (0-59)
Devuelve	Si el cambio tuvo éxito, AL = 0; en caso contrario, AL = FFh

Llamada de ejemplo	mov ah, 2Dh
	mov ch, horas
	mov cl, minutos
	mov dh, segundos
	int 21h
	cmp al,0
	jne fallo

Notas	Es probable que no funcione si se ejecuta en Windows NT, 2000 o XP con un perfil de usuario restringido

Servicios estándar de E/S de archivos de MS-DOS

Las funciones INT 21h proporcionan más servicios de E/S de archivos y directorios de las que se mostrarán en este resumen. La siguiente tabla muestra unas cuantas de las funciones que se utilizan con más frecuencia.

Funciones INT 21h relacionadas con archivos y directorios.

Función	Descripción
716Ch	Crear o abrir un archivo
3Eh	Cerrar el manejador del archivo
42h	Mover el apuntador de un archivo
5706h	Obtener la fecha y hora de creación del archivo

Manejadores de archivos/dispositivos MS-DOS y MS Windows utilizan enteros de 16 bits conocidos como *manejadores* para identificar a los archivos y dispositivos de E/S. Hay cinco manejadores de dispositivos predefinidos. Cada uno, excepto el manejador 2 (salida de error), soporta la dirección en el símbolo de sistema. Los siguientes manejadores están disponibles todo el tiempo:

0 Teclado (entrada estándar)
1 Consola (salida estándar)
2 Salida de error
3 Dispositivo auxiliar (asíncrono)
4 Impresora

Cada función de E/S tiene una característica común: si falla se activa la bandera Acarreo, y se devuelve un código de error en AX. podemos usar este código de error para mostrar un mensaje apropiado. La siguiente tabla contiene una lista de los códigos de error y sus descripciones:

Microsoft proporciona una gran cantidad de documentación, en relación con las llamadas a funciones de MS-DOS.

Códigos de error extendidos de MS-DOS.

Función	Descripción
01	Número de función inválido
02	No se encontró el archivo
03	No se encontró la ruta
04	Demasiados archivos abiertos (no hay más manejadores)
05	Acceso denegado

06	Manejador inválido
07	Se destruyeron los bloques de control de la memoria
08	Memoria insuficiente
09	Dirección de bloque de memoria inválida
0A	Entorno inválido
0B	Formato inválido
0C	Código de acceso inválido
0D	Datos inválidos
0E	Reservado
0F	Se específico una unidad inválida
10	Intento de eliminar el directorio actual
11	No es el mismo dispositivo
12	No hay más archivos
13	El disco flexible está protegido contra escritura
14	Unidad desconocida
15	La unidad no está lista
16	Comando desconocido
17	Error de datos (CRC)
18	Longitud de estructura de petición incorrecta
19	Error de búsqueda

Crear o abrir un archivo (716Ch)

La función 716Ch de INT 21h puede crear un nuevo archivo o abrir uno existente. Permite el uso de nombres de archivos extendidos y la compartición del archivo. Como se muestra en la siguiente tabla, el nombre de archivo puede incluir de manera opcional una ruta de directorio.

Función 716Ch de INT 21h

Descripción	Crea un nuevo archivo o abre uno existente
Recibe	AX = 716Ch BX = modo de acceso (0 = lectura, 1 = escritura, 2 = lectura/escritura) CX = atributos (0 = normal, 1 = solo lectura, 2 = oculto, 3 = sistema, 8 = ID de volumen, 20h = archivo) DX = acción (1 = abrir, 2 = truncar, 10h = crear) DS:SI = segmento/desplazamiento de nombre del archivo DI = sugerencia de alías (opcional)
Devuelve	Si la creación/apertura tuvo éxito, CF = 0, AX = manejador del archivo y CX = acción realizada. Si la creación falló, CF = 1

Llamada de ejemplo	`mov ax, 716Ch ; abrir/crear extendido`
	
	`mov bx, 0 ; sólo lectura`
	
	`mov cx, 0 ; atributo normal`
	
	`mov dx,1 ; abre archivo existente`
	
	`mov si, OFFSET Nombrearchivo`
	
	`int 21h`
	
	`jc fallo`
	
	`mov manejador, ax ; manejador del archivo`
	
	`mov accionRealizada, cx ; acción realizada`

Notas	El modo de acceso en BX puede combinarse de manera opcional con uno de los siguientes valores de modo de compartición: OPEN_SHARE_COMPATIBLE, OPEN_SHARE_DENYREADWRITE, OPEN_SHARE_DENYWRITE. OPEN_SHARE_DENYREAD, OPEN_SHARE_DENYNONE. La acción realizada que se devuelve en CX puede ser uno de los siguientes valores: ACTION_OPENED, ACTION_CREATED_OPENED, ACTION_REPLACED_OPENED. Todas estas constantes están definidas en Irvine16.inc

Ejemplos adicionales El siguiente código crea un archivo o trunca uno existente que tenga el mismo nombre:

```
mov ax, 716Ch                   ; Abrir/Crear extendida

mov bx,2                        ; lectura – escritura

mov cx,0                        ; atributo normal

mov dx, 10h + 02h               ; acción crear + truncar

mov si, OFFSET ArchivoNuevo

int 21h

jc fallo

mov manejador,ax                ; manejador de archivo

mov accionRealizada, cx         ; acción realizada para abrir el archivo
```

Cerrar manejador de archivo (3Eh)

La función 3Eh de INT 21h cierra un manejador de archivo. Esta función vacía el búfer de escritura del archivo, copiando cualquier información restante al disco, como se muestra en la siguiente tabla:

Función 3Eh de INT 21h

Descripción	Cierra el manejador de archivo
Recibe	AH = 3E BX = manejador del archivo
Devuelve	Si el archivo se cerró con éxito, CF = 0; en caso contrario, CF = 1

Llamada de ejemplo	`.data`
	
	`manejadorarchivo WORD ?`
	
	`.code`
	
	`mov ah,3Eh`
	
	`mov bx, manejadorarchivo`

	`int 21h`
	
	`jc fallo`

Notas	Si el archivo se modificó, se actualiza su estampa de fecha y su estampa de hora

Mover apuntador de archivo (42h)

La función 42h de INT 21h, como puede verse en la siguiente tabla, mueve el apuntador de posición de un archivo abierto a una nueva ubicación. Al llamar a esta función, el *código de método* en AL, identifica la forma en que se va a establecer el apuntador:

0 Desplazamiento a partir del principio del archivo

1 Desplazamiento a partir de la ubicación actual

2 Desplazamiento a partir del final del archivo

Función 42h de INT 21h

Descripción	Mueve el apuntador del archivo
Recibe	AH = 42h AL = código del método BX = manejador del archivo CX:DX = valor de desplazamiento de 32 bits
Devuelve	Si el apuntador del archivo se movió con éxito, CF = 0 y DX:AX regresa el nuevo desplazamiento del apuntador del archivo, en caso contrario, CF = 1.

Llamada de ejemplo	`mov ah,42h`
	
	`mov al,0 ; método: desplazamiento desde el principio`
	
	`mov bx, manejador`
	
	`mov cx, desplazamientoSup`
	
	`mov dx, desplazamientoInf`

	`int 21h`
Notas	El desplazamiento devuelto del apuntador del archivo en DX:AX siempre es relativo al principio del archivo

Obtener la fecha y hora de la creación de un archivo

La función 5706h de INT 21h, que se muestra en la siguiente tabla, obtiene la fecha y hora de cuando se creó un archivo. No es necesariamente la misma fecha y hora de la última modificación del archivo, o incluso de su último acceso.

Función 5706 de INT 21h

Descripción	Obtiene la fecha y hora de creación de un archivo
Recibe	AX = 5706h BX = manejador del archivo
Devuelve	Si la llamada a la función fue exitosa, CF = 0, DX = fecha (en formato empaquetado de DOS), CX = hora y SI = milisegundos. Si la función falla, CF = 1

Llamada de ejemplo	`mov ax, 5706h ; Obtiene fecha/hora de creación`
	
	`mov bx, manejador`
	
	`int 21h`
	
	`jc error ; termina si falla`
	
	`mov fecha, dx`
	
	`mov hora, cx`
	
	`mov milisegundos, si`

Notas	El archivo debe estar abierto. El valor *milisegundos* indica el número de 10 milisegundos que se van a agregar a la hora de MS-DOS. El rango es de 0 a 199, indicando que el campo puede agregar hasta 2 segundos al tiempo total

Procedimientos de biblioteca selectos

A continuación se muestra dos procedimientos de la biblioteca de vínculos Irvine16: **ReadString** y **WriteString**. **ReadString** es el más engañoso de los dos, ya que debe leer un carácter a la vez hasta encontrar el carácter de fin de línea (0Dh). Lee el carácter pero no lo copia al búfer.

ReadString

El procedimiento **ReadString** lee una cadena de la entrada estándar y coloca los caracteres en un búfer de entrada, como una cadena con terminación nula. Termina cuando el usuario oprime Intro:

```
;---------------------------------------

ReadString PROC

; Recibe:    DS:DX apunta al búfer de entrada,

 ;           CX = máximo de caracteres de entrada

 ; Devuelve: AX = tamaño de la cadena de entrada.

; Comentarios: Se detiene cuando se oprime Intro (0Dh).

;              o cuando se leen (CX-1) caracteres.

; ---------------------------------------

push cx                        ; guarda los registros

push si

push cx                        ; guarda la cuenta de dígitos otra vez

mov si, dx                     ; apunta al búfer de entrada

dec cx                         ; guarda espacio para el byte nulo

L1: mov ah, 1                  ; función: entrada del teclado

int 21h                        ; DOS devuelve el carácter en AL

cmp al,0DH                     ; ¿fin de línea?

je L2                          ; si: termina

mov [si],al                    ; no: guarda el carácter

inc si                         ; incrementa el apuntador al bufer
```

```
loop L1                         ; itera hasta que CX = 0
L2: mov byte ptr [si], 0        ; termina con un byte nulo
pop ax                          ; cuenta de dígitos original
sub ax, cx                      ; AX = tamaño de la cadena de entrada
dec ax
pop si
pop cx
ret
ReadString ENDP
```

WriteString

El procedimiento **WriteString** escribe una cadena con terminación nula en la salida estándar. Llama a un procedimiento ayudante llamado **Str_length**, el cual devuelve el número de bytes en una cadena:

```
; -----------------------------------------------
WriteString PROC
; Escribe una cadena con terminación nula en la salida estándar
; Recibe: DS:DX apunta a la cadena
; Devuelve: nada
; -----------------------------------------------
pusha
push ds                         ; ES = DS
pop es
mov di, dx                      ; ES:DI = apuntador a la cadena
call Str_length                 ; AX = longitud de la cadena
mov cx, ax                      ; CX = número de bytes
mov ah, 40h                     ; escribe al archivo o dispositivo
mov bx,1                        ; manejador de salida estándar
```

```
int 21h                          ; llamada a MS-DOS

popa

ret

WriteString ENDP
```

Leer la cola de comandos de MS-DOS

En los programas que mostramos a continuación, pasamos con frecuencia información a los programas en la línea de comandos. Supongamos que tenemos que pasar el nombre *archivo1.doc* a un programa llamado *attrib.exe*. La línea de comandos de MS-DOS sería:

```
attrib archivo1.doc
```

Cuando se inicia un programa, cualquier texto adicional en su línea de comando se almacena de manera automática en la *Cola de comandos de MS-DOS* de 128 bytes, la cual se ubica en la memoria, en desplazamiento 80h a partir del principio de la dirección de segmento especificada por el registro ES. Al área de memoria se le conoce como *prefijo de segmento de programa* (PSP).

El primer byte contiene la longitud de la línea de comandos. Si su valor es mayor que cero, el segundo byte contiene un carácter de espacio. El resto de los bytes contienen el texto escrito en la línea de comandos. Si utilizamos la línea de comandos de ejemplo para el programa *attrib.exe*, el contenido hexadecimal de la cola de comandos sería el siguiente:

80	81	82	83	84	85	86	87	88	89	8A	8B
0A	20	46	49	4C	45	31	2E	44	4F	43	0D
		F	I	L	E	1	.	D	O	C	

Podemos ver los bytes de la cola de comandos mediante el depurador CodeView de Microsoft, si cargamos el programa y establecemos los argumentos de la línea de comandos antes de ejecutar el programa.

Hay una excepción a la regla que establece que MS-DOS almacena todos los caracteres después del nombre del comando o del programa: no mantiene en uso los nombres de archivo y dispositivo al redirigir la entrada-salida. Por ejemplo, MS-DOS no guarda el texto en la cola de comandos cuando se escribe el siguiente comando, ya que tanto *archent.txt* como PRN se utilizan para la redirección:

```
prog1 < archent.txt > prn
```

Procedimiento GetCommandTail El procedimiento **GetCommandTail** de la biblioteca Irvine16 devuelve una copia de la cola de comandos del programa en ejecución bajo MS-DOS.

Al llamar a este procedimiento hay que agregar a DX el desplazamiento de búfer en el que se va a copiar la cola de comandos. A menudo, los argumentos en modo de direccionamiento real tratan directamente con los registros de segmento, para poder acceder a los datos en distintos segmentos de memoria. Por ejemplo. GetCommandTail guarda el valor actual de ES en la pila, obtiene el segmento PSP usando la función 62h de INT 21h y lo copia a ES:

```
push es

.

.

mov ah, 62h                    ; obtiene dirección de segmento PSP

int 21                         ; se devuelve en BX

mov es, bx                     ; se copia a ES
```

A continuación, localiza un byte dentro de PSP. Como ES no apunta al segmento de datos predeterminado del programa, debemos usar una *redefinición de segmento* (es:) para direccionar los datos dentro del prefijo de segmento del programa:

```
mov cl, es:[di–1]              ; obtiene byte de longitud
```

GetCommandTail omite los espacios a la izquierda con SACASB y establece la bandera Acarreo si la cola de comandos está vacía. Esto facilita que el programa que hace la llamada ejecute una instrucción JC (*salta si hay acarreo*) si no se escribe nada en la línea de comandos:

```
cld                            ; explora en dirección hacia adelante

mov al,20h                     ; carácter de espacio

repz scasb                     ; explora caracteres que no sean espacios

jz L2                          ; se encontraron solo espacios

.

.

.

L2: stc                        ; CF = 1 significa que no hay cola de
comandos
```

SCASB explora de manera automática la memoria a la que apuntan los registros de segmento ES, por lo que no tuvimos más opción que asignar a ES el segmento PSP al principio de GetCommandTail. He aquí un listado completo:

```
; ----------------------------------------------------
```

```
GetCommandTail PROC
;
; Obtiene una copia de la cola de comandos de MS-DOS en PSP:80h.
; Recibe: DX contiene el desplazamiento del búfer
;   que recibe una copia de la cola de comandos.
; Devuelve: CF = 1 si el búfer está vacío; en cualquier
; otro caso, CF = 0.
; -----------------------------------------------------
SPACE = 20h
   push es
   pusha                        ; guarda los registros generales
   mov ah,62h                   ; obtiene la dirección de segmento PSP
   int 21h                      ; se devuelve en BX
   mov es, bx                   ; se copia a ES
   mov si, dx                   ; apunta al búfer
   mov di,81h                   ; desplazamiento en PSP de la cola de
comandos
   mov cx,0                     ; cuenta de bytes
   mov cl, es:[di-1]            ; obtiene byte de longitud
   cmp cx,0                     ; ¿está vacía la cola?
   je L2                        ; si: termina
   cld                          ; explora en dirección hacia adelante
   mov al, SPACE                ; carácter de espacio
   repz scasb                   ; explora caracteres que no sean espacios
   jz L2                        ; se encontró un carácter de espacio
   dec di                       ; no se encontró un carácter de espacio
   inc cx
```

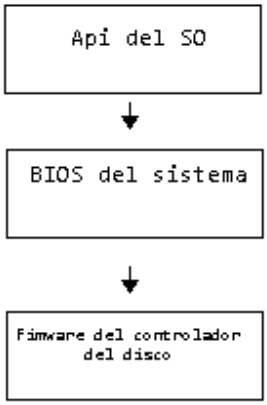
De manera predeterminada, el ensamblador asume que DI es un desplazamiento a partir de la dirección de segmento en DS. La redefinición del segmento (es:[di]) indica a la CPU que debe utiliza mejor la dirección de segmento ES.

```
L1: mov al, es:[di]            ; copia la cola al búfer

   mov [si], al                ; al que apunta DS:SI

   inc si

   inc di

   loop L1

   clc                         ; CF = significa que se encontró la cola

   jmp L3

L2: stc                        ; activa acarreo: no hay cola de comandos

L3: mov byte ptr [si],0        ; almacena byte nulo

   popa                        ; restaura los registros

   pop es

   ret

GetCommandTail ENDP
```

14 Fundamentos de los discos

Sistema de almacenamiento en disco

En este capítulo presentaremos los fundamentos de los sistemas de almacenamiento en disco. También trataremos cómo se relaciona el almacenamiento en disco con el almacenamiento en disco a nivel de BIOS en las computadoras basadas en Intel. Por último, mostraremos cómo interactúa MS Windows con los programas de aplicación para proporcionar el acceso a los archivos y directorios. La interacción entre los niveles virtuales de una computadora se vuelve aparente si se considera el almacenamiento en disco:



- En nivel bajo se encuentra el *firmware del controlador de disco*, el cual utiliza chips controladores inteligentes para crear un mapa de la geometría del disco (ubicaciones físicas), para las marcas y modelos específicos de unidades de disco.
- En el siguiente nivel se encuentra el BIOS del sistema, el cual proporciona una colección de bajo nivel de funciones que utilizan los sistemas operativos para realizar tareas como lectura de sectores, escritura en sectores y formato de pistas.
- En el siguiente nivel más alto se encuentra la API del *sistema operativo*, la cual proporciona una colección de funciones de la API que ofrece servicios como abrir y cerrar archivos, establecer las propiedades de los archivos, leer archivos y escribir en ellos.

Todos los sistemas de almacenamiento en disco tienen ciertas características comunes: manejan el particionamiento físico de los datos y el acceso a los mismos a nivel de archivo, y asignan los nombres de los archivos, al almacenamiento físico. En el nivel de hardware, el almacenamiento de disco se describe en términos de plato, lados, pistas, cilindros y sectores. En nivel de BIOS del sistema, el almacenamiento de discos se describe en términos de clústeres y sectores. En el nivel del SO, el almacenamiento en disco se describe en términos de directorios y archivos.

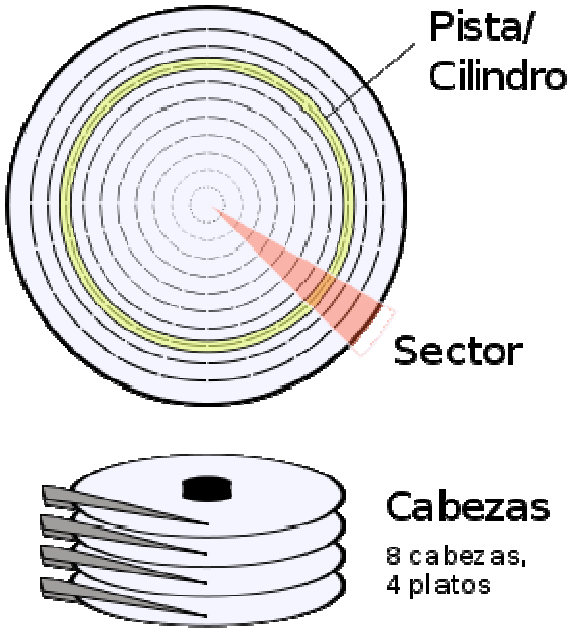
Programas en lenguaje ensamblador Los programas a nivel de usuario en lenguaje ensamblador pueden acceder de manera directa al BIOS del sistema en MS-DOS, Windows 95, 98 y Milenium. Por ejemplo, tal vez tengamos que almacenar y obtener los datos almacenados en un formato no convencional, recuperar datos perdidos o realizar diagnósticos en el hardware del disco. En este capítulo mostraremos ejemplos de funciones del BIOS del sistema para archivos y sectores. Como ilustración de un acceso ordinario a los datos, a nivel del sistema operativo, al final del capítulo se presenta una variedad de funciones de MS-DOS para manipulación de unidades y directorios.

Si utilizamos Windows NT, 2000 o XP, los programas a nivel de usuario sólo pueden acceder al sistema de discos usando la API Win32. Esa regla protege la seguridad del sistema, y sólo los programas controladores de dispositivos que se ejecutan en el nivel de privilegios más alto pueden pasarla por alto.

Pista, cilindros y sectores

Una unidad de disco ordinaria, como la que se muestra a continuación, está compuesta por varios platos unidos a un eje, el cual gira a una velocidad constante. Encima de la superficie de cada plato hay una cabeza de lectura/escritura, que graba pulsos magnéticos. Las cabezas de lectura/escritura avanzan hacia el centro y hacia el borde como un grupo, en pasos pequeños.

Elementos físicos de un disco duro.



A la superficie de un disco se le da un formato en forma de banda concéntricas invisibles llamadas, *pistas*, en las que los datos se almacenan de manera magnética. Una mitad de disco duro típica de 3,5" puede contener miles de pistas. Al proceso de desplazar las cabezas de lectura/escritura de una pista a otra se le conoce como *búsqueda*. El *tiempo promedio* de búsqueda es un tipo de medida de la velocidad de los discos. Otra medida es RPM (revolución por minuto) que, por lo general, es de 7,200. La pista exterior de un disco es la pista 0, y los números de pista se incrementan a medida que avanzan hacia el centro.

Un *cilindro* se refiere a todas las pistas a las que se puede acceder desde una sola posición de las cabezas de lectura/escritura. Al principio, un archivo se guarda en el disco usando cilindros adyacentes. Esto reduce la cantidad de movimiento de las cabezas de lectura/escritura.

Un *sector* es una posición de 512 bytes de una pista, como se mostró en la imagen anterior. El fabricante marca los sectores físicos en forma magnética (invisible) en el disco, usando lo que se conoce como *formato de bajo nivel*. Los tamaños de los sectores nunca cambian, sin

importar el sistema operativo instalado. Un disco duro puede tener 63 o más sectores por pista.

La *geometría física del disco* es una manera de describir su estructura, para que el BIOS del sistema pueda leerla. Consiste en el número de cilindros por disco, el número de cabezas de lectura/escritura por cilindro, y el número de sectores por pista. Existen las siguientes relaciones:

- El número de cilindros por disco es igual al número de pistas por superficies.
- El número total de pistas es igual al número de cilindros, multiplicado por el número de cabezas por cilindro.

Fragmentación Con el tiempo, a medida que los archivos se esparcen más a lo largo de un disco, se fragmentan. Un *archivo fragmentado* es uno cuyo sus sectores ya no se encuentran en áreas contiguas del disco. Cuando esto ocurre, las cabezas de lectura/escritura tienen que saltar entre las pistas para leer los datos del archivo. Esto reduce la velocidad de lectura y escritura de los archivos.

Traducción a números de sectores lógicos Los controladores de disco duro llevan a cabo un proceso llamado *traducción*, la conversión de la geometría física del disco a una estructura lógica que puede comprender el sistema operativo. Por lo general, el controlador está incrustado en el firmware ya sea en la misma unidad o en una tarjeta controladora separada. Después de la traducción, el sistema operativo puede trabajar con lo que se conoce como *número de sectores lógicos*. Estos números de sectores lógicos siempre se encuentran en forma secuencial, empezando con cero.

Particiones de disco (volúmenes)

En MS Windows, una sola unidad física de disco duro puede dividirse en una o más unidades lógicas llamadas *particiones* o *volúmenes*. Cada partición con formato se representa mediante una letra de unidad separada, como C, D o E, y se le puede dar formato usando uno o varios sistemas de archivos. Una unidad puede contener dos tipos de particiones: primarias y extendidas.

Por lo general, una partición primaria tiene capacidad de inicio y contiene un sistema operativo. Una *partición extendida* puede dividirse en un número ilimitado de *particiones lógicas*. Cada partición lógica se le asigna una letra de unidad (C, D , E etcétera). Las particiones lógicas pueden tener capacidad de inicio. Es posible dar formato a cada partición lógica o del sistema con un sistema de archivos distinto.

Por ejemplo supongamos que una unidad de disco duro de 20GN se le asigno una partición primaria de 10GB (unidad C), y que le instalamos el sistema operativo. Su partición extendida sería de 10GB. De manera arbitraria podríamos dividir esa misma partición en dos particiones lógicas de 2GB y 8GB, para después darles formatos con varios sistemas de archivo como FAT16, FAT32 o NTFS. Si suponemos que no había ninguna otra unidad de disco instalada, a las dos particiones lógicas se les asignarían las letras de unidad D y E.

Sistemas multi inicio Es bastante común crear varias particiones primarias, cada una de las cuales es capaz de arrancar (cargar) un sistema operativo distinto. Esto hace posible evaluar software en distintos entornos y aprovechar las ventajas de seguridad en los sistemas más avanzados. Muchos desarrolladores de software utilizan una partición primaria para crear un entorno de prueba para el software en desarrollo. Luego tienen otra partición primaria que almacena el software de producción ya probado y listo para que los clientes lo utilicen.

Por otro lado, las particiones lógicas están diseñadas principalmente para los datos. Es posible que distintos sistemas operativos compartan datos almacenados en la misma partición lógica. Por ejemplo, todas las versiones más recientes de MS Windows y Linux pueden leer discos FAT32. Una computadora puede arrancar desde cualquiera de estos sistemas operativos y leer los mismos archivos de datos, en una partición lógica compartida.

Registro de inicio maestro El registro de inicio maestro (MBR), que se crea al momento de crear la primera partición en un disco duro, se encuentra en el primer sector lógico de la unidad. El MBR contiene lo siguiente:

- La *tabla de particiones del disco*, que describe los tamaños y ubicaciones de todas las particiones en el disco.
- Un pequeño programa que localiza el sector de inicio de la partición y transfiere el control a un programa en el sector que carga el sistema operativo.

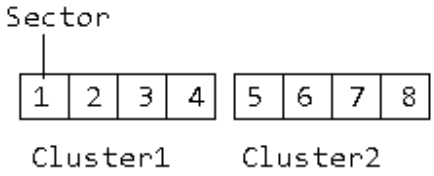
Sistemas de archivos

Todos los sistemas operativos tienen cierto tipo de sistemas de administración de discos. En el nivel más bajo administra las particiones. En el siguiente nivel, administra los archivos y directorios. Un sistema de archivos debe mantener un registro de la ubicación, los tamaños y atributos de cada archivo del disco. Vamos a analizar el sistema de archivos tipo FAT que se creó en un principio para la IBM PC, y que aun se utiliza en MS Windows. Un sistema de archivos tipo FAT utiliza la siguiente estructura:

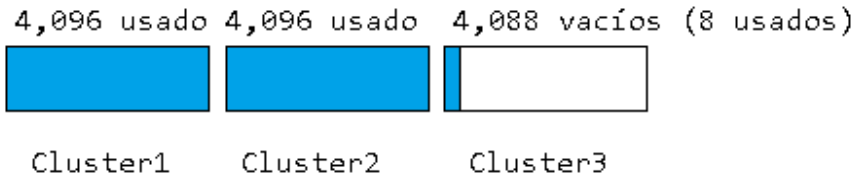
- Una asignación de sectores lógicos a *clústeres*, la unidad básica de almacenamiento para todos los archivos y directorios.
- Una asignación de los nombres de archivos y directorios a secuencias de clústeres.

Un *clúster* es la unidad más pequeña de espacio que utiliza un archivo; consiste en uno o más sectores de disco adyacentes. Un sistema de archivos almacena cada archivo como una secuencia enlazada de clústeres. El tamaño de un clúster depende tanto del tipo del sistema de archivos en uso, como el tamaño de su partición de disco. La siguiente figura muestra un archivo compuesto de dos clústeres de 2048 bytes, cada uno de los cuales contiene cuatro sectores de 512 bytes. Para hacer referencia a una cadena de clústeres se utiliza una *tabla de asignación de archivos* (FAT) que lleva el registro de todos los clústeres que utiliza un archivo. En la entrada de directorio de cada archivo se almacena un apuntador a la entrada del primer clúster en la FAT.

Ejemplo de cadenas de clústeres.



Espacio desperdiciado Inclusive hasta un archivo pequeño requiere por lo menos un clúster de almacenamiento en el disco, lo cual puede provocar un desperdicio de espacio. La siguiente figura muestra un archivo de 8.200 bytes, el cual llena por completo dos clústeres de 4096 bytes y utiliza sólo 8 bytes de un tercer clúster.



Esto deja 4088 bytes de espacio en disco desperdiciados en el tercer clúster. Un tamaño de clúster de 4096 (4KB) se considera una forma eficiente de almacenar archivos pequeños. Imaginemos lo que resultaría si nuestro archivo de 8200 bytes se almacenara en un volumen que tuviera clústeres de 32 KB. En ese caso, se desperdiciarían 24568 bytes. En los volúmenes que tienen una gran cantidad de archivos pequeños, es mejor usar tamaños pequeños para los clústeres.

Ejemplo en Windows 2000/XP en la siguiente tabla se muestran los tamaños de clústeres estándar y los tipos de sistemas de archivos, para unidades de disco que se utilizan en Windows 2000 y Windows XP. Estos valores cambian a menudo con los nuevos sistemas operativos, por lo que la información que se muestra en la tabla se vuelve obsoleta rápidamente.

Tamaño del volumen	Clúster de FAT16	Clúster de FAT32	Clúster de NTFS
1.25GB-2GB	32KB	4KB	2KB
2GB-4GB	64KB	4KB	4KB
4GB-8GB	*ns (no se soporta)*	4KB	4KB
8GB-16GB	*ns*	8KB	4KB
16GB-32GB	*ns*	15KB	4KB
32GB-2TB	*ns*	*ns*	4KB

FAT12

El sistema de archivos FAT12 se utilizó por primera vez en los disquetes de la IBM-PC. Aún se soporta en todas las versiones de MS Windows y Linux. El tamaño del clúster es de sólo 512 bytes, por lo que es ideal para guardar archivos pequeños. Cada entrada en su tabla de asignación de archivos es de 12 bits. Un volumen FAT12 almacena menos de 4087 clústeres.

FAT16

El sistema de archivos FAT16 es el único formato disponible para los discos duros a los que se da formato en MS-DOS. Se soporta en todas las versiones de MS Windows y Linux. Hay algunas desventajas en el FAT16:

- El almacenamiento es ineficiente en los volúmenes de más de 1 GB, ya que FAT16 utiliza tamaños de clústeres grandes.
- Cada entrada en la tabla de asignación de archivos es de 16 bits, lo cual limita el número total de clústeres.
- El volumen puede contener entre 4087 t 65,526 clústeres.
- El sector de inicio no está respaldado, por lo que un error de lectura en un solo sector puede ser catastrófico.
- No hay seguridad integrada en el sistema, ni permisos individuales para los usuarios.

FAT32

El sistema de archivos FAT32 se introdujo con la versión OEM2 de Windows 95, ya que se perfeccionó en Windows 98. Tiene varias mejoras, en comparación con FAT16:

- Un archivo individual puede ser de hasta 4GB menos 2 bytes.
- Cada entrada en la tabla de asignación de archivos es de 32 bits.
- Un volumen puede contener entre 65,526 y 268,435,456 clústeres.
- La carpeta raíz puede ubicarse en cualquier parte del disco, y puede tener casi cualquier tamaño.
- Los volúmenes pueden contener hasta 32GB.
- Utiliza un tamaño de clústeres más pequeños que FAT16 en los volúmenes que contienen de 1GB a 8GB, lo cual resulta en un menor desperdicio de espacio.
- El registro de inicio incluye una copia de respaldo de las estructuras de datos críticas. Esto significa que las unidades FAT32 son menos susceptibles a un solo punto de falla que las unidades FAT16.

NTFS

El sistema de archivos NTFS trabaja sobre Windows NT, 2000 y XP, tiene mejoras considerables, en comparación con FAT32:

- NTFS maneja volúmenes grandes, que pueden encontrarse en un solo disco duro o pueden espaciarse a través de varios discos duros.
- El tamaño de clústeres predeterminado es de 4KB, para discos de más de 2GB.

- Soporta los nombres de archivos Unicode (caracteres que no son ASCII) de hasta 255 caracteres de longitud.
- Permite establecer permisos en los archivos y carpetas. El acceso puede ser por usuarios individuales, o grupos de usuarios. Hay distintos niveles de acceso posible (leer, escribir, modificar, etcétera).
- Cuenta con cifrado de datos integrado y compresión en archivos, carpetas y volúmenes.
- Puede rastrear cambios individuales a los archivos en un tiempo específico, mediante un *diario de modificaciones*.
- Pueden establecerse cuotas de discos para usuarios individuales o grupos de usuarios.
- Cuenta con una capacidad robusta de recuperación de los errores de datos. Repara los errores de manera automática, manteniendo un registro de transacciones.
- Soporta los volúmenes reflejados (disk mirroring), en donde los mismos datos se escriben en forma simultánea en varías unidades.

La siguiente tabla presenta cada uno de los distintos sistemas de archivos que se utilizan por lo regular en las computadoras basadas en Intel, mostrando el soporte en varios sistemas operativos.

Soporte de los sistemas operativos para los sistemas de archivos.

Sistema de archivos	MS-DOS	Linux	Win 95/98	Win NT 4	Win 2000/XP
FAT12	X	X	X	X	X
FAT16	X	X	X	X	X
FAT32		X	X		X
NTFS				X	X

Áreas principales del disco

Los volúmenes FAT12 y FAT16 tienen ubicaciones específicas reservadas para el registro de inicio, la tabla de asignación de archivos y el directorio raíz (el directorio raíz en una unidad FAT32 no se almacena en una ubicación fija). El tamaño de cada área se determina cuando se da el formato al volumen. Por ejemplo, la asignación de sectores en un disquete de 3.5 pulgadas y 1.44MB se muestra en la siguiente tabla.

Sector lógico	Contenido
0	Registro de inicio
1-18	Tabla de asignación de archivos (FAT)

19-32	Directorio raíz
33-2,879	Área de datos

Registro de inicio El *registro de inicio* contiene una tabla que almacena la información del volumen y un programa de inicio corto, que carga a MS-DOS en la memoria. El programa de inicio comprueba la existencia de ciertos archivos del sistema operativo y los carga en la memoria. La siguiente tabla muestra una lista representativa de campos en un registro de inicio de MS-DOS típico. El orden exacto de los campos varía entre las distintas versiones del sistema operativo.

Desplazamiento	Longitud	Descripción
00	3	Salta al código de inicio (instrucción JMP)
03	8	Nombre del fabricante, número de versión
0B	2	Bytes por sector
0D	1	Sectores por clúster (potencia de 2)
0E	2	Número de sectores reservados (antes de FAT #1)
10	1	Número de copias de la FAT
11	2	Máximo número de entradas en el directorio raíz
13	2	Número de sectores de disco para las unidades menores de 32MB
15	1	Byte descriptor de medios
16	2	Tamaño de la FAT, en sectores
18	2	Sectores por pista
1A	2	Número de cabezas de la unidad
1C	4	Número de sectores ocultos
20	4	Número de sectores del disco para unidades mayores de 32MB
24	1	Número de unidad (modificado por MS-DOS)
25	1	Reservado
26	1	Firma de inicio extendida (siempre es 29h)

27	4	Número de ID del volumen (binario)
2B	11	Etiqueta de volumen
36	8	Tipo de sistema de archivos (ASCII)
3E	-	Inicio del programa de inicio y los datos

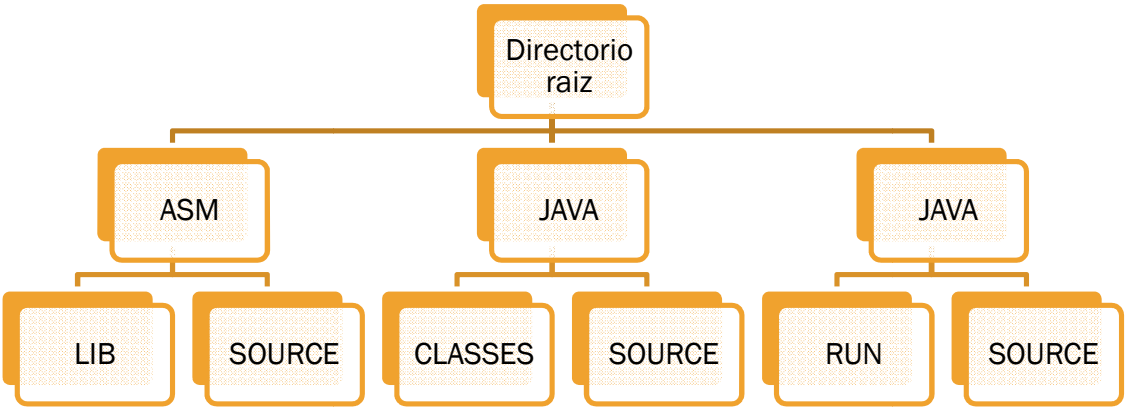
Directorio raíz El *directorio raíz* es el directorio principal de un volumen. Las entradas en el directorio puede ser otros nombres de directorios o referencias a archivos. Una entrada de directorio que hace referencia a un archivo, contiene su nombre, tamaño, atributo y número de clúster de inicio que utiliza el archivo.

Área de datos El *área de datos* del disco es donde se almacenan los archivos y subdirectorios.

Directorio de disco

Cada disco estilo FAT y NTFS tiene un *directorio raíz*, el cual contiene la lista principal de archivos en el disco. El directorio raíz también puede contener los nombres de otros directorios, conocidos como *subdirectorios*. Un subdirectorio puede considerarse como un directorio cuyo nombre aparece en algún otro directorio, a este último se le conoce como *directorio padre*. Cada subdirectorio puede contener nombres de archivos y nombres de directorios adicionales. El resultado es una estructura tipo árbol con el directorio raíz en la parte superior, con ramificaciones hacia otros subdirectorios en los niveles inferiores.

Ejemplo de un árbol de directorios de disco.



Cada nombre de directorio y cada archivo dentro de un directorio se califican mediante los nombres de los directorios encima de él, a lo cual se le conoce como *ruta*. Por ejemplo, la ruta para el archivo PROG1.ASM en el directorio SOURCE debajo de ASM en la unidad C es:

```
C: \ASM\SOURCE\PROG1.ASM
```

Por lo general, puede omitirse la letra de la unidad de la ruta cuando se lleva a cabo una operación de entrada salida en la unidad de disco actual. A continuación se muestra una lista completa de los nombres de los directorios en nuestro árbol de directorios de ejemplo:

C: \

\ASM\

\ASM\LIB

\ASM\SOURCE

\JAVA

\JAVA\CLASSES

\JAVA\SOURCE

\CPP

\CPP\RUN

\CPP\SOURCE

Por ende, una *especificación de archivo* puede tomar la forma de un nombre de archivo individual, o de una ruta de directorio seguida de un nombre de archivo. También se puede anteponer una especificación de unidad.

Estructura de directorios de MS-DOS

Si tratamos de explicar todos los diversos formatos de directorios disponibles hoy en día en las computadoras basadas en Intel, tendríamos por lo menos que incluir a Linux, MS-DOS y todas las versiones de MS Windows. En vez de ello, vamos a usar MS-DOS como un ejemplo básico y examinaremos su estructura con más detalle. Después continuaremos con una descripción de la estructura de nombres de archivos extendidos disponibles en MS Windows.

Cada entrada de directorio de MS-DOS es de 32 bytes y contiene los campos que se muestran en la siguiente tabla.

Desplazamiento Hexadecimal	Nombre del campo	Formato
00-07	Nombre de archivo	ASCII
08-0A	Extensión	ASCII
0B	Atributo	Binario de 8 bits
0C-15	Reservado para MS-DOS	
16-17	Etiqueta de hora	Binario de 16 bits
18-19	Etiqueta de fecha	Binario de 16 bits
1A-1B	Número de clúster inicial	Binario de 16 bits
1C-1F	Tamaño del archivo	Binario de 32 bits

El campo *nombre de archivo* contiene el nombre de un archivo, un subdirectorio o la etiqueta de volumen del disco. El primer byte puede indicar el estado del archivo, o puede ser el primer carácter de un nombre de archivo. En la siguiente tabla se muestran los posibles valores de estado.

Byte de estado del nombre de archivo.

Byte de estado	Descripción

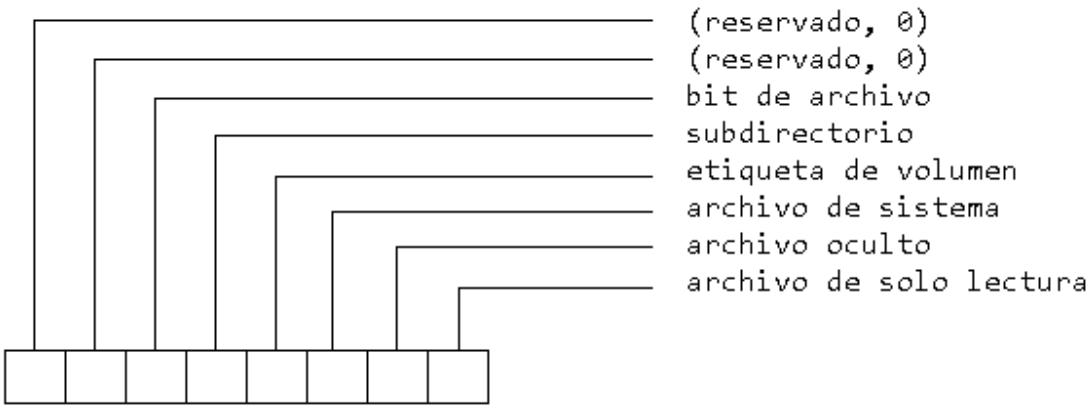
00h	La entrada nunca se ha utilizado
01h	Si el byte de atributo = 0Fh y el byte de estado = 01h, ésta es la primera entrada de nombre de archivo extenso. (Contiene la última parte del nombre,”.” y la extensión del nombre de archivo)
05h	El primer carácter del nombre del archivo es en realidad el carácter E5h (raro)
E5h	La entrada contiene un nombre de archivo, pero el archivo se borró
2Eh	La entrada (.) es para un nombre de directorio. Si el segundo byte también es 2Eh (..), el campo del clúster contiene el número de clúster del directorio padre de este directorio
4*n*h	Primera entrada de nombre de archivo extenso (contiene la primera parte del nombre): si el byte de atribuo =0Fh, esto marca la última de varias entradas que contiene un solo nombre de archivo extenso. El código *n* indica el número de entradas que utiliza el nombre de archivo

El campo *número de clúster inicial* de 16 bits se refiere al número del primer clúster asignado al archivo, así como su entrada inicial en la tabla de asignación de archivos (FAT). El campo *tamaño de archivo* es un número de 32 bits que indica el tamaño del archivo, en bytes.

Campo atributo

El campo *atributo* identifica el tipo de archivo. El campo está asignado por bits y, por lo general, contiene una combinación de uno de los valores que se muestran en la siguiente figura.

Campos de bytes en atributos de un archivo



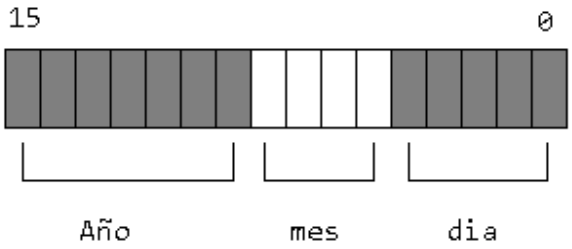
Los dos bits *reservados* siempre deben ser cero. El bit *archivo* está activo cuando se modifica un archivo. El bit *subdirectorio* está activo si la entrada al directorio de MS DOS contiene el

nombre de un subdirectorio. La *etiqueta de volumen* idenfica a la entrada como el nombre de un volumen de disco. El bit *archivo de sistema* indica que el archivo es parte del sistema operativo. El bit *archivo oculto* oculta el archivo; su nombre no aparece en una visualización de directorio. El bit *solo lectura* evita que el archivo se elimine o se modifique de cualquier forma. Por ejemplo un valor de atributo 0Fh indica que la entrada actual del directorio es para un nombre de archivo extendido.

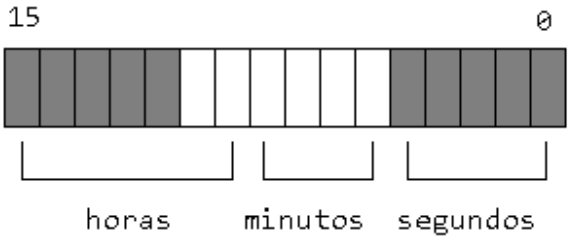
Fecha y hora

El campo *etiqueta de fecha* indica la fecha en que se creo el archivo, o la fecha de su última modificación, y se expresa como un valor asignado por bits. El valor del año está entre 0 y 119, y se le suma automáticamente a 1980 (el año en que salió al mercado la IBM-PC). El valor del mes está entre 1 y 12, y el valor del día entre 1 y 31.

Campo de etiqueta de fecha de un archivo



El campo *etiqueta de hora* indica la hora en que se creó o se modificó por última vez el archivo, y se expresa como un valor asignado por bits. Las horas pueden estar entre 0 y 23, los minutos entre 0 y 59, y los segundos entre 0 y 59, almacenados como una cuenta de incrementos de 2 segundos. Por ejemplo, un valor de 10100 binario equivale a 40 segundos.

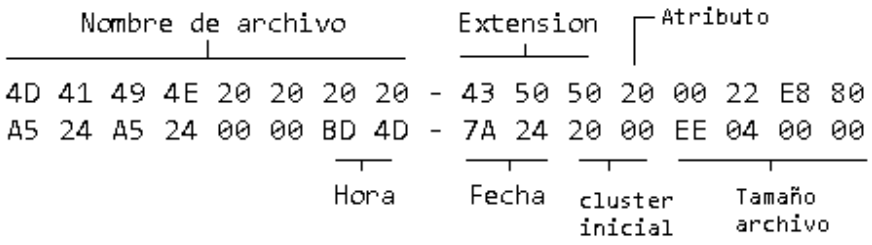


La etiqueta de hora a continuación indica las 14:02:40 horas.

```
01110 000010 10100

Horas minutos segundos
```

Ejemplo de entrada de directorio de un archivo Examinemos la entrada para un archivo llamado MAIN.CPP (siguiente figura). Este archivo tiene un atributo normal, y su bit de archivo (20h) está activo, lo cual muestra que ha sido modificado. Su número de clúster inicial es

0020h, su tamaño es de 000004EEh bytes, el campo *Hora* es igual a 4DBDh (9:45:58) y el campo *Fecha* es igual a 247Ah (Marzo 26, 1998).



En esta figura, la hora, fecha y número de clúster inicial son valores de 16 bits, almacenados en orden litte endian (byte inferior, seguido por el byte superior). El campo *Tamaño de archivo* es una doble palabra, que también se almacena en orden little endian.

Nombres de archivos extensos en MS Windows

En MS Windows, a un nombre de archivo mayor que 8 + 3 caracteres, o a un nombre que utilice una combinación de letras mayúsculas y minúsculas se le asignan varias entradas de directorio del disco. Si el byte de atributo es igual a 0Fh, el sistema analiza el byte en el desplazamiento 0. Si el dígito superior es igual a 4, esta entrada empieza una serie de entradas de nombre de archivo extenso. Las entradas subsiguientes cuentan en orden descendiente desde n – 1, en donde n = al número de entradas. Por ejemplo, si un nombre de archivo requiere tres entradas, el primer byte de estado será 43h. Las entradas subsiguientes serñan los bytes de estado iguales a 02h y 01h, como puede verse en la siguiente tabla:

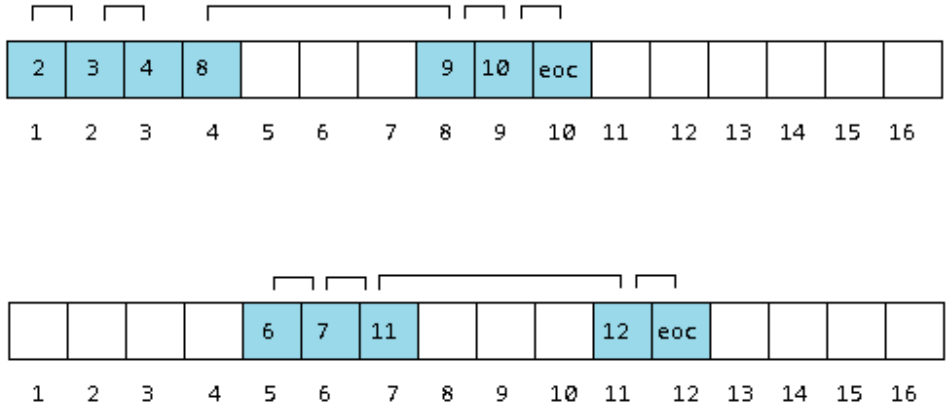
Bytes de estado	Descripción
43	Indica que se utilizan tres entradas para el nombre de archivo extenso, en total, y esta entrada contiene la última parte del nombre de archivo, “.”, y una extensión de tres caracteres
02	Contiene la segunda parte del nombre de archivo
01	Contiene la primera parte del nombre de archivo

Tabla de asignación de archivos (FAT)

Los sistemas de archivos FAT12, FAT16 y FAT32 utilizan una tabla conocida como *tabla de asignación de archivos* (FAT) para llevar el registro de la ubicación de cada archivo en el disco. La FAT asigna los clústeres de disco, mostrando a qué archivo específico pertenecen. Cada entrada corresponde a un número de clúster, y cada clúster contiene uno o más sectores. En otras palabras, la 10ºa Entrada en la FAT indica el 10vo clúster en el disco, la 11ºa entrada identifica el 11vo clúster, y así sucesivamente.

Cada archivo se representa en la FAT como una lista enlazada, llama *cadena de clústeres*. Cada entrada en la FAT contiene un entero que identifica a la siguiente entrada. En la siguiente figura se muestran dos cadenas de clústeres, una para **Archivo1** y otra para **Archivo2**.

Ejemplo: dos cadenas de clústeres



Archivo1 ocupa los clústeres 1,2,3,4,8 y 9. **Archivo2** ocupa los clústeres 5,6,7,11 y 12. El marcador **eoc** (*fin de cadena*) en la última entrada en la FAT para un archivo es un valor entero predefinido, que marca el clúster final en la cadena.

Cuando se crea un archivo, el sistema operativo busca la primera entrada disponible en la FAT. Se producen huecos cuando no hay suficientes clústeres contiguos para contener el archiv completo. En el diagrama anterior, esto ocurrió tanto a **Archivo1** como a **Archivo2** cuando se modifica un archivo y se guarda de vuelta en el disco, a menudo su cadena de clústeres se fragmenta en forma considerable. Si muchos archivos se fragmentan, el rendimiento del disco empieza a degradarse, ya que las cabezas de lectura/escritura deben saltar entre distintas pistas para localizar todos los clústeres de un archivo. La mayoría de los sistemas operativos cuentan con una herramienta de desfragmentación de disco integrada.

Lectura y escritura de sectores de disco (7305h)

La función 7305h de INT 21h (lectura y escritura absoluta de disco) nos permite leer y escribir en sectores de disco lógicos. Al igual que todas las instrucciones INT, está diseñada para ejecutarse sólo en modo de direccionamiento real de 16 bits. No trataremos de llamar a INT 21h (ni a cualquier otra interrupción) desde el modo protegido, debido a las complejidades implícitas.

La función 7305h funciona en los sistemas operativos FAT12, FAT16 y FAT32 en Windows 95, 98 y Windows ME. No funciona en Windows NT, 2000 o XP debido a que tienen una seguridad más estricta. Cualquier programa que tenga permitido leer y escribir sectores del disco podría ignorar con facilidad los permisos de compartición de archivo y directorios. Al llamar a la función 7305h, hay que pasarle los siguientes argumentos:

AX	7305h
DS:BX	Segmento/desplazamiento de una variable de la estructura ESDISCO
CX	0FFFFh
DL	Número de unidad (0 = predeterminada, 1 = A, 2 = B, 3 = C, etcétera)
SI	Bandera de lectura/escritura

Una estructura ESDISCO contiene el número del sector inicial, el número de sectores en los que se va a leer o escribir, y la dirección segmento/desplazamiento del búfer del sector:

```
ESDISCO STRUCT

   sectorInicial DWORD 0               ; número del sector inicial

   numSectores   WORD 1                ; número de sectores

   despBufer     WORD OFFSET bufer     ; desplazamiento del búfer

   segBufer      WORD SEG búfer        ; segmento del búfer
```

A continuación se muestra ejemplos de un búfer de entrada para guardar los datos de los sectores, junto con una variable de la estructura ESDISCO:

```
.data

bufer BYTE 512 DUP(?)

estrucDisco ESDISCO <>

estrucDisco2 ESDISCO <10,5>    ; sectores 10, 11, 12, 13, 14
```

Al llamar a la función 7305h, el argumento que se pasa en SI determina si queremos leer sectores o escriir en ellos. Para leerlos, se borra el bit 0; para escribir en ellos, se activa el bit 0. Además, los bits 13, 14 y 15 se configuran al escribir sectores mediante el uso del siguiente esquema:

Bits 15 – 13	Tipo de sector
000	Otro/desconocido
001	Datos de la FAT
010	Datos de directorio
011	Datos de archivo normal

El resto de los bits (del 1 al 12) siempre deben estar en cero.

Ejemplo 1: las siguientes instrucciones leen uno o más sectores de la unidad C:

```
mov ax, 7305h                   ; Lectura/Escritura absoluta
mov cx,0FFFFh                   ; siempre tiene este valor
mov dl, 3                       ; unidad C
mov bx, OFFSET estrucDisco      ; estructura ESDISCO
mov si,0                        ; lee el sector
int 21h
```

Ejemplo 2: las siguientes instrucciones escriben en uno o más sectores de la unidad A:

```
mov ax,7305h                    ; Lectura/Escritura absoluta
mov cx,0FFFFh                   ; siempre tiene este valor
mov dl,1                        ; unidad A
mov bx, OFFSET estrucDisco      ; estructura ESDISCO
mov si,6001h                    ; escribe en sector(es) normal(es)
int 21h
```

Funciones de archivo a nivel de sistema

En el modo direccionamiento real, INT 21h proporciona los servicios del sistema (tabla a continuación) para crear y combinar directorios, modificar los atributos de los archivos, buscar archivos que coincidan, etcétera. Estos servicios van más allá de lo que hay disponible generalmente en las bibliotecas de los lenguajes de programación de alto nivel. Al llamar a cualquiera de estos servicios, el número de la función se coloca en AH o AX. Otros registros pueden contener parámetros de entrada. Vamos a ver con detalle algunas de las funciones de uso común.

Número de función	Nombre de función
0Eh	Establece la unidad predeterminada
19h	Obtener la unidad predeterminada
7303h	Obtener espacio libre en el disco
39h	Crear subdirectorio

3Ah	Eliminar subdirectorio
3Bh	Establecer directorio actual
41h	Eliminar directorio
43h	Obtener/establecer atributo de archivo
47h	Obtener la ruta del directorio actual
4Eh	Buscar el primer archivo que coincida
4Fh	Buscar el siguiente archivo que coincida
56h	Cambiar el nombre del archivo
57h	Obtener/establecer la fecha y hora del archivo
59h	Obtener la información de error extendida

Windows 95/98/Me Soporta todas las funciones INT 21h existentes de MS-DOS y proporciona extensiones que permiten a las aplicaciones basadas en MS-DOS aprovechar las características como los nombres de archivo extenso y el bloqueo exclusivo de volúmenes. La función 7303h de INT 21h (obtener espacio libre en el disco) es un ejemplo de una función mejorada del sistema que reconoce los discos mayores de los que se soportan origialmente en MS-DOS.

Obtener espacio libre del disco (7303h)

La función 7303h de INT 32h puede usarse para averiguar el tamaño de un volumen de disco y cuánto espacio libre hay disponible en una unidad FAT16 o FAT32. La información se devuelve en una estructura estándar llamada **ExtGetDskFreSpcStruc**, como se muestra a continuación:

```
ExtGetDskFreeSpcStruc STRUC

   StructSize                 WORD ?

   Level                      WORD ?

   SectorsPerCluster          DWORD ?

   BytesPerSector             DWORD ?

   AvaliableClusters          DWORD ?

   TotalClusters              DWORD ?

   AvaliablePhysSectors       DWORD ?
```

```
    TotalPhysSectors            DWORD ?
    AvaliableAllocationUnits    DWORD ?
    TotalAllocationUnits        DWORD ?
    Rsvd                        DWORD 2 DUP (?)
ExtGetDskFreeSpcStruc ENDS
```

La siguiente siguiente lista contiene una breve descripción de cada campo:

- **StructSize**: un valor de retorno que representa el tamaño de la estructura ExtGetDskFreSpcStruct en bytes. Cuando se ejecuta la función 7303h de INT 21, coloca el tamaño de la estructura en este miembro.
- **Level**: un valor de nivel de entrada y retorno. Este cambio debe inicializarse con cero.
- **SectorsPerCluster**: el número de sectores dentro de cada clúster.
- **BytesPerSector**: el número de bytes en cada sector.
- **AvaliableClusters**: el número de clústeres disponibles.
- **TotalClusters**: el número total de clústeres en el volumen.
- **AvaliablePhysSectors**: el número de sectores físicos disponibles en el volumen, sin ajuste para compresión.
- **TotalPhysSectors**: el número total de sectores físicos en el volumen, sin ajuste para compresión.
- **AvaliableAllocationUnits**: el número de unidades de asignación disponible en el volumen, sin ajuste para compresión.
- **TotalAllocationUnits**: el número de unidades de asignación en el volumen, sin ajuste para compresión.
- **Rsv:** miembro reservado.

Llamada a la función Al llamar a la función 7303h de INT 21h, se requieren los siguientes parámetros:

- AX debe ser igual a 7303h.
- ES:DI debe apuntar a una variable **ExtGetDskFreSpcStruct**.
- CX debe contener el tamaño de la variable **ExtGetDskFreSpcStruct**.
- DS:DX deben apuntar a una cadena con terminación nula que contenga el nombre de la unidad. Podemos usar el tipo de especificación de unidad de MS-DOS tal como (“C:\”), o podemos usar una especificación de volumen de la convención de nomenclatura universal como (\\Servidor\RecursoCompartido).

Si la función se ejecuta con éxito, borra la bandera Acarreo y llena la estructura. En caso contrario, activa la andera Acarreo. Después de llamara la función, los siguientes tipos de cálculos podrían ser útiles:

- Para averiguar que tan grande es el volumen en kilobytes, debemos usar la fórmula (TotalCluster * SectorsPerCluster * BytesPerSector)/1024.

- Para averiguar cuánto espacio libre hay en el volumen, en kilobytes, la fórmula es (AvaliableClusters * SectorsPerCluster * BytesPerSector)/1024.

Crear subdirectorio (39h)

La función 39h de INT 21h crea un nuevo subdirectorio. Recibe un apuntador en DS:DX a una cadena con terminación nula que contiene una especificación de ruta. El siguiente ejemplo muestra cómo crear un nuevo subdirectorio llamado ASM en el directorio raíz de la unidad predeterminada:

```
.data

nombreruta BYTE “\ASM”,0

.code

   mov ah,39h                  ; crea un subdirectorio

   mov dx, OFFSET nombreruta

   int 21h

   jc mostrar_error
```

La bandera Acarreo se activa si la función falla. Los códigos de error posibles son: 3 (no *se encontró la ruta*), 5 (*acceso denegado: el directorio contiene archivos*), 6 (*manejador inválido*), y 16 (*se intentó eliminar el directorio actual*).

Establecer el directorio actual (3Bh)

La función 3Bh de INT 32h establece el directorio actual. Recibe un apuntador en DS:DX a una cadena con terminación nula que contiene la unidad y ruta de destino. Por ejemplo, las siguientes instrucciones establecen el directorio actual a C: \ASM\PROGS:

```
.data

nombreruta BYTE “C: \ASM\PROGS”,0

.code

   mov ah,3Bh                  ; establece el directorio actual

   mov dx,OFFSET nombreruta

   int 21h

   jc mostrar_error
```

Obtener el directorio actual (47h)

La función 47h de INT 21h devuelve una cadena que contiene el directorio actual. Recibe un número de unidad en DL (0 = predeterminada, 1 = A, 2 = B, etcétera) y un apuntador en DS:SI a un búfer de 64 bytes. En este búfer, MS-DOS coloca una cadena con terminación nula que contiene el nombre de ruta completo del directorio raiz hasta el directorio actual (se omiten la letra de la unidad y la barra diagonal inversa que va al principio). Si se activa la bandera Acarreo cuando la función regresa, el único código de retorno de error posible en AX es 0Fh (*especificación de unidad inválida*).

En el siguiente ejemplo, MS-DOS devuelve la ruta del directorio actual en la unidad predeterminada. Suponiendo que el directorio actual es C: \ASM\PROGS, la cadena devuelta por MS-DOS es "ASM\PROGS":

```
.data

nombreruta BYTE 64 dup(0)      ; ruta almacenada aquí por MS-DOS

.code

   mov ah,47h                  ; obtiene ruta del directorio actual

   mov dl,0                    ; en la unidad predeterminada

   mov si,OFFSET nombreruta

   int 21h

   jc mostrar_error
```

Obtener y establecer atributos de archivo (7143h)

La función 7143h de INT 21h obtiene o establece los atributos de un archivo, entre otras tareas. (En Windows 9x, sustituye la función 39 de INT 21h de MS-DOS, que es más antigua). Recibe el desplazamiento de un nombre de archivo en DX. Para establecer los atributos del archivo, se asigna 1 a BL y a CX se le asignan uno o más atributos de los que se presentan en la siguiente tabla.

Valor	Significado
_A_NORMAL (0000h)	Se puede leer el archivo o escribir en él. Este valor es válido unicamente si se utiliza solo
_A_RDONLY (0001h)	Se puede leer el archivo, pero no escribir en él
_A_HIDDEN (0002h)	El archivo está oculto y no aparece en un listado de directorio ordinario
_A_SYSTEM (0004h)	El archivo es parte del sistema operativo, o éste lo utiliza en forma exclusiva

_A_ARCH (0020h)	El archivo es un archivo almacenado. Las aplicaciones utilizan este valor para marcar los archivos con el fin de respaldarlos o eliminarlos

El siguiente código estable los atributos de un archivo a sólo lectura y oculto:

```
mov ax,7143h

mov bl,1

mov cx, _A_HIDDEN + _A_RDONLY

mov dx, OFFSET nombrearchivo

int 21h
```

Para obtener los atributos actuales de un archivo, hay que establecer BX a 0 y llamar a la misma función. Los valores de los archivos se devuelven en CX como una combinación de potencias de 2. Debemos usar la instrucción TEST para evaluar los atributos individuales. Por ejemplo,

```
test cx, _A_RDONLY

jnz archivoSoloLectura          ; el archivo es de sólo lectura
```

El atributo _A_ARCH puede aparecer con cualquiera de los demás atributos.

15 Programación a nivel del BIOS

Área de datos del BIOS

El área de datos del BIOS, que se muestra parcialmente en la siguiente tabla, contiene los datos del sistema que utilizan las rutinas de servicio del BIOS de ROM. Por ejemplo, el búfer de escritura delantada del teclado (en el desplazamiento 001Eh) contiene los códigos ASCII y los códigos de exploración de las teclas que esperan a ser procesadas por el BIOS.

Área de datos del BIOS, en el segmento 0040h.

Desplazamiento HEX	Descripción
0000-0007	Direcciónes de puertos, COM1-COM4
0008-000F	Direcciones de puertos LPT1-LPT4
0010-0011	Lista del hardware instalado
0012	Banderade inicialización
0013-0014	Tamaño de memoria, en kilobytes

0015-0016	Memoria en el canal de E/S
0017-0018	Bandera de estado del teclado
0019	Almacenamiento alternativo de entrada de teclas
001A-001B	Apuntador del búfer de teclado (inicio)
001C-001D	Apuntador del búfer de teclado (final)
001E-003D	Búfer de escritura adelantada del teclado
003E-0048	Área de datos del disquete
0049	Modo actual de video
004A-004B	Número de columnas en la pantalla
004C-004D	Longitud del búfer de regeneración (video), en bytes
004E-004F	Desplazamiento inicial del búfer de regeneración (video)
0050-005F	Posiciones del cursor, páginas de video 1-8
0060	Línea final del cursor
0061	Línea inicial del cursor
0062	Número de página de video actual en visualización
0063-0064	Dirección base de la pantalla activa
0065	Registro de modo de CRT
0066	Registro para el adaptador de gráficos a color
0067-06B	Área de datos del casete
006C-0070	Área de datos del temporizador

Entrada de teclado mediante INT 16h

En este capítulo tendremos la oportunidad de trabajar directamente en el nivel del BIOS, llamando a las funciones instaladas (en gran parte) por el fabricante de la computadora. En este nivel sólo se encuentra a un nivel por encima del hardware, por lo cual se tiene mucha flexibilidad y control.

El BIOS maneja la entrada del teclado mediante llamadas a la interrupción 16h. Las rutinas del BIOS no permiten la redirección, pero facilitan la lectura de teclas extendidas del teclado,

como las teclas de función, de dirección, AvPág y RePág. Cada tecla extendida genera un código *de exploración de 8* bits. Los códigos de exploración son únicos para las computadoras compatibles con IBM. Todas las teclas generan códigos de exploración pero, por lo general, no ponemos atención a los códigos de exploración de los caracteres ASCII, ya que son estandarizados en casi todas las computadoras. En MS Windows, cuando se oprime una tecla extendida, su código ASCII es 00h o E0h, como se muestra en la siguiente tabla:

Teclas	Código ASCII
Ins, Supr, AvPág, RePág, Inicio, Fin, Flecha arriba, Flecha abajo, Flecha izquierda, Flecha derecha	E0h
Teclas de función (F1 – F12)	00h

Cómo funciona el teclado

La entrada del teclado sigue una rutina de eventos, que empieza con el chip controlador del teclado y termina cuando los caracteres se colocan en un arreglo llamado *búfer de escritura adelantada del teclado*. Pueden meterse hasta 15 tecleos en el búfer, ya que un tecleo genera 2 bytes (Código ASCII + código de exploración). Los siguientes eventos ocurren cuando el usuario oprime una tecla:

- El chip controlador del teclado envía un código de exploración numérico de 8 bits (*sc*) al puerto de entrada del teclado de la PC.
- El puerto de entrada está diseñada de manera que active una *interrupción*, una señal predefinida que indica a la CPU que un dispositivo de entrada-salida necesita atención. La CPU responde ejecutando la rutina de servicio INT 9h.
- La rutina de servicio INT 9h obtiene el código de exploración del teclado (*sc*) del puerto de entrada y busca el código ASCII correspondiente (*ac*), si lo hay. Inserta el código de exploración y el código ASCII en el búfer de escritura adeltanda del teclado (si el código de exploración no tiene código ASCII que coincida, el código ASCII de la tecla en el búfer de escritura adelantada es igual a cero 0 o E0h).

Una vez que el código de exploración y el código ASCII se encuentra seguros en el búder de escritura adelantada, permanecen ahí hasta que el programa actual en ejecución los extrae. Hay dos formas de hacer esto en las aplicaciones en modo real:

- Llamar a una función del BIOS mediante el uso de INT 16h, que obtenga tanto el código de exploración como el código ASCII del búfer de escritura adelantada del teclado. Esto es útil cuando se procesan las teclas extendidas, como las teclas de función y las flechas del cursor, que no tienen códigos ASCII.
- Llamar a una función a nivel del MS-DOS mediante el uso de INT 21h que obtiene el código ASCII del búfer de entrada. Si se oprimió una tecla extendida, hay que llamar a INT 21h una segunda vez para obtener el código de exploración.

Funciones de INT 16h

INT 16h presenta varias ventajas concisas en comparación con INT 21h, para el manejo del teclado. En primer luga, INT 16h puede obtener tanto el código de exploración como el código ASCII en un solo paso. En segundo lugar, INT 16h tiene varias operaciones adicionales, como establecimiento de la velocidad de repetición y obtener el estado de las banderas de teclado. La *velocidad de repetción* es la velocidad con la que se repite una tecla cuando se mantiene oprimida. Cuando no se sabe si el usuario oprimirá una tecla ordinaria o una tecla extendida, por lo general INT 16h es la función más adecuada.

Establecer velocidad de repetición (03h)

La función 03h de INT 16h nos permite establecer la velocidad de repetición del teclado, como se muestra en la siguiente tabla. Al mantener oprimida una tecla, hay retrasos de 250 a 1000 milisegundos antes de que la tecla empiece a repetirse. La velocidad de repetición puede ser entre 1Fh (más lenta) y 0 (más rápida).

Función 03h de INT 16h

Descripción	Establece la velocidad de repetición del teclado
Recibe	AH = 3 AL = 5 BH = retraso de repetición (0 = 250 ms; 1 = 500 ms; 2 = 750 ms; 3 = 1000 ms) BL = velocidad de repetición = más rápida (30/seg), 1F = más lenta (2/seg)
Devuelve	Nada

Llamada de ejemplo	`mov ax, 0305h`
	
	`mov bh, 1 ; retraso de repetición 500ms`
	
	`mov bl, 0Fh ; velocidad de repetición`
	
	`int 16h`

Meter tecla en búfer de teclado (05h)

Como se muestra en la siguiente tabla, la función 05h de INT 16h nos permite meter una tecla en el búfer de escritura adelantada del teclado. Una tecla consiste en dos enteros de 8 bits: el código ASCII y el código de exploración del teclado.

Función 05h de INT 16h

Descripción	Mete una tecla en el búfer del teclado
Recibe	AH = 5 CH = Código de exploración CL = Código ASCII
Devuelve	Si el búfer de escritura adelantada está lleno, CF = 1 y AL =1; en caso contrario, CF = 0, AL = 0

Llamada de ejemplo	`mov ah, 5`
	
	`mov ch, 3Bh ; código de exploración`
	
	`mov cl,0 ; código ASCII`
	
	`int 16h`

Esperar tecla (10h)

La función 10h de INT 16h elimina la siguiente tecla disponible del búfer de escritura adelantada del teclado. Si no hay una tecla de espera, el manejador del teclado espera a que el usuario oprima una, como se muestra en la siguiente tabla:

Función 10h de INT 16h

Descripción	Espera una tecla y explora una tecla del teclado
Recibe	AH = 10h
Devuelve	AH = código de exploración del teclado AL = código ASCII

Llamada de ejemplo	`mov ah,10h`
	
	`int 16h`
	
	`mov codigoExp1,ah`
	
	`mov codigoASCII,al`

Notas	Si no existe ya una tecla en el búfer, la función espera una. Sustituye a la función 00h de INT 16h

Programa de ejemplo

El siguiente programa de visualización del teclado utiliza un ciclo con INT 16h para recibir tecleos de entrada y mostrar el código ASCII junto con el código de exploración de cada tecla. Termina cuando se oprime ESC:

```
TITLE Visualización del teclado (teclado.asm)

; Este programa muestra los códigos de exploración

; del teclado y los códigos ASCII, usando INT 16h.

Include Irvine16.inc

.code

main PROC

   mov ax, @data

   mov ds, ax

   call ClrScr                ; borra la pantalla

L1: mov ah, 10h               ; entrada del teclado

   int 16h                    ; usando BIOS

   call DumpRegs              ; analiza AH, AL = ASCII

   cmp al, 1Bh                ; ¿Se oprimió ESC?

   jne L1                     ; no: repite el ciclo

   call ClrScr                ; borra la pantalla

   exit

main ENDP

END main
```

La llamada **DumpRegs** muestra todos los registros, pero sólo necesitamos ver AH (código de exploración) y AL (código ASCII).

Comprobación búfer del teclado (11h)

La función 11h de INT 16h nos permite hurgar en el búfer de escritura adelantada del teclado, para ver si hay teclas esperando. Devuelve el código ASCII y el código de exploración de las siguientes teclas disponibles, si la hay. Podemos utilizar esta función dentro de un ciclo para llevar a cabo otras tareas del programa. Observamos que la función no elimina la tecla del búfer de escritura adelantada.

Función 11h de INT 16h

Descripción	Comprueba el búfer del teclado
Recibe	AH = 11h
Devuelve	Si hay una tecla en espera, ZF = 0, AH = código de exploración, AL = código ASCII; en caso contrario, ZF = 1

Llamada de ejemplo	`mov ah,11h`
	
	`int 16h`
	
	`jz NoHayTeclaEnEspera  ; no hay tecla en el búfer`
	
	`mov codigoExpl, ah`
	
	`mov codigoASCII, al`

Notas	No elimina la tecla (si la hay) del búfer

Obtener banderas de teclado

La función 12h de INT 16h devuelve información valiosa acerca del estado actual de las banderas del teclado. Tal vez hemos notado que los programas de procesamiento de palabra a menudo muestran banderas o notaciones en la parte inferior de la pantalla, cuando se oprimen teclas como BloqMayús, BloqNúm e Insert. Para ello, examinan en forma continua la bandera de estado del teclado, vigilando cualquier cambio.

Función 12h de INT 16h

Descripción	Obtiene las banderas del teclado
Recibe	AH = 12h
Devuelve	AX = copia de las banderas del teclado

Llamada de ejemplo	mov ah,12h
	
	int 16h
	
	mov anderasTeclado,ax

Notas	Las banderas del teclado se encuentran en las direcciones 00417h – 00418h en el área de datos del BIOS

Las banderas del teclado, que se muestran en la siguiente tabla, son particularmente interesantes debido a que nos dicen mucho acerca de lo que el usario está haciendo con el teclado. ¿Está oprimiendo la tecla mayúsculas izquierda, o la derecha?. ¿Está oprimiendo también la tecla Alt?. Podemos responder a preguntas de este tipo mediante el uso de INT 16h. Cada bit es un 1 cuando la tecla que coincide se mantiene oprimida o se activa (Bloq mayús, Bloq despl, Bloq núm e Insert). En Windows 95 y 98, los bytes de la bandera del teclado también pueden obtenerse leyendo la memoria en el segmento 0040h, desplazamientos 17h 18h.

Valores de la bandera del teclado

Bit	Descripción
0	Se orpimió la tecla Mayúsculas derecha
1	Se oprimió la tecla Mayúsculas izquierda
2	Se oprimió cualquiera de las teclas Ctrl
3	Se oprimió cualquiera de las teclas Alt
4	Se activó la tecla Bloq Despl
5	Se activo la tecla Bloq Núm
6	Se activó la tecla Bloq Mayús
7	Se activo la tecla Insert
8	Se oprimió la tecla Ctrl izquierda
9	Se oprimió la tecla Alt izquierda
10	Se oprimió la tecla Ctrl derecha
11	Se oprimió la tecla Alt Derecha
12	Se oprimió la tecla Bloq Despl

13	Se oprimió la tecla Bloq Núm
14	Se oprimió la tecla Bloq Mayús
15	Se oprimió la tecla PetSis

Borrar el búfer del teclado

A menudo, los programas tienen un ciclo de procesamiento que sólo se puede interrumpir mediante teclas previamente ordenadas. Por ejemplo, los programas de juegos basados en DOS comprueba con frecuencia el búfer del teclado, para ver si se oprimieron teclas de flechas y otras especiales, mientras que al mismo tiempo se muestran imágenes de gráficos. El usuario podría oprimir una cantidad de teclas irrelevantes que sólo llenan el búfer de escritura adelantada del teclado, pero cuando se oprime la tecla correcta, se espera que el programa responda de inmediato al comando.

Mediante el uso de funciones INT 16h sabemos cómo comprobar el búfer del teclado para ver si hay teclas esperando (función 11h), y sabemos cómo eliminar una tecla del búfer (función 10h). El siguiente programa demuestra un procedimiento llamado **BorrarTeclado**, que utiliza un ciclo para borrar el búfer del teclado, al tiempo que comprueba un código de exploración de tecla específico. Para fines de prueba, el programa comprueba si se oprimió la tecla ESC, pero el procedimiento puede comprobar cualquier tecla:

```
TITLE Prueba de BorrarTeclado (BorrarTecl.asm)

; Este programa muestra cómo borrar el búfer del

; teclado, mientras se espera una tecla específica.

; Para probarlo, podemos oprimir rápidamente teclas al azar

; para llenar el búfer. Cuando se oprima Esc, el programa

; termina de inmediato

INCLUDE Irvine16.inc

BorrarTeclado PROTO, codigoExp1:BYTE

tecla_ESC = 1                          ; código de exploración

.code

main PROC

L1:

   ; Muestra un punto, para indicar el progreso del programa
```

```
   mov ah,2
   mov dl,'.'
   int 21h
   mov eax,300                          ; retraso de 300 ms
   call Delay
   INVOKE BorrarTeclado, tecla_ESC      ; comprueba la tecla Esc
   jnz L1                               ; continúa el ciclo si ZF = 0
terminar:
   call Clrscr
   exit
main ENDP
; -----------------------------------------------------
BorrarTeclado PROC,
   codigoExpl:BYTE
;
; Borra el teclado, al tiempo que comprueba un
; código de exploración específico.
; Recibe: código de exploración del teclado
; Devuelve: se activa la bandera Cero si el código ASCII
; se encontró, en caso contrario, la bandera cero se borra.
; -----------------------------------------------------
   push ax
L1:
   mov ah, 11h                  ; comprueba el búfer del teclado
   int 16h                      ; ¿se oprimió alguna tecla?
   jz noHayTecla                ; no: termina ahora (ZF=0)
```

```
   mov ah,10h                  ; sí: la elimina del búfer
   int 16h
   cmp ah, codigoExpl          ; ¿fue la tecla para salir?
   je terminar                 ; sí: termina ahora (ZF = 1)
   jmp L1                      ; no: comprueba el búfer otra vez
noHayTecla:                    ; no se oprimió ninguna tecla
   or al,1                     ; borra la bandera Cero
terminar:
   pop ax
   ret
BorrarTeclado ENDP
END main
```

El programa muestra un punto en la pantalla cada 300 milisegundos. Al presionar cualquier secuencia de teclas aleatorias estas son ignoradas. El programa se detendrá tan pronto como se oprima ESC.

Programación de VIDEO con INT 10h

Fundamentos

Tres niveles de acceso

Cuando un programa de aplicación necesita escribir caracteres en la pantalla, en modo de texto, podemos elegir entre tres tipos de salida:

- **Acceso a nivel de MS-DOS**: cualquier computadora que ejecute o emule a MS-DOS puede utilizar a INT 21h para escribir textos en la pantalla de video. La entrada/salida puede redirigirse fácilmente hacia otros dispositivos, como una impresora o disco. La salida es bastante lenta, y no podemos controlar el color del texto.
- **Acceso a nivel de BIOS**: los caracteres se introducen usando la función INT 10h, conocida como *servicios de BIOS*. Estos servicios se ejecutan con más rapidez que INT 21h y nos permiten especificar el color del texto. Al llenar grandes áreas de la pantalla, por lo general, puede detectarse un ligero retraso. La salida no puede redirigirse.
- **Acceso directo al video**: los caracteres se mueve directamente a la RAM de video, por lo que la ejecución es instantánea. La salida no puede redirigirse. Duran la era de MS-DOS, los programas procesadores de palabras y las hojas electrónicas de cálculo utilzaban este método. El uso de este método está restringido al modo de pantalla completa en Windows NT, 2000 y XP.

Los programas de aplicaciones varían en su elección de nivel de acceso a utilizar. Los que requieren el rendimiento más alto eligen el acceso directo al video; otros eligen el acceso a nivel del BIOS. El acceso a nivel de MS-DOS se utiliza cuando puede ser necesario redirigir la salida, o cuando la pantalla se comparte con otros programas. Hay que mencionar que las interrupciones de MS-DOS utilizan rutinas a nivel del BIOS para hacer su trabajo, y las rutinas del BIOS utilizan el acceso directo al video para producir sus resultados.

Ejecución de programas en modo de pantalla completa

Los programas que dibujan gráficos usando el BIOS de Video deen ejecutarse en uno de los siguientes entornos:

- MS-DOS puro.
- Un emulador de DOS en Linux.
- En el modo de pantalla completa de MS Windows.

En MS Windows, hay varias formas de cambiar al modo de pantalla completa:

- En Windows XP, debemos crear un acceso directo al archivo EXE del programa. Después debemos abrir el cuadro de dialogo Propiedades para el acceso directo, seleccionamos Opciones y *Modo de pantalla completa* en el grupo Uso de la ficha Pantalla (Display options).
- Abrir una ventana de Símbolo del sistema desde el menú de inicio, y oprimimos Alt-Intro para cambiar al modo de pantalla completa. Podemos usar el comando CD (cambiar directorio), navegar hasta el directorio de nuestro archivo EXE y ejecutar el programa escribiendo su nombre. Alt-Intro es un *conmutador*, por lo que si es oprimido de nuevo, el programa regresará al modo de Ventana.

Funcionamiento del texto de video

Hay dos modos de video básicos en los sistemas basados en Intel: el modo de texto y el modo de gráficos. Un programa puede ejecutarse en un modo o en el otro, pero no en ambos al mismo tiempo:

- En el *modo de texto*, los programas escriben caracteres ASCII en la pantalla. El generador de caracteres integado en el BIOS gemera una imagen de mapa de bits para cada carácter. Un programa no puede dibujar líneas y figuras al azar en el modo de texto.
- En el *modo de gráficos*, los programas controlan la apariencia de cada píxel de la pantalla. La operación es algo primitiva, ya que no hay funciones integradas para el dibujo de líneas y figuras. En el modo de gráficos podemos utilizar las funciones integradas para escribir texto en la pantalla, y podemos sustituir distintas fuentes por las funetes integradas. MS Windows proporciona una colección de funciones para dibujar figuras y líneas en modo de gráficos.

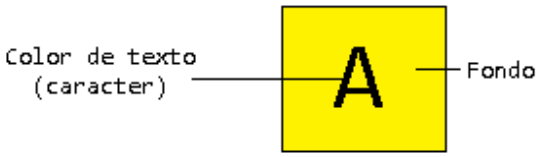
Cuando una computadora inicia en MS-DOS, el controlador de video se establece en Modo de video 3 (texto a color, 80 columnas por 25 filas de manera predeterminada). En el modo de

texto, las filas se enumeran empezando desde la parte superior de la pantalla, fila 0. Cada fila es la altura de una celda de carácter, y utiliza la fuente activa en ese momento. Las columnas se enumeran empezando desde el lado izquierda de la pantalla, columna 0. Cada columna es la anchura de una celda de carácter.

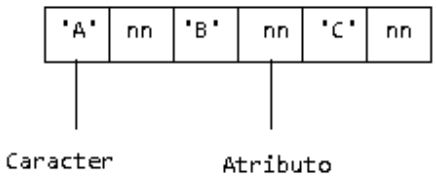
Fuentes Los caracteres se genera desde una tabla residente en memoria de fuente de caracteres. El BIOS permite a los programas reescribir las tablas de caracteres en tiempo de ejecución, por lo que puede mostrarse fuentes personalizadas.

Páginas de texto de video La memoria de video en modo texto se divide en varias páginas de video separadas, cada una de las cuales puede contener una pantalla completa de texto. Los programas pueden mostrar una página mientras escriben texto en otras ocultas, y pueden cambiar rápidamente de una página a otra. En los días de las aplicaciones MS-DOS de alto rendimiento, a menudo era necesario mantener varias pantallas de texto en memoria al mismo tiempo. Con la popularidad actual de las interfaces gráficas, esta característica de las páginas de texto ya no es importante (la función 05h de INT 10h establece la página de video actual). La página de video predeterminada es la página 0.

Atributos Como se ilustra en los siguientes diagramas, a cada carácter de la pantalla se le asigna un byte de atributo, el cual controla el color del carácter (conocido como *color de texto*) y el color de la pantalla detrás del carácter (conocido como *fondo*).



Cada posición en la pantalla de video contiene un solo carácter, junto con su propio *atributo* (color). El atributo se almacena en un byte separado, después del carácter en la memoria. En la siguiente figura, tres posiciones en la pantalla contienen las letras ABC:

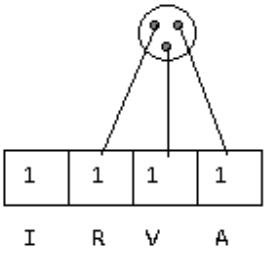


Destello Los caracteres en la pantalla de video pueden destellar. Para ello, el controlador de video invierte los colores de texto y de fondo de un carácter, a una velocidad predeterminada. De manera predeterminada, cuando una PC inicia en el modo de MS-DOS, el destello está habilitado. Es posible desactivarlo mediante una función del BIOS de video. Además, el destello está desactivado de manera predeterminada cuando se abre una ventana de emulación de MS-DOS en MS Windows.

Control del color

Mezcla de colores primarios

Cada pixel de color en una pantalla de video CRT se genera usando tres rayos de electrones separados. Rojo, verde y Azul. Un cuarto canal controla la intensidad total, o brillo del pixel. Por lo tanto, todos los colores de texto disponible se pueden presentar mediante valores binarios de 4 bits, en la siguiente forma (I = intensidad, R = rojo, V = verde, A = azul). El siguiente diagrama muestra la composición de un pixel blanco:



Al mezclar los tres colores primarios pueden generarse nuevos colores. Además, al encender el bit de intensidad podemos hacer los colores mezclados más brillantes.

Ejemplo de mezcla de colores

Mezcla estos colores primarios	Para obtener este color	Agrega el bit de intensidad
rojo + verde + azul	gris oscuro	blanco
verde + azul	cyan	cyan blanco
rojo + azul	magenta	magenta claro
rojo + verde	café	amarillo
(ningún color)	negro	gris oscuro

Los colores primarios estilo MS-DOS y los colores mixtos se compilan en una lista de todos los posibles colores de 4 bits., como se muestra en la siguiente tabla. Cada color en la columna de la derecha tiene activado su bit de intensidad.

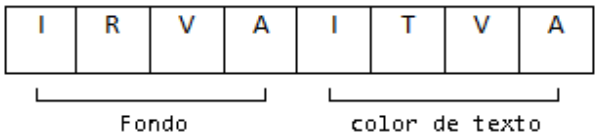
Codificación de texto a color de cuatro bits

IRVA	Color	IRVA	Color
0000	negro	1000	gris

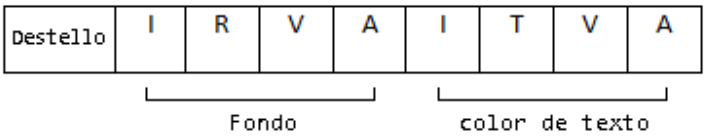
0001	azul	1001	azul claro
0010	verde	1010	verde claro
0011	cyan	1011	cyan claro
0100	rojo	1100	rojo claro
0101	magenta	1101	magenta claro
0110	café	1110	amarillo
0111	gris claro	1111	blanco

Bytes de atributos

En el modo de texto a color, a cada carácter se le asigna un byte de atributos, el cual consiste en dos códigos de colores de 4 bits: fondo y color de texto:



Destello Hay una complicación en este esquema simple de colores. Si el adaptador de video tiene habilitado el destello, el bit superior de color de fondo controla el destello de carácter. Cuando este bit se activa, el carácter destella:



* Destello habilitado

Cuando está habilitado el destello, sólo los colores de baja intensidad en la columna izquierda de la tabla anterior están disponibles como colores de fondo (negro, azul, verde, cyan, rojo, magenta, café y gris claro). El color predeterminado cuando se inicia MS-DOS es el 00000111 binario (gris claro sobre fondo negro).

Construcción de los bytes de atributos Para construir un byte de atributos de video a partir de dos colores (color de texto y de fondo), debemos usar el operador SHL de ensamblador para desplazar los bits del color de fondo cuatro posiciones a la izquierda, y aplicarle OR entre esos bits y el color de texto. Por ejemplo, las siguientes instrucciones crean un atributo de texto gris claro sobre un fondo azul:

```
azul = 1
```

```
grisClaro = 111b

mov bh, (azul SHL 4) OR grisClaro       ; 00010111
```

Las siguientes instrucciones crean caracteres blancos sobre un fondo rojo:

```
blanco = 1111b

rojo = 100b

mov bh, (rojo SHL 4) OR blanco ; 01001111
```

Las siguientes líneas producen letras azules sobre un fondo café:

```
azul = 1

café = 110b

mov bh,((café SHL 4) OR azul)  ; 01100001
```

Las funciones y colores pueden aparecer un poco distintos al ejecutar el mismo programa en diferentes sistemas operativos. Por ejemplo, en Windows 2000 y XP el destello está deshabilitado, al menos que cambiemos a modo de pantalla completa. Lo mismo se aplica para la visualización de gráficos mediante INT 10h.

Funciones de video de INT 10h

La siguiente tabla presenta las funciones INT 10h de uso más frecuente. Hablaremos sobre cada una de ellas por separado, con su propio ejemplo corto. Explicaremos las funciones 0Ch y 0Dh hasta la sección de gráficos.

Funciones selectas de INT 10h.

Número de función	Descripción
0	Establece la pantalla de video a un de los modos de texto o de gráficos
1	Establece las líneas del cursor, con la cual se controla la forma y el tamaño del mismo
2	Posiciona el cursor en la pantalla
3	Obtiene la posición del cursor en la pantalla y su tamaño
6	Desplaza una ventana en la página de video actual hacia arriba,

	sustituyendo las líneas desplazadas con espacios en blanco
7	Desplaza una ventana en la página de video actual hacia abajo, sustituyendo las líneas desplazadas con espacios en blanco
8	Lee el carácter y su atributo en la posición actual del cursor
9	Escribe un carácter y su atributo en la posición actual del cursor
0Ah	Escribe un carácter en la posición actual del cursor, sin cambiar el atributo de color
0Ch	Escribe un pixel de gráficos en la pantalla, en modo de gráficos
0Dh	Lee un color de un pixel de gráficos individual, en la ubicación dada
0Fh	Obtiene información sobre el modo de video
10h	Establece los modos de destello/intensidad
13h	Escribe una cadena en modo de teletipo
1Eh	Escribe una cadena en la pantalla, en modo de teletipo

Es conveniente preservar los registros de propósito general (mediante PUSH) antes de llamara a INT 10h, ya que las distintas versiones del BIOS no son consistentes en los registros que preservan.

Establecer modo de video (00h)

La función 0 de INT 10h nos permite establecer el modo de video actual a uno de los modos de texto o de gráficos. La siguiente tabla presenta los modos de texto que se utilizan con más frecuencia:

Modos de texto de videos reconocidos pot INT 10h.

Modo	Resolución (columnas x filas)	Número de colores
0	40 x 25	16
1	40 x 25	16
2	80 x 25	16
3	80 x 25	16

7°a	80 x 25	2
14h	132 x 25	16

*Mnitor minocromático.

Es conveniente obtener el modo de video actual (función 0Fh de INT 10h) y guardarlo en una variable antes de asignarle un nuevo valor. Después podemos restaurar el modo de video original cuando terminemos nuestro programa. La siguiente tabla muestra cmo establecer el modo de video.

Función 0 INT 10h

Descripción	Establece el modo de video
Recibe	AH = 0 AL = modo de video
Devuelve	Nada

Llamada de ejemplo	`mov ah,0`
	
	`mov al, 3   ; modo de video 3 (texto a color)`

Notas	La pantalla se borra de manera automática, a menos que se active el bit superior en AL, antes de llamar a esta función

Establecer líneas del cursor (01h)

La función 01h de INT 10h, como se muestra en la siguiente tabla, establece el tamaño del cursor de texto. Éste se muestra usando líneas de exploración inicial y final, las cuales pueden controlar su tamaño. Los programas de aplicaciones pueden hacer esto para mostrar el estado actual de una operación. Por ejemplo, un editor de texto podría incrementar el tamaño del cursor cuando se active la tecla BloqNúm; al oprimirla otra vez, el cursor regresa a su tamaño original.

Función 01 de INT 10h

Descripción	Establece las líneas del cursor
Recibe	AH = 01h

	CH = línea superior CL = línea inferior
Devuelve	Nada

Llamada de ejemplo	`mov ah,1`
	
	`mov cx, 0607 ; tamaño predeterminado de cursor a color`
	
	`int 10h`

Notas	La pantalla de video a color utiliza ocho líneas para su cursor

El cursor se describe como una secuencia de líneas horizontales, en donde la línea 0 se encuentra en la parte superior. El cursor a color predeterminado empieza en la línea 6 y termina en la línea 7, como se muestra en la siguiente figura:

0
1
2
3
4
5
6 -> Superior
7 -> Inferior

Establecer la posición del cursor (02h)

La función 2 de INT 10h lcaliza el cursor en la fila y columna específicas en la página de video de elección, como puede verse en la siguiente tabla.

Función 02h de INT 10h

Descripción	Establecer la posición del cursor

Recibe	AH = 2 DH, DL = valores de fila, columna BH = página de video
Devuelve	Nada

Llamada de ejemplo	`mov ah, 2`
	
	`mov dh, 10  ; fila 10`
	
	`mov dl, 20  ; columna 20`
	
	`mov bh, 0  ; página de video 0`
	
	`int 10h`

Notas	Para los modos de 80 x 25, DH = 0 a 24, DL = 0 a 79

Obtener posición y tamaño del cursor (03h)

La función 3 de INT 10h, que se muestra en la siguiente tabla, devuelve la posición de la fila y columna del cursor, así como las líneas inicial y final que determinan el tamaño del cursor. Esta función puede ser bastante útil en los programas en los que el usuario desplaza el cursor alrededor de un menú. Dependiendo de en dónde se encuentre el cursor, sabemos que opción se seleccionó del menú.

Función 03h de INT 10h

Descripción	Obtiene la posición y tamaño del cursor
Recibe	AH = 3 BH = página de video
Devuelve	CH, CL = líneas de exploración inicial y final DH, DL = fila, columna de la ubicación del cursor
Llamada de ejemplo	`mov ah,3`

	mov bh,0 ; página de video 0
	int 10h
	mov cursor, CX
	mov posición, DX

Mostrar y ocultar el cursor Es útil poder ocultar de manera temporar el cursr al mostrar menús, escribir en forma contínua en la pantalla o leer la entrada del ratón. Para ocultar el cursor, podemos establecer el valor de su línea superior a un valor ilegar (grande). Para volver a mostrar el cursor, devolvemos las líneas del cursor a sus valores predeterminados (líneas 6 y 7):

```
OcultarCursor PROC

mov ah, 3                       ; obtiene el tamaño del cursor

int 10h

or ch, 30h                      ; establece la fila superior a un valor
ilegal

mov ah,1                        ; establecer el tamaño del cursor

int 10h

ret

OcultarCursor ENDP

MostrarCursor PROC

mov ah,3                        ; obtiene el tamaño del cursor

int 10h

mov ah,1                        ; establece el tamaño del cursor

mov cx, 0607h                   ; tamaño predeterminado

int 10h

ret

MostrarCursor ENDP
```

Estamos ignorando la posibilidad de que el usuario pueda haber establecido el cursor a un tamaño distinto, antes de ocultarlo. He aquí una versión alternativa de **MostrarCursor** que sólo

borra los 4 bits superiores de CH sin tocar los 4 bits inferiores, en donde se almacenan las líneas del cursor:

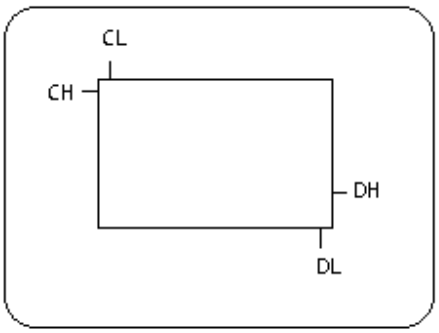
```
MostrarCursor PROC

mov ah,3                        ; obtiene el tamaño del cursor

int 10h

mov ah,1                        ; establece el tamaño del cursor

and ch, 0Fh                     ; borra los 4 bits superiores

int 10h

ret

MostrarCursor ENDP
```

Por desgracia, este método de ocultar el cursor no siempre funciona. Un método alternativo es utilizar la función 02h de INT 10h para posicionar el cursor fuera del borde de la pantalla (fila 25, por ejemplo).

Desplazar ventana hacia arriba (06h)

La función 6 de INT 10h desplaza todo el texto dentro de un área rectangular de la pantalla (*conocida como ventana*) hacia arriba. Una *ventana* se define mediante coordenadas de fila y columna para sus esquinas superior izquierda e inferior derecha. La pantalla predeterminada de MS-DOS tiene las filas numeradas de 0 a 24, empezando desde la parte superior, y las columnas 0 a 79 empezando desde la izquierda. Por lo tanto, una ventana que cubre toda la pantalla completa abarca desde 0,0 hasta 24,79. En la siguiente figura, los registros CH/CL definen la fila y la columna de la esquina superior izquierda, y DH/DL definen la fila y la columna de la esquina inferior derecha. Esta función no tiene un efecto predecible sobre la posición del cursor.

Definición de una ventana mediante el uso de INT 10h.



A medida que se desplaza una ventana hacia arriba, su línea inferior se sustituye por una línea en blanco. Si todas las lineas se desplazan, la ventana se borra (se pone en blanco). Las líneas que se desplazan fuera de la pantalla no pueden recuperarse. La siguiente tabla describe la función 6 de INT 10h.

Función 06h de INT 10h

Descripción	Desplaza la ventana hacia arriba
Recibe	AH = 6 AL = número de lineasa desplazar (0 = todas) BH = atributo de video para el área en blanco CH, CL = fila, columna de la esquina superior izquierda de la ventana DH, DL = fila, columna de la esquina superior derecha de la ventana
Devuelve	Nada

Llamada de ejemplo	`mov ah,6 ; desplaza la ventana hacia arriba`
	`mov al,0 ; toda la ventana`
	`mov ch,0 ; fila superior izquierda`
	`mov cl,0 ; columna superior izquierda`
	`mov dh,24 ;fila inferior derecha`
	`mov dl, 79 ; columna inferior derecha`
	`mov bh,7 ; atributo para el área en blanco`

	int 10h

Ejemplo: escribir texto en una ventana

Cuando la función 6 (o 7) de INT 10h desplaza una ventana, establece los atributos de las líneas desplazadas dentro de la ventana. Si después escribimos texto dentro de la ventana mediante una llamada a una función del DOS, el texto utilizará los mismo colores de texto y de fondo. El siguiente programa (*VenTexto.asm*) demuestra esta técnia:

```
TITLE Ventana de texto a color (VenTexto.asm)

; Muestra una ventana a color y escribe texto en su interior,

INCLUDE Irvine16.inc

.data

mensaje BYTES “Mensaje en la ventana”,0

.code

main PROC

   mov ax,@data

   mov ds,ax

; Desplaza una ventana.

   mov ax, 0600h               ; desplaza la ventana

   mov bh, (blue SHL 4) OR yellow      ; atributo

   mov cx, 050Ah               ; esquina superior izquierda

   mov dx, 0A30h               ; esquina inferior derecha

   int 10h

; Posiciona el cursor dentro de la ventana.

   mov ah,2                    ; establece la posición del cursor

   mov dx,0714h                ; fila 7, col 20

   mov bh,0                    ; página de video 0

   int 10h
```

```
; Escribe texto en la ventana.
   mov dx, OFFSET mensaje
   call WriteString
; Espera un tecleo.
   mov ah,10h
   int 16h
   exit
main ENDP
END main
```

Desplazar ventana hacia abajo (07h)

La función para desplaza la ventana hacia abajo es idéntica a la función 06h, sólo que el texto dentro de la ventana se mueve hacia abajo. Utiliza los mismos parámetros de entrada.

Leer carácetr y atributo (08h)

La función 8 de INT 10h devuelve el carácter y su atributo en la posición actual del cursor. Los programas pueden usarla para leer texto directamente de la pantalla; una técnica conocida como *copiar a la pantalla* (*screen scraping*). Los programas podrían convertir el texto a voz para los usuarios con discapacidad visual.

Función 08h de INT 10h

Descripción	Lee el carácter y el atributo en la posición actual del cursor
Recibe	AH = 8 BL = página de video
Devuelve	AL = código ASCII del carácter AH = atributo del carácter

Llamada de ejemplo	`mov ah, 8`
	
	`mov bh,0 ; página de video 0`

	int 10h
	mov car, al ; guarda el carácter
	mov atrib, ah ; guarda el atributo

Escribir carácter y atributo (09h)

La función 9 de INTh escribe una carácter a color en la posición actual del cursor. Como podemos ver en la siguiente tabla, esta función puede mostrar cualquier carácter ASCII, incluyendo los caracteres gráficos respeciales del BIOS que coinciden con los códigos ASCII del 1 al 31.

Función 09h de INT 10h

Descripción	Escribe un carácter y el atributo
Recibe	AH = 9 AL = código ASCII del carácter BH = página de video BL = atributo CX = cuenta de repetición
Devuelve	Nada

Llamada de ejemplo	mov ah 9
	mov al,'A' ; carácter ASCII
	mov bh,0 ; página de video 0
	mov bl,71h ; atributo (azul sobre gris claro)
	mov cx,1 ; cuenta de repetición

Notas	No avanza el cursor después de escribir el carácter. Puede llamarse en modods de texto y gráficos

La *cuenta de repetición* en CX especifíca cuantas veces se va a repetir el carácter (no debe repetirse más allá de la línea actual de la pantalla). Una vez que se escribe un carácter, hay

que llamar a la función 2 de INT 10h para avanzar el cursor si se van a escribir más caracteres en la misma línea.

Escribir carácter (0Ah)

La función 0Ah de INT 10h escribe un carácter en la pantalla en la posición actual, sin cambiar el atributo actual de la pantalla. Como se muestra en la siguiente tabla, es idéntica a la función 9, sólo que no se especifica el atributo.

Función 0Ah de INT 10h

Descripción	Escribe un carácter
Recibe	AH = 0Ah AL = carácter BH = página de video CX = cuenta de repetición
Devuelve	Nada

Llamada de ejemplo	`mov ah, 0Ah`
	`mov al,'A'  ; carácter ASCII`
	`mov bh,0  ; página de video 0`
	`mov cx,1  ; cuenta de repetición`
	`int 10h`

Notas	No avanza el cursor

Obtener información del modo de video (0Fh)

La función 0Fh de INT 10 devuelve información acerca del modo actual de video, incluyendo el número del modo, el número de columnas en la pantalla y el número de página activa de video, como podemos ver en la siguiente tabla. Esta función es útil al principio de un programa, podemos restablecer el modo de video (con la función 0 de INT 10h) al valor guardado.

Función 0Fh de INT 10h

Descripción	Obtiene la información sobre el modo actual de video
Recibe	AH = 0Fh
Devuelve	AL = modo de pantalla actual AH = número de columnas (caracteres o pixeles) BH = página activa de video

Llamada de ejemplo	`mov ah,0Fh`
	`int 10h`
	`mov modov,al ; guarda el modo`
	`mov clumnas,ah ; guarda las columnas`
	`mov pagina,bh ; guarda la página`

Notas	Trabaja en modo de texto y de gráficos

Establecer modo de destello e intensidad (10h; 03h)

La función 10h de INT 10h tiene varias subfunciones útiles, incluyendo la número 03h, que permite que el bit superior de un atributo de color controle la intenstidad o el destello del carácter. En la siguiente tabla podremos consultar los detalles:

Función 10h de INT 10h, subfunción 03h

Descripción	Establece el modo de destello e intensidad
Recibe	AH = 10h AL = 3 BL = modo de destello (0 = habilita intensidad, 1 = habilita destello)
Devuelve	Nada

Llamada de ejemplo	`mov ah, 10h`
	`mov al,3`
	`mov bl,1 ; habilita el destello`

	`int 10h`
Notas	Cambia el texto en la pantalla, entre el modo de destello y el modo de alta intensidad, el destello solo puede ocurri cuando la aplicación se ejecuta en modo de pantalla completa

Escribir cadena en modo teletipo (13h)

La función 13h de INT 10h, que se muestra en la siguiente tabla, escribe una cadena en la pantalla, en una ubicación especificada por la fila y la columna. La cadena puede contener de manera opcional tanto caracteres como valores de atributos. Esta función puede usarse en modo de texto o de gráficos.

Función 13h de INT 10h

Descripción	Escribe una cadena en modo de teletipo
Recibe	AH = 13h AL = modo de escritura (ver abajo) BH = página de video BL = atributo (si AL = 00h o 01h) CX = longitud de la cadena (cuenta de caracteres) DH, DL = fila, columna de la pantalla ES:BP = segmento: desplazamiento de la cadena
Devuelve	Nada

Llamada de ejemplo	`.data`
	`cadenaColor BYTE 'A',1Fh'B',1Ch,'C',1Bh,'D',1Ch`
	`fila BYTE 10`
	`columna BYTE 20`
	`.code`
	`mov ax, SEG cadenaColor  ; establece el segmento ES`

	mov es,ax
	mov ah, 13h ; escribe la cadena
	mov al, 2 ; modo de escritura
	mov bh,0 ; página de video
	mov cx,(SIZEOF cadenaColor) / 2 ; longitudad cadena
	mov dh, fila ; fila inicial
	mov dl, columna ; columna inicial
	mov bp,OFFSET cadenaColor ; ES:BP apunta a la cadena
	int 10h

Notas	Podemos llamarlo cuando el adaptador de pantalla está en modo de texto o de gráficos. Valores del modo de escritura: • 0 = la cadena sólo contiene códigos de caracteres; el cursor no se actualiza después de la escritura y el atributo está en BL. • 1 = la cadena sólo contiene códigos de caracteres; el cursor se actualiza después de la escritura y el atributo está en BL. • 2 = la cadena contiene código de caracteres y el byte de atributos, alterando uno y uno; la posición del cursor no se actualiza después de la escritura. • 3 = la cadena contiene código de caracteres y el byte de atributos, alterando uno a uno; la posición del cursor se actualiza después de la escritura.

Dibujo de gráficos mediante INT 10h

La función 0Ch de INT 10h dibuja un píxel individual en modo de gráficos. Podríamos utilizarlo para dibujar formas y líneas complejas, pero es demasiado lento. Para aprender los fundamentos, empezaremos con esta función y más adelante veremos cómo dibujar gráficos, escribiendo diectamente en la RAM del video.

Podemos dibujar texto en la pantalla utilizando la función 9h de INT 10h, cuando el adaptador de video se encuentra en modo de gráficos.

Antes de dibujar píxeles, debemos poner el adaptador de video en uno de los modos de gráficos estándar, que se muestra en la siguiente tabla. Cada modo puede establecerse mediante el uso de la función 0 de INT 10h (establecer modo de video).

Modos gráficos de video reconocidos por INT 10h

Modo	Resolución (columnas X filas, en píxeles)	Número de colores
6	640 x 200	2
0Dh	320 x 200	16
0Eh	640 x 200	16
0Fh	640 x 350	2
10h	640 x 350	16
11h	640 x 480	2
12h	640 x 480	16
13h	320 x 200	256
6Ah	800 x 600	16

Coordenadas Para cada modo de video, la resolución se expresa como *horizontal X vertical*, y se mide en píxeles. Las coordenadas de la ventana varían de x = 0, y = 0 en la esquina superior izquierda de la pantalla, *hasta x = XMax -1*, *y = YMAx – 1* en la esquina inferior derecha de la pantalla.

Funciones INT 10h relacionadas con píxeles

Escribir píxel de gráficos (0Ch)

La función 0Ch de INT 10h, como se muestra en la siguiente tabla, dibuja un pixel en la pantalla cuando el controlador de video se encuentra en modo de gráficos. La función 0Ch se ejecuta con mucha lentitud, en especial cuando se dibujan muchos píxeles. La mayoría de las aplicaciones de video escriben directamente a memoria de video después de calcular el número de colores por pixel, la resolución horizontal, etcétera.

Función 0Ch de INT 10h	
Descripción	Escribe un píxel de gráficos

Recibe	AH = 0Ch AL = valor del pixel BH = página de video CX = coordenada x DX = coordenada y
Devuelve	Nada

Llamada de ejemplo	`mov ah, 0Ch`
	`mov al, ValorPixel`
	`mov bh, paginaVideo`
	`mov cx, coord_x`
	`mov dx, coord_y`
	`int 10h`

Notas	La pantalla de video debe estar en modo de gráficos. El rango de valores de los píxeles y los rangos de las coordenadas dependen del modo de gráficos actual. Si el bit 7 se activa en AL, se aplica un XOR al nuevo píxel con el contenido actual del píxel (permitiendo que se borre)

Leer píxel de gráficos (0Dh)

La función 0Dh, que se muestra a continuación, lee un píxel de gráficos de la pantalla, en la posición indicada por la fila y columna, devuelve el valor del pixel en AL.

Función 0Dh de INT 10h

Descripción	Lee un píxel de gráficos
Recibe	AH = 0Dh BH = página de video CX = coordenada x DX = coordenada y

Devuelve	AL = valor del píxel

Llamada de ejemplo	`mov ah, 0Dh`
	`mov bh, 0   ; página de video 0`
	`mov cx, coord_x`
	`mov dx, coord_y`
	`int 10h`
	`mov varloPixel, al`

Notas	La pantalla de video debe estar en modo de gráficos. El rango de valores de los píxeles y los rangos de las coordenadas dependen del modo de gráficos actual

Graficos de mapas de memoria

Ya hemos visto cómo el proceso de dibujar píxeles mediante INT 10h es demasiado lento, excepto para las salida de gráficos más rudimentaria. Se ejecuta una cantidad considerable de código cada vez que el BIOS dibuja un píxel. Ahora podemos ver una manera más eficiente de dibujar gráficos, como se hace en el software profesional. Escribiremos los datos de gráficos directamente a la RAM de video (VRAM), a través de los puertos de entrada-salida.

Modo 13h: 320 x 200, 256 colores

El modo de video 13h es el modo más sencillo de utilizar para los gráficos de mapas de memoria. Los píxeles de la pantalla se asignan como un arreglo bidimensional de bytes, 1 byte por píxel. El arreglo empieza con el píxel en la esquina superior izquierda de la pantalla y continúa a lo largo de la línea superior para 320 bytes. El byte en el desplazamiento 320 se asigna al primer píxel en la segunda línea de la pantalla, que continúa en forma secuencial a lo largo de la pantalla. El resto de las líneas se asignan de manera similar. El último byte en el arreglo se asigna al píxel en la esquina inferior derecha de la pantalla. ¿Por quñe utilizar todo un byte para cada píxel? Porque el byte contiene una eferencia a uno de 256 valores distintos de color.

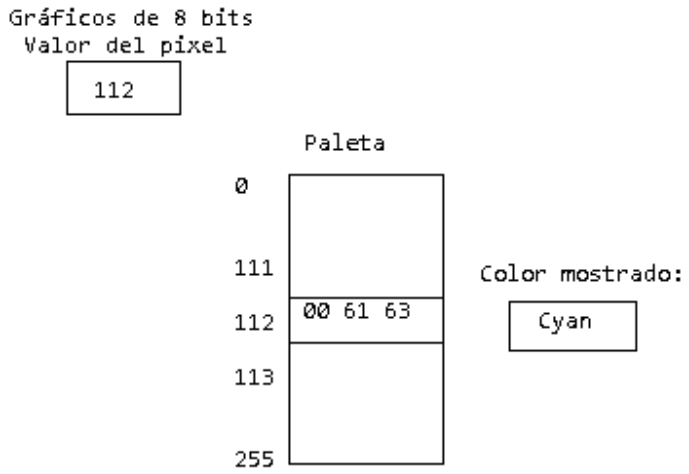
Instrucción OUT Los valores de píxel y de color se trasmiten al hardware del adaptador de video mediante el uso de la instrucción OUT (salida a puerto). La dirección de puerto de 16 bits se asigna a DX; y el valor que se envía al puerto se encuentra en AL, AX o EAX. Por ejemplo, la

paleta de color de video se encuentra en la dirección de puerto 3C8h. Las siguientes instrucciones envían el valor 20h al puerto:

```
mov dx, 3C8h                    ; dirección de puerto

mov al, 20h                     ; valor a enviar

OUT dx,al                       ; envía el valor al puerto
```

Índices de color Lo interesante acerca de los colores en el modo 13h es que cada entero de color no indica directamente un color. En vez de ello, representa a un índice en una tabla de colores, conocida como *paleta* (*imagen mostrada a continuación*).

Conversión de índices de color de píxel a colores de pantalla.



Cada entrada en la paleta consiste en tres valores enteros (de 0 a 63), conocidos como RGB (rojo, verde, azul). La entrada 0 en la paleta de colores controla el color de fondo de la pantalla.

Podemos crear 262,144 colores distintos (64^3) con este esquema. Sólo pueden mostrarse 256 colores distintos en un momento dado, pero nuestro programa puede modificar la paleta en tiempo de ejecución, para variar los colores de la pantalla. Los sistemas operativos modernos, como Windows y Linux, ofrecen (por lo menos) colore de 24 bits, en donde cada

valor RGB tiene un rango de 0 a 255. Ese esquema ofrece 256^3 (16.7 millones) colores distintos.

Colores RGB

Los colores RGB se basan en la mezcla aditiva de la luz, en contraste al método de resta que se utiliza al mezclar la pintura líquida. Por ejemplo, con la mezcla aditiva podemos crear el color negro manteniendo todos los niveles de intensidad de color en cero. Por otro lado, el color blanco se crea estableciendo todos los niveles de color en 63 (máximo). De hecho, y como se demuestra en la siguiente tabla, cuando los tres niveles son iguales obtenemos distintas tonalidades de gris.

Rojo	Verde	Azul	Color
0	0	0	negro
20	20	20	gris oscuro
35	35	35	gris mediano
50	50	50	gris claro
63	63	63	blanco

Los colores puros se cean estableciendo todos menos un nivel de color a cero. Para obtener un color claro, se incrementa los otros dos colores en cantidades iguales. He aquí algunas variedades del color rojo:

Rojo	Verde	Azul	Color
63	0	0	rojo brillante
10	0	0	rojo oscuro
30	0	0	rojo mediano
63	40	40	rosa

Los colores azul brillante, azul oscuro, azul claro, verde brillante, verde oscuro, verde claro se cean de manera similar. Desde luego podemos mezclar pares de colores en otras cantidades para crear colores como magenta y lavanda. A continuación se muestran algunos ejemplos:

Rojo	Verde	Azul	Color
0	30	30	cyan

30	30	0	amarillo
30	0	30	magenta
40	0	63	lavanta

Programación del ratón

Por lo general, el ratón se conecta a la tarjeta madre de la computadora a través de un puerto de ratón PS-2, un puerto serial RS-232, un puerto USB o una conexión inalámbrica. Para poder detectar el ratón, MS-DOS requiere que se instale un programa controlador de dispositivo. MS Windows también cuenta con controladores de ratón integrados, pero por ahora nos concentraremos en las funciones que proporciona MS-DOS.

Los movimientos del ratón se rastrean en una unidad de medida conocida como *mickeys*. Un mickey representa aproximadamente 1/2000 pulgadas de recorrido físico del ratón. Se puede establecer la proporción de mickeys a píxeles para el ratón, cuyo valor predeterminado es de 8 mickeys por cada 8 píxeles horizontales, y 16 mickeys por cada 8 píxeles verticales. También hay un umbral de doble velocidad, cuyo valor predeterminado es de 64 mickeys por segundo.

Funciones INT 33h para el ratón

INT 33h proporciona información acerca del ratón, incluyendo su posición actual, el último botón que se oprimió, la velocidad, etcétera. Podemos usarla para mostrar u ocultar el cursor del ratón. En esta sección veremos algunas de las funciones más esenciales del ratón. INT 33h escribe el número de función en el registro AX, en vez de AH (que es la norma para las interupciones del BIOS).

Restablecer el ratón y obtener el estado

La función 0 de INT 33h restablece el ratón y confirma que esté disponible. El ratón (si se encontró) se centra en la pantalla, su página de visualización se establece a la página de video 0, su puntero se oculta, sus proporciones de mickeys a píxeles y su velocidad se establecen a valores predeterminados. El rango de movimiento del ratón se establece a toda el área de la pantalla. En la siguiente tabla se muestra los detalles:

Función 0 de INT 33h	
Descripción	Restablece el ratón y obtiene su estado
Recibe	AX = 0
Devuelve	Si hay soporte para el ratón disponible, AX = FFFFh y BX =

	números de botones del raton; en caso contrario AX = 0

Llamada de ejemplo	`mov ax,0`
	
	`int 33h`
	
	`cmp eax,0`
	
	`je RatonNoDisponible`
	
	`mov numeroDeBotones,bx`

Notas	Si el ratón era visible antes de esta llamada, se oculta mediante esta función

Mostrar y ocultar el puntero del ratón

Las funciones 1 y 2 de INT 33h, que se muestran en las dos tablas siguientes, muestra y ocultan el punter del ratón, respectivamente. El controlador del ratón mantiene un contador interno, que se incrementa (si es distinto de cero) mediante las llamadas a la función 1, y se decrementa mediante las llamadas a la función 2. Cuando el contador no es negativo, se muestra el puntero de ratón. La función 0 (restablecer puntero del ratón) establece el contador a -1.

Función 1 de INT 33h

Descripción	Muestra el puntero del ratón
Recibe	AX = 1
Devuelve	Nada

Llamada de ejemplo	`mov ax, 1`
	
	`int 33h`

Notas	El controlador del ratón mantiene un conteo del número de veces que se llama a esta función. Suma 1 a su contador interno para mostrar/ocultar

Función 2 de INT 33h

Descripción	Ocultar el puntero del ratón
Recibe	AX = 2
Devuelve	Nada
Llamada de ejemplo	`mov ax, 2` `int 33h`
Notas	El controlador del ratón continúa rastreando la posición del mismo. Resta 1 a su contador interno para mostrar/ocultar

Obtener posición y estado del ratón

La función 3 de INT 33h obtiene la posición y el estado del ratón. En la siguiente tabla se muestran los detalles sobre esta función:

Función de 3 INT 33h

Descripción	Obtiene la posición y el estado del ratón
Recibe	AX = 3
Devuelve	BX = estado de los botones del ratón CX = coordenada X (en píxeles) DX = coordenada y (en píxeles)

Llamada de ejemplo	`mov ax, 3`
	`int 33h`
	`test bx, 1`
	`jne Boton_Izq_Oprimido`
	`test bx, 2`
	`jne Boton_Der_Oprimido`
	`test bx,4`

	`jne Boton_Cent_Oprimido`
	
	`mov coordX, cx`
	
	`mov coordY, dx`

Notas	El estado de los botones del ratón se devuelve en BX de la siguiente manera: Si está activo el bit 0, el botón izquierdo está oprimido; si está activo el bit 1, el botón derecho está oprimido, si está activo el bit 2, el botón central está oprimido.

Conversión de coordenadas de píxel a coordenadas de carácter Las fuentes de texto estándar en MS-DOS son de 8 píxeles de ancho por 16 píxeles de alto, por lo que podemos converitr las coordenadas de píxeles a coordenadas de carácter, dividiendo las primeras entre el tamaño del carácter. Suponiendo que tanto los píxeles como los caracteres empiezan a enumerarse desde cero, la siguiente fórmula convierte una coordenada de píxel P a una coordenada de carácter C, usando la medida de carácter M:

```
C = int(P/M)
```

Por ejemplo, vamos a suponer que los caracteres tienen 8 píxeles de ancho. Si la coordenada X devuelta por la función 3 de INT 33h es 100 (píxeles), la coordenada caería dentro de la posición carácter 12:C = int(100/8).

Establecer la posición del ratón

La función 4 de INT 33h, que se muestra en la siguiente tabla, desplaza la posición del ratón a las coordenadas de píxel X y Y especificadas.

Función 4 de INT 33h

Descripción	Establece la posición del ratón
Recibe	AX = 4 CX = coordenada X (en píxeles) DX = coordenada Y (en píxeles)
Devuelve	Nada

Llamada de ejemplo	`mov ax, 4`
	
	`mov cx, 200   ; posición X`

	`mov dx, 100   ; posición Y`
	
	`int 33h`

Notas	Si la posición se encuentra dentro de un área de exclusión, el ratón no se muestra en la pantalla

Conversión de coordenadas de carácter a coordenadas de pixel Podemos convertir una coordenada de carácter de la pantalla a una coordenada de píxel, usando la siguiente fórmula, en donde C = coordenada de carácter; P = coordenada de píxel y M = medida del carácter:

```
P = C x M
```

En la dirección horizontal, P será la coordenada de píxel del lado izquierdo de la celda del carácter. En la dirección vertical, P será la coordenada de píxel de la parte superior de la celda del carácter. Por ejemplo, si los caracteres tienen 8 píxeles de ancho y deseamos colocar el ratón en la celda del carácter 12, la coordenada X del píxel que se encuentra más a la izquierda de esta celda es 96.

Obtener el estado de los botones (se oprimió/se soltó)

La función 5 devuelve el estado de todos los botones del ratón, así como la posición del último botón que se oprimió. En entorno de programación controlado por eventos, un evento *arrastrar* siempre empieza con un botón oprimido. Una vez que se hace una llamada a esta función para un boton en especial, el estado del botón se restablece, y una segunda llamada a la función no devuelve nada:

Función 5 de INT 33h

Descripción	Obtiene la información acerca de si se oprimió un botón
Recibe	AX = 5 BX = ID del botón (0 = izquierdo, 1 = derecho, 2 = centro)
Devuelve	AX = estado del botón BX = contador de veces que se oprimió el botón CX = coordenada X del último botón que se oprimió DX = coordenada Y del último botón que se oprimió
Llamada de ejemplo	mov ax, 5

	mov bx, 0 ; ID del botón
	int 33h
	test ax, 1 ; ¿Se oprimió el botón izquierdo?
	jz saltar
	mov coord_X, cx ; no – salta
	mov coord_Y, dc

Notas	El estado del ratón se devuelve en AX de la siguiente manera; Si está activo el bit 0, el botón izquierdo está oprimido; si está activo el bit 1, el botón derecho está oprimido; si está activo el bit 2, el botón central está oprimido

La función 6 obtiene la información de cuando se sueltan los botones del ratón, como se muestra en la siguiente tabla. En el programación controlada por eventos, un evento de *clíc* del ratón ocurre cuando se suelta uno de los botones del mismo. De manera similar, un evento *arrastrar* termina cuando se suelta el botón del ratón.

Función 6 de INT 33h

Descripción	Obtiene la información acerca de si se soltó el ratón
Recibe	AX = 6 BX = ID del ratón (0 = izquierda, 1 = derecho, 2 = central)
Devuelve	AX = estado del ratón BX = contador de veces que se soltó el ratón CX = coordenada X del último botón que se soltó DX = coordenada Y del último botón que se soltó

Llamada de ejemplo	`mov ax, 6`
	`mov bx,0 ; ID del botón`
	`int 33h`
	`test ax,1 ; ¿se soltó el botón izquierdo?`

	jz saltar ; no – salta
	
	mov coord_X, cx ; si – guarda las coordenadas
	
	mov coord_Y, dx

Notas	El estado del botón se devuelve en AX de la siguiente manera: Si está activo el bit 0, se soltó el botón izquierdo; si está activo el bit 1, se soltó el botón derecho; si está activo el bit 2, se soltó el botón central

Establecer los límites horizontal y vertical

Las funciones 7 y 8 de INT 33h, como se ilustra en las dos tablas siguientes, nos permiten establecer límites a los lugares a donde se puede mover el ratón en la pantalla. Para ello se establecen las coordenadas máximas y mínimas del cursor del ratón. Si es necesario, el puntero del ratón se mueve de manera que se encuentre dentro de los nuevos límites.

Función 7 de INT 33h

Descripción	Establece los límites horizontales
Recibe	AX = 7 CX = coordenada X mínima (en píxeles) DX = coordenada X mínima (en píxeles)
Devuelve	Nada

Llamada de ejemplo	mov ax, 7
	
	mov cx, 100 ; establece el rango de X a
	
	mov dx, 700 ; (100,700)
	
	int 33h

Funcón 8 de INT 33h

Descripción	Establece los límites verticales
Reci be	AX = 8 CX = coordenada Y mínima (en píxeles) DX = coordenada Y máxima (en píxeles)
Devuelve	Nada

Llamada de ejemplo	`mov ax, 8`
	`int 33h`
	`mov cx, 100 ; establecer el rango Y a`
	`mov dx, 500 ; (100, 500)`
	`int 33h`

Funciones varias del ratón

Hay otras funciones de INT 33h que son útiles para configurar el ratón y controla su comportamiento. No hablaremos con detalle sobre estas función, pero se presenta a continuación:

Función	Descripción	Parámetos de entrada/salida
AX = 0Fh	Establece el número de mickeys por cada 8 píxeles, para el movimiento horizontal y vertical del ratón	Recibe: CX = mickeys horizontales, DX = mickeys verticales. Los valores predeterminados son CX = 8, DX = 16
AX = 10h	Establece el áea de exclusión del ratón (evita que el ratón entre en un rectángulo)	Recibe: CX, DX = coordenadas X, Y de la esquina superieor izquierda, SI, DI = coordenadas X, Y de la esquina inferior derecha
AX = 13h	Establece el umbral de doble velocidad	Recibe: DX = velocidad de umbral en mickeys por segundo (el valor predeterminado es 64)
AX = 1Ah	Establece la sensibilidad y el umbral del ratón	Devuelve BX = velocidad horizontal, CX = velocidad vertical, DX = umbral de velocidad doble
AX = 1Fh	Deshabilita el controlador del ratón	Devuelve: Si no tiene éxito, AX = FFFFh
AX = 20h	Habilita el controlador del ratón	Ninguno
AX = 24h	Obtiene información sobre el ratón	Devuelve FFFFh si hay eror; en caso contrario, devuelve: BH = número mayor de versión, BL = número menor de

		versión, CH = tipo de ratón (1 = bus, 2 = serial, 3 = InPort, 4 = PS/2, 5 = HP); CL = número de IRQ (0 para ratón PS/2)

16 Programación experta en MS-DOS

Definición de segmentos

Los programas escritos para las primeras versiones de MASM tenían que crear definiciones bastante elaboradas para los segmentos de código, datos y pila. Todos los instructores se sintieron aliviados cuando llegaron las directivas simplificadas de segmento (.code, .stack y .data). Sin embargo, también quedó claro que los programadores expertos tal vez preferían la flexibilidad en vez de la simplicidad, y se apegarían a la forma tradicional de hacer las cosas. Si llegamos a este capítulo (y comprendimos todos los capítulos anteriores), estaremos listos para dominar los misteriosos detalles de las directivas explícitas de segmento.

Sin embargo, en primer lugar vamos a explorar las diversas formas en que pueden usarse las directivas simplificadas, sólo en caso que satisfagan nuestras necesidades.

Directivas de segmento simplificadas

Cuando se utiliza la directiva .MODEL, el ensamblador define de manera automática a DGROUP para el segmento de datos cercano. Los segmentos en DGROUP forman los datos cercanos, a los que por lo general, se pueden acceder directamente a través de DS o SS.

La directiva .DATA y .DATA? crean un segmento de datos cercano, que puede ser de hasta 64Kb cuando se ejecuta en modo de direccionamiento real. Se coloca en un grupo especial identificado como DGROUP. el cual está limitado a 64Kb. Cuando .FARDATA y .FARDATA? se utilizan en modelos pequeños y mediano de memoria, el ensamblador crea los segmentos de datos lejanos llamados FAR_DATA y FAR_BSS, respectivamente

Identificación del segmento de una variable Algunas funciones del BIOS y de DOS requieren el uso de un registro de segmentos específico para pasar los datos de los argumentos. Podemos asignar la dirección de un segmento a un registro de segmento, usando el operador SEG. Por ejemplo, el siguiente fragmento de código establece DS al segmento que contiene **varlejana**:

```
mov ax, SEG varlejana

mov ds, ax
```

Segmentos de código Como sabemos, los segmentos de código se definen mediante la directIva .CODE. En un programa en el modelo pequeño de memoria, la directiva .CODE hace que el ensamblador genere un segmento llamado _TEXT. Podemos ver esto en la sección Segments and Groups (segmentos y códigos) de un archivo de listado:

```
_TEXT . . . . .16 Bit 0009     Word Public 'CODE'
```

Esta entrada indica que un segmento de 16 bits llamado _TEXT tiene 9 bytes de longitud. Se alínea en un límite de palabra par, es un segmento público y su clase de segmento es 'CODE'.

En los programas en los modelos mediano, grande y enorme, a cada módulo de código fuente se le asigna un nombre de segmento distinto. El nombre consiste en el nomre del módulo seguido de _TEXT. Por ejemplo en un programa llamado *miProg.asm* que utiliza la directiva .MODEL LARGE, el listado genera la siguiente entrada del segmento de código:

```
MIPROG_TEXT . . . . .16 Bit 0009        Word Public 'CODE'
```

También podemos declarar varios segmentos de código dentro del mismo módulo, sin importar el modelo de memoria. Para ello, hay que agregar un nombre de segmento opcional a la directiva .CODE:

```
.code miCodigo
```

Debemos recordar que si llamamos a los procedimientos de la biblioteca de vínculos de 16 bits, nuestro código debe estar ubicado dentro de un segmento llamado **_TEXT**. Por ejemplo, el siguiente extracto de un programa hace que el enlazador (vinculador) genere un mensaje de *desbordamiento de corrección (fixup overflow):*

```
.code MiCodigo

   mov dx, offset msj

   call WriteString
```

Programa con varios segmentos de código El siguiente programa *MultCodigo.asm* contiene dos segmentos de código. Al no incluir el archivo *Irvine16.inc*, podemos ver todas las directivas de MASM que se utilizan en el programa:

```
TITLE Múltiples segmentos de código (MultCodigo.asm)

; Este programa en el modelo pequeño contiene varios

; segmentos de código

.model small, stdcall

.stack 100h

WriteString PROTO
```

```
.data
msj1 db “Primer mensaje”, 0dh, 0ah, 0
msj2 db “Segundo mensaje”, 0dh, 0ah, “$”
.code
main PROC
   mov ax, @data
   mov ds, ax
   mov dx, OFFSET msj1
   call WriteString            ; llamada NEAR
   call Mostrar                ; llamada FAR
main ENDP
.code OtroCodigo
Mostrar PROC FAR
   mov ah, 9
   mov dx, offset msj2
   int 21h
   ret
Mostrar ENDP
END main
```

En el ejemplo anterior, el segmento **_TEXT** contiene el procedimiento **main** y el segmento **OtroCodigo** contiene el procedimiento **Mostrar**. Podemos observar que este procedimiento debe tener un modificador FAR para indicar al ensamblador que debe generar al tipo de instrucción de llamada que guarda el segmento y desplazamiento actuales en la pila. Como confirmación, podemos ver los nombres de los dos segmentos de código en el archivo de listado *MultCodigo.lst*:

```
OtroCodigo  . . . .16 Bit 0008 Word     Public ‘CODE’
_TEXT . . . . . .16 Bit 0014 Word     Public ‘CODE’
```

Definición explícitas de segmentos

Hay ocaciones en las que tal vez sea preferible crear definiciones explícitas. Por ejemplo, tal vez queramos definir varios segmentos de datos con búferes de memoria adicionales. O tal vez vayamos a enlazar nuestro programa con una biblioteca de objetos que utilice nuestro propios nombres de segmentos propietarios. Por último, tal vez estemos escribiendo un pocedimiento que deba llamarse desde un compilador de un lenguaje de alto nivel, que no utilice los nombres de segmentos de Microsoft.

Un programa con definiciones explícitas de segmentos tiene que realizar dos tareas: en primer lugar, debe establecerse un registro de segmento (DS, ES o SS) a la ubicación de cada segmento para que podamos utilizarlo. En segundo lugar, hay que indicar al ensamblador cómo calcular los desplazamientos de las etiquetas dentro de los segmentos correctos.

Las directivas SEGMENT y ENDS definen el inicio y el fin de un segmento, respectivamente. Un programa puede contener casi cualquier número de segmentos, cada uno de ellos con un nombre único. Los segmentos también pueden agruparse (combinarse). La sintaxis es:

```
nombre SEGMENT [alineación] [combinación] ['clase']

   lista-instrucciones

nombre ENDS
```

- *nombre identifica al segmento*; puede ser únio o puede ser el nombre de un segmento existente.
- *alineación* puede ser BYTE, WORD, DWORD, PARA o PAGE.
- *combinación* puede ser PRIVATE, PUBLIC, STACK, COMMON, MEMORY o AT *dirección*.
- *clase* es un identificador encerrado entre comillas sencillas, que se utiliza para identificar un tipo específico de segmento, como CODE o STACK.

Por ejemplo, he aquí como podría definirse un segmento llamado **DatosExtra**:

```
DatosExtra SEGMENT PARA PUBLIC 'DATA'

   var1 BYTE 1

   var2 WORD 2

DatosExtra ENDS
```

Tipo de alineación

Cuando se van a combinar dos o más segmentos, sus *tipos de alineación* indican al enlazador cómo alinear sus direcciones iniciales. El valor prdeterminado es PARA, el cual indica que el segmento debe empezar en un límite par de 16 bytes. He aquí algunos ejemplos de direcciones hexadecimales de 20 bits que se encuentran en los límites de párrafo. Observamos que el último dígito siempre es cero:

```
0A150 81B30 07460
```

Para crear la alineación especificada, el ensamblador inserta bytes al final de cualquier segmento existente, hasta que se llega a la dirección inicial correcta para el nuevo segmento. A los bytes adicionales se les conoce como *bytes sobrantes*. Esto sólo afecta a los segmentos que se unen a un segmento existente, debido a que el primer segmento en un grupo siempre empieza en un límite de párrafo (en el capítulo 2 vimos que las direcciones de segmentos siempre contienen cuatro bits cero de menor orden implícitos). Existen los siguientes tipos de alineación:

- El tipo de alineación BYTE inicia el segmento en el siguiente byte después del segmento anterior.
- El tipo de alineación WORD inicia el segmento en el siguiente límite de 16 bits.
- DWORD inicia el segmento en el siguiente límite de 32 bits.
- PARA inicia el segmento en el siguiente límite de 16 bytes.
- PAGE inicia el segmento en el siguiente límite de 256 bytes.

Si es probable que un programa se ejecute en un procesado 8086 o 80286, es mejor un tipo de alineación WORD (o mayor) para los segmentos de datos, ya que esos procesadores tienen un bus de datos de 16 bits. Dichos procesadores siempre mueven 2 bytes, el primero de los cuales tiene una dirección con numeración par. Por lo tanto, una variable en un límite par requiere una operación de obtención de datos de la memoria, mientras que una variable límite impar requiere dos. Por otro lado, un procesador IA-32 obtiene 32 bits a la vez, y debe usar el tipo de alineación DWORD.

Tipo de combinación

El tipo de combinación de un segmento indica al enlazador cómo combinar segmentos que tienen el mismo nombre. El tipo predeterminado es PRIVATE, cual indica que dicho segmento no se combinará con ningún otro segmento.

Los tipos de combinación PUBLIC y MEMORY hacen que un segmento se combine con todos los demás segmentos públicos o de memoria con el mismo nombre; en realidad se convierten en un solo segmento. Los desplazamientos de todas las etiquetas se ajustan, de manera que sean relativos al inicio del mismo segmento.

El tipo de combinación STACK se asemeja al tipo PUBLIC, en cuanto a los demás segmentos de pila se combinan con él. MS-DOS inicializa de manera automática el registro SS con el inicio del primer segmento que encuentra con un tipo de combinación de STACK; MS-DOS establece SP a la longitud del segmento (menos 1) cuando se carga el programa. En un programa EXE, debe haber por lo menos un segmento con un tipo de combinación STACK; en caso contrario, el enlazador muestra un mensaje de advertencia.

El tipo de combinación COMMON hace que un segmento empieze en la misma dirección que cualquier otro segmento COMMON con el mismo nombre. En realidad, los segmentos se traslapan unos con otros. Todos los desplazamientos se calculan a partir de la misma dirección inicial, y las variables pueden traslaparse.

El tipo de combinación AT *dirección* nos permite crear un segmento en una dirección absoluta; a menudo se utiliza para los datos cuya ubicación está predefinida en el hardware o sistema operativo. No pueden inicializarse variables o datos, pero se pueden crear nombres de variables que hagan referencia a desplazamientos específicos. Por ejemplo,

```
bios SEGMENT AT 40h

   ORG 17h

   bandera_teclado BYTE ?      ; bandera del teclado de MS-DOS

bios ENDS

.code

   mov ax, bios                ; apunta al segmento BIOS

   mov ds, ax

   and ds:bandera_teclado,7Fh  ; borra el bit superior
```

En este ejemplo se requirió una redefinición del segmento (DS), ya que **bandera_teclado** no se encuentra en el segmento de datos estándar.

*Más adelante se explicará las redefiniciones del segmento al igual que la directiva ORG.

Tipo de clase

El *tipo de clase* de un segmento proporciona otra forma de combinar segmentos, en especial aquellos con distintos nombres. El tipo de clase es una cadena sensibla a mayúsculas y minúsculas, encerrada entre comillas sencillas. Los segmentos con el mismo tipo de clase se cargan juntos, aunque pueden estar en orden distinto al del programa original. Uno de los tipos estándar, CODE, es reconocido por el enlazador y debe usarse para los segmentos que contengan instrucciones. Debemos incluir la etiqueta de este tipo si planeamos usar un depurador.

Directiva ASSUME

La directiva ASSUME indica al ensamblador cómo calcular los desplazamientos de código y etiquetas de datos en tiempo de ensamblado. Por lo general, se coloca justo después de la directiva SEGMENT en el segmento de código. Su sintaxis requiere el nombre de un registro de segmento, seguido de dos puntos y el nombre de un segmento:

```
ASSUME regseg : nombreseg
```

ASSUME en realidad no modifica el valor de un registro de segmento. Esto debe hacerse en tiempo de ejecución, usando instrucciones para asignar valores de segmento a los registros de segmento. Nuestro código puede contener varias directivas ASSUME. Cuando se encuentra una nueva, el ensamblador modifica la manera en que calcula las direcciones a partir de ese momento.

La siguiente directiva ASSUME indica al ensamblador que debe utilizar DS como registro predeterminado para el segmento **datos1**:

```
ASSUME ds:datos1
```

La siguiente instrucción asocia a CS con **miCodigo** y SS se asocia con **miPila**:

```
ASSUME cs:miCodigo, ss:miPila
```

Ejemplo: varios segmentos de datos

Anteriormente en esta sección mostramos un programa que tiene dos segmentos de código. Ahora vamos a crear un programa (MultDatos.asm) que contiene dos segmentos de datos llamados **datos1** y **datos2**. Ambos se declaran con el nombre de clase DATA. La directiva ASSUME asocia a DS con **datos1** y a ES con **datos2**:

```
ASSUME cs:segc, ds:datos1, es:datos2, ss:mipila

datos1 SEGMENT ‘DATA’

datos2 SEGMENT ‘DATA’
```

He aquí un listado completo del programa:

```
TITLE varios segmentos de datos         (MultDatos.asm)

; Este programa muestra cómo declarar de manera explícita

; varios segmentos de datos.

segc SEGMENT ‘CODE’

   ASSUME cs:segc, ds:datos1, es:datos2, ss:mipila

main PROC

   mov ax, datos1              ; apunta DS al segmento datos1

   mov ds, ax

   mov ax, seg val2            ; apunta ES al segmento datos2

   mov es, ax

   mov ax, val1                ; segmento datos1

   mov bx, val2                ; segmento datos2

   mov ax, 4C00h               ; termina programa

   int 21h
```

```
main ENDP

segc ENDS

datos1 SEGMENT ‘DATA’          ; especifica el tipo de clase

   val1 WORD 1001h

datos1 ENDS

datos2 SEGMENT ‘DATA’

   val2 WORD 1002h

datos2 ENDS

mipila SEGMENT para STACK ‘STACK’

   BYTE 100h dup(‘S’)

mipila ENDS

END main
```

Se utiliza dos métodos para establecer los valores de los registros de segmento en tiempo de ejecución. El primero utiliza un nombre de segmento (**datos1**):

```
mov ax, datos1                 ; apunta DS al segmento datos1

mov ds, ax
```

El segundo método utiliza el operador SEG para obtener la dirección de segmento de **val2**:

```
mov ax, SEG val2               ; apunta ES al segmento datos2

mov es, ax
```

El archivo de listado que creó el ensamblador muestra dos variables **val1** y **val2** que tienen los mismo valores (desplazamientos iniciales), pero distintos atributos de segmento:

```
NAME                           Type     Value  Attr

val1 . . . . . . . . . . . . . Word     0000   datos1

val2 . . . . . . . . . . . . . Word     0000   datos2
```

Redefiniciones de segmentos

Una *redefinición de segmentos* es un prefijo de un byte que hace que la instrucción actual utilice un registro de segmento distinto del especificado por la directiva ASSUME al calcular la

dirección efectiva. Por ejemplo, podemos usarlo para acceder a una variable de un segmento distinto del que está asociado actualmente con CS o DS:

```
mov al, cs:var1                ; segmento al que apunta CS

mov al, es:var2                ; segmento al que apunta ES
```

Hay que recalcar que en el modo de direccionamiento real, podemos colocar variables en el segmento de código. ¡Nunca podríamos hacer eso en modo protegido!

La siguiente instrucción obtiene el desplazamiento de una variable en un segmento al que no se le haya especificado la directiva ASSUME para DS o ES:

```
mov bx, OFFSET SegAlt: var2
```

Las referencias múltiples a variables pueden manejarse con más facilidad mediante la inserción de una directiva ASSUME , para modificar de manera temporal las referencias a los segmentos predeterminados:

```
ASSUME ds:SegAlt               ; usa SegAlt por unos momentos

mov ax, SegAlt

mov ds, ax

mov al, var1

.

.

ASSUME ds:data                 ; usa el segmento de datos predeterminado

mov ax, data

mov ds, ax
```

Combinación de segmentos

Los programas extensos deben dividirse en módulos separados para simplificar su edición y depuración. Incluso hasta el código de fuente ubicado en distintos módulos puede combinarse en el mismo segmento. Sólo hay que usar el mismo nomre de segmento en cada módulo y especificar un tipo de combinación PUBLIC. Eso es exactamente lo que ocurre cuando se enlaza un programa de 16 bits con la biblioteca de vínculos Irvine16.inc, usando directivas de segmento simplificadas.

Si utilizamos tipo de alineación BYTE, cada segmento va justo después del anterior. Si utilizamos un tipo de alineación WORD, un segmento irá después de otro segmento en el siguiente límite de palabra par. El tipo de alineación predeterminado es PARA, en el cual cada segmento va en el siguiente límite de párrafo.

Programa de ejemplo Vamos a ver un programa de dos módulos que contiene un segmento de código (SEGC), un segmento de datos (SEGD), y un segmento de pila (SEGP). El módulo principal (main) contiene los tres segmentos; SEGC y SEGD tienen un tipo de combinación PUBLIC. Se utiliza un tipo de alineación BYTE para SEGC, con lo que evita la creación de un hueco entre el código de los dos módulos.

Módulo principal:

```
TITLE Ejemplo de segmentos (módulo principal, Seg2.asm)

EXTRN var2: WORD, subrutina_1 PROTO     ; se sustituye por la siguiente
línea

EXTRN subrutina_1:PROC                  ; requerida por HASH 8.0

segc SEGMENT BYTE PUBLIC ‘CODE’

ASSUME cs:segc, ds:segd, ss:segp

main PROC

   mov ax, segd                         ; inicializa DS

   mov ds, ax

   mov ax, var1                         ; variable local

   mov bx, var2                         ; variable externa

   call subrutina_1                     ; procedimiento externo

   mov ax, 4C00h                        ; sale al SO

   int 21h

main ENDP

segc ENDS

segd SEGMENT WORD PUBLIC ‘DATA’         ; segmento de datos local

   var1WORD 1000h

segd ends

segp SEGMENT STACK ‘STACK’              ; segmento de pila

   BYTE 100h dup(‘S’)

segp ENDS
```

```
END main
```

Submódulo:

```
TITLE ejemplo de segmentos (sumódulo, SEG2A.ASM)
PUBLIC subrutina_1, var2
segc SEGMENT BYTE PUBLIC ‘CODE’
ASSUME cs:segc, ds:segd
subrutina_PROC                          ; se llama desde MAIN
   mov ah,9
   mov dx, OFFSET msj
   int 21h
   ret
subrutina_1 ENDP
segc ENDS
segd SEGMENT WORD PUBLIC ‘DAT’A
var2 WORD 2000h                         ; se accesa desde MAIN
msj BYTE ‘Ahora estamos en subrutina_1’
   BYTE 0Dh, 0Ah, $
segd ENDS
END
```

El enlazador ceo el siguiente archivo MAP, en el cual se muestra un segmento de código, un segmento de datos y un segmento de pila:

```
Start     Stop      Lenght    Name      Class
00000H    0001BH    0001CH    CSEG      Code
0001CH    00035H    0001AH    DSEG      Data
00040H    0013FH    00100H    SSEG      STACK
Program entry point at 0000:0000
```

Estructura de un programa en tiempo de ejecución

Un programador eficiente en lenguaje ensamblador necesita saber mucho acerca de MS-DOS. Esta sección describe a command.com, el prefijo de segmento del programa y la estructura de los programas COM y EXE. Al programar *command.com* que se incluye con MS-DOS y Windows 95/98 se le conoce como procesador de comandos. En Windows 2000 y XP, se llama *cmd.exe*, Interpreta cada comando que se escribe en un símbolo del sistema. Cuando escribimos un comando, se realiza la siguiente secuencia:

1. MS-DOS comprueba si el comando es interno, como DIR, REN o DEL (eliminar). Si lo es, el comando se ejecuta de inmediato mediante una rutina de MS-DOS residente en memoria.

2. MS-DOS busca un archivo que coincida con una extensión de COM. Si el archivo se encuentra en el directorio actual, se ejecuta.

3. MS-DOS busca un archivo que coincida con una extensión de EXE. Si el archivo se encuentra en el directorio actual, se ejecuta.

4. MS-DOS busca un archivo que coincida con una extensión de BAT. Si el archivo se encuentra en el directorio actual, se ejecuta. A un archivo con una extensión BAT se le conoce como archivo de procesamiento por lotes, que es un archivo de texto que contiene comandos de MS-DOS que deben ejecutarse como si se hubiera escrito en la consola.

5. Si MS-DOS no puede encontrar un archivo COM, EXE o BAT que coincida en el directorio actual, busca en el primer directorio de la ruta actual. Si no puede encontrar una coincidencia, procede al siguiente directorio en la ruta y continúa este proceso hasta que se encuentra un archivo que coincida o que se agota la búsqueda de rutas.

Los programas de aplicaciones con extensiones COM y EXE se llaman *programas transientes*. En general, se cargan en la memoria durante el tiempo necesario para ejecutarse; cuando terminan, se libera de la memoria que ocupan. Si es necesario, los programas transientes pueden dejar una porción de su código en memoria al salir, a éstos se les conoce como *programas residentes en memoria* o TSRs.

Prefijo de segmento de programa

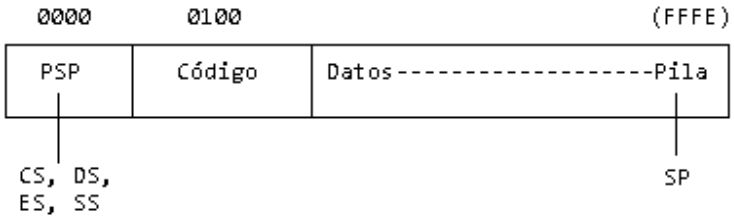
MS-DOS crea un bloque especial de 256 bytes al principio de un programa, a medida que se carga en memoria; a este bloque se le conoce como el Prefijo *de segmento del programa*. La estructura del Prefijo de segmento del programa (PSP) se muestra en la siguiente tabla.

El Prefijo de segmento del programa (PSP).

Desplazamiento	Comentarios
00 – 15	Apuntadores y direcciones de vectores de MS-DOS
16 - 2B	Reservado por MS-DOS
2C – 2D	Dirección de segmento de la cadena de entorno actual
2E – 5B	Reservado por MS-DOS
5C – 7F	Bloques de control de archivos 1 y 2; los utilizan principalmente los programas previos a MS-DOS 2.0
80 – FF	Área de transferencia de disco predeterminada y una copia de la cola de comandos actual de MS-DOS

Programas COM

Existen dos tipos de programas transientes, los cuales se identifican según su extensión de archivo (COM o EXE). Un programa COM es una imagen binaria sin modificar de un programa en lenguaje máquina. MS-DOS lo carga en memoria en la dirección de segmento más baja que esté disponible , y se crea un PSP en el desplazamiento 0. El código, los datos y la pila se almacenan en el mismo segmento físico (y lógico). El programa puede tener hasta 64K de longitud, menos el tamaño del PSP y dos bytes reservados al final de la pila. Como se muestra en el siguiente diagrama, todos los registros de segmento se establecen a la dirección base del PSP. El área de código empieza en el desplazamiento 100h, y el área de datos sigue justo después del código. El área de la pila se encuentra al final del segmento, ya que MS-DOS inicializa a SP con FFFEh:



Veamos un programa simple escrito en formato COM. MASM requiere que un programa COM utilice el modelo *diminuto* (tiny) de memoria. Además, debe usarse la directiva ORG para establecer el contador de la ubicación inicial para el código del programa al desplazamiento 100h. Esto deja 100h bytes disponibles para el PSP, que ocupa ubicaciones desde 0 hasta 0FFh:

```
TITLE Programa Hola en formato COM (HolaCom.asm)

.model tiny
```

```
.code
org 100h                        ; debe estar antes de main
main PROC
   mov ah,9
   mov dx, OFFSET mensaje_hola
   int 21h
   mov ax, 4C00h
   int 21h
main ENDP
mensaje_hola BYTE ‘Hola, mundo!’, 0dh, 0ah, ‘$’
END main
```

Por lo general, las variables se encuentran después del procedimiento principal (main), ya que no hay un segmento separado para los datos. Si colocamos los datos en la parte superior del programa, la CPU trataría de ejecutar los datos. Una alternativa es colocar una instrucción JMP al principio para que salte por encima de los datos hasta la primera instrucción real:

```
TITLE Programa Hola en forma COM (HolaCom.asm)
.model tiny
.code
.org 100h                       ; debe estar antes del punto de entrada
main PROC
   jmp inicio                   ; salta los datos
mensaje_hola BYTE ‘Hola, mundo’, 0dh, 0ah, ‘$’
inicio:
   mov ah,9
   mov dx, OFFSET mensaje_hola
   int 21h
   mov ax, 4C00h
```

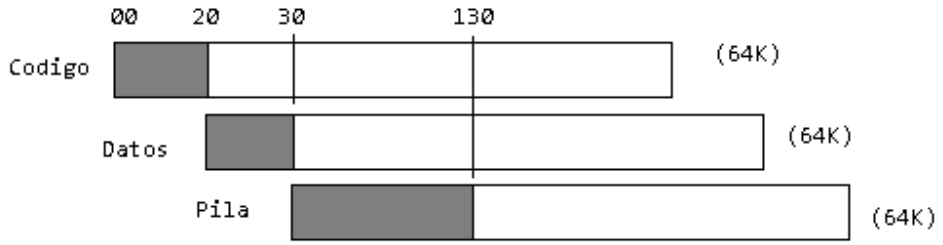
```
   int 21h

main ENDP

END main
```

El enlazador de Microsoft requiere el parámetro /T para indicarle que debe crear un archivo COM en vez de un archivo .EXE. Los programas COM siempre son más pequeños que sus contrapartes EXE; por ejemplo, HolaCom.asm es de sólo 17 bytes cuando se almacena en el disco. No obstante, al estar en memoria, un programa COM ocupa todo un segmento de memoria de 64KB, ya sea que se necesite el espacio o no. Los programas COM no se diseñaron para ejecutarse en un entorno multitarea.

Programas EXE

Un programa EXE se almacena en el disco con un encabezado EXE, seguido de un módulo de carga que contiene el programa en sí. En realidad, el encabezado del programa no se carga en memoria; en vez de ello, contiene información que MS-DOS utiliza para cargar y ejecutar el programa.

Cuando MS-DOS carga un programa EXE, se crea un prefijo de segmento del programa (PSP) en la primera sección disponible, y el programa se coloca en memoria justo encima del prefijo. A medida que MS-DOS decodifica el encabezado del programa, asigna a DS y ES la dirección de carga del programa, también conocida como *Prefijo de segmento del programa* (PSP). CS e IP se establecen con el punto de entrada del código del programa, desde donde el programa empieza a ejecutarse. SS se establece al inicio del segmento de pila, y SP se establece con el tamaño de la pila. He aquí un diagrama que muestra los segmentos de código, datos y pila que se traslapan:



En este programa, el área del código es de 20h bytes, el área de datos es de 10h bytes y el área de la pila es de 10h bytes.

Un programa EXE puede contener hasta 65,535 segmentos, aunque sería inusual tener tantos. Si un programa tiene varios segmentos de datos, por lo general, el programador tiene que establecer manualmente a DS o ES con cada nuevo segmento.

Uso de la memoria

La cantidad de memoria que utiliza un programa EXE se especifica mediante su encabezado; en especial, los valores para el número mínimo y máximo de párrafos (16 bytes cada uno)

necesarios en memoria después del área de código, para manejar variables y la pila en tiempo de ejecución. De manera predeterminada, el enlazador establece el valor máximo a 65,535 párrafos, que es más memoria de la que puede haber disponible en MS-DOS. Por ende, cuando se carga el programa, MS-DOS asigna de manera automática la memoria que haya disponible.

La asignación máxima puede establecerse cuando se enlaza un programa, mediante el uso de la opción /CP. Aquí se muestra para un programa llamado *prog1.obj*. El número 1024 se refiere al número de parámetros de 16 bytes, expresado en decimal:

```
link16 /cp:1024 prog1;
```

Los valores del encabezado EXE pueden modificarse después de compilar un programa EXE, usando el progama **exehdr** que se incluye con el ensamblador de Microsoft. Por ejemplo, el comando para establecer la asignación máxima a 400h párrafos (16,384 bytes) para un programa llamada *prog1.exe* es:

```
exehdr prog1 /max 400
```

Exehdr puede mostrar importantes estadísticas acerca de un programa. A continuación se muestra resultados de ejemplo que describen el programa *prog1.exe*, después de enlazarlo con la asignación máxima establecida en 1024 párrafos:

PROG1	**(Hex)**	**(Dec)**
EXE size (bytes)	876	2166
Minimum load size (bytes)	786	1926
Overlay number	0	0
Initial CS:IP	0000:0010	16
Initial SS:SP	0068:0100	256
Minimum allocation (para)	11	17
Maximum allocation (para)	400	1024
Header size (para)	20	32
Relocation size offset	1E	30
Relocation entries	1	1

Encabezado EXE

MS-DOS utiliza el área de encabezado de un programa EXE para calcular correctamente las direcciones de los segmentos y otros componentes. El encabezado contiene la siguiente información:

- Una tabla de reubicación, la cual contiene direcciones que deben calcularse cuando se carga el programa.
- El tamaño del archivo del programa EXE, que se mide en unidades de 512 bytes.
- Asignación mínima: el número mínimo de párrafos de memoria que deben reservarse después del área de código del programa. Parte de este almacenamiento podría utilizarse para un montón de datos en tiempo de ejecución que contengan datos dinámicos.
- Asignación máxima: el número máximo de párrafos necesarios por encima del programa.
- Valores iniciales para dar a los registros IP y SP.
- *Desplazamiento* (medido en párrafos de 16 bytes) de los segmentos de pila y de código, a partir del inicio del módulo de carga.
- Una *suma de comprobación* de todas las palabras en el archivo, que se utiliza para atrapar errores de datos cuando el programa se carga en memoria.

Manejo de interrupciones

En esta sección hablaremos sobre las formas de personalizar el BIOS y el MS-DOS mediante la instalación de *manejadores de interrupciones (rutinas de servicio de interrupciones)*. Como vimos en capítulos anteriores, el BIOS y el MS-DOS contienen manejadores de interrupciones que simplifican la entrada/salida, así como las tareas básicas del sistema. Vimos muchos de éstos: las rutinas INT 10h para la manipulación del video, las rutinas INT 16h para el teclado, los servicios INT 21h de MS-DOS, etcétera. Pero una parte igualmente importante del sistema operativo es su conjunto de manejadores de interrupciones, que responden a las interrupciones del hardware. MS-DOS nos permite sustituir cualquiera de estas rutinas de servicio con nuestras propias rutinas.

Limitaciones: los manejadores de interrupciones que presentamos en este capítulo funcionan sólo cuando nuestra computadora se inicia en modo de MS-DOS. Podemos hacer esto usando Windows 95, 98, pero no Windows NT, 2000 o XP. Estos últimos sistemas operativos enmascaran el hardware del sistema de los programas de aplicaciones, para obtener una mayor estabilidad, y seguridad del sistema. Si el SO permitiera que dos programas que se ejecutan al mismo tiempo modificaran las configuraciones internas en el mismo dispositivo hardware, los resultados serían cuando menos impredecibles.

Existen varias razones para escribir un manejador de interrupciones. Tal vez queramos que nuestro progama se active al oprimir una tecla *activa*, incluso aunque el usuario esté ejecutando otra aplicación. Por ejemplo, el SideKick de Borland fue uno de los primeros programas en los que aparecía un bloc de notas o una calculadora cada vez que se oprimía una combinación de teclas activas.

Podemos sustituir uno de los manejadores de interrupciones predeterminados de MS-DOS para ofreser servicios más completos. Por ejemplo, la interrupción de *división* entre *cero* se activa cuando la CPU trata de dividir un número entre cero, pero no hay una manera estandar para que un programa se recupere.

Podemos sustituir el manejador de errores críticos de MS-DOS o el manejador de Ctrl-Inter con nuestro propio manejador. El manejador de errores críticos predeterminado de MS-DOS hace que un programa aborte y regrese a MS-DOS. Nuestro manejador podría recuperarse de un error y dejar que el usuario continuara ejecutando el programa de aplicación actual.

Una rutina de servicio de interrupciones escrita por el usuario puede manejar las interrupciones de hardware en forma más efectiva que MS-DOS. Por ejemplo, el manejador de comunicación asíncrona de la PC (INT 14h) no usa búfer en las operaciones de entrada/salida. Esto significa que un carácter de entrada se pierde si no se copia del puerto antes de que llegue otro carácter. Un programa residente en memoria podría esperar a que un carácter entrante generara una interrupción de hardware, recibir el carácter del puerto y almacenarlo en un búfer circular. Esto evita que un programa de aplicación invierta su valioso tiempo en comprobar el puerto serial en forma repetida, descuidando otras letras.

Tabla de vectores de interrupción La clave de la flexibilidad de MS-DOS recae en la tabla de vectores de interrupción que se encuentra en los primeros 1024 bytes de RAM (ubicaciones 0:0 a 0:03FF). La siguiente tabla contiene un corto ejemplo de entradas en la tabla de vectores. Cada entrada en la tabla (conocida como vector de interrupción) es una dirección tipo segmento-desplazamiento de 32 bits, que apunta a una de las rutinas de servicio existentes.

Ejemplo de tabla de vectores de interrupciones.

Número de interrupción	Desplazamiento	Vectores de interrupción
00-03	0000	02C1:5186 0070:0C67 0DAD:21CB 0070:0C67
04-07	0010	0070:0C67 F000:FF54 F000:837B F000:837B
08-0B	0020	0D70:022C 0DAD:2BAD 0070:0325 0070:039F
0C-0F	0030	0070:0419 0070:0493 0070:050D 0070:0C67
10-13	0040	C000:0CD7 F000:F84D F000:F841 0070:237D

En cualquier computadora dada, los valores de los vectores variarán debido a las distintas versiones del BIOS y de MS-DOS. Cada vector de interrupción corresponde a un número de interrupción. En la tabla, la dirección del manejador de INT 0 (división entre cero) es 02C1:5186h. Para obtener el desplazamiento de cualquier vector de interrupción, se

multiplica su número de interrupción por 4. Por ende, el desplazamiento del vector para INT 9h es 9*4, o 0024 hexadecimal.

Ejecución de los manejadores de interrupciones Un manejador de interrupciones puede ejecutarse en una de dos formas (1) un programa de aplicación que contenga una instrucción INT podría producir una llamada a la rutina, a lo cual se conoce como *interrupción de software*; (2) una *interrupción de hardware* ocurre cuando un dispositivo de hardware (puerto asíncrono, teclado, temporizador, etcétera). envía una señal al chip Controlador de interrupciones programable.

Interrupciones de hardware

Una interrupción de hardware se genera mediante el Controlador *de interrupciones programable* (PIC) Intel 8259, el cual indica a la CPU que debe suspender la ejecución del programa actual y ejecutar una rutina de servicio de interrupción. Por ejemplo, un carácter del teclado que espera en el puerto de entrada se perdería si la CPU no lo guarda, o los caracteres recibidos del puerto serial se perderían si no fuera por una rutina controlada por interrupciones, que almacena en un búfer.

En ocaciones, los programas deben deshabilitar las interrupciones de hardware al realizar operaciones delicadas en los registros de segmento y pila. La instrucción CLI (*borra bandera de interrupción*) deshabilita las interrupciones, y la instrucción STI (*establece bandera de interrupción*) habilita las interrupciones.

Niveles de IRQ Las interrupciones pueden activarse por una variedad de dispositivos en una PC, incluyendo los que se presentan en la siguiente tabla. Cada dispositivo tiene una prioridad, basada en su *nivel de petición de interrupción* (IRQ). El nivel 0 tiene la prioridad más alta, y el nivel 15 tiene la más baja. Una interrupción de un nivel más bajo no puede interrumpir a una de nivel más alto que se encuentre en progreso. Por ejemplo, si el puerto de comunicaciones 1 (COM 1) trata de interrumpir el manejador de interrupciones del teclado, tendría que esperar hasta que este último terminara. Además, dos o má peticiones de interrupcion simultaneas se procesan de acuerdo con sus niveles de prioridad. La programación de interrupciones se maneja mediante el PIC 8259.

Asignación de IRC (Bus ISA)

IRQ	Número de interrupción	Descripción
0	8	Temporizador del sistema (18.2 veces/segundo)
1	9	Teclado
2	0Ah	Controlador de interrupciones programable
3	0Bh	COM2 (puerto serial 2)

4	0Ch	COM1 (puerto serial 1)
5	0Dh	LPT2 (puerto paralelo 2)
6	0Eh	Controlador de disco flexible
7	0Fh	LPT1 (puerto paralelo 1)
8	70h	Reloj CMOS en tiempo real
9	71h	(Se redirige hacia INT 0Ah)
10	72h	(Disponible) tarjeta de sonido
11	73h	(Disponible) tarjeta SCSI
12	74h	Ratón PS/2
13	75h	Coprocesador matemático
14	76h	Controlador de disco duro
15	77h	(Disponible

Vamos a utilizar el teclado como un ejemplo: cuando se oprime una tecla, el PIC 8259 envía una señal INTR a la CPU y le pasa el número de interrupción; si la interrupciónes externas no se encuentran deshabilitadas, la CPU hace lo siguiente, en secuencia:

1. Mete el registro Flags en la pila.
2. Borra la bandera de Interrupción, evitando cualquier otra interrupción de hardware.
3. Mete los valores actuales de CS e IP en la pila.
4. Localiza la entrada en la tabla de vectores de interrupción para INT 9 y coloca esta dirección en CS e IP.

A continuación, se ejecuta la rutina del BIOS para INT 9 y realiza lo siguiente, en secuencia:

1. Rehabilita las interrupciones de hardware, para que el temporizador del sistema no se vea afectado.
2. Introduce un código de exploración del puerto del teclado, trata de convertirlo en un carácter ASCII o asigna un código ASCII igual a cero. Después almacena el código de exploración y el código ASCII en el búfer del teclado, un búfer circular de 32 bytes en el área de datos del BIOS.
3. Ejecuta una instrucción IRET (retorno de interrupción), la cual saca a IP, CS y el registro Flags de la pila. El control regresa al programa que se estaba ejecutando cuando ocurrió la interrupción

Instrucciones de control de interrupciones

La CPU tiene una bandera llamada *bandera de interrupción* (IF), la cual controla la forma en que la CPU responde a las interrupciones externas (hardware). Si la bandera de interrupción se activa (IF = 1), decimos que las interrupciones están *habilitadas*; si la bandera se borra (IF = 0), entonces las instrucciones están *deshabilitadas*.

Instrucción STI La instrucción STI habilita las interrupciones externas. Por ejemplo, el sistema responde a la entrada del teclado suspendiendo un programa en proceso, y hace lo siguiente: Llama a INT 9, que almacena la tecla presionada en un búfer y después regresa al programa actual. Por lo general, se habilita la bandera Interrupción. En caso contrario, el temporizador del sistema no calcularía la hora y fecha en forma aporpiada, y se perderían las siguientes teclas introducidas.

Instrucción CLI La instrucción CLI deshabilita las interrupciones externas. Debe utilizarse con precaución; sólo cuando se vaya a realizar una operación crítica, una que no pueda interrumpirse. Por ejemplo, supongamos que nuestro código se interrumpió cuando estaba en el proceso de modificar los valores SS y SP. Nuestro registro SS podría apuntar a un nuevo segmento de pila, mientras que el apuntador de pila tendría que actualizarse:

```
mov ax, mipila

mov ss,ax                       ; restablece SS

; ¡SE INTERRUMPE AQUÍ!

mov sp,100h                     ; restablece SP
```

Para estar seguros, hay que deshabilitar las interrupciones borrando la bandera Interrupción (CLI) y habilitarlas usando STI:

```
cli                             ; deshabilita las interrupciones

mov ax, mipila                  ; restablece SS

mov ss, ax

mov sp, 100h                    ; restablece SP

sti                             ; rehabilita las interrupciones
```

Las interrupciones no deben deshabilitarse por más de unos cuantos milisegundos en un momento dado, o podría perderse los datos de teclas presionadas y el temporizador del sistema reduciría su velocidad. Cuando la CPU responde a un manejador de interrupciones, las demás interrupciones se deshabilitan de inmediato. Las rutinas de servicio de interrupciones de MS-DOS y del BIOS rehabilitan las interrupciones tan pronto como empiezan a ejecutarse.

Escritura de un manejador de interrupciones personalizado

Uno podría preguntarse para qué existe la tabla de vectores de interrupción. Desde luego podríamos llamar a los procedimientos específicos en ROM para procesar las interrupciones.

Los diseñadores de la IBM-PC querían tener la capacidad de realizar modificaciones y correcciones a las rutinas del BIOS, sin tener que sustituir los chips de ROM. Al tener una tabla de interrupción, era posible sustituir las direcciones en la tabla para que apuntaran a procedimientos en la RAM.

Cada dirección en la tabla de vectores de interrupción apunta a un procedimiento conocido como *manejador de interrupciones* o *rutina de servicio de interrupciones* (ISR). Los programas de aplicaciones pueden sustituir una dirección en la tabla con otra que apunte a un nuevo manejador de interrupciones. Por ejemplo, podríamos escribir un manejador de interrupción personalizado para el teclado. Tendría que haber una razón muy fuerte para hacerlo, debido al esfuerzo implicado. Una alternativa más conveniente sería que un manejador de interrupciones llamara directamente a la interrupción INT 9 predeterminada del teclado para leer una tecla presionada del puerto del teclado. Una vez que se colocara la tecla en el búfer de escritura adelantada del teclado, podríamos manipular su contenido.

Las funciones 25h y 35h de INT 21h permiten instalar manejadores de interrupciones. La función 35h (obtener vector de interrupción) devuelve la dirección tipo segmento-desplazamiento de un vector de interrupción. Para llamar a la función se coloca el número de interrupción deseado en AL. MS-DOS devuelve el vector de 32 bits en ES:BX. Por ejemplo, las siguientes instrucciones obtienen el vector INT 9:

```
.data

guardarInt9 LABEL WORD

DWORD ?                         ; aquí almacena la dirección anterior

                                ; de INT9

.code

mov ah, 35h                     ; obtiene el vector de interrupción

mov al, 9                       ; para INT 9

int 21h                         ; llama a MS-DOS

mov guardarINT9, BX             ; guarda el desplazamiento

mov guardarInt9+2, ES           ; guarda el segmento
```

La función 25h de INT 21h (establecer vector de interrupción) nos permite sustituir un manejador de interrupciones existente con uno nuevo. Para llamar a esta función, colocamos el número de interrupción en AL y la dirección segmento-desplazamiento de nuestro propio manejador de interrupciones en DS:DX. Por ejemplo,

```
mov ax, SEG tecl_retn           ; manejador del teclado

mov ds, ax                      ; segmento
```

```
mov dx, OFFSET tecl_rtn        ; desplazamiento
mov ah, 25h                    ; establece el vector de interrupción
mov al, 9h                     ; para INT 9h
int 21h
.
.
tecl_rtn PROC                  ; (aquí empieza el nuevo manejador de
                               ; interrupciones)
```

Ejemplo: manejador de Ctrl-Inter

Si el usuario oprime Ctrl-Inter cuando un programa de MS-DOS espera datos de entrada, el control pasa al procedimiento del manejador de interrupciones predeterminado para INT 23h. El manejador de Ctrl-Inter predeterminado termina el programa que esté ejecutando. Esto puede dejar al programa en un estado inestable, ya que se podrían quedar archivos abiertos, la memoria podría quedar sin liberarse, etcétera. No obstante, es posible sustituir nuestro propio código en el manejador de INT 23h y evitar que el programa se detenga. El siguiente programa instala un manejador simple de Ctrl-Inter:

```
TITLE manejador de Ctrl-Inter  (CtrlInter.asm)
; Este programa instala su propio manejador de Ctrl-Inter y
; evita que el usuario utilice Ctrl-Inter (o Ctrl-C)
; para detener el programa. Recibe e imprime en pantalla
; las teclas presionadas hasta que se oprime Esc.
INCLUDE Irvine16.inc
.data
MsjInter BYTE “INTER”,0
msg BYTE “Demostracion de Ctrl-Inter.”
   BYTE 0dh, 0ah
   BYTE “Este programa deshabilita Ctrl-Break (Ctrl-C). Oprima cualquier”
   BYTE 0dh, 0ah
   BYTE “tecla para continuar, u oprima ESC para terminar.”
```

```
   BYTE 0dh, 0ah
.code
main PROC
   mov ax, @data
   mov ds, ax
   mov dx, OFFSET msg          ; muestra mensaje de bienvenida
   call WriteString
instalar_manejador:
   push ds                     ; guarda DS
   mov ax, @code               ; inicializa DS a segmento de código
   mov ds, ax
   mov ah, 25h                 ; establece vector de interrupción
   mov al, 23h                 ; para la interrupción 23h
   mov dx, OFFSET manejador_inter
   int 21h
   pop ds                      ; restaura DS
L1: mov ah, 1                  ; espera una tecla, la imprime en pantalla
   int 21h
   cmp al, 1Bh                 ; ¿se oprimió ESC?
   jnz L1                      ; no: continúa
   exit
main ENDP
; El siguiente procedimiento se ejecuta cuando
; se oprime Ctrl-Break (Ctrl-C). Todos los registros deben preservarse.
manejador_inter PROC
   push ax
```

```
    push dx
    mov dx, OFFSET MsjInter
    call WriteString
    pop dx
    pop ax
    iret
manejador_inter ENDP
END main
```

El procedimiento **main** inicializa el vector de interrupción para INT 23h. Los parámetros requeridos de entrada para la función 25h de INT 21h son:

- AH = 25h.
- AL = Interrupción que se va a manejar (23h).
- DS:DX = dirección tipo segmento/desplazamiento del nuevo manejador de Ctrl-Inter.

El ciclo principal de este programa simplemente recibe e imprime las teclas presionadas en pantalla hasta que se prime Esc.

En algunos sistemas, tal vez tengamos que oprimir Ctrl-C en vez de Ctrl-Inter para activar el mensaje del manejador de Ctrl-Inter

El procedimiento **manejador_inter** se ejecuta al oprimir Ctrl-Inter; muestra un mensaje llamando a WriteString y regresa de inmediato al programa que hizo la llamada. Cuando la instrucción IRET (retorno de interrupción) se ejecuta al final de manejador_inter, el control regresa al programa principal y se reinicia cualquier función de MS-DOS que hata estado en progreso cuando se oprimió Ctrl-Inter. En general, podemos llamar a cualquier interrupción MS-DOS desde el interior de un manejador Ctrl-Break. Debemos preservar todos los registros en un manejador de interrupciones.

No hay que restaurar el vector INT 23h, ya que MS-DOS lo hace de manera automática cuando termina un programa. MS-DOS almacena el vector original en el desplazamiento 000EFh, en el prefijo de segmento del programa.

Programas TSR (Terminar y permanecer residente)

Un programa TSR (*terminar y permanecer residente*) se instala en memoria y permanece ahí hasta que lo elimina un software de utiliería especial o cuando se reinicia la computadora. Un TSR permanece dormido hasta que se activa mediante algún evento, como presionar una tecla.

En los primeros días de los TSRs, surgían problemas de compatibilidad cuando dos o más progamas sustituían el mismo vector de interrupción. Los programas antiguos hacían que el vector apuntara a su propio programa y no proporcionaban una cadena de avance a los demás programas que utilizaban el mismo vector. Después, para remediar este problema, los autores de los TSRs guardaban el vector existente para la interrupción que estaban sustituyendo y proveían una cadena de avance al manejador de interrupciones original, una vez que su propio procedimiento terminaba de lidiar con la interrupción. Desde luego que esto fue una mejora en comparación con el método anterior, pero significaba que el último TSR en instalarse tenía de manera automática la prioridad más alta en cuanto al manejo de la interrupción. Esto significaba que los usuarios algunas veces debían tener cuidado de cargar los programas TSR en un orden específico. Cuando las aplicaciones de MS-DOS se utilizaban demasiado, existían herramientas de programación comerciales para administrar los diversos programas residentes en memoria.

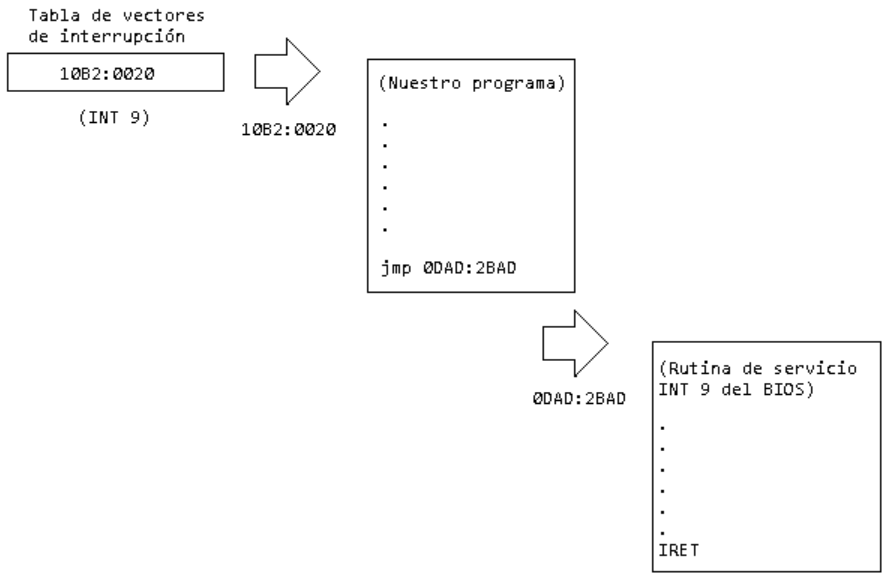
Ejemplo del teclado

Supongamos que escribimos una rutina de servicio de interrupciones que puede inspeccionar cada carácter que se escribe en el teclado y que puede almacenarlo en la ubicación 10B2:0020. Para instalar la ISR, obtene,ps el vector actual de INT 9 de la tabla de vectores de interrupción, lo guardamos y sustitumos la entrada en la tabla con la dirección de nuestra ISR.

Cuando se oprime una tecla, se transfiee un byte individual al controlador del teclado hacia el puerto de teclado de la computadora, y se activa una interrupción de hadware. El PIC 8259 pasa el número de la interrupción a la CPU, ésta salta a la dirección INT 9 en la tabla de vectores de interrupción, la dirección de nuestra ISR. Nuestro procedimiento recibe una oportunidad para inspeccionar el byte del teclado. Cuando nuestro manejador de teclado termina. ejecuta un salto hacia el procedimiento original del manejador de teclado del BIOS.

Este proceso de almacenamiento se muestra en la siguiente figura. Las direcciones son hipotéticas. Cuando termina la rutina INT 9h del BIOS, la instrucción IRET saca el registro Flags de la pila y devuelve el control al programa que se estaba ejecutando cuando se oprimió el carácter.

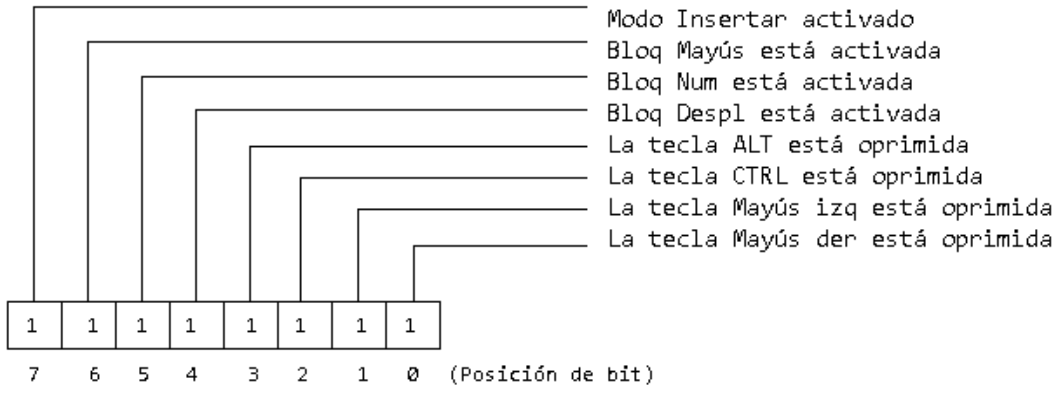
Vectorización de una interrupción.



Aplicación: el programa No_reinicio

Un tipo simple de programa residente en memoria debe evitar que el sistema se reinicie mediante las teclas Ctrl-Alt-Supr. Una vez que nuestro programa se instala en memoria, el sistema sólo puede reiniciarse oprimiendo una combinación especial de teclas: Ctrl-Alt-MayúsDer-Supr (la única otra forma de desactivar el programa es apagar y reiniciar la computadora). Este programa funciona sólo si se inicia la computadora en MS-DOS. Microsoft Windows NT, 2000 y XP evitan que un programa TSR intercepte las teclas.

El byte de estado del teclado de MS-DOS Un poco de información que necesitamos saber antes de reemplazar es la ubicación del byte de estado del teclado que mantiene MS-DOS en la memoria inferior, que se muestra en la siguiente figura:



Nuestro programa inspeccionará esta bandera para ver si están oprimidas las teclas Ctrl, Alt, Supr y MayúsDer. La bandera de estado del programa se almacena en RAM, en la ubicación 0040:017h. La etiqueta del lado derecho del diagrama muestra lo que significada cada bit, cuando es igual a 1.

Un byte de estado adicional del teclado, ubicado en 0040:0018k, duplica las banderas anteriores, excepto que el bit 3 muestra cuando las teclas Ctrl-BloqNúm están activadas en un momento dado.

Instalación del programa El código residente en memoria debe instalarse en la memoria para que pueda funcionar. De ahí en adelante, toda la entrada proveniente del telcado se filtrará a través del programa. Si la rutina tiene errores, es probable que el teclado se bloquee y tengamos que reiniciar la máquina. En especial, los manejadores de interrupciones del teclado son difíciles de depurar, ya que estamos usando el teclado constantemente al depurar los programas. Los profesionales que escriben programas TSR con frecuencia invierten generalmente en depuradores asistidos por hardware, los cuales mantienen un búfer de rastreo en la memoria protegida. Uno de los errores más elusivos aparece sólo cuando un programa se ejecuta en tiempo real, no cuando estamos avanzando paso a paso a través de él. *Nota: debemos iniciarl la computadora en modo MS-DOS antes de instalar este programa.*

Listado del programa En el siguiente listado del programa, el código de instalación se encuentra al final, ya que no permanecerá residente en memoria. La porción residente, que empieza con la etiqueta **manejadpr_int9**, se deja en memoria y es la que apunta el vector INT 9h:

```
TITLE Programa para deshabilitar reinicios (No_Reinicio.asm)

; Este programa deshabilita el comando usual de reinicio del DOS

; (Ctrl-Alt-Supr), interceptando la interrupción de hardware

;  de teclado INT 9. Comprueba los bits de estado de mayúsculas

; en la bandera del teclado de MS-DOS y cambia cualquier Ctrl-Alt-Del

; por Alt-Supr. La computadora sólo puede reiniciarse si

; se escribe Ctrl-Alt + Mayús der + Supr. Debemos ensamblar el código,

; enlazarlo y convertirlo en un programa COM incluyendo el comando /T

; en la línea de comandos de Microsoft LINK.

; Debemos reiniciar en modo de MS-DOS puro antes de ejecutar este

; programa.

.model tiny

.386

.code

   mayus_der   EQU 01h         ; Tecla Mayús der: bit 0
```

```
   tecla_ctrl  EQU 04h         ; Tecla CTRL: bit 2
   tecla_alt   EQU 08h         ; Tecla ALT: bit 3
   tecla_supr  EQU 53h         ; código de exploración para la tecla SUPR
   puerto_tecl EQU 60h         ; puerto de entrada del teclado
   ORG 100h                    ; éste es un programa COM
inicio:
   jmp configuracion           ; salta a la instalación del TSR
; El código residente en memoria empieza aquí
manejador_int9 PROC FAR
   sti                         ; habilita interrupciones de hardware
   pushf                       ; guarda registros y banderas
   push es
   push ax
   push di
; Apunta ES:DI al byte de la bandera del teclado de DOS:
L1: mov ax, 40h                ; el segmento de datos de DOS está en 40h
   mov es, ax
   mov di, 17h                 ; ubicación de la bandera del teclado
   mov ah, es:[di]             ; copia la bandera del teclado en AH
; Prueba las teclas CTRL y ALT:
L2: test ah, tecla_ctrl        ; ¿Está oprimida CTRL?
   jz L5                       ; no: termina
   test ah, tecla_alt          ; ¿Está oprimida ALT?
   jz L5                       ; no: termina
; Prueba las teclas DEL y Mayús-Der:
L3: in al, puerto_tecl         ; lee el puerto del teclado
```

```
   cmp al, tecla_supr          ; ¿Se oprimió DEL?
   jne L5                      ; no: termina
   test ah, mayus_der          ; ¿se oprimió Mayús Der?
  jnz L5                       ; si: permite que el sistema se reinicie
L4: and ah, NOT tecla_ctrl     ; no: apaga el bit para CTRL
   mov es:[di], ah             ; almacena bandera_tecl
L5: pop di                     ; restaura registros y banderas
   pop ax
   pop es
   popf
   jmp cs:[interrup9_ant]      ; salta a la rutina de INT 9
interrup9_ant DWORD ?
manejador_int9 ENDP
fin_ISR label BYTE
; --------------------(fin del programa TSR) -----------------------
; Guarda una copia del vector INT 9 original y establece
; la dirección de nuestro programa como el nuevo vector. Termina
; este programa y deja el procedimiento manejador_int9 en la memoria.
configuracion:
   mov ax, 3509h                       ; obtiene el vector INT 9
   int 21h
   mov word ptr interrup9_ant, bx      ; guarda el vector INT 9
   mov word ptr interrup9_ant+2, es
   mov ax, 2509 ; establece el vector de interrupción, INT 9
   mov dx, offset manejador_int9
   int 21h
```

```
   mov ax, 31000h              ; termina y permanece residente
   mov dx, OFFSET fin_ISR      ; apunta al final del código residente
   shr dx, 4                   ; divide entre 16
   inc dx                      ; redondea hacia arriba, el siguiente
párrafo
   int 21h                     ; ejecuta la función de MS-DOS
END inicio
```

Primero vamos a ver a las instrucciones que instalan el programa. En la etiqueta llamada **configuracion**, llamamos a la función 35h de INT 21h para obtener el vector INT 9h actual, después se almacena en **interrup9_ant**. Esto se hace para que el programa pueda encadenar hacia delante el procedimiento del manejador de teclado existente. En la misma parte del programa, la función 25h de INT 21h establece el vector de interrupción 9h a la dirección de la porción residente de este programa. Al final del mismo, la llamada a la función 31h de INT 21h sale a MS-DOS, dejando el programa residente en memoria. La función guarda de manera automática todo, desde el inicio del PSP hasta el desplazamiento que se coloca en DX.

El programa residente El manejador de interrupciones residente en memoria empieza en la etiqueta llamada **manejador_int9**. Se ejecuta cada vez que se oprime una tecla. Rehabilitamos las interrupciones tan pronto como el manejador obtiene el control, debido a que el PIC 8259 ha deshabilitado las interrupciones de manera automática:

```
manejador_int9 PROC FAR
   sti                         ; habilita interrupciones de hardware
   pushf                       ; guarda registros y banderas
   (etc…)
```

Debemos tener en cuenta que a menudo una interrupción del teclado ocurre mientras otro programa se está ejecutando. Si modificamos los registros o las banderas de estado aquí, provocaríamos resultados impredecibles en un programa de aplicación.

Las siguientes instrucciones localizan el byte de la bandera del teclado que se almacena en la dirección 0040:0017 y lo copian en AH. El byte debe probarse para ver qué teclas se mantienen oprimidas en un momento dado:

```
L1: mov ax, 40h             ; el segmento de datos de DOS está en 40h
   mov es, ax
   mov di, 17h              ; ubicación de la bandera del teclado
   mov ah, es:[di]          ; copia la bandera del teclado en AH
```

Las siguientes instrucciones comprueba las teclas Ctrl y Alt. Si no se mantienen ambas oprimidas, nos salimos:

```
L2: test ah, tecla_ctrl        ; ¿Está oprimida CTRL?

   jz L5                       ; no: termina

   test ah, tecla_alt          ; ¿Está oprimida ALT?

   jz L5                       ; no: termina
```

Si las teclas Ctrl y Alt se mantienen oprimidas a la vez, alguien podría estar tratando de reiniciar la computadora. Para averiguar qué carácter se presionó, recibimos el carácter del puerto del teclado y lo comparamos con la tecla Supr:

```
L3: in al, puerto_tecl         ; lee el puerto del teclado

   cmp al, tecla_supr          ; ¿Se oprimió DEL?

   jne L5                      ; no: termina

   test ah, mayus_der          ; ¿Se oprimió Mayús Der?

   jnz L5                      ; si: permite que el sistema se reinicie
```

Si el usuario no ha oprimida la tecla Supr, simplemente salimos y dejamos que INT 9h procese la tecla presionada. Si la tecla Supr se mantiene oprimida, sabemos que se oprimió Ctrl-Alt-Del; sólo permitimos que el sistema se reinicie si el usuario también mantiene oprimida la tecla Mayús derecha. En caso contrario, el bit de la tecla Ctrl en el byte de la bandera del teclado se borra, con lo cual se deshabilita efectivamente el intento del usuario por reiniciar la computadora:

```
L4: and ah, NOT tecla_ctrl     ; no: apaga el bit para CTRL

   mov es:[di], ah             ; almacena la bandera_tecl
```

Por último, ejecutamos un salto lejano a la rutina existente de BIOS INT 9h, almacenada en la varible **interrup9_ant**. Esto permite procesar todas las teclas presionadas normales, lo cual es vital para la opración básica de la computadora:

```
jmp cs:[interrup9_ant]         ; salta a la rutina de INT 9
```

Control de hardware mediante el uso de puerto de E/S

Los sistemas IA-32 ofrecen dos tipos de entrada-salida de hardware: por *asignación de memoria* y por *asignación de puerto*. Cuando se utiliza la E/S por *asignación de memoria*, un programa puede escribir datos a una dirección de memoria específica, y los datos se transfieren al dispositivo de salida. De manera similar, se pueden leer los datos de un dispositivo de entrada si se copian desde una dirección de memoria predefinida. La pantalla

de video de texto es un ejemplo de un dispositivo de asignación de memoria. Cuando colocamos caracteres en el segmento de video, aparecen de inmediato en la pantalla.

La E/S por *asignación de puerto* requiere que las instrucciones IN y OUT lean y escriban datos en ubicaciones con numeración específicas, conocidas como *puertos*. Los puertos son conexiones, o compuertas, entre la CPU y otros dispositivos, como el teclado, los parlantes, el módem y la terjeta de sonido.

Puertos de entrada-salida

Cada puerto de entrada-salida tiene un número específico entre 0 y FFFFh. Por ejemplo, para controlar los parlantes se utiliza un puerto, haciendo que el cono de la misma se mueva con rapidez hacia dentro y hacia afuera. Podemos comunicarnos directamente con el adaptador asíncrono a través de un puerto serial, estableciendo los parámetros del puerto (velocidad en baudios, paridad, etcétera) y enviando datos a través del puerto.

El puerto del teclado es un buen ejemplo de un puerto de entrada-salida. Cuando se oprime una tecla, el chip controlador del teclado envía un código de exploración de 8 bits al puerto 60h. Cuando se presiona una tecla, se activa una interrupción de hardware, la cual pide a la CPU que llame a INT 9 en el BIOS de ROM. INT 9 recibe el código de exploración del puerto, busca el código ASCII de la tecla y almacena ambos valores en el búfer de entrada del teclado. De hecho, sería posible pasar por alto el sistema operativo por completo, y leer los caracteres directamente del puerto 60h.

Además los puertos que transfieren datos, la mayoría de los dispositivos de hardware tienen puertos que nos permiten monitorear el estado de un dispositivo y controlar su comportamiento.

Instrucciones IN y OUT la instrucción IN recibe un byte, palabra o doble palabra de un puerto. En constraste, la instrucción OUT envía un valor a un puerto. La sintaxis para ambas instrucciones es:

```
IN acumulador, puerto
```

```
OUT puerto, acumulador
```

Puerto puede ser una constante en el rango de 0 a FFh, o puede ser un valor en DX; entre 0 y FFFFh. *Acumulador* debe ser AL para transferencias de 8 bits, AX para transferencias de 16 bits y EAX para transferencias de 32 bits. A continuación se muestran algunos ejemplos:

```
in al, 3Ch                    ; recibe byte del puerto 3Ch

out 3Ch, al                   ; envía byte al puerto 3Ch

mov dx, numeroPuerto          ; DX puede contener un número de puerto

in ax, dx                     ; recibe palabra del puerto nombrado en DX

out dx, ax                    ; envía palabra al mismo puerto
```

```
in eax, dx                    ; doble palabra del puerto

out dx, eax                   ; envía doble palabra al mismo puerto
```

Programa de sonido de PC

Podemos escribir un programa que utilice las instrucciones IN y OUT para generar sonido a través del altavoz integrado a la PC. El puerto de control del altavoz (número 61h) enciende y apaga la bocina, manipulando el chip de *interfaz periférica programable* Intel 8255. Para encender el altavoz, se introduce el valor actual en el puerto 61h, se establecen los 2 bits inferiores y se envía el byte de vuelta a través del puerto. Para apagar el altavoz, se borran los bits 0 y 1, se imprime el estado de nuevo.

Nuestro programa de sonido no produciría sonidos en una computadora portatil si su altavoz está conectado directamente a la tarjeta de sonido, en vez de estar conectado al puerto de altavoz (61h).

El chip Temporizador Intel 8253 controla la frecuencia (tono) del sonido que se va a generar. Para usarlo, enviamos un valor entre 0 y 255 al puerto 42h. El programa Demo de altavoz muestra cómo generar sonido, reproduciendo una serie de notas ascendentes:

```
TITLE Programa Demo de altavoz (Altavoz.asm)

; Este programa reproduce una serie de notas ascendentes en

; el altavoz de la PC.

INLCUDE Irvine16.inc

altavoz      = 61h            ; dirección del puerto de altavoz

temporizador = 42h            ; dirección del puerto del altavoz

retraso1     = 500

retraso2     = 0D000h         ; retraso entre las notas

.code

main PROC

   in al, altavoz             ; obtiene el estado del altavoz

   push ax                    ; guarda el estado

   or al, 00000011b           ; establece los dos bits inferiores

   out altavoz, al            ; enciende el altavoz
```

```
   mov al, 60                  ; tono inicial
L2: out temporizador, al       ; puerto del temporizador: envía pulso al
altavoz
   ; Crea un ciclo de retraso entre los tonos:
   mov cx, retraso1
L3: push cx                    ; ciclo exterior
   mov cx, retraso2
L3a:                           ; ciclo interior
   loop L3a
   pop cx
   loop L3
   sub al, 1                   ; eleva el tono
   jnz L2                      ; reproduce otro tono
   pop eax                     ; obtiene el estado original
   and al, 11111100b           ; borra los 2 bits inferiores
   out altavoz, al             ; apaga el altavoz
   exit
main ENDP
 END main
```

Primero, el programa enciende el altavoz usando el puerto 61h, para lo cual establece los 2 bits inferiores en el byte de estado del altavoz:

```
or al, 00000011b               ; establece los 2 bits inferiores
out altavoz, al                ; enciende el altavoz
```

Después establece el tono, enviando 60 al chip temporizador:

```
   mov al, 60                  ; tono inicial
L2: out temporizador, al       ; puerto del temporizador: envía pulso al
altavoz
```

Un ciclo de retraso hace que el programa se detenga antes de cambiar el tono de nuevo. La cantidad de retraso varía entre distintas computadoras, debido a la diferencia en la velocidad de sus procesadores. Tal vez tengamos que ajustar los valores de **retraso1** y **retraso2**:

```
mov cx, retraso1

L3: push cx                    ; ciclo exterior

   mov cx, retraso2            ; ciclo interior

L3a: loop L3a

   pop cx

   loop L3
```

Después del retraso, el programa resta 1 al periodo (*1/frecuencia*), lo cual eleva el tono. La nueva frecuencia se envía al temporizador cuando se repite el ciclo. Este proceso continúa hasta que el contador de frecuencia en AL, es igual a 0. Por último, el programa saca el byte de estado original del puerto del altavoz y lo apaga borrando los 2 bits inferiores:

```
pop ax

and al, 11111100b              ; borra los 2 bits inferiores

out altavoz, al
```

17 Procesamiento de punto flotante y codificación de instrucciones

Representación binaria de punto flotante

Un número de punto flotante contiene tres componentes: un signo, una mantisa y un exponente. Por ejemplo, en el número -1.23154 x 10^5, el signo es negativo, la mantisa es 1.23154 y el exponente es 5.

Representación de punto flotante binaria IEEE

Los procesadores Intel utilizan tres formatos de almacenamiento binario de punto flotante, los cuales se especifican en el *Estándar* 754-1985 *para la aritmética binaria de punto flotante*, producido por la organización IEEE. La siguiente tabla describe sus características.

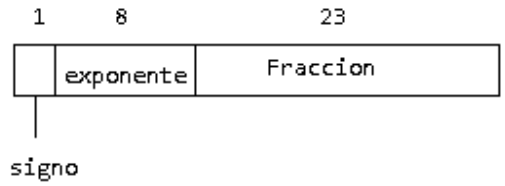
Formatos binarios de punto flotante de IEEE.

Precisión simple	32 bits: 1 bit para el signo, 8 bits para el exponente y 32 bits para la parte fraccional de la mantisa. Rango normalizado aproximado: 2^-126 a 2^127. También se le conoce como *real corto*.

Precisión doble	64 bits: 1 bit para el signo, 11 bits para el exponente y 52 bits para la parte fraccional de la mantisa. Rango normalizado aproximado: 2^-1022 a 2^1023. También se le conoce como *real largo*.
Precisión doble extendida	80 bits: 1 bit para para el signo, 16 bits para el exponente y 63 bits para la parte fraccional de la mantisa. Rango normalizado aproximado: 2^-16382 a 2^16383. También se le conoce como *real extendido*.

Como los tres formatos son muy similares, nos enfocaremos en el formato de precisión simple (figura a continuación). Los 32 its se ordenan con el bit más significativo (MSB) a la izquierda. El segmento se marca como *fracción* indica la parte fraccional de la mantisa. Como podríamos esperar, los bytes individuales se almacenan en memoria en orden little endian (LSB en la dirección inicial).

Formato de precisión simple.



El signo

Si el bit de signo es 1, el número es negativo: si el bit es 0, el número es positivo. Cero se considera positivo.

La mantisa

En el número de punto flotante representado por la expresión *m * b^e*, a **m** se le conoce como mantisa; **b** es la base y **e** es el exponente. La *mantisa* de un número de punto flotante consiste en los dígitos decimales a la izquierda y derecha del punto decimal. En el capítulo 1 presentamos el concepto de la notación posicional ponderada al explicar los sistemas numéricos binarios, decimal y hexadecimal. El mismo concepto puede extenderse para incluir la parte fraccional de un número de punto flotante. Por ejemplo, el valor decimal 123.154 se representa mediante la siguiente suma.

123.154 = (1 x 10^2) + (2x10^1) + (3x10^0) + (1x10^-1) + (5x10^-2) + (4x10^-3)

Todos los dígitos a la izquierda del punto decimal tienen exponentes positivos, y todos los dígitos a la derecha tienen exponentes negativos.

Los números binarios de punto flotante también utilizan notación posicional ponderada. El valor binario de punto flotante 11.1011 se expresa como:

11.1011 = (1x2^1) + (1x2^0) + (1x2^-1) + (0x^2^-2) + (1x2^-3) + (1x2^-4)

(Otra forma de expresar los valores a la derecha del punto binario es presentarlo como una suma de fracciones, cuyos denominadores sean potencias de 2. En nuestro ejemplo, la suma es 11/16 (o 0.6875):

.1011 = (1/2) + (0/4) + (1/8) + 1/16 = 11/16

El proceso de generar la fracción decimal es bastante evidente. El numerador decimal (11) representa el patron de bits binario 1011. Si e es el número de bits significativos a la derecha del punto binario, el denominador decimal es 2^e. En nuestro ejemplo e = 4 por lo que 2^e = 16. La siguiente tabla muestra ejemplos adicionales de cómo traducir notación binaria de punto flotante a fracciones de base 10. La última entrada en la tabla contiene la fracción más pequeña que puede almacenarse en una mantisa normalizada de 23 bits.

Ejemplos: Traducción de números binarios de punto flotante a fracciones.

Binario de punto flotante	Fracción de base 10
11.11	3 3/4
101.0011	5 3/16
1101.100101	13 37/64
0.00101	5/32
1.011	1 3/8
0.00000000000000000000001	1/8388608

La precisión de la mantisa

El contínuo completo de números reales no puede representarse en ningún formato de punto flotante que tenga un número finito de bits. Por ejemplo, supongamos que un formato de punto flotante simplificado tiene mantisa de 5 bits. No habría forma de representar los valores que se encuentra entre los números 1.1111 y 10.0000 binarios. Por ejemplo, el valor 1.1111 requiere una mantisa más precisa. Si extendemos esta idea al formato IEEE de doble precisión, podemos ver que una mantisa de 53 bits no puede representar un valor binario que requiera 54 bits o más.

El exponente

Los exponentes de precisión simple se almacena como enteros sin signo de 8 bits con una desviación de 127. El exponente actual del número debe sumarse a 127. Consideramos el valor binario 1.101x2^5. Después de que se le suma el exponente actual (5) a 127, el exponente de desviación (132) se almacena en la representación del número. La siguiente tabla muestra ejemplos de exponentes en decimal con signo, después en decimal con desviación y, por último, en binario sin signo.

Ejemplos de exponentes representados en binario.

Exponente (E)	Con desviación (E + 127)	Binario
+ 5	132	10000000
0	127	01111111
- 10	117	01110101
+ 127	254	11111110
- 126	1	00000001
- 1	126	01111110

El exponente con desviación siempre es positivo, entre 1 y 254. Como dijimos antes, el rango actual del exponente es de -126 a + 127. El rango se eligió de manera que el menor recíproco posile del exponente no pueda ocasionar un desbordamiento.

Números de punto binarios normalizados

La mayoría de los números binarios de punto flotante se almacenan en formato n*ormalizado*, para maximizar la precisión de la mantisa. Dado cualquier número binario de punto flotante, podemos normalizarlo si desplazamos el punto binario hasta que aparezca un solo "1" a la izquierda del mismo. El exponente expresa el número de posiciones que se desplaza el punto binario a la izquierda (exponente positivo) o a la derecha (exponente negativo). He aquí algunos ejemplos:

Sin normalizar	Normalizado
1110.1	1.1101 x 2^3
.000101	1.01 x 2^-4
1010001.	1.010001 x 2^6

Valores sin normalizar Podríamos decir *denormalizar* un número binario de punto flotante sería invertir la operación de normalización. Se desplaza el punto binario hasta que el exponente sea cero. Si el exponente es *n* positivo, se desplaza el punto binario *n* posiciones a la derecha; si el exponente es n negativo, se desplaza el punto binario *n* posiciones a la izquierda, rellenando con ceros si es necesario.

Creación de la representación IEEE

Codificaciones de números reales

Una vez que se normalizan y codifican los campos bit de signo, exponente y mantisa, es fácil generar un número real corto IEEE binario completo. Si utilizamos la figura que muestra el *Formato de precisión simple* (mostrada anteriormente) como referencia, podemos colocar el bit de signo primero, los bits del exponente a continuación, y la parte fraccional de la mantisa al último. Por ejemplo el número binario 1.101 x 2^0 se representa de la siguiente manera:

- Bit de signo: 0
- Exponente: 01111111
- Fracción: 10100000000000000000000

El exponente con desviación (01111111) es la representación binaria de 127 decimal. Todas las mantisas normalizadas tienen 1 a la izquierda del punto binario, por lo que no hay necesidad de codificar el bit en forma explícita. En la siguiente tabla se muestran ejemplos adicionales.

Ejemplos de codificaciones de bits de precisión simple.

Valor binario	Exponente con desviación	Signo, Exponente, Fracción
-1.11	127	1 01111111 11000000000000000000000
+1101.101	130	0 10000010 10110100000000000000000
-.00101	124	1 01111100 01000000000000000000000
+100111.0	132	0 10000100 00111000000000000000000
+.0000001101011	120	0 01111000 10101100000000000000000

La unidad Intel de punto flotante utiliza los números indefinidos como respuestas para algunas operaciones inválidas cn números de punt flotante.

La especificación IEEE incluye varias codificaciones de números reals y no numéricas.

- Cero positivo y negativo.
- Números finitos denormalizados.
- Números finitos normalizados.
- Infinito positivo y negativo.
- Valores no numéricos (NaN, conocido como *No es un número*).
- Números indefinidos.

Normalizados y denormalizados Los *números finitos normalizados* son todos los valores finitos distintos de cero que pueden codificarse en un número real normalizado, entre cero e infinito. Aunque podría parecer que todos los números de punto flotante finitos distinto de cero deben normalizarse, no es posible cuando sus valores están cerca de cero. Esto ocurre cuando

la FPU no puede desplazar el punto binario hacia una posición normalizada, dada la limitación impuesta por el rango del exponente. Supongamos que la FPU calcula un resultado de 1.0101111 x 2^-129, que tiene un exponente demasiado pequeño como para almacenarlo en un número de precisión simple. Se genera una condición de excepción de desbordamiento, y el número se denormaliza de forma gradual mediante el desplazamiento del punto binario hacia la izquierda 1 bit a la vez, hasta que el exponente llega a un rango válido:

1.01011110000000000001111 x 2^-129

0.10101111000000000000111 x 2^-128

0.01010111100000000000011 x 2^-127

0.00101011110000000000001 x 2^-126

En este ejemplo se produjo una cierta pérdida de precisión en el significando, como resultado del desplazamiento del punto decimal.

Infinito positivo y negativo El infinito positivo (+ ∞) representa al máximo numero real positivo, el infinito negativo (- ∞) representa el máximo numero real negativo. Podemos comparar a los infinitos con otros valores: - ∞ es menor que + ∞, - ∞ es menor que cualquier número finito, y + ∞ es mayor que cualquier número finito. Cualquiera de los infinitos puede representar una condición de desbordamiento de punto flotantes. El resultado de un cálculo no puede normalizarse, ya que su exponente sería demasiado grando como para poder representarse mediante el número dispoinble de bits del exponente:

NaNs Los *Nans* son patrones de bits que no representan ningún número real válido. La arquitectura IA-32 incluye dos tipos de NaNs: Un *NaN silencioso* puede propagarse a través de la mayoría de las operaciones aritméticas sin provocar una excepción. Un *NaN de señalización* puede usarse para generar una excepción de operación inválida de punto flotante. Un compilador podría llenar un arreglo sin inicializar con valores NaN de señalización, para que cualquier intento de realizar cálculos en el arreglo genere una excepción. Un NaN silencioso se puede usar para guardar la información de diagnóstico que se crea durante las sesiones de depuarción.

Un programa es libre de codificar cualquier información que desee en un NaN. La unidad de punto flotante no trata de realizar operaciones con NaNs. El manual de la familiar Intel IA-32 detalla un conjunto de reglas que determinan los resultados de las instrucciones cuando se utilizan combinaciones de los tipos de Nans como operandos.

Codificaciones específicas Hay varias codificaciones para los valores que encontramos a menudo en las operaciones de punto flotante; estas codificaciones se muestran en la siguiente tabla.

Valor	Signo, Exponente, Mantisa
Cero positivo	0 00000000 00000000000000000000000

Cero negativo	1 00000000 00000000000000000000000
Infinito positivo	0 11111111 00000000000000000000000
Infinito negativo	1 11111111 00000000000000000000000
QNaN	x 11111111 1xxxxxxxxxxxxxxxxxxxxxxx
SNaN	x 11111111 0xxxxxxxxxxxxxxxxxxxxxxx°a

* El campo de la mantisa NaN empieza con 0, pero por lo menos uno de los bits restantes debe ser 1.

Las posiciones de bit marcadas con la letra x pueden ser 1 o 0 QNaN es un NaN silecioso, y sNaN es un NaN de señalización.

Conversión de fracciones decimales a reales binarios

Cuando una fracción decimal puede representarse como una suma de fracciones en la forma (1/2 + 1/4 + 1/8 + ...), es muy sencillo descubrir el real binario correspondiente. En la siguiente tabla, la mayoría de las fracciones en la columna izquierda no se encuentran en formato que se traduzca facilmente a binario. Sin embargo puede escribirse como en la segunda columna.

Ejemplos de fracciones decimales y reales binarios

Fracción decimal	Se factoriza como...	Real binario
1/2	1/2	.1
1/4	1/4	.01
3/4	1/2 + 1/4	.11
1/8	1/8	.001
7/8	1/2 + 1/4 + 1/8	.111
3/8	1/4 + 1/8	.011
1/16	1/16	.0001
3/16	1/8 + 1/16	.0011
5/16	1/4 + 1/16	.0101

Muchos números reales como 1/10 (0,1) o 1/100 (0,01), no pueden representarse mediante un número finito de dígitos binarios. Sólo podemos aproximarlos a una fracción así mediante una suma de fracciones, cuyos denominadores sean potencias de 2.

Método alternativo, Uso de la división de números largos binarios Cuando se involucran valores decimales pequeños, una manera sencilla de convertir las fracciones decimales en números binarios es convertir primero el númerador y el denominador a binario y después realizar la división larga. Por ejemplo, el 0,5 decimal se representa como la fracción 5/10. El 5 decimal es el 0101 binario y el 10 decimal es 1010 binario. Si realizamos una división larga binaria, encontraremos que el cociente es el 0.1 binario:

```
        .1
1010 ⌈ 0101.0
     ⌊ -101 0
       -----
          0
```

Cuando el 1010 binario se resta del dividendo el residuo es cero, y la división se detiene. Por lo tanto, la fracción 5/10 decimal es igual al número 0.1 binario. A este método lo llamaremos *método de división laga binaria*.

Representación de 0.2 en binario Vamos a convertir el 0.2 (2/10) decimal a binario, mediante el método de división larga binaria. Primero dividimos el 10 binario entre 1010 binario (10 decimal):

```
          .00110011 (etc.)
1010 ⌈10.00000000
     ⌊ 1 010
       ------
         1100
         1010
         ----
          10000
           1010
           ----
            1100
            1010
            ----
              etc.
```

El primer cociente es lo bastante grande para usar 10000. Después de dividir 1010 entre 10000, el residuo es 110. Si le adjuntamos otro cero, el nuevo dividendo es 1100. Después de dividir 1010 entre 1100, el residuo es 10. Después de adjuntar tres ceros, el nuevo dividendo es 10000. Éste es el mismo dividendo con el que empezamos. De aquí en adelante, la secuencia de los bits en el cociente se repite (0011...), por lo que sabemos que no encontraremos un cociente exacto y que 02 no puede representarse mediante un número finito de bits. La mantisa codificada de precisión simple es 00110011001100110011001.

Conversión de valores de precisión simple a decimales

A continuación se muestran los pasos sugeridos para convertir un valor IEEE de presición simple (SP) a decimal:

1. Si el MSB es 1, el número es negativo; en caso contrario es positivo.
2. Los siguientes 8 bits representan el exponente. Hay que restarle el 01111111 binario (decimal), con lo que se poduce el exponente sin desviación. Luego se convierte en exponente sin desviación a decimal.

3. Los siguientes 23 bits representan la mantisa. Se anota un “1.”, seguido de los bits de la mantisa. Los ceros a la izquierda pueden ignorarse. Hay que crear un número binario de punto flotante, usando la mantisa, el signo que se determinó en el paso 1 y el exponente que se calculó en el paso 2.
4. Denormalizar el número binario producido en el paso 3. Hay que desplazar el punto binario un número de lugares igual al valor del exponente. Se desplaza a la derecha si el exponente es positivo, o a la izquierda si es negativo.
5. De izquierda a derecha, hay que usar la notación posicional ponderada para formar la suma decimal de las potencias de 2, representadas por el número binario de punto flotante.

Ejemplo: convertir el número IEEE (0 100000010 01011000000000000000000) a decimal

1. El número es positivo.
2. El exponente sin desviación es 00000011 binario, o 3 decimal.
3. Si combinamos signo, exponente y mantisa, el número binario es + 1.01011 x 2^ 3
4. El número binario denormalizado es + 1010.11-
5. El valor decimal es + 10 3/4 o + 10.75

Unidad de punto flotante

El procesador Intel 8086 se diseñó para manejar sólo la aritmética de enteros. Esto resultó ser un problema para el software de gráficos y de uso intensivo de cálculos, en donde se utilizaban los cálculos con números de punto flotante. Era posible emular la aritmética de punto flotante sólo mediante el software, pero el castigo en el rendimiento era severo. Los programas como *AutoCad* (de Autodesk) demandaban un método más poderoso para realizar operaciones aritméticas de punto flotante. Intel vendió un chip coprocesador de punto flotante por separado, llamado 8087, y lo actualizó junto con cada generación de procesadores. Con la llegada del Intel486, el hadware de punto flotante se integró a la CPU principal y se llamo *Unidad de punto flotante* (FPU).

Pila de registros FPU

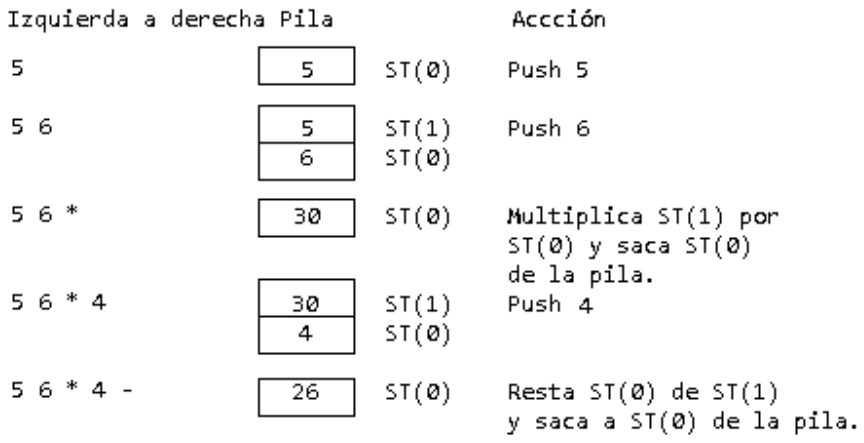
La FPU no utiliza los registros de propósito general (EAX, EBX etcétera). En vez de ello, tiene su propio conjunto de registros llamado *pila de registros*. Carga los valores de memoria y los coloca en la pila de registros, realiza cálculos y almacena los valores de la pila en la memoria. Las instrucciones de la FPU evalúan expresiones matemáticas en formato *postfijo*, en forma muy parecida a las calculadoras Hewlett-Packard. Por ejemplo, a la siguiente expresión se le conoce como *expresión infijo*: (5 * 6) + 4. El equivalente en postfijo es:

5 6 * 4 +

La expresión infijo **(A + B) * C** requiere paréntesis para ignorar las reglas de precedencia predeterminadas (la multiplicación antes de la suma). La expresión postfijo equivalente no requiere paréntesis:

A B * C +

Pila de expresiones Una pila almacena los valores intermedios durante la evaluación de las expresiones postfijo. La siguiente figura muestra los pasos requeridos para evaluar la expresión postfijo **5 6 * 4 -.**



Las entradas de la pila se etiquetan como ST(0) y ST(1), en donde ST(0) indica la posición a la que apuntaría generalmente el apuntador de la pila.

Los métodos de uso común para traducir expresiones infijo a expresiones postfijo están bien documentadas en los textos de introducción a las ciencias computacionales y en Internet, por lo que no lo explicaremos aquí. La siguiente tabla contiene algunos ejemplos de expresiones equivalentes.

Ejemplos de infijo a postfijo

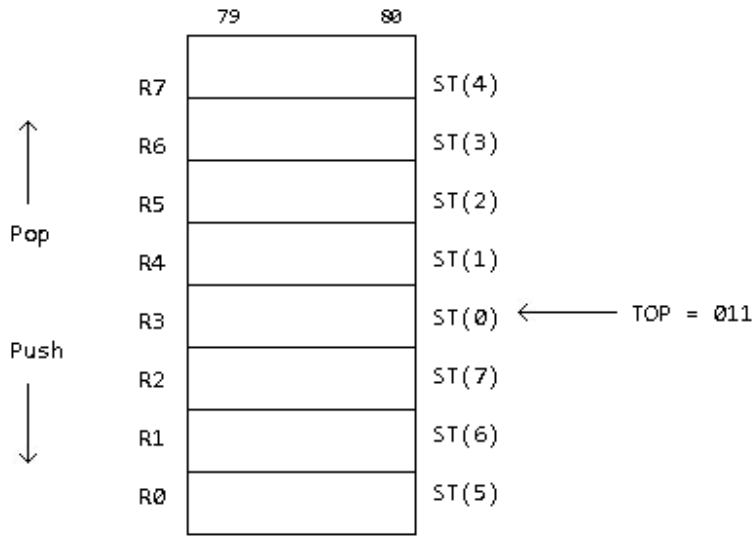
Infijo	Postfijo
A + B	A B +
(A – B) / D	A B – D /
(A + B) * (C + D)	A B + C D + *
((A + B) / C) * (E – F)	A B + C / E F - *

Registro de datos de la FPU

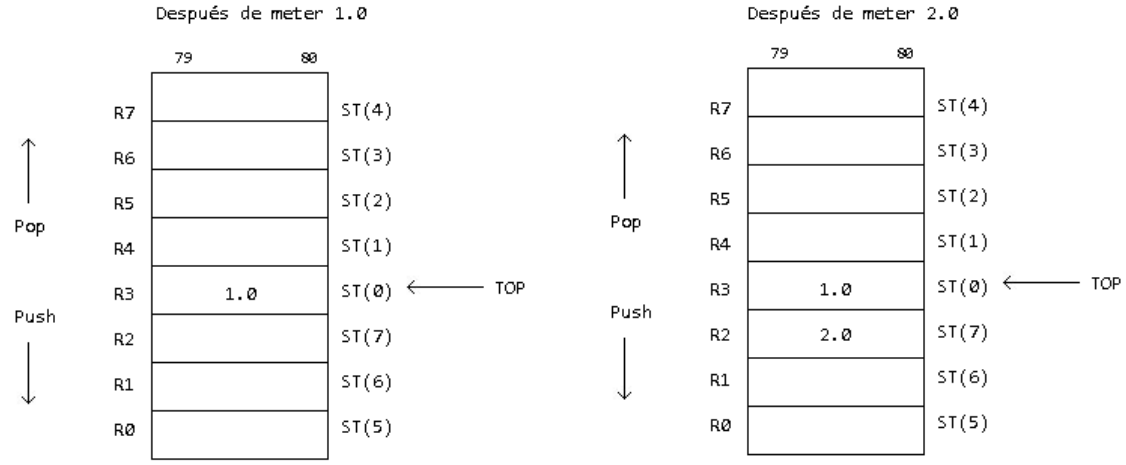
La FPU tiene ocho registros de datos de 80 bits que se pueden direccionar en forma individual, denominados R0 a R7. En conjunto se les conoce como *pila de registros*. Un campo de tres bits llamado TOP en la palabra de estado de la FPU identifica el número de registro que se encuentra en la parte superior de la pila en ese momento. Por ejemplo en la siguiente figura TOP es igual al número 011 binario, con lo cual indetifica a R3 como la parte superior de la

pila. Esta ubicación se le conoce también como ST(0) (o simplemente ST) al recibir instrucciones de punto flotante. El último registro es ST(7).

Pila de registros de datos de punto flotante



Como podríamos esperar, una operación *push* (también conocida como *cargar*) decrementa a TOP en 1 y copia un operando en el registro identificado como ST(0). Si TOP es igual a 0 antes de la operación push, TOP pasa al registro ST(7). Una operación *pop* (también conocida como *almacenar*) copia los datos que hay en ST(0) a un operando y después suma 1 a TOP. Si TOP es igual a 7 antes de la operación pop, pasa al registro R0. Si al cargar un valor en la pila se sobreescriben los datos existentes en la pila de registros, se genera *una excepción de punto flotante*. La siguiente figura muestra la misma pila después de haber metido (cargado) los valores 1.0 y 2.0.



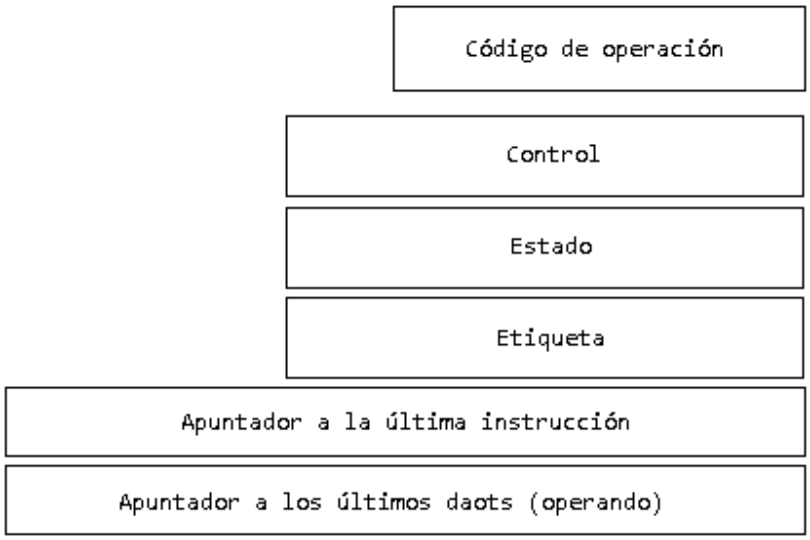
Aunque es interesante comprender cómo la FPU implementa la pila usando un conjunto limitado de registros, sólo tenemos que enfocarnos en la notación ST(*n*), en donde ST(0)

siempre es la parte superior de la pila. De aquí en adelante, nos referimos a los registros de la pila como ST(0), ST(1), etcétera. Los operandos de las instrucciones no pueden hacer referencia directa a los números de los registros.

Los valores de punto flotante en los registros utilizan el formato *real extendido* de 10 bytes del IEEE (también conocido como *real temporal*). Cuando la FPU almacena el resutlado de una operación aritmética en memoria, traduce el resultado a uno de los siguientes formatos: entero, largo, de precisión simple (real corto), de precisión doble (real largo), o de decimal codificado en binario empaquetado.

Registro de propósito especial

La FPU tiene seis registros de *propósito especial*:



- **Registro de código de operación:** almacena el código de operación de la última instrucción ejecutada que no sea de control.
- **Registro de control:** controla la precisión y el método de redondeo utilizado por la FPU al realizar cálculos. También puede ser usado para enmascarar (ocultar) las excepciones individuales de punto flotante.
- **Registro de estado**: contiene el apuntador de la parte superior de la pila, los códigos de condición y las advertencias sobre las expeciones.
- **Registro de etiqueta:** indica el contenido de cada registro en la pila de registros de datos de la FPU. Utiliza dos bit por registro para indicar si el registro contiene un número válido, cerp o un valor especial (NaN, infinito, denormalizado o formato no soportado), o está vacío.
- **Registro de apuntador a la última instrucción**: almacena un apuntador a la última instrucción ejecutada que no sea de control.
- **Registro apuntador a los últimos datos (operando):** almacena un apuntador a un operando de datos, si lo hay, que haya sido utilizado por la última instrucción ejecutada.

Los sistemas operativos utilizan los registro de propósito especial para preservar información de estado al cambiar de una tarea a otra.

Redondeo

La FPU trata de generar un resultado infinitamente preciso de un cálculo de un punto flotante. En muchos casos esto es imposible, ya que el operando de destino tal vez no pueda representar de manera precisa el resultado calculado. Por ejemplo, supongamos que cierto formato de almacenamiento sólo permite tres bits fraccionales. Nos permitiría almacenar valores como 1.011 o 1.101, pero no 1.0101. Supongamos que el resultado preciso de un cálculo produjo +1.0111 (1.4375 decimal). Podríamos redondear el número hasta el siguiente valor más alto al sumarle .0001, o redondeando hacia abajo, restándole .0001:

a. 1.0111 → 1.100
b. 1.0111 → 1.011

Si el resultado preciso fuera negativo, al sumarle -.0001, el resultado redondeado se movería cerca del infinito negativo. Al restarle -.0001 se movería más cerca de cero como del infinito positivo.

a. -1.0111 → -1.100
b. -1.0111 → -1.011

La FPU nos permite seleccionar uno de cuatro métodos de redondeo:

- *Redondeo al número más cercano*: el resultado redondeado es el más cercano al resultado infinitamente preciso. Si dos valores están igual de cerca, el resultado es un valor par (bit menos significativo = 0).
- *Redondeo hacia - ∞:* el resultado redondeado es menor o igual al resultado infinitamente preciso.
- *Redondeo hacia + ∞:* el resultado redondeado es mayor o igual que el resultado infinitamente preciso.
- *Redondeo hacia cero*: también conocido como truncamiento; el valor absoluto del resultado redondeado es menor o igual que el resultado infinitamente preciso.

Palabra de contol de la FPU La palabra de control de la FPU contiene dos bit llamados *campo RC*, que especifican el método de redondeo a utiliza. Los valores de los campos son:

- 00 binario: redondea el número par más cercano (predeterminado).
- 01 binario: redondea hacia infinito negativo.
- 10 binario: redondea hacia infinito positivo.
- 11 binario redondea hacia cero (trunca).

Redondear el número par más cercano es el método predeterminado, y se considera el más apropiado y preciso para la mayoría de los programas de aplicaciones. La siguiente tabla muestra cómo se aplicarían los cuatro métodos de redondea al número +1.0111 binario. De

manera similar la tabla a continuación muestra los posibles redondeos del número -1.0111 binario:

Ejemplo: redondeo de +1.0111.

Método	Resultado preciso	Redondeado
Redondea al número par más cercano	1.0111	1.100
Redondea hacia - ∞	1.0111	1.011
Redondea hacia + ∞	1.0111	1.100
Redondea hacia 0	1.0111	1.011

Ejemplo redondeo de -1.0111.

Método	Resultado preciso	Redondeado
Redondea al número par más cercano	-1.0111	-1.100
Redondea hacia - ∞	-1.0111	-1.100
Redondea hacia + ∞	-1.0111	-1.011
Redondea hacia 0	-1.0111	-1.011

Excepciones de punto flotante

En todo programa puede haber errores, y la FPU tiene que lidiar con los resultados. En consecuencia, reconoce y detecta seis tipos de condiciones de excepción: Operación inválida (#I), División entre cero (#Z), Operando denormalizado (#D), Desbordamiento numérico (#O), Subdesbordamiento numérico (#U), y Precisión inexacta (#P). Los primeros tres (#I, #Z, #D) se detecta antes de que ocurra cualquier operación aritmética. Los últimos tres (#O, #U, #P), se detectan después de que se realiza una operación.

Cada tipo de excepción tiene un bit de bandera y un bit de máscara correspondiente. Cuando se detecta una excepción de punto flotante, el procesador activa el bit de bandera correspondiente. Para cada excepción marcada por el procesador, hay dos cursos de acción:

- Si se **activó** el correspondiente bit de máscara, el procesador maneja la excepción de manera automática y deja que el programa continúe.
- Si se **borró** el correspondiente bit de máscara, el procesador invoca a un manejador de excepción de software.

Por lo general, las respuestas enmascaradas (automáticas) del procesador son aceptables para la mayoría de los programas. Pueden utilizarse manejadores de excepciones personalizados en casos en que la aplicación requiere respuestas específicas. Una sola instrucción puede activar varias excepciones, por lo que el procesador mantiene un registro continuo de todas las excepciones que ocurren desde la última vez que se borraron las excepciones. Al completarse una secuancia de cálculos, podemos comprobar si ocurrió alguna excepción.

Conjunto de instrucciones de punto flotante

El conjunto de instrucciones de la FPU es algo complejo, por lo que aquí trataremos de ver las generalidades acerca de sus capacidades, junto con ejemplos específicos que demuestren el código que por lo regular generan los compiladores. Además. veremos cómo podemos ejercer un control sobre la FPU, cambiando su modo de redondeo. El conjunto de instrucciones contiene las siguientes categorías básicas de instrucciones:

- Transferencia de datos.
- Aritmética básica.
- Comparación.
- Trascendental.
- Constantes de carga (sólo constantes predefinidas especializadas).
- Control de la FPU x87.
- Administración de estado de SIMD y de la FPU x87.

Los nombres de las instrucciones de punto flotante empiezan con la letra F, para distinguirlas de las instrucciones de la CPU. La segunda letra del nemónico de instrucción (por lo general, B o I) indica cómo se debe interpretar un operando de memoria: B indica un operando decimal codificado en binario (BCD) e I indica un operando entero binario. Si no se especifica ninguno, se asume que el operando de memoria está en formato de número real. Por ejemplo, FBLD opera con números BCD, FILD opera con enteros , y FLD opera con números reales.

Operandos Una instrucción de punto flotante puede tener cero, uno o dos operandos. Si hay dos operandos, uno debe ser un registro de punto flotante. No hay operandos inmediatos, pero pueden cargarse ciertas constantes predefinidas (como 0.0, π) en la pila. Los registros de propósito general como EAX, EBX, ECX y EDX no pueden ser operandos (la única excepción FNSTSW, que almacena la palabra de estado de la FPU en AX). No se permiten operaciones de memoria a memoria.

Los operandos enteros deben cargarse en la FPU desde la memoria (nunca desde los registros de la CPU); se convierten en forma automática al formato de punto flotante. De manera similar, al almacenar valores de punto flotante en los operandos de memoria de enteros, los valores se truncan o se redondean de manera automática a enteros.

Inicialización (FINIT)

La instrucción FINIT inicializa la unidad de punto flotante. Establece la palabra de control de la FPU a 037Fh, que enmascara (oculta) todas las excepciones de punto flotante, establece el

redondeo al número más cercano y la precisión de cálculo a 64 bits. Es recomendable llamar a FINIT al principio de los programas, para conocer el estado inicial del procesador.

Tipos de datos de punto flotante

Vamos a ver un breve repaso de los tipos de datos de punto flotante que soporta MASM (QWORD, TBYTE, REAL4, REAL8 y REAL10). que se presentan en la siguiente tabla. Debemos utilizar estos tipos cuando definamos operandos de memoria para las instrucciones de la FPU. Por ejemplo, al cargar una variable de punto flotante en la pila de la FPU). la variable se define como REAL4, REAL8 o REAL10:

```
.data

valGrande REAL10 1.21234234235234243E+864

.code

fld valGrande                    ; carga la variable en la pila
```

Tipos de datos intrínsecos.

Tipo	Uso
QWORD	Entero de 64 bits
TBYTE	Entero de 80 bits (10 bytes)
REAL4	Real corto IEEE de 32 bits (4 bytes)
REAL8	Real largo IEEE de 64 bits (8 bytes)
REAL10	Real extendido IEEE de 80 bits (10 bytes)

Cargar valor de punto flotante (FLD)

La instrucción FLD (cargar valor de punto flotante) copia un operando de punto flotante a la parte superior de la pila de la FPU (conocida como ST(0)). El operando puede ser un operando de memoria de 32 bits, 64 bits, 80 bits (REAL4, REAL8, REAL10), o cualquier otro registro de la FPU:

```
FLD m32pf

FLD m64pf

FLD m80pf

FLD ST(i)
```

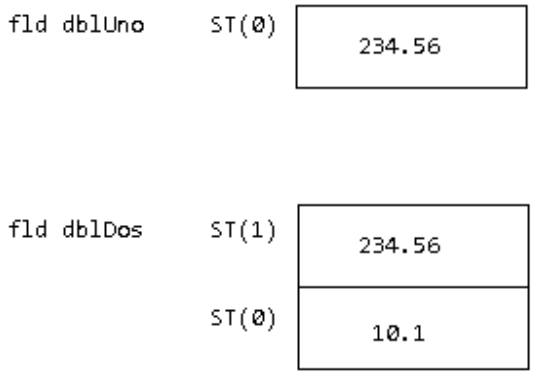
Tipos de operandos de memoria FLD soporta los mismos operandos de memoria que MOV, He aquí algunos ejemplos:

```
.data

arreglo REAL8 10 DUP(?)

.code

fld arreglo                              ; directo

fld [arreglo+16]                         ; directo-desplazamiento

fld REAL8 PTR[esi]                       ; indirecto

fld arreglo[esi]                         ; indexado

fld arreglo[esi*8]                       ; indexado, escalado

fld arreglo[esi*TYPE arreglo]            ; indexado, escalado

fld REAL8 PTR[ebx+esi]                   ; base-indice

fld arreglo[ebx+esi]                     ; base-indice-desplazamiento

fld arreglo[ebx+esi*TYPE arreglo]        ; base-indice-desplazamiento,
escalado
```

Ejemplo El siguiente ejemplo carga dos operandos directos en la pila de la FPU:

```
.data

dblUno REAL8 234.56

dblDos REAL8 10.1

.code

fld dbluno                      ; ST(0) = dblUno

fld dblDos                      ; ST(0) = dblDos, ST(1) = dblUno
```

La siguiente figura muestra el contenido de la pila después de ejecutar cada instrucción:



Cuando se ejecuta la segunda instrucción FLD, TOP se decrementa, haciendo que el elemento de la pila que se había etiquetado antes como ST(0) se convierta en ST(1).

FILD La instrucción FILD (cargar entero) convierte un operando de origen entero con signo de 64 bits en un valor de punto flotante de presición doble y lo carga en ST(0). Se preserva el signo del operando de destino. Demostrataremos su uso dentro de la sección Aritmética en modo mixto. FILD soporta los mismo tipos de operandos de memoria que MOV (indirecto, indexado, base-indexado, etcétera).

Carga de constantes Las siguientes instrucciones cargan constantes especializadas en la pila. No tienen operandos:

- La instrucción FLD1 mete 1.0 en la pila de registros.
- La instrucción FLDL2T mete log de 10 en base 2 en la pila de registros.
- La instrucción FLDL2E mete log de e en base 2 en la pila de registros.
- La instrucción FLDPI mete π en la pila de registros.
- La instrucción FLDLG2 mete log de 2 en base 10 en la pila de registros.
- La instrucción FLDLN2 mete log de 2 en base e en la pila de registros.
- La instrucción FLDZ (*cargar cero*) mete 0.0 en la pila de la FPU.

Almacenar un valor de punto flotante (FST, FSTP)

La instrucción FST (almacenar valor de punto flotante) copia un operando de punto flotante de la parte superior de la pila de la FPU a la memoria. FST soporta los mismos tipos de operandos que FLD. El operando puede ser un operando de memoria de 32 bits, 64 u 80 bits (REAL4, REAL8, REAL10), o puede ser otro registro de la FPU:

FST *m32pf*

FST *m64pf*

FST *ST(i)*

FST no saca de la pila. Las siguientes instrucciones almacenan a ST(0) en la memoria. Vamos a suponer que ST(0) es igual que 10.1 y que ST(1) es igual a 234.56:

```
fst dblTres                    : 10.1

fst dblCuatro                  ; 10.1
```

Por intuición, podríamos haber esperado que dblCuatro fuera igual a 234.56. Pero la primera instrucción FST dejó a 10.1 en ST(0). Si nuestra intención es copiar ST(1) en dblCuatro, deberíamos usar la instrucción FSTP.

FSTP La instrucción FSTP (almacenar valor de punto flotante y sacar) copia el valor que hay en ST(0) a la memoria y saca a ST(0) de la pila. Vamos a suponer que ST(0) es igual a 10.1 y que ST(1) es igual que 234.56, antes de ejecutar las siguientes instrucciones:

```
fstp dblTres                   ; 10.1

fstp dbCuatro                  ; 234.56
```

Después de la ejecución, los dos valores se elimanro lógicamente de la lista. Físicamente el apuntador TOP se incrementa cada vez que se ejecuta FSTP, cambiando la ubicación de ST(0).

La instrucción FIST (almacenar entero) convierte el valor que hay en ST(0) a un entero con signo y almacena el resultado en el operando de destino. Los valores pueden almacenarse como palabras o dobles palabras. Demostrataremos su uso dentro de la sección Aritmética en modo mixto. FIST sopoarta los mismo tipos de operando de memoria que FST.

Instrucciones aritméticas

En la siguiente tabla se presentan las operaciones aritmétcas básicas. Todas las instrucciones aritméticas soportadan los mismos tipos de operandos que FLD (cargar) y FST (almacenar), por lo que los operandos pueden ser indirectos, indexados, de base-índice, etcétera.

Instrucciones básicas de aritmétca de punto flotante.

FCHS	Cambiar signo
FADD	Sumar el origen al destino
FSUB	Restar el origen del destino
FSUBR	Restar el destino del origen
FMUL	Multiplicar el origen por el destino
FDIV	Dividir el destino entre el origen
FDIVR	Dividir el origen entre el destino

FCHS y FABS

La instrucción FCHS (cambiar signo) invierte el signo del valor de punto flotante en ST(0). La instrucción FABS (valor absoluto) borra el signo del número en ST(0) para crear su valor absoluto. Ninguna instrucción tiene operandos:

```
FCHS
```

```
FABS
```

FADD, FADDP, FIADD

La instrucción FADD (sumar) tiene los siguientes formatos, en donde *m32pf* es un operando de memoria REAL4, *m64pf* es un operando REAL8, e *i* es un número de registro:

```
FADD
```

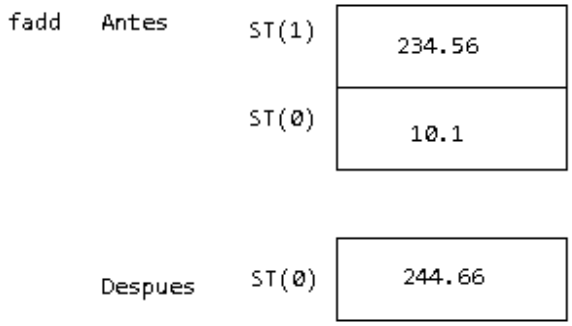
```
FADD m32pf
```

```
FADD m64pf
```

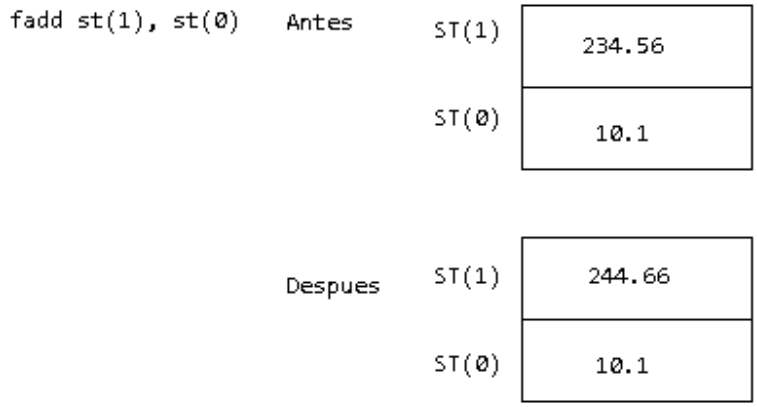
```
FADD ST(0), ST(i)
```

```
FADD ST(i), ST(0)
```

Sin operandos Si no se utilizan operandos con FADD, ST(0) se suma a ST(1). El resultado se almacena temporalmente en ST(1). Después, ST(0) se saca de la pila y el resultado queda en la parte superior de la misma. La siguiente figura demuestra a FADD, asumiendo que la pila ya contiene dos valores:



Operandos de registro Empezando con el mismo contenido de la pila, la siguiente ilustración demuestra cómo se suma ST(0) a ST(1):



Operando de memoria Al utilizarse con un operando de memoria, la instrucción FADD suma el operando a ST(0). He aquí algunos ejemplos:

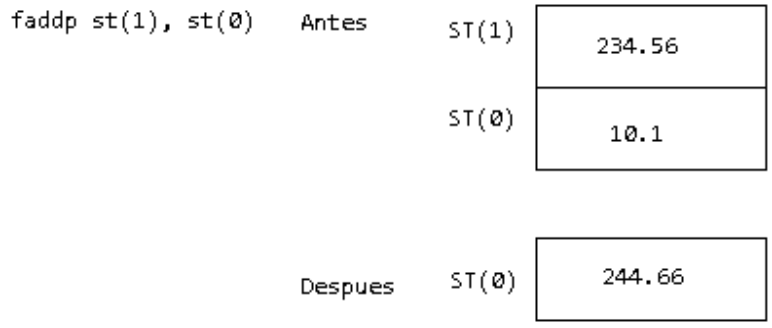
fadd miSimple ; ST(0) + = miSimple

fadd REAL PTR[esi] ; ST(0) + = [esi]

FADDP La instrucción FADDP (sumar con pop) saca a ST(0) de la pila, después de realizar la operación de suma. MASM soporta el siguiente formato:

`FADDP ST(i), ST(0)`

La siguiente figura muestra cómo funciona FADDP:



FIADD La instrucción FIADD (sumar entero) convierte el operando de origen al formato de punto flotante y precisión doble extendida antes de sumarlo a ST(0). Tiene la siguiente sintaxis:

FIADD m16ent

FIADD m32ent

Ejemplo

.data

```
miEntero DWORD 1

.code

fiadd miEntero                    ; ST (0) + = miEntero
```

FSUB, FSUBP, FISUB

La instrucción FSUB resta un operando de origen a un operando de destino y almacena la diferencia en el operando de destino. El destino siempre es un registro de la FPU y el origen puede ser un registro de la FPU o memoria. Acepta los mismos operandos que FADD:

```
FSUB

FSUB m32pf

FSUB m64pf

FSUB ST(0), ST(i)

FSU ST(i), ST(0)
```

La operación de FSUB es similar a la de FADD, solo que resta en vez de sumar. Por ejemplo, la forma sin operando de FSUB resta ST(0) de ST(1). El resultado se almacena temporalmente en ST(1). Después, ST(0) se saca de la pika, dejando el resultado en la parte superior de la misma. FSUB con un operando de memoria resta el operando de ST(0) y no lo saca de la pila.

Ejemplos:

```
fsub miSimple                     ; ST(0) - = miSimple

fsub arreglo[edi*8]               ; ST(0) - = arreglo[edi*8]
```

FSUBP La instrucción FSUBP (restar con pop) saca a ST(0) de la pila, después de realizar la resta. MASM soporta el siguiente formato:

```
FSUBP ST(i), ST(0)
```

FISUB La instrucción FISUB (restar entero) convierte el operando de origen al formato de punto flotante y precisión doble extendida, antes de restar el operando de ST(0):

```
FISUB m16ent

FISUB m32ent
```

FMUL, FMULP, FIMUL

La instrucción FMUL multiplica un operando de origen por un operando de destino, almacenando el producto en el operando de destino. El destino siempre es un registro de la FPU, y el origen puede ser un registro o un operando de memoria. Utiliza la misma sintaxis que FADD y FSUB:

```
FMUL

FMUL m32pf

FMUL m64pf

FMUL ST(0), ST(i)

FMUL ST(i), ST(0)
```

La operación de FMUL es similar a la de FADD, sólo que multiplica en vez de sumar. Por ejemplo, la forma sin operandos de FMUL multiplica a ST(0) por ST(1). El producto se almacena temporalmente en ST(1). Después, ST(0) se saca de la pila y el producto queda en la parte superior de la misma. De manera similar, FMUL con un operando de memoria multiplica a ST(0) por el operando de memoria:

```
fmul miSimple                    ; ST(0) * = miSimple
```

FMULP La instrucción FMULP (multiplicar con pop) saca a ST(0) de la pila, después de realizar la multiplicación. MASM soporta el siguiente formato:

```
FMULP ST(i), ST(0)
```

FIMUL es idéntica a FIADD, sólo que multiplica en vez de sumar:

```
FIMUL m16ent

FIMUL m32ent
```

FDIV, FDIVP, FIDIV

La instrucción FDIV divide un operando de destino entre un operando de origen, almacenando el dividendo en el operando de destino. El destino siempre es un registro y el operando de origen puede ser un registro o memoria. Tiene la isma sintaxis que FADD y FSUB:

```
FDIV

FDIV m32pf

FDIV m64pf

FDIV ST(0), ST(i)

FDIV ST(i), ST(0)
```

La operación de FDIV es similar a la de FADD, sólo que divide en vez de sumar. Por ejemplo, la forma de FDIV sin operandos divide a ST(1) entre ST(0). ST(0) se saca de la pila y el dividendo queda en la parte superior de la misma. FDIV con un operando de memoria divide a ST(0) entre el operando de memoria. El siguiente código divide a **dblUno** entre **dblDos** y almacena el cociente en **dblCoc**:

```
.data

dblUno REAL8 1234.56

dblDos REAL8 10.0

dblCoc REAL8 ?

.code

fld dblUno                    : lo carga en ST(0)

fdiv dblDos                   ; divide a ST(0) entre dblDos

fstp dblCoc                   ; guarda ST(0) en dblCoc
```

Si el operando de origen es cero, se genera una excepción de división entre cero. Se aplica un número de casos especiales cuando se divide con operandos iguales a infinito positivo o negativo, cero y *NaN*. Para más detalle debemos consultar el manual de Conjunto de instrucciones Intel IA-32.

FIDIV La instrucción FIDIV convierte un operando de origen entero al formato de punto flotante y precisión doble extendida, antes de dividirlo entre ST(0). Sintaxis:

```
FIDIV m16ent

FIDIV m32ent
```

Comparación de valores de punto flotante

Los valores de punto flotante no pueden compararse mediante la instrucción CMP, ya que ésta utiliza la resta de enteros para realizar comparaciones. En vez de ello, debe usarse la instrucción FCOM.Después de ejecutar FCOM, hay que llevar a cabo ciertos pasos especiales para poder utilizar las instrucciones de salto condicional (JA, JB, JE, etcétera.) en las instrucciones IF lógicas.

FCOM, FCOMP, FCOMPP La instrucción FCOMP (comparar valores de punto flotante) compara a ST(0) con su operando de origen. El origen puede ser un operando de memoria o un registro de la FPU. Sintaxis:

Instrucción	Descripción
FCOM	Compara ST(0) con ST(1)
FCOM m32pf	Compara ST(0) con m32pf
FCOM m64pf	Compara ST(0) con m64pf
FCOM ST(i)	Compara ST(0) con ST(i)

Las instrucciones FCOMP lleva a cabo las mismas operaciones y termina sacando a ST(0) de la pila. La instrucción FCOMPP es igual que FCOMP, sólo que saca de la pila una vez más.

Códigos de condición La FPU tiene tres banderas de código de condición, C3, C2 y C0, que indican los resulados de comparar valores de punto flotante:

Condición	C3 (bandera Cero)	C2 (bandera Paridad)	C0 (Bandera Acarreo)	Salto condicional a utilizar
ST(0)>SRC	0	0	0	JA, JNBE
ST(0)<SRC	0	0	1	JB, JNAE
ST(0) = SRC	1	0	0	JE, JZ
Sin orden	1	1	1	(*Ninguno*)

* Si se genera una excepción de operando aritmético inválido (debido a operandos inválidos) y la excepción está enmascarada, C3, C2 C0 se establecen de acuerdo con la fila marcada *Sin orden*.

Los encabezados de las columnas muestran banderas de estado equivalentes de la CPU, ya que C3, C2 y C0 tienen una función similar a las banderas Cero, Paridad y Acarreo, respectivamente.

El principal reto después de comparar dos valores y establecer los códigos de condición de la FPU es buscar la forma de bifurcar hacia una etiqueta basada en las condiciones. Se requieren dos pasos:

- Usar la instrucción FNSTSW para mover la palabra de estado de la FPU hacia AX.
- Usar la instrucción SAHF para copiar AH al registro FLAGS.

Una vez que los códigos de condición están en EFLAGS, podemos usar saltos condicionales basados en las banderas Cero, Paridad y Acarreo. La tabla anterior mostró el salto condicional apropiado para cada combinación de banderas. Podemos inferir los saltos adicionales: La instrucción JAE ocasiona una transferencia de control si CF = 0. JBE ocasiona una transferencia de control si CF = 1 o ZF = 1. JNE transfiere si ZF = 0.

Ejemplo Asumimos el siguiente código de C++:

```
double X = 1.2;

double Y = 3.0;

int N = 0;

if ( X < Y)
```

```
   N = 1;
```

A continuación se muestra el código equivalente en lenguaje ensamblador:

```
.data
X REAL8 1.2
Y REAL8 3.0
N DWORD 0
.code
; if ( X < Y)
;  N = 1
   fld X                        ; ST(0) = X
   fcomp Y                      ; compara ST(0) con Y
   fnstsw ax                    ; mueve la palabra de estado hacia AX
   sahf                         ; copia AH a EFLAGS
   jnb L1                       ; ¿X no es < Y? salta
   mov N,1                      ; N = 1
L1:
```

Mejoras de P6 Algo para destacar del ejemplo anterior es que las comparaciones de punto flotante incurren en más sobrecarga en tiempo de ejecución que las comparaciones de enteros. Con esto en mente, la familia P6 de Intel presentó la instrucción FCOMI. Esta instrucción comapra los valores de punto flotante y activa las banderas Cero, Paridad y Acarreo de manera directa. (La familia P6 empezó con los procesadores Pentium Pro y Pentiun II). FCOMI tiene la siguiente sintaxis:

```
FCOMI ST(0), ST(i)
```

Vamos a rescribir nuestro ejemplo de código anterior (Comparar X con Y) usando FCOM1:

```
.code
; if (X < Y)
;   N = 1
   fld Y                        ; ST(0) = Y
```

```
   fld X                        ; ST(0) = X, ST(1) = Y
fcomi ST(0), ST(I)              ; compara a ST(0) con ST(1)
jnb L1                          ; ¿ST(0) no es < ST(1)? Salta
mov N,1                         ; N = 1
```

La instrucción FCOMI tomó el lugar de tres instrucciónes en la versión anterior, pero requirió una instrucción FLD más. La instrucción FCOMI no acepta operandos de memoria.

Comparación de igualdad

*sqrt = raiz cuadrada

Casi todos los libros de texto de programación para principiantes advierten al lector que no deben comparar la igualdad entre valores de punto flotante debido a los errores de redondeo que ocurren durante los cálculos. Podemos demostrar este problema calculando la siguiente expresión: (sqrt(2.0)*sqrt(2.0)) – 2.0. En términos matemátivos, debería ser igual a cero pero los resultados son bastantes distintos (aproximadamente 4.408921E-016). Utilizaremos los siguientes datos y mostraremos la pila de la FPU después de cada paso en la siguiente tabla:

Cálculo de (sqrt(2.0) * sqrt(2.0)) – 2.0

Instrucción	Pila de la FPU
fld val1	ST(0): +2.0000000E+000
fsqrt	ST(0): +1.4142135E+000
fmul ST(0), ST(0)	ST(0): +2.0000000E+000
fsub val1	ST(0): +4.4408921E-016

La forma correcta de comparar los valores de punto flotante *X* e *Y* es tomar el valor absoluto de su diferencia, |x - y|. y compararlo con un valor pequeño definido por el usuario, llamado *épsilon*. A continuación se muestra el código en lenguaje ensamblador para hacer esto, usando épsilon como máxima diferencia que puede tener para seguir considerándose iguales:

```
.data
epsilon REAL8 1.0E-12
val2 REAL8 0.0                  ; valor a comparar
val3 REAL8 1.001E-13            ; se considera igual a val2
.code
```

```
; if (val2 == val3), mostrar “Los valores son iguales”.
   fld epsilon
   fld val2
   fsub val3
   fabs
   fcomi ST(0), ST(1)
   ja saltar
   mWrite <”Los valores son iguales”, 0dh, 0ah>
saltar:
```

La siguiente tabla rastrea el progreso del programa, mostrando la pila después de la ejecución de cada una de las primeras cuatro instrucciones.

Cálculo de un producto punto (6.0 * 2.0) + (4.5 * 3.2)

Instrucción	Pila de la FPU
`fld epsilon`	`ST(0): +1.0000000E-012`
`fld val2`	`ST(0): +0.0000000E+000` `ST(1): +1.0000000E-012`
`fsub val3`	`ST(0): -1.0010000E-013` `ST(1): +1.0000000E-012`
`fabs`	`ST(0): +1.0010000E-013` `ST(1): +1.0000000E-012`
`fcomi ST(0), ST(1)`	`ST(0) < ST(1), por lo que CF = 1, ZF = 0`

Si redefinieramos a val3 como mayor que épsilon, no sería igual a val2:

```
val3 REAL8 1.001E-12          ; no es igual
```

Lectura y escritura de valores de punto flotante

En la biblioteca de vínculos del libro se incluyen dos procedimientos para operaciones de entrada-salida de punto flotante, que creo Willian Barret de la Universidad Estatal de San Jose:

- **ReadFloat**: lee un valor de punto flotante del teclado y lo mete en la pila de punto flotante.
- **WriteFloat**: escribe el valor de punto flotante que hay en ST(0) en la ventana de consola, en formato exponencial.

ReadFloat acepta una amplia variedad de formatos de punto flotante. He aquí algunos ejemplos:

35

+35.

-3.5

.35

3.5E5

-3.5E+5

ShowFPUStack Otro útil procedimiento, que escribió James Brink de la Universidad Pacific Leutheran, muestra la pila de la FPU. Se llama sin parámetros:

```
call ShowFPUStack
```

Sincronización de excepciones

La CPU (enteros) y la FPU son unidades separadas, por lo que las instrucciones de punto flotante se pueden ejecutar al mismo tiempo que las instrucciones con enteros y las del sistema. Esta capacidad, llamada *concurrencia*, puede ser un problema potencial si se producen excepciones de punto flotante no enmascaradas. Por otro lado, las excepciones enmascaradas no son un problema, ya que la FPU siempre completa la operación actual y almacena el resultado.

Cuando ocurre una excepción no enmascarada, la instrucción de punto flotante actual se interrumpe y la FPU indica el evento de excepción. Cuando está a punto de ejecutarse la siguiente instrucción de punto flotante o la instrucción FWAIT (WAIT), la FPU verifica las excepciones pendientes. Si encuentra alguna, invoca al manejador de excepciones de punto flotante (una subrutina).

¿Qué pasa si la instrucción de punto flotante que produce la excepción va seguida de una instrucción con enteros o del sistema? Por desgracia, dichas instrucciones no verifican las excepciones pendientes; se ejecutan en forma inmediata. Si se supone que la primera instrucción va a almacenar sus resultados en un operando de memoria y la segunda instrucción modifica a ese mismo operando de memoria, el manejador de excepciones no se puede ejecutar en forma apropiada. He aquí un ejemplo:

```
.data

valEnt DWORD 25

.code

fild valEnt                    ; carga entero en ST(0)

inc valEnt                     ; incrementa el entero
```

Las instrucciones WAIT y FWAIT se crearon para forzar al procesador a verificar las excepciones de punto flotante pendientes, no enmascaradas, antes de continuar con la siguiente instrucción. Cualquiera de las dos resuelve nuestro problema potencial de sincronización, evitando que la instrucción INC se ejecute hasta que el manejador de excepciones tenga oportunidad de terminar:

```
fild valEnt                    : carga entero en ST(0)

fwait                          ; espera las excepciones pendientes

inc valEnt                     ; incrementa el entero
```

Ejemplos de código

En esta sección veremos ejemplos cortos que demuestran las instrucciones aritméticas de punto flotante. Una excelente forma de aprender es codificar las expresiones en C++, compilarlas e inspeccionar el código que produce el compilador:

Expresión

Vamos a codificar la expresión valD = -valA + (valB * valC). Una posible solución paso a paso es: cargar valA en la pila y negar su valor. Cargar valB en ST(0), mover valA hacia ST(1). Multiplicar ST(0) por valC, dejando el producto en ST(0). Sumar(1) y ST(0), almacenando la suma en valD:

```
.data

valA REAL8 1.5

valB REAL8 2.5

valC REAL8 3.0

valD REAL8 ?                   ; +6.0

.code

fld valA                       ; ST(0) = valA

fchs                           ; cambia el signo de ST(0)
```

```
fld valB                    ; carga valB en ST(0)

fmul valC                   ; ST(0) *= ST(1)

fadd                        ; ST(0) += ST(1)

fstp valD                   ; almacena ST(0) en valD
```

Suma de un arreglo

El siguiente código calcula y muestra la suma de un arreglo de números reales de doble precisión:

```
TAM_ARREGLO = 20

.data

arregloSimp REAL8 TAM_ARREGLO DUP(?)

.code

   mov esi, 0               ; indice del arreglo

   fldz                     ; mete 0.0 en la pila

   mov ecx, TAM_ARREGLO

L1: fld arregloSimp[esi]    ; carga mem en ST(0)

   fadd                     ; suma ST(0), ST(1), pop

   add esi, TYPE REAL8      ; se mueve el siguiente elemento

   loop L1

   call WriteFloat          ; muestra la suma en ST(0)
```

Suma de raices cuadradas

La intrucción FSQRT sustituye el numero en ST(0) con su raíz cuadrada. El siguiente código calcula la suma de dos raices cuadradas:

```
.data

valA REAL8 25.0

valB REAL8 36.0

.code
```

```
fld valA                        ; mete valA

fsqrt                           ; ST(0) = sqrt(valA)

fld valB                        ; mete valB

fsqrt                           ; ST(0) = sqrt(valB)

fadd                            ; suma ST(0), ST(1)
```

Producto punto de un arreglo

El siguiente código calcula la expresión (arreglo[0] * arreglo[1]) + (arreglo[2] * arreglo[3]). A este cálculo algunas veces se le conoce como *producto punto*. La siguiente tabla muestra la pila de la FPU después de la ejecución de cada instrucción. He aquí los datos de entrada

```
.data

arreglo REAL4 6.0, 2.0, 4.5, 3.2
```

Cálculo de un producto punto (6.0 * 2.0) + (4.5 * 3.2).

Instrucción	Pila de la FPU
fld arreglo	ST(0): +6.0000000E+000
fmul [arreglo+4]	ST(0): +1.2000000E+001
fld [arreglo+8]	ST(0): +4.5000000E+000 ST(1): +1.2000000E+001
fmul [arreglo+12]	ST(0): +1.4400000E+001 ST(1): +1.2000000E+001
fadd	ST(0): +2.6400000E+001

Aritmética de modo mixto

Hasta este punto hemos realizado operaciones aritméticas que involucran solo números reales. A menudo, las aplicaciones realizan operaciones aritméticas en modo mixto, combinando enteros y reales. Las instrucciones aritméticas de enteros como ADD y MUL no pueden manejar reales, por lo que nuestra única opción es utilizar instrucciones de punto flotante. El conjunto de instrucciones Intel proporciona instrucciones que promueven los enteros a reales y cargan los valores en la pila de punto flotante.

Ejemplo El siguiente código en C++ suma un entero a un doble y almacena la suma en un doble. C++ promueve de manera automática el entero a real, antes de realizar la suma:

```
int N = 20;

double X = 3.5;

double Z = N + X;
```

He aquí el código equivalente en lenguaje ensamblador:

```
.data

N SDWORD 20

X REAL8 3.5

Z REAL8 ?

.code

fild N                          ; carga entero en ST(0)

fadd X                          ; suma mem a ST(0)

fstp Z                          ; almacena ST(0) en mem
```

Ejemplo El siguiente programa en C++ promueve N a un doble, evalúa una expresión real y almacena el resultado en una variable entera:

```
int N = 20;

double X = 3.5;

int Z = (int) (N + X);
```

El código generado por Visual C++ llama a una función de conversión (ftol) antes de guardar el resultado truncado en Z. Si codificamos la expresión en lenguaje ensamblador usando FIST, podemos evitar la llamada a la función, pero Z (demanera predeterminada) se redondea hasta 24:

```
fild N                          ; carga entero en ST(0)

fadd X                          ; suma mem a ST(0)

fist Z                          ; guarda ST(0) en entero de memoria
```

Cambio al modo de redondeo El campo RC del control de la FPU nos permite especificar el tipo de redondeo a realizar. Podemos usar FSTCW para guardar la palabra de control en una variable, modificar el campo RC (bit 10 y 11), y utilizar la instrucción FLDCW para cargar la variable de vuelta en la palabra de control:

```
fstcw palabCtrl                 ; almacena palabra de control
```

```
or palabCtrl, 110000000000b    ; establece RC = truncar

fldcw palabCtrl                ; carga palabra de control
```

Después realizamos cálculos que requieren truncamiento, para producir Z = 23:

```
fild N                         ; carga entero en ST(0)

fadd X                         ; suma mem a ST(0)

fist Z                         ; guarda ST(0) en entero de memoria
```

De manera opcional, restablecemos el modo a su valor predeterminado (*se redondea al número par más cercano*):

```
fstcw palabCtrl                ; almacena la palabra de control

and palabCtrl, 001111111111b   ; restablece el redondea al predeterminado

fldcw palabCtrl                ; carga palabra de control
```

Enmascaramiento y desenmascaramiento de excepciones

Las excepciones se enmascaran de manera predeterminada, de manera que cuando se genera una excepción de punto flotante, el procesador asigna un valor predeterminado al resultado y continúa realizando su trabajo silencioso. Por ejemplo, al dividir un número de punto flotante entre cero se produce infinito sin detener el programa:

```
.data

val1 DWORD 1

val2 REAL8 0.0

.code

fild val1                      ; carga entero en ST(0)

fdiv val2                      ; ST(0) = infinito positivo
```

Si desenmascaramos la excepción en la palabra de control de la FPU, el procesador tratara de ejecutar un manejador de excepción apropiado. El desenmascaramiento se logra borrando el bit apropiada en la palabra de control de la FPU:

Campos en la palabra de control de la FPU

Bit(s)	Descripción
0	Máscara de excepción por operación inválida

1	Máscara de excepción por operando denormalizado
2	Máscara de excepción por división entre cero
3	Máscara de excepción por desbordamiento
4	Máscara de excepción por subdesbordamiento
5	Máscara de excepción por precisión
8 – 9	Control de precisión
10 – 11	Control de redondeo
12	Control de infinito

Supongamos que deseamos desenmascarar la excepción de divisón entre Cero. He aquí los pasos requeridos:

1. Almacenar la palabra de control de la FPU en una variable de 16 bits.
2. Borrar el bit 2 (bandera de divisón entre cero).
3. Cargar la variable de vuelta en la palabra de control.

El siguiente código desenmascara las excepciones de punto flotante:

```
.data

palabCtrl WORD ?

.code

fstcw palabCtrl                ; obtiene la palabra de control

and palabCtrl, 111111111011b   ; desenmascara división entre cero

fldcw palabCtrl                ; la carga de vuelta en la FPU
```

Ahora, si ejecutamos código que divida entre cero, se genera una excepción desenmascarada:

```
fild val1

fdiv val2

fst val2                       ; división entre cero
```

Tan pronto como la instrucción FST empieza a ejecutarse, MS Windows muestra un cuadro de dialogo indicando que ha ocurrido una excepción.

Enmascaramiento de excepciones Para enmascarar una excepción, se aciva el bit apropiado en la palabra de control de la FPU. El siguiente código enmascara las excepciones de división entre cero:

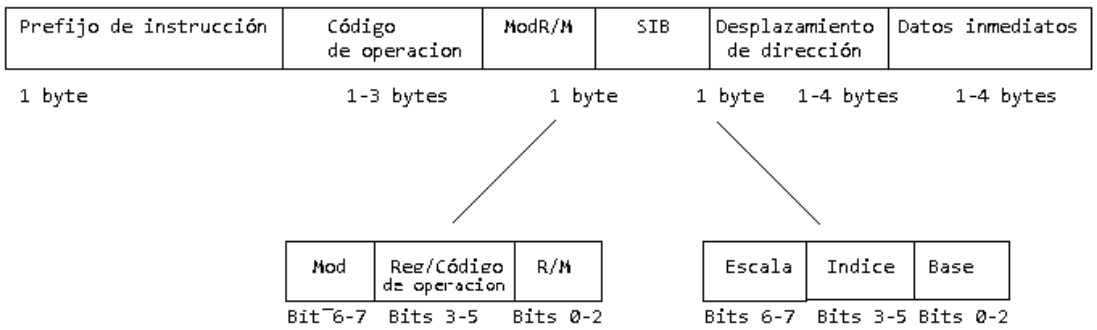
```
.data

palabCtrl WORD ?

.code

fstcw palabCtrl                ; obtiene la palabra de control

or palabCtrl, 100b             ; enmascara la división entre cero

fldcw palabCtrl                ; la carga de vuelta en la FPU
```

Codificación de instrucciones Intel

Para comprender el lenguaje ensamblador por completo, debemos invertir tiempo analizando la forma en que las instrucciones en ensamblador se traducen a lenguaje máquina. El tema es bastante complejo, debido a la extensa variedad de instrucciones y modos de direccionamientos disponibles en el conjunto de instrucciones Intel. Empezaremos con el procesador 8086/8088 como un ejemplo ilustrativo, ejecutándose en modo de direccionamiento real. Más adelante mostraremos algunos cambios que realizó Intel al introducir los procesadores de 32 bits.

Como mencionamos en el capítulo 2, el procesador Intel 8086 fue el primero en la línea de procesadores que utilizan el diseño de *Computadora con un conjunto complejo de instrucciones* (CISC.) El conjunto de instrucciones incluye una amplia variedad de operaciones de direccionamiento de memoria, de desplazamiento, aritmética, de movimiento de datos y lógicas. En comparación con las instrucciones RISC (*Computadora con un conjunto reducido de instrucciones*), las instrucciones de Intel son algo difíciles de codificar y decodificar. *Codificar* una instrucción significa convertir una instrucción en código máquina a lenguaje ensamblador.

Formato de instrucciones IA-32



El formato general de instrucciones de máquina de IA-32 contiene un byte de prefijo de instrucción, código de operación, byte Mod R/M, byte de índice de escala (SIB), desplazamiento de dirección y datos inmediatos. Las instrucciones se almacenan en orden little endian, por lo que el byte de prefijo se encuentra en la dirección inicial de la instrucción. Cada instrucción tiene un código de operación, pero el resto de los campos son opcionales. Pocas instrucciones contienen todos los campos; en promedio, la mayoría de las instrucciones son de 2 o 3 bytes. He aquí un breve resumen de los campos:

- El **prefijo de instrucción** ignora los tamaños predeterminados de los operandos.
- El **código de operación** (opcode) identifica a una variable específica de una instrucción. Por ejemplo la instrucción ADD tiene nueve códigos de operación distintos, dependiendo de los tipos de parámetros que se utilicen.
- El campo **Mod R/M** identifica el modo de direccionamiento y los operandos. La notación "R/M" significa *registro* y *modo*. La siguiente tabla describe el campo Mod.

Valores del campo Mod.

Mod	Desplazamiento
00	DISP = 0, disp-bajo y disp-alto están ausentes (a menos que r/m = 110)
01	DISP = disp-bajo con extensión de signo a 16 bits; disp-alto está ausente
10	DISP = se utilizan disp-alto y disp-bajo
11	El campo R/M contiene un número de registro

La siguiente tabla describe el campo R/M para las aplicaciones de 16 bits, cuando MOD = 10 binario.

Valores del campo R/M de 16 bits (para Mod = 10)

R/M	Dirección efectiva
000	[BX + SI] + D16
001	[BX + DI] + D16
010	[BP + SI] + D16
011	[BP + DI] + D16
100	[SI] + D16
101	[DI] + D16

110	[BP] + D16
111	[BX] + D16

*D16 indica un desplazamiento de 16 bits.

- El **byte indice de escala** (SIB) se utiliza para calcular los desplazamientos de los índices de un arreglo.
- El campo **desplazamiento de dirección** contiene el desplazamiento de un operando, o puede sumarse a los registros base e índice en los modos de direccionamiento como base-desplazamiento o base-índice-desplazamiento.
- El campo **datos inmediatos** contiene operandos constantes.

Instrucciones de un solo byte

El tipo más simple de instrucción es uno sin operando, o con un operando implícito. Dichas instrucciones requieren sólo el campo de código de operación, cuyo valor está predeterminado por el conjunto de instrucciones del procesador. La siguiente tabla presenta algunas instrucciones comunes de un solo byte. Pareciera que la instrucción INC DX se escabulló en la tabla por error, pero los diseñadores del conjunto de instrucciones Intel decidieron suministrar códigos de operación únicos para ciertas instrucciones de uso común. Como consecuencia, los incrementos de los registros están optimizados para el tamaño del código y la velocidad de ejecución.

instrucción de un solo byte.

Instrucción	Código de operación
AAAA	37
AAS	3F
CBW	98
LODSB	AC
XLAT	D7
INC DX	42

Movimiento inmediato a un registro

Los operandos inmediatos (constantes) se adjuntan a las instrucciones en orden little endian (el menor byte primero). Nos enfocaremos primero en las instrucciones que mueven valores inmediatos a los registros, dejando pendiente las complicaciones de los modos de direccionamiento de memoria. El formato de codificacion de una instrucción MOV que mueve una palabra inmediata a un registro es **B8 + rw dw**, en donde el valor del byte del código de operación es **B8 + rw**, indicando que se agrega un número de registro (del 0 al 7) a B8; *dw* es el operando tipo palabra inmediato, el byte inferior primero. Los números de los registros que se utilizan en los códigos de operación se presentan en la siguiente tabla. Todos los valores numéricos en los siguientes ejemplos son hexadecimales:

Número de los registros (8/16 bits).

Registro	Código
AX/AL	0
CX/CL	1
DX/DL	2
BX/BL	3
SP/AH	4
BP/CH	5
SI/DH	6
DI/BH	7

Ejemplo: *PUSH CX* La instrucción de máquina es **51**. Los pasos de codificación son los siguientes:

1. El código de operación para PUSH con un operando tipo de 16 bits es **50**.
2. El número de registro para CX es 1, por lo que se suma 1 a 50 para producir el código de operación **51**.

Ejemplo: *MOV AX, 1* La instrucción de máquina es **B8 01 00** (hexadecimal). He aquí cómo está codificada:

1. El código de operación para mover un valor inmediato a un registro de 16 bits es **B8**.
2. El número de registro para AX es 0, por lo que se suma 0 a B8.
3. El operando inmediato (001) se ajunta a la instrucción en orden little endian (01, 00).

Ejemplo: *MOV BX, 1234h* La instrucción de máquina es **BB 34 12**. Los pasos de codificación son los siguientes:

1. El código de operación para mover un valor inmediato a un registro de 16 bits es **B8**.
2. El número de registro BX es 3, por lo que se le suma 3 a B8 para producir el código de operación **BB**.
3. Los bytes del operando inmediato son **34 12**.

Instrucciones en modo de registro

En las instrucciones que utilizan operandos tipo registro, el byte Mod R/M contiene un identificador de 3 bits para cada operando tipo registro. La siguiente tabla presenta las codificaciones de los bits para los registros. La elección de un registro de 8 o 16 bits depende del bit 0 del campo del código de operación: 1 indica un registro de 16 bits y 0 indica un registro de 8 bits.

Identificación de registros en el campo Mod R/M,

R/M	Registro	R/M	Registro
000	AX o AL	100	SP o AH
001	CX o CL	101	BP o CH
010	DX o DL	110	SI o DH
011	BX o BL	111	DI o BH

Por ejemplo, el lenguaje máquina para **MOV AX, BX** es **89 D8**. La codificación Intel de una instrucción MOV de 16 bits, de un registro a cualquier otro operando es **89/r**, mientras que /r indica que un byte Mod R/M sigue el código de operación. El byte R/M está compuesto de tres campos (mod, reg y r/m). Por ejemplo, un valor Mod R/M de D8 contiene los siguientes campos:

mod	reg	r/m
11	011	000

- Los bits 6 a 7 son el campo *mod*, que identifica el modo de direccionamiento. El campo mod es 11, que indica que el campo r/m contiene un número de registro.
- Los bits 3 a 5 son el campo *reg*, que identifica el operando de origen. En nuestro ejemplo, BX es el registro 011.
- Los bits 0 a 2 son el campo *r/m*, que identifica al operando de destino. En nuestro ejemplo, AX es el registro 000.

La siguiente tabla presenta algunos ejemplos más que utilizan operandos tipo registro de 8 bits y 16 bits.

Ejemplos de codificaciones de instrucciones MOV, operandos de registro.

Instrucción	Código de operacón	mod	reg	r/m
mov ax, dx	8B	11	000	010
mov al, dl	8A	11	000	010
mov cx, dx	8B	11	001	010
mov cl, dl	8a	11	001	010

Prefijo de tamaño de operando del procesador IA-32

Ahora vamos a poner nuestra atención en la codificación de instrucciones para los procesadores de 32 bits (IA-32). Algunas instrucciones en lenguaje máquina, generadas para los procesadores IA-32, empiezan con un prefijo de tamaño (66h) que redefine el atributo de segmento predeterminado para la instrucción que modifica. La pregunta es ¿Por qué tener un prefijo de instrucción? Cuando se creó el conjunto de instrucciones del 8088/8086, se usaron casi todos los 256 códigos de operación posibles para manejar instrucciones mediante operandos de 8 y 16 bits. Cuando Intel presentó los procesadores de 32 bits, tuvieron que buscar la forma de inventar nuevos códigos de operación para manejar operandos de 32 bits sin perder la compatibilidad con los procesadores anteriores. Para los programas orientados a los procesadores de 16 bits, agregaron un byte de prefijo a cualquier instrucción que utilizara operandos de 32 bits. Para los programas orientados a los procesadores de 32 bits, los operandos de 32 bits eran los predeterminados, por lo que se agregó un byte de prefijo a cualquier instrucción que utilizara operandos de 16 bits. Los operadores de ocho bits no necesitan prefijo.

Ejemplo: operandos de 16 bits podemos ver cómo funcionan los bytes de prefijo en el modo de 16 bits; para ello hay que ensamblar la instrucción MOV que presentamos anteriormente. La directiva .286 indica el procesador de destino para el código compilado, asegurando (por una parte) que no se utilicen registros de 32 bits. Junto con cada instrucción MOV, mostraremos su codificación:

```
.model small

.286

.code

.stack 100h

main PROC

   mov ax, dx                ; 8B C2

   mov al, dl                ; 8A C2
```

No utiiizamos el archivo Irvine16.inc, ya que está orientado al procesador 386.

Vamos a ensamblar la mismas instrucciones para un procesador de 32 bits, usando la directiva .386; el tamaño de operando predeterminado es de 32 bits. Incluiremos operandos de 16 bits y de 32 bits. La primera instrucción MOV (EAX, EDX) no necesita prefijo, ya que utiliza operandos de 32 bits. La segunda instrucción MOV (AX, DX) requiere un prefijo de tamaño de operando (66), ya que utiliza operandos de 16 bits:

```
.model small

.386

.stack 100h

.code

main PROC

   mov eax, edx                ; 8B C2

   mov ax, dx                  ; 66 8B C2

   mov al, dl                  ; 8A C2
```

Instrucciones en modo de memoria

Si el byte Mod R/M se utilizara sólo para identificar los operandos de tipo registro, la codificación de las instrucciones Intel sería muy simple. De hecho, el lenguaje ensamblador de Intel tiene una amplia variedad de modos de direccionamiento de memoria, lo cual hace que la codificación del byte Mod R/M sea bastante compleja (la complejidad del conjunto de instrucciones IA-32 es un blanco constante de crítica por parte de los que proponen diseños de computadoras con el conjunto reducido de instrucciones).

Pueden especificarse exactamente 256 combinaciones distintas mediante el byte Mod R/M. La siguiente tabla presenta los bytes Mod R/M (en hexadecimal) para Mod 00:

Lista parcial de bytes Mod R/M (segmentos de 16 bits).

Byte:		AL	CL	DL	BL	AH	CH	DH	BH	
Palabra:		AX	CX	DX	BX	SP	BP	SI	DI	
ID de registro:		000	001	010	011	100	101	110	111	
Mod	R/M	Valor de Mod R/M								Dirección efectiva
00	000	00	08	10	18	20	28	30	38	[BX + SI]
	001	01	09	11	19	21	29	31	39	[BX + DI]
	010	02	0A	12	1A	22	2A	32	3A	[BP + SI]
	011	03	0B	13	1B	23	2B	33	3B	[BP + DI]
	100	04	0C	14	1C	24	2C	34	3C	[SI]
	101	05	0D	15	1D	25	2D	35	3D	[DI]
	110	06	0E	16	1E	26	2E	36	3E	Desplazamiento de 16 bits
	111	07	0F	17	1F	27	2F	37	3F	[BX]

He aquí como funciona la codificación de bytes Mod R/M Los dos bits en la columna **Mod** indican grupos de modos de direccionamiento. Por ejemplo, Mod 00 tiene ocho valores posibles para **R/M** (000 a 111 binario) que identifican los tipos de operandos presentados en la columna de **dirección efectiva.**

Supongamos que queremos codificar **MOV AX, [SI]**; los bits Mod son 00 y los bits R/M son 100 binario. Sabemos. Sabemos que AX es el número de registro 000 binario, por lo que el byte Mod R/M completo es 00 000 100 binario o 04 hexadecimal:

mod	reg	r/m
00	000	100

Resumen por: Victor Gutierrez Gutierrez.