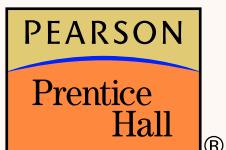
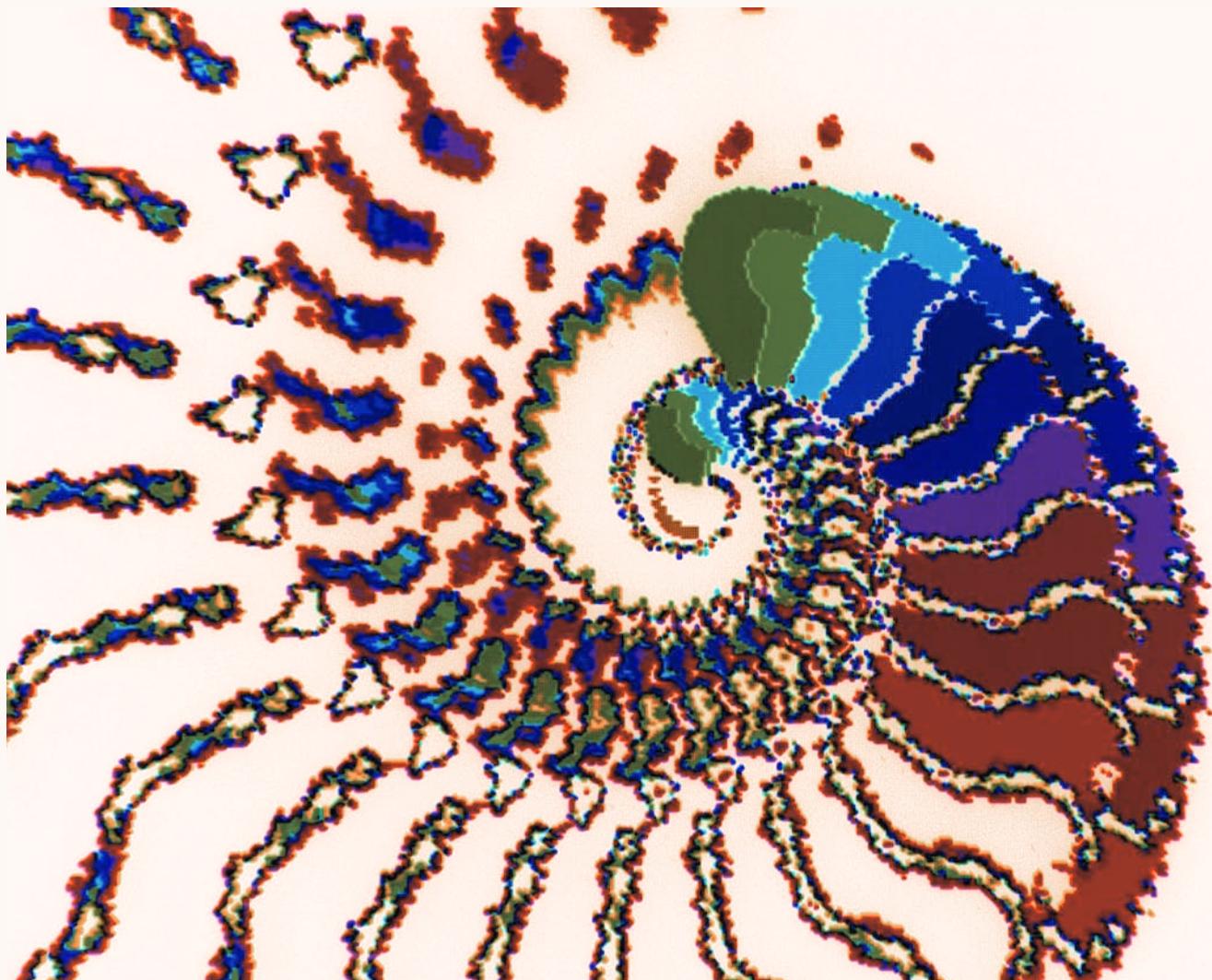


LENGUAJE ENSAMBLADOR

PARA COMPUTADORAS BASADAS EN INTEL®

QUINTA EDICIÓN



KIP R. IRVINE

CARACTERES ASCII DE CONTROL

La siguiente lista muestra los códigos ASCII que se generan al oprimir una combinación de teclas de control. Los nemáticos y las descripciones se refieren a las funciones ASCII que se utilizan para el formato de la pantalla y la impresora, y para las comunicaciones de datos.

Código ASCII*	Ctrl-	Nemónico	Descripción	Código ASCII*	Ctrl-	Nemónico	Descripción
00		NUL	Carácter nulo	10	Ctrl-P	DLE	Escape de vínculo de datos
01	Ctrl-A	SOH	Inicio de encabezado	11	Ctrl-Q	DC1	Control de dispositivo 1
02	Ctrl-B	STX	Inicio de texto	12	Ctrl-R	DC2	Control de dispositivo 2
03	Ctrl-C	ETX	Fin de texto	13	Ctrl-S	DC3	Control de dispositivo 3
04	Ctrl-D	EOT	Fin de transmisión	14	Ctrl-T	DC4	Control de dispositivo 4
05	Ctrl-E	ENQ	Investigación	15	Ctrl-U	NAK	Reconocimiento negativo
06	Ctrl-F	ACK	Reconocimiento	16	Ctrl-V	SYN	Inactividad síncrona
07	Ctrl-G	BEL	Campana	17	Ctrl-W	ETB	Fin del bloque de transmisión
08	Ctrl-H	BS	Retroceso	18	Ctrl-X	CAN	Cancelar
09	Ctrl-I	HT	Tabulación horizontal	19	Ctrl-Y	EM	Fin del medio
0A	Ctrl-J	LF	Avance de línea	1A	Ctrl-Z	SUB	Sustituto
0B	Ctrl-K	VT	Tabulación vertical	1B	Ctrl-I	ESC	Escape
0C	Ctrl-L	FF	Avance de página	1C	Ctrl-	FS	Separador de archivo
0D	Ctrl-M	CR	Retorno de carro	1D	Ctrl-I	GS	Separador de grupo
0E	Ctrl-N	SO	Desplazamiento hacia fuera	24	Ctrl-^	RS	Separador de registro
0F	Ctrl-O	SI	Desplazamiento hacia dentro	25	Ctrl-†	US	Separador de unidad

* Los códigos ASCII están en hexadecimal.

† El código ASCII 1FH es Ctrl-Guion corto (-).

COMBINACIONES ALT+TECLA

Los siguientes códigos de exploración hexadecimales se producen al oprimir la tecla ALT y cada carácter:

Tecla	Código de exploración	Tecla	Código de exploración	Tecla	Código de exploración
1	78	A	1E	N	31
2	79	B	30	O	18
3	7A	C	2E	P	19
4	7B	D	20	Q	10
5	7C	E	12	R	13
6	7D	F	21	S	1F
7	7E	G	22	T	14
8	7F	H	23	U	16
9	80	I	17	V	2F
0	81	J	24	W	11
-	82	K	25	X	2D
=	83	L	26	Y	15
		M	32	Z	2C

decimal	⇒	128	144	160	176	192	208	224	240
	hexa-decimal	8	9	A	B	C	D	E	F
0	0	ç	É	á	⋮	₼	₩	α	≡
1	1	ü	æ	í	⊗	⊥	⊤	β	±
2	2	é	Æ	ó	϶	⊤	⊠	Γ	≥
3	3	â	ô	ú		⊤	⊠	π	≤
4	4	ä	ö	ñ	-	—	⊜	Σ	ʃ
5	5	à	ò	Ñ	`	+	Ϝ	σ	ʃ
6	6	å	û	݁		܁	܂	μ	÷
7	7	ç	ù	܉	܊	܋	܌	τ	≈
8	8	ê	ÿ	܇	܈	܉	܊	܋	○
9	9	ë	ö	܏	ܐ	ܑ	ܒ	ܓ	•
10	A	è	Ü	܏		܊	܏	ܓ	•
11	B	ï	ç	½	ܐ	܊	ܔ	δ	√
12	C	î	£	¼	ܐ	܁	ܔ	∞	n
13	D	ì	¥	ି	ܐ	=	ܔ	ϕ	²
14	E	Ä	Pt	≪	ܐ	ܔ	ܔ	€	■
15	F	Å	f	≫	܏	≠	ܔ	∩	blanco

LENQUAJE ENSAMBLADOR PARA COMPUTADORAS BASADAS EN INTEL®

Quinta edición

KIP R. IRVINE

Florida International University
School of Computing and Information Sciences

TRADUCCIÓN

Alfonso Vidal Romero Elizondo

Ingeniero en Sistemas Electrónicos

*Instituto Tecnológico y de Estudios Superiores
de Monterrey - Campus Monterrey*

REVISIÓN TÉCNICA

María Concepción Villar Cuesta

Armandina J. Leal Flores

Departamento de Ciencias Computacionales

*Instituto Tecnológico y de Estudios Superiores
de Monterrey - Campus Monterrey*

José Miguel Morán Loza

Presidente de la Academia de Sistemas Digitales Avanzados

Centro Universitario de Ciencias Exactas e Ingenierías

Universidad de Guadalajara



México • Argentina • Brasil • Colombia • Costa Rica • Chile • Ecuador
España • Guatemala • Panamá • Perú • Puerto Rico • Uruguay • Venezuela

Datos de catalogación bibliográfica

Irvine, Kip R.

Lenguaje ensamblador para computadoras basadas en Intel®

PEARSON EDUCACIÓN, México, 2008

ISBN: 978-970-26-1081-6

Área: Ingeniería

Formato: 18.5 × 23.5 cms

Páginas: 752

Authorized translation from the English language edition, entitled *Assembly language for intel-based computers, 5e* by Kip R. Irvine, published by Pearson Education, Inc., publishing as Prentice Hall, Copyright ©2007. All rights reserved.

ISBN 0132383101

Traducción autorizada de la edición en idioma inglés. *Assembly language for intel-based computers, 5e* por Kip R. Irvine, publicada por Pearson Education, Inc., publicada como Prentice Hall, Copyright ©2007. Todos los derechos reservados.

Edición en español

Editor: Luis Miguel Cruz Castillo
e-mail: luis.cruz@pearsoned.com
Editor de desarrollo: Bernardino Gutiérrez Hernández
Supervisor de producción: Enrique Trejo Hernández

Edición en inglés

Vice President and Editorial Director, ECS: *Marcia J. Horton*
Executive Editor: *Tracy Dunkelberger*
Associate Editor: *Carole Snyder*
Editorial Assistant: *Christianna Lee*
Executive Managing Editor: *Vince O'Brien*
Managing Editor: *Camille Trentacoste*
Production Editor: *Karen Ettinger*
Director of Creative Services: *Paul Belfanti*
Creative Director: *Juan Lopez*
Managing Editor, AV Management and Production: *Patricia Burns*
Art Editor: *Gregory Dulles*
Manufacturing Manager, ESM: *Alexis Heydt-Long*
Manufacturing Buyer: *Lisa McDowell*
Executive Marketing Manager: *Robin O'Brien*
Marketing Assistant: *Mack Patterson*

QUINTA EDICIÓN, 2008

D.R. © 2008 por Pearson Educación de México, S.A. de C.V.
Atlacomulco 500-5º piso
Col. Industrial Atoto
C.P. 53519, Naucalpan de Juárez, Edo. de México

Cámara Nacional de la Industria Editorial Mexicana. Reg. Núm. 1031.

Prentice Hall es una marca registrada de Pearson Educación de México, S.A. de C.V.

Reservados todos los derechos. Ni la totalidad ni parte de esta publicación pueden reproducirse, registrarse o transmitirse, por un sistema de recuperación de información, en ninguna forma ni por ningún medio, sea electrónico, mecánico, fotoquímico, magnético o electroóptico, por fotocopia, grabación o cualquier otro, sin permiso previo por escrito del editor.

El préstamo, alquiler o cualquier otra forma de cesión de uso de este ejemplar requerirá también la autorización del editor o de sus representantes.



ISBN 10: 970-26-1081-8
ISBN 13: 978-970-26-1081-6

Impreso en México. Printed in Mexico.

1 2 3 4 5 6 7 8 9 0 - 10 09 08

Para Jack y Candy Irvine

CONTENIDO

Prefacio xxi

1 Conceptos básicos 1

1.1 Bienvenido al lenguaje ensamblador 1

- 1.1.1 Preguntas importantes por hacer 2
- 1.1.2 Aplicaciones del lenguaje ensamblador 5
- 1.1.3 Repaso de sección 6

1.2 Concepto de máquina virtual 7

- 1.2.1 Historia de los ensambladores de la PC 9
- 1.2.2 Repaso de sección 9

1.3 Representación de datos 9

- 1.3.1 Números binarios 10
- 1.3.2 Suma binaria 11
- 1.3.3 Tamaños de almacenamiento de enteros 12
- 1.3.4 Enteros hexadecimales 13
- 1.3.5 Enteros con signo 14
- 1.3.6 Almacenamiento de caracteres 16
- 1.3.7 Repaso de sección 18

1.4 Operaciones booleanas 20

- 1.4.1 Tablas de verdad para las funciones booleanas 22
- 1.4.2 Repaso de sección 23

1.5 Resumen del capítulo 23

2 Arquitectura del procesador IA-32 25

2.1 Conceptos generales 25

- 2.1.1 Diseño básico de una microcomputadora 26
- 2.1.2 Ciclo de ejecución de instrucciones 27
- 2.1.3 Lectura de la memoria 30
- 2.1.4 Cómo se ejecutan los programas 31
- 2.1.5 Repaso de sección 32

2.2 Arquitectura del procesador IA-32 33

- 2.2.1 Modos de operación 33
- 2.2.2 Entorno básico de ejecución 34

2.2.3	Unidad de punto flotante	36
2.2.4	Historia del microprocesador Intel	37
2.2.5	Repasso de sección	39
2.3	Administración de memoria del procesador IA-32	39
2.3.1	Modo de direccionamiento real	39
2.3.2	Modo protegido	41
2.3.3	Repasso de sección	43
2.4	Componentes de una microcomputadora IA-32	43
2.4.1	Tarjeta madre	43
2.4.2	Salida de video	44
2.4.3	Memoria	45
2.4.4	Puertos de entrada/salida e interfaces de dispositivos	45
2.4.5	Repasso de sección	46
2.5	Sistema de entrada/salida	46
2.5.1	Cómo funciona todo	46
2.5.2	Repasso de sección	49
2.6	Resumen del capítulo	49

3 Fundamentos del lenguaje ensamblador 51

3.1 Elementos básicos del lenguaje ensamblador 51

3.1.1	Constantes enteras	52
3.1.2	Expresiones enteras	52
3.1.3	Constantes numéricas reales	53
3.1.4	Constantes tipo carácter	54
3.1.5	Constantes tipo cadena	54
3.1.6	Palabras reservadas	54
3.1.7	Identificadores	54
3.1.8	Directivas	55
3.1.9	Instrucciones	55
3.1.10	La instrucción NOP (ninguna operación)	57
3.1.11	Repasso de sección	58

3.2 Ejemplo: suma y resta de enteros 58

3.2.1	Versión alternativa de SumaResta	60
3.2.2	Plantilla de programa	61
3.2.3	Repasso de sección	61

3.3 Ensamblado, enlazado y ejecución de programas 62

3.3.1	El ciclo de ensamblado-enlazado-ejecución	62
3.3.2	Repasso de sección	64

3.4 Definición de datos 64

3.4.1	Tipos de datos intrínsecos	64
3.4.2	Instrucción de definición de datos	64
3.4.3	Definición de datos BYTE y SBYTE	66

- 3.4.4 Definición de datos WORD y SWORD 67
- 3.4.5 Definición de datos DWORD y SDWORD 68
- 3.4.6 Definición de datos QWORD 69
- 3.4.7 Definición de datos TBYTE 69
- 3.4.8 Definición de datos de números reales 69
- 3.4.9 Orden Little Endian 69
- 3.4.10 Agregar variables al programa SumaResta 70
- 3.4.11 Declaración de datos sin inicializar 71
- 3.4.12 Repaso de sección 71

3.5 Constantes simbólicas 72

- 3.5.1 Directiva de signo de igual 72
- 3.5.2 Cálculo de los tamaños de los arreglos y cadenas 73
- 3.5.3 Directiva EQU 74
- 3.5.4 Directiva TEXTEQU 74
- 3.5.5 Repaso de sección 75

3.6 Programación en modo de direccionamiento real (opcional) 75

- 3.6.1 Cambios básicos 75

3.7 Resumen del capítulo 76

3.8 Ejercicios de programación 77

4 Transferencias de datos, direccionamiento y aritmética 79

4.1 Instrucciones de transferencia de datos 79

- 4.1.1 Introducción 79
- 4.1.2 Tipos de operandos 80
- 4.1.3 Operandos directos de memoria 80
- 4.1.4 Instrucción MOV 81
- 4.1.5 Extensión con cero y con signo de enteros 82
- 4.1.6 Instrucciones LAHF y SAHF 84
- 4.1.7 Instrucción XCHG 84
- 4.1.8 Operandos de desplazamiento directo 84
- 4.1.9 Programa de ejemplo (movimientos) 85
- 4.1.10 Repaso de sección 86

4.2 Suma y resta 87

- 4.2.1 Instrucciones INC y DEC 87
- 4.2.2 Instrucción ADD 87
- 4.2.3 Instrucción SUB 88
- 4.2.4 Instrucción NEG 88
- 4.2.5 Implementación de expresiones aritméticas 89
- 4.2.6 Banderas afectadas por la suma y la resta 89
- 4.2.7 Programa de ejemplo (SumaResta3) 92
- 4.2.8 Repaso de sección 93

4.3 Operadores y directivas relacionadas con los datos 94

- 4.3.1 Operador OFFSET 94
- 4.3.2 Directiva ALIGN 95
- 4.3.3 Operador PTR 95
- 4.3.4 Operador TYPE 96
- 4.3.5 Operador LENGTHOF 97
- 4.3.6 Operador SIZEOF 97
- 4.3.7 Directiva LABEL 97
- 4.3.8 Repaso de sección 98

4.4 Direccionamiento indirecto 99

- 4.4.1 Operandos indirectos 99
- 4.4.2 Arreglos 100
- 4.4.3 Operandos indexados 101
- 4.4.4 Apuntadores 102
- 4.4.5 Repaso de sección 103

4.5 Instrucciones JMP y LOOP 104

- 4.5.1 Instrucción JMP 104
- 4.5.2 Instrucción LOOP 105
- 4.5.3 Suma de un arreglo de enteros 106
- 4.5.4 Copia de una cadena 106
- 4.5.5 Repaso de sección 107

4.6 Resumen del capítulo 108**4.7 Ejercicios de programación 109****5 Procedimientos 111****5.1 Introducción 111****5.2 Enlace con una biblioteca externa 111**

- 5.2.1 Antecedentes 112
- 5.2.2 Repaso de sección 113

5.3 La biblioteca de enlace del libro 113

- 5.3.1 Generalidades 113
- 5.3.2 Descripciones de los procedimientos individuales 115
- 5.3.3 Programas de prueba de la biblioteca 125
- 5.3.4 Repaso de sección 129

5.4 Operaciones de la pila 129

- 5.4.1 La pila en tiempo de ejecución 129
- 5.4.2 Instrucciones PUSH y POP 131
- 5.4.3 Repaso de sección 134

5.5 Definición y uso de los procedimientos 134

- 5.5.1 Directiva PROC 134
- 5.5.2 Instrucciones CALL y RET 136

- 5.5.3 Ejemplo: suma de un arreglo de enteros 139
- 5.5.4 Diagramas de flujo 140
- 5.5.5 Almacenamiento y restauración de registros 140
- 5.5.6 Repaso de sección 142

5.6 Diseño de programas mediante el uso de procedimientos 143

- 5.6.1 Programa para sumar enteros (diseño) 143
- 5.6.2 Implementación de la suma de enteros 145
- 5.6.3 Repaso de sección 147

5.7 Resumen del capítulo 147

5.8 Ejercicios de programación 148

6 Procesamiento condicional 150

6.1 Introducción 150

6.2 Instrucciones booleanas y de comparación 151

- 6.2.1 Las banderas de la CPU 151
- 6.2.2 Instrucción AND 152
- 6.2.3 Instrucción OR 153
- 6.2.4 Instrucción XOR 154
- 6.2.5 Instrucción NOT 155
- 6.2.6 Instrucción TEST 155
- 6.2.7 Instrucción CMP 156
- 6.2.8 Cómo establecer y borrar banderas individuales de la CPU 157
- 6.2.9 Repaso de sección 157

6.3 Saltos condicionales 158

- 6.3.1 Estructuras condicionales 158
- 6.3.2 Instrucción *Jcond* 158
- 6.3.3 Tipos de instrucciones de saltos condicionales 159
- 6.3.4 Aplicaciones de saltos condicionales 163
- 6.3.5 Instrucciones de prueba de bits (opcional) 167
- 6.3.6 Repaso de sección 168

6.4 Instrucciones de saltos condicionales 169

- 6.4.1 Instrucciones LOOPZ y LOOPE 169
- 6.4.2 Instrucciones LOOPNZ y LOOPNE 169
- 6.4.3 Repaso de sección 170

6.5 Estructuras condicionales 170

- 6.5.1 Instrucciones IF con estructura de bloque 170
- 6.5.2 Expresiones compuestas 173
- 6.5.3 Ciclos WHILE 174
- 6.5.4 Selección controlada por tablas 177
- 6.5.5 Repaso de sección 178

6.6 Aplicación: máquinas de estado finito 179

- 6.6.1 Validación de una cadena de entrada 180

- 6.6.2 Validación de un entero con signo 180
- 6.6.3 Repaso de sección 183

6.7 Directivas de decisión 184

- 6.7.1 Comparaciones con signo y sin signo 185
- 6.7.2 Expresiones compuestas 186
- 6.7.3 Directivas .REPEAT y .WHILE 188

6.8 Resumen del capítulo 189

6.9 Ejercicios de programación 190

7 Aritmética de enteros 193

7.1 Introducción 193

7.2 Instrucciones de desplazamiento y rotación 194

- 7.2.1 Desplazamientos lógicos y desplazamientos aritméticos 194
- 7.2.2 Instrucción SHL 195
- 7.2.3 Instrucción SHR 196
- 7.2.4 Instrucciones SAL y SAR 196
- 7.2.5 Instrucción ROL 197
- 7.2.6 Instrucción ROR 198
- 7.2.7 Instrucciones RCL y RCR 198
- 7.2.8 Desbordamiento con signo 199
- 7.2.9 Instrucciones SHLD/SHRD 199
- 7.2.10 Repaso de sección 200

7.3 Aplicaciones de desplazamiento y rotación 201

- 7.3.1 Desplazamiento de varias dobles palabras 201
- 7.3.2 Multiplicación binaria 202
- 7.3.3 Visualización de bits binarios 202
- 7.3.4 Aislamiento de campos de datos de archivos de MS-DOS 203
- 7.3.5 Repaso de sección 203

7.4 Instrucciones de multiplicación y división 204

- 7.4.1 Instrucción MUL 204
- 7.4.2 Instrucción IMUL 205
- 7.4.3 Evaluación del rendimiento de las operaciones de multiplicación 207
- 7.4.4 Instrucción DIV 208
- 7.4.5 División de enteros con signo 209
- 7.4.6 Implementación de expresiones aritméticas 211
- 7.4.7 Repaso de sección 212

7.5 Suma y resta extendidas 213

- 7.5.1 Instrucción ADC 213
- 7.5.2 Ejemplo de suma extendida 213
- 7.5.3 Instrucción SBB 214
- 7.5.4 Repaso de sección 215

7.6 Aritmética ASCII y con decimales desempaquetados 215

- 7.6.1 Instrucción AAA 216
- 7.6.2 Instrucción AAS 218
- 7.6.3 Instrucción AAM 218
- 7.6.4 Instrucción AAD 218
- 7.6.5 Repaso de sección 219

7.7 Aritmética con decimales empaquetados 219

- 7.7.1 Instrucción DAA 219
- 7.7.2 Instrucción DAS 220
- 7.7.3 Repaso de sección 220

7.8 Resumen del capítulo 221**7.9 Ejercicios de programación 222****8 Procedimientos avanzados 224****8.1 Introducción 224****8.2 Marcos de pila 225**

- 8.2.1 Parámetros de pila 225
- 8.2.2 Variables locales 233
- 8.2.3 Instrucciones ENTER y LEAVE 236
- 8.2.4 Directiva LOCAL 237
- 8.2.5 Procedimiento WriteStackFrame 240
- 8.2.6 Repaso de sección 241

8.3 Recursividad 242

- 8.3.1 Cálculo recursivo de una suma 243
- 8.3.2 Cálculo de un factorial 243
- 8.3.3 Repaso de sección 245

8.4 Directiva .MODEL 246

- 8.4.1 Especificadores de lenguaje 247
- 8.4.2 Repaso de sección 248

8.5 INVOKE, ADDR, PROC y PROTO (opcional) 248

- 8.5.1 Directiva INVOKE 248
- 8.5.2 Operador ADDR 249
- 8.5.3 Directiva PROC 250
- 8.5.4 Directiva PROTO 253
- 8.5.5 Clasificaciones de parámetros 255
- 8.5.6 Ejemplo: intercambio de dos enteros 256
- 8.5.7 Tips de depuración 256
- 8.5.8 Repaso de sección 257

8.6 Creación de programas con varios módulos 258

- 8.6.1 Ocultar y exportar nombres de procedimientos 258
- 8.6.2 Llamadas a procedimientos externos 258

- 8.6.3 Uso de variables y símbolos a través de los límites de los módulos 259
- 8.6.4 Ejemplo: programa SumaArreglo 260
- 8.6.5 Creación de módulos mediante el uso de Extern 261
- 8.6.6 Creación de módulos mediante el uso de INVOKES y PROTO 264
- 8.6.7 Repaso de sección 266

8.7 Resumen del capítulo 267

8.8 Ejercicios de programación 268

9 Cadenas y arreglos 269

9.1 Introducción 269

9.2 Instrucciones primitivas de cadenas 270

- 9.2.1 MOVSB, MOVSW y MOVSD 271
- 9.2.2 CMPSB, CMPSW y CMPSD 272
- 9.2.3 SCASB, SCASW y SCASD 274
- 9.2.4 STOSB, STOSW y STOSD 274
- 9.2.5 LODSB, LODSW y LODSD 275
- 9.2.6 Repaso de sección 275

9.3 Procedimientos de cadenas seleccionados 276

- 9.3.1 Procedimiento Str_compare 276
- 9.3.2 Procedimiento Str_length 277
- 9.3.3 Procedimiento Str_copy 278
- 9.3.4 Procedimiento Str_trim 278
- 9.3.5 Procedimiento Str_ucase 279
- 9.3.6 Programa de demostración de la biblioteca de cadenas 280
- 9.3.7 Repaso de sección 282

9.4 Arreglos bidimensionales 282

- 9.4.1 Ordenamiento de filas y columnas 282
- 9.4.2 Operandos base-índice 283
- 9.4.3 Operandos base-índice-desplazamiento 285
- 9.4.4 Repaso de sección 285

9.5 Búsqueda y ordenamiento de arreglos de enteros 285

- 9.5.1 Ordenamiento de burbuja 286
- 9.5.2 Búsqueda binaria 287
- 9.5.3 Repaso de sección 293

9.6 Resumen del capítulo 294

9.7 Ejercicios de programación 295

10 Estructuras y macros 299

10.1 Estructuras 299

- 10.1.1 Definición de estructuras 300

- 10.1.2 Declaración de variables de estructura 301
- 10.1.3 Referencias a variables de estructura 302
- 10.1.4 Ejemplo: mostrar la hora del sistema 305
- 10.1.5 Estructuras que contienen estructuras 307
- 10.1.6 Ejemplo: paso del borracho 307
- 10.1.7 Declaración y uso de uniones 310
- 10.1.8 Repaso de sección 312

10.2 Macros 313

- 10.2.1 Generalidades 313
- 10.2.2 Definición de macros 313
- 10.2.3 Invocación de macros 314
- 10.2.4 Características adicionales de los macros 315
- 10.2.5 Uso de la biblioteca de macros del libro 318
- 10.2.6 Programa de ejemplo: envolturas 324
- 10.2.7 Repaso de sección 325

10.3 Directivas de ensamblado condicional 326

- 10.3.1 Comprobación de argumentos faltantes 326
- 10.3.2 Inicializadores de argumentos predeterminados 328
- 10.3.3 Expresiones booleanas 328
- 10.3.4 Directivas IF, ELSE y ENDIF 328
- 10.3.5 Las directivas IFIDN e IFIDNI 329
- 10.3.6 Ejemplo: suma de la fila de una matriz 330
- 10.3.7 Operadores especiales 333
- 10.3.8 Macrofunciones 336
- 10.3.9 Repaso de sección 337

10.4 Definición de bloques de repetición 338

- 10.4.1 Directiva WHILE 338
- 10.4.2 Directiva REPEAT 338
- 10.4.3 Directiva FOR 339
- 10.4.4 Directiva FORC 340
- 10.4.5 Ejemplo: lista enlazada 340
- 10.4.6 Repaso de sección 342

10.5 Resumen del capítulo 342

10.6 Ejercicios de programación 343

11 Programación en MS Windows 346

11.1 Programación de la consola Win32 346

- 11.1.1 Antecedentes 347
- 11.1.2 Funciones de la consola Win32 350
- 11.1.3 Visualización de un cuadro de mensaje 352
- 11.1.4 Entrada de consola 354
- 11.1.5 Salida de consola 360
- 11.1.6 Lectura y escritura de archivos 361

- 11.1.7 E/S de archivos en la biblioteca Irvine32 365
- 11.1.8 Prueba de los procedimientos de E/S de archivos 367
- 11.1.9 Manipulación de ventanas de consola 370
- 11.1.10 Control del cursor 373
- 11.1.11 Control del color de texto 373
- 11.1.12 Funciones de hora y fecha 375
- 11.1.13 Repaso de sección 379

11.2 Escritura de una aplicación gráfica de Windows 379

- 11.2.1 Estructuras necesarias 380
- 11.2.2 La función MessageBox 381
- 11.2.3 El procedimiento WinMain 382
- 11.2.4 El procedimiento WinProc 382
- 11.2.5 El procedimiento ErrorHandler 383
- 11.2.6 Listado del programa 383
- 11.2.7 Repaso de sección 386

11.3 Asignación dinámica de memoria 387

- 11.3.1 Programas PruebaMonton 390
- 11.3.2 Repaso de sección 393

11.4 Administración de memoria en la familia IA-32 393

- 11.4.1 Direcciones lineales 394
- 11.4.2 Traducción de páginas 397
- 11.4.3 Repaso de sección 398

11.5 Resumen del capítulo 399

11.6 Ejercicios de programación 400

12 Interfaz con lenguajes de alto nivel 402

12.1 Introducción 402

- 12.1.1 Convenciones generales 402
- 12.1.2 Repaso de sección 404

12.2 Código ensamblador en línea 404

- 12.2.1 La directiva __asm en Microsoft Visual C++ 404
- 12.2.2 Ejemplo de cifrado de archivos 406
- 12.2.3 Repaso de sección 409

12.3 Enlace con C/C++ en modo protegido 409

- 12.3.1 Uso de lenguaje ensamblador para optimizar código en C++ 410
- 12.3.2 Llamadas a funciones en C y C++ 415
- 12.3.3 Ejemplo de tabla de multiplicación 416
- 12.3.4 Llamadas a funciones de la biblioteca de C 419
- 12.3.5 Programa de listado de directorios 422
- 12.3.6 Repaso de sección 423

12.4 Enlace con C/C++ en modo de direccionamiento real 423

- 12.4.1 Enlace con Borland C++ 424

- 12.4.2 Ejemplo: LeerSector 425
- 12.4.3 Ejemplo: enteros aleatorios grandes 428
- 12.4.4 Repaso de sección 430

12.5 Resumen del capítulo 430

12.6 Ejercicios de programación 431

13 Programación en MS-DOS de 16 bits 432

13.1 MS-DOS y la IBM-PC 432

- 13.1.1 Organización de la memoria 433
- 13.1.2 Redirección de entrada-salida 434
- 13.1.3 Interrupciones de software 435
- 13.1.4 Instrucción INT 435
- 13.1.5 Codificación para los programas de 16 bits 436
- 13.1.6 Repaso de sección 437

13.2 Llamadas a funciones de MS-DOS (INT 21h) 438

- 13.2.1 Funciones de salida selectas 439
- 13.2.2 Ejemplo de programa:Hola programador 441
- 13.2.3 Funciones de entrada selectas 442
- 13.2.4 Funciones de fecha/hora 446
- 13.2.5 Repaso de sección 449

13.3 Servicios estándar de E/S de archivos de MS-DOS 449

- 13.3.1 Crear o abrir un archivo (716Ch) 451
- 13.3.2 Cerrar manejador de archivo (3Eh) 452
- 13.3.3 Mover apuntador de archivo (42h) 452
- 13.3.4 Obtener la fecha y hora de la creación de un archivo 453
- 13.3.5 Procedimientos de biblioteca selectos 453
- 13.3.6 Ejemplo: leer y copiar un archivo de texto 454
- 13.3.7 Leer la cola de comandos de MS-DOS 456
- 13.3.8 Ejemplo: creación un archivo binario 458
- 13.3.9 Repaso de sección 461

13.4 Resumen del capítulo 461

13.5 Ejercicios del capítulo 463

14 Fundamentos de los discos 464

14.1 Sistemas de almacenamiento en disco 464

- 14.1.1 Pistas, cilindros y sectores 465
- 14.1.2 Particiones de disco (volúmenes) 466
- 14.1.3 Repaso de sección 468

14.2 Sistemas de archivos 468

- 14.2.1 FAT12 469

- 14.2.2 FAT16 469
- 14.2.3 FAT32 469
- 14.2.4 NTFS 470
- 14.2.5 Áreas principales del disco 470
- 14.2.6 Repaso de sección 471

14.3 Directorio de disco 472

- 14.3.1 Estructura de directorios de MS-DOS 473
- 14.3.2 Nombres de archivos extensos en MS Windows 475
- 14.3.3 Tabla de asignación de archivos (FAT) 476
- 14.3.4 Repaso de sección 477

14.4 Lectura y escritura de sectores de disco (7305h) 477

- 14.4.1 Programa para visualización de sectores 478
- 14.4.2 Repaso de sección 482

14.5 Funciones de archivo a nivel de sistema 482

- 14.5.1 Obtener el espacio libre del disco (7303h) 483
- 14.5.2 Crear subdirectorio (39h) 485
- 14.5.3 Eliminar subdirectorio (3Ah) 486
- 14.5.4 Establecer el directorio actual (3Bh) 486
- 14.5.5 Obtener el directorio actual (47h) 486
- 14.5.6 Obtener y establecer atributos de archivo (7143h) 486
- 14.5.7 Repaso de sección 487

14.6 Resumen del capítulo 487

14.7 Ejercicios de programación 488

15 Programación a nivel del BIOS 490

15.1 Introducción 490

- 15.1.1 Área de datos del BIOS 491

15.2 Entrada de teclado mediante INT 16h 492

- 15.2.1 Cómo funciona el teclado 492
- 15.2.2 Funciones de INT 16h 493
- 15.2.3 Repaso de sección 497

15.3 Programación de VIDEO con INT 10h 498

- 15.3.1 Fundamentos 498
- 15.3.2 Control del color 499
- 15.3.3 Funciones de video de INT 10h 501
- 15.3.4 Ejemplos de procedimientos de la biblioteca 511
- 15.3.5 Repaso de sección 512

15.4 Dibujo de gráficos mediante INT 10h 512

- 15.4.1 Funciones de INT 10h relacionadas con píxeles 513
- 15.4.2 Programa DibujarLinea 514

- 15.4.3 Programa de coordenadas cartesianas 515
- 15.4.4 Conversión de coordenadas cartesianas a coordenadas de pantalla 517
- 15.4.5 Repaso de sección 518

15.5 Gráficos de mapas de memoria 519

- 15.5.1 Modo 13h: 320 x 200, 256 colores 519
- 15.5.2 Programa de gráficos de mapas de memoria 520
- 15.5.3 Repaso de sección 523

15.6 Programación del ratón 523

- 15.6.1 Funciones INT 33h para el ratón 523
- 15.6.2 Programa para rastrear el ratón 528
- 15.6.3 Repaso de sección 532

15.7 Resumen del capítulo 533

15.8 Ejercicios del capítulo 534

16 Programación experta en MS-DOS 536

16.1 Introducción 536

16.2 Definición de segmentos 537

- 16.2.1 Directivas de segmento simplificadas 537
- 16.2.2 Definiciones explícitas de segmentos 539
- 16.2.3 Redefiniciones de segmentos 542
- 16.2.4 Combinación de segmentos 542
- 16.2.5 Repaso de sección 543

16.3 Estructura de un programa en tiempo de ejecución 544

- 16.3.1 Prefijo de segmento de programa 544
- 16.3.2 Programas COM 545
- 16.3.3 Programas EXE 546
- 16.3.4 Repaso de sección 547

16.4 Manejo de interrupciones 548

- 16.4.1 Interrupciones de hardware 549
- 16.4.2 Instrucciones de control de interrupciones 550
- 16.4.3 Escritura de un manejador de interrupciones personalizado 551
- 16.4.4 Programas TSR (Terminar y permanecer residente) 553
- 16.4.5 Aplicación: el programa No_reinicio 554
- 16.4.6 Repaso de sección 557

16.5 Control de hardware mediante el uso de puertos de E/S 558

- 16.5.1 Puertos de entrada-salida 558
- 16.5.2 Programa de sonido de PC 558

16.6 Resumen del capítulo 560

17 Procesamiento de punto flotante y codificación de instrucciones 562

17.1 Representación binaria de punto flotante 562

- 17.1.1 Representación de punto flotante binaria IEEE 563
- 17.1.2 El exponente 564
- 17.1.3 Números de punto flotante binarios normalizados 565
- 17.1.4 Creación de la representación IEEE 565
- 17.1.5 Conversión de fracciones decimales a reales binarios 567
- 17.1.6 Repaso de sección 568

17.2 Unidad de punto flotante 569

- 17.2.1 Pila de registros FPU 569
- 17.2.2 Redondeo 571
- 17.2.3 Excepciones de punto flotante 573
- 17.2.4 Conjunto de instrucciones de punto flotante 573
- 17.2.5 Instrucciones aritméticas 576
- 17.2.6 Comparación de valores de punto flotante 579
- 17.2.7 Lectura y escritura de valores de punto flotante 582
- 17.2.8 Sincronización de excepciones 583
- 17.2.9 Ejemplos de código 584
- 17.2.10 Aritmética de modo mixto 585
- 17.2.11 Enmascaramiento y desenmascaramiento de excepciones 586
- 17.2.12 Repaso de sección 587

17.3 Codificación de instrucciones Intel 588

- 17.3.1 Formato de instrucciones IA-32 588
- 17.3.2 Instrucciones de un solo byte 589
- 17.3.3 Movimiento inmediato a un registro 590
- 17.3.4 Instrucciones en modo de registro 591
- 17.3.5 Prefijo de tamaño de operando del procesador IA-32 591
- 17.3.6 Instrucciones en modo de memoria 592
- 17.3.7 Repaso de sección 595

17.4 Resumen del capítulo 596

17.5 Ejercicios de programación 597

Apéndice A Referencia de MASM 600

Apéndice B El conjunto de instrucciones IA-32 619

Apéndice C Interrupciones del BIOS y de MS-DOS 650

Apéndice D Respuestas a las preguntas de repaso 659

Índice 705

PREFACIO

La quinta edición de *Lenguaje ensamblador para computadoras basadas en Intel*, enseña la programación en lenguaje ensamblador y la arquitectura del procesador Intel IA-32; el texto es apropiado para los siguientes cursos universitarios:

- Programación en lenguaje ensamblador.
- Fundamentos de los sistemas computacionales.
- Fundamentos de la arquitectura computacional.

Los estudiantes utilizan los procesadores Intel o AMD y programan con **Microsoft Macro Assembler (MASM) 8.0**, que se ejecuta en cualquiera de las siguientes plataformas MS-Windows: Windows 95, 98, Millenium, NT, 2000 y XP.

Aunque este libro se diseñó en un principio como texto de programación para estudiantes universitarios, ha evolucionado a lo largo de los últimos 15 años en algo mucho más completo. Muchas universidades utilizan el libro para sus cursos introductorios de arquitectura computacional. Como una muestra de su popularidad, la cuarta edición se tradujo al coreano, chino, francés, ruso y polaco.

Énfasis de los temas Esta edición incluye temas que conducen de una manera natural hacia cursos subsiguientes en arquitectura computacional, sistemas operativos y escritura de compiladores:

- Concepto de máquina virtual.
- Operaciones booleanas elementales.
- Ciclo de ejecución de instrucciones.
- Acceso a memoria e intercambio (handshaking).
- Interrupciones y sondeo.
- Concepto de canalización y superescalares.
- E/S basada en hardware.
- Representación binaria para punto flotante.

Otros temas se relacionan específicamente a la arquitectura Intel IA-32:

- Memoria protegida y paginación en la arquitectura IA-32.
- Segmentación de memoria en modo de direccionamiento real.
- Manejo de interrupciones de 16 bits.
- Llamadas al sistema en MS-DOS y BIOS (interrupciones).
- Arquitectura y programación de la Unidad de punto flotante de IA-32.
- Codificación de instrucciones de IA-32.

Ciertos ejemplos que aparecen en el libro se pueden emplear en cursos que se imparten en la parte final de un plan de estudios de ciencias computacionales:

- Algoritmos de búsqueda y ordenación.
- Estructuras de lenguajes de alto nivel.
- Máquinas de estado finito.
- Ejemplos de optimización de código.

Mejoras en la quinta edición En esta edición hemos agregado una variedad de mejoras y nueva información, que mencionamos en la siguiente tabla, por número de capítulo:

Capítulo	Mejoras
2	Una explicación mejorada del ciclo de ejecución de instrucciones.
5	Una biblioteca de vínculos expandida, con subrutinas adicionales para escribir interfaces de usuario robustas, calcular la sincronización de los programas, generar enteros seudoaleatorios y analizar cadenas de enteros. La documentación de la biblioteca tiene mejoras considerables.
6	Una explicación mejorada de la codificación de saltos condicionales y los rangos de saltos relativos.
7	Se agregaron las instrucciones IMUL de dos y tres operandos. Se muestran comparaciones de rendimiento para distintos enfoques, en relación con la multiplicación de enteros.
8	Se rediseñó por completo, de manera que se expliquen primero los detalles de bajo nivel de los marcos de pila (registros de activación), antes de presentar las directivas de alto nivel INVOKE y PROC de MASM.
10	Una documentación mejorada de la biblioteca de macros del libro.
11	Nuevo tema: Asignación de memoria dinámica en aplicaciones MS-Windows. Se mejoró la cobertura sobre el manejo de archivos y reporte de errores en aplicaciones MS-Windows.
12	Una cobertura mejorada acerca de las llamadas a funciones de C y C++ desde lenguaje ensamblador.
17	Introducción al conjunto de instrucciones de punto flotante de la arquitectura IA-32. Tipos de datos de punto flotante. Codificación y decodificación de instrucciones IA-32.

Aún sigue siendo un libro de programación Este libro continúa con su misión original: enseñar a los estudiantes cómo escribir y depurar programas a nivel de máquina. Nunca sustituirá a un libro completo sobre arquitectura computacional, pero ofrece a los estudiantes la experiencia práctica de escribir software en un entorno que les enseñe cómo funciona una computadora. Nuestra premisa es que los estudiantes retienen mejor el conocimiento cuando se combina la teoría con la experiencia. En un curso de ingeniería, los estudiantes construyen prototipos; en un curso de arquitectura computacional, los estudiantes deberían escribir programas a nivel de máquina. En ambos casos, obtendrán una experiencia memorable que les brindará la confianza de trabajar en cualquier entorno orientado a SO/máquina.

Modo real y modo protegido Esta edición hace énfasis en el modo protegido de 32 bits, pero cuenta con tres capítulos dedicados a la programación en modo real. Por ejemplo, hay un capítulo completo acerca de la programación del BIOS para el teclado, la pantalla de video (incluyendo gráficos) y el ratón. Otro capítulo trata acerca de la programación en MS-DOS mediante el uso de interrupciones (llamadas al sistema). Los estudiantes pueden sacar provecho de la programación directa del hardware y del BIOS.

Casi todos los ejemplos en la primera mitad del libro se presentan como aplicaciones orientadas a texto de 32 bits, que se ejecutan en modo protegido usando el modelo de memoria plana. Este enfoque es maravilloso, tan sólo porque evita las complicaciones relacionadas con el direccionamiento tipo segmento-desplazamiento. Los párrafos marcados en forma especial y los cuadros contextuales destacan las diferencias ocasionales entre la programación en modo protegido y en modo real. La mayoría de las diferencias se abstraen mediante las bibliotecas de vínculos paralelas del libro, para la programación en modo real y modo protegido.

Bibliotecas de vínculos Suministramos dos versiones de la biblioteca de vínculos que utilizan los estudiantes para las operaciones básicas de entrada-salida, simulaciones, sincronización y demás cosas útiles. La versión de 32 bits (*Irvine32.lib*) se ejecuta en modo protegido y envía su salida a la consola Win32. La versión de 16 bits (*Irvine16.lib*) se ejecuta en modo de direccionamiento real. En el sitio Web del libro se ofrece el código fuente completo para las bibliotecas. Estas bibliotecas de vínculos están disponibles sólo como apoyo, no para evitar que los estudiantes aprendan a programar la entrada-salida por su cuenta. Alentamos a los estudiantes para que creen sus propias bibliotecas.

Software y ejemplos incluidos Todos los programas de ejemplo se probaron con Microsoft Macro Assembler versión 8.0. Las aplicaciones en C++ de 32 bits del capítulo 12 se probaron con Microsoft Visual C++ .NET. Los programas en modo de direccionamiento real del capítulo 12 (enlazados a C++) se ensamblaron con Borland Turbo Assembler (TASM).

Información del sitio Web En el sitio Web www.pearsoneducacion.net/irvine, encontrará las actualizaciones y correcciones a este libro (en inglés) incluyendo proyectos de programación adicionales, para que los instructores los asignen al final de cada capítulo así como código en español. Si por alguna razón no puede acceder a este sitio, visite <http://www.asmirevine.com> donde encontrará información acerca del libro y un vínculo hacia su sitio Web actual.

Objetivos generales

Los siguientes objetivos de este libro están diseñados para ampliar el interés y conocimiento del estudiante sobre los temas relacionados con el lenguaje ensamblador:

- Arquitectura y programación de los procesadores IA-32 de Intel.
- Programación en modo de direccionamiento real y en modo protegido.
- Directivas, macros, operadores y estructura de programas en lenguaje ensamblador.
- Metodología de programación, para mostrar cómo usar el lenguaje ensamblador para crear herramientas de software a nivel de sistema y programas de aplicación.
- Manipulación del hardware de computadora.
- Interacción entre los programas en lenguaje ensamblador, el sistema operativo y otros programas de aplicación.

Uno de nuestros objetivos es ayudar a los estudiantes a lidiar con los problemas de la programación mediante un enfoque mental a nivel de máquina. Es importante pensar en la CPU como una herramienta interactiva, y aprender a supervisar su operación de la forma más directa posible. Un depurador es el mejor amigo del programador, no sólo para atrapar errores, sino también como una herramienta educativa que nos enseña acerca de la CPU y el sistema operativo. Motivamos a los estudiantes para que busquen más allá de la superficie de los lenguajes de alto nivel, para que descubran que la mayoría de los lenguajes de programación están diseñados para ser portables y, por lo tanto, independientes de sus equipos anfitriones.

Además de los ejemplos cortos, este libro contiene cientos de programas listos para ejecutarse, los cuales demuestran el uso de instrucciones o ideas, a medida que se presentan en el libro. Los materiales de referencia, como las guías a las interrupciones de MS-DOS y los nemáticos de las instrucciones, están disponibles al final del libro.

Requisitos previos El lector deberá ser capaz de programar hábilmente cuando menos en algún otro lenguaje de programación, de preferencia en Java, C o C++. Uno de los capítulos trata acerca de la interconexión con C++, por lo que sería muy útil tener un compilador a la mano. He utilizado este libro en el salón de clases con estudiantes con maestrías en ciencias computacionales y sistemas de información administrativa; también se ha usado en diversos cursos de ingeniería.

Características

Listados completos de programas En la página Web encontrará el código fuente de los ejemplos de este libro, y en el sitio en inglés hay listados adicionales. Se suministra una biblioteca completa de vínculos, la cual contiene más de 30 procedimientos que simplifican la entrada y salida del usuario, el procesamiento numérico, el manejo de discos y archivos, y el de cadenas. En las etapas iniciales del curso, los estudiantes pueden utilizar esta biblioteca para mejorar sus programas para posteriormente crear sus propios procedimientos y agregarlos a la biblioteca.

Lógica de programación Dos de los capítulos enfatizan la lógica booleana y la manipulación a nivel de bits. Hicimos nuestro mejor esfuerzo por tratar de relacionar la lógica de la programación de alto nivel con los detalles de bajo nivel de la máquina. Este enfoque ayuda a los estudiantes a crear implementaciones más eficientes y a comprender mejor la forma en que los compiladores generan código objeto.

Conceptos de hardware y sistemas operativos Los primeros dos capítulos introducen los conceptos básicos de hardware y representación de datos, incluyendo números binarios, arquitectura de la CPU, banderas de estado y asignación de memoria. Una investigación acerca del hardware de computadora y una perspectiva histórica de la familia de procesadores Intel ayudan a los estudiantes a comprender mejor el sistema computacional en el que van a programar.

Enfoque hacia la programación estructurada En el capítulo 5 se inicia un énfasis en los procedimientos y la descomposición funcional. Se proporcionan ejercicios de programación más complejos a los estudiantes, con lo que se ven obligados a enfocarse en el diseño, antes de empezar a escribir código.

Conceptos de almacenamiento en disco Los estudiantes aprenden los principios fundamentales detrás del sistema de almacenamiento en disco en los sistemas basados en MS-Windows, desde los puntos de vista de hardware y de software.

Creación de bibliotecas de vínculos Los estudiantes pueden agregar sus propios procedimientos a la biblioteca de vínculos del libro, así como crear algunas nuevas. En este libro aprenderán a usar un enfoque de “caja de herramientas” hacia la programación, y a escribir código que sea útil en más de un programa.

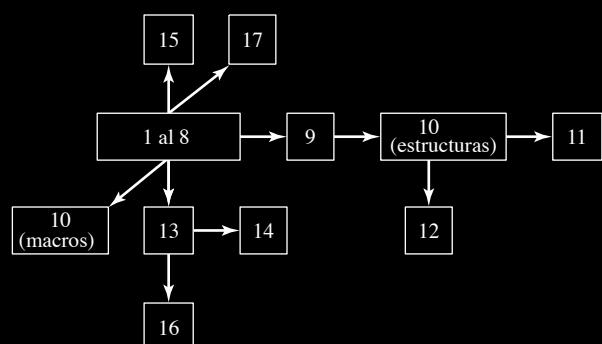
Macros y estructuras Existe un capítulo dedicado a la creación de estructuras, uniones y macros, que son esenciales en el lenguaje ensamblador y la programación de sistemas. Las macros condicionales con operadores avanzados sirven para hacer las macros más profesionales.

Interfaz con lenguajes de alto nivel Hay un capítulo dedicado exclusivamente a la interconexión de lenguaje ensamblador con C y C++. Ésta es una importante habilidad de trabajo para los estudiantes que tengan una alta probabilidad de encontrar trabajos relacionados con la programación en lenguajes de alto nivel. Pueden aprender a optimizar su código y ver ejemplos de cómo los compiladores de C++ optimizan el código.

Ayudas para los instructores Todos los listados de los programas están disponibles en la Web. Hay un banco de exámenes disponible para los instructores, así como preguntas de repaso, soluciones a los ejercicios de programación y una presentación en diapositivas de Microsoft Power Point para cada capítulo (todo en inglés).

Resumen de los capítulos

Los capítulos 1 a 8 contienen los fundamentos básicos del lenguaje ensamblador, y deben cubrirse en secuencia. Después de eso, hay un buen grado de libertad. El siguiente gráfico de dependencia de los capítulos muestra cómo los últimos dependen del conocimiento obtenido en los anteriores. El capítulo 10 se dividió en dos partes para este gráfico, ya que ningún otro depende del conocimiento de cómo crear macros:



- 1. Conceptos básicos:** aplicaciones del lenguaje ensamblador, conceptos básicos, lenguaje máquina y representación de datos.
- 2. Arquitectura del procesador IA-32:** diseño básico de una microcomputadora, ciclo de ejecución de instrucciones, arquitectura del procesador IA-32, administración de memoria en los procesadores IA-32, componentes de una microcomputadora y el sistema de entrada-salida.

3. **Fundamentos de lenguaje ensamblador:** introducción al lenguaje ensamblador, enlace y depuración, definición de constantes y variables.
4. **Transferencias de datos, direccionamiento y aritmética:** transferencia simple de datos e instrucciones aritméticas, ciclo de ensamblado-enlace-ejecución, operadores, directivas, expresiones, instrucciones JMP y LOOP, y direccionamiento indirecto.
5. **Procedimientos:** enlace a una biblioteca externa, descripción de la biblioteca de vínculos del libro, operaciones de pila, definición y uso de procedimientos, diagramas de flujo y diseño estructurado descendente (top-down).
6. **Procesamiento condicional:** instrucciones booleanas y de comparación, saltos condicionales y ciclos, estructuras lógicas de alto nivel y máquinas de estado finito.
7. **Aritmética de enteros:** instrucciones de rotación y corrimiento con aplicaciones útiles, multiplicación y división, suma y resta extendidas, aritmética ASCII y con decimales empaquetados.
8. **Procedimientos avanzados:** pila de parámetros, variables locales, directivas avanzadas PROC e INVOKE, y recursividad.
9. **Cadenas y arreglos:** primitivas de cadena, manipulación de arreglos de caracteres y enteros, arreglos bidimensionales, ordenación y búsqueda.
10. **Estructuras y macros:** estructuras, macros, directivas de ensamblado condicional y definición de bloques de repetición.
11. **Programación en MS-Windows:** conceptos de administración de memoria en modo protegido, uso de la API de Microsoft Windows para mostrar texto y colores, y asignación de memoria dinámica.
12. **Interfaz de lenguajes de alto nivel:** convenciones para paso de parámetros, código de ensamblado en línea, y enlace de módulos en lenguaje ensamblador con programas en C y C++.
13. **Programación en MS-DOS de 16 bits:** llamadas a las interrupciones de MS-DOS para operaciones de entrada-salida de consola y archivos.
14. **Fundamentos de los discos:** sistemas de almacenamiento en disco, sectores, clústeres, directorios, tablas de asignación de archivos, manejo de los códigos de error de MS-DOS, manipulación de unidades y directorios.
15. **Programación a nivel de BIOS:** entrada desde el teclado, video con texto y gráficos, y programación del ratón.
16. **Programación experta en MS-DOS:** segmentos con diseño personalizado, estructura de un programa en tiempo de ejecución, y manejo de interrupciones. control del hardware mediante el uso de puertos de E/S.
17. **Procesamiento de punto flotante y codificación de instrucciones:** representación binaria de punto flotante y aritmética de punto flotante. Aprenda a programar la Unidad de punto flotante del procesador IA-32. Comprensión de la codificación de instrucciones de máquina del procesador IA-32.

Apéndice A: referencia de MASM.

Apéndice B: el conjunto de instrucciones IA-32.

Apéndice C: interrupciones del BIOS y de MS-DOS.

Apéndice D: respuestas a las preguntas de repaso.

Materiales de referencia

Sitio Web El autor mantiene un sitio Web activo en www.asmirvine.com.

Archivo de ayuda Archivo de ayuda (en formato de Ayuda de Windows) creado por Gerald Cahill del Antelope Valley College. Documenta las bibliotecas de vínculos del libro, así como las estructuras de datos Win32.

Libro de trabajo de lenguaje ensamblador En el sitio Web del libro encontrará un libro de trabajo interactivo que trata temas importantes, como conversiones numéricas, modos de direccionamiento, uso de registros, programación con el depurador y números binarios de punto flotante. Las páginas de contenido son documentos HTML, de manera que los estudiantes e instructores pueden agregar fácilmente su propio contenido personalizado.

Herramientas de depuración Tutoriales acerca del uso de Microsoft CodeView, Microsoft Visual Studio y Microsoft Windows Debugger (WinDbg).

Interrupciones de BIOS y MS-DOS El apéndice C contiene un breve listado de las funciones INT 10h (video), INT 16h (teclado), e INT 21h (MS-DOS) más utilizadas.

Conjunto de instrucciones El apéndice B presenta la mayoría de las instrucciones no privilegiadas para la familia de procesadores IA-32.

Para cada instrucción describimos su efecto, mostramos su sintaxis y cuáles banderas se ven afectadas.

Presentaciones en PowerPoint Un conjunto completo de presentaciones en Microsoft PowerPoint, escritas por el autor.

Agradecimientos

Queremos agradecer de manera especial a Tracy Dunkleberger, Editora en Jefe de Ciencias computacionales en Prentice Hall, que proporcionó una guía útil y amigable durante la escritura de esta quinta edición. Karen Ettinger hizo un magnífico trabajo como editora de producción, con un seguimiento constante a los numerosos pequeños detalles . Camille Trentacoste participó como gerente editorial del libro.

Quinta edición

Ofrezco mi agradecimiento especial a los siguientes profesores que impulsaron mi moral, me dieron estupendos consejos pedagógicos y examinaron minuciosamente todo el libro. Ellos han sido una enorme influencia para el desarrollo de este libro, en algunos casos en varias ediciones:

- **Gerald Cahill**, Antelope Valley College.
- **James Brink**, Pacific Lutheran University.
- **William Barrett**, San Jose State University.

Quiero agradecer también a **Scott Blackledge** y **John Taylor**, ambos programadores profesionales, que revisaron la mayor parte del manuscrito e identificaron numerosos errores. Varias personas revisaron capítulos individuales:

- Jerry Joyce, Keene State College.
- Tianzheng Wu, Mount Mercy College.
- Ron Davis, Kennedy-King College.
- David Topham, Ohlone College.
- Harvey Nice, DePaul University.

Cuarta edición

Las siguientes personas fueron de invaluable ayuda para crear la cuarta edición:

- Gerald Cahill, Antelope Valley College.
- James Brink, Pacific Lutheran University.
- Maria Kolatis, County College of Morris.
- Tom Joyce, Ingeniero en jefe de Premier Heart, LLC.
- Jeff Wothke, Purdue Calumet University.
- Tim Downey, Florida International University.

Los siguientes individuos fueron de invaluable ayuda en la corrección de la cuarta edición:

- Andres Altamirano, Miami.
- Courtney Amor, Los Angeles.
- Scott Blackledge, Platform Solutions, Inc.
- Ronald Davis, Kennedy-King College.
- Ata Elahi, Southern Connecticut State University.
- Jose Gonzalez, Miami.
- Leroy Highsmith, Southern Connecticut State University.

- Sajid Iqbal, Faran Institute of Technology.
- Charles Jones, Maryville College.
- Vincent Kayes, Mount St. Mary College.
- Eric Kobrin, Miami.
- Pablo Maurin, Miami.
- Barry Meaker, Ingeniero de diseño, Boeing Corporation.
- Ian Merkel, Miami.
- Sylvia Miner, Miami.
- M. Nawaz, OPSTEC College of Computer Science.
- Kam Ng, Universidad China de Hong Kong.
- Hien Nguyen, Miami.
- Ernie Philipp, Northern Virginia Community College.
- Boyd Stephens, UGMO Research, LLC.
- John Taylor, Inglaterra.
- Zachary Taylor, Columbia College.
- Virginia Welsh, Community College of Baltimore County.
- Robert Workman, Southern Connecticut State University.
- Tianzheng, Wu, Mount Mercy College.
- Matthew Zukoski, Universidad Lehigh.

CÓDIGOS DE EXPLORACIÓN DEL TECLADO

Los siguientes códigos de exploración del teclado pueden obtenerse al llamar a INT 16h, o a INT 21h para la entrada del teclado una segunda vez (la primera vez, la lectura del teclado devuelve 0). Todos los códigos están en hexadecimal:

TECLAS DE FUNCIÓN

Tecla	Normal	Con mayúsculas	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

CONCEPTOS bÁSICOS

- | | |
|---|---|
| <ul style="list-style-type: none">1.1 Bienvenido al lenguaje ensamblador<ul style="list-style-type: none">1.1.1 Preguntas importantes por hacer1.1.2 Aplicaciones del lenguaje ensamblador1.1.3 Repaso de sección1.2 Concepto de máquina virtual<ul style="list-style-type: none">1.2.1 Historia de los ensambladores de la PC1.2.2 Repaso de sección1.3 Representación de datos<ul style="list-style-type: none">1.3.1 Números binarios1.3.2 Suma binaria | <ul style="list-style-type: none">1.3.3 Tamaños de almacenamiento de enteros1.3.4 Enteros hexadecimales1.3.5 Enteros con signo1.3.6 Almacenamiento de caracteres1.3.7 Repaso de sección <ul style="list-style-type: none">1.4 Operaciones booleanas<ul style="list-style-type: none">1.4.1 Tablas de verdad para las funciones booleanas1.4.2 Repaso de sección1.5 Resumen del capítulo |
|---|---|

1.1 Bienvenido al lenguaje ensamblador

Lenguaje ensamblador para computadoras basadas en microprocesadores Intel se enfoca en la programación de microprocesadores compatibles con la familia de procesadores IA-32 de Intel, en la plataforma MS-Windows. Puede usar un procesador Intel o AMD de 32/64 bits para ejecutar todos los programas de este libro.

La familia IA-32 empezó con el Intel 80386 y continúa con el Pentium 4. Microsoft MASM (*Macro Assembler*) 8.0 es nuestro ensamblador preferido, el cual se ejecuta en MS-Windows. Hay otros ensambladores muy buenos para las computadoras basadas en Intel, incluyendo TASM (*Turbo Assembler*), NASM (*Netwide Assembler*), y el ensamblador de GNU. De todos ellos, TASM tiene la sintaxis más parecida a MASM, por lo que usted (con ayuda de su instructor) podría ensamblar y ejecutar la mayoría de los programas presentados en este libro. Los otros ensambladores (NASM y GNU) tienen una sintaxis un poco distinta.

El lenguaje ensamblador es el lenguaje de programación más antiguo y, de todos los lenguajes, es el que más se asemeja al lenguaje máquina nativo. Proporciona un acceso directo al hardware de la computadora, por lo que usted debe tener una buena comprensión acerca de la arquitectura y el sistema operativo de su computadora.

Valor educativo ¿Por qué leer este libro? Tal vez esté tomando un curso universitario con un nombre similar a alguno de los siguientes:

- Lenguaje ensamblador para microcomputadoras.
- Programación en lenguaje ensamblador.
- Introducción a la arquitectura computacional.
- Fundamentos de los sistemas computacionales.
- Programación de los sistemas embebidos (inrustados).

Éstos son nombres de los cursos en colegios y universidades que utilizan ediciones anteriores de este libro, el cual cubre los principios básicos acerca de la arquitectura computacional, el lenguaje máquina y la programación de bajo nivel. Aprenderá suficiente lenguaje ensamblador como para probar su conocimiento en la familia de los microprocesadores más utilizada en la actualidad. No aprenderá a programar una computadora “de juguete”, usando un ensamblador simulado; MASM es un ensamblador de nivel industrial, usado por profesionales con experiencia práctica. Conocerá la arquitectura de la familia de procesadores IA-32 de Intel desde el punto de vista del programador.

Si duda acerca del valor de la programación de bajo nivel y del estudio de los detalles acerca del software y hardware de computadora, preste atención a la siguiente cita de un científico computacional líder en la industria, Donald Knuth, al hablar sobre su famosa serie de libros: *El arte de programar computadoras (The Art of Computer Programming)*:

Algunas personas [dicen] que tener el lenguaje máquina en sí, fue el más grande error que cometí. En realidad no creo que se pueda escribir un libro para verdaderos programadores de computadoras, a menos que se pueda hablar sobre los detalles de bajo nivel.¹

Le recomiendo visitar el sitio Web de este libro, en donde encontrará una gran cantidad de información complementaria, tutoriales y ejercicios: www.asm Irvine.com.

1.1.1 Preguntas importantes por hacer

¿Qué conocimientos previos debo tener? Antes de leer este libro, debe haber completado un curso universitario de nivel básico sobre programación de computadoras. De esta forma, comprenderá mejor las instrucciones de programación de alto nivel tales como IF, ciclos y arreglos, al implementarlos en lenguaje ensamblador.

¿Qué son los ensambladores y los enlazadores? Un *ensamblador* es un programa utilitario que convierte el código fuente de los programas escritos en lenguajes ensamblador a lenguaje máquina. Un *enlazador* es un programa utilitario que combina los archivos individuales creados por un ensamblador, en un solo programa ejecutable. Hay una herramienta relacionada, llamada *depurador*, la cual le permite avanzar paso a paso a través de un programa mientras se ejecuta, para poder examinar los registros y la memoria.

¿Qué hardware y software necesito? Necesita una computadora con un procesador Intel386, Intel486, Pentium o compatible. Por ejemplo, los procesadores AMD funcionan muy bien con este libro. MASM (el ensamblador) es compatible con todas las versiones de 32 bits de Microsoft Windows, empezando con Windows 95. Algunos de los programas avanzados, relacionados con el acceso directo al hardware y la programación de los sectores de disco deben ejecutarse en MS-DOS, Windows 95/98/Me, debido a las estrictas restricciones de seguridad impuestas por Windows NT/2000/XP.

Además, necesitará lo siguiente:

- **Editor:** use un editor de texto o un editor para programadores, para crear los archivos de código fuente en lenguaje ensamblador.
- **Depurador de 32 bits:** en sentido estricto, no necesita un depurador, pero es muy conveniente tener uno. El depurador que se incluye con Visual C++ 2005 Express es excelente.

¿Qué tipos de programas podré crear? Este libro muestra cómo crear dos clases generales de programas:

- **Modo de direccionamiento real de 16 bits:** los programas en modo de direccionamiento real de 16 bits se ejecutan en MS-DOS y en la ventana de consola en MS-Windows. También se les conoce como programas en *modo real*, ya que utilizan un modelo segmentado de memoria, requerido en programas escritos para los procesadores Intel 8086 y 8088. Hay notas a lo largo del libro con tips acerca de cómo programar en modo de direccionamiento real, y se dedican dos capítulos exclusivamente a la programación de colores y gráficos en modo real.
- **Modo protegido de 32 bits:** los programas en modo protegido de 32 bits se ejecutan en todas las versiones de 32 bits de Microsoft Windows. Por lo general son más fáciles de escribir y de comprender que los programas en modo real.

¿Qué obtengo con este libro? Además de una buena cantidad de papel impreso podrá descargar Microsoft Assembler del sitio Web de Microsoft. En el sitio Web www.asm Irvine.com podrá consultar los detalles acerca de cómo obtener el ensamblador.

En el sitio Web del libro encontrará lo siguiente:

- **Archivo de ayuda en línea,** en donde se detallan los procedimientos de la biblioteca del libro y las estructuras esenciales de la API de Windows, por Gerald Cahill.
- **Libro de trabajo de lenguaje ensamblador,** una colección de tutoriales escritos por el autor.
- **Bibliotecas de vínculos Irvine32 e Irvine16,** para la programación en modo de direccionamiento real y modo protegido, con código fuente completo.
- **Programas de ejemplo,** con todo el código fuente del libro.
- **Correcciones** al libro y a los programas de ejemplo. ¡Esperamos que no sean demasiadas!
- **Tutoriales** acerca de cómo instalar el ensamblador.
- **Artículos** sobre temas avanzados que no se incluyeron en el libro impreso por falta de espacio.
- **Grupo de discusión,** que cuenta con más de 500 miembros.

¿Qué voy a aprender? Este libro le ofrece mucha información sobre la arquitectura computacional, la programación y las ciencias computacionales. He aquí lo que verá:

- Los principios básicos de la arquitectura computacional, aplicados en la familia de procesadores IA-32 de Intel.
- La lógica booleana básica y su aplicación en relación con la programación y el hardware de computadora.
- La manera en que los procesadores IA-32 administran la memoria, usando modo real, modo protegido y modo virtual.
- La manera en que los compiladores de lenguajes de alto nivel (tales como C++) traducen las instrucciones de su lenguaje a lenguaje ensamblador y código de máquina nativo.
- La manera en que los lenguajes de alto nivel implementan expresiones aritméticas, ciclos y estructuras lógicas a nivel de máquina.
- La representación de los datos, incluyendo enteros con y sin signo, números reales y datos tipo carácter.
- A depurar programas a nivel de máquina. La necesidad de esta habilidad es imprescindible cuando se trabaja en lenguajes tales como C y C++, los cuales proporcionan acceso a los datos y el hardware de bajo nivel.
- La manera en que los programas de aplicación se comunican con el sistema operativo de la computadora, a través de manejadores de interrupciones, llamadas al sistema y áreas comunes de memoria.
- A interconectar el código en lenguaje ensamblador con programas en C++.
- A crear programas de aplicación en lenguaje ensamblador.

¿Cómo se relaciona el lenguaje ensamblador con el lenguaje máquina? El lenguaje máquina es un lenguaje numérico que un procesador de computadora (la CPU) entiende de manera específica. Los procesadores compatibles con IA-32 entienden un lenguaje máquina común. El lenguaje ensamblador consiste en instrucciones escritas con nemónicos cortos, tales como ADD, MOV, SUB y CALL. El lenguaje ensamblador

tiene una relación de *uno a uno* con el lenguaje máquina: cada una de las instrucciones en lenguaje ensamblador corresponden a una sola instrucción en lenguaje máquina.

¿Cómo se relacionan C++ y Java con el lenguaje ensamblador? Los lenguajes de alto nivel, tales como C++ y Java, tienen una relación de *uno a varios* con el lenguaje ensamblador y el lenguaje máquina. Una sola instrucción en C++ se expande en varias instrucciones en lenguaje ensamblador o lenguaje máquina. Podemos mostrar cómo las instrucciones en C++ se expanden en código máquina. La mayoría de las personas no puede leer código de máquina puro, por lo que utilizaremos su parente más cercano, el lenguaje ensamblador. La siguiente instrucción en C++ lleva a cabo dos operaciones aritméticas y asigna el resultado a una variable. Suponga que X y Y son enteros:

```
int Y;
int X = (Y + 4) * 3;
```

A continuación se muestra la traducción de esta instrucción a lenguaje ensamblador. La traducción requiere varias instrucciones, ya que el lenguaje ensamblador funciona a un nivel detallado:

mov eax,Y	; mueve Y al registro EAX
add eax,4	; suma 4 al registro EAX
mov ebx,3	; mueve el 3 al registro EBX
imul ebx	; multiplica EAX por EBX
mov X,eax	; mueve EAX a X

(Los *registros* son ubicaciones de almacenamiento con nombre en la CPU, que almacenan los resultados intermedios de las operaciones).

El punto en este ejemplo no es afirmar que C++ es superior al lenguaje ensamblador o viceversa, sino mostrar su relación.

¿Nosotros? ¿Quiénes somos? A lo largo de este libro verá referencias constantes a *nosotros*. A menudo, los autores de libros de texto y artículos académicos utilizan *nosotros* como una referencia formal a ellos mismos. Esto se debe a que parece tan informal decir, “Ahora le mostraré cómo” hacer tal y tal cosa. Si le es de ayuda, piense en *nosotros* como una referencia al autor, sus revisores (quienes en realidad lo ayudaron de manera considerable), su editor (Prentice Hall), y sus estudiantes (miles de ellos).

¿Es portable el lenguaje ensamblador? Una importante distinción entre los lenguajes de alto nivel y el lenguaje ensamblador está relacionada con la portabilidad. Se dice que un lenguaje cuyos programas de código fuente pueden compilarse y ejecutarse en una amplia variedad de sistemas computacionales es *portable*. Por ejemplo, un programa en C++ puede compilarse y ejecutarse en casi cualquier computadora, a menos que haga referencias específicas a funciones de biblioteca que existan en un solo sistema operativo. Una importante característica del lenguaje Java es que los programas compilados se ejecutan en casi cualquier sistema computacional.

El lenguaje ensamblador no es portable, ya que está diseñado para una familia de procesadores específica. Hay una gran variedad de lenguajes ensambladores en uso actualmente, cada uno de los cuales está basado en una familia de procesadores. Algunas familias de procesadores reconocidas son: Motorola 68x00, Intel IA-32, SUN Sparc, Vax e IBM-370. Las instrucciones en lenguaje ensamblador pueden coincidir directamente con la arquitectura de la computadora, o pueden traducirse durante la ejecución mediante un programa dentro del procesador, al cual se le conoce como *intérprete de microcódigo*.

¿Por qué aprender lenguaje ensamblador? ¿Por qué no sólo leer un buen libro acerca del hardware y la arquitectura de las computadoras, y evitar aprender a programar en lenguaje ensamblador?

- Si estudia ingeniería computacional, es muy probable que le pidan que escriba programas *embebidos*. Éstos son programas cortos que se almacenan en una pequeña cantidad de memoria, en dispositivos de un solo propósito tales como los teléfonos, los sistemas del combustible y la ignición del automóvil, los sistemas de control de aire acondicionado, los sistemas de seguridad, los instrumentos para la adquisición de datos,

las tarjetas de video, las tarjetas de sonido, los discos duros, los módems y las impresoras. El lenguaje ensamblador es ideal para escribir programas embebidos, debido a que utilizan muy poca memoria.

- Las aplicaciones en tiempo real, tales como las simulaciones y el monitoreo de hardware, requieren precisión en la sincronización y en las respuestas. Los lenguajes de alto nivel no proporcionan a los programadores un control exacto sobre el código máquina generado por los compiladores. El lenguaje ensamblador nos permite especificar con precisión el código ejecutable de un programa.
- Las consolas de videojuegos requieren que su software esté altamente optimizado para que su tamaño de código sea pequeño y se ejecute con la mayor rapidez posible. Los programadores de videojuegos son expertos en la escritura de código que aproveche al máximo las características de hardware del sistema destino. Utilizan el lenguaje ensamblador como su herramienta preferida, ya que les permite un acceso directo al hardware de la computadora, y el código puede optimizarse en forma manual para obtener la máxima velocidad.
- El lenguaje ensamblador nos ayuda a obtener una comprensión general en cuanto a la interacción entre el hardware de computadora, los sistemas operativos y los programas de aplicación. Mediante el uso de lenguaje ensamblador, usted puede aplicar y probar la información teórica que recibe en los cursos de arquitectura computacional y sistemas operativos.
- En ocasiones, los programadores de aplicaciones encuentran que las limitaciones en los lenguajes de alto nivel les impiden realizar tareas de bajo nivel con eficiencia, como la manipulación a nivel de bits y el cifrado de datos. A menudo hacen llamadas a subrutinas escritas en lenguaje ensamblador para lograr su objetivo.
- Los fabricantes de hardware crean controladores de dispositivos para el equipo que venden. Los *controladores de dispositivos* son programas que traducen los comandos generales del sistema operativo en referencias específicas a los detalles relacionados con el hardware. Por ejemplo, los fabricantes de impresoras crean un controlador de dispositivo de MS-Windows distinto para cada modelo que venden. Lo mismo se aplica para los sistemas operativos Mac OS, Linux y otros.

¿Hay reglas en el lenguaje ensamblador? La mayoría de las reglas en el lenguaje ensamblador se basan en las limitaciones físicas del procesador de destino y su lenguaje máquina. Por ejemplo, la CPU requiere que los dos operandos de una instrucción sean del mismo tamaño. El lenguaje ensamblador tiene menos reglas que C++ o Java, ya que estos dos lenguajes de alto nivel utilizan reglas de sintaxis para reducir los errores de lógica involuntarios, a expensas del acceso a los datos de bajo nivel. Los programadores de lenguaje ensamblador pueden evadir con facilidad las restricciones características de los lenguajes de alto nivel. Por ejemplo, Java no permite el acceso a direcciones de memoria específicas. Para evadir esta restricción podemos hacer una llamada a una subrutina en C que utilice clases de la JNI (Interfaz nativa de Java), pero puede ser complicado mantener el programa resultante. Por otro lado, el lenguaje ensamblador puede acceder a cualquier dirección de memoria. El precio por dicha libertad es alto: ¡los programadores de lenguaje ensamblador invierten mucho tiempo en el proceso de la depuración!

1.1.2 Aplicaciones del lenguaje ensamblador

En los primeros días de la programación, la mayoría de las aplicaciones se escribían parcial o totalmente en lenguaje ensamblador. Tenían que ajustarse en una pequeña área de memoria y se ejecutaban de la manera más eficiente posible en los procesadores lentos. A medida que la memoria aumentó su capacidad y los procesadores su velocidad, los programas se hicieron más complejos. Los programadores cambiaron a lenguajes de alto nivel tales como C, FORTRAN y COBOL, los cuales contenían una cierta capacidad de estructuración. Los lenguajes de programación orientados a objetos más recientes, tales como C++, C# y Java, han hecho posible la escritura de programas complejos que contienen millones de líneas de código.

Es raro ver programas de aplicación extensos codificados por completo en lenguaje ensamblador, ya que se requeriría mucho tiempo para escribirlos y darles mantenimiento. En vez de ello, el lenguaje ensamblador se utiliza para optimizar ciertas secciones de los programas de aplicación en relación con la velocidad y para tener acceso al hardware de la computadora. La tabla 1-1 compara la capacidad de adaptación del lenguaje de ensamblador con los lenguajes de alto nivel, en relación con varios tipos de aplicaciones.

Tabla 1-1 Comparación entre el lenguaje ensamblador y los lenguajes de alto nivel.

Tipo de aplicación	Lenguajes de alto nivel	Lenguaje ensamblador
Software de aplicación comercial, escrito para una sola plataforma de tamaño mediano a grande	Las estructuras formales facilitan la organización y el mantenimiento de secciones extensas de código	Tiene una estructura formal mínima, por lo que los programadores con distintos niveles de experiencia deben imponer una. Esto produce dificultades en cuanto al mantenimiento del código existente
Controlador de dispositivo de hardware	Tal vez el lenguaje no proporciona un acceso directo al hardware. Aún si lo hace, pueden requerirse técnicas de codificación complicadas, lo cual dificulta el mantenimiento	El acceso al hardware es directo y simple. Es fácil de mantener cuando los programas son cortos y están bien documentados
Aplicación comercial escrita para varias plataformas (distintos sistemas operativos)	Por lo general es portable. El código fuente puede recompilarse en cada sistema operativo de destino, con mínimas modificaciones	Debe volver a codificarse por separado para cada plataforma, usando un ensamblador con una sintaxis diferente. Es difícil de mantener
Sistemas embebidos y juegos de computadora que requieren de un acceso directo a la memoria	Produce demasiado código ejecutable, y tal vez no se ejecute con eficiencia	Ideal, ya que el código ejecutable es pequeño y se ejecuta con rapidez

C++ tiene la cualidad única de ofrecer un compromiso entre la estructura de alto nivel y los detalles de bajo nivel. Es posible el acceso directo al hardware, pero no es nada portable. La mayoría de los compiladores de C++ tienen la habilidad de generar código fuente en lenguaje ensamblador, que el programador puede personalizar y refinar antes de ensamblarlo en código ejecutable.

1.1.3 Repaso de sección

1. ¿Cómo funcionan los ensambladores y los enlazadores en conjunto?
2. ¿De qué forma el estudio del lenguaje ensamblador puede mejorar su comprensión de los sistemas operativos?
3. ¿Qué significa una *relación de uno a varios*, cuando se compara un lenguaje de alto nivel con el lenguaje máquina?
4. Explique el concepto de *portabilidad*, empleado en los lenguajes de programación.
5. ¿El lenguaje ensamblador para la familia de procesadores Intel 80x86 es el mismo que para los sistemas computacionales tales como Vax o Motorola 68x00?
6. Dé un ejemplo de una aplicación de *sistemas embebidos*.
7. ¿Qué es un controlador de dispositivo?
8. ¿Cree que la comprobación de tipos en variables apuntador es más fuerte (estricta) en lenguaje ensamblador que en C++?
9. Mencione dos tipos de aplicaciones que se adaptan mejor al lenguaje ensamblador que a un lenguaje de alto nivel.
10. ¿Por qué un lenguaje de alto nivel no sería una herramienta ideal para escribir un programa que acceda en forma directa a una marca específica de impresora?
11. En general, ¿por qué no se utiliza el lenguaje ensamblador cuando se escriben programas de aplicación extensos?
12. *Reto:* traduzca la siguiente expresión en C++ a lenguaje ensamblador, usando como guía el ejemplo que presentamos en una sección anterior de este capítulo: $X = (Y * 4) + 3$.

1.2 Concepto de máquina virtual

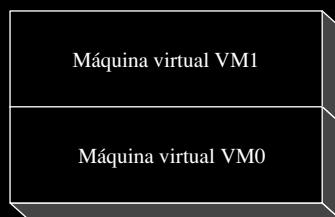
Una manera bastante efectiva de explicar la relación entre el hardware y el software es el *concepto de máquina virtual*. Nuestra explicación de este modelo se deriva del libro de Andrew Tanenbaum, *Structured Computer Organization*.² Para explicar este concepto, empecemos con la función más básica de una computadora: la ejecución de programas.

Por lo general, una computadora ejecuta programas escritos en su *lenguaje máquina* nativo. Cada instrucción en este lenguaje es lo bastante simple como para ejecutarse usando una cantidad relativamente pequeña de circuitos electrónicos. Por cuestión de simplificación, llamaremos a este lenguaje **L0**.

Los programadores tendrían muchas dificultades al escribir programas en L0, debido a que está en extremo detallado y consta sólo de números. Si se construyera un nuevo lenguaje **L1** que fuera más fácil de usar, los programas podrían escribirse en L1. Hay dos formas de lograr esto:

- *Interpretación*: a medida que se ejecutara el programa en L1, cada una de sus instrucciones podría decodificarse y ejecutarse mediante un programa escrito en lenguaje L0. El programa en L1 empezaría a ejecutarse de inmediato, pero cada instrucción tendría que decodificarse para poder ejecutarla.
- *Traducción*: todo el programa en L1 podría convertirse en un programa en L0, mediante un programa en L0 diseñado en específico para este fin. Después, el programa resultante en L0 podría ejecutarse directamente en el hardware de la computadora.

Máquinas Virtuales En vez de utilizar sólo lenguajes, es más fácil pensar en términos de una computadora hipotética, o *máquina virtual*, en cada nivel. La máquina virtual **VM1**, como la llamaremos, puede ejecutar comandos escritos en lenguaje L1. La máquina virtual **VM0** puede ejecutar comandos escritos en lenguaje L0:



Cada máquina virtual puede construirse ya sea con hardware o software. Las personas pueden escribir programas para la máquina virtual VM1 y, si es práctico implementar a VM1 como una computadora real, los programas pueden ejecutarse directamente en el hardware. O los programas escritos en VM1 pueden interpretarse/traducirse y ejecutarse en la máquina VM0.

La máquina VM1 no puede ser completamente distinta de VM0, ya que el proceso de traducción o interpretación consumiría demasiado tiempo. ¿Qué pasaría si el lenguaje que soporta la máquina VM1 no es lo suficiente amigable para el programador, como para poder usarlo en aplicaciones útiles? Entonces, podría diseñarse otra máquina virtual, VM2, que pudiera comprenderse con mayor facilidad. Este proceso puede repetirse hasta que pueda diseñarse una máquina virtual VM_n que soporte un lenguaje poderoso y fácil de usar.

El lenguaje de programación Java se basa en el concepto de máquina virtual. Un programa escrito en el lenguaje Java se traduce mediante un compilador de Java, que lo convierte en *código byte de Java*. Este código es un lenguaje de bajo nivel que se ejecuta con rapidez en tiempo de ejecución mediante un programa conocido como *máquina virtual de Java (JVM)*. La JVM se ha implementado en muchos sistemas computacionales distintos, por lo cual los programas en Java son relativamente independientes del sistema.

Máquinas específicas Vamos a relacionar estos conceptos con las computadoras y los lenguajes reales, usando nombres como **Nivel 1** para VM1 y **Nivel 0** para VM0, como se muestra en la figura 1-1. El hardware lógico digital de una computadora representa a la máquina Nivel 0, y el Nivel 1 se implementa a través de un intérprete conectado mediante cables al procesador, a lo que se le conoce como *microarquitectura*. Por encima de este nivel está el Nivel 2, que se conoce como *arquitectura del conjunto de instrucciones*. Éste es el primer nivel en el que los usuarios pueden, por lo general, escribir programas, aunque éstos consisten en números binarios.

Microarquitectura (Nivel 1) Por lo general, los fabricantes de chips de computadora no permiten que los usuarios promedio escriban microinstrucciones. Con frecuencia, los comandos de microarquitectura específicos son un secreto propietario. Podrían requerirse hasta tres o cuatro instrucciones en microcódigo para llevar a cabo una instrucción primitiva, tal como obtener un número de memoria e incrementarlo en 1.

FIGURA 1–1 Los niveles del 0 al 5 de una máquina virtual.



Arquitectura del conjunto de instrucciones (Nivel 2) Los fabricantes de chips de computadora diseñan en el procesador un *conjunto de instrucciones* para llevar a cabo las operaciones básicas, tales como mover, sumar o multiplicar. Este conjunto de instrucciones se conoce también como *lenguaje máquina convencional*, o simplemente *lenguaje máquina*. Cada instrucción en lenguaje máquina se ejecuta mediante varias microinstrucciones.

Sistema operativo (Nivel 3) Cuando las computadoras evolucionaron, se crearon máquinas virtuales adicionales, para que los programadores fueran más productivos. Una máquina de Nivel 3 comprende comandos interactivos que introducen los usuarios para cargar y ejecutar programas, mostrar directorios, etcétera. A esto se le conoce como el *sistema operativo de la computadora*. El software de sistema operativo se traduce en código máquina, el cual se ejecuta en una máquina de Nivel 2.³

Lenguaje ensamblador (Nivel 4) Por encima del sistema operativo, los lenguajes de programación proporcionan las capas de traducción para hacer que el desarrollo de software a gran escala sea práctico. El lenguaje ensamblador, que aparece en el Nivel 4, utiliza nemáticos cortos tales como ADD, SUB y MOV, los cuales se traducen fácilmente al nivel de arquitectura del conjunto de instrucciones (Nivel 2). Hay otras instrucciones en lenguaje ensamblador, como las llamadas a interrupciones, que el sistema operativo (Nivel 3) ejecuta de manera directa. Los programas en lenguaje ensamblador se traducen (*ensamblan*) en su totalidad a lenguaje máquina, antes de que empiecen a ejecutarse.

Lenguajes de alto nivel (Nivel 5) En el Nivel 5 están los lenguajes de programación de alto nivel tales como C++, C#, Visual Basic y Java. Los programas en estos lenguajes contienen poderosas instrucciones que se traducen en varias instrucciones del Nivel 4. En su interior, los compiladores traducen los programas de Nivel 5 en programas de Nivel 4, que a su vez se traducen en código de Nivel 4. Este código se ensambla en lenguaje máquina convencional.

La arquitectura del procesador Intel IA-32 soporta varias máquinas virtuales. Su modo de operación *virtual-86* emula la arquitectura del procesador Intel 8086/8088, utilizado en la Computadora Personal IBM original. El Pentium puede ejecutar varias instancias de la máquina virtual-86 al mismo tiempo, por lo que los programas independientes que se ejecutan en cada máquina virtual parecen tener el completo control de su equipo anfitrión.

1.2.1 Historia de los ensambladores de la PC

No hay un lenguaje ensamblador estándar oficial para los procesadores Intel. Lo que ha surgido a través de los años es un estándar *de facto*, establecido por el popular ensamblador MASM Versión 5 de Microsoft. Borland International se estableció a sí misma como una de las principales competidoras a principios de la década de 1990 con TASM (Turbo Assembler). TASM agregó muchas mejoras y produjo lo que se conoce como *Modo ideal*; además, Borland proporcionó también un *modo de compatibilidad con MASM*, que coincidía con la sintaxis de MASM Versión 5.

Microsoft lanzó a MASM 6.0 en 1992, la cual fue una actualización importante con muchas nuevas características. Desde entonces, Microsoft ha lanzado una variedad de actualizaciones para estar a la par con el cada vez más grande conjunto de instrucciones de la familia Pentium. La sintaxis del ensamblador MASM no ha sufrido cambios fundamentales desde la versión 6.0. Borland lanzó a TASM 5.0 de 32 bits en 1996, que coincide con la sintaxis actual de MASM. Hay otros ensambladores populares, todos los cuales varían con relación a la sintaxis de MASM, en mayor o menor grado. Algunos de ellos son: NASM (Netwide Assembler) para Windows y Linux; MASM32, un intérprete creado por encima de MASM; Asm86; y el ensamblador de GNU, distribuido por la Fundación de software libre.

1.2.2 Repaso de sección

1. Describa, en sus propias palabras, el concepto de *máquina virtual*.
2. ¿Por qué los programadores no escriben programas en lenguaje máquina?
3. (*Verdadero/Falso*): cuando se ejecuta un programa interpretado escrito en lenguaje L1, cada una de sus instrucciones se decodifica y se ejecuta mediante un programa escrito en lenguaje L0.
4. Explique la técnica de traducción al tratar con lenguajes en diferentes niveles de máquina virtual.
5. ¿Cómo es que la arquitectura del procesador Intel IA-32 contiene un ejemplo de una máquina virtual?
6. ¿Qué software permite que los programas compilados en Java se ejecuten en casi cualquier computadora?
7. Mencione (de menor a mayor) los seis niveles de máquina virtual que señalamos en esta sección.
8. ¿Por qué los programadores no escriben aplicaciones en microcódigo?
9. ¿En qué nivel de la máquina virtual, que se muestra en la figura 1-1, se utiliza el lenguaje máquina convencional?
10. ¿En qué otro(s) nivel(es) se traducen en instrucciones las instrucciones en el nivel de lenguaje ensamblador de una máquina virtual?

1.3 Representación de datos

Antes de hablar sobre la organización de una computadora y del lenguaje ensamblador, vamos a aclarar los conceptos de almacenamiento binario, hexadecimal, decimal y basado en caracteres. Los programadores en lenguaje ensamblador trabajan con los datos en el nivel físico, por lo que deben ser expertos a la hora de examinar la memoria y los registros. A menudo se utilizan los números binarios para describir el contenido de la memoria de la computadora; otras veces se utilizan números decimales y hexadecimales. Los programadores desarrollan cierta fluidez con los formatos numéricos, y pueden traducir con rapidez números de un formato a otro.

Cada formato (o sistema) de numeración tiene una *base*, o número máximo de símbolos que pueden asignarse a un solo dígito. La tabla 1-2 muestra los posibles dígitos para los sistemas de numeración que se utilizan con más frecuencia en la literatura computacional. En la última fila de la tabla, los números hexadecimales utilizan los dígitos del 0 al 9, y continúan con las letras de la A a la F para representar los valores decimales del 10 al 15. Es bastante común utilizar números hexadecimales cuando se muestra el contenido de la memoria de la computadora y las instrucciones a nivel de máquina.

Tabla 1-2 Los dígitos binarios, octales, decimales y hexadecimales.

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

1.3.1 Números binarios

Una computadora almacena instrucciones y datos en memoria, en forma de colecciones de cargas electrónicas. Para representar a estas entidades con números, se requiere un sistema que se adapte a los conceptos de *encendido y apagado*, o de *verdadero y falso*. Los *números binarios* son números en base 2, en donde cada dígito binario (llamado *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 de 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). En la siguiente figura se muestran los números de los bits MSB y LSB 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

Los enteros binarios pueden ser con o sin signo. Un entero con signo es positivo o negativo. Un entero sin signo es positivo, de manera predeterminada. El cero se considera positivo. Podemos representar a los números reales en binario mediante el uso de esquemas de codificación especiales pero dejaremos ese tema para un capítulo posterior. Por ahora, vamos a empezar con los enteros binarios sin signo.

Enteros binarios sin signo

Empezando con el bit menos significativo, cada bit en un entero binario sin signo representa una potencia incremental de 2. La siguiente figura contiene un número binario de 8 bits que muestra cómo se incrementan las potencias de dos, de derecha a izquierda:

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

La tabla 1-3 presenta los valores decimales desde 2^0 hasta 2^{15} .

Tabla 1-3 Valores de las posiciones de los bits binarios.

2^n	Valor decimal	2^n	Valor decimal
2^0	1	2^8	256
2^1	2	2^9	512
2^2	4	2^{10}	1024
2^3	8	2^{11}	2048
2^4	16	2^{12}	4096
2^5	32	2^{13}	8192
2^6	64	2^{14}	16384
2^7	128	2^{15}	32768

Traducción de enteros binarios sin signo a decimal

La notación posicional ponderada representa una manera conveniente para calcular el valor decimal de un entero binario sin signo que tiene n dígitos:

$$\text{dec} = (D_{n-1} \times 2^{n-1}) + (D_{n-2} \times 2^{n-2}) + \cdots + (D_1 \times 2^1) + (D_0 \times 2^0)$$

D indica un dígito binario. Por ejemplo, el número binario 00001001 es igual a 9. Para calcular este valor, eliminamos los términos iguales a cero:

$$(1 \times 2^3) + (1 \times 2^0) = 9$$

La siguiente figura muestra el mismo cálculo:

$$\begin{array}{r}
 & & 8 \\
 & & + 1 \\
 \hline
 & & 9
 \end{array}$$

0 0 0 0 1 0 0 1

Traducción de enteros decimales sin signo a binario

Para traducir un entero decimal sin signo a binario, divida en forma repetida el entero entre 2 y guarde cada residuo como un dígito binario. La siguiente tabla muestra los pasos requeridos para traducir el 37 decimal a binario. El resto de los dígitos, empezando desde la fila superior, son los dígitos binarios D_0 , D_1 , D_2 , D_3 , D_4 y D_5 :

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

Al recolectar en orden inverso los dígitos binarios en la columna de los residuos se produce el número binario 100101. Como el almacenamiento en las computadoras Intel siempre consiste en números binarios cuyas longitudes sean múltiplos de 8, rellenamos las posiciones de los dos dígitos a la izquierda con ceros, lo cual produce el número binario 00100101.

1.3.2 Suma binaria

Al sumar dos enteros binarios hay que proceder bit por bit, empezando con el par de bits de menor orden (a la derecha) y luego se suma cada par subsiguiente de bits. Hay cuatro maneras de sumar dos dígitos binarios, como se muestra aquí:

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

Al sumar 1 y 1, el resultado es un 10 binario (considérelo como el valor decimal 2). El dígito adicional genera un acarreo hacia la posición del siguiente bit más alto. En la siguiente figura, sumamos los números binarios 00000100 y 00000111:

$$\begin{array}{r}
 & \text{Acarreo: 1} \\
 & \boxed{ } \quad (4) \\
 + & \boxed{ } \quad (7) \\
 \hline
 & \boxed{ } \quad (11)
 \end{array}$$

Posición del bit: 7 6 5 4 3 2 1 0

Empezando con el menor bit en cada número (posición de bit 0), sumamos $0 + 1$, lo cual produce un 1 en la fila inferior. Lo mismo ocurre en el siguiente bit más alto (posición 1). En la posición de bit 2, sumamos $1 + 1$, lo cual genera una suma de cero y un acarreo de 1. En la posición de bit 3, sumamos el bit de acarreo a $0 + 0$, lo cual produce un 1. El resto de los bits son ceros. Para verificar la suma, realice la suma de los equivalentes decimales que se muestran en la parte derecha de la figura ($4 + 7 = 11$).

1.3.3 Tamaños 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), y *palabra cuádruple* (8 bytes). En la siguiente figura se muestra el número de bits para cada tamaño:



La tabla 1-4 muestra el rango de posibles valores para cada tipo de entero sin signo.

Tabla 1-4 Rangos de enteros sin signo.

Tipo de almacenamiento	Rango (Bajo-Alto)	Potencias de 2
Byte sin signo	0 a 255	0 a $(2^8 - 1)$
Palabra sin signo	0 a 65,535	0 a $(2^{16} - 1)$
Doble palabra sin signo	0 a 4,294,967,295	0 a $(2^{32} - 1)$
Palabra cuádruple sin signo	0 a 18,446,744,073,709,551,615	0 a $(2^{64} - 1)$

Mediciones grandes Al hacer referencia a la memoria y al espacio en disco, se utilizan varias medidas grandes:⁴

- Un *kilobyte* es igual a 2^{10} , o 1024 bytes.
- Un *megabyte* (MB) es igual a 2^{20} , o 1,048,576 bytes.
- Un *gigabyte* (GB) es igual a 2^{30} , o $1,024^3$, o 1,073,741,824 bytes.

- Un *terabyte* (TB) es igual a 2^{40} , o 1,099,511,627,776 bytes.
- Un *petabyte* es igual a 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 bytes.
- Un *zettabyte* es igual a 2^{70} bytes.
- Un *yottabyte* es igual a 2^{80} bytes.

1.3.4 Enteros hexadecimales

Los números binarios grandes son difíciles de leer, por lo que los dígitos hexadecimales ofrecen una manera conveniente de representar los datos binarios. Cada dígito en un entero hexadecimal representa a cuatro dígitos binarios, y dos dígitos hexadecimales juntos representan a un byte. Un solo dígito hexadecimal representa un número decimal del 0 al 15, por lo que las letras A a la F representan los valores decimales en el rango del 10 al 15. La tabla 1-5 muestra cómo cada secuencia de cuatro bits binarios se traduce a un valor decimal o hexadecimal.

Tabla 1-5 Equivalentes en binario, decimal y hexadecimal.

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

El siguiente ejemplo muestra cómo el número binario 000101101010011110010100 es equivalente al número hexadecimal 16A794:

1	6	A	7	9	4
0001	0110	1010	0111	1001	0100

Conversión de hexadecimal sin signo a decimal

En hexadecimal, la posición de cada dígito representa a una potencia de 16. Esto es útil cuando se calcula el valor decimal de un entero hexadecimal. Suponga que numeramos con subíndices los dígitos en un entero hexadecimal de cuatro dígitos: $D_3D_2D_1D_0$. La siguiente fórmula calcula el valor decimal del número:

$$\text{dec} = (D_3 \times 16^3) + (D_2 \times 16^2) + (D_1 \times 16^1) + (D_0 \times 16^0)$$

Esta fórmula puede generalizarse para cualquier número hexadecimal de n dígitos:

$$\text{dec} = (D_{n-1} \times 16^{n-1}) + (D_{n-2} \times 16^{n-2}) + \cdots + (D_1 \times 16^1) + (D_0 \times 16^0)$$

Por ejemplo, el número hexadecimal 1234 es igual a $(1 \times 16^3) + (2 \times 16^2) + (3 \times 16^1) + (4 \times 16^0)$, que viene siendo el número decimal 4660. De manera similar, el número hexadecimal 3BA4 es igual a $(3 \times 16^3) + (11 \times 16^2) + (10 \times 16^1) + (4 \times 16^0)$, que viene siendo el número decimal 15,268. La siguiente figura muestra este último cálculo:

$$\begin{array}{r}
 & 3 * 16^3 = 12,288 \\
 & 11 * 16^2 = 2,816 \\
 & 10 * 16^1 = 160 \\
 & 4 * 16^0 = + 4 \\
 \hline
 & \text{Total: } 15,268
 \end{array}$$

3 B A 4

La tabla 1-6 presenta las potencias de 16, de 16^0 hasta 16^7 .

Tabla 1-6 Potencias de 16 en decimal.

16^n	Valor decimal	16^n	Valor decimal
16^0	1	16^4	65,536
16^1	16	16^5	1,048,576
16^2	256	16^6	16,777,216
16^3	4096	16^7	268,435,456

Conversión de decimal sin signo a hexadecimal

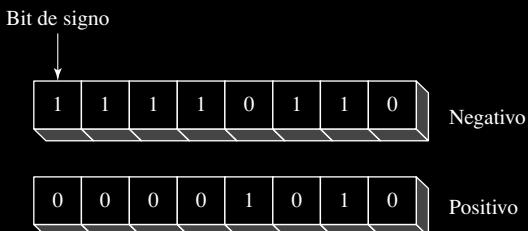
Para convertir un entero decimal sin signo a hexadecimal, divida repetidas veces el valor decimal entre 16 y conserve el residuo de cada división como un dígito hexadecimal. Por ejemplo, la siguiente tabla presenta los pasos para convertir el número 422 decimal a hexadecimal:

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

Si recolectamos los dígitos de la columna del residuo en orden inverso, la representación hexadecimal es **1A6**. En la sección 1.3.1 utilizamos el mismo algoritmo para los números binarios.

1.3.5 Enteros con signo

Los enteros binarios con signo son positivos o negativos. En las computadoras basadas en Intel, el bit más significativo (MSB) indica el signo: 0 es positivo y 1 es negativo. La siguiente figura muestra ejemplos de enteros negativos y positivos de 8 bits:



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 suma un número a su *inverso aditivo*, el resultado de la suma 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 sólo necesita convertirla en una expresión de suma: $A + (-B)$.

Para formar el complemento a dos de un entero binario, se invierten (complementan) sus bits y se le suma 1. Por ejemplo, si utilizamos el valor binario de 8 bits 00000001, su complemento a dos resulta ser 11111111, como puede verse a continuación:

Valor inicial	00000001
Paso 1: invertir los bits	11111110
Paso 2: sumar 1 al valor del paso 1	$ \begin{array}{r} 11111110 \\ +00000001 \\ \hline 11111111 \end{array} $
Suma: representación en complemento a dos	11111111

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

Complemento a dos de hexadecimal Para formar el complemento a dos de un entero hexadecimal, invierta todos los bits y sume 1. Una manera sencilla de invertir los bits de un dígito hexadecimal es restar 15 al dígito. He aquí varios ejemplos de enteros hexadecimales convertidos a sus complementos a dos:

6A3D \rightarrow 95C2 + 1 \rightarrow 95C3
 95C3 \rightarrow 6A3C + 1 \rightarrow 6A3D
 21F0 \rightarrow DE0F + 1 \rightarrow DE10
 DE10 \rightarrow 21EF + 1 \rightarrow 21F0

Conversión de binario con signo a decimal Para calcular el equivalente decimal de un entero binario con signo, realice una de las siguientes opciones:

- Si el bit más alto es un 1, el número está almacenado en notación de complemento a dos. Forme su complemento a dos una segunda vez para obtener su equivalente positivo. Después convierta este nuevo número en decimal, como si fuera un entero binario sin signo.
- Si el bit más alto es un 0, puede convertirlo en decimal como si fuera un entero binario sin signo.

Por ejemplo, el número binario con signo 11110000 tiene un 1 en el bit más alto, lo cual indica que es un entero negativo. Primero hay que formar su complemento a dos, y después el resultado se convierte en decimal. He aquí los pasos en el proceso:

Valor inicial	11110000
Paso 1: invertir los bits	00001111
Paso 2: sumar 1 al valor del paso 1	$ \begin{array}{r} 00001111 \\ + 1 \\ \hline 00010000 \end{array} $
Paso 3: formar el complemento a dos	00010000
Paso 4: convertir en decimal	16

Como el entero original (11110000) era negativo, inferimos que su valor decimal era de -16 .

Conversión de entero decimal a binario Para determinar la representación binaria de un entero decimal con signo, haga lo siguiente:

1. Convierta el valor absoluto del entero decimal a binario.
2. Si el entero decimal original era negativo, forme el complemento a dos del número binario del paso anterior.

Por ejemplo, el número decimal -43 se traduce a binario de la siguiente manera:

1. La representación binaria del 43 sin signo es 00101011 .
2. Como el valor original era negativo, formamos el complemento a dos de 00101011 , que es 11010101 . Ésta es la representación del número -43 decimal.

Conversión de decimal con signo a hexadecimal Para convertir un entero decimal con signo en hexadecimal, haga lo siguiente:

1. Convierta el valor absoluto del entero decimal a hexadecimal.
2. Si el entero decimal era negativo, forme el complemento a dos del número hexadecimal del paso anterior.

Conversión de hexadecimal con signo a decimal Para convertir un entero hexadecimal con signo a decimal, haga lo siguiente:

1. Si el entero hexadecimal es negativo, forme su complemento a dos; en caso contrario, retenga el entero como está.
2. Use el entero del paso anterior y conviértalo a decimal. Si el valor original era negativo, agregue un signo de menos al principio del entero decimal.

Para saber si un entero hexadecimal es positivo o negativo, inspeccionamos su dígito más significativo (más alto). Si el dígito es ≥ 8 , el número es negativo; si el dígito es ≤ 7 , el número es positivo. Por ejemplo, el número hexadecimal $8A20$ es negativo y el $7FD9$ es positivo.

Valores máximo y mínimo

Un entero con signo de n bits sólo utiliza $n - 1$ bits para representar su magnitud. La tabla 1-7 muestra los valores mínimos y máximos para los bytes, palabras, dobles palabras y palabras cuádruples con signo.

Tabla 1-7 Tamaños y rangos de almacenamiento de los enteros con signo.

Tipo de almacenamiento	Rango (Bajo-Alto)	Potencias de 2
Byte con signo	$-128 \text{ a } +127$	$-2^7 \text{ a } (2^7 - 1)$
Palabra con signo	$-32,768 \text{ a } +32,767$	$-2^{15} \text{ a } (2^{15} - 1)$
Doble palabra con signo	$-2,147,483,648 \text{ a } 2,147,483,647$	$-2^{31} \text{ a } (2^{31} - 1)$
Palabra cuádruple con signo	$-9,223,372,036,854,775,808 \text{ a } +9,223,372,036,854,775,807$	$-2^{63} \text{ a } (2^{63} - 1)$

1.3.6 Almacenamiento de caracteres

Si las computadoras sólo pueden almacenar datos binarios, ¿cómo representan los caracteres? Se requiere un *conjunto de caracteres*, una asignación de caracteres a enteros. Hasta hace unos cuantos años, los conjuntos de caracteres utilizaban sólo 8 bits. Incluso ahora, cuando las microcomputadoras IBM compatibles se ejecutan en modo de caracteres (como el MS-DOS), utilizan el conjunto de caracteres ASCII (se pronuncia como “aski”). ASCII es el acrónimo de *Código estándar estadounidense para el intercambio de información*. En ASCII, se asigna un entero único de 7 bits a cada carácter. Como los códigos ASCII utilizan sólo los 7 bits menores de cada byte, el bit adicional se utiliza en varias computadoras para crear un conjunto de caracteres propietario. Por ejemplo, en las microcomputadoras IBM compatibles, los valores del 128 al 255 representan símbolos gráficos y caracteres griegos.

Conjunto de caracteres ANSI El Instituto Nacional Estadounidense de Estándares (ANSI) define un conjunto de caracteres de 8 bits que se utiliza para representar hasta 256 caracteres. Los primeros 128 caracteres corresponden a las letras y símbolos en un teclado estadounidense estándar. Los otros 128 caracteres representan caracteres especiales, como las letras en los alfabetos internacionales, los acentos, los símbolos de moneda y las fracciones. MS-Windows Me, 98 y 95 utilizan el conjunto de caracteres ANSI. Para incrementar el número de caracteres disponibles, MS-Windows alterna entre las tablas de caracteres, conocidas como *páginas de códigos*.

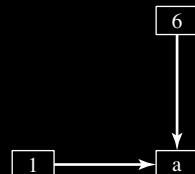
Estándar Unicode Desde hace tiempo existe la necesidad de representar una amplia variedad de lenguajes internacionales en el software de computadora y para evitar el atestamiento de los diversos esquemas de codificación existentes. Como resultado se creó el estándar *Unicode*, como una forma universal de definir caracteres y símbolos. Este estándar define códigos para caracteres, símbolos y signos de puntuación utilizados en todos los principales lenguajes, así como las escrituras alfábéticas europeas, las escrituras de derecha a izquierda del Medio Oriente y muchas escrituras de Asia.⁵ Hay tres formatos de codificación en Unicode, de manera que los datos pueden transmitirse en formatos de byte, palabra o doble palabra:

- **UTF-8:** se utiliza en HTML y tiene los mismos valores de bytes que ASCII (Código estándar estadounidense para el intercambio de información). Puede incorporarse en un sistema de codificación de longitud variable para todos los caracteres Unicode.
- **UTF-16:** se utiliza en entornos que balancean el acceso eficiente a los caracteres con el uso económico del almacenamiento. Por ejemplo, Windows NT, 2000 y XP utilizan la codificación UTF-16. Cada carácter se codifica en 16 bits.
- **UTF-32:** se utiliza en entornos en los que el espacio no es una limitación, y se requieren caracteres de anchura fija. Cada carácter se codifica en 32 bits.

Podemos copiar un valor Unicode más pequeño (byte, por ejemplo) en uno más grande (palabra o doble palabra) sin que haya pérdida de datos.

Cadenas ASCII A una secuencia de uno o más caracteres se le conoce como *cadena*. En forma más específica, una *cadena ASCII* se almacena en memoria como una sucesión de bytes que contienen códigos ASCII. Por ejemplo, los códigos numéricos para la cadena “ABC123” son 41h, 42h, 43h, 31h, 32h y 33h. Una cadena con *terminación nula* es una cadena de caracteres que va seguida de un solo byte que contiene un cero. Los lenguajes C y C++ utilizan cadenas con terminación nula, y muchas funciones de MS-DOS y MS-Windows requieren que las cadenas se encuentren en este formato.

Uso de la tabla ASCII Hay una tabla en la contraportada interior de este libro en la que se presentan los códigos ASCII utilizados cuando se ejecutan programas en modo de MS-DOS. Para encontrar el código ASCII hexadecimal de un carácter, busque a lo largo de la fila superior de la tabla y encuentre la columna que contiene el carácter que desea traducir. El dígito más significativo del valor hexadecimal está en la segunda fila, en la parte superior de la tabla; el dígito menos significativo está en la segunda columna a partir de la izquierda. Por ejemplo, para encontrar el código ASCII de la letra **a**, encuentre la columna que contiene la **a** y busque en la segunda fila: el primer dígito hexadecimal es 6. A continuación, busque a la izquierda, a lo largo de la fila que contiene la **a**, y observe que la segunda columna contiene el dígito 1. Por lo tanto, el código ASCII de **a** es el número hexadecimal 61. Esto se muestra a continuación en forma simplificada:



Caracteres de control ASCII Los códigos de caracteres en el rango del 0 al 31 se conocen como *caracteres de control ASCII*. Si un programa escribe estos códigos en la salida estándar (como en C++), los caracteres de control llevarán a cabo acciones predefinidas. La tabla 1-8 presenta los caracteres en este rango que se utilizan con más frecuencia.

Tabla 1-8 Caracteres de control ASCII.

Código ASCII (decimal)	Descripción
8	Retroceso (avanza una columna a la izquierda)
9	Tabulación horizontal (avanza n columnas hacia delante)
10	Avance de línea (avanza a la siguiente línea de salida)
12	Avance de página (avanza a la siguiente página de impresora)
13	Retorno de carro (avanza a la columna de salida de más a la izquierda)
27	Carácter de escape

Terminología para la representación numérica de datos Es importante utilizar una terminología precisa al describir la manera en que los números y los caracteres se representan en memoria y en la pantalla. Por ejemplo, el número decimal 65 se almacena en memoria como un solo byte binario, representado por 01000001. Probablemente un programa depurador muestre el byte como “41”, que es la representación hexadecimal del número. Si el byte se copiara a la memoria de video, aparecería la letra “A” en la pantalla. ¿Por qué? Debido a que el número 01000001 es el código ASCII para la letra A. Como la interpretación de un número puede depender del contexto en el que aparece, asignamos un nombre específico a cada tipo de representación de datos, para aclarar las futuras confusiones:

- Un *entero binario* es un entero almacenado en memoria en su formato puro, listo para usarse en un cálculo. Los enteros binarios se almacenan en múltiplos de 8 bits (8, 16, 32, 48 o 64).
- Una *cadena de dígitos ASCII* es una cadena de caracteres ASCII, tal como “123” o “65”, a la cual se le da una apariencia de número. Ésta es una simple representación del número y puede estar en cualquiera de los formatos que se muestran para el número decimal 65 en la tabla 1-9:

Tabla 1-9 Tipos de cadenas numéricas.

Formato	Valor
ASCII binario	“01000001”
ASCII decimal	“65”
ASCII hexadecimal	“41”
ASCII octal	“101”

1.3.7 Repaso de sección

1. Explique el término LSB.
2. Explique el término MSB.
3. ¿Cuál es la representación decimal de cada uno de los siguientes enteros binarios sin signo?
 - a. 11111000
 - b. 11001010
 - c. 11110000
4. ¿Cuál es la representación decimal de cada uno de los siguientes enteros binarios sin signo?
 - a. 00110101
 - b. 10010110
 - c. 11001100
5. ¿Cuál es la suma de cada par de enteros binarios?
 - a. 00001111 + 00000010

- b. $11010101 + 01101011$
 - c. $00001111 + 00001111$
6. ¿Cuál es la suma de cada par de enteros binarios?
- a. $10101111 + 11011011$
 - b. $10010111 + 11111111$
 - c. $01110101 + 10101100$
7. ¿Cuántos bytes hay en cada uno de los siguientes tipos de datos?
- a. palabra
 - b. doble palabra
 - c. palabra cuádruple
8. ¿Cuántos bits hay en cada uno de los siguientes tipos de datos?
- a. palabra
 - b. doble palabra
 - c. palabra cuádruple
9. ¿Cuál es el número mínimo de bits binarios necesarios para representar a cada uno de los siguientes enteros decimales sin signo?
- a. 65
 - b. 256
 - c. 32768
10. ¿Cuál es el número mínimo de bits binarios necesarios para representar a cada uno de los siguientes enteros decimales sin signo?
- a. 4095
 - b. 65534
 - c. 2134657
11. ¿Cuál es la representación hexadecimal de cada uno de los siguientes números binarios?
- a. 1100 1111 0101 0111
 - b. 0101 1100 1010 1101
 - c. 1111 0011 1101 1011
12. ¿Cuál es la representación hexadecimal de cada uno de los siguientes números binarios?
- a. 0011 0101 1101 1010
 - b. 1100 1110 1010 0011
 - c. 1111 1110 1101 1011
13. ¿Cuál es la representación binaria de los siguientes números hexadecimales?
- a. E5B6AED7
 - b. B697C7A1
 - c. 234B6D92
14. ¿Cuál es la representación binaria de los siguientes números hexadecimales?
- a. 0126F9D4
 - b. 6ACDFA95
 - c. F69BDC2A
15. ¿Cuál es la representación decimal sin signo de cada entero hexadecimal?
- a. 3A
 - b. 1BF
 - c. 4096
16. ¿Cuál es la representación decimal sin signo de cada entero hexadecimal?
- a. 62
 - b. 1C9
 - c. 6A5B

17. ¿Cuál es la representación hexadecimal de 16 bits de cada entero decimal con signo?
 - a. -26
 - b. -452
18. ¿Cuál es la representación hexadecimal de 16 bits de cada entero decimal con signo?
 - a. -32
 - b. -62
19. Los siguientes números hexadecimales de 16 bits representan enteros con signo. Conviértalos a números decimales.
 - a. 7CAB
 - b. C123
20. Los siguientes números hexadecimales de 16 bits representan enteros con signo. Conviértalos a números decimales.
 - a. 7F9B
 - b. 8230
21. ¿Cuál es la representación decimal de los siguientes números binarios con signo?
 - a. 10110101
 - b. 00101010
 - c. 11110000
22. ¿Cuál es la representación decimal de los siguientes números binarios con signo?
 - a. 10000000
 - b. 11001100
 - c. 10110111
23. ¿Cuál es la representación binaria de 8 bits (complemento a dos) de cada uno de los siguientes enteros decimales con signo?
 - a. -5
 - b. -36
 - c. -16
24. ¿Cuál es la representación binaria de 8 bits (complemento a dos) de cada uno de los siguientes enteros decimales con signo?
 - a. -72
 - b. -98
 - c. -26
25. ¿Cuáles son las representaciones hexadecimal y decimal del carácter ASCII letra X mayúscula?
26. ¿Cuáles son las representaciones hexadecimal y decimal del carácter ASCII letra M mayúscula?
27. ¿Por qué se inventó Unicode?
28. *Reto:* ¿cuál es el valor más grande que se puede representar, utilizando un entero *sin signo* de 256 bits?
29. *Reto:* ¿cuál es el valor positivo más grande que se puede representar, utilizando un entero *con signo* de 256 bits?

1.4 Operaciones booleanas

El *álgebra booleana* define un conjunto de operaciones con los valores **true** (verdadero) y **false** (falso). Su inventor fue George Boole, un matemático de mediados del siglo diecinueve, quien diseñó el primer modelo de una computadora (llamada la *Máquina analítica*⁶). Cuando se inventaron las primeras computadoras digitales, era evidente que podía utilizarse el álgebra de Boole para describir el diseño de los circuitos digitales. Al mismo tiempo, las expresiones booleanas se utilizan en la programación para expresar operaciones lógicas.

Expresión booleana Una expresión booleana involucra a un operador booleano y uno o más operandos. Cada expresión booleana implica un valor de verdadero o falso. El conjunto de operadores incluye:

- NOT: se escribe como $\neg o \sim o'$
- AND: se escribe como $\wedge o \bullet$
- OR: se escribe como $\vee o +$

El operador NOT es unario, y los otros operadores son binarios. Los operandos de una expresión booleana también pueden ser expresiones booleanas. A continuación se muestran algunos ejemplos:

Expresión	Descripción
$\neg X$	NOT X
$X \wedge Y$	X AND Y
$X \vee Y$	X OR Y
$\neg X \vee Y$	(NOT X) OR Y
$\neg(X \wedge Y)$	NOT (X AND Y)
$X \wedge \neg Y$	X AND (NOT Y)

NOT La operación NOT invierte un valor booleano. Puede escribirse en notación matemática como $\neg X$, en donde X es una variable (o expresión) que contiene un valor de verdadero (V) o falso (F). La siguiente tabla de verdad muestra todos los posibles resultados de NOT, usando una variable X. Las entradas están en el lado izquierdo y las salidas (sombreadas) en el lado derecho:

X	$\neg X$
F	V
V	F

Una tabla de verdad puede usar 0 para falso y 1 para verdadero.

AND La operación AND booleana requiere dos operandos, y puede expresarse mediante la notación $X \wedge Y$. La siguiente tabla de verdad muestra todos los posibles resultados (sombreados) para los valores de X y Y:

X	Y	$X \wedge Y$
F	F	F
F	V	F
V	F	F
V	V	V

El resultado es verdadero sólo cuando el valor de ambas entradas es verdadero. Esto corresponde al AND lógico que se utiliza en expresiones booleanas compuestas en C++ y Java.

OR La operación OR booleana requiere dos operandos, y a menudo se expresa mediante la notación $X \vee Y$. La siguiente tabla de verdad muestra todos los posibles resultados (sombreados) para los valores de X y Y:

X	Y	$X \vee Y$
F	F	F
F	V	V
V	F	V
V	V	V

El resultado es falso sólo cuando el valor de ambas entradas es falso. Esta tabla de verdad corresponde al OR lógico que se utiliza en expresiones booleanas compuestas en C++ y Java.

Precedencia de los operadores En una expresión booleana que involucra a más de un operador, la precedencia es importante. Como se muestra en la siguiente tabla, el operador NOT tiene la precedencia más alta, seguido de AND y OR. Para evitar la ambigüedad, utilice paréntesis para forzar la evaluación inicial de una expresión:

Expresión	Orden de las operaciones
$\neg X \vee Y$	NOT, después OR
$\neg(X \vee Y)$	OR, después NOT
$X \vee(Y \wedge Z)$	AND, después OR

1.4.1 Tablas de verdad para las funciones booleanas

Una función booleana recibe entradas booleanas y produce un resultado booleano. Podemos construir una tabla booleana para cualquier función booleana, mostrando todas las posibles entradas y salidas. A continuación se muestran tablas booleanas que representan funciones booleanas con dos entradas llamadas X y Y. La columna sombreada a la derecha es la salida de la función:

Ejemplo 1: $\neg X \vee Y$

X	$\neg X$	Y	$\neg X \vee Y$
F	V	F	V
F	V	V	V
V	F	F	F
V	F	V	V

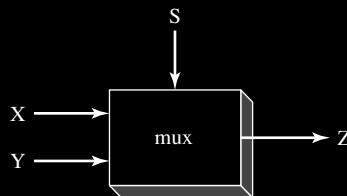
Ejemplo 2: $X \wedge \neg Y$

X	Y	$\neg Y$	$X \wedge \neg Y$
F	F	V	F
F	V	F	F
V	F	V	V
V	V	F	F

Ejemplo 3: $(Y \wedge S) \vee (X \wedge \neg S)$

X	Y	S	$Y \wedge S$	$\neg S$	$X \wedge \neg S$	$(Y \wedge S) \vee (X \wedge \neg S)$
F	F	F	F	V	F	F
F	V	F	F	V	F	F
V	F	F	F	V	V	V
V	V	F	F	V	V	V
F	F	V	F	F	F	F
F	V	V	V	F	F	V
V	F	V	F	F	F	F
V	V	V	V	F	F	V

Esta función booleana describe a un *multiplexor*, un componente digital que utiliza un bit selector (S) para seleccionar una de dos salidas (X o Y). Si S = falso, la salida de la función (Z) es igual que X. Si S = verdadero, la salida de la función es igual que Y. He aquí un diagrama de bloques de un multiplexor:

**1.4.2 Repaso de sección**

1. Describa la siguiente expresión booleana: $\neg X \vee Y$.
2. Describa la siguiente expresión booleana: $(X \wedge Y)$.
3. ¿Cuál es el valor de la expresión booleana $(V \wedge F) \vee V$?
4. ¿Cuál es el valor de la expresión booleana $\neg(F \vee V)$?
5. ¿Cuál es el valor de la expresión booleana $\neg F \wedge \neg V$?
6. Cree una tabla de verdad para mostrar todas las posibles entradas y salidas para la función booleana descrita por $\neg(A \vee B)$.
7. Cree una tabla de verdad para mostrar todas las posibles entradas y salidas para la función booleana descrita por $(\neg A \wedge \neg B)$.
8. *Reto:* si una función booleana tiene cuatro entradas, ¿cuántas filas se requieren para su tabla de verdad?
9. *Reto:* ¿cuántos bits selectores se requieren para un multiplexor de cuatro entradas?

1.5 Resumen del capítulo

Este libro se enfoca en la programación de los microprocesadores compatibles con la familia de procesadores IA-32 de Intel, usando la plataforma MS-Windows.

Cubriremos los principios básicos acerca de la arquitectura de una computadora, el lenguaje máquina y la programación de bajo nivel. Usted aprenderá suficiente lenguaje ensamblador como para aplicar sus conocimientos en la familia de microprocesadores más utilizada en la actualidad.

Antes de leer este libro, deberá haber completado un curso universitario o equivalente en programación de computadoras.

Un ensamblador es un programa que convierte el código fuente de los programas escritos en lenguaje ensamblador a lenguaje máquina. Un programa complementario, llamado enlazador, combina en un solo programa ejecutable los archivos individuales creados por un ensamblador. Un tercer programa, llamado depurador, proporciona los medios para que el programador rastree la ejecución de un programa y examine el contenido de la memoria.

Usted creará dos tipos básicos de programas: programas en modo de direccionamiento real de 16 bits y programas en modo protegido de 32 bits.

Aprenderá los siguientes conceptos de este libro: la arquitectura computacional básica, aplicada a los procesadores Intel IA-32; la lógica booleana elemental; la forma en que los procesadores IA-32 administran la memoria; la manera en que los compiladores de lenguajes de alto nivel traducen las instrucciones de su lenguaje a lenguaje ensamblador y código máquina nativo; la forma en que los lenguajes de alto nivel implementan las expresiones aritméticas, los ciclos y las estructuras lógicas a nivel de máquina; y la representación de datos de los enteros con y sin signo, los números reales y los datos tipo carácter.

El lenguaje ensamblador tiene una relación de *uno a uno* con el lenguaje máquina, en donde una sola instrucción en lenguaje ensamblador corresponde a una instrucción en lenguaje máquina. El lenguaje ensamblador no es portable, ya que está ligado a una familia específica de procesadores.

Los lenguajes son herramientas que podemos emplear en las aplicaciones individuales, o en algunas de sus partes. Algunas aplicaciones, tales como los controladores de dispositivos y las rutinas de interfaz de hardware, se adaptan mejor al lenguaje ensamblador. Otras aplicaciones, como las aplicaciones comerciales multiplataforma, se adaptan mejor a los lenguajes de alto nivel.

El concepto de *máquina virtual* es una manera efectiva de mostrar cómo cada nivel en la arquitectura de una computadora representa a una abstracción de una máquina. Los niveles pueden construirse de hardware o software, y los programas escritos en cualquier nivel pueden traducirse o interpretarse mediante el siguiente nivel inferior. El concepto de máquina virtual puede relacionarse con los niveles de las computadoras reales, incluyendo la lógica digital, la microarquitectura, la arquitectura del conjunto de instrucciones, el sistema operativo, el lenguaje ensamblador y los lenguajes de alto nivel.

Los números binarios y hexadecimales son herramientas de notación esenciales para los programadores que trabajan a nivel de máquina. Por esta razón, usted debe comprender cómo manipular y traducir números entre un sistema numérico y otro, y la forma en que las computadoras crean las representaciones de los caracteres.

En este capítulo se presentaron los siguientes operadores booleanos: NOT, AND y OR. Una expresión booleana combina a un operador booleano con uno o más operandos. Una tabla de verdad es una manera efectiva de mostrar todas las posibles entradas y salidas de una función booleana.

Notas finales

1. Donald Knuth, *MMIX, A RISC Computer for the New Millennium*, Transcripción de una conferencia impartida en el Instituto de Tecnología de Massachusetts, 30 de diciembre de 1999.
2. Andrew S. Tanenbaum, *Structured Computer Organization*, 5a. edición, Prentice Hall, 2005.
3. Su código fuente podría haberse escrito en C o en lenguaje ensamblador, pero una vez compilado, el sistema operativo es simplemente un programa de Nivel 2 que interpreta comandos de Nivel 3.
4. Fuente: www.webopedia.com.
5. En el sitio Web <http://www.unicode.org> podrá leer acerca del estándar Unicode.
6. En el Museo de Ciencias de Londres podrá ver un modelo funcional de la Máquina Analítica de Boole.

2

ARQUITECTURA DEL PROCESADOR IA-32

2.1 Conceptos generales

- 2.1.1 Diseño básico de una microcomputadora
- 2.1.2 Ciclo de ejecución de instrucciones
- 2.1.3 Lectura de la memoria
- 2.1.4 Cómo se ejecutan los programas
- 2.1.5 Repaso de sección

2.2 Arquitectura del procesador IA-32

- 2.2.1 Modos de operación
- 2.2.2 Entorno básico de ejecución
- 2.2.3 Unidad de punto flotante
- 2.2.4 Historia del microprocesador Intel
- 2.2.5 Repaso de sección

2.3 Administración de memoria del procesador IA-32

- 2.3.1 Modo de direccionamiento real

2.3.2 Modo protegido

2.3.3 Repaso de sección

2.4 Componentes de una microcomputadora IA-32

- 2.4.1 Tarjeta madre
- 2.4.2 Salida de video
- 2.4.3 Memoria
- 2.4.4 Puertos de entrada/salida e interfaces de dispositivos
- 2.4.5 Repaso de sección

2.5 Sistema de entrada/salida

- 2.5.1 Cómo funciona todo
- 2.5.2 Repaso de sección

2.6 Resumen del capítulo

2.1 Conceptos generales

En este capítulo 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. En este grupo se incluyen todos los procesadores compatibles con Intel, como AMD Athlon y Opteron. 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 computadora. Para este propósito, los conceptos y detalles que veremos en este capítulo le ayudarán a comprender el lenguaje ensamblador que escriba.

Buscamos un balance entre los conceptos que se aplican a todos los sistemas de microcomputadora y los detalles específicos acerca de los procesadores IA-32. Tal vez llegue a trabajar con varios procesadores en el futuro, por lo que le presentaremos conceptos completos. Para evitar que obtenga una comprensión superficial de la arquitectura de una máquina, nos enfocaremos en los detalles específicos acerca de la familia IA-32, que le brindarán una base sólida a la hora de programar en lenguaje ensamblador.

Si desea aprender más acerca de la arquitectura de la familia IA-32, lea el *Manual para el desarrollador de software de la arquitectura IA-32 de Intel (IA-32 Intel Architecture Software Developer's Manual), Volumen I: Arquitectura básica*. Puede descargar este manual en forma gratuita, a través del sitio Web de Intel (www.intel.com).

2.1.1 Diseño básico de una microcomputadora

La figura 2-1 muestra el diseño básico de una microcomputadora hipotética. La *unidad central de procesamiento* (CPU), 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 aritmética-lógica.

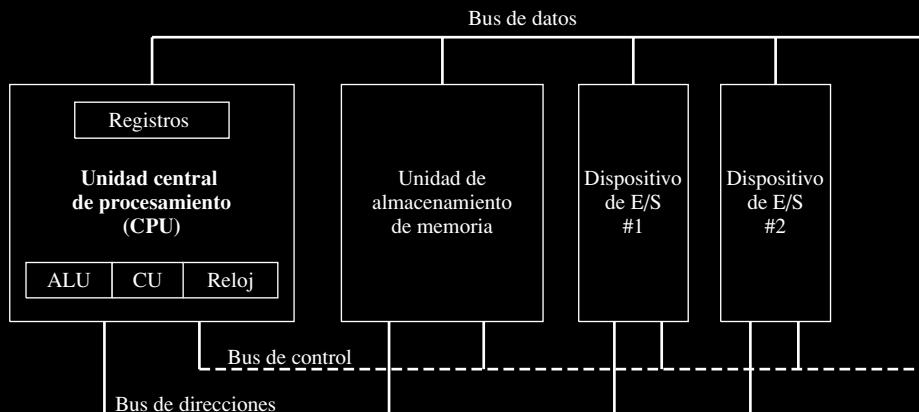
- El *reloj* sincroniza las operaciones internas de la CPU con los demás componentes del sistema.
 - La *unidad de control* (CU) coordina la secuencia de los pasos involucrados en la ejecución de instrucciones de máquina.
 - La *unidad aritmética-lógica* (ALU) realiza operaciones aritméticas como la suma y la resta, y operaciones lógicas como AND, OR y NOT.

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

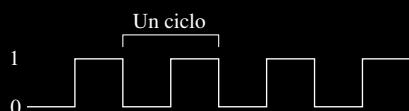
La unidad de almacenamiento de memoria es en donde se mantienen las instrucciones y los datos mientras se ejecuta un programa en la computadora. La unidad de almacenamiento recibe solicitudes de datos por parte de la CPU, transfiere datos de la memoria de acceso aleatorio (RAM) a la CPU, y transfiere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

Un *bus* es un grupo de cables en paralelo que transfieren datos de una parte de la computadora a otra. Por lo general, el bus del sistema de una computadora consiste en tres buses separados: el bus de datos, el bus de control y el bus de direcciones. El *bus de datos* transfiere instrucciones y datos entre la CPU y la memoria. El *bus de control* utiliza señales binarias para sincronizar las acciones de todos los dispositivos conectados al bus del sistema. El *bus de direcciones* almacena las direcciones de las instrucciones y los datos, cuando la instrucción actual que está en ejecución transfiere datos entre la CPU y la memoria. Muchas computadoras personales utilizan el bus PCI (*Interconexión de componentes periféricos*), desarrollado por Intel Corporation. Además, muchas computadoras tienen una ranura *PCI Express* para gráficos, la cual es mucho más rápida que la anterior ranura AGP para gráficos.

FIGURA 2–1 Diagrama de bloques de una microcomputadora.



Reloj Cada operación en la que intervienen la CPU y el bus del sistema se sincroniza mediante un reloj interno que emite pulsos a una velocidad constante. La unidad básica de tiempo para las instrucciones de máquina es un *ciclo de máquina* (o *ciclo de reloj*). La longitud de un ciclo de reloj es el tiempo requerido para un pulso completo del reloj. En la siguiente figura, un ciclo de reloj se ilustra como el tiempo que transcurre entre una caída y la siguiente:



La duración de un ciclo de reloj es el recíproco de la velocidad del mismo, y se mide en oscilaciones por segundo. Por ejemplo, un reloj que oscila mil millones de veces por segundo (1 GHz) produce un ciclo de reloj con una duración de mil millonésimas de un segundo (1 nanosegundo).

Una instrucción de máquina requiere, por lo menos, un ciclo de reloj para ejecutarse, y unas cuantas instrucciones requieren más de 50 ciclos de reloj (por ejemplo, la instrucción para multiplicar en el procesador 8088). A menudo, las instrucciones que requieren acceso a la memoria tienen ciclos de reloj vacíos, llamados *estados de espera*, debido a las diferencias en las velocidades de la CPU, el bus del sistema y los circuitos de memoria. (Una investigación reciente sugiere que en el futuro cercano es probable que abandonemos el modelo de computación sincronizado, para sustituirlo por un tipo de operación asíncrona que no requiera de un reloj del sistema).

2.1.2 Ciclo de ejecución de instrucciones

La ejecución de una sola instrucción de máquina puede dividirse en una secuencia de operaciones individuales, conocidas como el *ciclo de ejecución de instrucciones*. Antes de ejecutarse, un programa se carga en la 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 básicos: *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*. A continuación se describe cada uno de estos pasos:

- **Búsqueda:** la unidad de control busca la instrucción en la cola de instrucciones e incrementa el apuntador de instrucciones (IP). A este apuntador también se le conoce como el *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 programas 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 información acerca del estado del procesador.
- **Almacenamiento del operando del resultado:** si el operando de resultante está en memoria, la unidad de control utiliza una operación de escritura para almacenar el dato.

La secuencia de pasos puede expresarse muy bien en pseudocódigo:

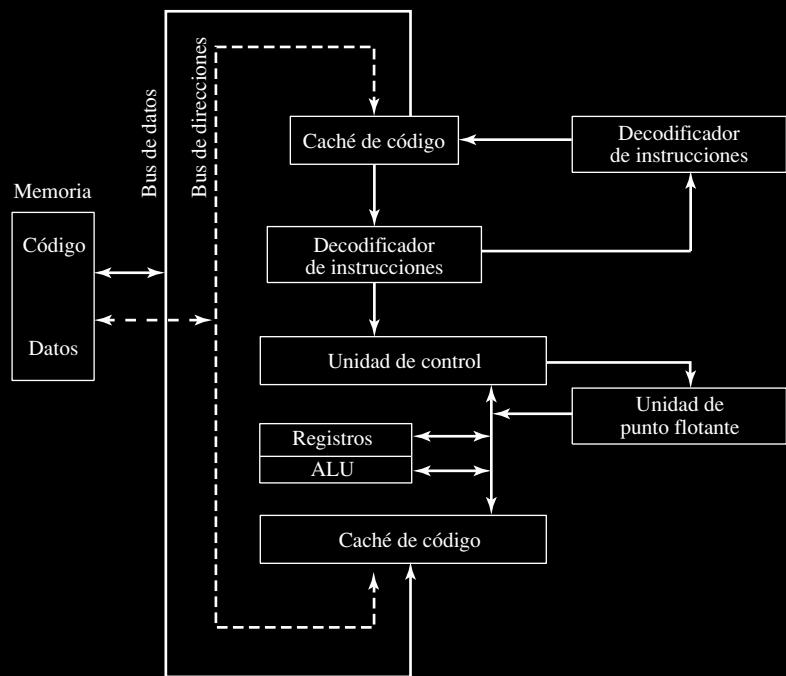
```
iterar
    obtener la siguiente instrucción
    avanzar el apuntador de instrucciones (IP)
    decodificar la instrucción
    si se necesita un operando en memoria, leer el valor de la memoria
    ejecutar la instrucción
    si el resultado es un operando en memoria, escribir el resultado en la memoria
    continuar el ciclo
```

El diseño básico del procesador Pentium, que se muestra en la figura 2-2, nos ayuda a mostrar las relaciones entre los componentes que interactúan durante el ciclo de ejecución de la instrucción. Por ejemplo, podemos ver la ruta que toman los datos al transferirse de la memoria a la caché de datos, a los registros y a la ALU. De manera similar, el diagrama muestra cómo la ALU y los registros pueden leer directamente de la caché de datos. El apuntador de instrucciones hace referencia a la caché de código, un área en la que se mantienen las instrucciones antes de ejecutarse. El decodificador de instrucciones lee de la caché de código y envía su salida a la unidad de control.

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 una técnica conocida

FIGURA 2–2 Diagrama de bloques simplificado de un procesador Pentium.



como *canalización*. Por ejemplo, el procesador Intel486 tiene un ciclo de ejecución de seis etapas. En la siguiente secuencia, cada etapa se asocia con una parte del procesador que ejecuta la etapa:

1. *Unidad de interfaz de bus (BIU)*: accede a la memoria y proporciona operaciones de entrada-salida.
2. *Unidad de búsqueda anticipada de código*: recibe las instrucciones de máquina de la BIU y las inserta en un área de retención llamada *cola de instrucciones*.
3. *Unidad de decodificación de instrucciones*: decodifica las instrucciones de máquina de la cola de búsqueda anticipada y las traduce en microcódigo.
4. *Unidad de ejecución*: ejecuta las instrucciones de microcódigo que produce la unidad de decodificación de instrucciones.
5. *Unidad de segmentación*: traduce las direcciones lógicas en direcciones lineales y realiza comprobaciones de protección.
6. *Unidad de paginación*: traduce las direcciones lineales en direcciones físicas, realiza comprobaciones de protección de páginas y mantiene una lista de las páginas con acceso reciente.

Ejemplo Suponga que cada etapa de ejecución en el procesador requiere un solo ciclo de reloj. La figura 2-3 utiliza una cuadrícula para representar a un procesador de seis etapas *no canalizado*, el tipo utilizado por Intel antes del Intel486. Cuando la instrucción I-1 termina la etapa S6, empieza la instrucción I-2. Se requieren doce ciclos de reloj para ejecutar las dos instrucciones. En otras palabras, para k etapas de ejecución, n instrucciones requieren $(n * k)$ ciclos para procesarse.

Canalización El procesador descrito en la figura 2-3 desperdicia recursos de la CPU, ya que cada etapa sólo se utiliza una sexta parte del tiempo. Si, por otra parte, un procesador soporta la canalización, como en la figura 2-4, una nueva instrucción puede entrar a la etapa S1 durante el segundo ciclo de reloj. Mientras tanto, la primera instrucción ha entrado a la etapa S2, con lo que se permite la ejecución traslapada de las instrucciones. La figura 2-4 muestra cómo dos instrucciones, I-1 e I-2, van en progreso a través de la canalización. I-2 entra a la etapa S1 tan pronto como I-1 se mueve a la etapa S2. Sólo se requieren siete ciclos de reloj para ejecutar dos instrucciones. Cuando la canalización está llena, hay seis etapas en uso continuo. En

general, para k etapas de ejecución, n instrucciones requieren $k + (n - 1)$ ciclos para procesarse. Mientras que el procesador no canalizado que describimos antes requiere 12 ciclos para procesar dos instrucciones, el procesador con canalización puede procesar siete instrucciones en el mismo tiempo.

FIGURA 2–3 Ejecución de una instrucción no canalizada de seis etapas.

		Etapas					
		S1	S2	S3	S4	S5	S6
Ciclos	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

FIGURA 2–4 Ejecución de una instrucción canalizada de seis etapas.

		Etapas					
		S1	S2	S3	S4	S5	S6
Ciclos	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. Para comprender mejor por qué sería útil un procesador superescalar, vamos a considerar el ejemplo anterior con canalización, en el que asumimos que la etapa de ejecución (S4) requería un solo ciclo de reloj. Ése fue un enfoque demasiado simple. ¿Qué ocurriría si la etapa S4 requiriera dos ciclos de reloj? Se produciría un cuello de botella, como se muestra en la figura 2–5. La instrucción I-2 no puede entrar a la etapa S4 hasta que I-2 haya completado la etapa, por lo que I-2 tiene que esperar un ciclo más antes de entrar a la etapa S4. A medida que van entrando más instrucciones a la canalización, se producen ciclos desperdiciados (sombreados en color gris). En general, para k etapas (en donde una etapa requiere dos ciclos), n instrucciones requieren $(k + 2n - 1)$ ciclos para procesarse.

FIGURA 2–5 Ejecución con canalización, utilizando una sola canalización.

		Etapas exe					
		S1	S2	S3	S4	S5	S6
Ciclos	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	
	8						I-2
	9						I-2
	10						I-3
	11						I-3

Un procesador superescalar permite que varias instrucciones estén en la etapa de ejecución al mismo tiempo. Para n canalizaciones, se pueden ejecutar n instrucciones durante el mismo ciclo de reloj. El Pentium de Intel, con dos canalizaciones, fue el primer procesador superescalar en la familia IA-32. El procesador Pentium Pro fue el primero en usar tres canalizaciones.

La figura 2-6 muestra un esquema de ejecución con dos canalizaciones, en una canalización de seis etapas. Asumimos que la etapa S4 requiere dos ciclos. Las instrucciones con numeración impar entran a la *canalización u*, y las instrucciones con numeración par entran a la *canalización v*. Los ciclos desperdiciados se eliminan, por lo que pueden ejecutarse n instrucciones en $(k + n)$ ciclos, en donde k indica el número de etapas.

FIGURA 2-6 Procesador escalar canalizado de 6 etapas.

Ciclos	Etapas						
	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-1	I-2		
6			I-4	I-3	I-2	I-1	
7				I-3	I-4	I-2	I-1
8					I-4	I-3	I-2
9						I-4	I-3
10							I-4

2.1.3 Lectura de la memoria

A menudo, el rendimiento de un programa depende de la velocidad del acceso a la memoria. La velocidad de acceso a la CPU podría ser de 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*.

Al leer las instrucciones o datos de la memoria se requieren varios pasos, los cuales se controlan mediante el reloj interno del procesador. La figura 2-7 muestra, las subidas y bajadas del reloj (CLK) del procesador, a intervalos de tiempo regulares. En la figura, un ciclo de reloj empieza cuando la señal del reloj cambia de nivel alto a bajo. Los cambios se llaman *orillas*, e indican el tiempo que ocupa la transición entre los estados. A continuación se muestra una descripción simplificada de lo que ocurre durante cada ciclo de reloj, en una operación de lectura de memoria:

Ciclo 1: los bits de dirección del operando en memoria se colocan en el *bus de direcciones* (ADDR).

Ciclo 2: la *línea de lectura* (RD) se establece en nivel bajo (0), para notificar a la memoria que se va a leer un valor.

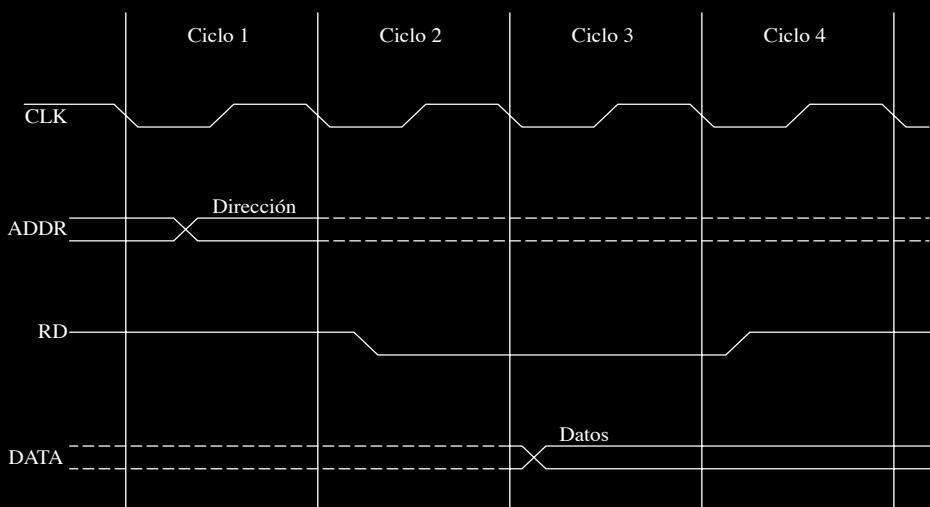
Ciclo 3: la CPU espera un ciclo para dar tiempo a la memoria a que responda. Durante este ciclo, el controlador de la memoria coloca el operando en el *bus de datos* (DATA).

Ciclo 4: la línea de lectura cambia a 1, indicando a la CPU que lea el dato en el bus de datos.

Memoria caché Como la memoria convencional es mucho más lenta que la CPU, las computadoras utilizan memoria en 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 la caché. Si el programa necesita leer los mismos datos una segunda vez, los busca en la caché. Un *acuerdo en la caché* indica que los datos se encuentran en la caché; un *fallo en la caché* indica que los datos no se encuentran en la 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. Los procesadores IA-32 tienen dos tipos de memoria caché: la *caché de Nivel 1* es más pequeña, más rápida y más costosa. La *caché de Nivel 2*, que una vez se encontraba fuera del procesador, ahora está integrada en el chip.

FIGURA 2–7 Ciclo de lectura de memoria.



2.1.4 Cómo se ejecutan los programas

Proceso de carga y ejecución

Los siguientes pasos describen, en secuencia, lo que ocurre cuando un usuario de computadora ejecuta un programa desde la línea de comandos:

- El sistema operativo (OS) busca el nombre de archivo del programa en el directorio del disco 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 de 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 se 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. Algunos ejemplos de recursos son: memoria, archivos en disco y dispositivos de entrada-salida.
- Cuando termina el proceso, se elimina su manejador y la memoria que utilizó se libera para que otros programas puedan utilizarla.

Si utiliza Windows 2000 o XP, oprima *Ctrl-Alt-Supr* y haga clic en el botón *Administrador de tareas*. Hay dos fichas llamadas *Aplicaciones* y *Procesos*. Las aplicaciones son los nombres de los programas completos que se encuentran en ejecución, como el Explorador de Windows o Microsoft Visual C++. Si hace clic en la ficha *Procesos*, verá que aparecen 30 o 40 nombres, de los cuales tal vez no reconozca muchos. Cada uno de estos procesos es un pequeño programa que se ejecuta en forma independiente de los demás. Observe que cada uno tiene un PID (*ID del proceso*) y que puede llevar el registro en forma continua del tiempo de CPU y la memoria que utiliza. La mayoría de los procesos se ejecutan en segundo plano. Puede terminar un proceso que de alguna forma se haya quedado ejecutándose en memoria por error. Desde luego que, si termina el proceso incorrecto, su computadora podría dejar de funcionar y tendrá que reiniciar su equipo.

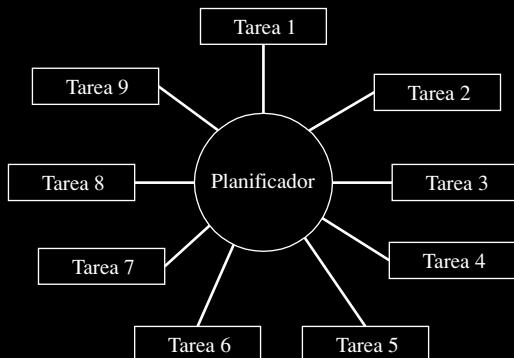
Multitarea

Un sistema operativo *multitarea* puede ejecutar varias tareas al mismo tiempo. Una *tarea* se define ya sea como un programa (un proceso) o un hilo (subproceso) de ejecución. Un proceso tiene su propia área de memoria y puede contener varios hilos (o subprocesos). Un hilo comparte su memoria con otros hilos que pertenecen al mismo proceso. Por ejemplo, los programas de juegos a menudo utilizan hilos individuales para controlar en forma simultánea varios objetos gráficos. Los exploradores Web utilizan varios hilos para cargar en forma simultánea las imágenes de gráficos y responder a la entrada que proporciona el usuario.

La mayoría de los sistemas operativos modernos ejecutan tareas en forma simultánea, las cuales se comunican con el hardware, muestran las interfaces de usuario, realizan el procesamiento de archivos en segundo plano, etcétera. 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. Durante una sola partición de tiempo, la CPU ejecuta un bloque de instrucciones y se detiene cuando termina esa partición de tiempo.

Al cambiar rápido de tareas, el procesador crea la ilusión de que se ejecutan en forma simultánea. Un tipo de planificador utiliza el OS que se conoce como *planificador por turnos (round-robin)*. En la figura 2-8 hay nueve tareas activas. Suponga que el *planificador* asigna en forma arbitraria 100 milisegundos a cada tarea, y que la acción de cambiar de una tarea a otra tarda 8 milisegundos. Un circuito completo de la lista de tareas requiere 964 milisegundos (9×100) + (10×8) para completarse.

FIGURA 2-8 Planificador de tareas por turnos.



Un OS multitareas se ejecuta en un procesador con soporte para *commutación 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 OS multitareas asigna distintas prioridades a las tareas, con lo cual reciben particiones de tiempo relativamente más grandes o más pequeñas. Un OS multitareas *preferente* (como Windows XP o Linux) permite que una tarea con mayor prioridad interrumpa a una con menor prioridad, lo cual conlleva a una mejor estabilidad del sistema. Suponga que un programa de aplicación está bloqueado en un ciclo y ha dejado de responder a la entrada. El manejador del teclado (una tarea de alta prioridad del OS) puede responder al comando Ctrl-Alt-Supr del usuario y cerrar el programa de aplicación que tuvo el error.

2.1.5 Repaso de sección

1. ¿Qué otros elementos básicos contiene la unidad central de procesamiento (CPU), aparte de los registros?
2. ¿Cuáles son los tres buses que utiliza la unidad central de procesamiento para conectarse al resto del sistema computacional?
3. ¿Por qué el acceso a la memoria ocupa más ciclos de máquina que el acceso a los registros?
4. ¿Cuáles son los tres pasos básicos en el ciclo de ejecución de instrucciones?
5. ¿Cuáles son los dos pasos adicionales que se requieren en el ciclo de ejecución de instrucciones, cuando se utiliza un operando en memoria?

6. ¿En qué etapa del ciclo de ejecución de instrucciones se incrementa el contador del programa?
7. Defina *ejecución canalizada*.
8. En un procesador no canalizado de cinco etapas, ¿cuántos ciclos de reloj se necesitarían para ejecutar dos instrucciones?
9. En un procesador de cinco etapas con una canalización, ¿cuántos ciclos de reloj se necesitarían para ejecutar ocho instrucciones?
10. ¿Qué es un *procesador superescalar*?
11. Suponga que un procesador de cinco etapas con doble canalización tiene una etapa que requiere dos ciclos de reloj para ejecutarse, y que hay dos canalizaciones para esa etapa. ¿Cuántos ciclos de reloj se necesitarían para ejecutar 10 instrucciones?
12. Cuando se ejecuta un programa, ¿qué información lee el OS de la entrada de directorio en el disco del nombre de archivo?
13. Una vez que se carga un programa en la memoria, ¿cómo empieza a ejecutarse?
14. Defina el concepto de *multitarea*.
15. ¿Cuál es la función del planificador del OS?
16. Cuando el procesador comuta de una tarea a otra, ¿qué valores deben preservarse en el estado de la primera tarea?
17. ¿Cuál es la duración de un ciclo de reloj individual en un procesador de 3 GHz?

2.2 Arquitectura del procesador IA-32

Como dijimos antes, IA-32 se refiere a una familia de procesadores, que inicia con el Intel386 y continúa hasta el procesador más reciente de 32 bits, el Pentium 4. Con el tiempo se han realizando numerosas mejoras a la arquitectura interna de los procesadores Intel, como las canalizaciones, la tecnología superescalar, la predicción de saltos, y la tecnología Hyperthreading. En términos de programación, los cambios visibles son las extensiones del conjunto de instrucciones para el procesamiento de multimedia y los cálculos de los gráficos.

2.2.1 Modos de operación

Los procesadores IA-32 tienen tres modos principales de operación: modo protegido, modo de direccionamiento real y modo de administración del sistema. Hay otro modo llamado 8086 virtual, que es un caso especial del modo protegido. He aquí las descripciones breves de cada modo:

Modo protegido El modo protegido es el estado nativo del procesador, en el que están disponibles todas las instrucciones y características. Los programas reciben áreas separadas de memoria llamadas *segmentos*, y el procesador evita que los programas hagan referencia a la memoria que se encuentra fuera de sus segmentos asignados.

Modo 8086 virtual Mientras se encuentra en modo protegido, el procesador puede ejecutar en forma directa el software para 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 ejecuten 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 el acceso directo a la memoria del sistema y a los dispositivos de hardware. Los programas que se ejecutan en modo de direccionamiento real pueden hacer que el sistema operativo falle (que deje de responder a los comandos).

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, quienes personalizan el procesador para una configuración específica del sistema.

2.2.2 Entorno básico de ejecución

Espacio de direcciones

Los procesadores IA-32 pueden acceder a 4 GB 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 modo de 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.

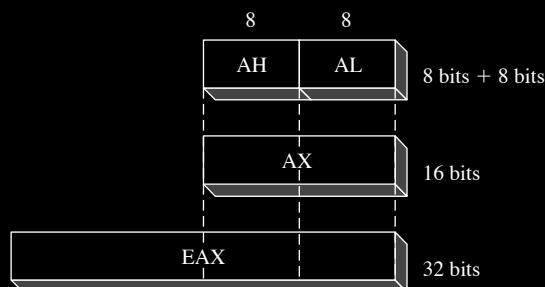
Registros básicos de ejecución de un programa

Los *registros* son ubicaciones de almacenamiento de alta velocidad, que se encuentran directamente dentro de la CPU, y están diseñados para una velocidad de acceso mucho mayor que la de la memoria convencional. Por ejemplo, cuando se optimiza un ciclo de procesamiento en base a la velocidad, los contadores del ciclo se guardan en registros, en vez de variables. La figura 2-9 muestra los *registros básicos de ejecución de un programa*. Hay ocho registros de propósito general, seis registros de segmento, un registro de las banderas de estado del procesador (EFLAGS), y un apuntador de instrucciones (EIP).

FIGURA 2-9 Registros básicos de ejecución de un programa de los procesadores IA-32.



Registros de propósito general Los *registros de propósito general* se utilizan principalmente para las operaciones aritméticas y el movimiento de datos. Como se muestra en la siguiente figura, cada registro puede direccionarse como un valor individual de 32 bits, o como dos valores de 16 bits.



Hay partes de algunos registros que pueden direccionarse como valores de 8 bits. Por ejemplo, el registro EAX de 32 bits tiene una mitad inferior de 16 bits llamada AX. A su vez, el registro AX tiene una mitad

superior de 8 bits llamada AH, y una mitad inferior de 8 bits llamada AL. La misma relación de traslape existe para los registros EAX, EBX, ECX y EDX:

32 bits	16 bits	8 bits (superior)	8 bits (inferior)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

El resto de los registros de propósito general sólo tienen 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 utilizan de manera automática las instrucciones de multiplicación y división. A menudo se le conoce como el registro *acumulador extendido*.
- La CPU utiliza de manera automática a ECX como su contador de ciclo.
- ESP dirige 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 le conoce como el registro *apuntador de pila extendido*.
- ESI y EDI son los 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 un nivel avanzado de programación. A menudo se le conoce como el registro *apuntador de estructura extendido*.

Registros de segmento En el modo de direccionamiento real, los registros de segmento indican las direcciones base de las áreas preasignadas 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 otro 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 ubicación.

Registro EFLAGS El registro EFLAGS (o simplemente *Flags*) consiste en bits binarios individuales que controlan la operación de la CPU, o que reflejan el resultado 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* (o borra) 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 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 al modo protegido.

Los programas pueden activar bits individuales en el registro EFLAGS para controlar la operación de la CPU. Algunos ejemplos son las banderas de *Dirección* y de *Interrupción*.

Banderas de estado Las banderas de estado reflejan los resultados de las operaciones aritméticas y lógicas que realiza la CPU. Estas banderas son: Desbordamiento, Signo, Cero, Acarreo Auxiliar, Paridad y Acarreo. Sus abreviaturas se muestran justo después de sus nombres:

- La bandera **Acarreo** (CF) se activa cuando el resultado de una operación aritmética *sin signo* es demasiado grande para caber en el destino.
- La bandera **Desbordamiento** (OF) se activa cuando el resultado de una operación aritmética *con signo* es demasiado grande o pequeño para caber en el destino.
- La bandera **Signo** (SF) se activa cuando el resultado de una operación aritmética o lógica genera un resultado negativo.
- La bandera **Cero** (ZF) se activa cuando el resultado de una operación aritmética o lógica genera un resultado de cero.
- La bandera **Acarreo auxiliar** (AC) se activa cuando una operación aritmética produce un acarreo del bit 3 al bit 4, en un operando de 8 bits.
- La bandera **Paridad** (PF) se activa si el byte menos significativo en el resultado contiene un número par de bits que sean 1. En caso contrario, PF está desactivada. En general, se utiliza para comprobar errores cuando existe la posibilidad de que los datos estén alterados o corruptos.

Registros del sistema

Los procesadores IA-32 tienen una variedad de registros importantes del sistema. MS-Windows sólo permite el acceso a estos registros a los programas que se ejecutan en el nivel más alto de privilegio (nivel 0). El núcleo (kernel) del sistema operativo es uno de estos programas. Los registros del sistema son:

- **IDTR (Registro de tabla de descriptores de interrupciones):** este registro contiene la dirección de la Tabla de descriptores de interrupciones, la cual proporciona los medios para manejar las interrupciones (rutinas del sistema, diseñadas para responder a eventos tales como los generados por el teclado y el ratón).
- **GDTR (Registro de tabla de descriptores globales):** el registro GDTR contiene la dirección de la Tabla de descriptores globales, una tabla que contiene apuntadores a los segmentos de estado de las tareas y tablas de descriptores locales de los programas.
- **LDTR (Registro de tabla de descriptores locales):** el registro LDTR contiene apuntadores al código, los datos y la pila de los programas que se están ejecutando en un momento dado.
- **Registro de tareas:** este registro contiene la dirección del TSS (Segmento de estado de tarea) para la tarea que se está ejecutando en un momento dado.
- **Registros de depuración:** los registros de depuración permiten que los programas establezcan puntos de interrupción al momento de depurar los programas.
- **Registros de control CR0, CR2, CR3, CR4:** los registros de control contienen banderas de estado y campos de datos que controlan las operaciones a nivel del sistema, como la conmutación, la paginación y la habilitación de la memoria caché (el registro CR1 no se utiliza).
- **Registros específicos del modelo:** estos registros se utilizan para tareas del sistema operativo tales como el monitoreo del rendimiento y la comprobación de la arquitectura de la máquina. Su uso varía, dependiendo de los distintos procesadores IA-32.

En el capítulo 11 hablaremos sobre los registros GDTR y LDTR, en el contexto de la administración de memoria en modo protegido. Los programas de aplicaciones no pueden acceder a los registros del sistema. Como este libro se centra en los programas de aplicaciones en lenguaje ensamblador, no utilizaremos los registros del sistema.

2.2.3 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 del Intel486

a la fecha, la FPU está integrada en el chip 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) y ST(7). El resto de los registros de control y de apuntadores se muestran en la figura 2-10.

FIGURA 2-10 Registros de la unidad de punto flotante.



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).

2.2.4 Historia del microprocesador Intel

Vamos a dar un corto paseo por la historia de la memoria de computadora, desde que la IBM-PC salió al mercado por primera vez, cuando las PCs tenían 64K de RAM y no utilizaban discos duros.

Intel 8086 El procesador Intel 8086 (1978) marca el inicio de la familia de la arquitectura moderna de Intel. Las principales innovaciones del 8086 en comparación con los primeros procesadores fueron que tenía registros de 16 bits y un bus de datos de 16 bits; además utilizaba un modelo de memoria segmentada, el cual permitía a los programas direccionar hasta 1MB de RAM. El mayor acceso a memoria hizo posible la escritura de aplicaciones comerciales complejas. La IBM-PC (1980) contenía un procesador Intel 8088, que era idéntico al 8086, con la excepción de que tenía un bus de datos de 8 bits y, por lo tanto, era menos costosa su producción. Hoy en día, el Intel 8088 se utiliza en los microcontroladores de bajo costo.

Compatibilidad descendente. Cada procesador que se introdujo en la familia Intel desde el 8086 tiene compatibilidad descendente con los procesadores anteriores. Este enfoque permite que el software antiguo funcione en computadoras más recientes, sin necesidad de modificación. Con el tiempo apareció software más nuevo, que requería las características de los procesadores más avanzados.

Intel 80286 El procesador Intel 80286, que se utilizó por primera vez en la computadora IBM-PC/AT, estableció un nuevo estándar de velocidad y potencia. Era el primer procesador Intel que se ejecutaba en modo protegido. El 80286 puede direccionar hasta 16MB de RAM, mediante el uso de un bus de direcciones de 24 bits.

La familia de procesadores IA-32

El procesador Intel386 introdujo los registros de datos de 32 bits y un bus de direcciones de 32 bits, además de una ruta de datos externa. Como tal, fue el primer miembro de la familia IA-32. Los procesadores IA-32 pueden direccionar una memoria virtual más grande que la memoria física de la computadora. A cada programa se le asigna un espacio de direcciones lineales de 4GB.

Intel486 Continuando con la familia IA-32, el procesador Intel486 contiene una microarquitectura de conjunto de instrucciones en la que se utilizan técnicas de canalización, las cuales le permiten procesar varias instrucciones al mismo tiempo.

Pentium El procesador Pentium agregó muchas mejoras en cuanto al rendimiento, incluyendo un diseño superescalar con dos canalizaciones de ejecución en paralelo. Puede decodificar y ejecutar dos instrucciones en forma simultánea. El Pentium utiliza un bus de direcciones de 32 bits y una ruta de datos interna de 64 bits; además introdujo la tecnología MMX en la familia IA-32.

La familia de procesadores P6

La familia de procesadores P6 se introdujo en 1995, con base en un nuevo diseño de microarquitectura que mejoraba la velocidad de ejecución. También extendió la arquitectura IA-32 básica. La familia P6 incluye al Pentium Pro, Pentium II y Pentium III. El Pentium Pro introdujo técnicas avanzadas para mejorar la forma en que se ejecutaban las instrucciones. El Pentium II agregó la tecnología MMX a la familia P6. El Pentium III introdujo las extensiones SIMD (extensiones de flujo continuo) en la familia IA-32, con registros especializados de 128 bits, diseñados para mover grandes cantidades de datos con rapidez.

La familia de procesadores Pentium 4 y Xeon

Los procesadores Pentium 4 y Xeon utilizan la microarquitectura *NetBurst* de Intel, la cual permite al procesador operar a velocidades más altas que los procesadores IA-32 anteriores. Está optimizada para aplicaciones multimedia de alto rendimiento. Los procesadores Pentium 4 más avanzados incluyen la tecnología *Hyperthreading*, la cual ejecuta aplicaciones con subprocesamiento múltiple en paralelo, en un procesador multinúcleo.

CISC y RISC

El procesador 8086 fue el primero en una línea de procesadores que utilizaba el diseño de *Computadora con un conjunto complejo de instrucciones* (CISC). El conjunto de instrucciones es extenso e incluye una amplia variedad de operaciones para direccionamiento de memoria, de corrimientos, aritméticas, para mover datos y lógicas. Los conjuntos complejos de instrucciones permiten que los programas compilados contengan un número relativamente pequeño de instrucciones. Una de las principales desventajas del diseño CISC es que las instrucciones complejas requieren de un tiempo relativamente extenso para decodificarse y ejecutarse. Un intérprete dentro de la CPU, escrito en un lenguaje llamado *microcódigo*, decodifica y ejecuta cada instrucción de máquina. Una vez que Intel sacó al mercado el 8086, era necesario que todos los subsiguientes procesadores Intel fueran compatibles con el primero. Los clientes no querían deshacerse de su software existente cada vez que se sacaba al mercado un nuevo procesador.

Un enfoque completamente distinto en el diseño de microprocesadores es el *Conjunto reducido de instrucciones* (RISC). Un RISC consiste en un número relativamente pequeño de instrucciones cortas y simples, que se ejecutan con una rapidez relativa. En vez de usar un intérprete de microcódigo para decodificar y codificar instrucciones de máquina, un procesador RISC decodifica y ejecuta instrucciones en forma directa, mediante el uso de hardware. Las estaciones de trabajo con gráficos e ingeniería de alta velocidad se construyeron usando procesadores RISC durante muchos años. Por desgracia, los sistemas han sido costosos ya que los procesadores se producían en pequeñas cantidades.

Debido a la enorme popularidad de las computadoras compatibles con la IBM PC, Intel pudo reducir el precio de sus procesadores y dominar el mercado de los microprocesadores. Al mismo tiempo, Intel reconoció muchas ventajas en relación con el enfoque RISC y encontró una manera de utilizar características similares a las de RISC (como la canalización y la tecnología superescalar) en la serie Pentium. El conjunto de instrucciones IA-32 continúa siendo complejo y está en constante expansión.

2.2.5 Repaso de sección

1. ¿Cuáles son los tres modos básicos de operación del procesador IA-32?
2. Mencione los ocho registros de propósito general de 32 bits.
3. Mencione los seis registros de segmento.
4. ¿Qué propósito especial tiene el registro ECX?
5. Además del apuntador de pila (ESP), ¿qué otro registro apunta a las variables en la pila?
6. Mencione por lo menos cuatro banderas de estado de la CPU.
7. ¿Qué bandera se activa cuando el resultado de una operación aritmética *sin signo* es demasiado grande como para caber en el destino?
8. ¿Qué bandera se activa cuando el resultado de una operación aritmética *con signo* es demasiado grande o demasiado pequeño como para caber en el destino?
9. ¿Qué bandera se activa cuando una operación aritmética o lógica genera un resultado negativo?
10. ¿Qué parte de la CPU realiza operaciones aritméticas de punto flotante?
11. ¿De cuántos bits son los registros de datos de la FPU?
12. ¿Qué procesador Intel fue el primer miembro de la familia IA-32?
13. ¿Qué procesador Intel introdujo por primera vez la ejecución superescalar?
14. ¿Qué procesador Intel utilizó por primera vez la tecnología MMX?
15. Describa el enfoque del diseño CISC.
16. Describa el enfoque del diseño RISC.

2.3 Administración de memoria del procesador IA-32

Los procesadores IA-32 administran la memoria de acuerdo a los modos básicos de operación que vimos en la sección 2.2.1. 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 el modo de *direcciónamiento real* sólo puede direccionarse 1MB de memoria, del 00000 al FFFFF hexadecimal. El procesador sólo puede ejecutar un programa a la vez, pero puede interrumpir en forma momentánea ese programa para procesar las solicitudes (conocidas como *interrupciones*) de los periféricos. Los programas de aplicación pueden leer y modificar cualquier área de la RAM (memoria de acceso aleatorio) y pueden leer pero no modificar cualquier área de la ROM (memoria de sólo lectura). El sistema operativo MS-DOS se ejecuta en modo de direcciónamiento real, y Windows 95/98 puede cargarse en este modo.

En el 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 los datos de los demás programas. MS-Windows y Linux se ejecutan en modo protegido.

En el modo *8086 virtual*, la computadora se ejecuta en modo protegido y crea una máquina 8086 virtual con su propio espacio de direcciones de 1MB, que simula a una computadora 80×86 que se ejecuta en modo de direcciónamiento real. Por ejemplo, Windows NT y 2000 crean una máquina 8086 virtual cuando abrimos una ventana de *Comandos*. Puede ejecutar muchas de esas ventanas al mismo tiempo, y cada una está protegida contra las acciones de las demás. 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.

En las secciones 2.3.1 y 2.3.2 explicaremos los detalles del modo de direcciónamiento real y del modo protegido. Si desea estudiar este tema con más detalle, una buena fuente de información es el *Manual para el desarrollador de software de la arquitectura Intel IA-32 (IA-32 Intel Architecture Software Developer's Manual)*, que consta de tres volúmenes. Puede leerlo o descargarlo del sitio Web de Intel (www.intel.com).

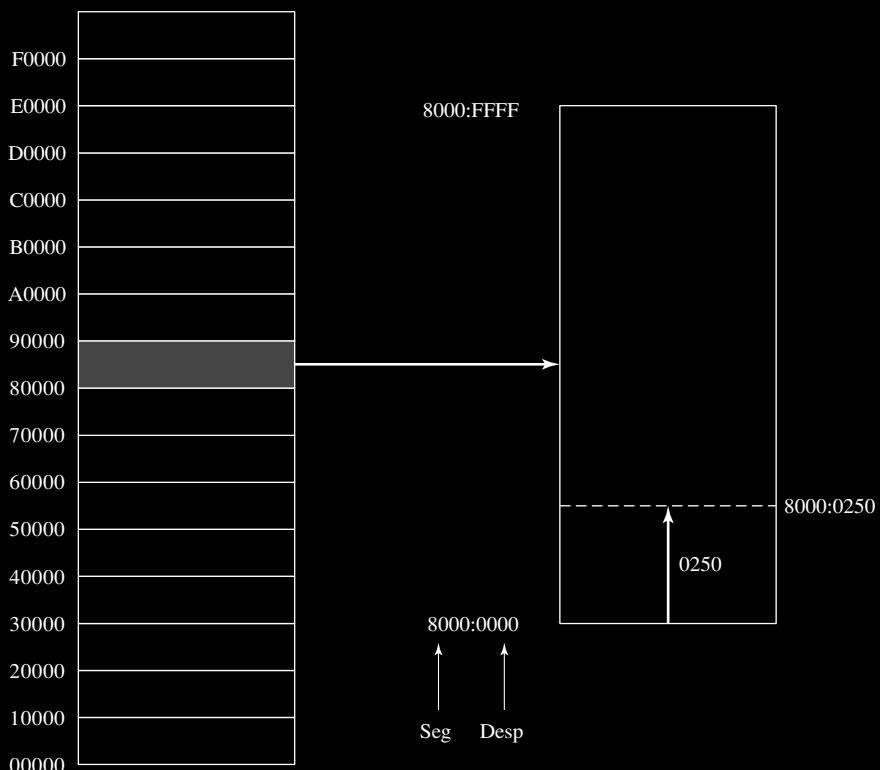
2.3.1 Modo de direcciónamiento real

En el modo de direcciónamiento real, el procesador IA-32 puede acceder a 1,048,576 bytes de memoria (1MB) 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 64 Kilobytes, a las cuales se les llama *segmentos*, como se muestra en la figura 2-11. Una analogía es un edificio extenso, en el que los *segmentos* representan a los pisos del edificio. Una persona puede tomar el elevador hacia un piso específico, bajarse y empezar a seguir los números de los cuartos para localizar uno. El *desplazamiento* de un cuarto puede considerarse como la distancia desde el elevador hasta ese cuarto.

En la figura 2-11, 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 de C000 hace referencia al segmento en la dirección C0000. La misma figura muestra una expansión del segmento en la dirección 80000. Para acceder a un byte en este segmento, se suma un desplazamiento de 16 bits (de 0 a FFFF) 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.

FIGURA 2-11 Mapa de la memoria segmentada en el modo de direccionamiento real.



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 tipo segmento-desplazamiento en una dirección lineal de 20 bits. Suponga que la dirección segmento-desplazamiento hexadecimal de una variable es 08F1:0100. La CPU multiplica el valor del segmento por 16 (10 hexadecimal) y suma el producto al desplazamiento de la variable:

$08F1h * 10h = 08F10h$	(valor de segmento ajustado)
Valor de segmento ajustado:	0 8 F 1 0
Se suma el desplazamiento:	0 1 0 0
Dirección lineal:	0 9 0 1 0

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 **pila** de 16 bits.
- ES, FS y GS pueden apuntar a segmentos de datos alternativos.

2.3.2 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 desde 0 hasta FFFFFFFF hexadecimal. En el contexto de Microsoft Assembler, el modelo de memoria **plano** (consulte la directiva .MODEL) 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 la 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 uno 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). La figura 2-12 muestra un descriptor de segmento cuyo campo *dirección base* apunta a la primera ubicación disponible en la memoria (00000000). El campo *límite del 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.

Suponga que una computadora tiene 256MB de RAM. El campo límite del segmento contendría el número 10000 hex, ya que su valor se multiplica en forma implícita por 1000 hex, produciendo el número 10000000 hex (256MB) como resultado.

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. En la figura 2-13, cada entrada en la LDT apunta a un segmento distinto en la memoria. Cada descriptor de segmento especifica el tamaño exacto de su segmento. Por ejemplo, el segmento que empieza en 3000 tiene un tamaño de 2000 hexadecimal, que se calcula como (0002 * 1000 hexadecimal). El segmento que empieza en 8000 tiene el tamaño de A000 hexadecimal.

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, conocidos 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*. Los sistemas operativos tienen programas utilitarios llamados *administradores de memoria virtual*.

FIGURA 2–12 Modelo de segmentación plano.

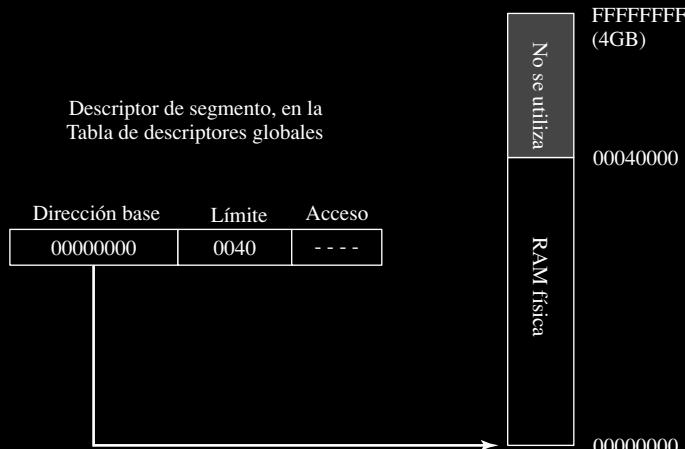
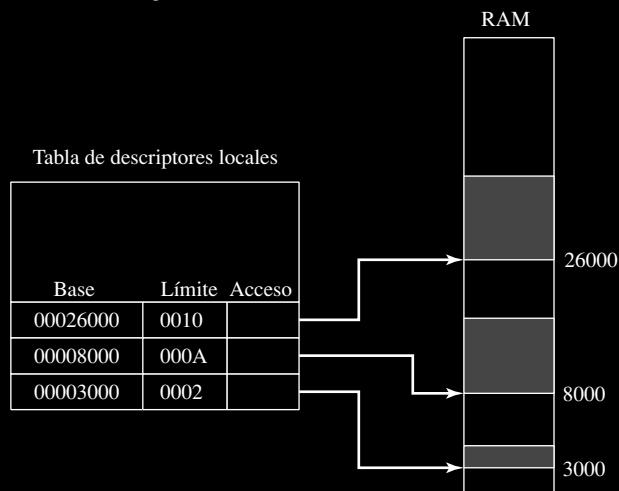


FIGURA 2–13 Modelo de varios segmentos.



La paginación es una solución importante para un molesto problema al que se enfrentan los diseñadores de software y hardware. Un programa debe cargarse en la memoria principal para poder ejecutarse, pero la memoria es costosa. Los usuarios desean cargar numerosos programas en la memoria y cambiar de uno a otro según lo deseen. Por otro lado, el almacenamiento en disco es económico y vasto. La paginación crea la ilusión de que la memoria es casi ilimitada en tamaño. El acceso al disco es mucho más lento que el acceso a la memoria principal, por lo que entre más dependa un programa de la paginación, se ejecutará con más lentitud.

Cuando una tarea se ejecuta, algunas de sus partes pueden almacenarse en disco si no se encuentran en uso en ese momento. Las partes de la tarea se *paganan* (intercambian) en el disco. Otras páginas activas en ejecución permanecen en la memoria. Cuando el programa empieza a ejecutar código que se ha paginado fuera de la memoria, produce un *fallo de página* para que la página o páginas que contienen el código o datos requeridos se carguen de vuelta en la memoria. Para ver cómo funciona esto, elija una computadora con memoria algo limitada y ejecute muchas aplicaciones extensas al mismo tiempo. Deberá observar un retraso al cambiar de un programa a otro, debido a que el OS debe transferir las porciones paginadas de cada programa, del disco

a la memoria. Una computadora funciona a una mayor velocidad cuando hay más memoria instalada, ya que los archivos y programas de aplicaciones extensas pueden mantenerse por completo en la memoria, con lo que se reduce la cantidad de paginación.

2.3.3 Repaso de sección

1. ¿Cuál es el rango de memoria direccionable en modo protegido?
2. ¿Cuál es el rango de memoria direccionable en modo de direccionamiento real?
3. Las dos formas de describir una dirección en el modo de direccionamiento real son: segmento-desplazamiento y _____.
4. En el modo de direccionamiento real, convierta la siguiente dirección segmento-desplazamiento hexadecimal en una dirección lineal: 0950:0100.
5. En el modo de direccionamiento real, convierta la siguiente dirección segmento-desplazamiento hexadecimal en una dirección lineal: OCD1:02E0.
6. En el modelo de memoria plano de MASM, ¿cuántos bits almacenan la dirección de una instrucción o variable?
7. En el modo protegido, ¿qué registro hace referencia al descriptor para el segmento de pila?
8. En el modo protegido, ¿qué tabla contiene apuntadores a segmentos de memoria utilizados por un solo programa?
9. En el modelo de segmentación plano, ¿qué tabla contiene apuntadores a los últimos dos segmentos?
10. ¿Cuál es la principal ventaja al utilizar la característica de paginación de los procesadores IA-32?
11. *Reto:* ¿puede pensar por qué MS-DOS no se diseñó para soportar la programación en modo protegido?
12. *Reto:* en el modo de direccionamiento real, demuestre dos direcciones tipo segmento-desplazamiento que apuntan a la misma dirección lineal.

2.4 Componentes de una microcomputadora IA-32

En este capítulo le presentaremos la arquitectura de los procesadores IA-32, desde varios puntos de vista. Primero, podemos ver el hardware (las partes físicas de la computadora) desde un *macro* nivel, analizando los periféricos. Después podemos ver los detalles internos del procesador Intel, conocido como *unidad central de procesamiento* (CPU). Por último vamos a ver la arquitectura de software, que es la forma en que está organizada la memoria, y cómo interactúa el sistema operativo con el hardware.

2.4.1 Tarjeta madre

El corazón de una microcomputadora es su *tarjeta madre*, un tablero de circuitos plano en el que se colocan la CPU de la computadora, los procesadores de soporte (*juego de chips*), la memoria principal, los conectores de entrada-salida, los conectores de la 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 docenas de tarjetas madre disponibles en el mercado de las PCs, las cuales varían en cuanto a su capacidad de expansión, los componentes integrados y la velocidad. Los siguientes componentes se encuentran de manera tradicional en las tarjetas madre de las PCs:

- Un zócalo para la CPU. Hay zócalos de distintas figuras y tamaños, dependiendo del tipo de procesador que soportan.
- Ranuras de memoria (SIMM o DIMM) que alojan pequeñas tarjetas de memoria insertables.
- Chips del BIOS (*sistema básico de entrada-salida*) de la computadora, que almacena el software del sistema.
- 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.
- Puertos 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 importantes 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* 8284/82C284, conocido simplemente como el *reloj*, oscila a una velocidad constante. El generador de reloj sincroniza a la CPU con el resto de la computadora.
- El *Controlador de interrupciones programable* (PIC) 8259A maneja las interrupciones externas de los dispositivos de hardware, como el teclado, el reloj de sistema y las unidades de disco. Estos dispositivos interrumpen a la CPU y hacen que procese sus solicitudes en forma inmediata.
- El *Temporizador/Contador de intervalos programable* 8253 interrumpe al sistema 18.2 veces por segundo, actualiza la fecha y el reloj del sistema, y controla el altavoz. También es responsable de actualizar la memoria en forma constante, ya que los chips de memoria RAM pueden recordar sus datos sólo durante unos cuantos milisegundos.
- El *Puerto paralelo programable* 8255 transfiere datos desde y hacia la computadora, usando la interfaz IEEE de puerto paralelo. Este puerto se utiliza, por lo general, para las impresoras, pero puede utilizarse también con otros dispositivos de entrada-salida.

Arquitecturas de los buses PCI y PCI Express

El bus **PCI** (*Interconexión de componentes periféricos*) proporciona un puente de conexión entre la CPU y 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 de gráficos, y puede transferir datos a una velocidad aproximada de 4GB por segundo.

Juego de chips de la tarjeta madre

La mayoría de las tarjetas madre 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. Los nombres aquí presentados son producidos por Intel, pero muchas tarjetas madre utilizan juegos de chips compatibles de otros fabricantes:

- El controlador Intel 8237 de Acceso directo a memoria (DMA) transfiere datos entre los dispositivos externos y la RAM, sin requerir que la CPU haga algo.
- El Controlador de interrupciones Intel 8259A maneja las solicitudes del hardware para interrumpir a la CPU.
- El Temporizador Contador 8254 maneja el reloj del sistema, el cual emite pulsos 18.2 veces por segundo, el temporizador de actualización de la memoria, y el reloj de la hora del día.
- El puente del bus local del microprocesador al bus PCI.
- El controlador de memoria del sistema y el controlador de la caché.
- El puente del bus PCI al bus ISA.
- El microcontrolador Intel 8042 de teclado y ratón.

2.4.2 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 enviarlos 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.

Los monitores de video de tubo de rayos catódicos (CRT) utilizan una técnica conocida como *barrido de trama* (*raster scanning*) para mostrar imágenes. Un rayo de electrones ilumina los puntos de fósforo en la pantalla, llamados *píxeles*. Empezando en la parte superior de la pantalla, el cañón dispara electrones desde

el lado izquierdo hasta el lado derecho en una fila horizontal, se apaga brevemente y regresa al lado izquierdo de la pantalla para empezar una nueva fila. El *retorno de barrido horizontal (horizontal retrace)* se refiere al tiempo durante el cual el cañón se apaga entre una fila y otra. Cuando se dibuja la última fila, el cañón se apaga (a lo que se le conoce como *retorno de barrido vertical*) y se desplaza hasta la esquina superior izquierda de la pantalla, para empezar de nuevo.

Un monitor de pantalla de cristal líquido (LCD) digital directo recibe un flujo de bits digitales directamente desde el controlador de video, y no requiere del barrido de trama. Por lo general, las pantallas digitales muestran un texto más fino que las pantallas analógicas.

2.4.3 Memoria

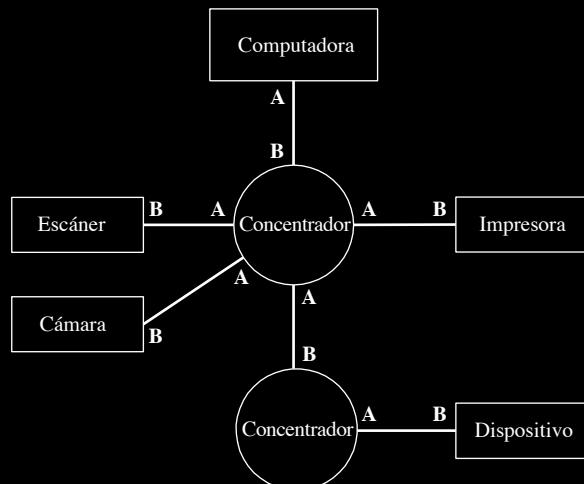
En los sistemas basados en Intel se utilizan varios tipos de memoria: memoria de sólo lectura (ROM), memoria de sólo lectura programable y borrible (EPROM), memoria dinámica de acceso aleatorio (DRAM), RAM estática (SRAM), RAM de video (VRAM) y RAM de metal-óxido semiconductor complementario (CMOS):

- La **ROM** se quema de manera permanente en un chip y no puede borrarse.
- La **EPROM** puede borrarse lentamente con luz ultravioleta, y puede volver a programarse.
- La **DRAM**, conocida comúnmente como memoria principal, es en donde se guardan los programas y datos cuando hay un programa en ejecución. Es económica pero debe actualizarse en un lapso no mayor de 1 milisegundo, ya que de lo contrario pierde su contenido. Algunos sistemas utilizan memoria ECC (comprobación y corrección de errores).
- La **SRAM** se utiliza principalmente para la memoria caché de alta velocidad, que es costosa. No tiene que actualizarse. La memoria caché de la CPU está compuesta de SRAM.
- La **VRAM** almacena datos de video. Tiene doble puerto, y permite que un puerto actualice en forma continua la pantalla, mientras que otro puerto escribe datos.
- La **RAM de CMOS** en la tarjeta madre del sistema almacena la información de configuración. Se actualiza mediante una batería, por lo que su contenido se retiene cuando se apaga la computadora.

2.4.4 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 de 480 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. En la figura 2-14 se muestra un concentrador USB, que es un dispositivo compuesto conectado a otros dispositivos, incluyendo otros concentradores USB.

FIGURA 2-14 Configuración de un concentrador USB.



Cuando un dispositivo se conecta a la computadora mediante USB, la computadora consulta (enumera) al dispositivo para obtener su nombre, el tipo de dispositivo y el tipo de controlador de dispositivo que soporta. La computadora puede suspender la energía de cada dispositivo, para colocarlo en un estado suspendido.¹

Puerto paralelo Antes las impresoras se conectaban mediante los *puertos paralelos*. El término *paralelo* indica que los bits en un byte o palabra de datos viajan en forma simultánea, desde la computadora hasta el dispositivo. Los datos se transfieren a una alta velocidad (1MB por segundo) a través de distancias cortas, por lo general, de no más de 10 pies. El DOS reconoce de manera automática tres puertos paralelos: LPT1, LPT2 y LPT3. Los puertos paralelos pueden ser *bidireccionales*, lo cual permite a la computadora enviar datos y recibir información desde y hacia un dispositivo. Aunque ahora muchas impresoras utilizan conectores USB, los puertos paralelos son útiles para las conexiones de alta velocidad con los instrumentos de laboratorio y los dispositivos de hardware personalizados.

IDE Las interfaces IDE, conocidas como *electrónica de unidad inteligente* o *electrónica de dispositivo integrado*, conectan a las computadoras con los dispositivos de almacenamiento masivo, como los discos duros, las unidades de DVD y de CD-ROM. Los dispositivos IDE casi siempre se encuentran dentro de la unidad de sistema de la computadora. La mayoría de los dispositivos IDE hoy en día son en realidad dispositivos ATA (*Tecnología avanzada de conexión*) paralelos, en los que el controlador de la unidad se encuentra en la misma unidad. Los dispositivos con lógica de controlador integrado liberan a la CPU de la computadora de tener que controlar la lógica de las unidades internas. Una interfaz relacionada es SATA (ATA serial), que proporciona mayores velocidades de transferencia de datos que los dispositivos ATA paralelos.

FireWire FireWire es un estándar de bus externo de alta velocidad, el cual soporta velocidades de transferencia de datos de hasta 800MB por segundo. Hay una gran cantidad de dispositivos que pueden conectarse a un solo bus FireWire, y los datos pueden entregarse a una velocidad garantizada (transferencia de datos *síncrona*).

Puerto serial Un *puerto serial RS-232* envía los bits binarios uno a la vez, con una velocidad más lenta que los puertos paralelo y USB, pero tiene la habilidad de enviar datos a través de mayores distancias. La velocidad de transferencia de datos más alta es de 19,200 bits por segundo. Los dispositivos de adquisición de laboratorio utilizan con frecuencia interfaces en serie, al igual que el módem telefónico. El chip UART (*Transmisor-receptor asíncrono universal*) 16550 controla la transferencia de datos en serie.

2.4.5 Repaso de sección

1. Describa la SRAM y su uso más común.
2. ¿Qué procesador Intel estaba detrás de la creación del bus PCI?
3. En el juego de chips de la tarjeta madre, ¿qué tarea realiza el Intel 8259A?
4. ¿En dónde se encuentra la memoria que utiliza la pantalla de video?
5. Describa el barrido de trama en un monitor de video CRT.
6. Mencione cuatro tipos de RAM que hemos visto en este capítulo.
7. ¿Qué tipo de RAM se utiliza para la memoria caché de Nivel 2?
8. ¿Qué ventajas ofrece un dispositivo USB, en comparación con un dispositivo serial o paralelo estándar?
9. ¿Cuáles son los nombres de los dos tipos de conectores USB?
10. ¿Qué chip del procesador controla el puerto serial?

2.5 Sistema de entrada/salida

¿Desea escribir juegos de computadora? Por lo general, el uso de memoria y de E/S es intensivo, y la computadora se utiliza al máximo. Los programadores expertos en la programación de juegos conocen mucho acerca del hardware de video y sonido, y optimizan su código para aprovechar las características de hardware.

2.5.1 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 operaciones de E/S disponibles en distintos niveles de acceso, de manera similar al concepto de máquina virtual que vimos en el capítulo 1. Existen tres niveles principales:

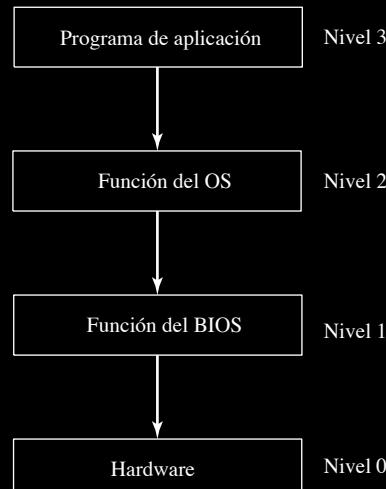
- **Funciones de lenguaje de alto nivel (HLL):** un lenguaje de programación de alto nivel, como C++ o Java, contiene funciones para realizar operaciones de entrada-salida. Estas funciones son portables, ya que trabajan en una variedad de sistemas computacionales distintos, y no dependen de ningún sistema operativo.
- **Sistema operativo:** los programadores pueden hacer llamadas a las funciones del 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, la 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.

Podemos poner a la jerarquía de E/S en perspectiva, mostrando lo que ocurre cuando un programa de aplicación muestra una cadena de caracteres en la pantalla (figura 2-15). Se llevan a cabo los siguientes pasos:

1. Una 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 le pasa un apuntador de cadena.
3. La función del sistema operativo (Nivel 2) utiliza un ciclo para llamar a una subrutina del BIOS, pasárle el código ASCII y el color de cada carácter. El sistema operativo llama a otra subrutina del BIOS para desplazar el cursor a la siguiente posición en la pantalla.
4. La subrutina del 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 píxeles.

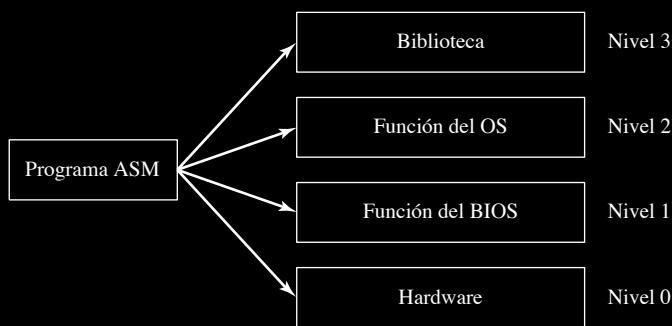
FIGURA 2-15 Niveles de acceso para las operaciones de entrada-salida.



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 (figura 2-16):

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

FIGURA 2-16 Niveles de acceso del lenguaje ensamblador.



¿Cuáles son las concesiones? La principal es la del control sobre la portabilidad. El nivel 2 (OS) funciona en cualquier computadora que ejecute el mismo sistema operativo. Si un dispositivo de E/S carece de ciertas capacidades, el OS hará su mejor esfuerzo por aproximarse al resultado esperado. El nivel 2 no es en especial rápido, debido a que cada llamada de E/S debe pasar a través de varias capas antes de poder ejecutarse.

El nivel 1 (BIOS) funciona en todos los sistemas que tienen un BIOS estándar, pero no produce el mismo resultado en todos los sistemas. Por ejemplo, dos computadoras pueden tener pantallas de video con distintas capacidades de resolución. Un programador en el Nivel 1 tendría que escribir código para detectar la configuración de hardware del usuario, y ajustar el formato de salida de manera acorde. El Nivel 1 se ejecuta mucho más rápido que el Nivel 2, ya que sólo está a un nivel por encima del hardware.

El nivel 0 (hardware) funciona con los dispositivos genéricos como los puertos seriales, y con dispositivos de E/S específicos que producen los fabricantes reconocidos. Los programas que utilizan este nivel deben extender su lógica de codificación para manejar las variaciones en los dispositivos de E/S. Los programas de juegos en modo real son vivos ejemplos, ya que, por lo general, toman el control de la computadora. Los programas en este nivel se ejecutan con la velocidad que les permita el hardware.

Por ejemplo, suponga que desea reproducir un archivo WAV mediante un dispositivo controlador de audio. En el nivel del OS, no tendría que saber qué tipo de dispositivo está instalado, por lo que no habría que preocuparse por las características no estándar que pudiera tener la tarjeta. En el nivel del BIOS, sondearía la tarjeta de sonido (usando su software controlador de dispositivo instalado) para averiguar si pertenece a cierta clase de tarjetas de sonido que tengan características conocidas. En el nivel del hardware, tendría que optimizar el programa para ciertas marcas de tarjetas de audio, aprovechando las características especiales de cada tarjeta.

Por último, no todos los sistemas operativos permiten que los programas de los usuarios accedan directo al hardware del sistema. Dicho acceso está reservado para el mismo sistema operativo y los programas controladores de dispositivos especializados. Éste es el caso con Windows NT, 2000 y XP, en donde los recursos vitales del sistema están aislados de los programas de aplicación. Por otro lado, MS-DOS no tiene dichas restricciones.

2.5.2 Repaso de sección

1. De los tres niveles de entrada/salida en un sistema computacional, ¿cuál es el más universal y portable?
2. ¿Qué características distinguen a las operaciones de entrada/salida a nivel del BIOS?
3. ¿Por qué son necesarios los controladores de dispositivos, dado que el BIOS ya cuenta con código que se comunica con el hardware de la computadora?
4. En el ejemplo relacionado con la visualización de una cadena de caracteres, ¿qué nivel existe entre el sistema operativo y la tarjeta controladora de video?
5. ¿En qué nivel(es) un programa en lenguaje ensamblador puede manipular las operaciones de entrada/salida?
6. ¿Por qué los programas de juegos envían a menudo su salida de sonido directo a los puertos de hardware de la tarjeta de sonido?
7. *Reto:* ¿es probable que el BIOS para una computadora que ejecuta MS-Windows sea distinto al de una computadora que ejecuta Linux?

2.6 Resumen del capítulo

La unidad central de procesamiento (CPU) es en donde se realizan los cálculos y el procesamiento lógico. Contiene un número limitado de ubicaciones de almacenamiento llamadas *registros*, un reloj de alta frecuencia para sincronizar sus operaciones, una unidad de control, y la unidad aritmética-lógica. La unidad de almacenamiento de memoria es en donde se guardan las instrucciones y los datos, mientras se ejecuta un programa de computadora. Un *bus* es una serie de alambres paralelos que transmiten datos entre las diversas partes de la computadora.

La ejecución de una sola instrucción de máquina puede dividirse en una secuencia de operaciones individuales, conocidas como *ciclo de ejecución de instrucciones*. Las tres operaciones primarias son: búsqueda, decodificación y ejecución. Cada paso en el ciclo de instrucciones ocupa cuando menos un pulso del reloj del sistema, conocido como *ciclo de reloj*. La secuencia *cargar y ejecutar* describe la manera en que el sistema operativo ubica a un programa, lo carga en memoria y lo ejecuta.

La ejecución *canalizada* mejora de forma considerable el desempeño de varias instrucciones en una CPU, al permitir la ejecución traslapada de instrucciones de varias etapas. Un procesador *superescalar* es un procesador canalizado con varias canalizaciones de ejecución. Dicho procesador es muy útil cuando una de las etapas de ejecución requiere varios ciclos de reloj.

Un sistema operativo *multitareas* puede ejecutar varias tareas al mismo tiempo. Se ejecuta en un procesador que soporta la *comutación de tareas*, la habilidad de guardar el estado de la tarea actual y transferir el control a una tarea distinta.

Los procesadores IA-32 tienen tres modos básicos de operación: modo *protigido*, modo de *direcccionamiento real* y modo de *administración del sistema*. Además, el modo *8086 virtual* es un caso especial del modo protegido.

Los *registros* son ubicaciones con nombre dentro de la CPU, a las que se puede acceder con mucha mayor rapidez que la memoria convencional. A continuación se muestran descripciones breves de los tipos de registros:

- Los registros de *propósito general* se utilizan principalmente para las operaciones aritméticas, de movimiento de datos y lógicas.
- Los *registros de segmento* se utilizan como ubicaciones base para áreas preasignadas de memoria llamadas *segmentos*.
- El registro EIP (*apuntador de instrucciones*) contiene la dirección de la siguiente instrucción a ejecutar.
- El registro EFLAGS (*de banderas extendido*) consiste en bits binarios individuales que controlan la operación de la CPU y reflejan el resultado de las operaciones de la ALU.

La familia IA-32 tiene una unidad de punto flotante (FPU) que se utiliza específicamente para ejecutar instrucciones de punto flotante y alta velocidad.

El procesador Intel 8086 marcó el principio de la familia de la arquitectura moderna de Intel. El procesador Intel386, el primero de la familia IA-32, contiene registros de 32 bits, además de un bus de direcciones y

una ruta de datos externa de 32 bits. La familia de procesadores P6 (del Pentium Pro en adelante) se basa en un nuevo diseño de microarquitectura, en el que se mejora la velocidad de ejecución.

Los primeros procesadores Intel para la computadora personal IBM se basaban en el enfoque del *conjunto complejo de instrucciones* (CISC). El conjunto de instrucciones Intel incluye poderosas formas de direccionar datos e instrucciones, que son relativamente operaciones complejas de alto nivel. Un enfoque completamente distinto para el diseño de microprocesadores es el *conjunto reducido de instrucciones* (RISC). Un lenguaje máquina RISC consiste en un número muy pequeño de instrucciones cortas y simples, que el procesador pueda ejecutar con rapidez.

En el modo de direccionamiento real sólo puede direccionarse 1MB de memoria, usando las direcciones hexadecimales de 00000 a FFFFF. En el 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 virtual. En el modo 8086 virtual, la computadora se ejecuta en modo protegido y crea una máquina 8086 virtual, con su propio espacio de direcciones de 1MB que simula a una computadora 80 × 86 ejecutándose en modo de direccionamiento real.

En el modelo de segmentación plano, todos los segmentos se asignan al espacio completo de direcciones físicas de la computadora. En el modelo de varios segmentos, cada tarea recibe su propia tabla de descriptores de segmento, conocida como tabla de descriptores locales (LDT). Los procesadores IA-32 soportan una característica llamada *paginación*, la cual permite dividir un segmento en bloques de memoria de 4096 bytes, conocidos como páginas. La paginación permite que el total de memoria utilizada por todos los programas que se ejecutan al mismo tiempo sea mucho mayor que la memoria actual (física) de la computadora.

El corazón de cualquier microcomputadora es su tarjeta madre, ya que aloja a su CPU, los procesadores de soporte, la memoria principal, los conectores de entrada-salida, los conectores de la fuente de alimentación y las ranuras de expansión. El bus PCI (Interconexión de componentes periféricos) ofrece una ruta de actualización conveniente para los procesadores Pentium. La mayoría de las tarjetas madre contienen un conjunto integrado de varios microprocesadores y controladores, al cual se le conoce como juego de chips. El juego de chips determina en gran parte las capacidades de la computadora.

El adaptador de video controla la visualización de texto y gráficos en las computadoras IBM compatibles. Tiene dos componentes: el controlador de video y la memoria de visualización de video.

En las PCs se utilizan varios tipos básicos de memoria: ROM, EPROM, RAM dinámica (DRAM), RAM estática (SRAM), RAM de video (VRAM) y RAM de CMOS.

El puerto del Bus serial universal (USB) proporciona una conexión inteligente de alta velocidad entre la computadora y los dispositivos con soporte USB. Un puerto paralelo transmite 8 o 16 bits de datos en forma simultánea, de un dispositivo a otro. Un puerto serial RS-232 envía bits binarios, uno a la vez, a velocidades más lentas que las de los puertos paralelo y USB.

Las operaciones de entrada-salida se realizan a través de distintos niveles de acceso, de manera similar al concepto de máquina virtual. El sistema operativo se encuentra en el nivel más alto. El BIOS (Sistema básico de entrada-salida) es una colección de funciones que se comunican directamente con los dispositivos de hardware. Los programas también pueden acceder en forma directa a los dispositivos de entrada-salida.

Nota final

1. Para obtener más información, consulte el artículo *An Introduction to USB Development*, Embedded Systems Programming de Jack G. Ganssle, disponible en el sitio www.embedded.com/2000/0003/0003ia2.htm.

FUNDAMENTOS DEL LENGUAJE ENSAMBLADOR

- 3.1 Elementos básicos del lenguaje ensamblador
 - 3.1.1 Constantes enteras
 - 3.1.2 Expresiones enteras
 - 3.1.3 Constantes numéricas reales
 - 3.1.4 Constantes tipo carácter
 - 3.1.5 Constantes tipo cadena
 - 3.1.6 Palabras reservadas
 - 3.1.7 Identificadores
 - 3.1.8 Directivas
 - 3.1.9 Instrucciones
 - 3.1.10 La instrucción NOP (ninguna operación)
 - 3.1.11 Repaso de sección
- 3.2 Ejemplo: suma y resta de enteros
 - 3.2.1 Versión alternativa de SumaResta
 - 3.2.2 Plantilla de programa
 - 3.2.3 Repaso de sección
- 3.3 Ensamblado, enlazado y ejecución de programas
 - 3.3.1 El ciclo de ensamblado-enlazado-ejecución
 - 3.3.2 Repaso de sección
- 3.4 Definición de datos
 - 3.4.1 Tipos de datos intrínsecos
 - 3.4.2 Instrucción de definición de datos
- 3.4.3 Definición de datos BYTE y SBYTE
- 3.4.4 Definición de datos WORD y SWORD
- 3.4.5 Definición de datos DWORD y SDWORD
- 3.4.6 Definición de datos QWORD
- 3.4.7 Definición de datos TBYTE
- 3.4.8 Definición de datos de números reales
- 3.4.9 Orden Little Endian
- 3.4.10 Agregar variables al programa SumaResta
- 3.4.11 Declaración de datos sin inicializar
- 3.4.12 Repaso de sección
- 3.5 Constantes simbólicas
 - 3.5.1 Directiva de signo de igual
 - 3.5.2 Cálculo de los tamaños de los arreglos y cadenas
 - 3.5.3 Directiva EQU
 - 3.5.4 Directiva TEXTEQU
 - 3.5.5 Repaso de sección
- 3.6 Programación en modo de direccionamiento real (opcional)
 - 3.6.1 Cambios básicos
- 3.7 Resumen del capítulo
- 3.8 Ejercicios de programación

3.1 Elementos básicos del lenguaje ensamblador

Existe algo de verdad al decir que “*El lenguaje ensamblador es simple*”. Este lenguaje se diseñó para ejecutarse en poca memoria y consiste principalmente en operaciones sencillas de bajo nivel. Entonces ¿por qué tiene la reputación de ser difícil de aprender? Después de todo, ¿qué tan difícil puede ser mover datos de un registro

a otro y realizar un cálculo? He aquí una prueba del concepto, un programa simple en lenguaje ensamblador que suma dos números y muestra el resultado:

```
main PROC
    mov    eax,5          ; mueve 5 al registro EAX
    add    eax,6          ; suma 6 al registro EAX
    call   EscribeInt    ; muestra el valor en EAX
    exit
main ENDP
```

Simplificamos un poco las cosas al llamar a una subrutina de biblioteca llamada **EscribeInt**, la cual contiene en sí una cantidad considerable de código. Pero en general, el lenguaje ensamblador no es difícil de aprender si usted puede escribir sin problema programas cortos que prácticamente no hacen nada.

Detalles, detalles Para convertirse en un programador experimentado en lenguaje ensamblador se requieren muchos detalles. Primero cree una base de información fundamental y poco a poco vaya llenando los detalles, hasta que tenga algo sólido. En el capítulo 1 se presentaron los conceptos numéricos y las máquinas virtuales. En el capítulo 2 se presentaron los fundamentos de hardware. Ahora está listo para empezar a programar. Si fuera cocinero, le mostraríamos la cocina y le explicaríamos cómo utilizar las licuadoras, las moledoras, los cuchillos, los hornos y los sartenes. De manera similar, vamos a identificar los ingredientes del lenguaje ensamblador, los mezclaremos y cocinaremos unos cuantos programas con buen sazón.

3.1.1 Constantes enteras

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

[{+|-}] *dígitos* [*raíz*]

En este capítulo se utiliza la notación de sintaxis de Microsoft. Los elementos dentro de los corchetes [...] son opcionales, y los elementos dentro de las llaves {} requieren una elección de uno de los elementos en su interior (separados por el carácter |). Los elementos en *cursiva* destacan los elementos que tienen definiciones o descripciones conocidas.

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

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

Si no se da una raíz, se asume que la constante entera es decimal. He aquí algunos ejemplos que utilizan distintas raíces:

26	Decimal	420	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 la interprete como un identificador.

3.1.2 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). En la tabla 3-1 se presentan los operadores aritméticos de acuerdo con su orden de precedencia, de mayor (1) a menor (4).

Tabla 3-1 Operadores aritméticos.

Operador	Nombre	Nivel de precedencia
()	Paréntesis	1
+, -	Unario positivo, unario negativo	2
*, /	Multiplicación, división	3
MOD	Módulo	3
+, -	Suma, resta	4

La *precedencia* 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

Los siguientes son ejemplos de expresiones válidas y sus valores:

Expresión	Valor
16 / 5	3
-(3 + 4) * (6 - 1)	-35
-3 + 4 * 6 - 1	20
25 mod 3	1

Use paréntesis en las expresiones para ayudar a clarificar el orden de las operaciones, de manera que no tenga que recordar las reglas de precedencia.

3.1.3 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 mostramos 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 para los reales cortos (vea el capítulo 17). Por ejemplo, la representación binaria del número +1.0 decimal es:

```
0011 1111 1000 0000 0000 0000 0000 0000
```

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

```
3F800000r
```

3.1.4 Constantes 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 son:

```
'A'  
"d"
```

En la guarda de este libro aparece impresa una lista completa de códigos ASCII.

3.1.5 Constantes tipo cadena

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

```
'ABC'  
'X'  
"Buenas noches, Gracie"  
'4096'
```

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

```
"Ésta no es una prueba"  
'Diga "Buenas noches", Gracie'
```

3.1.6 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, como MOV, ADD y MUL.
- Directivas, las cuales indican a MASM cómo ensamblar programas.
- Atributos, que proporcionan información acerca del tamaño y uso de las variables y operandos. Dos ejemplos son BYTE y WORD.
- Operadores, que se utilizan en expresiones constantes.
- Símbolos predefinidos como @data, que devuelven valores enteros constantes en tiempo de ensamblado.

En el apéndice A encontrará una lista completa de las palabras reservadas de MASM.

3.1.7 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. Considere 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 (A..Z, a..z), guión bajo (_), @, ?, o \$. Los caracteres subsiguientes también pueden ser dígitos.
- Un identificador no puede ser igual que una palabra reservada para el lenguaje ensamblador.

Para hacer que todas las palabras clave y los identificadores sean sensibles a mayúsculas/minúsculas, agregue el modificador de línea de comandos –Cp cuando 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 usted cree. Utilice nombres descriptivos y fáciles de entender para sus identificadores. He aquí algunos identificadores válidos:

var1	Cuenta	\$primero
_main	MAX	archivo_abierto
@@miarchivo	valorX	_12345

3.1.8 Directivas

Una *directiva* es un comando incrustado en el código 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 que las directivas 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 NASM (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
```

El apéndice A es una referencia útil para las directivas y operadores de MASM.

3.1.9 Instrucciones

Una *instrucción* es un enunciado que se vuelve ejecutable cuando se ensambla un programa. El ensamblador traduce las instrucciones en bytes 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 instrucción (requerido).
- Operando(s) (por lo general, son requeridos).
- Comentario (opcional).

Ésta es la sintaxis básica:

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

Vamos a explorar cada parte por separado, empezando con el campo *etiqueta*.

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 que se coloca justo antes de una variable, representa 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 le siguen en la memoria van inmediatamente después:

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

En la sección 3.4.2 explicaremos el uso de las variables, y en la sección 4.1.4 explicaremos el uso de la instrucción MOV.

Etiquetas de código Una etiqueta en el área de 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 destinos de las instrucciones de saltos y de ciclos. Por ejemplo, la siguiente instrucción JMP (salto) transfiere el control a la ubicación marcada por la etiqueta llamada destino, con lo cual se crea 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 sí 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 que vimos en la sección 3.1.7. Los nombres de las etiquetas de datos deben ser únicos dentro del mismo archivo de código fuente; las etiquetas de código sólo deben ser únicas dentro del mismo procedimiento.

Nemónico 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 nemáticos de instrucciones en el lenguaje ensamblador, como mov, add y sub, proporcionan sugerencias acerca del tipo de operación que realizan:

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. En el capítulo 2 hablamos sobre los nombres de los registros, y en la sección 3.1.2 sobre las expresiones constantes. Un *operando de memoria* se especifica mediante el nombre de una variable o mediante uno o más registros que contengan la dirección de una variable. El nombre de una variable indica la dirección de ésta, e instruye a la computadora para que haga referencia al contenido de la memoria en la dirección dada. La siguiente tabla contiene varios operandos de ejemplo:

Ejemplo	Tipo de operando
96	Constante (<i>valor inmediato</i>)
2 + 4	Expresión constante
eax	Registro
cuenta	Memoria

A continuación se muestran ejemplos de instrucciones en lenguaje ensamblador que tienen números variables de operandos. Por ejemplo, la instrucción STC no tiene operandos:

```
stc ; activa la bandera Acarreo
```

La instrucción INC tiene un operando:

```
inc eax ; suma 1 a EAX
```

La instrucción MOV tiene dos operandos:

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

En una instrucción con dos operandos, al primero se le llama el *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 fuente. Con frecuencia se incluye la siguiente información en la parte superior del listado de un programa:

- La descripción del 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 la directiva COMMENT y un símbolo especificado por el usuario. El ensamblador ignora todas las líneas subsiguientes de texto, hasta que aparezca el mismo símbolo especificado por el usuario. Por ejemplo,

```
COMMENT !
  Esta linea es un comentario.
  Esta linea también es un comentario.
!
```

También podemos usar cualquier otro símbolo:

```
COMMENT &
  Esta linea es un comentario.
  Esta linea también es un comentario.
&
```

3.1.10 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 la utilizan para alinear el código con los límites de las direcciones pares. En el siguiente ejemplo, la instrucción MOV genera tres bytes de código máquina. La instrucción NOP alinea la dirección de la tercera instrucción con un límite de doble palabra (múltiplo par de 4):

```
00000000 66 8B C3 mov 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.

3.1.11 Repaso de sección

1. Identifique los caracteres de sufijo válidos que se utilizan en constantes enteras.
2. (Sí/No): ¿A5h es una constante hexadecimal válida?
3. (Sí/No): ¿el signo de multiplicación (*) tiene mayor precedencia que el signo de división (/) en las expresiones enteras?
4. Escriba una expresión constante que divida 10 entre 3 y devuelva el residuo entero.
5. Muestre un ejemplo de una constante numérica real válida con un exponente.
6. (Sí/No): ¿las constantes de cadena deben encerrarse entre comillas sencillas?
7. Las palabras reservadas pueden ser nemónicos de instrucciones, atributos, operadores, símbolos predefinidos y _____.
8. ¿Cuál es la longitud máxima de un identificador?
9. (Verdadero/Falso): un identificador no puede empezar con un dígito numérico.
10. (Verdadero/Falso): los identificadores en lenguaje ensamblador son (de manera predeterminada) insensibles al uso de mayúsculas/minúsculas.
11. (Verdadero/Falso): las directivas de ensamblador se llevan a cabo en tiempo de ejecución.
12. (Verdadero/Falso): las directivas de ensamblador pueden escribirse en cualquier combinación de letras mayúsculas y minúsculas.
13. Mencione las cuatro partes básicas de una instrucción en lenguaje ensamblador.
14. (Verdadero/Falso): MOV es un ejemplo de un nemónico de instrucción.
15. (Verdadero/Falso): una etiqueta de código va seguida de un signo de dos puntos (:), pero una etiqueta de datos no tiene un signo de dos puntos.
16. Muestre un ejemplo de un comentario de bloque.
17. ¿Por qué no es conveniente utilizar direcciones numéricas al escribir instrucciones que acceden a variables?

3.2 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 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. En cada caso, el código del programa aparece antes de su explicación:

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

La directiva TITLE marca toda la línea como un comentario. Puede poner lo que quiera 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 de punto y coma, así que lo utilizamos para los comentarios.

```
INCLUDE Irvine32.inc
```

La directiva INCLUDE copia las definiciones necesarias y la información de configuración de un archivo de texto llamado *Irvine32.inc*, ubicado en el directorio INCLUDE del ensamblador (en el capítulo 5 describiremos este archivo).

```
.code
```

La directiva .code marca el inicio del *segmento de 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 SUB resta 20000h del registro EAX.

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

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

```
    exit
main ENDP
```

La instrucción **exit** llama (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 en *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).

Resultados del programa A continuación se muestra una salida en pantalla de los resultados del programa, generados por la llamada a DumpRegs:

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

Las primeras dos filas de resultados muestran los valores hexadecimales de los registros de propósito general de 32 bits. EAX equivale a 00030000h, el valor producido por las instrucciones ADD y SUB en el programa. La tercera fila muestra los valores de los registros EIP (apuntador de instrucciones extendido) y EFL (de banderas extendido), así como los valores de las banderas Acarreo, Signo, Cero, Desbordamiento, Acarreo auxiliar y Paridad.

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 de código contiene uno o más procedimientos, en donde uno de ellos se designa como el procedimiento de *arranque*.

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

Estilos 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 del código fuente. Para su fácil lectura, debemos ser consistentes en el uso de mayúsculas y minúsculas, así como en los nombres que asignemos a los identificadores. A continuación se muestran algunas metodologías relacionadas con el uso de mayúsculas y minúsculas que se aconseja adoptar:

- Utilice minúsculas para las palabras clave, mayúsculas y minúsculas para los identificadores, y sólo mayúsculas para las constantes. Esta metodología sigue el modelo general de C, C++ y Java.
- Utilice mayúsculas en todo. Esta metodología se utilizó en el software anterior a la década de 1970, cuando muchas terminales de computadora no tenían soporte para 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.
- Use mayúsculas para las palabras reservadas en el lenguaje ensamblador, incluyendo los nemáticos de instrucciones y los nombres de los registros. Esta metodología nos ayuda a diferenciar entre los identificadores y las palabras reservadas.
- Use mayúsculas en las directivas y los operadores de lenguaje ensamblador, use una mezcla de mayúsculas y minúsculas para los identificadores, y minúsculas para todo lo demás. En este libro empleamos esta metodología, con la excepción de que utilizamos minúsculas para las directivas .code, .stack, .model y .data.

3.2.1 Versión alternativa de SumaResta

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

```
TITLE Suma y resta          (SumaRestaAlt.asm)
; Este programa suma y resta enteros de 32 bits.

.386
.model flat,stdcall
.stack 4096
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
    INVOKE ExitProcess,0
main ENDP
END main
```

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

.386

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

.model flat,stdcall

La directiva .MODEL instruye al 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 pasa un código de retorno de cero. INVOKE es una directiva del ensamblador, que llama a un procedimiento o a una función.

3.2.2 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, marcando los puntos en donde usted debe agregar su propio código:

```
TITLE Plantilla de programa      (plantilla.asm)
; Descripción del programa:
; Autor:
; Fecha de creación:
; Revisiones:
; Fecha de última modificación:
INCLUDE Irvine32.inc
; (aquí se insertan las definiciones de símbolos)
.data
    ; (aquí se insertan las variables)
.code
main PROC
    ; (aquí se insertan las instrucciones ejecutables)
    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 muy 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 (incluyéndolo a usted, 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. Si está tomando un curso de programación, tal vez su instructor insista en que debe agregar información adicional.

3.2.3 Repaso de sección

1. En el programa SumaResta (sección 3.2), ¿cuál es el significado de la directiva INCLUDE?
2. En el programa SumaResta, ¿qué identifica la directiva .CODE?
3. ¿Cuáles son los nombres de los segmentos en el programa SumaResta?

4. En el programa SumaResta, ¿cómo se muestran los registros de la CPU?
5. En el programa SumaResta, ¿qué instrucción detiene la ejecución del programa?
6. ¿Qué directiva empieza un procedimiento?
7. ¿Qué directiva termina un procedimiento?
8. ¿Cuál es el propósito del identificador en la instrucción END?
9. ¿Qué es lo que hace la directiva PROTO?

3.3 Ensamblado, enlazado y ejecución de programas

En los dos capítulos anteriores, vimos ejemplos de programas simples en lenguaje máquina, por lo que está claro que un programa de código fuente escrito 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 todavía listo para ejecutarse. Debe pasarse a otro programa llamado *enlazador*, que a su vez produce un *archivo ejecutable*. Este archivo está listo para ejecutarse desde MS-DOS/Windows.

3.3.1 El ciclo de ensamblado-enlazado-ejecución

El proceso de editar, ensamblar, enlazar y ejecutar programas en lenguaje ensamblador se resume en la figura 3-1. A continuación presentamos una descripción detallada de cada paso.

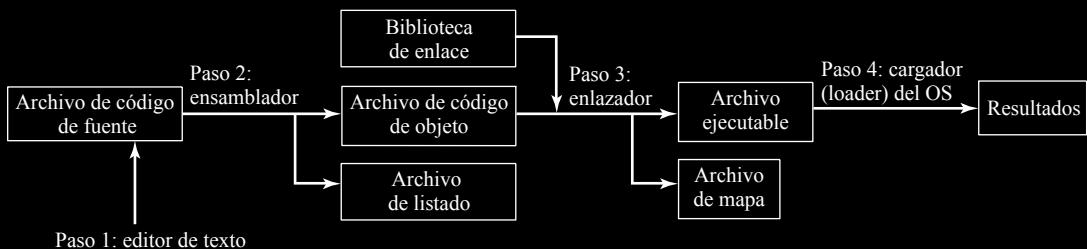
Paso 1: un programador utiliza un **editor de texto** para crear un archivo de texto ASCII, conocido como *archivo de código fuente*.

Paso 2: el **ensamblador** lee el archivo de código fuente y produce un *archivo de código objeto*, una traducción del programa a lenguaje máquina. De manera opcional, produce un *archivo de listado*. Si ocurre un error, el programador debe regresar al paso 1 y corregir el programa.

Paso 3: el **enlazador** lee el archivo de código objeto y verifica si el programa contiene alguna llamada a los procedimientos en una biblioteca de enlace. El **enlazador** copia cualquier procedimiento requerido de la biblioteca de enlace, lo combina con el archivo de código objeto y produce el *archivo ejecutable*. De manera opcional, el enlazador puede producir un *archivo de mapa*.

Paso 4: la herramienta **cargador (loader)** del sistema operativo lee el archivo ejecutable y lo carga en memoria, y bifurca la CPU hacia la dirección inicial del programa, para que éste empiece a ejecutarse.

FIGURA 3-1 Ciclo de ensamblado-enlazado-ejecución.



En el sitio Web del libro podrá consultar las instrucciones detalladas acerca del proceso de ensamblar, enlazar y ejecutar programas en lenguaje ensamblador, mediante el uso de Microsoft Visual C++ 2005 Express.

Archivo de listado

Un archivo de *listado* contiene una copia del código fuente del programa, listo para imprimirse, con números de línea, direcciones de desplazamiento, el código de máquina traducido y una tabla de símbolos. Veamos el archivo de listado para el programa **SumaResta** que creamos en la sección 3.2:

```

Microsoft (R) Macro Assembler Version 8.00
Suma y resta (SumaResta.asm)           Page 1 - 1

TITLE Suma y resta                   (SumaResta.asm)
; Este programa suma y resta enteros de 32 bits.

INCLUDE Irvine32.inc
C      ; Include file for Irvine32.lib (Irvine32.inc)
C      INCLUDE SmallWin.inc

00000000    .code
00000000    main PROC
00000000    B8 00010000    mov eax,10000h ; EAX = 10000h
00000005    05 00040000    add eax,40000h ; EAX = 50000h
0000000A    2D 00020000    sub eax,20000h ; EAX = 30000h
0000000F    E8 0000000E    call DumpRegs ; muestra los registros

        exit
0000001B    main ENDP
        END main

Structures and Unions: (se omitió)

Segments and Groups:
Name          Size   Length   Align  Combine Class
FLAT . . . . . GROUP
STACK. . . . . 32 Bit  00001000  Para   Stack   'STACK'
_DATA. . . . . 32 Bit  00000000  Para   Public 'DATA'
_TEXT. . . . . 32 Bit  0000001B  Para   Public 'CODE'

Procedures, parameters, and locals: (se abrevió la lista)
Name          Type Value Attr
CloseHandle. . . . P Near 00000000 FLAT Length=00000000 External STDCALL
Clrscr . . . . . P Near 00000000 FLAT Length=00000000 External STDCALL
.

.

Main . . . . . P Near 00000000 _TEXT Length=0000001B Public STDCALL

Symbols: (se abrevió la lista)
Name          Type Value Attr
@CodeSize . . . . . Number 00000000h
@DataSize . . . . . Number 00000000h
@Interface . . . . . Number 00000003h
@Model . . . . . Number 00000007h
@code . . . . . Text  _TEXT
@data . . . . . Text  FLAT
@fardata? . . . . . Text  FLAT
@fardata . . . . . Text  FLAT
@stack . . . . . Text  FLAT
.

.

exit . . . . . . . . . Text  INVOKE ExitProcess,0
0 Warnings
0 Errors

```

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:

- El nombre del módulo, utilizado como el nombre base del archivo EXE que produce el enlazador.
- La etiqueta de hora y fecha del encabezado del archivo del programa (no del sistema de archivos).
- Una lista de grupos de segmentos, que contiene la dirección inicial, la longitud, el nombre y la clase de cada grupo.
- Una lista 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 fuente del programa, sus datos, la pila en tiempo de ejecución y demás información de utilidad.

3.3.2 Repaso de sección

1. ¿Qué tipos de archivos produce el ensamblador?
2. (Verdadero/Falso): el enlazador extrae los procedimientos ensamblados de la biblioteca de enlace y los inserta en el programa ejecutable.
3. (Verdadero/Falso): cuando se modifica el código fuente de un programa, debe ensamblarse y enlazarse de nuevo para poder ejecutarlo con las modificaciones.
4. ¿Qué componente del sistema operativo lee y ejecuta programas?
5. ¿Qué tipos de archivos produce el enlazador?

3.4 Definición de datos

3.4.1 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 dado. La característica esencial de cada tipo es su tamaño en bits: 8, 16, 32, 48, 64 y 80. Las demás características (por ejemplo: con signo, apuntador o de punto flotante) son opcionales y su propósito principal es para beneficio de los programadores que desean 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.

En la tabla 3-2, 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.

3.4.2 Instrucción de definición de datos

Una *instrucción* o *estatuto* de definición de datos separa espacio de almacenamiento en 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 (tabla 3-2). Una definición de datos tiene la siguiente sintaxis:

[*nombre*] *directiva inicializador* [, *inicializador*] ...

Tabla 3-2 Tipos de datos intrínsecos.

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 [también puede ser un apuntador cercano (near) en modo protegido]
SDWORD	Entero de 32 bits con signo
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)

Nombre El nombre opcional que se asigna a una variable debe apegarse a las reglas para los identificadores (sección 3.1.7).

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 3-2. Además, puede ser cualquiera de las directivas de definición de datos heredadas que se muestran en la tabla 3-3, que también soportan los ensambladores NASM y TASM.

Tabla 3-3 Directivas de datos heredadas.

Directiva	Uso
DB	Entero de 8 bits
DW	Entero de 16 bits
DD	Entero o real de 32 bits
DQ	Entero o 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, *inicializador*

es una constante o expresión entera, que coincide con el tamaño del tipo de la variable, como BYTE o WORD. Si usted prefiere dejar la variable sin inicializar (que se le asigne un valor aleatorio), puede usar 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.

3.4.3 Definición de datos BYTE y SBYTE

Las directivas 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. Por ejemplo,

```
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 de 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 1 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
val2 DB -128            ; byte con signo
```

Múltiples inicializadores

Si se utilizan varios inicializadores en la misma definición de datos, su etiqueta sólo hace referencia al desplazamiento del primer inicializador. En el siguiente ejemplo, asuma que **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 0003:

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

La siguiente ilustración muestra a **lista** como una secuencia de bytes, cada uno con su propio desplazamiento:

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 los 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 a 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 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 "envíeme 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 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. Es especial, es útil cuando se asigna 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"

3.4.4 Definición de datos WORD y SWORD

Las directivas 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 del 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
```

3.4.5 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
```

También puede usarse la directiva DD heredada:

```
val1 DD 12345678h ; sin signo
val2 DD -2147483648 ; con 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	Valores
0000:	1
0004:	2
0008:	3
000C:	4
0010:	5

3.4.6 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
```

3.4.7 Definición de datos TBYTE

La directiva TBYTE (define diez 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 1000000000123456789Ah
```

También puede utilizarse la directiva DT heredada:

```
val1 DT 1000000000123456789Ah
```

3.4.8 Definición de datos de números reales

REAL4 define a una variable real de 4 bytes y precisión simple. REAL8 define a un real de 8 bytes y precisión doble, y REAL10 define a un real de 10 bytes y doble precisión extendida. Cada uno requiere de uno o más inicializadores de 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	1.18×10^{-38} a 3.40×10^{38}
Real largo	15	2.23×10^{-308} a 1.79×10^{308}
Real de precisión extendida	19	3.37×10^{-4932} a 1.18×10^{4932}

Las directivas 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

3.4.9 Orden LittleEndian

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 12345678h. Si se coloca en el desplazamiento 0000 de la memoria, se almacenaría 78h en el primer byte, 56h en el segundo byte, y el resto de los bytes estarían en los desplazamientos 0003 y 0004:

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

Little endian

Hay otros sistemas computacionales 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
0002:	56
0003:	78

Big endian

3.4.10 Agregar variables al programa SumaResta

Vamos a utilizar el programa **SumaResta** de la sección 3.2 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)
```

3.4.11 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
arregloPequeno 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
arregloPequeno DWORD 10 DUP(0) ; 40 bytes
arregloGrande DWORD 5000 DUP(?) ; 20,000 bytes
```

Mezcla de código y datos El ensamblador nos permite cambiar entre el 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 las instrucciones ejecutables, MASM coloca a **temp** en el segmento de datos, separada del segmento que guarda el código compilado.

3.4.12 Repaso de sección

1. Cree una declaración de datos sin inicializar para un entero de 16 bits con signo.
2. Cree una declaración de datos sin inicializar para un entero de 8 bits sin signo.
3. Cree una declaración de datos sin inicializar para un entero de 8 bits con signo.
4. Cree una declaración de datos sin inicializar para un entero de 64 bits.
5. ¿Qué tipo de datos puede almacenar un entero de 32 bits con signo?
6. Declare una variable entera de 32 bits con signo e inicialícela con el valor decimal negativo más pequeño que sea posible (*Sugerencia*: consulte los rangos de los enteros en el capítulo 1).
7. Declare una variable entera de 16 bits sin signo, llamada **arregloW**, que utilice tres inicializadores.
8. Declare una variable de cadena que contenga el nombre de su color favorito. Inicialícela como una cadena con terminación nula.
9. Declare un arreglo sin inicializar de 50 dobles palabras sin signo, llamado **arregloD**.
10. Declare una variable de cadena que contenga la palabra “PRUEBA” repetida 500 veces.
11. Declare un arreglo de 20 bytes sin signo, llamado **arregloB**, e inicialice todos los elementos a cero.
12. Muestre el orden de los bytes individuales en la memoria (de menor a mayor) para la siguiente variable tipo doble palabra:

```
val1 DWORD 87654321h
```

3.5 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	Sí
¿Cambia su valor en tiempo de ejecución?	No	Sí

Le 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.

3.5.1 Directiva de signo de igual

La directiva de *signo de igual* asocia el nombre de un símbolo con una expresión entera (consulte la sección 3.1.2). La sintaxis es

nombre = *expresión*

Por lo común, *expresión* es un valor entero de 32 bits. 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 sólo codificar la instrucción MOV con el valor literal 500, pero la experiencia nos ha demostrado que es más fácil leer y mantener los programas si se utilizan símbolos. Suponga que CUENTA se utilizara 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 fuente del programa. ¿Qué pasaría si una ocurrencia del número 500 no estuviera relacionada con todas las demás? Entonces se produciría un error si se cambiara por 600.

Definiciones del teclado A menudo, los programas definen símbolos para los 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. Utilice:

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

en vez de

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

Uso del operador DUP La sección 3.4.3 mostró cómo utilizar el operador DUP para crear almacenamiento para los arreglos y cadenas. 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 a1,CUENTA ; AL = 5
CUENTA = 10
mov a1,CUENTA ; AL = 10
CUENTA = 100
mov a1,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 fuente que hace el ensamblador.

3.5.2 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 calcule la **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 numero de caracteres"
miCadena_longitud = ($ - miCadena)
```

Arreglos de palabras y dobles palabras Al calcular el número de elementos en un arreglo que contiene palabras de 16 bits, 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 del arreglo:

```
Lista DWORD 10000000h,20000000h,30000000h,40000000h
TamLista = ($ - lista) / 4
```

3.5.3 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 (consulte la sección 3.1.2). 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 se encuentra a *nombre* más adelante en el programa, sustituye el valor entero o el 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 del símbolo:

```
oprímaTecla EQU <"Oprima cualquier tecla para continuar...",0>
.
.
.data
Indicador BYTE oprímaTecla
```

Ejemplo Suponga 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 a un símbolo se le asigne sin querer un nuevo valor.

3.5.4 Directiva TEXTEQU

La directiva TEXTEQU, similar a EQU, crea lo que se conoce como una *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 la macro de texto **msjContinuar**:

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

```
.data
Indicador1 BYTE msgContinuar
```

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.

3.5.5 Repaso de sección

1. Declare una constante simbólica, utilizando la directiva de signo igual que contenga el código ASCII (08h) para la tecla Retroceso.
2. Declare una constante simbólica llamada **SegundosEnDia**, usando la directiva de signo igual, y asígnele una expresión aritmética que calcule el número de segundos en un periodo de 24 horas.
3. Escriba una instrucción que haga que el ensamblador calcule el número de bytes en el siguiente arreglo, y asigne el valor a una constante simbólica llamada **TamArreglo**:

```
miArreglo WORD 20 DUP(?)
```

4. Muestre cómo calcular el número de elementos en el siguiente arreglo, y asigne el valor a una constante simbólica llamada **TamArreglo**:

```
miArreglo DWORD 30 DUP(?)
```

5. Utilice una expresión TEXTEQU para redefinir a “PROC” como “PROCEDIMIENTO”.
6. Utilice TEXTEQU para crear un símbolo llamado **Ejemplo** para una constante de cadena, y después utilice el símbolo para definir a una variable de cadena llamada **MiCadena**.
7. Utilice TEXTEQU para asignar el símbolo **EstablerESI** a la siguiente línea de código:

```
mov esi,OFFSET miArreglo
```

3.6 Programación en modo de 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 de direccionamiento real utilizan segmentos de 16 bits y siguen el esquema de direccionamiento segmentado que vimos en la sección 2.3.1. Si usted utiliza un procesador IA-32, aún puede utilizar los registros de 32 bits de propósito general para los datos.

3.6.1 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 de 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
```

- Consulte el sitio Web del libro para obtener instrucciones sobre cómo ensamblar programas de 16 bits.
- Los desplazamientos (direcciones) de las etiquetas de datos y de código son de 16 bits.

No 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
```

3.7 Resumen del capítulo

Una expresión entera es una expresión matemática que involucra constantes enteras, constantes simbólicas y operadores aritméticos. La *precedencia* se refiere al orden implícito de las operaciones cuando una expresión contiene dos o más operadores.

Una *constante de tipo carácter* es un solo carácter encerrado entre comillas. El ensamblador convierte un carácter en un byte que contiene el código ASCII binario de ese carácter. Una *constante de cadena* es una secuencia de caracteres encerrados entre comillas, que puede o no terminar con un byte nulo.

El lenguaje ensamblador tiene un conjunto de *palabras reservadas* con significados especiales, que sólo pueden utilizarse en el contexto apropiado. Un *identificador* es un nombre elegido por el programador que identifica a una variable, una constante simbólica, un procedimiento o una etiqueta de código. Los identificadores no pueden ser palabras reservadas.

Una *directiva* es un comando incrustado en el código fuente, que el ensamblador interpreta. Una *instrucción* es un enunciado de código fuente, que el procesador realiza en tiempo de ejecución. Un *nemónico de instrucción* es una palabra clave corta que identifica a la operación que una instrucción lleva a cabo. Una *etiqueta* es un identificador que actúa como marcador de posición para las instrucciones o datos.

Los *operandos* son valores que se pasan a las instrucciones. Una instrucción en lenguaje ensamblador puede tener entre cero y tres operandos, cada uno de los cuales puede ser un registro, operando de memoria, expresión constante o número de puerto de E/S.

Los programas contienen *segmentos lógicos* llamados código, datos y pila. El segmento de código contiene instrucciones ejecutables. El segmento de pila almacena parámetros de procedimientos, variables locales y direcciones de retorno. El segmento de datos almacena variables.

Un *archivo de código fuente* contiene instrucciones en lenguaje ensamblador. Un *archivo de listado* contiene una copia del código fuente del programa, adecuado para su impresión, con números de línea, direcciones de desplazamiento, código máquina traducido y una tabla de símbolos. Un *archivo de mapa* contiene información acerca de los segmentos de un programa. Un archivo de código fuente se crea con un editor de texto. Un *ensamblador* es un programa que lee el archivo de código fuente y produce archivos de código objeto y de listado. El *enlazador* es un programa que lee uno o más archivos de código objeto, y produce un archivo ejecutable. Este archivo se ejecuta mediante el cargador (loader) del sistema operativo.

MASM reconoce los tipos de datos intrínsecos, cada uno de los cuales describe un conjunto de valores que pueden asignarse a variables y expresiones del tipo dado:

- BYTE y SBYTE definen variables de 8 bits.
- WORD y SWORD definen variables de 16 bits.
- DWORD y SDWORD definen variables de 32 bits.
- QWORD y TBYTE definen variables de 8 y 10 bytes, respectivamente.
- REAL4, REAL8 y REAL10 definen variables numéricas reales de 4, 8 y 10 bytes, respectivamente.

Una instrucción de definición de datos aparta un espacio de almacenamiento en memoria para una variable, y puede asignarle (de manera opcional) un nombre. Si se utilizan varios inicializadores en la misma definición de datos, su etiqueta se refiere sólo al desplazamiento del primer inicializador. Para crear una definición de datos de cadena, se encierra una secuencia de caracteres entre comillas. El operador DUP genera una asignación de almacenamiento repetida, usando una expresión constante como contador. El operador del contador de ubicación actual (\$) se utiliza en las expresiones para calcular direcciones.

Los procesadores Intel almacenan y recuperan datos de la memoria usando el orden *little endian*: el byte menos significativo de una variable se almacena en su dirección inicial.

Una *constante simbólica* (o definición de símbolo) asocia a un identificador con una expresión entera o de texto. Hay tres directivas para crear constantes simbólicas:

- La directiva del signo de igual (=) asocia el nombre de un símbolo con una expresión entera.
- Las directivas EQU y TEXTEQU asocian un nombre simbólico con una expresión entera o algún texto arbitrario.

Podemos convertir casi cualquier programa del modo protegido de 32 bits al modo de direccionamiento real de 16 bits. En el sitio Web de este libro se incluyen dos bibliotecas de enlaces que contienen los mismos nombres de procedimientos para ambos tipos de programas.

3.8 Ejercicios de programación

Los siguientes ejercicios pueden hacerse en modo protegido o en modo de direccionamiento real.

1. Resta de tres enteros

Usando el programa **SumaResta** de la sección 3.2 como referencia, escriba un programa para restar tres enteros, usando sólo registros de 16 bits. Inserte una instrucción **call DumpRegs** para mostrar los valores de los registros.

2. Definiciones de datos

Escriba un programa que contenga una definición de cada tipo de dato listado en la sección 3.4. Inicialice cada variable con un valor que sea consistente con su tipo de dato.

3. Constantes enteras simbólicas

Escriba un programa que defina constantes simbólicas para todos los días de la semana. Cree una variable arreglo que utilice los símbolos como inicializadores.

4. Constantes de texto simbólicas

Escriba un programa que defina nombres simbólicos para varias literales de cadena (caracteres entre comillas). Use cada nombre simbólico en una definición de variable.

4

TRANSFERENCIAS DE DATOS, DIRECCIONAMIENTO Y ARITMÉTICA

- 4.1 Instrucciones de transferencia de datos
 - 4.1.1 Introducción
 - 4.1.2 Tipos de operandos
 - 4.1.3 Operandos directos de memoria
 - 4.1.4 Instrucción MOV
 - 4.1.5 Extensión con cero y con signo de enteros
 - 4.1.6 Instrucciones LAHF y SAHF
 - 4.1.7 Instrucción XCHG
 - 4.1.8 Operandos de desplazamiento directo
 - 4.1.9 Programa de ejemplo (movimientos)
 - 4.1.10 Repaso de sección
- 4.2 Suma y resta
 - 4.2.1 Instrucciones INC y DEC
 - 4.2.2 Instrucción ADD
 - 4.2.3 Instrucción SUB
 - 4.2.4 Instrucción NEG
 - 4.2.5 Implementación de expresiones aritméticas
 - 4.2.6 Banderas afectadas por la suma y la resta
 - 4.2.7 Programa de ejemplo (SumaResta3)
 - 4.2.8 Repaso de sección
- 4.3 Operadores y directivas relacionadas con los datos
 - 4.3.1 Operador OFFSET
- 4.3.2 Directiva ALIGN
- 4.3.3 Operador PTR
- 4.3.4 Operador TYPE
- 4.3.5 Operador LENGTHOF
- 4.3.6 Operador SIZEOF
- 4.3.7 Directiva LABEL
- 4.3.8 Repaso de sección
- 4.4 Direccionamiento indirecto
 - 4.4.1 Operandos indirectos
 - 4.4.2 Arreglos
 - 4.4.3 Operandos indexados
 - 4.4.4 Apuntadores
 - 4.4.5 Repaso de sección
- 4.5 Instrucciones JMP y LOOP
 - 4.5.1 Instrucción JMP
 - 4.5.2 Instrucción LOOP
 - 4.5.3 Suma de un arreglo de enteros
 - 4.5.4 Copia de una cadena
 - 4.5.5 Repaso de sección
- 4.6 Resumen del capítulo
- 4.7 Ejercicios de programación

4.1 Instrucciones de transferencia de datos

4.1.1 Introducción

En este capítulo presentaremos muchos detalles, y destacaremos una diferencia fundamental entre el lenguaje ensamblador y los lenguajes de alto nivel: en el lenguaje ensamblador, uno debe estar al pendiente del almacenamiento de datos y los detalles específicos de la máquina. Los compiladores de lenguajes de alto

nivel como C++ y Java realizan una comprobación de tipos estricta en las variables y las instrucciones de asignación. Los compiladores hacen esto para ayudar a los programadores a evitar errores lógicos relacionados con datos de tipos incorrectos. Por otro lado, los ensambladores proporcionan una tremenda libertad cuando se declaran y se mueven datos. Realizan muy poca comprobación de errores y ofrecen una gran variedad de operadores y expresiones de direcciones. ¿Qué precio debemos pagar por esta libertad? Hay que dominar muchos detalles para poder escribir programas significativos.

Si se toma un tiempo para aprender a conciencia todo el material que presentamos en este capítulo, el resto de la lectura de este libro le será más fácil de comprender. A medida que los programas de ejemplo se vuelvan más complicados, será imprescindible que domine las herramientas fundamentales que presentamos en este capítulo.

4.1.2 Tipos de operandos

En este capítulo presentamos tres tipos de operandos de instrucciones: *inmediatos, de registro y de memoria*. De los tres, sólo el tercero es un poco complicado. La tabla 4-1 presenta una notación simple para los operandos, adaptada de los manuales de la familia IA-32 de Intel. Utilizaremos esta notación a partir de este capítulo para describir la sintaxis de las instrucciones Intel individuales.

Tabla 4-1 Notación de operandos de instrucciones.

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

4.1.3 Operandos directos de memoria

En la sección 3.4 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. Suponga que var1 se ubicó en el 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 podrían escribir programas sólo con direcciones numéricas, los nombres simbólicos como **var1** facilitan el proceso de referenciar 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 a1,[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 los corchetes, sólo los utilizaremos en este libro cuando esté involucrada una expresión aritmética:

```
mov a1,[var1 + 5]
```

(A éste se le conoce como operando de desplazamiento directo, un tema que veremos con detalle en la sección 4.1.8).

4.1.4 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 primero 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 a un registro de segmento.

He aquí una lista de las 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 opciones están disponibles en modo real, con la excepción de que 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 su valor 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 copiarla a una variable o registro. Para las constantes enteras sin signo, consulte la tabla 1-4 del capítulo 1. Para las constantes enteras con signo, consulte la tabla 1-7.

4.1.5 Extensión con cero y con signo de enteros

Copia de 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. Suponga que **cuenta** (sin signo, 16 bits) debe moverse a ECX (32 bits). Podemos establecer ECX en 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,valorConSigno ; ECX = 0000FFF0h (+65520)
```

El valor en 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,0xFFFFFFFFh
mov cx,valorConSigno ; ECX = FFFFFFFF0h (-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 en Intel detectaron este problema al diseñar el procesador Intel386, e introdujeron las instrucciones MOVZX y MOVSX para manejar enteros sin signo y enteros con signo.

Instrucción MOVZX

La instrucción MOVZX (*movez 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 sólo con enteros sin signo. Hay tres variantes:

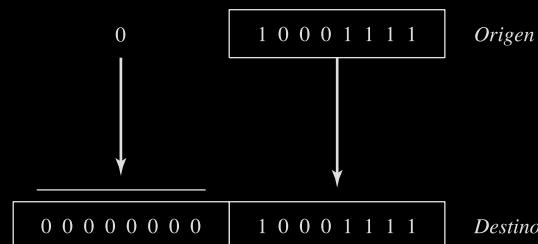
```
MOVZX r32,r/m8
MOVZX r32,r/m16
MOVZX r16,r/m8
```

En la tabla 4-1 explicamos la notación de los operandos. En cada una de las tres variantes, el primer operando (un registro) es el destino y el segundo es el origen. La siguiente instrucción mueve el número binario 10001111 a AX:

```
movzx ax,10001111b
```

La figura 4-1 muestra cómo se extiende con ceros el operando de origen hacia el destino de 16 bits.

FIGURA 4–1 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,b1          ; EDX = 0000009Bh
movzx  cx,b1           ; CX = 009Bh
```

Los siguientes ejemplos utilizan operandos de memoria para el origen y producen los mismos resultados:

```
.data
byte1  BYTE  9Bh
palab1 WORD  0A69Bh
.code
movzx  eax,palab1      ; EAX = 0000A69Bh
movzx  edx,byte1         ; EDX = 0000009Bh
movzx  cx,byte1          ; CX = 009Bh
```

Si desea ejecutar y probar los ejemplos de este capítulo en modo de 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 con 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 variantes:

```
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 se repite (replica) este bit a lo largo de los bits extendidos en el operando de destino. Suponga que el valor de 8 bits 10001111b se mueve a un destino de 16 bits:

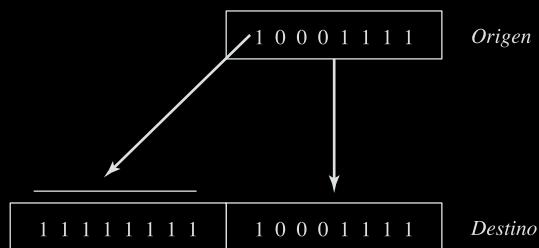
```
movsx  ax,10001111b
```

Los 8 bits inferiores se copian como están (figura 4–2). El bit superior del origen se copia en cada una de las posiciones de los 8 bits superiores del destino.

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

```
mov     bx,0A69Bh
movsx  eax,bx          ; EAX = FFFFA69Bh
movsx  edx,b1          ; EDX = FFFFFFF9Bh
movsx  cx,b1           ; CX = FF9Bh
```

FIGURA 4–2 Diagrama de MOVsx ax,10001111b.



4.1.6 Instrucciones LAHF y SAHF

La instrucción LAHF (cargar banderas de estado 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 las banderas de estado) copia AH al 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 en AH
sahf                         ; las copia al registro Flags
```

4.1.7 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 (sección 4.1.4), 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, utilice un registro como contenedor temporal y combine MOV con XCHG:

```
mov  ax,var1
xchg ax,var2
mov  var1,ax
```

4.1.8 Operandos de desplazamiento directo

Para crear un operando de desplazamiento directo podemos sumar un desplazamiento al nombre de 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 el 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 que 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 a arreglos:

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

Arreglos 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 tipo doble palabra se encuentra a 4 bytes más adelante del primero:

```
.data
arregloD DWORD 10000h, 20000h
.code
mov eax, arregloD ; EAX = 10000h
mov eax, [arregloD+4] ; EAX = 20000h
```

4.1.9 Programa de ejemplo (movimientos)

El siguiente programa demuestra la mayor parte de los ejemplos de transferencias de datos de la sección 4.1:

```
TITLE Ejemplos de transferencias de datos (Moves.asm)
; Ejemplo del capítulo 4. Demostración de MOV y
; XCHG con operando directos y de 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,b1          ; EDX = 0000009Bh
    movzx  cx,b1           ; CX  = 009Bh

; MOVSX
    mov     bx,0A69Bh
    movsx  eax,bx          ; EAX = FFFFA69Bh
    movsx  edx,b1          ; EDX = FFFFFFF9Bh
    mov    b1,7Bh
    movsx  cx,b1           ; 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 palabras):
    mov    ax,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, pero puede (y debe) ejecutarlo mediante un depurador. Consulte los tutoriales en el sitio Web del libro para ver cómo utilizar el depurador de Microsoft Visual Studio. La sección 5.3 explica cómo mostrar enteros usando una biblioteca de funciones incluida en el sitio Web del libro.

4.1.10 Repaso de sección

1. ¿Cuáles son los tres tipos básicos de operandos?
2. (Verdadero/Falso): el operando de destino de una instrucción MOV no puede ser un registro de segmento.
3. (Verdadero/Falso): en una instrucción MOV, el segundo operando se conoce como el operando *de destino*.
4. (Verdadero/Falso): el registro EIP no puede ser el operando de destino de una instrucción MOV.
5. En la notación de operandos utilizada por Intel, ¿qué significa *r/m32*?
6. En la notación de operandos utilizada por Intel, ¿qué significa *imm16*?

Use las siguientes definiciones de variables para el resto de las preguntas en esta sección:

```
.data
var1 SBYTE -4,-2,3,1
var2 WORD 1000h,2000h,3000h,4000h
var3 SWORD -16,-42
var4 DWORD 1,2,3,4,5
```

7. Para cada una de las siguientes instrucciones, indique si es válida o no:

- a. mov ax,var1
- b. mov ax,var2

- c. mov eax,var3
 - d. mov var2,var3
 - e. movzx ax,var2
 - f. movzx var2,al
 - g. mov ds,ax
 - h. mov ds,1000h
8. ¿Cuál será el valor hexadecimal del operando de destino, después de que cada una de las siguientes instrucciones se ejecuten en secuencia?
- | | |
|------------------------------|------|
| <code>mov al,var1</code> | ; a. |
| <code>mov ah,[var1+3]</code> | ; b. |
9. ¿Cuál será el valor del operando de destino, después de que se ejecute cada una de las siguientes instrucciones en secuencia?
- | | |
|------------------------------|------|
| <code>mov ax,var2</code> | ; a. |
| <code>mov ax,[var2+4]</code> | ; b. |
| <code>mov ax,var3</code> | ; c. |
| <code>mov ax,[var3-2]</code> | ; d. |
10. ¿Cuál será el valor del operando de destino, después de que se ejecute cada una de las siguientes instrucciones en secuencia?
- | | |
|-------------------------------|------|
| <code>mov edx,var4</code> | ; a. |
| <code>movzx edx,var2</code> | ; b. |
| <code>mov edx,[var4+4]</code> | ; c. |
| <code>movsx edx,var1</code> | ; d. |

4.2 Suma y resta

La aritmética es un tema bastante extenso en el lenguaje ensamblador, 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.) por estas instrucciones es importante, por lo que hablaremos sobre esto en la sección 4.2.6.

4.2.1 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 Acarreo (lo cual es un poco sorprendente).

4.2.2 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 (sección 4.1.4). 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.

4.2.3 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 las instrucciones ADD y MOV (vea la sección 4.1.4). La sintaxis es:

SUB *dest,origen*

He aquí un ejemplo breve de código que 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:

$$\begin{array}{r}
 \text{Acarreo:} \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \\
 \boxed{ } \quad (4) \\
 + \quad \boxed{1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1} \quad (-1) \\
 \hline
 \boxed{0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1} \quad (3)
 \end{array}$$

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

4.2.4 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*

(Recuerde que para encontrar el complemento a dos de un número se invierten todos los bits 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 del operando de destino.

4.2.5 Implementación de expresiones aritméticas

Armado con las instrucciones ADD, SUB y NEG, tiene 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 tal como:

```
valR = -valX + (valY - valZ);
```

Se utilizarán las siguientes variables de 32 bits:

```
valR SDWORD ?
valX SDWORD 26
valY SDWORD 30
valZ SDWORD 40
```

Al traducir una expresión, evalúe cada término por separado y combine 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**:

```
; segundo término: (valY - valZ)
mov ebx, valY
sub ebx, valZ ; EBX = -10
```

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

```
; se suman los términos y se almacenan:
add eax, ebx
mov valR, eax ; -36
```

4.2.6 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 operando 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 las 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 Acarreo indica un desbordamiento de enteros sin signo. Por ejemplo, si una instrucción tiene un operando de destino de 8 bits, pero la instrucción genera un resultado mayor que el 11111111 binario, se activa la bandera Acarreo.
- La bandera 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 decimal, se activa la bandera 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 del 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 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

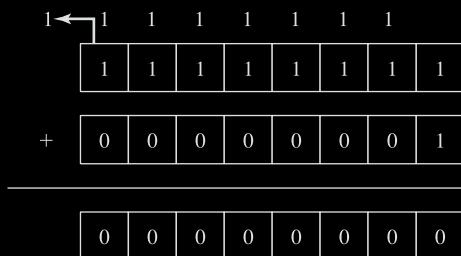
La bandera Cero se activa cuando el resultado de una operación aritmética es cero. Los siguientes ejemplos muestran el estado del registro de destino y de la bandera Cero, después de ejecutar las instrucciones SUB, INC y DEC:

```
mov  ecx,1
sub  ecx,1 ; ECX = 0, ZF = 1
mov  eax,0FFFFFFFh
inc  eax ; EAX = 0, ZF = 1
inc  eax ; EAX = 1, ZF = 0
dec  eax ; EAX = 0, ZF = 1
```

La suma y la bandera Acarreo La operación de la bandera Acarreo es más fácil de explicar si consideramos la suma y la 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 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 un 1 a 0FFh. El acarreo que sale de la posición del bit más alto de AL se copia en la bandera Acarreo:



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 Acarreo Una operación de resta activa la bandera 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 del MSB se invierte y se copia a la bandera Acarreo.

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

$\begin{array}{cccccccc} 0 & \leftarrow \\ \boxed{0} & & 0 & & 0 & & 0 & & 0 & & 0 & & 0 & & 1 \end{array}$	(1)
$+ \begin{array}{cccccccc} 1 & & 1 & & 1 & & 1 & & 1 & & 1 & & 1 & & 0 \end{array}$	(-2)
\hline	
$\begin{array}{cccccccc} 0 & & 0 & & 0 & & 0 & & 0 & & 0 & & 1 & & 1 \end{array}$	(nv)

La suma (255) no es v lida. El acarreo que sale del bit 7 se invierte y se coloca en la bandera Acarreo, por lo que CF = 1. He aqu  el correspondiente c digo 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 Acarreo auxiliar (AC) indica un acarreo o p stamo (borrow) en el bit 3 del operando de destino. Se utiliza principalmente en la aritm tica con n meros decimales codificados en binario (BCD) (secci n 7.6), pero puede usarse en otros contextos. Suponga que sumamos 1 a 0Fh. La suma (10h) contiene un 1 en la posici n del bit 4 que se acarre  de la posici n del bit 3:

```
mov al,0Fh
add al,1 ; AC = 1
```

He aqu  la aritm tica:

$$\begin{array}{r} 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1 \\ + 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1 \\ \hline 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0 \end{array}$$

Paridad La bandera Paridad (PF) se activa cuando el byte menos significativo del destino tiene un n mero 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
```

Despu s de la instrucci n ADD, AL contiene el n mero binario 10001110 (cuatro bits 0 y cuatro bits 1), y PF = 1. Despu s de SUB, AL contiene un n mero 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 operaci n aritm tica con signo es negativo. El siguiente ejemplo resta un entero m s grande (5) de un entero m s peque o (4):

```
mov eax,4
sub eax,5 ; EAX = -1, SF = 1
```

Desde un punto de vista mec nico, 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
sub bl,2 ; BL = 01h
; BL = FFh (-1)
```

Bandera Desbordamiento La bandera Desbordamiento se activa 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 1 se produce un 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 es -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 de 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					
			1	0	0	0	0
CF = 1	←		1	1	1	1	1
			+				
			=				
				0	1	1	1
					1	1	0

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 = 10000000b
neg al ; AL = 10000000b, OF = 1
```

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

```
mov al,+127 ; AL = 0111111b
neg al ; AL = 10000001b, OF = 0
```

¿Cómo sabe la CPU cuando una operación aritmética es con signo o sin signo? Sólo podemos brindarle lo que parece una respuesta tonta: ¡No lo sabe! La CPU activa todas las banderas de estado después de una operación aritmética usando un conjunto de reglas booleanas, sin importar qué banderas son relevantes. Usted (el programador) decide cuáles banderas interpretar y cuáles ignorar, de acuerdo con su conocimiento del tipo de operación realizada.

4.2.7 Programa de ejemplo (SumaResta3)

El siguiente programa implementa varias expresiones aritméticas, usando las 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 las 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 la bandera Acarreo:
    mov al,0FFh
    add al,1           ; CF = 1, AL = 00
    ; Ejemplo de la bandera Desbordamiento:
    mov al,+127
    add al,1           ; OF = 1
    mov al,-128
    sub al,1           ; OF = 1
    exit
main ENDP
END main
```

4.2.8 Repaso de sección

Use estos datos para las siguientes preguntas:

```
.data
val1 BYTE 10h
val2 WORD 8000h
val3 DWORD 0FFFFh
val4 WORD 7FFFh
```

1. Escriba una instrucción para incrementar val2.
2. Escriba una instrucción para restar val3 de EAX.

3. Escriba instrucciones que resten **val4** de **val2**.
4. Si **val2** se incrementa en 1 usando la instrucción ADD, ¿cuáles serán los valores de las banderas Acarreo y Signo?
5. Si **val4** se incrementa en 1 usando la instrucción ADD, ¿cuáles serán los valores de las banderas Desbordamiento y Signo?
6. En donde se indica, escriba los valores de las banderas Acarreo, Signo, Cero y Desbordamiento después de la ejecución de cada instrucción:

```

mov  ax,7FF0h
add  al,10h          ; a. CF = SF = ZF = OF =
add  ah,1          ; b. CF = SF = ZF = OF =
add  ax,2          ; c. CF = SF = ZF = OF =

```

7. Implemente la siguiente expresión en lenguaje ensamblador: $AX = (-val2 + BX) - val4$.
8. (Sí/No): ¿es posible activar la bandera Desbordamiento si sumamos un entero positivo a un entero negativo?
9. (Sí/No): ¿se activará la bandera Desbordamiento si sumamos un entero negativo a un entero negativo y se produce un resultado positivo?
10. (Sí/No): ¿es posible que la instrucción NEG active la bandera Desbordamiento?
11. (Sí/No): ¿es posible que las banderas Signo y Cero se activen al mismo tiempo?
12. Escriba una secuencia de dos instrucciones que activen las banderas Acarreo y Desbordamiento al mismo tiempo.
13. Escriba una secuencia de instrucciones que muestren cómo puede utilizarse la bandera Cero para indicar un desbordamiento sin signo, después de ejecutar las instrucciones INC y DEC.
14. En nuestra discusión de la bandera Acarreo, restamos un 2 sin signo a un 1; para ello negamos el 2 y lo sumamos al 1. La bandera Acarreo fue la inversión del acarreo que salió del MSB de la suma. Demuestre este proceso restando 3 a 4, y muestre cómo se produce el valor de la bandera Acarreo.

4.3 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:

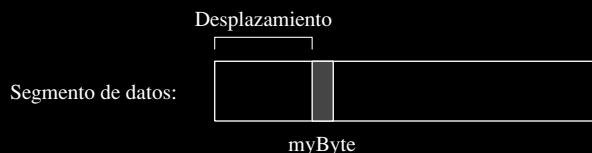
- El operador OFFSET devuelve la distancia de una variable, a partir del inicio de su segmento circundante.
- El operador PTR nos permite redefinir el tamaño predeterminado de una variable.
- El operador TYPE devuelve el tamaño (en bytes) de un operando, o de cada elemento en un arreglo.
- El operador 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 los operadores que soporta MASM. Tal vez quiera dar un vistazo al listado completo en el apéndice D.

MASM sigue ofreciendo soporte para las directivas heredadas LENGTH (en vez de LENGTHOF) y SIZE (en vez de SIZEOF).

4.3.1 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. Para ilustrar esto, la siguiente figura muestra una variable llamada **myByte** dentro del segmento de datos:



En el modo protegido, los desplazamientos son de 32 bits. En el modo de 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. Suponga 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
```

4.3.2 Directiva ALIGN

La directiva ALIGN alinea una variable en 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 vacíos 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 4.3.1, **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.

4.3.3 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, suponga que deseamos mover hacia AX los 16 bits inferiores de una variable tipo doble palabra llamada **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* (sección 3.4.9), 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 de **miDoble** de tres formas: primero como doble palabra, después como dos palabras (5678h, 1234h), y por último como cuatro bytes (78h, 56h, 34h, 12h):

Doble palabra	Palabra	Byte	Desplazamiento	
12345678	5678	78	0000	miDoble
		56	0001	miDoble + 1
	1234	34	0002	miDoble + 2
		12	0003	miDoble + 3

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 b1,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 TBYTE.

Mover valores más pequeños a destinos más grandes En algunas ocasiones será necesario mover dos valores más pequeños de la 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
```

4.3.4 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 ?
```

La siguiente tabla muestra el valor de cada expresión TYPE:

Expresión	Valor
TYPE var1	1
TYPE var2	2
TYPE var3	4
TYPE var4	8

4.3.5 Operador LENGTHOF

El operador LENGTHOF cuenta el número de elementos en un arreglo, definido por los valores que aparecen en la misma línea que su etiqueta. Utilizaremos los siguientes datos como 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 declara un arreglo que abarca varias líneas del 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, puede terminar la primera línea con 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,
           60,70,80,90,100
```

4.3.6 Operador SIZEOF

El operador SIZEOF devuelve un valor que equivale a 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
```

4.3.7 Directiva LABEL

La directiva LABEL nos permite insertar una etiqueta y proporcionarle un atributo de tamaño sin asignar espacio de almacenamiento. Con LABEL pueden usarse todos los atributos de tamaño estándar, como BYTE, WORD, DWORD, 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 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
```

4.3.8 Repaso de sección

1. (*Verdadero/Falso*): en el modo protegido de 32 bits, el operador OFFSET devuelve un valor de 16 bits.
2. (*Verdadero/Falso*): el operador PTR devuelve la dirección de 32 bits de una variable.
3. (*Verdadero/Falso*): el operador TYPE devuelve un valor de 4 para los operandos tipo doble palabra.
4. (*Verdadero/Falso*): el operador LENGTHOF devuelve el número de bytes en un operando.
5. (*Verdadero/Falso*): el operador SIZEOF devuelve el número de bytes en un operando.

Use estas definiciones de datos para los siguientes siete ejercicios:

```
.data
misBytes BYTE 10h,20h,30h,40h
misPalabras WORD 3 DUP(?),2000h
miCadena BYTE "ABCDE"
```

6. Inserte una directiva en los datos proporcionados, para alinear **misBytes** con una dirección de numeración par.
7. ¿Cuál será el valor de EAX después de que se ejecute cada una de las siguientes instrucciones?


```
mov eax, TYPE misBytes      ; a.
mov eax, LENGTHOF misBytes   ; b.
mov eax, SIZEOF misBytes     ; c.
mov eax, TYPE misPalabras   ; d.
mov eax, LENGTHOF misPalabras ; e.
mov eax, SIZEOF misPalabras   ; f.
mov eax, SIZEOF miCadena     ; g.
```
8. Escriba una sola instrucción para mover los primeros dos bytes en **misBytes** al registro DX. El valor resultante será 2010h.
9. Escriba una instrucción para mover el segundo byte en **misPalabras** al registro AL.
10. Escriba una instrucción para mover los cuatro bytes en **misBytes** al registro EAX.
11. Inserte una directiva LABEL en los datos proporcionados, para permitir que **misPalabras** se mueva directamente a un registro de 32 bits.
12. Inserte una directiva LABEL en los datos proporcionados, para permitir que **misBytes** se mueva directamente a un registro de 16 bits.

4.4 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 un arreglo. No es conveniente utilizar desplazamientos constantes para direccionar más de unos cuantos elementos del 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*.

4.4.1 Operandos indirectos

Modo protegido Un operando indirecto puede ser cualquier registro de propósito general de 32 bits (EAX, EBX, ECX, EDX, ESI, EDI, EBP 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 el origen, el desplazamiento en ESI se emplea para 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 memoria, en la ubicación a la que apunta el registro:

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

Modo de direccionamiento real En el modo de direccionamiento real, 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 dirige 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 un área afuera 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 generaría 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 lenguajes 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 mensaje de error del tipo “el operando debe tener un tamaño”:

```
inc [esi] ; error
```

El ensamblador no sabe si ESI apunta a un byte, una palabra, una doble palabra o a cualquier otro tamaño. El operador PTR aclara el tamaño del operando:

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

4.4.2 Arreglos

Los operandos indirectos son útiles al manejar 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 a 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
```

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

Desplazamiento	Valor
10200	1000h
10202	2000h
10204	3000h

← [esi]

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:

Desplazamiento	Valor
10200	10000h
10204	20000h
10208	30000h

← [esi]
← [esi] + 4
← [esi] + 8

4.4.3 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 que 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 iniciarse con cero antes de acceder al primer elemento del arreglo:

```
.data
arregloB BYTE 10h,20h,30h
.code
mov esi,0
mov al,[arregloB + 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 como 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 en el caso de los operandos indirectos, evite utilizar BP, excepto cuando direccione datos en la pila.

Factores de escala en operandos indexados

Los operandos indexados deben tener en cuenta el tamaño de cada elemento del 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 del 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 proporcionaron 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 nuestro ejemplo anterior, asignando a ESI el subíndice (3) del arreglo y multiplicando ESI por el factor de escala (4) para las 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 de otro tipo en el futuro:

```
mov esi,3 ; subíndice
mov eax,arregloD[esi*TYPE arregloD] ; EAX = 400h
```

4.4.4 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 actual del procesador (real de 16 bits o protegido de 32 bits), como se muestra en la tabla 4-2:

Tabla 4-2 Tipos de apuntadores en los modos de 16 y 32 bits.

	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 48 bits

En este libro, 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 de **arregloB** y **apuntW** contiene el desplazamiento de **arregloW**:

```
arregloB BYTE 10h,20h,30h,40h
arregloW WORD 1000h,2000h,3000h
apuntB     DWORD arregloB
apuntW     DWORD arregloW
```

Si lo desea, 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 con 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 10h,20h,30h,40h
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 arreglos y emplea los apuntadores para hacer la referencia a memoria:

```
TITLE Apuntadores (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 arregloB
apunt2 PWORD arregloW
apunt3 PDWORD arregloD

.code
main PROC
; Usa los apuntadores para acceder a los datos.
    mov esi,apunt1
    mov al,[esi] ; 10h
    mov esi,apunt2
    mov ax,[esi] ; 1
    mov esi,apunt3
    mov eax,[esi] ; 4
    exit
main ENDP
END main
```

4.4.5 Repaso de sección

1. (Verdadero/Falso): cualquier registro de propósito general de 16 bits puede usarse como operando indirecto.
2. (Verdadero/Falso): cualquier registro de propósito general de 32 bits puede usarse como operando indirecto.
3. (Verdadero/Falso): por lo general, el registro BX se reserva para direccionar la pila.
4. (Verdadero/Falso): un error de protección general ocurre en modo de direccionamiento real, cuando el subíndice de un arreglo está fuera de rango.
5. (Verdadero/Falso): la siguiente instrucción es inválida: inc [esi]
6. (Verdadero/Falso): el siguiente es un operando indexado: arreglo[esi]

Use las siguientes definiciones de datos para el resto de las preguntas en esta sección:

```
misBytes BYTE 10h,20h,30h,40h
misPalabras WORD 8Ah,3Bh,72h,44h,66h
misDobles DWORD 1,2,3,4,5
miApuntador DWORD misDobles
```

7. Llene los valores de los registros requeridos en el lado derecho de la siguiente secuencia de instrucciones:

```
mov  esi,OFFSET misBytes
mov  al,[esi]           ; a. AL =
mov  al,[esi+3]         ; b. AL =
mov  esi,OFFSET misPalabras + 2
mov  ax,[esi]           ; c. AX =
mov  edi,8
mov  edx,[misDobles + edi]    ; d. EDX =
mov  edx,misDobles[edi]      ; e. EDX =
mov  ebx,miApuntador
mov  eax,[ebx + 4]          ; f. EAX =
```

8. Llene los valores de los registros requeridos en el lado derecho de la siguiente secuencia de instrucciones:

```
mov  esi,OFFSET misBytes
mov  ax,WORD PTR [esi]       ; a. AX =
mov  eax,DWORD PTR misPalabras   ; b. EAX =
mov  esi,miApuntador
mov  ax,WORD PTR [esi+2]       ; c. AX =
mov  ax,WORD PTR [esi+6]       ; d. AX =
mov  ax,WORD PTR [esi-4]        ; e. AX =
```

4.5 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 amplia 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.

4.5.1 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 de 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.

4.5.2 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 realiza 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 de 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 del ciclo, y la instrucción LOOPW utiliza a CX como el contador del 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 (en modo de 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 promedio 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 14 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 le agotan los registros y necesita ECX para otro fin, guarde su contenido en una variable al principio del ciclo y restáurelo 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 de otro, hay que tener cierta consideración especial con el contador del ciclo exterior en ECX. Puede guardarlo 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            ; establece la cuenta del ciclo interior
L2:
    .
    .
    loop L2              ; repite el ciclo interior
    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 de los ciclos se vuelve demasiado problemática. Si el algoritmo que utiliza requiere de ciclos con más de 2 niveles de anidación, mueva algunos de los ciclos internos hacia las subrutinas.

4.5.3 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 el lenguaje ensamblador, deben seguirse estos pasos:

1. Asignar la dirección del arreglo a un registro que servirá como operando indexado.
2. Establecer ECX con el número de los 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 al 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 de arregloInt
    mov  ecx,LENGTHOF arregloInt   ; contador del ciclo
    mov  ax,0                        ; pone el acumulador en ceros
L1:
    add  ax,[edi]                  ; agregar un entero
    add  edi,TYPE arregloInt       ; apunta al siguiente entero
    loop L1                         ; repite hasta que ECX = 0
exit
main ENDP
END main

```

4.5.4 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 el 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.

Cuando programan en C++ o Java, es común que los programadores principiantes no se den cuenta de la frecuencia con la que se llevan a cabo las operaciones de copia en segundo plano. En Java, por ejemplo, si se excede la capacidad existente de un objeto ArrayList al agregar un nuevo elemento, el sistema en tiempo de ejecución asigna un bloque de almacenamiento nuevo, copia los datos existentes a una nueva ubicación y elimina los datos anteriores. (Lo mismo se aplica cuando se utiliza un vector en C++). Si se llevan a cabo muchas operaciones de copia, tienen un efecto considerable sobre la velocidad de ejecución de un programa.

4.5.5 Repaso de sección

1. (*Verdadero/Falso*): una instrucción JMP sólo puede saltar a una etiqueta dentro del procedimiento actual, a menos que esa etiqueta se haya designado como global.
2. (*Verdadero/Falso*): JMP es una instrucción de transferencia condicional.
3. Si ECX se inicializa con cero antes de empezar un ciclo, ¿cuántas veces se repetirá la instrucción LOOP? (Suponga que no hay instrucciones que modifiquen a ECX dentro del ciclo).
4. (*Verdadero/Falso*): la instrucción LOOP primero verifica si ECX es mayor que cero; después decrementa ECX y salta a la etiqueta de destino.
5. (*Verdadero/Falso*): la instrucción LOOP hace lo siguiente: decrementa ECX; después, si ECX es mayor que cero, la instrucción salta a la etiqueta de destino.
6. En el modo de direccionamiento real, ¿qué registro utiliza la instrucción LOOP como contador?
7. En el modo de direccionamiento real, ¿qué registro utiliza la instrucción LOOPD como contador?
8. (*Verdadero/Falso*): el destino de una instrucción LOOP debe estar a no más de 256 bytes de distancia de la ubicación actual.
9. *Reto*: ¿cuál será el valor final de EAX en este ejemplo?

```

        mov    eax,0
        mov    ecx,10           ; contador del ciclo exterior
L1:
        mov    eax,3
        mov    ecx,5            ; contador de ciclo interior

```

```

L2:
    add  eax,5
    loop L2           ; repite el ciclo interior
    loop L1           ; repite el ciclo exterior

```

10. Revise el código de la pregunta anterior, para que el contador del ciclo exterior no se borre cuando empiece el ciclo interior.

4.6 Resumen del capítulo

MOV, una instrucción de transferencia de datos, copia un operando de origen a un operando de destino. La instrucción MOVZX extiende con ceros un operando más pequeño en uno más grande. La instrucción MOVSX extiende con signo un operando más pequeño en un registro más grande. La instrucción XCHG intercambia el contenido de dos operandos. Cuando menos un operando debe ser un registro.

Tipos de operandos En este capítulo presentamos los siguientes tipos de operandos:

- Un operando *directo* es el nombre de una variable, y representa su dirección.
- Un operando de *desplazamiento directo* suma un desplazamiento al nombre de una variable, con lo que genera un nuevo desplazamiento, que puede usarse para acceder a los datos en la memoria.
- Un operando *indirecto* es un registro que contiene la dirección de los datos. Al encerrar el registro entre corchetes (como en [esi]), un programa hace referencia a la dirección y obtiene los datos de la memoria.
- Un operando *indexado* combina una constante con un operando indirecto. La constante y el valor del registro se suman, y el desplazamiento resultante se emplea para hacer referencia a la dirección. Por ejemplo, [arreglo + esi] y arreglo[esi] son operandos indexados.

Las siguientes operaciones aritméticas son importantes:

- La instrucción INC suma 1 a un operando.
- La instrucción DEC resta 1 a un operando.
- La instrucción ADD suma un operando de origen a un operando de destino.
- La instrucción SUB resta un operando de origen de un operando de destino.
- La instrucción NEG invierte el signo de un operando.

Cuando convierta expresiones aritméticas simples en lenguaje ensamblador, use las reglas de precedencia de los operadores estándar para seleccionar qué expresión se debe evaluar primero.

Banderas de estado Las siguientes banderas de estado de la CPU se ven afectadas por las operaciones aritméticas:

- La bandera Signo se activa cuando el resultado de una operación aritmética es negativo.
- La bandera Acarreo se activa cuando el resultado de una operación aritmética sin signo es demasiado grande para el operando de destino.
- La bandera Acarreo auxiliar se activa cuando ocurre un acarreo o un préstamo en la posición del bit 3 del operando de destino.
- La bandera Cero se activa cuando el resultado de una operación aritmética es cero.
- La bandera Desbordamiento se activa cuando el resultado de una operación aritmética con signo es demasiado grande para el operando de destino. Por ejemplo, en una operación con bytes, la CPU detecta el desbordamiento aplicando un OR exclusivo al acarreo que sale del bit 6 con el acarreo que sale del bit 7.

Operadores Los siguientes operadores son de uso común en el lenguaje ensamblador:

- El operador OFFSET devuelve la distancia de una variable, a partir del inicio de su segmento circundante.
- El operador PTR redefine el tamaño declarado de una variable.
- El operador TYPE devuelve el tamaño (en bytes) de una sola variable, o de un elemento individual de 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.
- El operador TYPEDEF crea un tipo definido por el usuario.

Ciclos La instrucción JMP se bifurca en forma incondicional hacia otra ubicación. La instrucción LOOP se utiliza en los ciclos con conteo. En el modo de 32 bits, LOOP utiliza a ECX como contador; en el modo de 16 bits, CX es el contador. En ambos modos de 16 y 32 bits, LOOPD (ciclo doble) utiliza a ECX como contador.

4.7 Ejercicios de programación

Los siguientes ejercicios pueden realizarse en modo protegido o en modo de direccionamiento real.

1. Bandera Acarreo

Escriba un programa que utilice la suma y la resta para activar y borrar la bandera Acarreo. Después de cada instrucción, inserte la instrucción **call DumpRegs** para mostrar los registros y las banderas. Utilice comentarios para explicar cómo (y por qué) cada instrucción afectó a la bandera Acarreo.

2. INC y DEC

Escriba un programa corto para demostrar que las instrucciones INC y DEC no afectan a la bandera Acarreo.

3. Banderas Cero y Signo

Escriba un programa que utilice operaciones de suma y resta para activar y borrar las banderas Cero y Signo. Después de cada instrucción de suma o de resta, inserte la instrucción **call DumpRegs** (vea la sección 3.2) para mostrar los registros y las banderas. Utilice comentarios para explicar cómo (y por qué) cada instrucción afectó a las banderas Cero y Signo.

4. Bandera Desbordamiento

Escriba un programa que utilice operaciones de suma y resta para activar y borrar la bandera Desbordamiento. Después de cada instrucción de suma o resta, inserte la instrucción **call DumpRegs** (vea la sección 3.2) para mostrar los registros y las banderas. Utilice comentarios para explicar cómo (y por qué) cada instrucción afectó a la bandera Desbordamiento. Incluya una instrucción ADD para activar las banderas Acarreo y Desbordamiento.

5. Direccionamiento por desplazamiento directo

Inserte las siguientes variables en su programa:

```
.data  
arregloU WORD 1000h,2000h,3000h,4000h  
arregloS SWORD -1,-2,-3,-4
```

Escriba instrucciones que utilicen el direccionamiento por desplazamiento directo para mover los cuatro valores en **arregloU** a los registros EAX, EBX, ECX y EDX. Si inserta la instrucción **call DumpRegs** después de este código (vea la sección 3.2), deberán aparecer los siguientes valores en los registros:

EAX=00001000 EBX=00002000 ECX=00003000 EDX=00004000

A continuación, escriba instrucciones que utilicen direccionamiento por desplazamiento directo para mover los cuatro valores en **arregloS** a los registros EAX, EBX, ECX y EDX. Si inserta la instrucción **call DumpRegs** después de este código, deberán aparecer los siguientes valores en los registros:

EAX=FFFFFFFEBX=FFFFFFFE ECX=FFFFFFFD EDX=FFFFFFFC

6. Números de Fibonacci

Escriba un programa que utilice un ciclo para calcular los primeros 12 valores en la secuencia de números de Fibonacci, {1,1,2,3,5,8,13,...}. Coloque cada valor en el registro EAX y muéstrela con una instrucción **call DumpRegs** (vea la sección 3.2) dentro del ciclo.

7. Expresión aritmética

Escriba un programa que implemente la siguiente expresión aritmética:

EAX = -val2 + 7 - val3 + val1

Utilice las siguientes definiciones de datos:

```
val1 SDWORD 8  
val2 SDWORD -15  
val3 SDWORD 20
```

En comentarios enseguida de cada instrucción, escriba el valor hexadecimal de EAX. Inserte una instrucción **call DumpRegs** al final del programa.

8. Copia de una cadena al revés

Escriba un programa en el que utilice la instrucción LOOP con direccionamiento indirecto, para copiar una cadena de **origen** a **destino**, invirtiendo el orden de los caracteres en el proceso. Use las siguientes variables:

```
origen BYTE "Esta es la cadena de origen",0  
destino BYTE SIZEOF origen DUP('#')
```

Inserte las siguientes instrucciones justo después del ciclo para mostrar el contenido hexadecimal de la cadena de destino:

```
mov esi,OFFSET destino ; desplazamiento de la variable  
mov ebx,1 ; formato de byte  
mov ecx,SIZEOF destino ; contador  
call DumpMem
```

Si su programa funciona en forma correcta, mostrará la siguiente secuencia de bytes hexadecimales:

```
67 6E 69 72 74 73 20 65 63 72 75 6F 73 20 65 68  
74 20 73 69 20 73 68 68 54
```

(En la sección 5.3.2 explicaremos el procedimiento DumpMem).

PROCEDIMIENTOS

- 5.1 Introducción
- 5.2 Enlace con una biblioteca externa
 - 5.2.1 Antecedentes
 - 5.2.2 Repaso de sección
- 5.3 La biblioteca de enlace del libro
 - 5.3.1 Generalidades
 - 5.3.2 Descripciones de los procedimientos individuales
 - 5.3.3 Programas de prueba de la biblioteca
 - 5.3.4 Repaso de sección
- 5.4 Operaciones de la pila
 - 5.4.1 La pila en tiempo de ejecución
 - 5.4.2 Instrucciones PUSH y POP
 - 5.4.3 Repaso de sección
- 5.5 Definición y uso de los procedimientos
 - 5.5.1 Directiva PROC
 - 5.5.2 Instrucciones CALL y RET
 - 5.5.3 Ejemplo: suma de un arreglo de enteros
 - 5.5.4 Diagramas de flujo
 - 5.5.5 Almacenamiento y restauración de registros
 - 5.5.6 Repaso de sección
- 5.6 Diseño de programas mediante el uso de procedimientos
 - 5.6.1 Programa para sumar enteros (diseño)
 - 5.6.2 Implementación de la suma de enteros
 - 5.6.3 Repaso de sección
- 5.7 Resumen del capítulo
- 5.8 Ejercicios de programación

5.1 Introducción

Podemos pensar en varias buenas razones por las que usted debe leer este capítulo:

- Desea saber cómo funcionan las operaciones de entrada-salida en el lenguaje ensamblador.
- Aprenderá acerca de la *pila en tiempo de ejecución*, el mecanismo fundamental para llamar a las subrutinas y regresar de ellas.
- Aprenderá a dividir programas extensos en subrutinas modulares.
- Aprenderá acerca de los *diagramas de flujo*: herramientas gráficas que representan la lógica de un programa.
- Su instructor podría aplicarle una prueba.

5.2 Enlace con una biblioteca externa

Si invierte el tiempo, podrá escribir código detallado para las operaciones de entrada-salida en lenguaje ensamblador. Es algo muy parecido al proceso de ensamblar el motor de su automóvil cada vez que quiere dar un paseo. Un trabajo interesante, pero que consume mucho tiempo. En el capítulo 11 tendrá la oportunidad de ver cómo se manejan las operaciones de entrada-salida en el modo protegido de Windows. Es muy divertido,

además de que se le abrirá un mundo nuevo cuando vea las herramientas disponibles. Sin embargo, por ahora las operaciones de entrada-salida deben ser sencillas mientras aprende los fundamentos del lenguaje ensamblador. En la sección 5.3 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 los 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 los 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.

5.2.1 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 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. Suponga que un programa muestra una cadena en la ventana de la consola, llamando a un procedimiento de nombre **EscribirCadena**. El código fuente del programa debe contener una directiva **PROTO** que identifique al procedimiento **EscribirCadena**:

```
EscribirCadena PROTO
```

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

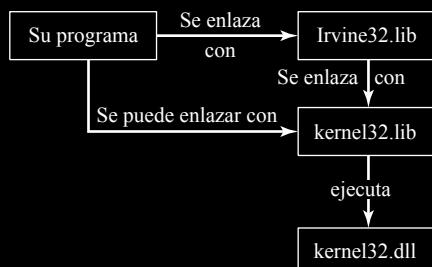
```
call EscibirCadena
```

Cuando el programa se ensambla, el ensamblador deja la dirección de destino de la instrucción CALL en blanco, sabiendo que el enlazador la llenará. El enlazador busca a **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 de **EscribirCadena** en la instrucción CALL. Si un procedimiento que usted llama no se encuentra en la biblioteca de enlace, el enlazador genera un mensaje de error y no genera un archivo ejecutable.

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 las bibliotecas **irvine32.lib** y **kernel32.lib**:

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

Cómo enlazar programas de 32 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 la plataforma Microsoft Windows, contiene información de enlace para las funciones del sistema que se encuentran 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 cómo **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**.

5.2.2 Repaso de sección

1. (*Verdadero/Falso*): una biblioteca de enlace consiste en código fuente en lenguaje ensamblador.
2. Use la directiva PROTO para declarar un procedimiento llamado **MiProc** en una biblioteca de enlace externa.
3. Escriba una instrucción CALL para llamar a un procedimiento de nombre **MiProc** en una biblioteca de enlace externa.
4. ¿Cuál es el nombre de la biblioteca de enlace de 32 bits que se incluye en el sitio Web?
5. ¿Qué biblioteca contiene funciones que se llaman desde **Irvine32.lib**?
6. ¿Qué tipo de archivo es **kernel32.dll**?
7. ¿Qué nombre se utiliza para el parámetro de nombre de archivo sustituible en el archivo **make32.bat**?

5.3 La biblioteca de enlace del libro

5.3.1 Generalidades

La tabla 5-1 contiene una lista de los 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 de 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 del sistema. Para mostrarla, haga clic en Inicio | Ejecutar y escriba **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 ratón en el menú del 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 25 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
```

Tabla 5-1 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 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 de <i>n</i> 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 las banderas de estado más comunes de la CPU
GetCommandTail	Copia los argumentos de línea de comandos del programa (llamados la <i>cola de comandos</i>) 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

(Continúa)

Tabla 5-1 (Continuación)

Procedimiento	Descripción
IsDigit	Activa la bandera Cero si el registro AL contiene el código ASCII para un dígito decimal (0-9)
MsgBox	Muestra un cuadro de mensaje contextual*
MsgBoxAsk	Muestra una pregunta tipo sí/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
ParseInteger32	Convierte una cadena de enteros decimales con signo a un número binario de 32 bits
Random32	Genera un entero seudoaleatorio de 32 bits en el rango de 0 a FFFFFFFFh
Randomize	Siembra el generador de números aleatorios con un valor único
RandomRange	Genera un entero seudoaleatorio dentro de un rango especificado
ReadChar	Espera a que se escriba un solo 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 binario a la ventana de consola en formato de byte, palabra o doble palabra
WriteChar	Escribe un solo carácter a la ventana de 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 con 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*

* Este procedimiento no está disponible en la biblioteca Irvine16.

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-salida 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: suponga que un programa llamado *ejemplo.exe* escribe a la salida estándar; entonces, podemos usar el siguiente comando (en el símbolo de 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 tanto la entrada como la salida con un solo comando:

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

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

```
prog1 | prog2
```

Podemos enviar la salida estándar 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 usted puede elegir los nombres que desee.

5.3.2 Descripciones de los procedimientos individuales

CloseFile (Irvine32 solamente) El procedimiento *CloseFile* cierra un archivo que estaba abierto previamente. El archivo se identifica mediante un *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, a 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 a *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 de 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 una llamada de ejemplo:

```
.data  
Nombrearchivo BYTE "nuevoarchivo.txt",0  
manejador DWORD ?  
.code  
mov edx,OFFSET nombrearchivo  
call CreateOutputFile  
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 hacia 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 llamada 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       ; formato de doble palabra
    call   DumpMem
```

Se produce la siguiente salida:

```
00000001 00000002 00000003 00000004 00000005 00000006 00000007
00000008 00000009 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 las banderas Acarreo, Signo, Cero, Desbordamiento, Acarreo auxiliar y Paridad. He aquí una llamada de ejemplo:

```
call DumpRegs
```

Resultados 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 GetCommandTail 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 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. Suponga 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 la línea de comandos cuando ejecute el programa:

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

Al iniciar, el programa Cifrar puede llamar a GetCommandTail y obtener los dos nombres de archivos. 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 colaComandos
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, aparecen 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 (*Irvine32 solamente*). No tiene parámetros de entrada. Devuelve el color de fondo en los cuatro bits superiores de AL y el color de texto en los 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 Gotoxy                     ; 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 6.3.2, 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 de dialogo", 0
MsjHola 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
```

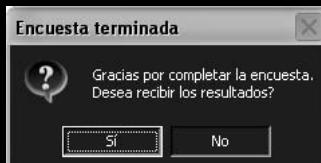
Resultados de ejemplo:



MsgBoxAsk (*Irvine32 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 en 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 por completar la encuesta."
        BYTE 0dh,0ah
        BYTE "Desea recibir los resultados?",0
resultados BYTE "Los resultados se enviaran via correo electronico.",0dh,0ah,0
.code
mov ebx,OFFSET leyenda
mov edx,OFFSET pregunta
call MsgBoxAsk
;(comprobar el valor de retorno en EAX)
```

Salida de ejemplo:



OpenInputFile (*Irvine32 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 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 las 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 con 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
bufer BYTE "8193"
tamBufer = ($ - bufer)
.code
mov edx,OFFSET bufer
mov ecx,tamBufer
call ParseDecimal32 ; devuelve EAX
```

Consulte la descripción del procedimiento **ReadDec** para ver los detalles acerca de cómo se ve afectada la bandera Acarreo.

ParseInteger32 El procedimiento ParseInteger32 convierte una cadena de enteros decimales con 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
bufer BYTE "-8193"
tamBufer = ($ - bufer)
.code
mov edx,OFFSET bufer
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 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*.¹ 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. Sólo necesitamos llamar a Randomize 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 del 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 de ejemplo:

```
.data
car BYTE ?
.code
call ReadChar
mov car, al
```

Si el usuario oprime una tecla extendida, como una tecla de función, tecla de flecha de cursor, Insert o Supr, el procedimiento establece AL en cero y AH contiene un código de exploración del teclado. En las guardas de este libro (a la vuelta de la portada) se presenta una lista de códigos de exploración. La mitad superior de EAX no se preserva.

ReadDec El procedimiento ReadDec 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 con 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 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 (Irvine32 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_BUFER = 5000
.data
bufer BYTE TAM_BUFER DUP(?)
bytesLeidos DWORD ?
.code
mov edx,OFFSET bufer           ; apunta al búfer
mov ecx,TAM_BUFER             ; 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 los caracteres inválidos. Puede 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 SDWORD ?
.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 10ms
    call Delay
    call ReadKey        ; comprueba la tecla
    jz L1               ; repite si no hay tecla
    mov car,AL          ; guarda el carácter
```

Observe que agregamos 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 utiliza la biblioteca Irvine16, 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 el usuario, más 1 (para guardar espacio para el byte de terminación nulo). El procedimiento devuelve la cuenta del número de caracteres escritos por el usuario en EAX. He aquí una llamada de ejemplo:

```
.data
bufer BYTE 21 DUP(0)      ; búfer de entrada
cuentaBytes DWORD ?       ; guarda el contador
.code
mov edx,OFFSET bufer     ; apunta al búfer
mov ecx,SIZEOF bufer     ; especifica el máximo de caracteres
call ReadString           ; recibe la cadena de entrada
mov cuentaBytes,eax       ; número 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 **bufer**, después de que el usuario introduce la cadena “ABCDEFG”:

41 42 43 44 45 46 47 00	ABCDEFG
-------------------------	---------

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*. Multiplique el color de fondo por 16 y súmelo al color de texto.² Por ejemplo, la siguiente constante indica caracteres amarillos en un fondo azul:

```
yellow + (blue * 16)
```

Las siguientes instrucciones establecen el color a blanco, en un fondo azul:

```
mov eax,white + (blue * 16)      ; blanco sobre azul
call SetTextColor
```

En la sección 15.3.2 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
bufer BYTE "abcde",0
longBufer DWORD ?
.code
mov edx,OFFSET bufer          ; 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...” (“Oprima cualquier tecla para continuar”) 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 parámetros de entrada. He aquí una llamada de ejemplo:

```
call WaitMsg
```

WriteBin El procedimiento WriteBin escribe un entero en la ventana de 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 se 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 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 0011 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 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 una llamada 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 edx,OFFSET indicador
call WriteString
```

WriteToFile (*Irvine32 solamente*) El procedimiento WriteToFile escribe el contenido de un búfer en un archivo de salida. Se le pasa un manejador de archivos válido 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_BUFER = 5000
.data
manejadorArchivo DWORD ?
bufer BYTE TAM_BUFER DUP(?)
bytesEscritos DWORD ?
.code
mov eax,manejadorArchivo
mov edx,OFFSET bufer
mov ecx,TAM_BUFER
call WriteToFile
mov bytesEscritos,eax ; guarda el valor de retorno
```

WriteWindowsMsg (*Irvine32 solamente*) El procedimiento WriteWindowsMsg 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 encontrar 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 consola:

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

El archivo de inclusión *Irvine32.inc*

A continuación se muestra un listado parcial del archivo de inclusión *Irvine32.inc*. Este archivo contiene un prototipo para cada uno de los procedimientos de la biblioteca, así como las constantes de color, estructuras y definiciones de símbolos. Este archivo cambiará con el tiempo, por lo que debe descargar las últimas actualizaciones a la biblioteca desde el sitio Web del libro:

```
; Include file for Irvine32.lib          (Irvine32.inc)
INCLUDE SmallWin.inc
.NOLIST

; Procedure Prototypes
;-----
Clrscr PROTO
CrLf PROTO
Delay PROTO
DumpMem PROTO
DumpRegs PROTO
GetCommandTail PROTO
(más procedimientos...)

.
.

Standard 4-bit color definitions
;-----
black      = 0000b
blue       = 0001b
green      = 0010b
cyan       = 0011b
red        = 0100b
magenta   = 0101b
brown      = 0110b
lightGray  = 0111b
gray       = 1000b
lightBlue  = 1001b
lightGreen = 1010b
lightCyan  = 1011b
lightRed   = 1100b
lightMagenta = 1101b
yellow     = 1110b
white      = 1111b
.LIST
```

La directiva .NOLIST en la parte superior de este archivo evita que estas líneas se muestren en los listados de código fuente creados por el ensamblador. Al final de este archivo, la directiva .LIST permite listar las líneas de código fuente otra vez. La directiva INCLUDE al principio de este archivo hace que se incluya otro archivo de inclusión (SmallWin.inc) en el flujo de texto que se pasa al ensamblador.

5.3.3 Programas de prueba de la biblioteca

Programa de prueba #1: E/S de enteros

Vamos a ver varios programas para probar la biblioteca de enlace del libro. El programa de prueba #1 cambia el color del texto a caracteres amarillos en un fondo azul, vacía el contenido de un arreglo en hexadecimal, pide al usuario que introduzca un entero con signo y vuelve a mostrar ese entero en decimal, hexadecimal y binario:

```
TITLE Prueba de biblioteca #1: E/S de enteros (PruebaBib1.asm)
; Prueba los procedimientos Clrscr,CrLf,
; DumpMem, ReadInt, SetTextColor, WaitMsg,
```

```

; WriteBin, WriteHex y WriteString.
; Última actualización: 06/01/2006

INCLUDE Irvine32.inc
.data
arregloD DWORD 1000h,2000h,3000h
indicador1 BYTE "Escriba un entero con signo de 32 bits: ",0
valDword DWORD ?

.code
main PROC
; Establece el color del texto a texto amarillo sobre fondo azul:
    mov    eax,yellow + (blue * 16)
    call   SetTextColor
    call   Clrscr           ; borra la pantalla

; Muestra el arreglo usando DumpMem.
    mov    esi,OFFSET arregloD      ; desplazamiento inicial
    mov    ecx,LENGTHOF arregloD    ; número de unidades en valDword
    mov    ebx,TYPE arregloD       ; tamaño de una doble palabra
    call   DumpMem               ; muestra la memoria
    call   Crlf                  ; nueva línea

; Pide al usuario que introduzca un entero decimal con signo.
    mov    edx,OFFSET indicador1
    call   WriteString
    call   ReadInt                ; recibe el entero de entrada
    mov    valDword,eax           ; lo guarda en una variable

; Muestra el entero en decimal, hexadecimal y binario.
    call   Crlf                  ; nueva línea
    call   WriteInt               ; lo muestra en decimal con signo
    call   Crlf
    call   WriteHex               ; lo muestra en hexadecimal
    call   Crlf
    call   WriteBin               ; lo muestra en binario
    call   Crlf
    call   WaitMsg                ; "Oprima cualquier tecla..."

; Devuelve los colores predeterminados de la ventana de consola.
    mov    eax,lightGray + (black * 16)
    call   SetTextColor
    call   Clrscr                 ; borra la pantalla
    exit
main ENDP
END main

```

Resultados de ejemplo, programa de prueba #1 He aquí un ejemplo de los resultados que genera el programa (el texto es amarillo sobre un fondo azul):

Dump of offset 00405000

00001000 00002000 00003000
Escriba un entero con signo de 32 bits: 4333
+4333
000010ED
0000 0000 0000 0000 0001 0000 1110 1101
Press any key to continue...

Programa de prueba #2: enteros aleatorios

Nuestro segundo programa de prueba de la biblioteca de enlace demuestra sus capacidades de generación de números aleatorios. Primero, genera 10 enteros sin signo al azar, en el rango de 0 a 4,294,967,294. A continuación, genera 10 enteros con signo en el rango de -50 a +49:

```
TITLE Prueba de biblioteca de enlace #2      (PruebaBib2.asm)
; Prueba de los procedimientos de la biblioteca Irvine32.
; Última actualización: 06/01/2006

INCLUDE Irvine32.inc

TAB = 9                                ; Código ASCII para el tabulado

.code
main PROC
    call Randomize                  ; inicia el generador aleatorio
    call Rand1
    call Rand2
    exit
main ENDP

Rand1 PROC
; Genera diez enteros seudoaleatorios.
    mov  ecx,10                      ; itera 10 veces
L1: call Random32                     ; genera entero aleatorio
    call WriteDec                    ; lo escribe en decimal sin signo
    mov  al,TAB                      ; tabulador horizontal
    call WriteChar                   ; escribe el tabulador
    loop L1
    call Crlf
    ret
Rand1 ENDP

Rand2 PROC
; Genera diez enteros seudoaleatorios entre -50 y +49
    mov  ecx,10                      ; itera 10 veces
L1: mov  eax,100                       ; valores de 0-99
    call RandomRange                ; genera entero aleatorio
    sub  eax,50                      ; valores de -50 a +49
    call WriteInt                   ; escribe decimal con signo
    mov  al,TAB                      ; tabulador horizontal
    call WriteChar                   ; escribe el tabulador
    loop L1
    call Crlf
    ret
Rand2 ENDP
END main
```

He aquí un ejemplo de los resultados que produce el programa:

2315709604	632599385	1418799463	3347191992	1848985168
2046753509	1482200338	3150335687	2465298675	220669068
+28	-48	-1	+22	+6

Programa de prueba # 3: cronometraje del rendimiento

A menudo, el lenguaje ensamblador se utiliza para optimizar secciones del código que se considera crítico para el rendimiento de un programa. El procedimiento **GetMseconds** de la biblioteca de enlace devuelve el número de milisegundos transcurridos desde medianoche. En el tercer programa de prueba, llamamos a GetMseconds y ejecutamos un ciclo anidado 17 millones de veces, aproximadamente. Después del ciclo, llamamos a GetMseconds una segunda vez y reportamos el tiempo total transcurrido:

```
TITLE Prueba de biblioteca de enlace #3      (PruebaBib3.asm)
; Calcula el tiempo transcurrido al ejecutar un ciclo anidado
; 17 millones de veces, aproximadamente.
; Última actualización: 06/01/2006

INCLUDE Irvine32.inc

CUENTA_CICLO_EXTERIOR = 3                      ; se ajusta a la velocidad del procesador
.data
tiempoInicial DWORD ?
msj1 BYTE "Por favor espere...",0dh,0ah,0
msj2 BYTE "Milisegundos transcurridos: ",0

.code
main PROC
    mov    edx,OFFSET msj1
    call   WriteString

    ; Guarda el tiempo inicial.
    call   GetMSeconds
    mov    tiempoInicial,eax
    mov    ecx,CUENTA_CICLO_EXTERIOR

    ; Realiza un ciclo ocupado.
L1:   call   cicloInterior
    loop  L1

    ; Muestra el tiempo transcurrido.
    call   GetMSeconds
    sub    eax,tiempoInicial
    mov    edx,OFFSET msj2
    call   WriteString
    call   WriteDec
    call   CrLf
    exit
main ENDP

cicloInterior PROC
    push  ecx
    mov   ecx,0FFFFFFFh
L1:   mov   eax,eax
    loop  L1
    pop   ecx
    ret
cicloInterior ENDP
END main
```

He aquí un ejemplo de los resultados de este programa, ejecutándose en un procesador Pentium 4 de 3 GHz:

```
Por favor espere...
Milisegundos transcurridos: 9157
```

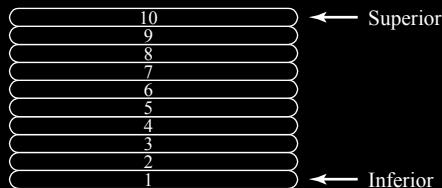
5.3.4 Repaso de sección

1. ¿Qué procedimiento de la biblioteca de enlace genera un entero aleatorio, dentro de un rango seleccionado?
2. ¿Qué procedimiento de la biblioteca de enlace muestra el mensaje “Press [Enter] to continue...”, y espera a que el usuario oprima la tecla Intro?
3. Escriba instrucciones que hagan que un programa detenga su ejecución durante 700 milisegundos.
4. ¿Qué procedimiento de la biblioteca de enlace escribe un entero sin signo en la ventana de consola, en formato decimal?
5. ¿Qué procedimiento de la biblioteca de enlace coloca el cursor en una posición específica de la ventana de consola?
6. Escriba la directiva INCLUDE requerida para utilizar la biblioteca Irvine32.
7. ¿Qué tipos de instrucciones se encuentran dentro del archivo *Irvine32.inc*?
8. ¿Cuáles son los parámetros de entrada requeridos para el procedimiento DumpMem?
9. ¿Cuáles son los parámetros de entrada requeridos para el procedimiento ReadString?
10. ¿Qué banderas de estado del procesador muestra el procedimiento DumpRegs?
11. *Reto*: escriba instrucciones para pedir al usuario un número de identificación y para introducir una cadena de dígitos en un arreglo de bytes.

5.4 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 a la parte superior de la pila, pero no abajo ni en medio (figura 5-1):

FIGURA 5-1 Pila de panqueques.



Los panqueques tienen algo en común con los programas de computadora. A una pila también se le conoce como estructura UEPS (*Último en entrar, primero en salir*), en inglés 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, pero los panqueques son mucho más interesantes).

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. Si usted ha tomado algún curso de programación en el que se hayan utilizado estructuras de datos, debe haber trabajado con el *tipo de datos abstracto pila*.

Sin embargo, 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 llamamos simplemente *pila*.

5.4.1 La 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 un 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 hexadecimal 00001000, el desplazamiento del valor más reciente que se metió a la pila (00000006):



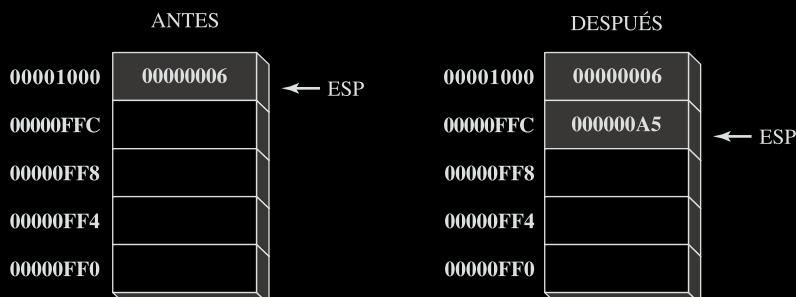
Cada posición de la pila en esta figura contiene 32 bits, que es el caso cuando el programa se ejecuta en modo protegido. En el modo de direccionamiento real de 16 bits, el registro SP apunta al valor más reciente que se metió a la pila, y las entradas de la pila son, por lo general, de 16 bits de longitud.

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 los cursos 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 figura 5-2, metemos 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 “último en entrar, primero en salir”. Antes de la operación push, ESP = 00001000h; después de push, ESP = 00000FFCh. La figura 5-3 muestra la misma pila, después de meter dos enteros más.

FIGURA 5–2 Meter enteros en la pila.



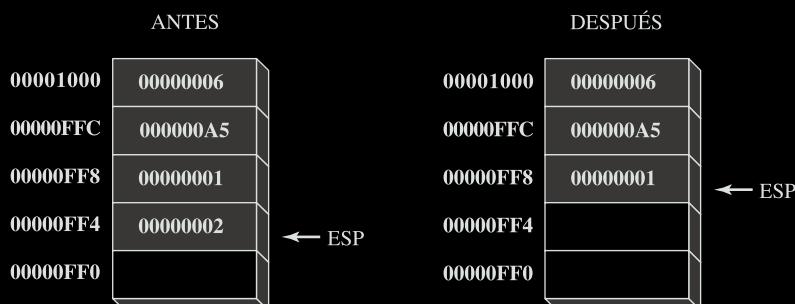
Operación Pop (sacar)

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 apuntar a la siguiente ubicación más alta en la pila. La figura 5-4 muestra la pila antes y después de sacar el valor 00000002.

FIGURA 5–3 La pila, después de meter 00000001 y 00000002.



FIGURA 5–4 Sacar un valor de la pila en tiempo de ejecución.



El área de la pila debajo de ESP está *lógicamente vacía*, y se sobrescribirá la próxima vez que el programa 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 llamados *argumentos*, los cuales se meten en la pila.
- La pila proporciona un área de almacenamiento temporal para las variables locales, dentro de los procedimientos.

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

Los valores inmediatos siempre son de 32 bits en modo protegido. En modo de direccionamiento real, los valores inmediatos son de 16 bits de manera predeterminada, a menos que se utilice la directiva del procesador .386 (o superior) (en la sección 3.2.1 hablamos sobre la directiva .386).

Instrucción POP

La instrucción POP primero copia el contenido del elemento de la pila al que apunta ESP, en un operando de destino de 16 o 32 bits, y después incrementa 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 copia 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 las 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 tiempo, puede ser difícil recordar en dónde 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, e inmediatamente después sacarlas y colocarlas en una variable:

```
.data
guardarBanderas DWORD ?
.code
pushfd           ; mete las banderas en la pila
pop  guardarBanderas ; las copia en una variable
```

Las siguientes instrucciones restauran las banderas desde la misma variable:

```
push guardarBanderas ; mete los valores guardados de las banderas
popfd              ; los 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, ESI 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 80286, mete los registros de propósito general de 16 bits (AX, CX, DX, BX, SP, BP, SI, DI) en la pila, en el orden listado. La instrucción POPA saca los mismos registros en orden inverso.

Si escribe un procedimiento para modificar varios registros de 32 bits, use 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. Suponga 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 UEPS (*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          ; obtiene el carácter
      mov  unNombre[esi],al ; lo almacena en la cadena
      inc  esi
      loop L2

; Muestra el nombre.
      mov  edx,OFFSET unNombre
      call WriteString
      call CrLf
      exit
main ENDP
END main

```

5.4.3 Repaso de sección

1. ¿Cuáles son los dos registros (en modo protegido) que manejan la pila?
2. ¿Qué diferencia hay entre la pila en tiempo de ejecución y el tipo de datos abstracto pila?
3. ¿Por qué a la pila se le conoce como estructura UEPS?
4. Cuando se mete un valor de 32 bits en la pila, ¿qué ocurre con ESP?
5. (Verdadero/Falso): cuando se utiliza la biblioteca Irvine32, sólo deben meterse valores de 32 bits en la pila.
6. (Verdadero/Falso): cuando se utiliza la biblioteca Irvine16, sólo deben meterse valores de 16 bits en la pila.
7. (Verdadero/Falso): las variables locales en los procedimientos se crean en la pila.
8. (Verdadero/Falso): la instrucción PUSH no puede tener un operando inmediato.
9. ¿Qué instrucción mete todos los registros de propósito general de 32 bits en la pila?
10. ¿Qué instrucción mete el registro EFLAGS de 32 bits en la pila?
11. ¿Qué instrucción saca elementos de la pila y los coloca en el registro EFLAGS?
12. *Reto:* otro ensamblador (llamado NASM) permite que la instrucción PUSH liste varios registros específicos. ¿Por qué este método podría ser mejor que la instrucción PUSHAD en MASM? He aquí un ejemplo de NASM:


```
PUSH EAX EBX ECX
```
13. *Reto:* suponga que no existe la instrucción PUSH. Escriba una secuencia de otras dos instrucciones que realicen lo mismo que PUSH EAX.

5.5 Definición y uso de los procedimientos

Si ya ha estudiado un lenguaje de programación de alto nivel, entonces 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 *procedimiento* para indicar una subrutina. En otros lenguajes, a las subrutinas se les llama métodos o funciones.

En términos de la 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 no 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.

5.5.1 Directiva PROC

Definición de un procedimiento

De manera informal, podemos definir a un *procedimiento* como un bloque de instrucciones con nombre, que termina en una instrucción de retorno. Para declarar un procedimiento se utilizan las directivas PROC y ENDP. Se le debe asignar un nombre (un identificador válido). Cada uno de los programas que hemos escrito 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 ENDP
```

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 alias para una llamada a **ExitProcess**, un procedimiento del sistema que termina el programa:

```
INVOKE ExitProcess,0
```

En la sección 8.5.1 presentaremos la directiva INVOKE, que puede llamar a un procedimiento y pasárle argumentos.

Si utiliza la instrucción INCLUDE *Irvine16.inc*, **exit** se traduce a la directiva de ensamblador **.EXIT**. Esta directiva hace que el ensamblador genere 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 sus programas. A continuación se muestran algunas sugerencias para la información que 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 sus valores de entrada, debe mostrarlos aquí.
- Una descripción de los valores devueltos por el procedimiento, etiquetados mediante una palabra como **Devuelve**.
- Una lista de los requerimientos espaciales, 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 el modo de 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 bits.
; Recibe: EAX, EBX, ECX, los tres enteros. Pueden ser
;         con o sin signo.
; Devuelve: EAX = suma
;
add eax,ebx
add eax,ecx
ret
SumaDe ENDP
```

Por lo general, las funciones escritas en lenguajes 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.

5.5.2 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 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 EIP (registro apuntador de instrucciones). En modo de 16 bits, IP apunta a la instrucción.

Ejemplo de llamada y retorno

Suponga que en **main**, una instrucción CALL se encuentra en el desplazamiento 00000020. Por lo general, esta instrucción requiere 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, suponga 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 (figura 5-5), la dirección que sigue después de la llamada (00000025) se mete en la pila, y la dirección de **MiSub** se carga en EIP. Todas las instrucciones en **MiSub** se ejecutan hasta su instrucción RET. Cuando se ejecuta la instrucción RET, el valor en la pila al que apunta ESP se saca y se coloca en EIP (paso 1 en la figura 5-6). En el paso 2, ESP se decrementa, de manera que apunta al valor anterior en la pila (paso 2).

Llamadas a procedimientos anidados

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. Suponga 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 al procedimiento **Sub3**. El proceso se muestra en la figura 5-7.

FIGURA 5-5 Ejecución de una instrucción CALL.

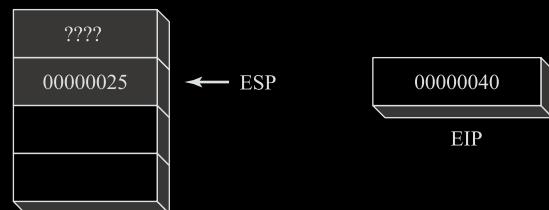
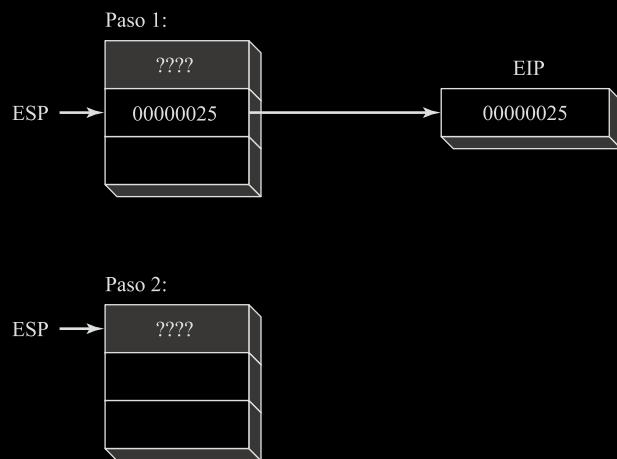
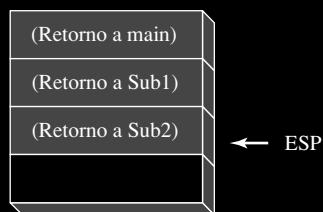


FIGURA 5-6 Ejecución de la instrucción RET.



Cuando se ejecuta la instrucción RET al final de **Sub3**, saca el valor que hay en 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:

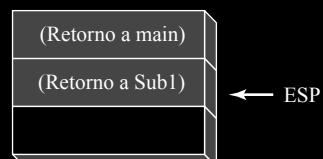
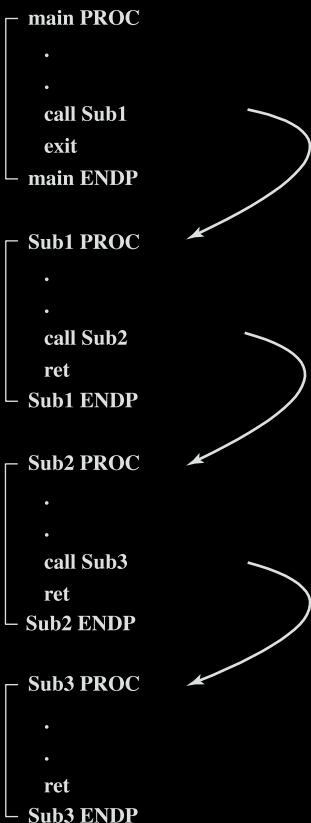
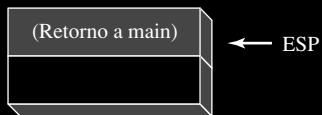


FIGURA 5–7 Llamadas a procedimientos anidadas.



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 que los programas deben volver a trazar sus pasos en un orden específico.

Paso de argumentos tipo registro a los procedimientos

Si usted escribe 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 lo hace, el procedimiento sólo podrá usarse con un arreglo. Un mejor método es pasar el desplazamiento de un arreglo al procedimiento, y pasárselo 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          ; argumento
    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.

5.5.3 Ejemplo: suma de un arreglo de enteros

Un tipo bastante común de ciclo, que probablemente ya codificó 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, podemos 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 que lo llama: un apuntador a un arreglo de enteros de 32 bits, y una cuenta del número de valores del arreglo. 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
    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
```

```

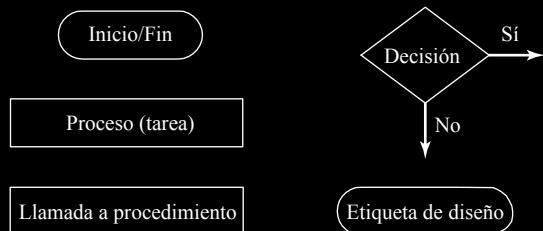
main PROC
    mov    esi,OFFSET arreglo      ; ESI apunta al arreglo
    mov    ecx,LENGTHOF arreglo    ; ECX = cuenta del arreglo
    call   SumaArreglo            ; calcula la suma
    mov    laSuma,eax             ; se devuelve en EAX

```

5.5.4 Diagramas de flujo

Un *diagrama de flujo* es una manera bien establecida de diagramar la lógica de un programa³. 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. La figura 5-8 muestra las 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.

FIGURA 5-8 Figuras básicas de un diagrama de flujo.



Las notaciones de texto como *sí* y *no* se agregan a un lado de los símbolos 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:

`cx = cx + 1`

`add cx, 1`

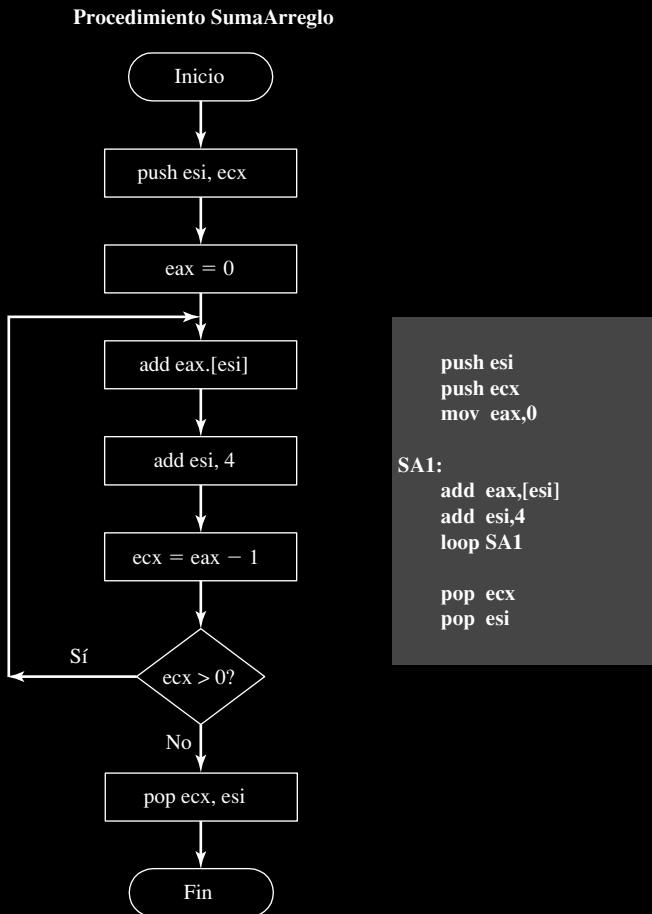
Vamos a utilizar el procedimiento **SumaArreglo** de la sección anterior para diseñar un diagrama de flujo simple, el cual se muestra en la figura 5-9. Utiliza un símbolo de decisión para la instrucción LOOP, ya que ésta debe determinar si se va a transferir o no el control a una etiqueta (con base en el valor de CX). El fragmento de código muestra el listado del procedimiento original.

5.5.5 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 los registros. 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 instrucciones PUSH que 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 espacios o tabuladores (no por comas).

FIGURA 5–9 Diagrama de flujo para el procedimiento **SumaArreglo**.

El procedimiento **SumaArreglo** de la sección 5.5.3 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 al 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, puede ver las instrucciones máquina ocultas que generan las directivas y operadores avanzados de MASM. Seleccione la opción *Desensamblador* en el menú *Depurar | Ventanas*. Esta ventana muestra el código fuente de su programa, junto con las instrucciones máquina 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 no 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

```

5.5.6 Repaso de sección

1. (*Verdadero/Falso*): la directiva PROC empieza un procedimiento y la directiva ENDP lo termina.
2. (*Verdadero/Falso*): es posible definir un procedimiento dentro de un procedimiento existente.
3. ¿Qué pasaría si se omitiera la instrucción RET de un procedimiento?
4. ¿Cómo se utilizan las palabras *Recibe* y *Devuelve* en la documentación sugerida para los procedimientos?
5. (*Verdadero/Falso*): la instrucción CALL mete en la pila el desplazamiento de la instrucción CALL.
6. (*Verdadero/Falso*): la instrucción CALL mete en la pila el desplazamiento de la instrucción que va después de CALL.
7. (*Verdadero/Falso*): la instrucción RET saca el valor de la parte superior de la pila y lo coloca en el apuntador de instrucciones.
8. (*Verdadero/Falso*): el ensamblador de Microsoft no permite las llamadas a procedimientos anidados, a menos que se utilice el operador NESTED en la definición del procedimiento.
9. (*Verdadero/Falso*): en modo protegido, cada llamada a procedimiento utiliza un mínimo de 4 bytes de almacenamiento en la pila.
10. (*Verdadero/Falso*): los registros ESI y EDI no pueden usarse para pasar parámetros a los procedimientos.
11. (*Verdadero/Falso*): el procedimiento **SumaArreglo** (sección 5.5.3) recibe un apuntador a cualquier arreglo de dobles palabras.
12. (*Verdadero/Falso*): el operador USES le permite nombrar todos los registros que se modifican dentro de un procedimiento.
13. (*Verdadero/Falso*): el operador USES sólo genera instrucciones PUSH, por lo que el programador debe codificar las instrucciones POP por su cuenta.
14. (*Verdadero/Falso*): la lista de registros en la directiva USES debe utilizar comas para separar los nombres de los registros.
15. ¿Qué instrucción(es) en el procedimiento **SumaArreglo** (sección 5.5.3) tendría(n) que modificarse para poder acumular un arreglo de palabras de 16 bits? Cree una versión así de SumaArreglo y pruébela.

5.6 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 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 de 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.

5.6.1 Programa para sumar enteros (diseño)

A continuación se muestran las especificaciones para un programa simple, al que llamaremos **Suma de enteros**:

Escriba 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
    PedirEnteros
    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, reciban un entero como entrada, y muestren un entero en pantalla:

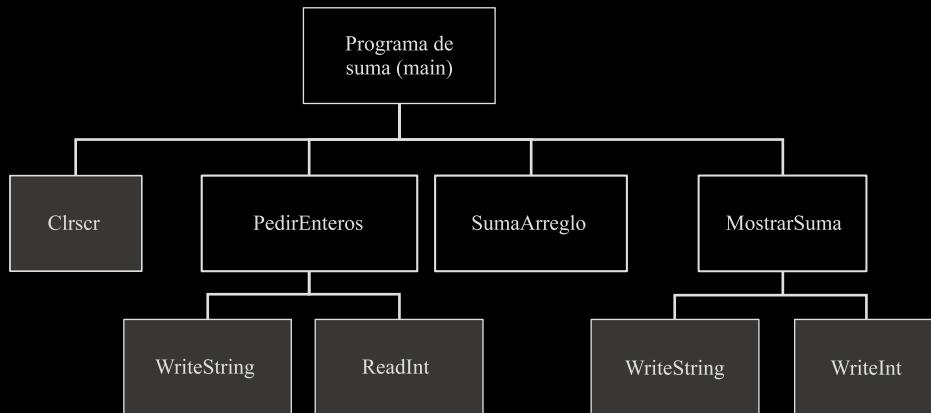
```

Main
    Clrsr                                ; 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 de la figura 5-10, conocido como *diagrama de estructura*, describe la estructura del programa. Los procedimientos de la biblioteca de enlace están sombreados.

FIGURA 5-10 Diagrama de estructura para el programa de suma.



Programa maestro Vamos a crear una versión mínima del programa, a la que llamaremos *programa maestro*. Este programa sólo contiene procedimientos vacíos (o casi vacíos). El programa se ensambla y se ejecuta, pero en realidad no hace nada útil:

```

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: Clrsr, 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
;         enteros tipo doble palabra, ECX = tamaño del arreglo.
; 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

```

Un programa maestro nos proporciona la oportunidad de asignar todas las llamadas a los procedimientos, estudiar las dependencias entre ellos, y posiblemente mejorar el diseño estructural antes de codificar los detalles. Use comentarios en cada procedimiento para explicar su propósito y los requerimientos de los parámetros.

5.6.2 Implementación de la suma de enteros

Vamos a completar el programa de sumas. Declararemos un arreglo de tres enteros y utilizaremos una constante definida para el tamaño del arreglo, en caso de que necesitemos cambiarlo posteriormente:

```

CUENTA_ENTEROS = 3
arreglo DWORD CUENTA_ENTEROS DUP (?)

```

Se utiliza un par de cadenas como indicadores en la pantalla:

```

cad1 BYTE "Escriba un entero con signo: ",0
cad2 BYTE "La suma de los enteros es: ",0

```

El procedimiento **main** borra la pantalla, pasa un apuntador de arreglo al procedimiento **PedirEnteros**, llama a **SumaArreglo** y a **MostrarSuma**:

```

call ClrsCr
mov esi,OFFSET arreglo
mov ecx,CUENTA_ENTEROS
call PedirEnteros
call SumaArreglo
call MostrarSuma

```

- **PedirEnteros** llama a **WriteString** para pedir un entero al usuario. Después llama a **ReadInt** para recibir el entero del usuario y almacena el entero en el arreglo al que apunta ESI. Un ciclo ejecuta estos pasos varias veces.
- **SumaArreglo** calcula y devuelve la suma de un arreglo de enteros.
- **MostrarSuma** muestra un mensaje en la pantalla (“La suma de los enteros es:”) y llama a **WriteInt** para mostrar el entero en EAX.

Listado del programa terminado La siguiente lista muestra el programa de Sumas completo:

```

TITLE Programa de suma de enteros      (Suma2.asm)

; Este programa pide tres enteros al usuario,
; los almacena en un arreglo, calcula la suma del
; arreglo y muestra la suma.
; Última actualización: 06/01/2006

```

```
INCLUDE Irvine32.inc

CUENTA_ENTEROS = 3

.data
cad1 BYTE      "Escriba un entero con signo: ",0
cad2 BYTE      "La suma de los enteros es: ",0
arreglo DWORD CUENTA_ENTEROS DUP(?)

.code
main PROC
    call ClrsCr
    mov  esi,OFFSET arreglo
    mov  ecx,CUENTA_ENTEROS
    call PedirEnteros
    call SumaArreglo
    call MostrarSuma
    exit
main ENDP

;-----
PedirEnteros PROC USES ecx edx esi
;
; Pide al usuario un número arbitrario de enteros
; y los inserta en un arreglo.
; Recibe: ESI apunta al arreglo, ECX = tamaño del arreglo
; Devuelve: nada
;-----
L1: mov  edx,OFFSET cad1          ; "Escriba un entero con signo"
    call WriteString            ; muestra la cadena
    call ReadInt                ; lee entero y lo coloca en EAX
    call CrLf                   ; avanza a la siguiente línea de salida
    mov  [esi],eax              ; almacena en el arreglo
    add  esi,TYPE DWORD         ; siguiente entero
    loop L1                     ; repite para el tamaño del arreglo
    ret
PedirEnteros ENDP

;-----
SumaArreglo PROC USES esi ecx
;
; Calcula la suma de un arreglo de enteros de 32 bits.
; Recibe: ESI apunta al arreglo, ECX = número
; de elementos del arreglo
; Devuelve: EAX = suma de los elementos del arreglo
;-----
L1: mov  eax,0                  ; establece la suma a cero
    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
    ret
SumaArreglo ENDP

;-----
MostrarSuma PROC USES edx
;
; Muestra la suma en la pantalla
; Recibe: EAX = la suma
```

```
; Devuelve: nada
;-----
    mov    edx,OFFSET cad2      ; "La suma de..."
    call   WriteString
    call   WriteInt            ; muestra EAX
    call   Crlf
    ret
MostrarSuma ENDP
END main
```

5.6.3 Repaso de sección

1. ¿Cómo se llama el proceso de dividir las tareas extensas en tareas más pequeñas?
2. ¿Qué procedimientos en el diseño del programa de Sumas (sección 5.6.1) se encuentran en la biblioteca Irvine32?
3. ¿Qué es un *programa maestro*?
4. (Verdadero/Falso): el procedimiento **SumaArreglo** del programa de Sumas (sección 5.6.1) hace referencia directa al nombre de una variable tipo arreglo.
5. ¿Qué líneas en el procedimiento **PedirEnteros** del programa de Sumas (sección 5.6.1) tendría que modificarse para poder manejar un arreglo de palabras de 16 bits? Cree una versión y pruébelo.
6. Dibuje un diagrama de flujo para el procedimiento **PedirEnteros** del programa de Sumas (en la sección 5.5.4 presentamos los diagramas de flujo).

5.7 Resumen del capítulo

Este capítulo presenta la biblioteca de enlace del libro, para facilitarle a usted el procesamiento de las operaciones de entrada-salida en las aplicaciones de lenguaje ensamblador.

La tabla 5-1 presenta la mayoría de los procedimientos de la biblioteca de enlace Irvine32. El listado más actualizado de todos los procedimientos está disponible en el sitio Web del libro.

El *programa de prueba de la biblioteca* en la sección 5.3.3 demuestra una variedad de funciones de entrada-salida de la biblioteca Irvine32. Genera y muestra una lista de números aleatorios, un vaciado de los registros, y un vaciado de memoria. Muestra enteros en varios formatos y demuestra la entrada/salida con cadenas.

La *pila en tiempo de ejecución* es un arreglo especial que se utiliza como área temporal de almacenamiento para direcciones y datos. El registro ESP almacena un desplazamiento (OFFSET) en alguna ubicación en la pila. A la pila se le conoce como estructura UEPS (*último en entrar, primero en salir*), ya que el último valor que se coloca en la pila es el primero que se saca. Una operación *push* (meter) copia un valor en la pila. Una operación *pop* (sacar) elimina un valor de la pila y lo copia en un registro o variable. A menudo, las pilas almacenan direcciones de retorno de procedimientos, parámetros de procedimientos, variables locales y registros que los procedimientos utilizan en forma interna.

La instrucción PUSH primero decrementa el apuntador de la pila y después copia un operando de origen en la pila. La instrucción POP primero copia el contenido de la pila al que apunta ESP en un operando de destino de 16 o 32 bits, y después incrementa a ESP.

La instrucción PUSHAD mete los registros de propósito general de 32 bits en la pila, y la instrucción PUSHAD hace lo mismo para los registros de propósito general de 16 bits. La instrucción POPAD saca valores de la pila y los coloca en los registros de propósito general de 32 bits, y la instrucción POPA hace lo mismo para los registros de propósito general de 16 bits.

La instrucción PUSHFD mete el registro EFLAGS de 32 bits en la pila, y POPFD saca un valor de la pila y lo coloca en EFLAGS. PUSHF y POPF hacen lo mismo para el registro FLAGS de 16 bits.

El programa *InvCad* (sección 5.4.2) utiliza la pila para invertir una cadena de caracteres.

Un *procedimiento* es un bloque de código con nombre, que se declara mediante las directivas PROC y ENDP. La ejecución de un procedimiento termina con la instrucción RET. El procedimiento **SumaDe**, que se muestra en la sección 5.5.1, calcula la suma de tres enteros. La instrucción CALL ejecuta un procedimiento, insertando la dirección del mismo en el registro apuntador de instrucciones. Cuando el procedimiento termina, la instrucción RET (retorno de procedimiento) regresa al procesador al punto en el programa desde donde se hizo la llamada al procedimiento. Una *llamada a procedimiento anidada* ocurre cuando un procedimiento que se llamó hace una llamada a otro procedimiento antes de regresar.

Una etiqueta de código seguida de un signo de dos puntos es local para su procedimiento circundante. Una etiqueta de código seguida de :: es global, lo que la hace accesible desde cualquier instrucción en el mismo archivo de código fuente.

El procedimiento **SumaArreglo**, que se muestra en la sección 5.5.3, calcula y devuelve la suma de los elementos en un arreglo.

El operador USES, junto con la directiva PROC, nos permite mostrar todos los registros que modifica un procedimiento. El ensamblador genera código que mete los registros al principio del procedimiento y los saca antes de regresar.

Un programa de cualquier tamaño debe diseñarse cuidadosamente, siguiendo un conjunto de especificaciones claras. Un método estándar es utilizar la descomposición funcional (diseño de arriba-abajo) para dividir el programa en procedimientos (funciones). Primero se determina el orden y las conexiones entre los procedimientos, y después se llenan los detalles de cada procedimiento.

5.8 Ejercicios de programación

Cuando escriba programas para resolver los ejercicios de programación, use varios procedimientos siempre que sea posible. Siga el estilo y las convenciones de nomenclatura que se utilizan en este libro, a menos que su instructor le indique lo contrario. Use comentarios explicativos en sus programas al principio de cada procedimiento, y enseguida de las instrucciones que sean complicadas. Como algo extra, es posible que su instructor le pida que proporcione diagramas de flujo o seudocódigo para los programas de las soluciones.

1. Dibujar colores de texto

Escriba un programa que muestre la misma cadena en cuatro colores distintos, usando un ciclo. Llame al procedimiento **SetTextColor** de la biblioteca de enlace del libro. Puede elegir cualquier color, pero tal vez sea más sencillo cambiar el color del texto.

2. Archivo de números de Fibonacci

Reto: usando el Ejercicio de programación 6 del capítulo 4 como punto de inicio, escriba un programa que genere los primeros 47 valores en la serie de *Fibonacci*, los almacene en un arreglo de dobles palabras, y escriba el arreglo de dobles palabras en un archivo en disco. No tiene que realizar la comprobación de errores en la E/S del archivo, ya que no hemos visto todavía el procesamiento condicional. El tamaño de su archivo de salida deberá ser de 188 bytes, debido a que cada doble palabra es de 4 bytes. Use debug.exe o Visual Studio para abrir e inspeccionar el contenido del archivo, que se muestra a continuación en hexadecimal:

```

00000000 01 00 00 00 01 00 00 00 02 00 00 00 03 00 00 00
00000010 05 00 00 00 08 00 00 00 0D 00 00 00 15 00 00 00
00000020 22 00 00 00 37 00 00 00 59 00 00 00 90 00 00 00
00000030 E9 00 00 00 79 01 00 00 62 02 00 00 DB 03 00 00
00000040 3D 06 00 00 18 0A 00 00 55 10 00 00 6D 1A 00 00
00000050 C2 2A 00 00 2F 45 00 00 F1 6F 00 00 20 B5 00 00
00000060 11 25 01 00 31 DA 01 00 42 FF 02 00 73 D9 04 00
00000070 B5 D8 07 00 28 B2 0C 00 DD 8A 14 00 05 3D 21 00
00000080 E2 C7 35 00 E7 04 57 00 C9 CC 8C 00 B0 D1 E3 00
00000090 79 9E 70 01 29 70 54 02 A2 OE C5 03 CB 7E 19 06
000000a0 6D 8D DE 09 38 0C F8 0F A5 99 D6 19 DD A5 CE 29
000000b0 82 3F A5 43 5F E5 73 6D E1 24 19 B1 |

```

3. Suma simple (1)

Escriba un programa que borre la pantalla, posicione el cursor cerca de la mitad de la pantalla, pida al usuario dos enteros, los sume y muestre el resultado.

4. Suma simple (2)

Use el programa de solución del ejercicio anterior como punto de inicio. Deje que este nuevo programa repita los mismos pasos tres veces, usando un ciclo. Borre la pantalla después de cada iteración del ciclo.

5. Enteros aleatorios

Escriba un programa que genere y muestre en pantalla 50 enteros aleatorios entre -20 y $+20$.

6. Cadenas aleatorias

Escriba un programa que genere y muestre 20 cadenas aleatorias, cada una de ellas debe consistir de 10 letras mayúsculas [A..Z].

7. Posiciones aleatorias en la pantalla

Escriba un programa que muestre un solo carácter en 100 posiciones aleatorias en la pantalla, usando un retraso de tiempo de 100 milisegundos. *Sugerencia:* use el procedimiento GetMaxXY para determinar el tamaño actual de la ventana de consola.

8. Matriz de colores

Escriba un programa que muestre un solo carácter en todas las posibles combinaciones de colores de texto y de fondo ($16 \times 16 = 256$). Los colores están numerados del 0 al 15, por lo que puede usar un ciclo anidado para generar todas las combinaciones posibles.

9. Programa de suma

Modifique el programa de Sumas en la sección 5.6.1 de la siguiente manera: seleccione un tamaño de arreglo usando una constante:

```
TAM_ARREGLO = 20  
arreglo DWORD TAM_ARREGLO DUP(?)
```

Escriba un nuevo procedimiento que pida al usuario el número de enteros a procesar. Pase el mismo valor al procedimiento PedirEnteros. Si el usuario introduce un valor más grande que TAM_ARREGLO, muestre un mensaje de error y detenga el procesamiento del arreglo. Por ejemplo,

```
Cuantos enteros se van a sumar? 21  
El arreglo no puede ser mayor de 20
```

Diseñe el programa de tal forma que al cambiar TAM_ARREGLO se actualice de manera automática el mensaje de error que se acaba de mostrar.

Notas finales

- Si desea leer más acerca de los generadores de números aleatorios, consulte el libro *The Art of Computer Programming*, Vol. 2 de Donald Knuth, Addison-Wesley, 1997.
- Esto se resume para desplazar los bits cuatro posiciones a la izquierda, lo cual veremos en el capítulo 7.
- Los diagramas de flujo han desaparecido de la mayoría de los libros de texto de programación para principiantes, ya que no son adecuados para la programación orientada a objetos. En el lenguaje ensamblador siguen siendo útiles.

6

PROCESAMIENTO CONDICIONAL

- 6.1 Introducción
- 6.2 Instrucciones booleanas y de comparación
 - 6.2.1 Las banderas de la CPU
 - 6.2.2 Instrucción AND
 - 6.2.3 Instrucción OR
 - 6.2.4 Instrucción XOR
 - 6.2.5 Instrucción NOT
 - 6.2.6 Instrucción TEST
 - 6.2.7 Instrucción CMP
 - 6.2.8 Cómo establecer y borrar banderas individuales de la CPU
 - 6.2.9 Repaso de sección
- 6.3 Saltos condicionales
 - 6.3.1 Estructuras condicionales
 - 6.3.2 Instrucción Jcond
 - 6.3.3 Tipos de instrucciones de saltos condicionales
 - 6.3.4 Aplicaciones de saltos condicionales
 - 6.3.5 Instrucciones de prueba de bits (opcional)
 - 6.3.6 Repaso de sección
- 6.4 Instrucciones de saltos condicionales
 - 6.4.1 Instrucciones LOOPZ y LOOPE
 - 6.4.2 Instrucciones LOOPNZ y LOOPNE
 - 6.4.3 Repaso de sección
- 6.5 Estructuras condicionales
 - 6.5.1 Instrucciones IF con estructura de bloque
 - 6.5.2 Expresiones compuestas
 - 6.5.3 Ciclos WHILE
 - 6.5.4 Selección controlada por tablas
 - 6.5.5 Repaso de sección
- 6.6 Aplicación: máquinas de estado finito
 - 6.6.1 Validación de una cadena de entrada
 - 6.6.2 Validación de un entero con signo
 - 6.6.3 Repaso de sección
- 6.7 Directivas de decisión
 - 6.7.1 Comparaciones con signo y sin signo
 - 6.7.2 Expresiones compuestas
 - 6.7.3 Directivas .REPEAT y .WHILE
- 6.8 Resumen del capítulo
- 6.9 Ejercicios de programación

6.1 Introducción

Un lenguaje de programación que permite tomar decisiones nos deja modificar el flujo de control, mediante una técnica conocida como bifurcación condicional. Todas las instrucciones IF, switch o de ciclo condicional que se utilizan en los lenguajes de alto nivel tienen una lógica de bifurcación integrada. El lenguaje ensamblador, a pesar de ser tan primitivo, cuenta con todas las herramientas necesarias para la lógica de toma de decisiones. En este capítulo veremos cómo funciona la traducción, desde las instrucciones condicionales de alto nivel al código de implementación de bajo nivel.

Los programas que tratan con dispositivos de hardware deben ser capaces de manipular bits individuales en los números. Los bits individuales deben probarse, borrarse y activarse. El cifrado y la compresión de datos también dependen de la manipulación de bits. Le mostraremos cómo realizar estas operaciones en el lenguaje ensamblador.

Este capítulo debe responder ciertas preguntas básicas:

- ¿Cómo puedo usar las operaciones booleanas que se presentaron en el capítulo 1 (AND, OR, NOT)?
- ¿Cómo escribo una instrucción IF en el lenguaje ensamblador?
- ¿Cómo traducen los compiladores las instrucciones IF anidadas en el lenguaje máquina?
- ¿Cómo puedo activar y borrar los bits individuales en un número binario?
- ¿Cómo puedo realizar el cifrado simple de datos binarios?
- ¿Cómo se diferencian los números con signo de los números sin signo en las expresiones booleanas?
- ¿Qué es una máquina de estado finito?
- ¿La instrucción GOTO es realmente dañina?¹

Este capítulo sigue un método de *abajo hacia arriba*, empezando con las bases binarias que hay detrás de la lógica de programación. A continuación veremos cómo la CPU compara los operandos de las instrucciones, mediante la instrucción CMP y las banderas de estado del procesador. Por último, reuniremos todo y le mostraremos cómo utilizar el lenguaje ensamblador para implementar estructuras lógicas características de los lenguajes de alto nivel.

6.2 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 (vea la tabla 6-1).

Tabla 6-1 Instrucciones booleanas seleccionadas.

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 activa las banderas de la CPU en forma apropiada
BT, BTC, BTR, BTS	Copia el bit <i>n</i> del operando de origen a la bandera de Acarreo, y complementa/restablece/activa el mismo bit en el operando de destino (se verá en la sección 6.3.5)

6.2.1 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 Acarreo se activa cuando una instrucción genera un resultado demasiado grande (o muy pequeño) para el operando de destino, cuando se le considera como 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 Desbordamiento se activa cuando una instrucción genera un resultado inválido con signo.
- La bandera Paridad se activa cuando una instrucción genera un número par de bits 1 en el byte inferior del operando de destino.

6.2.2 Instrucción AND

La instrucción AND realiza una operación AND booleana (a nivel de bits) entre cada par de bits coincidentes en dos operandos, y coloca el resultado en el operando de destino:

`AND destino,origen`

Se permiten 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 $x \wedge y$:

x	y	$x \wedge y$
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:

$$\begin{array}{r}
 & \begin{array}{r} 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \end{array} \\
 \text{AND } & \begin{array}{r} 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array} \\
 \hline
 \text{Se borran} & \boxed{\begin{array}{r} 0 & 0 & 0 & 0 \end{array} | \begin{array}{r} 1 & 0 & 1 & 1 \end{array}} \quad \text{Sin cambio}
 \end{array}$$

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 mayúscula. Si comparamos los códigos ASCII de la letra A y de la letra a, queda claro que sólo el bit 5 es diferente:

0	1	1	0	0	0	0	1	= 61h ('a')
0	1	0	0	0	0	1	= 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 a 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

```

6.2.3 Instrucción OR

La instrucción OR realiza una operación OR 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:

OR *reg,reg*
 OR *reg,mem*
 OR *reg,imm*
 OR *mem,reg*
 OR *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 (del capítulo 1) describe la expresión booleana $x \vee y$:

x	y	$x \vee y$
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:

$$\begin{array}{r}
 & 00111011 \\
 \text{OR} & \underline{00001111} \\
 \text{Sin cambio} & \boxed{00111111} \quad \text{Se activan}
 \end{array}$$

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. Si, por ejemplo, AL = 05h, podemos aplicarle un OR con 30h para convertirlo en el código ASCII para el dígito 5 (35h):

$$\begin{array}{r}
 & 00000101 \quad 05h \\
 \text{OR} & \underline{00110000} \quad 30h \\
 & 00110101 \quad 35h, '5'
 \end{array}$$

Las instrucciones en lenguaje máquina para hacer esto son:

mov dl,5	<i>; valor binario</i>
or dl,30h	<i>; lo convierte a ASCII</i>

Banderas La instrucción OR siempre borra las banderas Acarreo y Desbordamiento. Modifica las banderas Signo, Cero y Paridad de acuerdo con el valor del operando de destino. Por ejemplo, puede aplicar OR a un número con sí mismo (o cero) para obtener cierta información acerca 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

6.2.4 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 describe la expresión booleana $x \oplus y$:

x	y	$x \oplus y$
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 al valor opuesto (se complementa). XOR se invierte a sí mismo cuando se aplica dos veces al mismo operando. La siguiente tabla de verdad muestra que, cuando se aplica OR exclusivo al bit x con el bit y dos veces, se revierte a su valor original:

x	y	$x \oplus y$	$(x \oplus y) \oplus y$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Como veremos en la sección 6.3.4, esta propiedad “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 del operando de destino.

Comprobación de la bandera Paridad La bandera Paridad indica si el *bite 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 al número con puros ceros:

<code>mov al,10110101b</code>	<code>; 5 bits = paridad par</code>
<code>xor al,0</code>	<code>; La bandera Paridad se borra (PO)</code>
<code>mov al,11001100b</code>	<code>; 4 bits = paridad par</code>
<code>xor al,0</code>	<code>; La bandera Paridad se activa (PE)</code>

A menudo, los depuradores utilizan PE para indicar paridad par y 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:

<code>mov ax,64C1h</code>	<code>; 0110 0100 1100 0001</code>
<code>xor ah,al</code>	<code>; La bandera Paridad se activa (PE)</code>

Imagine 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 B_0 a B_3 , podemos calcular la paridad como $B_0 \text{ XOR } B_1 \text{ XOR } B_2 \text{ XOR } B_3$.

6.2.5 Instrucción NOT

La instrucción NOT cambia el valor de todos los bits en un operando. Al resultado se le llama *complemento a uno*. Se permiten los siguientes tipos de operandos:

`NOT reg`
`NOT mem`

Por ejemplo, el complemento a uno de F0h es 0Fh:

<code>mov al,11110000b</code>	
<code>not al</code>	<code>; AL = 00001111b</code>

Banderas Ninguna bandera se ve afectada por la instrucción NOT.

6.2.6 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. Suponga que deseamos saber si el bit 0 o el bit 3 están activos en el registro AL. Podemos usar la siguiente instrucción para averiguarlo:

<code>test al,00001001b</code>	<code>; prueba los bits 0 y 3</code>
--------------------------------	--------------------------------------

(Al valor 00001001 en este ejemplo se le 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:

<code>0 0 1 0 0 1 0 1</code>	<code><- valor de entrada</code>
<code>0 0 0 0 1 0 0 1</code>	<code><- valor de prueba</code>
<code>0 0 0 0 0 0 0 1</code>	<code><- resultado: ZF = 0</code>
<code>0 0 1 0 0 1 0 0</code>	<code><- valor de entrada</code>
<code>0 0 0 0 1 0 0 1</code>	<code><- valor de prueba</code>
<code>0 0 0 0 0 0 0 0</code>	<code><- resultado: ZF = 1</code>

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.

6.2.7 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 destino 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 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 ax,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
```

6.2.8 Cómo 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, aplique un OR al bit más alto de un operando con 1. Para borrar la bandera Signo, aplique 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, use la instrucción STC; para borrar la bandera Acarreo, use CLC:

```
stc ; activa la bandera Acarreo
clc ; borra la bandera Acarreo
```

Para activar la bandera Desbordamiento, sume dos valores de byte positivos que produzcan una suma negativa. Para borrar la bandera Desbordamiento, aplique 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
```

6.2.9 Repaso de sección

1. En la siguiente secuencia de instrucciones, muestre en binario el valor modificado de AL en donde se indica:

```
mov al,01101111b
and al,00101101b ; a.
mov al,6Dh
and al,4Ah ; b.
mov al,00001111b
or al,61h ; c.
mov al,94h
xor al,37h ; d.
```

2. En la siguiente secuencia de instrucciones, muestre en hexadecimal el valor modificado de AL en donde se indica:

```
mov al,7Ah
not al ; a.
mov al,3Dh
and al,74h ; b.
mov al 9Bh
or al,35h ; c.
mov al,72h
xor al,0DCh ; d.
```

3. En la siguiente secuencia de instrucciones, muestre los valores de las banderas Acarreo, Cero y Signo en donde se indica:

```
mov al,00001111b
test al,2 ; a. CF= ZF= SF=
mov al,6
cmp al,5 ; b. CF= ZF= SF=
mov al,5
cmp al,7 ; c. CF= ZF= SF=
```

4. Escriba una sola instrucción usando operandos de 16 bits, que borre los 8 bits superiores de AX y que no cambie los 8 bits inferiores.

5. Escriba una sola instrucción usando operandos de 16 bits, que active los 8 bits superiores de AX y que no cambie los 8 bits inferiores.
6. Escriba una sola instrucción (distinta de NOT) que invierta todos los bits en EAX.
7. Escriba instrucciones que activen la bandera Cero si el valor de 32 bits en EAX es par, y que borre la bandera CERO si EAX es impar.
8. Escriba una sola instrucción que convierta un carácter en mayúscula en AL a minúsculas, pero que no modifique a AL si ya contiene una letra en minúscula.
9. Escriba una sola instrucción que convierta un dígito ASCII en AL a su correspondiente valor binario. Si AL ya contiene un valor binario (00h a 09h), que lo deje sin cambios.
10. *Reto:* escriba instrucciones para calcular la paridad del operando de memoria de 32 bits. *Sugerencia:* use la fórmula que presentamos antes en esta sección: $B_0 \text{ XOR } B_1 \text{ XOR } B_2 \text{ XOR } B_3$

6.3 Saltos condicionales

6.3.1 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 segundo 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
.
.
L1:
```

Ejemplo 2 La instrucción AND en el siguiente ejemplo realiza un AND a nivel de bits en el registro DL, lo cual afecta a la bandera Cero. La instrucción JNZ (salta si no es Cero) salta si la bandera Cero se borra:

```
and    dl,10110000b
jnz   L2                      ; salta si ZF = 0
.
.
L2:
```

6.3.2 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 con 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.

Antes del Intel 386, el rango de salto estaba limitado a un desplazamiento de 1 byte (positivo o negativo), partiendo de la ubicación de la siguiente instrucción después del salto. Los procesadores IA-32 pueden saltar a cualquier parte dentro del mismo segmento de memoria.

Uso de la instrucción CMP Suponga que deseamos saltar a la ubicación L1 cuando AX es igual a 5. En el siguiente ejemplo, suponga 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, 6
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
```

6.3.3 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 los 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 tabla 6-2 muestra un listado de saltos con base en los valores específicos de las banderas de la CPU: Cero, Acarreo, Desbordamiento, Paridad y Signo.

Comparaciones de igualdad

La tabla 6-3 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*

Tabla 6-2 Saltos 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 relacionales en álgebra. Por ejemplo, en la expresión $X < Y$, X puede llamarse *opIzq* y Y puede llamarse *opDer*.

Tabla 6-3 Saltos con base en la evaluación de la igualdad.

Nemónico	Descripción
JE	Salta si es igual (<i>opIzq</i> = <i>opDer</i>)
JNE	Salta si no es igual (<i>opIzq</i> ≠ <i>opDer</i>)
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 L1           ; 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 tabla 6-4 se muestran los saltos que se basan específicamente en comparaciones de enteros sin signo. Este tipo de salto es útil cuando se comparan valores sin 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).

Tabla 6-4 Saltos con base en comparaciones sin signo.

Nemónico	Descripción
JA	Salta si es mayor (si $opIzq > opDer$)
JNBE	Salta si no es menor o igual (igual que JA)
JAE	Salta si es mayor o igual (si $opIzq \geq opDer$)
JNB	Salta si no es menor (igual que JAE)
JB	Salta si es menor (si $opIzq < opDer$)
JNAE	Salta si no es mayor o igual (igual que JB)
JBE	Salta si es menor o igual (si $opIzq \leq opDer$)
JNA	Salta si no es mayor (igual que JBE)

Comparaciones con signo

La tabla 6-5 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).

Tabla 6-5 Saltos con base en comparaciones con signo.

Nemónico	Descripción
JG	Salta si es mayor (si $opIzq > opDer$)
JNLE	Salta si no es menor o igual (igual que JG)
JGE	Salta si es mayor o igual (si $opIzq \geq opDer$)
JNL	Salta si no es menor (igual que JGE)
JL	Salta si es menor (si $opIzq < opDer$)
JNGE	Salta si no es mayor o igual (igual que JL)
JLE	Salta si es menor o igual (si $opIzq \leq opDer$)
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
j1 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
jge L1 ; 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
j1 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 rangos (sección 6.4).

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 dirección 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 es 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 por un desplazamiento de byte con signo, MASM genera un error llamado *fueras de rango de salto relativo*. Suponiendo que las instrucciones tengan una longitud 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.

6.3.4 Aplicaciones de saltos condicionales

Prueba de los bits de estado

A menudo, las instrucciones como AND, OR, NOT, CMP y TEST van seguidas de 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, suponga que un operando de 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 algunos 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          ; ¿todos 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 entonces
jae L1                     ; salta a L1
mov dx,bx                  ; de lo contrario, mueve BX a DX
                           ; DX contiene el entero mayor
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 contrario, 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:

```

Aplicación: búsqueda secuencial de un arreglo

Una tarea común es buscar valores en un arreglo, que cumplan con ciertos criterios. Cuando se encuentra el primer valor coincidente, se puede mostrar su valor o devolver un apuntador a su ubicación. Le demostraremos lo fácil que es hacer esto mediante el uso de un arreglo de enteros. El programa *ExploraArreglo.asm* busca el primer valor distinto de cero en un arreglo de enteros de 16 bits. Si encuentra uno, muestra el valor; en caso contrario, muestra un mensaje que indica que no pudo encontrarse un valor:

```

TITLE Explorar un arreglo          (ExploraArreglo.asm)
; Explora un arreglo en busca del primer valor distinto de cero.
; Última actualización: 06/01/2006

INCLUDE Irvine32.inc

.data
arregloInt SWORD 0,0,0,0,1,20,35,-12,66,4,0
;arregloInt SWORD 1,0,0,0
;arregloInt SWORD 0,0,0,0
;arregloInt SWORD 0,0,0,1
msjNinguno BYTE "No se encontró un valor distinto de cero",0

```

Este programa contiene datos de prueba alternativos que se excluyeron como comentarios. Puede quitar el signo de punto y coma de estas líneas para probar el programa con distintas configuraciones de datos.

```

.code
main PROC
    mov  ebx,OFFSET arregloInt ; apunta al arreglo
    mov  ecx,LENGTHOF arregloInt ; contador del ciclo

L1:
    cmp  WORD PTR [ebx],0      ; compara el valor con cero
    jnz  encontrado            ; encontró un valor
    add  ebx,2                  ; apunta al siguiente
    loop L1                    ; continúa el ciclo
    jmp  noEncontrado          ; no se encontró un valor

encontrado:                   ; en caso contrario, lo muestra
    movsx eax,WORD PTR [ebx]
    call WriteInt
    jmp  quit

noEncontrado:                ; muestra mensaje "no se encontró"
    mov  edx,OFFSET msjNinguno
    call WriteString

quit:
    call crlf
    exit
main ENDP
END main

```

Aplicación: cifrado de cadenas

La sección 6.2.4 mostró que la instrucción XOR tiene una propiedad interesante. Si se aplica un XOR a un entero X con Y y al valor resultante se le aplica un XOR con Y otra vez, el valor producido es X:

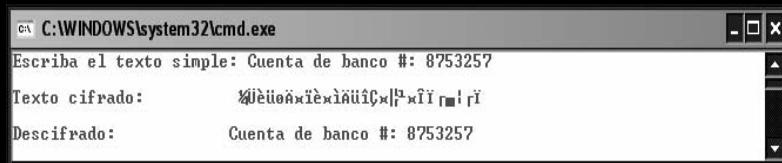
$$((X \otimes Y) \otimes Y) = X$$

Esta propiedad “reversible” de XOR proporciona una forma sencilla de realizar el cifrado de datos: un mensaje de *texto simple* que introduce el usuario se transforma en una cadena ininteligible llamada *texto cifrado*, mediante la aplicación de un XOR entre cada uno de sus componentes con un carácter de una tercera cadena, conocida como *clave*. El texto cifrado puede almacenarse o transmitirse a una ubicación remota, sin que personas autorizadas puedan leerlo. El destinatario utiliza la clave para descifrar el texto y producir el texto simple original.

Programa de ejemplo El siguiente programa utiliza el *cifrado simétrico*, un proceso mediante el cual se utiliza la misma clave para los procesos de cifrado y descifrado. Se llevan a cabo los siguientes pasos, en orden:

- El usuario introduce el texto simple.
 - El programa utiliza una clave de un solo carácter para cifrar el texto simple, produciendo el texto cifrado que se muestra en la pantalla.
 - El programa descifra el texto cifrado, para producir y mostrar el texto simple original.

He aquí un ejemplo de los resultados que genera el programa:



Listado del programa He aquí un listado completo del programa:

```

TITLE Programa de cifrado          (Cifrado.asm)
; Este programa demuestra el cifrado simétrico
; simple mediante el uso de la instrucción XOR.
; Ejemplo del capítulo 6.
; Última actualización: 06/01/2006

INCLUDE Irvine32.inc
CLAVE = 239                      ; cualquier valor entre 1-255
MAXBUF = 128                       ; tamaño máximo del búfer

.data
sIndicador    BYTE "Escriba el texto simple: ",0
sCifrado      BYTE "Texto cifrado:           ",0
sDescifrado   BYTE "Descifrado:             ",0
bufer         BYTE    MAXBUF+1 DUP(0)
tamBufer      DWORD   ?

.code
main PROC
    call  IntroducirLaCadena        ; introduce el texto simple
    call  TraducirBufer            ; cifra el búfer
    mov   edx,OFFSET sCifrado       ; muestra el mensaje cifrado
    call  MostrarMensaje
    call  TraducirBufer            ; descifra el búfer
    mov   edx,OFFSET sDescifrado   ; muestra el mensaje descifrado
    call  MostrarMensaje
    exit
main ENDP

```

```
-----  
IntroducirLaCadena PROC  
;  
; Pide al usuario una cadena de texto simple. Guarda la cadena  
; y su longitud.  
; Recibe: nada  
; Devuelve: nada  
-----  
    pushad  
    mov    edx,OFFSET sIndicador ; muestra un indicador  
    call   WriteString  
    mov    ecx,MAXBUF           ; cuenta máxima de caracteres  
    mov    edx,OFFSET bufer    ; apunta al búfer  
    call   ReadString          ; recibe la cadena de entrada  
    mov    tamBufer,eax        ; guarda la longitud  
    call   Crlf  
    popad  
    ret  
IntroducirLaCadena ENDP  
-----  
MostrarMensaje PROC  
;  
; Muestra el mensaje cifrado o descifrado.  
; Recibe: EDX apunta al mensaje  
; Devuelve: nada  
-----  
    pushad  
    call   WriteString  
    mov    edx,OFFSET bufer    ; muestra el búfer  
    call   WriteString  
    call   Crlf  
    call   Crlf  
    popad  
    ret  
MostrarMensaje ENDP  
-----  
TraducirBufer PROC  
;  
; Traduce la cadena mediante un OR exclusivo con cada  
; byte y el byte de la clave de cifrado.  
; Recibe: nada  
; Devuelve: nada  
-----  
    pushad  
    mov    ecx,tamBufer        ; contador del ciclo  
    mov    esi,0                 ; índice 0 en el búfer  
L1:  
    xor    bufer[esi],CLAVE      ; traduce un byte  
    inc    esi                  ; apunta al siguiente byte  
    loop   L1  
    popad  
    ret  
TraducirBufer ENDP  
END main
```

Los ejercicios del capítulo sugieren una mejora a este programa: use una clave de cifrado que contenga varios caracteres para cifrar y descifrar el texto simple. El usuario puede introducir la clave.

6.3.5 Instrucciones de prueba de bits (opcional)

Las instrucciones BT, BTC, BTR y BTS se llaman en 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 subprocessamiento 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 subprocesso del programa. En el sitio Web del libro podrá ver un ejemplo que describe un caso simple del subprocessamiento múltiple.

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, a la bandera Acarreo se le asigna el valor del bit 7 en la variable llamada **semaforo**:

```
.data
semaforo WORD 10001000b
.code
BT semaforo, 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 Acarreo:

```
mov ax, semaforo
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 Acarreo. En la sección 7.2.3 del capítulo 7 veremos la instrucción SHR).

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 **semaforo**, y se complementa el mismo bit:

```
.data
semaforo WORD 10001000b
.code
BTC semaforo, 6 ; CF = 0, semaforo=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 que BT y BTC. En el siguiente ejemplo, a la bandera Acarreo se le asigna el valor del bit 7 en semaforo y se borra el mismo bit:

```
.data
semaforo WORD 10001000b
.code
    ; CF = 1; semaforo=00001000b
BTR semaforo,7
```

Instrucción BTS

La instrucción BTS (prueba y activación de bit) selecciona el bit n en el primer operando, lo copia a la bandera 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 semaforo y después se activa el mismo bit:

```
.data
semaforo WORD 10001000b
.code
    ; CF = 0, semaforo=11001000b
BTS semaforo,6
```

6.3.6 Repaso de sección

1. ¿Qué banderas de estado de la CPU se utilizan en comparaciones sin signo?
2. ¿Qué banderas de estado de la CPU se utilizan en comparaciones con signo?
3. ¿Qué instrucción de salto condicional se basa en el contenido de ECX?
4. (Sí/No): ¿son equivalentes las instrucciones JA y JNBE? Explique su respuesta.
5. (Sí/No): ¿son equivalentes las instrucciones JB y JL? Explique su respuesta.
6. ¿Qué instrucción de salto es equivalente a la instrucción JNA?
7. ¿Qué instrucción de salto es equivalente a la instrucción JNGE?
8. (Sí/No): ¿saltará el siguiente código a la etiqueta llamada **Destino**?

```
mov ax,8109h
cmp ax,26h
jg Destino
```

9. (Sí/No): ¿saltará el siguiente código a la etiqueta llamada **Destino**?

```
mov ax,-30
cmp ax,-50
jg Destino
```

10. (Sí/No): ¿saltará el siguiente código a la etiqueta llamada **Destino**?
11. Escriba instrucciones que salten a la etiqueta L1, cuando el entero sin signo en DX sea menor o igual al entero en CX.
12. Escriba instrucciones que salten a la etiqueta L2, cuando el entero con signo en DX sea mayor que el entero en CX.
13. Escriba instrucciones para borrar los bits 0 y 1 en AL. Si el operando de destino es igual a cero, se debe hacer un salto a la etiqueta L3. En caso contrario, hay que saltar a la etiqueta L4.
14. *Reto:* dados los datos

```
semaforo WORD 10001000b
```

describa la diferencia en el estado resultante de los registros y las banderas entre la secuencia de instrucciones

```
mov ax,semaforo  
shr ax,7  
xor semaforo,01000000b
```

y la instrucción

BTC semaforo, 6

6.4 Instrucciones de saltos condicionales

6.4.1 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 de las banderas de estado.

Un programa que se ejecuta en modo de direccionamiento real utiliza a CX como el contador de ciclo predeterminado en la instrucción LOOPZ. Si desea forzar a que ECX sea el contador de ciclo, use mejor la instrucción LOOPZD.

6.4.2 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 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 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.

6.4.3 Repaso de sección

- (Verdadero/Falso): la instrucción LOOPE salta a una etiqueta cuando (y sólo cuando) la bandera Cero tiene un cero.
- (Verdadero/Falso): la instrucción LOOPNZ salta a una etiqueta cuando ECX es mayor que cero y la bandera Cero es cero.
- (Verdadero/Falso): la etiqueta de destino de una instrucción LOOPZ no debe estar más alejada de -128 o +127 bytes de la instrucción que sigue inmediatamente después de LOOPZ.
- Modifique el ejemplo de LOOPNZ en la sección 6.4.2, de forma que explore el primer valor negativo en el arreglo. Cambie la declaración de datos de manera apropiada, para que empiece con valores positivos.
- Reto:* el ejemplo de LOOPNZ en la sección 6.4.2 depende de un valor centinela para manejar la posibilidad de que no se pueda encontrar un valor positivo. ¿Qué pasaría si elimináramos el centinela?

6.5 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. Consideraremos 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.

A menudo, los estudiantes de ciencias computacionales toman un curso de *construcción de compiladores*, en donde implementan un compilador de un lenguaje de programación. Las técnicas de optimización de código que veremos aquí pueden ser útiles en un curso de ese tipo.

6.5.1 Instrucciones IF con estructura de bloque

En la mayoría de los lenguajes de alto nivel, una estructura IF implica que una expresión booleana debe 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 del `else` de la instrucción es opcional. El diagrama de flujo de la figura 6-1 muestra las dos rutas de bifurcación en una estructura IF condicional, las cuales se etiquetan como *verdadero* y *falso*.

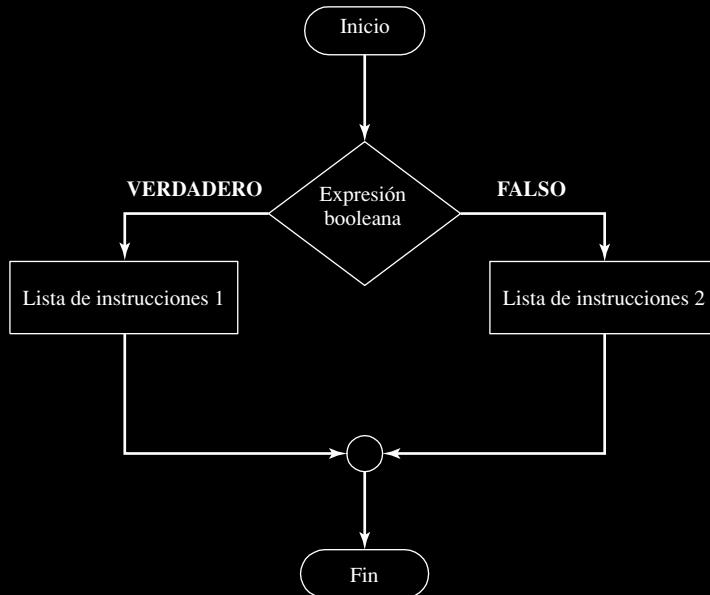
Ejemplo 1 Usando la sintaxis de Java/C++, se ejecutan dos instrucciones de asignación si `op1` es igual a `op2`:

```

if( op1 == op2 )
{
    X = 1;
    Y = 2;
}

```

FIGURA 6–1 Diagrama de flujo de una estructura IF.



La única manera de traducir esta instrucción IF en lenguaje ensamblador es utilizar una 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:
      mov    X,1
      mov    Y,2
L2:

```

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
      mov    Y,2
L2:

```

El mismo código de lenguaje de alto nivel puede traducirse a lenguaje ensamblador de muchas formas. Cuando se muestren ejemplos de código compilado en este capítulo, estos sólo representarán lo que podría producir un compilador hipotético.

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 pseudocódigo, 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, 8192           ; 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 la sección 14.2 hablaremos sobre los clústeres 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 tipo doble palabra con signo. El operador > se implementa mejor usando JNG, el complemento de IG:

```
mov eax,op1
cmp eax,op2           ; ¿op1 > op2?
jng A1               ; no: llama a Rutina2
call Rutina1         ; sí: 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 entradas, se realiza un rastreo manual por el código fuente, y se verifica la ruta de ejecución y las salidas producidas por la subrutina. Veamos cómo se hace esto. Suponga 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 Rutina3
```

```
; procesa la instrucción IF-ELSE interior
4:      mov     eax,X
5:      cmp     eax,Y                      ; ¿X > Y?
6:      jg      L1                         ; sí: llama a Rutina1
7:      call    Rutina2                   ; no: llama a Rutina2
8:      jmp     L3                         ; y termina
9: L1:   call    Rutina1                  ; llama a la Rutina1
10:    jmp    L3                         ; y termina
11: L2:   call    Rutina3
12: L3:
```

La tabla 6-6 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 verificaron las rutas de ejecución resultantes.

Tabla 6-6 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

6.5.2 Expresiones compuestas

Operador AND lógico

El lenguaje ensamblador implementa con facilidad las expresiones booleanas compuestas que contienen operadores AND. Considere el siguiente pseudocódigo, en el que se asume que las variables son enteros sin signo:

```
if (a1 > b1) AND (b1 > c1)
{
    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,b1                      ; primera expresión...
        ja     L1
        jmp    siguiente
L1:   cmp    b1,c1                      ; 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:

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 y 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           ; sí: establece bandera = verdadero

L1:  cmp bl,cl          ; ¿BL > CL?
    jna siguiente        ; no: la expresión 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 pseudocódigo 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            ; verdadero: establece X = 1
siguiente:
```

Para una expresión compuesta dada, hay por lo menos varias formas en que ésta puede implementarse en lenguaje ensamblador.

6.5.3 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, val2           ; if not (val1 < val2)
    jnl  finwhile            ; sale del ciclo
    inc  eax                ; val1++;
    dec  val2                ; val2--;
    jmp  @@while              ; repite el ciclo
finwhile:
    mov  val1, eax            ; guarda el nuevo valor para val1

```

EAX es un proxy (sustituto) para **val1** dentro del ciclo. Las referencias a **val1** deben ser a través de EAX. Se 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 en 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;
int TamArreglo = sizeof arreglo / sizeof ejemplo;
int indice = 0;
int suma = 0;
while( indice < TamArreglo )
{
    if( arreglo[indice] > ejemplo )
    {
        suma += arreglo[indice];
        indice++;
    }
}

```

Antes de codificar este ciclo en lenguaje ensamblador, utilizaremos el diagrama de flujo de la figura 6-2 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 = indice, 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. Observe la correlación directa entre las etiquetas del diagrama de flujo y las etiquetas que se utilizan en el siguiente código fuente (vea *DiagramaFlujo.asm*):

```

.data
suma DWORD 0
ejemplo DWORD 50
arreglo DWORD 10,60,20,33,72,89,45,65,72,18
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*4], edx
    jg    L3
    jmp    L4

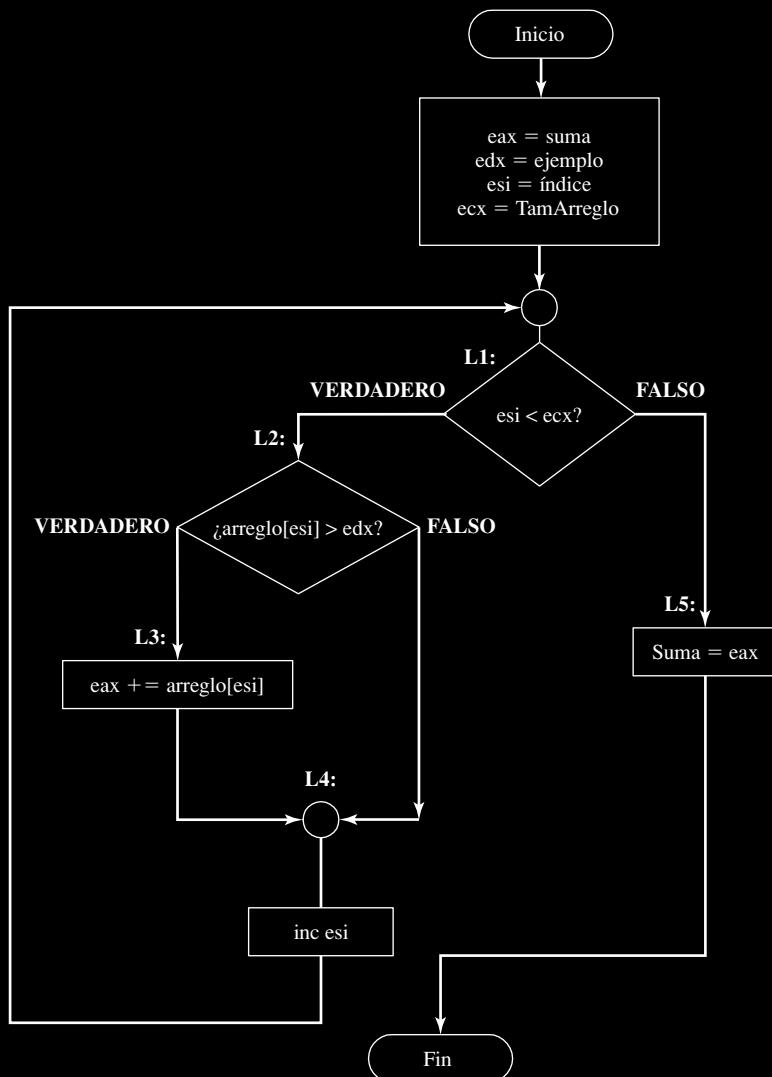
L3: add    eax,arreglo[esi*4]

L4: inc    esi
    jmp    L1

L5: mov    suma,eax
  
```

Una pregunta de repaso al final de la sección 6.5 le dará la oportunidad de mejorar este código.

FIGURA 6-2 Ciclo que contiene una instrucción IF.



6.5.4 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_          ; dirección de procedimiento
              BYTE   'B'
              DWORD Proceso_B
              (etc.)
```

Supongamos que Proceso_A, Proceso_B, Proceso_C y Proceso_D se encuentran en las direcciones 120h, 130h, 140h y 150h, respectivamente. La tabla se ordenaría en memoria como se muestra en la figura 6.3.

FIGURA 6-3 Tabla de desplazamiento de procedimientos.

'A'	00000120	'B'	00000130	'C'	00000140	'D'	00000150
					Dirección del Proceso_B		

Valor de búsqueda

Programa de ejemplo En el siguiente programa de ejemplo (*TablaProc.asm*), 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 desplazamientos 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 mayuscula: ",0
```

Defina una cadena de mensaje separada para cada procedimiento:

```
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
    mov    ebx,OFFSET TablaCasos        ; lee un carácter y lo coloca en AL
    mov    ecx,NumeroDeEntradas        ; apunta EBX a la tabla
    L1:                                ; contador del ciclo
        cmp    al,[ebx]                 ; ¿se encontró coincidencia?
        jne    L2                      ; no: continúa
        call   NEAR PTR [ebx + 1]        ; sí: 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                           ; sale de la búsqueda
L2:                                ; apunta a la siguiente entrada
    add   ebx,TamanoEntrada        ; repite hasta que ECX = 0
    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    edx,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.

6.5.5 Repaso de sección

Notas: en todas las expresiones compuestas, utilice la evaluación de corto circuito. Suponga que val1, val2 y val3 son variables de 16 bits.

1. Implemente el siguiente seudocódigo en lenguaje ensamblador:

```
if( bx > cx )
    X = 1;
```

2. Implemente el siguiente seudocódigo en lenguaje ensamblador:

```
if( dx <= cx )
    X = 1;
else
    X = 2;
```

3. Implemente el siguiente seudocódigo en lenguaje ensamblador:

```
if( val1 > cx AND cx > dx )
    X = 1;
else
    X = 2;
```

4. Implemente el siguiente seudocódigo en lenguaje ensamblador:

```
if( bx > cx OR bx > val1 )
    X = 1;
else
    X = 2;
```

5. Implemente el siguiente seudocódigo en lenguaje ensamblador:

```
if( bx > cx AND bx > dx) OR ( dx > ax )
    X = 1;
else
    X = 2;
```

6. En el programa de la sección 6.54, ¿por qué es mejor dejar que el ensamblador calcule NumeroDeEntradas, en vez de asignar una constante tal como NumeroDeEntradas = 4?

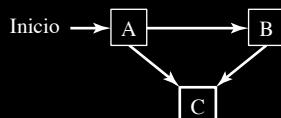
7. *Reto:* vuelva a escribir el código de la sección 6.5.3, de manera que sea funcionalmente equivalente, pero que utilice menos instrucciones.

6.6 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 figura 6-4 se muestra un ejemplo simple. Cada nodo representa un estado del programa, y cada flanco representa una transición de un 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 letras. Uno o más estados se designan como *estados terminales*, los cuales se muestran con 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.

FIGURA 6-4 Máquina de estado finito simple.



Los grafos dirigidos tienen muchas aplicaciones útiles en ciencias computacionales, relacionadas con las estructuras dinámicas de datos y las técnicas avanzadas de búsqueda.

6.6.1 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 clave, 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:

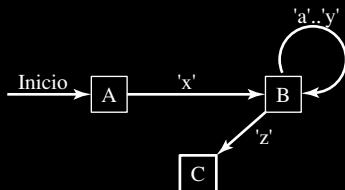
- El siguiente carácter de entrada no corresponde con ninguna transición del estado actual.
- Se llega al fin de la entrada y el estado actual no es un estado terminal.

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 figura 6-5 describe esta sintaxis. Cada transición se identifica mediante un tipo específico de entrada. Por ejemplo, la transición del 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 del estado B al estado C sólo ocurre cuando la letra **z** se lee del flujo de entrada.

FIGURA 6-5 FSM para una cadena.



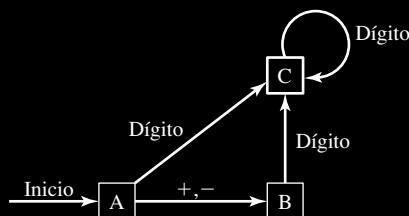
Si se llega al fin del flujo de entrada mientras el programa está en el estado A o B, se produce una condición de error debido a que sólo el estado C se marca como estado terminal. La FSM reconoce las siguientes cadenas de entrada:

xaabcdefgz
xz
xyyqqrrrstuvwxyz

6.6.2 Validación de un entero con signo

En la figura 6-6 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 un número máximo establecido de dígitos implicados por el diagrama.

FIGURA 6-6 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 en el programa 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, hay que comprobar para ver si el usuario oprimió la tecla Intro para terminar la entrada.
- Una o más instrucciones de comparación verifican cada posible transición que nos lleve hacia fuera de ese estado. Cada comparación 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 estado B
    cmp al, '-'                  ; ¿signo - a la izquierda?
    je EstadoB                   ; ir al estado B
    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 figura 6-7 representa el código adjunto a la etiqueta **EstadoA**.

Implementación de la FSM El siguiente programa implementa la máquina de estado finito de la figura 6-6, que describe a un entero con signo.

```
TITLE Máquina de estado finito          (Finito.asm)
; Este programa implementa una máquina de estado finito, que
; acepta un entero con un signo a la izquierda opcional.
; Última actualización: 06/01/2006

INCLUDE Irvine32.inc
CLAVE_INTRO = 13
.data
MsjEntradaInvalida BYTE "Entrada invalida",13,10,0

.code
main PROC
    call Clrscr

EstadoA:
    call ObtenerSiguiente          ; lee siguiente carácter y lo coloca en AL
    cmp al, '+'                  ; ¿signo + a la izquierda?
    je EstadoB                   ; ir al estado B
    cmp al, '-'                  ; ¿signo - a la izquierda?
    je EstadoB                   ; ir al estado B
    call IsDigit                 ; ZF = 1 si AL contiene un dígito
    jz EstadoC                   ; ir al estado C
    call MostrarMsjError         ; se encontró entrada inválida
    jmp Salir

EstadoB:
    call ObtenerSiguiente          ; lee siguiente carácter y lo coloca en AL
    call IsDigit                 ; ZF = 1 si AL contiene un dígito
    jz EstadoC                   ; ir al estado C
    call MostrarMsjError         ; se encontró entrada inválida
    jmp Salir
```

```

EstadoC:
    call ObtenerSiguiente      ; lee siguiente carácter y lo coloca en AL
    call IsDigit                ; ZF = 1 si AL contiene un dígito
    jz EstadoC
    cmp al,CLAVE_INTRO         ; ¿se oprimió la tecla Intro?
    je Salir                   ; sí: salir
    call MostrarMsjError       ; no: se encontró entrada inválida
    jmp Salir

Salir:
    call Crlf
    exit
main ENDP

;-----
; ObtenerSiguiente PROC
;
; Lee un carácter de la entrada estándar.
; Recibe: nada
; Devuelve: AL contiene el carácter
;-----
    call ReadChar           ; entrada del teclado
    call WriteChar          ; imprime en pantalla
    ret
ObtenerSiguiente ENDP

;-----
; MostrarMsjError PROC
;
; Muestra un mensaje de error indicando que
; el flujo de entrada contiene entrada ilegal.
; Recibe: nada.
; Devuelve: nada
;-----
    push edx
    mov edx,OFFSET MsjEntradaInvalida
    call WriteString
    pop edx
    ret
MostrarMsjError ENDP
END main

```

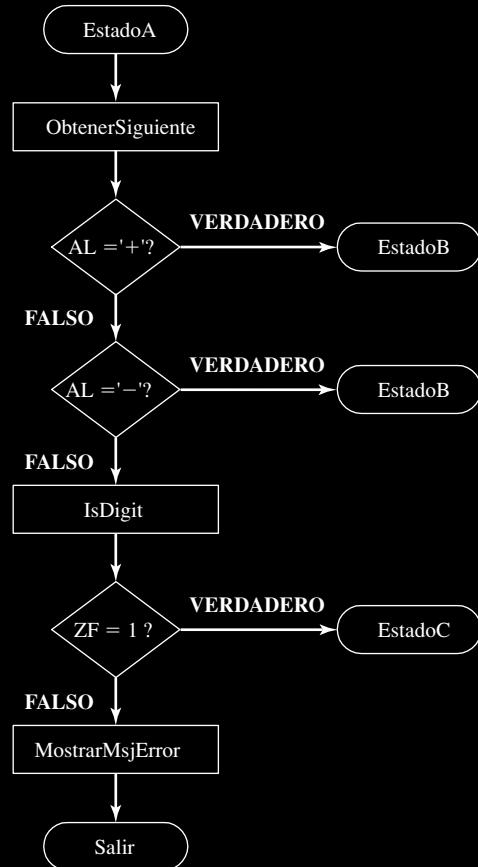
El procedimiento **IsDigit** de la biblioteca de enlace del libro activa la bandera Cero si el carácter en AL es un dígito decimal válido. En caso contrario, se borra la bandera Cero:

```

;-----
; IsDigit PROC
;
; Determina si el carácter en AL es un
; dígito decimal válido.
; Recibe: AL = carácter
; Devuelve: ZF=1 si AL contiene un dígito decimal
; válido; en caso contrario, ZF=0.
;-----
    cmp al,'0'
    jb ID1                  ; ZF = 0 cuando se realiza el salto
    cmp al,'9'
    ja ID1                  ; ZF = 0 cuando se realiza el salto
    test ax,0                ; establece ZF = 1
ID1: ret
IsDigit ENDP

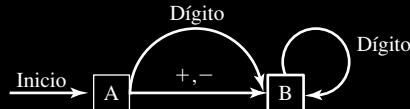
```

FIGURA 6-7 Diagrama de flujo de la FSM de un entero con signo.



6.6.3 Repaso de sección

1. ¿De qué tipo de estructura de datos es la máquina de estado finito (una aplicación específica)?
2. En un diagrama de máquina de estado finito, ¿qué representan los nodos?
3. En un diagrama de máquina de estado finito, ¿qué representan los flancos?
4. En la máquina de estado finito de un entero con signo (sección 6.6.2), ¿a qué estado se llega cuando la entrada consiste en “+5”?
5. En la máquina de estado finito de un entero con signo (sección 6.6.2), ¿cuántos dígitos puede haber después de un signo negativo?
6. ¿Qué ocurre en una máquina de estado finito cuando no hay más entrada disponible y el estado actual no es un estado terminal?
7. ¿Funcionaría la siguiente simplificación de una máquina de estado finito de un entero decimal con signo de igual forma que la que se muestra en la sección 6.6.2? Si no, ¿por qué no?



8. *Reto:* haga un diagrama de una máquina de estado finito que reconozca números reales sin exponentes. Se requiere el punto decimal. Algunos ejemplos son +3.5, -4.2342, 5., .2.

6.7 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 condicion1
    instrucciones
[.ELSEIF condicion2
    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)
(val2 != ebx) && !ACARREO?
```

En la tabla 6-7 se muestra una lista completa de operadores relacionales y lógicos.

Tabla 6-7 Operadores relacionales y lógicos en tiempo de ejecución.

Operador	Descripción
<i>expr1</i> == <i>expr2</i>	Devuelve verdadero cuando <i>expr1</i> es igual a <i>expr2</i>
<i>expr1</i> != <i>expr2</i>	Devuelve verdadero cuando <i>expr1</i> no es igual a <i>expr2</i>
<i>expr1</i> > <i>expr2</i>	Devuelve verdadero cuando <i>expr1</i> es mayor que <i>expr2</i>
<i>expr1</i> ≥ <i>expr2</i>	Devuelve verdadero cuando <i>expr1</i> es mayor o igual que <i>expr2</i>
<i>expr1</i> < <i>expr2</i>	Devuelve verdadero cuando <i>expr1</i> es menor que <i>expr2</i>
<i>expr1</i> ≤ <i>expr2</i>	Devuelve verdadero cuando <i>expr1</i> es menor o igual que <i>expr2</i>
! <i>expr</i>	Devuelve verdadero cuando <i>expr</i> es falsa
<i>expr1</i> && <i>expr2</i>	Realiza un AND lógico entre <i>expr1</i> y <i>expr2</i>
<i>expr1</i> // <i>expr2</i>	Realiza un OR lógico entre <i>expr1</i> y <i>expr2</i>
<i>expr1</i> & <i>expr2</i>	Realiza un AND a nivel de bits entre <i>expr1</i> y <i>expr2</i>
CARRY?	Devuelve verdadero si 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, asegúrese de comprender por completo las instrucciones de bifurcación condicional. Además, cuando se ensambla un programa que contiene directivas de decisión, inspeccione el archivo de listado para asegurarse de que el código generado por MASM sea lo que usted espera.

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 usted. 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 lenguaje ensamblador:

```
mov    eax,6
cmp    eax,val1
jbe    @C0001           ; salta en comparación sin signo
mov    resultado,1
@C0001:
```

El nombre de etiqueta @C001 lo crea el ensamblador. Esto se hace de una manera que garantice que todas las etiquetas dentro del mismo procedimiento serán únicas.

6.7.1 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
    mov resultado,1
@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:
```

Comparación 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.

6.7.2 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 expresión1 || expresión2
    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: establecerPosicionCursor

El procedimiento **EstablecerPosicionCursor**, que se muestra en el siguiente ejemplo, realiza comprobación de rangos en sus dos parámetros de entrada, DH y DL (vea *EstCur.asm*). 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 de 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

Suponga 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 (vea *Inscrip.asm*):

```

.data
VERDADERO = 1
FALSO = 0
promedioCalif WORD 275           ; valor de prueba
creditos      WORD 12            ; valor de prueba
SePuedeRegistrar BYTE ?
.code
    mov SePuedeRegistrar,FALSO
    .IF promedioCalif > 350
        mov SePuedeRegistrar,VERDADERO
    .ELSEIF (promedioCalif > 250) && (creditos <= 16)
        mov SePuedeRegistrar,VERDADERO
    .ELSEIF (creditos <= 12)
        mov SePuedeRegistrar,VERDADERO
    .ENDIF

```

La tabla 6-8 lista el código correspondiente generado por el ensamblador, que podemos ver mediante la ventana *Desensamblar* del depurador de Microsoft Visual Studio. Lo hemos arreglado un poco aquí para que sea más claro. El código generado por MASM aparecerá en el archivo de listado de código fuente si utilizamos la opción de línea de comandos /Sg al ensamblar los programas.

Tabla 6-8 Ejemplo de inscripción, código generado por MASM.

<pre> mov byte ptr SePuedeRegistrar,FALSO cmp word ptr promedioCalif,350 jbe @C0006 mov byte ptr SePuedeRegistrar,VERDADERO jmp @C0008 @C0006: cmp word ptr promedioCalif,250 jbe @C0009 cmp word ptr creditos,16 ja @C0009 mov byte ptr SePuedeRegistrar,VERDADERO jmp @C0008 @C0009: cmp word ptr creditos,12 ja @C0008 mov byte ptr SePuedeRegistrar,VERDADERO @C0008: </pre>
--

6.7.3 Directivas .REPEAT y .WHILE

Las directivas .REPEAT y .WHILE ofrecen alternativas para escribir nuestros propios ciclos, con instrucciones CMP y de salto condicional. Permiten las expresiones condicionales que se listan en la tabla 6-7. 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 del 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, en la sección 6.5.3, le mostramos cómo escribir código en 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 a 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
```

6.8 Resumen del capítulo

A las instrucciones AND, OR, XOR, NOT y TEST se les llama *instrucciones a nivel de bits*, ya que funcionan a nivel de bits. Cada bit en un operando de origen se relaciona con un bit en la misma posición del operando de destino:

- La instrucción AND produce 1 cuando ambos bits de entrada son 1.
- La instrucción OR produce 1 cuando por lo menos uno de los bits de entrada es 1.
- La instrucción XOR produce 1 sólo cuando los bits de entrada son distintos.
- La instrucción TEST realiza una operación AND implícita en el operando de destino, activando las banderas en forma apropiada. El operando de destino no se cambia.
- La instrucción NOT invierte todos los bits en un operando de destino.

La instrucción CMP compara un operando de destino con un operando de origen. Realiza una resta implícita del origen con el destino y modifica las banderas de estado de la CPU en forma apropiada. Por lo general, CMP va seguida de una instrucción de salto condicional, la cual puede producir una transferencia de control a una etiqueta de código.

En este capítulo se muestran cuatro tipos de instrucciones de salto condicional:

- La tabla 6-2 contiene ejemplos de saltos basados en valores específicos de las banderas, como JC (salta si hay acarreo), JZ (salta si es cero), y JO (salta si hay desbordamiento).
- La tabla 6-3 contiene ejemplos de saltos con base en la igualdad, como JE (salta si es igual), JNE (salta si no es igual), y JECXZ (salta si ECX = 0).
- La tabla 6-4 contiene ejemplos de saltos condicionales con base en las comparaciones de enteros sin signo, como JA (salta si es mayor), JB (salta si es menor), y JAE (salta si es mayor o igual).
- La tabla 6-5 contiene ejemplos de saltos basados en comparaciones con signo, como JL (salta si es menor), y JG (salta si es mayor).

La instrucción LOOPZ (LOOPE) se repite cuando se activa la bandera Cero y ECX es mayor que Cero. La instrucción LOOPNZ (LOOPNE) se repite cuando la bandera Cero se borra y ECX es mayor que cero. (En modo de direccionamiento real, LOOPZ y LOOPNZ utilizan el registro CX).

El *cifrado* es un proceso que codifica datos, y el *descifrado* es un proceso que decodifica datos. La instrucción XOR puede usarse para realizar operaciones simples de cifrado y descifrado, un byte a la vez.

Los diagramas de flujo son una herramienta efectiva para representar la lógica de un programa en forma visual. Puede escribir código en lenguaje ensamblador fácilmente, usando un diagrama de flujo como modelo. Es útil adjuntar una etiqueta a cada símbolo del diagrama de flujo y utilizar la misma etiqueta en el código en lenguaje ensamblador.

Una *máquina de estado finito* (FSM) es una herramienta efectiva para validar cadenas que contienen caracteres reconocibles, como los enteros con signo. Es bastante sencillo implementar una máquina de estado finito en el lenguaje ensamblador, si cada estado se representa mediante una etiqueta.

Las directivas .IF, .ELSE, .ELSEIF y .ENDIF evalúan expresiones en tiempo de ejecución y simplifican en forma considerable la codificación en lenguaje ensamblador. En especial, son útiles cuando se codifican expresiones booleanas compuestas complejas. También puede crear ciclos condicionales, usando las directivas .WHILE y .REPEAT.

6.9 Ejercicios de programación

1. ExplorarArreglo usando LOOPZ

Utilice como modelo el programa ExplorarArreglo de la sección 6.3.4 para implementar la búsqueda usando la instrucción LOOPZ. *Opcional:* dibuje un diagrama de flujo del programa.

2. Implementación de ciclos

Implemente el siguiente código de C++ en lenguaje ensamblador, usando las directivas .IF con estructura de bloque y .WHILE. Asuma que todas las variables son enteros de 32 bits con signo:

```
int arreglo[] = {10,60,20,33,72,89,45,65,72,18};
int ejemplo = 50;
int TamArreglo = sizeof arreglo / sizeof ejemplo;
int indice = 0
int suma = 0;
while( indice < TamArreglo )
{
    if( arreglo[indice] <= ejemplo )
    {
        suma += arreglo[indice];
        indice++;
    }
}
```

Opcional: dibuje un diagrama de flujo de su código.

3. Evaluación de calificaciones de una prueba (1)

Usando la siguiente tabla como guía, escriba un programa que pida al usuario que introduzca una calificación de prueba entera, entre 0 y 100. El programa debe mostrar la letra de calificación apropiada:

Rango de calificaciones	Letra de calificación
90 a 100	A
80 a 89	B
70 a 79	C
60 a 69	D
0 a 59	F

Opcional: dibuje un diagrama de flujo del programa.

4. Evaluación de calificaciones de una prueba (2)

Usando el programa de solución del ejercicio anterior como punto de partida, agregue las siguientes características:

- Debe ejecutarse en un ciclo, para poder introducir varias calificaciones.
- Debe acumular un contador del número de calificaciones de la prueba.
- Debe realizar una comprobación de rango en la entrada del usuario: Mostrar un mensaje de error si la calificación de la prueba es menor que 0 o mayor que 100.

Opcional: dibuje un diagrama de flujo del programa.

5. Inscripción universitaria (1)

Usando el ejemplo de Inscripción universitaria de la sección 6.7.2 como punto de partida, haga lo siguiente:

- Vuelva a codificar la lógica usando instrucciones CMP y de salto condicional (en vez de las directivas .IF y .ELSEIF).

- Realice una comprobación de rango en el valor de **creditos**; no puede ser menor que 1 ni mayor que 30. Si se descubre una entrada inválida, muestre un mensaje de error apropiado.

Opcional: dibuje un diagrama de flujo del programa.

6. Inscripción universitaria (2)

Usando el programa de solución del ejercicio anterior como punto de partida, escriba un programa completo que realice lo siguiente:

1. Recibir como entrada del usuario **promedioCalif** y **creditos**. Si el usuario introduce un cero para cualquiera de estos dos valores, hay que detener el programa.
2. Realizar una comprobación de rango en **creditos** y en **promedioCalif**. Los **creditos** deben estar entre 1 y 30. **promedioCalif** debe estar entre 0 y 400. Si cualquiera de estos dos valores se encuentra fuera de rango, mostrar un mensaje de error apropiado.
3. Determinar si la persona puede inscribirse o no (usando el ejemplo anterior) y mostrar un mensaje apropiado.
4. Repetir los pasos del 1 al 3 hasta que el usuario decida terminar.

Opcional: dibuje un diagrama de flujo del programa.

7. Calculadora booleana (1)

Cree un programa que funcione como una calculadora booleana simple para enteros de 32 bits. Debe mostrar un menú que pida al usuario que realice una selección de la siguiente lista:

1. x AND y
2. x OR y
3. NOT x
4. x XOR y
5. Salir del programa

Cuando el usuario elija una opción, llame a un procedimiento que muestre el nombre de la operación que va a realizarse. (Implementaremos las operaciones en el ejercicio posterior a éste).

Opcional: dibuje un diagrama de flujo del programa.

8. Calculadora booleana (2)

Continúe el programa de solución del ejercicio anterior; implemente los siguientes procedimientos:

- **AND_op**: pida al usuario dos enteros hexadecimales. Aplique un AND entre estos dos números y muestre el resultado en hexadecimal.
- **OR_op**: pida al usuario dos enteros hexadecimales. Aplique un OR entre estos dos números y muestre el resultado en hexadecimal.
- **NOT_op**: pida al usuario un entero hexadecimal. Aplique un NOT al entero y muestre el resultado en hexadecimal.
- **XOR_op**: pida al usuario dos enteros hexadecimales. Aplique un OR exclusivo entre estos dos números y muestre el resultado en hexadecimal.

Opcional: dibuje un diagrama de flujo del programa.

9. Probabilidades ponderadas

Escriba un programa que elija al azar de entre tres colores distintos para mostrar texto en la pantalla. Use un ciclo para mostrar 20 líneas de texto, cada una con un color elegido al azar. Las probabilidades para cada color deben ser las siguientes: blanco = 30%, azul = 10%, verde = 60%. *Sugerencia:* genere un entero aleatorio entre 0 y 9. Si el entero resultante se encuentra en el rango de 0 a 2, elija blanco. Si el entero es igual a 3, elija azul. Si el entero está en el rango de 4 a 9, elija verde.

10. Imprimir Fibonacci hasta un desbordamiento

Escriba un programa que calcule la secuencia de números de Fibonacci {1, 1, 2, 3, 5, 8, 13,...} y se detenga sólo cuando se active la bandera Desbordamiento. Muestre cada valor entero decimal sin signo en una línea separada.

11. Cifrado de mensajes

Modifique el programa de cifrado de la sección 6.3.4 de la siguiente manera: Deje que el usuario introduzca una clave de cifrado, que consista de varios caracteres. Use esta clave para cifrar y descifrar el texto simple, aplicando un XOR a cada uno de los caracteres de la clave contra un byte correspondiente en el mensaje. Repita la clave todas las veces que sea necesario, hasta que se traduzcan todos los bytes de texto simple. Por ejemplo, suponga que la clave es igual a “ABXmv#7”. A continuación se muestra cómo se debe alinear la clave con los bytes de texto simple:

Texto simple	T	h	i	s		i	s	a		P	l	a	i	n	t	e	x	t		m	e	s	s	a	g	e	(etc.)	
Clave	A	B	X	m	v	#	7	A	B	X	m	v	#	7	A	B	X	m	v	#	7	A	8	X	m	v	#	7

(La clave se repite hasta que sea igual a la longitud del texto simple...)

12. Probabilidades ponderadas

Cree un procedimiento que reciba un valor N entre 0 y 100. Cuando se haga una llamada al procedimiento, debe haber una probabilidad de $N/100$ de que borre la bandera Cero. Escriba un programa que pida al usuario que escriba un valor de probabilidad entre 0 y 100. El programa deberá llamar a su procedimiento 30 veces, pasarle el mismo valor de probabilidad y mostrar el valor de la bandera Cero, una vez que el procedimiento regrese.

Nota al final

1. Título de un famoso artículo de 1968, por Edsger W. Dijkstra, “Go To Considered Harmful”, disponible en www.acm.org.

7

ARITMÉTICA DE ENTEROS

- 7.1 Introducción
- 7.2 Instrucciones de desplazamiento y rotación
 - 7.2.1 Desplazamientos lógicos y desplazamientos aritméticos
 - 7.2.2 Instrucción SHL
 - 7.2.3 Instrucción SHR
 - 7.2.4 Instrucciones SAL y SAR
 - 7.2.5 Instrucción ROL
 - 7.2.6 Instrucción ROR
 - 7.2.7 Instrucciones RCL y RCR
 - 7.2.8 Desbordamiento con signo
 - 7.2.9 Instrucciones SHLD/SHRD
 - 7.2.10 Repaso de sección
- 7.3 Aplicaciones de desplazamiento y rotación
 - 7.3.1 Desplazamiento de varias dobles palabras
 - 7.3.2 Multiplicación binaria
 - 7.3.3 Visualización de bits binarios
 - 7.3.4 Aislamiento de campos de datos de archivos de MS-DOS
 - 7.3.5 Repaso de sección
- 7.4 Instrucciones de multiplicación y división
 - 7.4.1 Instrucción MUL
 - 7.4.2 Instrucción IMUL
- 7.4.3 Evaluación del rendimiento de las operaciones de multiplicación
- 7.4.4 Instrucción DIV
- 7.4.5 División de enteros con signo
- 7.4.6 Implementación de expresiones aritméticas
- 7.4.7 Repaso de sección
- 7.5 Suma y resta extendidas
 - 7.5.1 Instrucción ADC
 - 7.5.2 Ejemplo de suma extendida
 - 7.5.3 Instrucción SBB
 - 7.5.4 Repaso de sección
- 7.6 Aritmética ASCII y con decimales desempaquetados
 - 7.6.1 Instrucción AAA
 - 7.6.2 Instrucción AAS
 - 7.6.3 Instrucción AAM
 - 7.6.4 Instrucción AAD
 - 7.6.5 Repaso de sección
- 7.7 Aritmética con decimales empaquetados
 - 7.7.1 Instrucción DAA
 - 7.7.2 Instrucción DAS
 - 7.7.3 Repaso de sección
- 7.8 Resumen del capítulo
- 7.9 Ejercicios de programación

7.1 Introducción

Todo lenguaje ensamblador tiene instrucciones que mueven bits en el interior de los operandos. Las instrucciones de *desplazamiento y rotación*, como se conocen, son en especial útiles para controlar dispositivos de hardware, cifrar datos e implementar gráficos de alta velocidad. En este capítulo le explicaremos cómo realizar operaciones de desplazamiento y rotación, y cómo llevar a cabo las operaciones de multiplicación y división de enteros con eficiencia, usando operaciones de desplazamiento.

Después, exploraremos las instrucciones de multiplicación y división de enteros en el conjunto de instrucciones IA-32. Intel clasifica las instrucciones de acuerdo con las operaciones con y sin signo. Mediante

el uso de estas instrucciones, le mostraremos cómo traducir expresiones matemáticas de C++ a lenguaje ensamblador. Los compiladores dividen las expresiones compuestas en secuencias discretas de instrucciones de máquina. La simulación de un compilador le ayudará a comprender mejor la forma en que éstos funcionan, y podrá optimizar manualmente el código en lenguaje ensamblador con más eficiencia. Aprenderá cómo funcionan las reglas de precedencia de los operadores y la optimización de registros a nivel de máquina.

¿Alguna vez se ha preguntado cómo las computadoras suman y restan enteros de varias palabras? Le mostraremos instrucciones como ADC (*suma con acarreo*) y SBB (*resta con préstamo*), las cuales funcionan con enteros de cualquier tamaño. Por último, presentaremos las instrucciones especializadas de Intel para realizar operaciones aritméticas con enteros decimales empaquetados y cadenas de enteros.

7.2 Instrucciones de desplazamiento y rotación

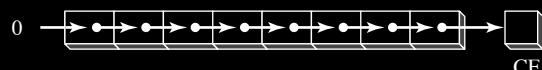
Junto con las instrucciones a nivel de bits que presentamos en el capítulo 6, las instrucciones de desplazamiento son las más características del lenguaje ensamblador. *Desplazar* significa mover bits a la derecha y a la izquierda dentro de un operando. Intel proporciona un conjunto bastante completo de instrucciones en esta área (tabla 7-1), todas las cuales afectan a las banderas Desbordamiento y Acarreo.

Tabla 7-1 Instrucciones de desplazamiento y rotación.

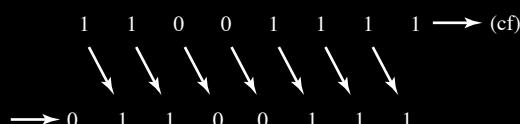
SHL	Desplazamiento a la izquierda
SHR	Desplazamiento a la derecha
SAL	Desplazamiento aritmético a la izquierda
SAR	Desplazamiento aritmético a la derecha
ROL	Rotación a la izquierda
ROR	Rotación a la derecha
RCL	Rotación con acarreo a la izquierda
RCR	Rotación con acarreo a la derecha
SHLD	Desplazamiento de doble precisión a la izquierda
SHRD	Desplazamiento de doble precisión a la derecha

7.2.1 Desplazamientos lógicos y desplazamientos aritméticos

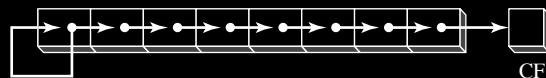
Existen dos formas de desplazar los bits de un operando. La primera, el *desplazamiento lógico*, llena la nueva posición de bit creada con cero. En el siguiente diagrama, un byte se desplaza en forma lógica una posición a la derecha. Observe que al bit 7 se le asigna un 0:



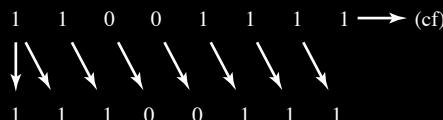
Suponga que ejecutamos un solo desplazamiento lógico a la derecha en el valor binario 11001111, lo cual produce 01100111. El bit inferior se desplaza hacia la bandera Acarreo:



Hay otro tipo de desplazamiento, conocido como *desplazamiento aritmético*. La nueva posición de bit creada se llena con una copia del bit de signo del número original:

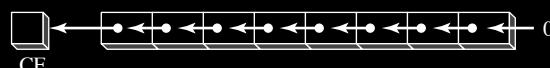


Por ejemplo, el número binario 11001111 tiene un 1 en el bit de signo. Cuando se desplaza aritméticamente 1 bit a la derecha, se convierte en 11100111:

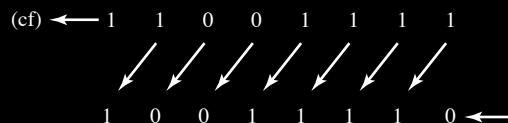


7.2.2 Instrucción SHL

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 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 en SHL es el destino, y el segundo es la cuenta de desplazamientos:

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* puede ser cualquier entero entre 0 y 255. En cualquier procesador Intel, CL puede contener una cuenta de desplazamientos. 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 a la posición del bit inferior se le asigna un cero:

```
mov b1, 8Fh ; BL = 10001111b
shl b1, 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 fuera 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
```

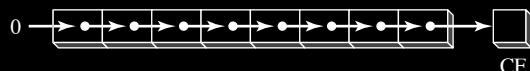
Antes:	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td></tr></table> = 5	0	0	0	0	0	1	0	1
0	0	0	0	0	1	0	1		
Después:	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr></table> = 10	0	0	0	0	1	0	1	0
0	0	0	0	1	0	1	0		

Si el número 10 decimal se desplaza 2 bits a la izquierda, el resultado es el mismo que si se multiplica el 10 por 2^2 :

```
mov dl, 10
shl dl, 2
; (10 * 4) = 40
```

7.2.3 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
shr al, 1
; AL = 11010000b
; 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 a dividir 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:	<table border="1"><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table> = 32	0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0		
Después:	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table> = 16	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0		

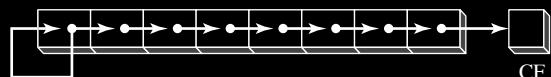
En el siguiente ejemplo, 64 se divide entre 2^3 :

```
mov al, 01000000b
shr al, 3
; AL = 64
; 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.

7.2.4 Instrucciones 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 cómo SAR duplica el bit de signo. AL es negativo antes y después de 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 una potencia de 2, mediante el uso de la instrucción SAR. En el siguiente ejemplo, -128 se divide entre 2^3 . El cociente es -16:

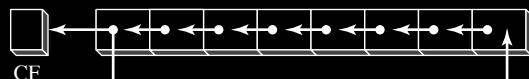
```
mov dl, -128 ; DL = 10000000b
sar dl, 3 ; DL = 11110000b
```

Extensión de AX con signo hacia EAX Suponga que AX contiene un entero con signo y deseamos extender su signo hacia EAX. Primero se desplaza EAX 16 bits a la izquierda, y luego se desplaza aritméticamente 16 bits a la derecha:

```
mov ax, -128 ; EAX = ????FF80h
shl eax, 16 ; EAX = FF800000h
sar eax, 16 ; EAX = FFFFFFF80h
```

7.2.5 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 la 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. Observe 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 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 numero 26h cuatro bits en cualquier dirección, se convierte en 62h:

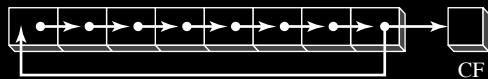
```
mov al, 26h
rol al, 4 ; AL = 62h
```

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
```

7.2.6 Instrucción ROR

La instrucción ROR (rotación a la derecha) desplaza cada bit a la derecha y copia el 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 en la bandera Acarreo como en la posición del bit superior del resultado:

```
mov al,01h ; AL = 00000001b
ror al,1 ; AL = 10000000b, CF = 1
ror al,1 ; AL = 01000000b, 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 = 10000010b, CF = 1
```

7.2.7 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:



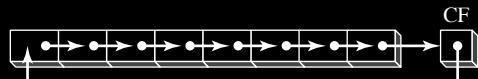
Si imaginamos la bandera Acarreo como un bit adicional que se agrega al extremo superior del operando, RCL se ve como una operación de rotación a la izquierda. En el siguiente ejemplo, la instrucción CLC borra la bandera Acarreo. La primera instrucción RCL mueve el bit superior de BL hacia la bandera Acarreo y desplaza los otros bits a la izquierda. La segunda instrucción RCL mueve la bandera 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
valprueba BYTE 01101010b
.code
shr valprueba,1 ; desplaza LSB hacia 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 = 10001000 0
```

7.2.8 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,-128      ; AL = 10000000b
shr al,1          ; OF = 1, AL = 01000000b
```

El valor de la bandera Desbordamiento es indefinido cuando la cuenta de desplazamientos o rotaciones es mayor que 1.

7.2.9 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 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*

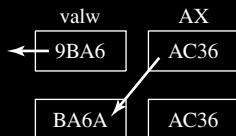
Los siguientes formatos de instrucciones se aplican tanto a SHLD como a SHRD. El operando de *destino* puede ser un registro u operando de memoria, mientras que el operando de *origen* debe ser un registro. El operando *cuenta* puede ser el registro CL o un operando inmediato de 8 bits:

```
SHLD reg16,reg16,CL/imm8
SHLD mem16,reg16,CL/imm8
SHLD reg32,reg32,CL/imm8
SHLD mem32,reg32,CL/imm8
```

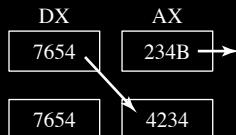
Ejemplo 1 Las siguientes instrucciones desplazan **valw** 4 bits a la izquierda, e insertan los 4 bits superiores de AX en las 4 posiciones de bits inferiores de **valw**:

```
.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 el desplazamiento de bits. Por último, las dos instrucciones pueden usarse al realizar operaciones rápidas de multiplicación y división con enteros muy grandes.

7.2.10 Repaso de sección

1. ¿Qué instrucción mueve cada bit en un operando a la izquierda y copia el bit superior tanto en la bandera Acarreo como en la posición del bit inferior?
2. ¿Qué instrucción mueve cada bit a la derecha, copia el bit inferior en la bandera Acarreo y copia la bandera Acarreo a la posición del bit superior?
3. ¿Qué instrucción desplaza cada bit a la derecha y replica el bit de signo?
4. ¿Qué instrucción realiza la siguiente operación (CF = bandera Acarreo)?

Antes: CF,AL = 1 11010101
Después: CF,AL = 1 10101011

5. Suponga que no hay instrucciones de rotación. Muestre cómo podríamos usar SHR y una instrucción de salto condicional para rotar AL una posición a la derecha.
6. ¿Qué ocurre con la bandera Acarreo cuando se ejecuta la instrucción SHR AX,1?
7. Escriba una instrucción de desplazamiento lógico, que multiplique el contenido de EAX por 16.
8. Escriba una instrucción de desplazamiento lógico que divida a EBX entre 4.
9. Escriba una sola instrucción de rotación que intercambie las mitades superior e inferior del registro DL.
10. Escriba una instrucción SHLD que desplace el bit superior en el registro AX a la posición del bit inferior de DX, y desplace DX un bit a la izquierda.
11. En la siguiente secuencia de código, muestre el valor de AL después de ejecutar cada instrucción de desplazamiento o de rotación:

```
mov al,0D4h
shr al,1
; a.
mov al,0D4h
sar al,1
; b.
mov al,0D4h
sar al,4
; c.
mov al,0D4h
rol al,1
; d.
```

12. En la siguiente secuencia de código, muestre el valor de AL después de ejecutar cada instrucción de desplazamiento o de rotación:

```

mov al,0D4h
ror al,3 ; a.
mov al,0D4h
rol al,7 ; b.
stc
mov al,0D4h
rcl al,1 ; c.
stc
mov al,0D4h
rcr al,3 ; d.

```

13. *Reto*: escriba una serie de instrucciones para desplazar el bit inferior de AX al bit superior de BX, sin utilizar la instrucción SHRD. A continuación, realice la misma operación usando SHRD.
14. *Reto*: una manera de calcular la paridad de un número de 32 bits en EAX es usar un ciclo que desplace cada bit en la bandera Acarreo, y que acumule una cuenta del número de veces que se activó esta bandera. Escriba un código que haga esto, y que active la bandera Paridad en forma apropiada.

7.3 Aplicaciones de desplazamiento y rotación

7.3.1 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 lo cual se le conoce como *orden little-endian*). Los siguientes pasos le mostrarán cómo desplazar un arreglo de este tipo un bit 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:

99999999	99999999	99999999
[esi]	[esi + 4]	[esi + 8]

El programa llamado *DespMulti.asm* implementa el siguiente código. Utilizamos 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 bandera Acarreo

```

La salida del programa muestra los números en binario, antes y después del desplazamiento:

1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001... (etc.)
0100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100... (etc.)

7.3.2 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:

$$\begin{aligned} \text{EAX} * 36 &= \text{EAX} * (32 + 4) \\ &= (\text{EAX} * 32) + (\text{EAX} * 4) \end{aligned}$$

La siguiente figura muestra la multiplicación 123 * 36, que produce 4428, el producto:

$$\begin{array}{r} 01111011 & 123 \\ \times 00100100 & 36 \\ \hline 01111011 & 123 \text{ SHL } 2 \\ + 01111011 & 123 \text{ SHL } 5 \\ \hline 0001000101001100 & 4428 \end{array}$$

Los bits 2 y 5 se activan en el multiplicador (36) y también son los contadores de desplazamientos 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
```

Como ejercicio de capítulo, se le pedirá que generalice este ejemplo y cree un procedimiento que multiplique dos enteros sin signo de 32 bits cualesquiera, mediante el desplazamiento y la suma.

7.3.3 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 con dígitos ASCII binarios
;-----
push ecx
push esi
    mov ecx,32 ; número de bits en EAX
L1: shl eax,1 ; desplaza bit superior hacia bandera Acarreo
```

```

        mov    BYTE PTR [esi], '0'      ; elige 0 como dígito predeterminado
        jnc    L2J                  ; 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

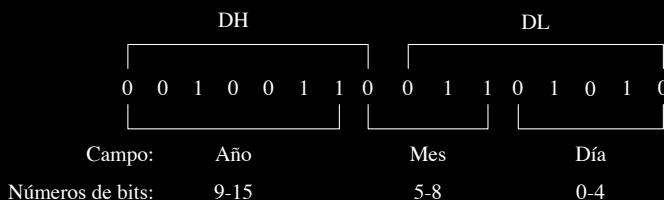
        pop    esi
        pop    ecx
        ret

BinAAsc ENDP

```

7.3.4 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 de 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 la última modificación que se realizó a ese archivo). Los bits del 0 al 4 representan un número de día entre 1 y 31, los bits del 5 al 8 son el número del mes, y los bits del 9 al 15 guardan el número del año. Suponga 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,d1 ; saca una copia de DL  
and al,00011111b ; borra los bits 5-7  
mov dia,al ; guarda en dia
```

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 del año (bits 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 ax,1980 ; el año es relativo a 1980
mov anio.ax ; guarda en el año
```

7.3.5 Repaso de sección

- Escriba una secuencia de instrucciones que desplacen tres bytes de memoria 1 posición de bit a la derecha.
Use la siguiente definición de datos:

arregloBytes BYTE 81h,20h,33h

2. Escriba una secuencia de instrucciones que desplacen tres palabras de memoria 1 posición de bit a la izquierda. Use la siguiente definición de datos:
`arregloBytes WORD 810Dh, 0C064h, 93ABh`
3. Escriba instrucciones ASM que calculen $EAX * 24$, usando la multiplicación binaria.
4. Escriba instrucciones ASM que calculen $EAX * 21$, usando la multiplicación binaria. *Sugerencia:* $21 = 2^4 + 2^2 + 2^0$.
5. ¿Qué cambio se realizaría al programa *BinAAsc* en la sección 7.3.3. si quisiéramos mostrar los bits binarios en orden inverso?
6. La estampa de fecha de un archivo utiliza los bits del 0 al 4 para los segundos, los bits del 5 al 10 para los minutos, y los bits del 11 al 15 para las horas. Escriba instrucciones que extraigan los minutos y copie el valor a una variable tipo byte, llamada **minutosB**.

7.4 Instrucciones de multiplicación y división

Las instrucciones MUL e IMUL 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.

7.4.1 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 multiplicando son del mismo tamaño, y el producto es del 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 tabla 7-2 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 un 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 la aritmética sin signo, por lo que aquí nos enfocaremos 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.

Tabla 7-2 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  a1,5h
mov  b1,10h
mul  b1          ; 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
```

7.4.2 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. Puede utilizar esta información para decidir si debe ignorar o no la mitad superior del producto.

Formatos 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. Asegúrese de verificar una de estas banderas antes de realizar una operación IMUL con dos operandos.

Formatos 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. Asegúrese de 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 de 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 multiplican 48 por 4, produciendo +192 en DX:AX. DX 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 (4823424 * -423), produciendo -2,040,308,352 en EDX:EAX. EDX es una extensión del signo de EAX, por lo que la bandera Desbordamiento se borra:

```
mov eax,+4823424
mov ebx,4
imul ebx ; EDX:EAX = FFFFFFFF86635D80h, OF = 0
```

Las siguientes instrucciones demuestran los formatos de dos operandos:

```
.data
word1 SWORD 4
dword1 SDWORD 4
.code
mov ax,-16 ; AX = -16
mov bx,2 ; BX = 2
imul bx,ax ; BX = -32
imul bx,2 ; BX = -64
imul bx,word1 ; BX = -256
mov eax,-16 ; EAX = -16
mov ebx,2 ; EBX = 2
imul ebx,ea x ; EBX = -32
imul ebx,2 ; EBX = -64
imul ebx,dword1 ; EBX = -256
```

Las siguientes instrucciones de dos operandos demuestran 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,dword1,-16 ; EBX = -64
imul ebx,dword1,-2000000000 ; OF = 1
```

7.4.3 Evaluación del rendimiento de las operaciones de multiplicación

Ahora que hemos visto cómo 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 = 0FFFFFFFh
.data
valInt DWORD 5
tiempoInicio 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 con 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! (Vea el programa *CompararMult.asm*).

7.4.4 Instrucción DIV

La instrucción DIV (división sin signo) realiza la división de enteros con 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	AX	DX
EDX:EAX	r/m32	EAX	EDX

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 debe borrarse 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:

7.4.5 División de enteros con 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 del 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 (en la sección 4.1.5 se explicó la extensión del signo). 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 ; 9Bh
.code
mov al, valByte
cbw ; AL = 9Bh
; AX = FF9Bh
```

La instrucción CWD (convertir palabra a doble palabra) extiende el bit de signo de AX hacia DX:

```
.data
valWord WORD -101 ; FF9Bh
.code
mov ax, valWord ; AX = FF9Bh
cwd ; DX:AX = FFFFFFF9Bh
```

La instrucción CDQ (convertir doble palabra a palabra cuádruple) extiende el bit de signo de EAX hacia EDX:

```
.data
valDword DWORD -101 ; FFFFFFF9Bh
.code
mov eax, valDword
cdq ; EDX:EAX = FFFFFFFFFFFFF9Bh
```

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 8 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. Despues 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 b1,+5 ; divisor
idiv b1 ; AL = -9, AH = -3
```

Ejemplo 2 La división de 16 bits requiere que se extienda el signo de AX hacia DX. El siguiente ejemplo divide -5000 entre 256:

```
.data
valWord WORD -5000
.code
mov ax, valWord ; dividendo, inferior
cwd ; 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 + 50000
.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 b1,10h
div b1 ; AL no puede guardar 100h
```

Cuando este código se ejecuta en MS-Windows, la figura 7-1 muestra el cuadro de diálogo de error resultante, producido por MS-Windows:

FIGURA 7-1 Ejemplo de error por desbordamiento en la división.



Al escribir instrucciones que intenten dividir entre cero, aparece una ventana de cuadro de diálogo similar:

```
mov ax,dividendo
mov b1,0
div b1
```

Utilice 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, evalúe el divisor antes de la división:

```
mov ax,dividendo
mov b1,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")

```

7.4.6 Implementación de expresiones aritméticas

La sección 4.2.5 demostró cómo implementar expresiones aritméticas mediante la suma y la resta. Ahora podemos incluir la multiplicación y la división. Al principio, la implementación de estas expresiones parece ser una actividad más adecuada para los escritores de compiladores, pero hay mucho que obtener a través del estudio práctico. Puede aprender de qué manera los compiladores optimizan el código. Además, 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 del 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 las secciones 7.4.1 y 7.4.2 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 Implemente la siguiente instrucción de C++ en 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    ; muestra mensaje de error
    muyGrande:          ; muestra mensaje de error

```

Si la instrucción MUL genera un producto mayor que 32 bits, la instrucción JC la envía a una etiqueta que maneja el error.

Ejemplo 2 Implemente 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 a EBX, y la división final completa la expresión:

```

    mov    eax,var1          ; lado izquierdo
    mov    ebx,5              ; EDX:EAX = producto
    mul    ebx
    mov    ebx,var2          ; lado derecho
    sub    ebx,3
    div    ebx               ; división final
    mov    var4,eax

```

Ejemplo 3 Implemente 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
idiv var3 ; dividendo con signo extendido
            ; 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
```

7.4.7 Repaso de sección

- Explique por qué no puede ocurrir un desbordamiento cuando se ejecutan las instrucciones MUL e IMUL de un operando.
- ¿Qué diferencia hay entre la instrucción IMUL de un operando y la instrucción MUL, en cuanto a la forma en que generan un producto de la multiplicación?
- ¿Qué tiene que ocurrir para que la instrucción IMUL de un operando active las banderas Acarreo y Desbordamiento?
- Cuando EBX es el operando en una instrucción DIV, ¿qué registro guarda el cociente?
- Cuando BX es el operando en una instrucción DIV, ¿qué registro guarda el cociente?
- Cuando BL es el operando en una instrucción MUL, ¿qué registro guarda el producto?
- Muestre un ejemplo de extensión de signo antes de llamar a la instrucción IDIV con un operando de 16 bits.
- ¿Cuál será el contenido de AX y DX después de la siguiente operación?

```
mov dx,0
mov ax,222h
mov cx,100h
mul cx
```

- ¿Cuál será el contenido de AX después de la siguiente operación?

```
mov ax,63h
mov b1,10h
div b1
```

- ¿Cuál será el contenido de EAX y EDX después de la siguiente operación?

```
mov eax,123400h
mov edx,0
mov ebx,10h
div ebx
```

- ¿Cuál será el contenido de AX y DX después de la siguiente operación?

```
mov ax,4000h
mov dx,500h
mov bx,10h
div bx
```

- Escriba instrucciones que multipliquen -5 por 3 y almacenen el resultado en una variable de 16 bits llamada **val1**.

13. Escriba instrucciones que dividan -276 por 10 y almacenen el resultado en una variable de 16 bits llamada `val1`.
14. Implemente la siguiente expresión de C++ en lenguaje ensamblador, usando operandos de 32 bits sin signo:

$$\text{val1} = (\text{val2} * \text{val3}) / (\text{val4} - 3)$$
15. Implemente la siguiente expresión de C++ en lenguaje ensamblador, usando operandos de 32 bits con signo:

$$\text{val1} = (\text{val2}/\text{val3}) * (\text{val1} + \text{val2})$$

7.5 Suma y resta extendidas

El proceso de *Suma y resta con precisión extendida* se refiera a la suma y resta de números que tengan un tamaño casi ilimitado. Suponga que le piden que escriba un programa en C++ para sumar dos enteros de 1024 bits. ¡La solución no sería fácil! Pero en lenguaje ensamblador, las instrucciones ADC (suma con acarreo) y SBB (resta con préstamo) se adaptan muy bien a este tipo de problema.

7.5.1 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
ADC  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 (FFFFFFFh + FFFFFFFh), produciendo una suma de 64 bits en EDX:EAX: 00000001FFFFFFFEh:

```
mov  edx,0
mov  eax,0FFFFFFFh
add  eax,0FFFFFFFh
adc  edx,0
```

7.5.2 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:

```
;-----
; Suma_Extendida PROC
;
; Calcula la suma de dos enteros extendidos almacenados
; como un arreglo de dobles palabras.
; Recibe: ESI y EDI apuntan a los dos enteros,
; EBX apunta a una variable que guardará la suma, y
; ECX indica el número de dobles palabras que se van a sumar.
; La suma debe ser una doble palabra más grande que
; los operandos de entrada.
;-----
pushad
clc                      ; borra la bandera Acarreo
L1: mov    eax,[esi]        ; obtiene el primer entero
```

```

    adc    eax,[edi]           ; suma el segundo entero
    pushfd
    mov    [ebx],eax           ; guarda la bandera Acarreo
    add    esi,4               ; almacena la suma parcial
    add    edi,4               ; avanza los 3 apuntadores
    add    ebx,4
    popfd
    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 SumaExt.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(0FFFFFFFh)      ; = 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   WriteHex
    call   Crlf
    exit
main ENDP

```

El programa produce el siguiente resultado. La suma produce un acarreo:

```
0000000122C32B0674BB5736
```

7.5.3 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 1 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 00000000FFFFFFFFFFh.

7.5.4 Repaso de sección

1. Describa la instrucción ADC.
2. Describa la instrucción SBB.
3. ¿Cuáles serán los valores de EDX:EAX después de que se ejecuten las siguientes instrucciones?


```
mov edx,10h
mov eax,0A0000000h
add eax,20000000h
adc edx,0
```
4. ¿Cuáles serán los valores de EDX:EAX después de que se ejecuten las siguientes instrucciones?


```
mov edx,100h
mov eax,80000000h
sub eax,90000000h
sbb edx,0
```
5. ¿Cuál será el contenido de DX después de que se ejecuten las siguientes instrucciones (STC activa la bandera Acarreo)?


```
mov dx,5
stc                                ; activa la bandera Acarreo
mov ax,10h
adc dx,ax
```
6. *Reto:* se supone que el siguiente programa debe restar val2 de val1. Busque y corrija todos los errores lógicos (CLC borra la bandera Acarreo):


```
.data
val1 QWORD 20403004362046A1h
val2 QWORD 055210304A2630B2h
resultado QWORD 0
.code
    mov cx,8                  ; contador del ciclo
    mov esi,val1              ; establece índice para iniciar
    mov edi,val2              ; borra la bandera Acarreo
superior:
    mov al,BYTE PTR[esi]       ; obtiene el primer número
    sbb al,BYTE PTR[edi]       ; resta el segundo
    mov BYTE PTR[esi],al        ; guarda el resultado
    dec esi
    dec edi
loop superior
```

7.6 Aritmética ASCII y con decimales desempaquetados

La aritmética de enteros que hemos mostrado hasta ahora en este libro 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. Suponga 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:

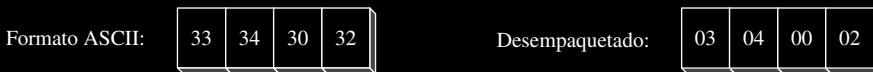
1. Convertir ambos operandos a binario, sumar los valores binarios y convertir la suma de binario a cadenas de dígitos ASCII.

2. Sumar las cadenas de dígitos directamente, mediante una suma sucesiva de cada par de dígitos ASCII ($2 + 6, 0 + 5, 4 + 2$ y $3 + 1$). La suma es una cadena de dígitos ASCII, por lo que puede mostrarse directamente en la pantalla.

La segunda opción requiere el uso de instrucciones especializadas para ajustar la suma después de sumar cada par de dígitos ASCII. A continuación se muestran las cuatro instrucciones que tratan 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 el 3402 en ambos formatos:



(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 la división.

7.6.1 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 a AX en dos dígitos decimales desempaquetados. Una vez en formato desempaquetado, AH y AL pueden convertirse fácilmente a 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 a 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 = 006Ah
aaa                ; AX = 0100h (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 (indice) = longitud de primer_numero - 1
edi (indice) = 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:

```

TITLE Suma ASCII                               (Suma_ASCII.asm)

; Realiza aritmética ASCII con cadenas de dígitos que tienen
; puntos decimales fijos implícitos.
; Last update: 06/01/2006

INCLUDE Irvine32.inc

DESP_DECIMAL = 5                                ; desplazamiento desde la derecha de la cadena
.data
uno_decimal BYTE "100123456789765"           ; 1001234567.89765
dos_decimal BYTE "900402076502015"          ; 9004020765.02015
suma BYTE (SIZEOF uno_decimal + 1) DUP(0),0

.code
main PROC
    ; Empieza en la posición del último dígito.
    mov    esi,SIZEOF uno_decimal - 1
    mov    edi,SIZEOF uno_decimal
    mov    ecx,SIZEOF uno_decimal
    mov    bh,0                                ; establece el valor del acarreo a cero

L1:   mov    ah,0                                ; borra AH antes de la suma
      mov    al,uno_decimal[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,dos_decimal[esi]             ; suma el segundo dígito
      aaa                                ; ajusta la suma (AH = acarreo)
      or     bh,30h                           ; aplica OR al acarreo con acarreo1
      or     al,30h                           ; lo convierte en ASCII
      mov    suma[edi],al                  ; convierte a AL de vuelta en ASCII
      dec    esi                             ; lo guarda en la suma
      dec    edi                             ; retrocede un dígito
      loop   L1                                ; guarda el último dígito del acarreo

    ; Muestra la suma como una cadena.
    mov    edx,OFFSET suma
    call   WriteString
    call   Crlf

    exit
main ENDP
END main

```

He aquí la salida del programa, que muestra la suma sin un punto decimal:

```
1000525533291780
```

7.6.2 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 el 9 del 8 ASCII:

```
.data
val1 BYTE '8'
val2 BYTE '9'
.code
mov ah,0
mov al,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 00FFh. La instrucción AAS convierte a AL en 09h y resta 1 de AH, con lo que lo establece en FFh y activa la bandera Acarreo.

7.6.3 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
aam ; AX = 0300h
```

7.6.4 Instrucción AAD

La instrucción AAD (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 ; AX = 0025h
mov bl,5 ; divisor
div bl ; AX = 0207h
mov cociente,al
mov residuo,ah
```

7.6.5 Repaso de sección

- Escriba una sola instrucción que convierta un entero decimal desempaquetado de dos dígitos en AX a decimal ASCII.
- Escriba una sola instrucción que convierta un entero decimal ASCII en AX a formato decimal desempaquetado.
- Escriba una secuencia de dos instrucciones que convierta un número decimal ASCII de dos dígitos en AX a binario.
- Escriba una sola instrucción que convierta un entero binario sin signo en AX a decimal desempaquetado.
- Reto:* escriba un procedimiento que muestre un valor binario de 8 bits sin signo en formato decimal. Pase el valor binario en AL. El rango de entrada está limitado de 0 a 99, en decimal. El único procedimiento que puede llamar de la biblioteca de enlace del libro es WriteChar. El procedimiento no debe contener más de ocho instrucciones. He aquí una llamada de ejemplo:

```
    mov al,65 ; límite de rango: 0 a 99
    call muestraDecimal8
```

- Reto:* suponga que AX contiene 0072h y que la bandera Acarreo auxiliar se activa como resultado de sumar dos dígitos decimales ASCII desconocidos. Use el Manual de referencia del conjunto de instrucciones IA-32 para determinar qué salida debe producir la instrucción AAA. Explique su respuesta.

7.7 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	WORD	0237h	; 237 decimal
bcd6	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) es relativamente simple.

Dos instrucciones, DAA (ajuste decimal después de la suma) y DAS (ajuste decimal después de la 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.

7.7.1 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 = 83h (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. Uno de los ejercicios del capítulo le pedirá que cree un procedimiento para sumar enteros decimales empaquetados de cualquier tamaño.

7.7.2 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 empaquetados 48 y 85, y ajustan el resultado:

```

mov  b1,48h
mov  al,85h
sub  al,b1          ; 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.

7.7.3 Repaso de sección

1. ¿Bajo qué circunstancias la instrucción DAA activa la bandera Acarreo? Dé un ejemplo.
2. ¿Bajo qué circunstancias la instrucción DAS activa la bandera Acarreo? Dé un ejemplo.
3. Si se suman dos enteros decimales empaquetados con n bytes de longitud, ¿cuántos bytes de almacenamiento deben reservarse para la suma?

4. *Reto:* suponga que AL contiene 3Dh, AF = 0 y CF = 0. Usando el Manual de referencia del conjunto de instrucciones IA-32 como guía, explique los pasos que utiliza la instrucción DAS para convertir AL en decimal empaqetado (37h).

7.8 Resumen del capítulo

Junto con las instrucciones a nivel de bits del capítulo anterior, las instrucciones de desplazamiento se encuentran entre las más características del lenguaje ensamblador. *Desplazar* un número significa mover sus bits a la derecha o a la izquierda.

La instrucción SHL (desplazamiento a la izquierda) desplaza cada bit en un operando de destino a la izquierda, y rellena el bit inferior con 0. Uno de los mejores usos de SHL es para realizar la multiplicación de alta velocidad mediante potencias de 2. Al desplazar cualquier operando n bits a la izquierda, se multiplica ese operando por 2^n . La instrucción SHR (desplazamiento a la derecha) desplaza cada bit a la derecha, sustituyendo el bit superior con un 0. Al desplazar cualquier operando n bits a la derecha, se divide ese operando entre 2^n .

SAL (desplazamiento aritmético a la izquierda) y SAR (desplazamiento aritmético a la derecha) son instrucciones de desplazamiento diseñadas específicamente para desplazar números con signo.

La instrucción ROL (rotación a la izquierda) desplaza cada bit a la izquierda y copia el bit superior a la bandera Acarreo y a la posición del bit inferior. La instrucción ROR (rotación a la derecha) desplaza cada bit a la derecha y copia el bit inferior a la bandera Acarreo y a la posición de bit superior.

La instrucción RCL (rotación a la izquierda con acarreo) desplaza cada bit a la izquierda y copia el bit superior en la bandera Acarreo, la cual se copia primero en el bit inferior del resultado. La instrucción RCR (rotación a la derecha con acarreo) desplaza cada bit a la derecha y copia el bit inferior a la bandera Acarreo. La bandera Acarreo se copia al bit superior del resultado.

Las instrucciones SHLD (desplazamiento doble a la izquierda) y SHRD (desplazamiento doble a la derecha), disponibles en los procesadores IA-32, son muy efectivas para desplazar bits en enteros grandes.

La instrucción MUL multiplica un operando de 8, 16 o 32 bits por AL, AX o EAX. La instrucción IMUL realiza multiplicaciones de enteros con signo. Tiene tres formatos: un operando, dos operandos y tres operandos.

La instrucción DIV realiza divisiones de 8, 16 y 32 bits con enteros sin signo. La instrucción IDIV realiza divisiones de enteros con signo, y utiliza los mismos operandos que la instrucción DIV.

La instrucción CBW (convierte byte a palabra) extiende el bit de signo de AL hacia el registro AH. La instrucción CDQ (convierte doble palabra a palabra cuádruple) extiende el bit de signo de EAX hacia el registro EDX. La instrucción CWD (convierte palabra a doble palabra) extiende el bit de signo de AX hacia el registro DX.

Suma y resta extendidas se refiere al proceso de sumar y restar enteros de un tamaño arbitrario. Las instrucciones ADC y SBB pueden usarse para implementar dichas operaciones de suma y resta. La instrucción ADC (suma con acarreo) suma un operando de origen y el contenido de la bandera Acarreo a un operando de destino. La instrucción SBB (resta con préstamo) resta un operando de origen y el valor de la bandera Acarreo de un operando de destino.

Los enteros *decimales ASCII* almacenan un dígito por byte, el cual se codifica como dígito ASCII. La instrucción AAA (ajuste ASCII después de la suma) convierte el resultado binario de una instrucción ADD o ADC en un decimal ASCII. La instrucción AAS (ajuste ASCII después de la resta) convierte el resultado binario de una instrucción SUB o SBB en un decimal ASCII.

Los enteros *decimales desempaquetados* almacenan un dígito decimal por byte, como un valor binario. La instrucción AAM (ajuste ASCII después de la multiplicación) convierte el producto binario de una instrucción MUL en un decimal desempaquetado. La instrucción AAD (ajuste ASCII antes de la división) convierte un dividendo decimal desempaquetado en binario, como preparación para la instrucción DIV.

Los enteros *decimales empaquetados* almacenan dos dígitos decimales por byte. La instrucción DAA (ajuste decimal después de la suma) convierte el resultado binario de una instrucción ADD o ADC en un decimal empaquetado. La instrucción DAS (ajuste decimal después de la resta) convierte el resultado binario de una instrucción SUB o SBB en un decimal empaquetado.

7.9 Ejercicios de programación

1. Procedimiento de suma extendida

Modifique el procedimiento **Suma_Extendida** en la sección 7.5.2 para sumar dos enteros de 256 bits (32 bytes).

2. Procedimiento de resta extendida

Cree y pruebe un procedimiento llamado **Resta_Extendida**, que reste dos enteros binarios de un tamaño arbitrario. Restricciones: el tamaño de almacenamiento de los dos enteros debe ser el mismo, y su tamaño debe ser un múltiplo de 32 bits.

3. MostrarHoraArchivo

La estampa de hora de una entrada en un directorio de archivos de MS-DOS utiliza los bits del 0 al 4 para el número de incrementos de 2 segundos, los bits del 5 al 10 para los minutos, y los bits del 11 al 15 para las horas (reloj de 24 horas). Por ejemplo, el siguiente valor binario indica la hora 02:16:14, en formato *hh:mm:ss*:

00010 010000 00111

Escriba un procedimiento llamado **MostrarHoraArchivo** que reciba un valor de hora de un archivo binario en el registro AX y muestre el tiempo en el formato *hh:mm:ss*.

4. Desplazamiento de varias dobles palabras

Escriba un procedimiento que desplace un arreglo de cinco enteros de 32 bits, usando la instrucción SHRD (sección 7.2.9). Escriba un programa que pruebe su procedimiento y muestre el arreglo.

5. Multiplicación rápida

Escriba un procedimiento llamado **MultiplicacionRapida** que multiplique un entero de 32 bits sin signo por EAX, usando sólo las operaciones de desplazamiento y suma. Pase el entero al procedimiento en el registro EBX y devuelva el producto en el registro EAX. Escriba un programa corto de prueba, que llame al procedimiento y muestre el producto (vamos a suponer que el producto nunca será mayor de 32 bits).

6. Máximo divisor común (GCD)

El máximo divisor común de dos enteros es el entero más grande que puede dividir ambos enteros. El algoritmo GCD implica la división de enteros en un ciclo, lo cual se describe mediante el siguiente código en C++:

```
int GCD(int x, int y)
{
    x = abs(x);                                // valor absoluto
    y = abs(y);
    do {
        int n = x % y;
        x = y;
        y = n;
    } while (y > 0);
    return x;
}
```

Implemente esta función en lenguaje ensamblador y escriba un programa de prueba para llamar a la función varias veces, y pasarle distintos valores. Muestre todos los resultados en la pantalla.

7. Programa de números primos

Escriba un procedimiento llamado **EsPrimo**, que active bandera Cero si el entero de 32 bits que se pasa en el registro EAX es primo. Optimice el ciclo del programa para que se ejecute con la mayor eficiencia posible. Escriba un programa de prueba que pida al usuario un entero, llame a **EsPrimo** y muestre un mensaje indicando si el valor es primo o no. Continúe pidiendo al usuario enteros y llamando a **EsPrimo** hasta que el usuario introduzca el valor -1.

8. Conversión de decimales empaquetados

Escriba un procedimiento llamado **EmpaquetadoAAsc**, que convierta un entero decimal empaquetado de 4 bytes a una cadena de dígitos decimales ASCII. Pase al procedimiento el entero empaquetado y la dirección de un búfer que guarde los dígitos ASCII. Escriba un programa corto de prueba, para mostrar varios enteros convertidos.

9. Procedimiento SumaAsc

Convierta el código para la suma con varios dígitos ASCII que presentamos en la sección 7.6.1, a un procedimiento llamado **SumaAsc** con los siguientes parámetros: ESI apunta al primer número, EDI apunta al segundo número, EDX apunta a la suma, y ECX contiene el número de dígitos en los operandos. Escriba un programa que llame a SumaAsc y llame a WriteString para mostrar que la suma funcionó correctamente.

10. Mostrar decimal ASCII

Escriba un procedimiento llamado EscribirEscalado, que imprima en pantalla un número ASCII decimal con un punto decimal implícito. Suponga que el siguiente número se define como se muestra a continuación, en donde DESP_DECIMAL indica que el punto decimal debe insertarse a cinco posiciones a partir del lado derecho del número:

```
DESP_DECIMAL = 5  
.data  
uno_decimal BYTE "100123456789765"
```

EscribirEscalado mostraría el número así:

1001234567.89765

Al llamar a EscribirEscalado, pase el desplazamiento del número en EDX, la longitud del número en ECX, y el desplazamiento decimal en EBX. Escriba un programa de prueba para mostrar tres números de distintos tamaños.

11. Suma de enteros empaquetados

Utilice el código de la sección 7.7.1 para escribir un procedimiento llamado SumaEmpaquetado, que sume dos enteros decimales empaquetados de un tamaño arbitrario (ambos deben ser iguales). Escriba un programa de prueba que pase a SumaEmpaquetado varios pares de enteros: de 4, 8 y 16 bytes. Muestre las sumas en hexadecimal. Use la siguiente lista de parámetros:

```
SumaEmpaquetado PROC  
    pNum1:PTR BYTE,          ; apuntador al primer número  
    pNum2:PTR BYTE,          ; apuntador al segundo número  
    pSuma:PTR BYTE,          ; apuntador a la suma  
    tamNum:DWORD             ; número de bytes a sumar
```

8

PROCEDIMIENTOS AVANZADOS

8.1	Introducción	8.5.4	Directiva PROTO
8.2	Marcos de pila	8.5.5	Clasificaciones de parámetros
8.2.1	Parámetros de pila	8.5.6	Ejemplo: intercambio de dos enteros
8.2.2	Variables locales	8.5.7	Tips de depuración
8.2.3	Instrucciones ENTER y LEAVE	8.5.8	Repaso de sección
8.2.4	Directiva LOCAL	8.6	Creación de programas con varios módulos
8.2.5	Procedimiento WriteStackFrame	8.6.1	Ocultar y exportar nombres de procedimientos
8.2.6	Repasso de sección	8.6.2	Llamadas a procedimientos externos
8.3	Recursividad	8.6.3	Uso de variables y símbolos a través de los límites de los módulos
8.3.1	Cálculo recursivo de una suma	8.6.4	Ejemplo: programa SumaArreglo
8.3.2	Cálculo de un factorial	8.6.5	Creación de módulos mediante el uso de Extern
8.3.3	Repasso de sección	8.6.6	Creación de módulos mediante el uso de INVOKES y PROTO
8.4	Directiva .MODEL	8.6.7	Repasso de sección
8.4.1	Especificadores de lenguaje	8.7	Resumen del capítulo
8.4.2	Repasso de sección	8.8	Ejercicios de programación
8.5	INVOKES, ADDR, PROC y PROTO (opcional)		
8.5.1	Directiva INVOKES		
8.5.2	Operador ADDR		
8.5.3	Directiva PROC		

8.1 Introducción

En este capítulo hablaremos sobre la estructura fundamental de las subrutinas y las llamadas a éstas. Debido a que existe una tendencia natural de buscar conceptos universales que faciliten el aprendizaje, utilizaremos este capítulo para mostrar de qué manera funcionan todos los procedimientos, usando el lenguaje ensamblador como herramienta de programación de bajo nivel. En otras palabras, lo que aprenderá aquí se ve a menudo en cursos de programación de nivel intermedio en C++ y Java, y en un curso básico de ciencias computacionales llamado *lenguajes de programación*. Los siguientes temas, que veremos en este capítulo, son conceptos básicos de los lenguajes de programación:

- Marcos de pila.
- Alcance y tiempo de vida de las variables.
- Tipos de parámetros de pila.
- Paso de argumentos por valor y por referencia.

- Creación e inicialización de variables locales en la pila.
- Recursividad.
- Escritura de programas con varios módulos.
- Modelos de memoria y especificadores de lenguaje.

Los siguientes temas opcionales demuestran las directivas de alto nivel que se incluyen en MASM, diseñadas para asistir a los programadores de aplicaciones:

- Las directivas INVOKE, PROC y PROTO.
- Los operadores USES y ADDR.

Sobre todo, gracias a su conocimiento del lenguaje ensamblador podrá penetrar en la mente del escritor del compilador, a medida que produzca el código de bajo nivel que hace que un programa se ejecute.

Observaciones sobre la terminología Los lenguajes de programación utilizan distintos términos para referirse a las subrutinas. Por ejemplo, en C y C++ a las subrutinas se les llama *funciones*; en Java se les llama *métodos*; y en MASM, se les llama *procedimientos*. Nuestro objetivo en este capítulo es mostrar las implementaciones de bajo nivel de las llamadas comunes a las subrutinas, como podrían aparecer en C y C++. Al inicio de este capítulo, al referirnos a los principios generales, utilizaremos el término general *subrutina*. Más adelante, cuando nos concentremos en las directivas específicas de MASM (como PROC y PROTO), utilizaremos el término específico *procedimiento*.

8.2 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.

La estructura de un marco de pila se ve afectada directamente por el modelo de memoria de un programa y su elección de la convención de paso de argumentos.

Existe un buen motivo para aprender acerca del paso de argumentos a la pila: casi todos los lenguajes de alto nivel los utilizan. Por ejemplo, si desea llamar a las funciones en la Interfaz de programación de aplicaciones (API) de MS Windows, debe pasar argumentos a la pila.

8.2.1 Parámetros de pila

Existen dos tipos básicos de parámetros de subrutinas: los *parámetros de registro* y los *parámetros de pila*. Las bibliotecas Irvine32 e Irvine16 utilizan parámetros de registro. En esta sección le mostraremos 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 del 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 registros antes de que se puedan cargar con los valores de los argumentos. Tal es el caso de cuando se hace una llamada a **DumpMem**, por ejemplo:

```
pushad  
mov  esi,OFFSET arreglo          ; desplazamiento (OFFSET) inicial  
mov  ecx,LENGTHOF arreglo       ; tamaño, en unidades
```

```

mov ebx,TYPE arreglo          ; formato de doble palabra
call DumpMem                  ; muestra la memoria
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. Suponga 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 );
```

Observe 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 a 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 lenguajes 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 llamada **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 formas estándar de inicializar y acceder a los parámetros durante las llamadas a funciones. Empiezan con un *prüfago* 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 el registro EBP y la instrucción RET regresa de la función, y borra los parámetros de la pila.

Ejemplo: SumarDos La siguiente función **SumarDos**, escrita en C, recibe dos enteros que se pasan por valor y devuelve su suma:

```
int SumarDos( int x, int y )
{
    return x + y;
}
```

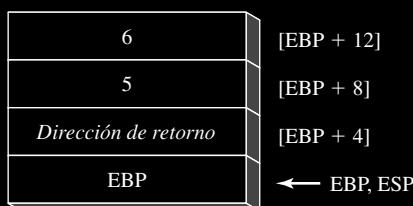
Vamos a crear una implementación equivalente en lenguaje ensamblador. En su prólogo, **SumarDos** mete a EBP en la pila para preservar su valor existente:

```
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 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 cambiaría 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
mov   eax,[ebp + 12]      ; base del marco de pila
mov   eax,[ebp + 8]        ; 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 regrese una subrutina. De no ser así se produce una fuga de memoria, y la pila se vuelve corrupta. Por ejemplo, suponga 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 a 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          ; la pila está bien
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 convención 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 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 indica 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 limpiar la pila, por lo que la responsabilidad se deja al proceso que hace la llamada.

La biblioteca Irvine32 utiliza la convención de llamadas de STDCALL para poder ser compatible con la biblioteca de la API de MS Windows. La biblioteca Irvine16 utiliza la misma convención, para ser consistente con la biblioteca Irvine32.

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 referiremos 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 **Mayuscula** recibe un argumento tipo carácter y devuelve su equivalente en mayúscula en AL:

```

Mayuscula PROC
    push  ebp
    mov   ebp,esp
    mov   al,[esp+8]          ; AL = carácter
    cmp   al,'a'              ; ¿es menor que 'a'?

```

```

jb    L1           ; sí: no hace nada
cmp   al,'z'       ; ¿es mayor que 'z'?
ja    L1           ; sí: 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 Mayuscula, 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 el carácter hacia EAX:

```

movzx eax, valCar      ; mueve con extensión
push  eax
call Mayuscula

```

Ejemplo de argumento de 16 bits Suponga 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 produciría 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 tamaño de los parámetros es importante.

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 figura 8-1 muestra una imagen del marco de pila, después de meter EBP dentro de `EscribirHex64`.

FIGURA 8-1 Marco de pila después de meter EBP.



Cómo 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, suponga que el siguiente procedimiento `MiSub` tiene un parámetro de pila. Mete a ECX y a EDX después de asignar a EBP la base del marco de pila, y carga el parámetro de pila en EAX:

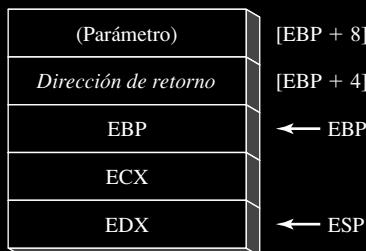
```

MiSub PROC
    push  ebp                ; guarda apuntador base
    mov   ebp,esp             ; base del marco de pila
    push  ecx
    push  edx                ; guarda EDX
    mov   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 (vea la figura 8-2).

FIGURA 8–2 Marco de pila para el procedimiento MiSub.



Cómo 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 operador 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
```

Suponga que combinamos a USES con un parámetro de pila, como en el siguiente procedimiento **MiSub2**. Se espera que su parámetro se encuentre en la pila, en EBP+8:

```
MiSub2 PROC USES ecx edx
    push  ebp          ; guarda apuntador base
    mov   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
```

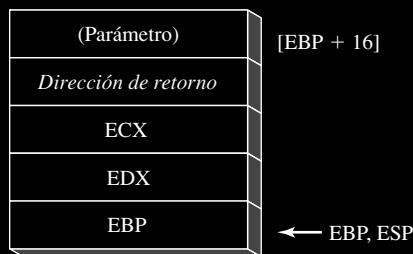
He aquí el código correspondiente, generado por MASM para **MiSub2**:

```
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 insertó las instrucciones PUSH para ECX y EDX al principio del procedimiento, alterando el desplazamiento del parámetro de pila. La figura 8-3 muestra cómo debe hacerse referencia ahora al parámetro de pila como [EBP + 16]. USES modifica la pila antes de guardar EBP, lo cual va en contra del código de prólogo estándar para las subrutinas. Como veremos en la sección 8.5.3, la directiva PROC tiene una sintaxis de alto nivel para declarar parámetros de pila. En ese contexto, el operador USES no causa problemas.

Los procedimientos que utilizan parámetros de pila explícitos deben evitar el operador USES.

FIGURA 8–3 Marco de pila del procedimiento MiSub2.



8.2.2 Variables locales

En los programas de lenguajes 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 con 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 utilizan las variables locales se libera cuando termina la subrutina.
- Una variable local puede tener el 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 subprocessos 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 predeterminados 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 un 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 pila. Utiliza la convención de llamadas de C:

```

MiSub PROC
    push    ebp

```

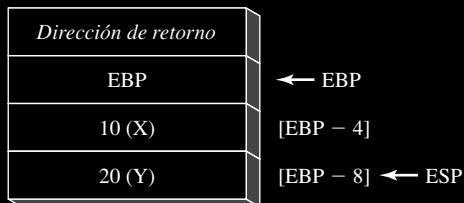
```

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 figura 8-4 muestra el marco de pila de la función, después de inicializar las variables locales.

FIGURA 8-4 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                ; ¡regresa a una dirección inválida!
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 del 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, suponga 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 mostrar a continuación, llena un arreglo con una secuencia pseudoaleatoria 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 WORD cuenta DUP(?)  

.code
push OFFSET arreglo
push CUENTA
call LlenarArreglo
```

Dentro de **LlenarArreglo**, el siguiente código de prólogo inicializa el apuntador al marco 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 guardó:



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 ; limpia la pila
    ret 8
LlenarArreglo ENDP
```

Instrucción LEA

La instrucción LEA devuelve el desplazamiento de un operando indirecto. Como los operandos indirectos contienen uno o más registros, sus desplazamientos se calculan en tiempo de ejecución. Para mostrar cómo

puede utilizarse LEA, veamos el siguiente programa de C++, que declara un arreglo local de caracteres y hace referencia a **miCadena** al asignar los 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 decremente por 32 para mantenerlo alineado en un límite de doble palabra. Observe 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 del 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 pila ya que OFFSET sólo funciona con las direcciones que se conocen en tiempo de compilación. La siguiente instrucción no se ensamblaría:

```
mov esi,OFFSET [ebp-30]           ; error
```

8.2.3 Instrucciones ENTER y LEAVE

La instrucción ENTER crea de manera automática un marco de pila para un procedimiento al que se llamó. Reserva espacio en la pila para las 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 a ESP en un límite de doble palabra. *Nivelanidamiento* determina el número de apuntadores al marco de pila que se copian en el marco de pila actual, provenientes del marco de pila del procedimiento al que se llamó. En nuestros programas, *nivelanidamiento* siempre es cero. Los manuales de los procesadores IA-32 de Intel explican cómo la instrucción ENTER soporta los niveles de anidamiento en lenguajes estructurados por bloques.¹

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 para las variables locales:

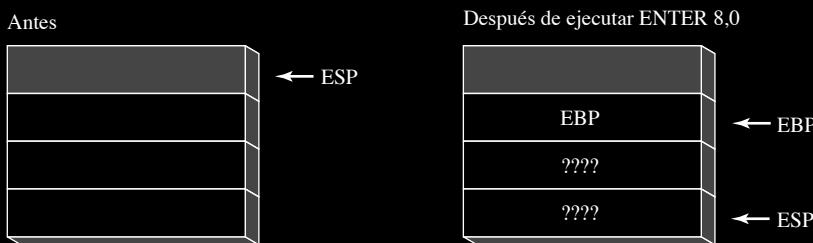
```
MiSub PROC
    enter 8,0
```

Esto es equivalente a las siguientes instrucciones:

```
MiSub PROC
    push ebp
    mov  ebp,esp
    sub  sp,8
```

La figura 8-5 muestra la pila antes y después de ejecutar ENTER.

FIGURA 8-5 Marco de pila, antes y después de ejecutar ENTER.



Si utiliza la instrucción ENTER, es imprescindible que 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 una 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 de espacio para las variables locales:

```
MiSub PROC
    push ebp
    mov  ebp,esp
    sub  esp,8
    .
    .
    .
    mov  esp,ebp
    pop  ebp
    ret
MiSub ENDP
```

8.2.4 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. En el capítulo 10 describiremos las estructuras y otros tipos definidos 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:

```
Mezcla PROC  
    LOCAL pArreglo:PTR WORD
```

La variable local **ArregloTemp** es un arreglo de 10 dobles palabras. Observe el uso de los corchetes para mostrar el tamaño del arreglo:

```
LOCAL ArregloTemp[10]:DWORD
```

Generación de código de MASM

Es conveniente analizar el código generado por MASM cuando se utiliza la directiva LOCAL; para esto se puede ver un desensamblado. El siguiente procedimiento **Ejemplo1** tiene una sola variable local 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

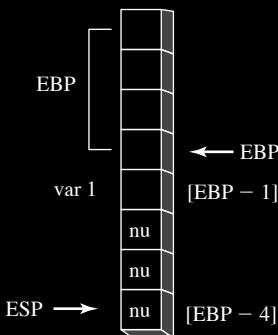
La directiva LOCAL tiene un comportamiento interesante cuando se declaran variables locales de distintos tamaños. A cada una se le asigna espacio de acuerdo a su tamaño: una variable de 8 bits se asigna al siguiente byte disponible, una variable de 16 bits se asigna a la siguiente dirección par (alineada 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 figura 8-6, 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 las letras *nu*).

FIGURA 8-6 Creación de espacio para variables locales (Procedimiento Ejemplo1).



El procedimiento **Ejemplo2** contiene una doble palabra, seguida de un byte:

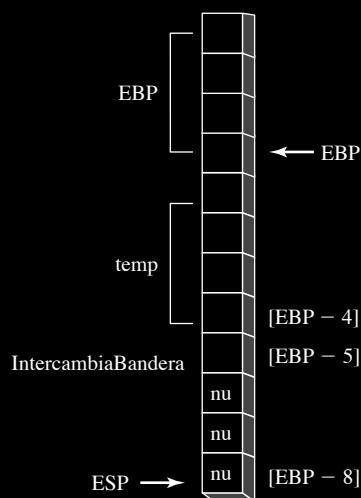
```
Ejemplo2 PROC
    LOCAL temp:DWORD, IntercambiaBandera:BYTE
    .
    .
    Ret
Ejemplo2 ENDP
```

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 b1,[ebp-5]                ; IntercambiaBandera
leave
ret
```

Aunque **IntercambiaBandera** es sólo un byte, ESP se redondea hacia abajo, a la siguiente ubicación tipo doble palabra de la pila. En la figura 8-7 se muestra una vista detallada de la pila, en forma de bytes individuales, para indicar la ubicación exacta de IntercambiaBandera y el espacio sin usar debajo de ésta (etiquetado como *nu*).

FIGURA 8–7 Creación de espacio en el ejemplo 2 para variables locales.



Reservación de espacio extra en la pila Si planea crear arreglos más grandes que unos cuantos cientos de bytes como variables locales, asegúrese de reservar el espacio adecuado para la pila en tiempo de ejecución, usando la directiva STACK. Por ejemplo, en el archivo de la biblioteca *Irvine32.inc* reservamos 4096 de espacio en la pila:

```
.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. Por ejemplo, suponga que **Sub1** llama a **Sub2** y que **Sub2** llama a **Sub3**. Cada uno de estos procedimientos podría tener una variable local tipo arreglo:

```
Sub1 PROC
    LOCAL arreglo1[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 requerirá 660 bytes utilizados por las variables locales, más las dos direcciones de retorno de los procedimientos (8 bytes), y cualquier registro que pudiera haberse metido a la pila dentro de los procedimientos.

8.2.5 Procedimiento WriteStackFrame

La biblioteca de enlace del libro tiene un procedimiento útil llamado **WriteStackFrame**, el cual muestra el contenido del marco de pila del procedimiento actual. Muestra los parámetros de pila del procedimiento, la dirección de retorno, las variables locales y los registros almacenados. El Profesor James Brink de la Pacific Lutheran University proporcionó generosamente este procedimiento. He aquí el prototipo:

```
WriteStackFrame PROTO,
    numParam:DWORD      ; número de parámetros que se pasan
    numLocalVar: DWORD, ; número de variables DWordLocal
    numSavedReg: DWORD  ; número de registros almacenados
```

He aquí un extracto de un programa que demuestra el uso de WriteStackFrame:

```
main PROC
    mov    eax,0EAEAEAEAh
    mov    ebx,0EBEBEBEBh
    INVOKE unProc, 1111h, 2222h      ; pasa dos argumentos enteros
    exit
main ENDP

unProc PROC USES eax ebx,
    x: DWORD, y: DWORD
    LOCAL a:DWORD, b:DWORD
    PARAMS = 2
    LOCALS = 2
    SAVED_REGS = 2
    mov    a,0AAAAAh
    mov    b,0BBBBBh
    INVOKE WriteStackFrame, PARAMS, LOCALS, SAVED_REGS
```

La llamada produjo los siguientes resultados de ejemplo:

Stack Frame
00002222 ebp+12 (parameter) 00001111 ebp+8 (parameter) 00401083 ebp+4 (return address) 0012FFF0 ebp+0 (saved ebp) <--- ebp 0000AAAA ebp-4 (local variable) 0000BBBB ebp-8 (local variable) EAEAEAEA ebp-12 (saved register) EBEBEBEB ebp-16 (saved register) <--- esp

Un segundo procedimiento llamado **WriteStackFrameName** tiene un parámetro adicional, que almacena el nombre del procedimiento al que corresponde el marco de pila:

```
WriteStackFrameName PROTO,
    numParam:DWORD,           ; número de parámetros que se pasan
    numLocalVal: DWORD,       ; número de variables DWordLocal
    numSavedReg: DWORD,       ; número de registros almacenados
    procName: PTR BYTE
```

Consulte el programa de ejemplo llamado *Prueba_WriteStackFrame.asm*, para ejemplos y documentación relacionados con este procedimiento. Además, el código fuente (en *\Lib32\Irvine32.asm*) contiene documentación detallada.

8.2.6 Repaso de sección

1. (*Verdadero/Falso*): el marco de pila de una subrutina siempre contiene la dirección de retorno del procedimiento que hace la llamada y las variables locales de la subrutina.
2. (*Verdadero/Falso*): los arreglos se pasan por referencia, para evitar copiarlos a la pila.
3. (*Verdadero/Falso*): el código del prólogo de un procedimiento siempre mete a EBP en la pila.
4. (*Verdadero/Falso*): las variables locales se crean sumando un entero al apuntador de la pila.
5. (*Verdadero/Falso*): en modo protegido de 32 bits, el último argumento a meter en la pila, en una llamada a un procedimiento, se almacena en la ubicación ebp+8.
6. (*Verdadero/Falso*): el paso por referencia requiere sacar el desplazamiento de un parámetro de la pila, dentro del procedimiento al que se llamó.
7. ¿Cuáles son los dos tipos comunes de parámetros de pila?
8. ¿Qué instrucciones pertenecen al epílogo de un procedimiento, cuando éste tiene parámetros de pila y variables locales?

9. Cuando una función de C devuelve un entero de 32 bits, ¿en dónde se almacena el valor de retorno?
10. ¿Cómo limpia la pila un programa que utiliza la convención de llamadas de STDCALL, después de la llamada a un procedimiento?
11. He aquí una secuencia de llamada para un procedimiento llamado **SumarTres**, que suma tres dobles palabras (asuma el uso de STDCALL):

```
push 10h
push 20h
push 30h
call SumarTres
```

Dibuje una imagen del marco de pila del procedimiento, justo después de haber metido a ESP en la pila.

12. ¿De qué manera la instrucción LEA es más poderosa que el operador OFFSET?
13. En el ejemplo de C++ que se muestra en la sección 8.2.2, ¿cuánto espacio de la pila utiliza una variable de tipo *int*?
14. Escriba instrucciones en el procedimiento **SumarTres** (de la pregunta anterior) que calculen la suma de los tres parámetros de pila.
15. ¿Cómo se pasa un argumento tipo carácter de 8 bits a un procedimiento que espera un parámetro entero de 32 bits?
16. Declare una variable local llamada **pArreglo**, que sea un apuntador a un arreglo de dobles palabras.
17. Declare una variable local llamada **bufer**, que sea un arreglo de 20 bytes.
18. Declare una variable local llamada **pwArreglo**, que apunte a un entero sin signo de 16 bits.
19. Declare una variable local llamada **miByte**, que almacene un entero de 8 bits con signo.
20. Declare una variable local llamada **miArreglo**, que sea un arreglo de 20 dobles palabras.
21. *Discusión:* ¿qué ventajas podría tener la convención de llamadas de C, en comparación con la convención de llamadas de STDCALL?

8.3 Recursividad

Una subrutina *recursiva* es una que se llama a sí 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 **sinfín**, 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 SinFin
    exit
main ENDP

SinFin PROC
    mov EDX,offset cadSinFin
    call WriteString
    call SinFin
    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.

Si tiene acceso a una herramienta de monitoreo del rendimiento, como el Administrador de tareas de Windows, ábrala y haga clic en el cuadro de diálogo Rendimiento. Después ejecute el programa *SinFin.exe*, que se encuentra en el directorio de este capítulo. La memoria se llenará lentamente y el programa consumirá el 100% de los recursos de la CPU. Después de unos cuantos minutos, la pila del programa se desbordará y se producirá una excepción del procesador (el programa se detendrá).

8.3.1 Cálculo 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 linea
    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, lo 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 tabla 8-1 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.

8.3.2 Cálculo de un factorial

A menudo, las llamadas recursivas almacenan los datos temporales en parámetros de pila. Cuando las llamadas recursivas se limpian, los datos guardados en la pila pueden ser útiles. El siguiente ejemplo que vamos

Tabla 8-1 Marco de pila y registros (CalcSuma).

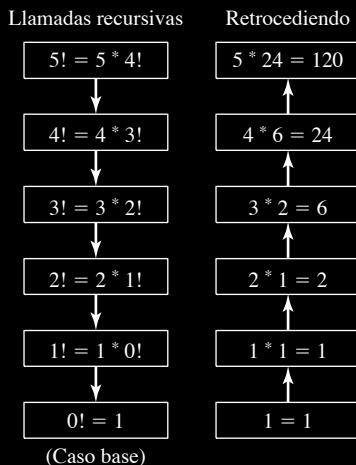
Se metió en la pila	Valor en ECX	Valor en EAX
L1	5	0
L2	4	5
L2	3	9
L2	2	12
L2	1	14
L2	0	15

a analizar calcula el 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 el 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, para calcular $5!$, el algoritmo recursivo desciende a lo largo de la columna izquierda de la figura 8-8 y retrocede a lo largo de la columna derecha.

FIGURA 8-8 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 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 de 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**. En la figura 8-9, la pila se muestra después de varias llamadas recursivas. Podemos ver que se meten nuevos valores para *n* y EBP en la pila, cada vez que Factorial se llama a sí mismo.

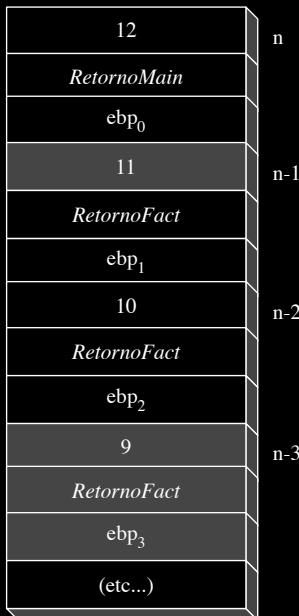
Cada llamada al procedimiento en nuestro ejemplo utiliza 12 bytes de almacenamiento en la pila. Justo antes de que **Factorial** se llame a sí mismo, *n* – 1 se mete en la pila como argumento de entrada. El procedimiento devuelve su propio valor de factorial en EAX, el cual se multiplica después por el valor que se metió en la pila antes de la llamada.

8.3.3 Repaso de sección

- (Verdadero/Falso): dada la misma tarea a realizar, por lo general, una subrutina recursiva utiliza menos memoria que una no recursiva.
- En la función Factorial, ¿qué condición termina la recursividad?
- ¿Qué instrucciones en el procedimiento Factorial en lenguaje ensamblador se ejecutan después de que termina cada llamada recursiva?
- ¿Qué ocurrirá a la salida del programa Factorial si tratamos de calcular 13!?

5. *Reto:* en el programa Factorial, ¿cuántos bytes de espacio de pila utiliza el procedimiento Factorial al calcular 12!?
6. *Reto:* escriba el seudocódigo para un algoritmo recursivo que genere los primeros 20 enteros de la serie de Fibonacci (1, 1, 2, 3, 5, 8, 13, 21, . . .).

FIGURA 8–9 Marco de pila parcial, programa Factorial.



8.4 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 los procedimientos, y la convención para el paso de parámetros. Los últimos dos son en especial importantes cuando los programas escritos en otros lenguajes 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 tabla 8-2. Todos los modos, con la excepción del plano, se utilizan cuando se programa en modo de direccionamiento real de 16 bits.

Todos los programas en modo de direccionamiento real que hemos mostrado hasta ahora en este libro 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 de 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 4GB. Por ejemplo, el archivo *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 los símbolos públicos. La *distancia de pila* puede ser NEARSTACK (el valor predeterminado) o FARSTACK.²

Tabla 8-2 Modelos de memoria.

Modelo	Descripción
Tiny (diminuto)	Un solo segmento, contiene código y datos. Este modelo lo utilizan los programas que tienen la extensión .com en sus nombres 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

8.4.1 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 libro, 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 de este capítulo. Es el especificador de lenguaje que se utiliza al llamar a las funciones de MS Windows. En el capítulo 12 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 subrutina se metan en la pila en orden inverso (del último al primero). Suponga que escribimos la siguiente llamada a una función en un 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 le 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
    add  eax,[ebp + 8]            ; segundo parámetro
    pop  ebp
    ret  8                      ; limpia la pila
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, suponga que el procedimiento **SumarDos** tiene dos parámetros 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 de **_MiSub@8**. Para ver todos los nombres de los procedimientos dentro de un archivo OBJ, use la herramienta DUMPBIN que se proporciona en Visual Studio, con la opción /SYMBOLS.

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 a ESP, con lo cual se restablece al valor que tenía antes de meter los argumentos:

```
push 6          ; segundo argumento  
push 5          ; primer argumento  
call SumarDos  
add esp,8       ; limpia la pila
```

El especificador de lenguaje C adjunta un carácter de guión bajo a la izquierda de los nombres de los procedimientos externos. Por ejemplo:

_SumarDos

8.4.2 Repaso de sección

1. Describa el modelo pequeño de memoria.
2. Describa el modelo plano de memoria.
3. ¿Qué diferencia hay entre la opción de lenguaje C (de la directiva .MODEL) y la de STDCALL, en relación con la eliminación de argumentos de la pila?

8.5 INVOKE, ADDR, PROC y PROTO (opcional)

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. A los estudiantes que están aprendiendo los fundamentos de las computadoras les sienta mejor una comprensión detallada de la mecánica de bajo nivel involucrada en las llamadas a las subrutinas.

Hay una situación en la que el uso de directivas de procedimientos avanzadas 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.

8.5.1 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 reemplazo 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,	; muestra un bloque de memoria
OFFSET arreglo,	; apunta al arreglo
LENGTHOF arreglo,	; la longitud del arreglo
TYPE arreglo	; tamaño de los componentes del arreglo

Los tipos de argumentos se listan en la tabla 8-3.

Tabla 8-3 Tipos de argumentos utilizados con INVOKE.

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 <i>nombre</i>	ADDR miLista
OFFSET <i>nombre</i>	OFFSET miLista

Sobrescritura de EAX y EDX Si pasa argumentos menores de 32 bits a un procedimiento, INVOKE ocasionalmente con frecuencia que el ensamblador sobrescriba 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.

8.5.2 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. La directiva .model en *Irvine32.inc* especifica el modelo plano de memoria.

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+4
push OFFSET Arreglo
call Intercambiar
```

8.5.3 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 en el capítulo 3. *Atributos* se refiere a cualquiera de los siguientes:

[*distancia*] [*tipoleng*] [*visibilidad*] [*prólogo*]

La tabla 8-4 describe cada uno de los atributos.

Tabla 8-4 El campo Atributos en la directiva PROC.

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 llamadas (convención de paso de parámetros) tal como C, PASCAL o STDCALL. Ignora el lenguaje especificado en la directiva .MODEL
Visibilidad	Indica la visibilidad del procedimiento para con los otros módulos. Las opciones 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 del prólogo y del epílogo. Consulte la sección titulada "Código de prólogo y epílogo definido por el usuario" en la Guía para el programador de MASM, capítulo 7

Listas 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
```

Observe las comas requeridas antes del primer parámetro, que 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	PTR SDWORD
PTR QWORD	PTR TBYTE

Aunque es posible agregar atributos NEAR y FAR a estas expresiones, sólo son relevantes en aplicaciones más especializadas. También pueden crearse tipos calificados mediante el uso de las directivas TYPEDEF y STRUCT, que explicaremos en el capítulo 10.

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
```

Tip: los detalles completos sobre el código de los procedimientos que genera MASM no aparecen en los archivos de listado (extensión .LST). Para ver estos detalles, abra su programa con un depurador y vea la ventana Desensamblado.

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 llamada **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 cómo se reserva el espacio en la pila para la variable local (manejadorArchivo) antes de meter EAX y EBX (especificados 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
    pop   ebx
    pop   eax
    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 predeterminada cuando se utiliza la instrucción INCLUDE Irvine32.inc, y es la convención de llamadas que se utiliza 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 predeterminada 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 Ejemplo1 usando **Invoke**, el ensamblador genera código consistente con la convención de llamadas de C. De manera similar, si declaramos Ejemplo1 usando **STDCALL**, **Invoke** genera código consistente con esa convención de lenguaje:

Ejemplo1 PROC STDCALL,
 parm1:DWORD, parm2:DWORD

8.5.4 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 los 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 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 de procedimiento  
. . .  
MiSub ENDP  
TINVOKE MiSub ; llamada a procedimiento
```

Suponiendo que ya ha escrito un procedimiento específico, puede crear su prototipo con facilidad copiando la instrucción PROC y haciendo los siguientes cambios:

- Cambie la palabra PROC a PROTO.
 - Elimine el operador USES si lo hay, junto con su lista de registros.

Por ejemplo, suponga que ya hemos creado el procedimiento **SumaArreglo**:

```
SumaArreglo PROC USES esi ecx,  
    ptrArreglo:PTR DWORD,           ; apunta al arreglo  
    tamArreglo:DWORD               ; tamaño del arreglo  
    ; (se omitieron las líneas restantes)  
SumaArreglo ENDP
```

Ésta es la declaración PROTO correspondiente:

```
SumaArreglo 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. 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, suponga 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     WORD 3000h
dpa1ab_1    DWORD 12345678h
```

La siguiente llamada a Sub1 es válida:

```
INVOKE Sub1, 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:[00404001h] ; 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 al 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,    ; error: demasiados argumentos
      palab_2, ADDR byte_1, palab_2
```

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, byte1, 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 404000h          ; 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** del capítulo 5, 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:

```
SumaArreglo 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 ; sí: 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 a **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
```

8.5.5 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 llamó:

- **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 búfer de cadena, en el que **ReadConsole** almacena el texto escrito por el usuario:

```
.data
bufer BYTE 80 DUP(?)
manejadorEntrada DWORD ?
.code
```

```
    INVOKE ReadConsole, manejadorEntrada, ADDR bufer,
          (etc.)
```

- **Entrada-Salida:** un parámetro de entrada-salida es idéntico a un parámetro de salida, con una excepción: el procedimiento que se llamó espera que la variable a la que hace referencia el parámetro contenga datos. También se espera que el procedimiento modifique la variable a través del apuntador.

8.5.6 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 con parámetros, el ensamblador cambia la instrucción RET al final de Intercambiar a **RET 8** (suponiendo que STDCALL es la convención de llamadas).

8.5.7 Tips de depuración

En esta sección destacaremos algunos errores comunes que encontramos al pasar argumentos a los procedimientos en lenguaje ensamblador. Esperamos que usted nunca los cometiera.

Conflictos de tamaño de los argumentos

Las direcciones de los arreglos se basan en los tamaños de sus elementos. Por ejemplo, para acceder al segundo elemento de un arreglo de dobles palabras, hay que sumar 4 a la dirección inicial del arreglo. Suponga que llamamos al procedimiento **Intercambiar** de la sección 8.5.6 y le pasamos apuntadores a los primeros dos elementos de **ArregloDoble**. Si calculamos en forma errónea la dirección del segundo elemento como **ArregloDoble + 1**, los valores hexadecimales resultantes en **ArregloDoble** después de llamar a **Intercambiar** son incorrectos:

```
.data
ArregloDoble DWORD 10000h,20000h
.code
Invoke Intercambiar, ADDR [ArregloDoble + 0], ADDR [ArregloDoble + 1]
```

Paso del tipo incorrecto de apuntador

Al utilizar INVOKE, recuerde que el ensamblador no valida el tipo de apuntador que le pasamos a un procedimiento. Por ejemplo, el procedimiento **Intercambiar** de la sección 8.5.6 espera recibir dos apuntadores a dobles palabras. Suponga que sin querer le pasamos apuntadores a bytes:

```
.data
ArregloByte BYTE 10h,20h,30h,40h,50h,60h,70h,80h
.code
Invoke Intercambiar, ADDR [ArregloByte + 0], ADDR [ArregloByte + 1]
```

El programa se ensamblará y se ejecutará, pero cuando se desreferencien ESI y EDI, se intercambiarán valores de 32 bits.

Paso de valores inmediatos

Si un procedimiento tiene un parámetro por referencia, no se le debe pasar un argumento inmediato. Considere el siguiente procedimiento, que tiene un solo parámetro por referencia:

```
Sub2 PROC apuntDatos:PTR WORD
    mov esi,apuntDatos           ; obtiene la dirección
    mov [esi],0                   ; desreferencia, asigna cero
    ret
Sub2 ENDP
```

La siguiente instrucción INVOKE se ensambla, pero produce un error en tiempo de ejecución. El procedimiento **Sub2** recibe 1000h como valor para el apuntador y desreferencia la ubicación de memoria 1000h:

```
Invoke Sub2, 1000h
```

Es probable que este ejemplo provoque un fallo de protección general, ya que la ubicación de memoria 1000h no está dentro del segmento de datos del programa.

8.5.8 Repaso de sección

1. (*Verdadero/Falso*): la instrucción CALL no puede incluir argumentos de procedimientos.
2. (*Verdadero/Falso*): la directiva INVOKE puede incluir hasta un máximo de tres argumentos.
3. (*Verdadero/Falso*): la directiva INVOKE sólo puede pasar operandos de memoria, pero no valores de registros.
4. (*Verdadero/Falso*): la directiva PROC puede contener un operador USES, pero la directiva PROTO no.
5. (*Verdadero/Falso*): al utilizar la directiva PROC, todos los parámetros deben listarse en la misma línea.
6. (*Verdadero/Falso*): si pasamos una variable que contenga el desplazamiento de un arreglo de bytes a un procedimiento que espera un apuntador a un arreglo de palabras, el ensamblador no atrapará el error.
7. (*Verdadero/Falso*): si pasamos un valor inmediato a un procedimiento que espera un parámetro por referencia, se puede generar un fallo de protección general (en modo protegido).
8. Declare un procedimiento llamado **ArregloMult** que reciba dos apuntadores a arreglos de dobles palabras y un tercer parámetro que indique el número de elementos del arreglo.
9. Cree una directiva PROTO para el procedimiento del ejercicio anterior.

10. ¿El procedimiento **Intercambiar** de la sección 8.5.6 utilizaba parámetros de entrada, parámetros de salida, o parámetros de entrada-salida?
11. En el procedimiento **ReadConsole** de la sección 8.5.5, ¿es **bufer** un parámetro de entrada o de salida?

8.6 Creación de programas con varios módulos

Los archivos de código fuente extensos 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 formas requeriría que se volvieran 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 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 fuente.

Hay dos métodos generales para crear programas con 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 las directivas avanzadas INVOKE y PROTO de Microsoft, que simplifican las llamadas a procedimientos y ocultan ciertos detalles de bajo nivel. Le presentaremos ambos métodos y dejaremos que decida cuál desea utilizar.

8.6.1 Ocultar y exportar nombres de procedimientos

De manera predeterminada, MASM hace a 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 del *encapsulamiento* para ocultar los procedimientos dentro de módulos y evitar conflictos de nombre 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 utiliza OPTION PROC:PRIVATE en el módulo de arranque de su programa, asegúrese de designar su procedimiento de arranque (por lo general, main) como PUBLIC, o el cargador del sistema operativo no podrá encontrarlo. Por ejemplo,

```
main PROC PUBLIC
```

8.6.2 Llamadas a procedimientos externos

La directiva EXTERN, que se utiliza al llamar a un procedimiento que está fuera del módulo actual, 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 de un procedimiento identifica el espacio total de la pila utilizado por los parámetros declarados (vea la directiva PROC extendida en la sección 8.5). Si utiliza la directiva PROC básica sin parámetros declarados, el sufijo en cada nombre de procedimiento en EXTERN será **@0**. Si declara un procedimiento usando la directiva PROC, agregue 4 bytes por cada parámetro. Suponga que declaramos a **SumarDos** con dos parámetros tipo doble palabra:

```
SumarDos PROC,
    va11:DWORD,
    va12:DWORD
    .
    .
SumarDos ENDP
```

La correspondiente directiva EXTERN es **EXTERN SumarDos@8:PROC**. Si planea llamar a SumarDos mediante INVOKE (sección 8.5), use la directiva PROTO en vez de EXTERN:

```
SumarDos PROTO,
    va11:DWORD,
    va12:DWORD
```

8.6.3 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 las 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 una 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, SIM1:ABS
```

Ahora vamos a crear un archivo de código 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 a **SIM1**:

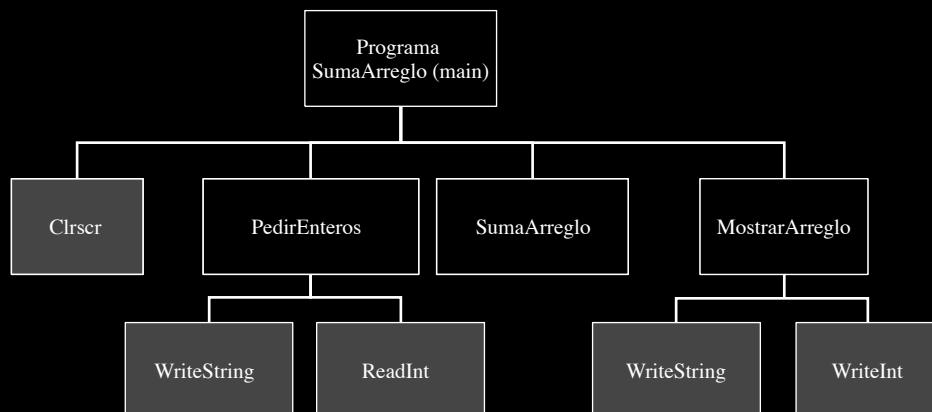
```
TITLE main.asm
INCLUDE Irvine32.inc
INCLUDE vars.inc
.code
main PROC
    mov    cuenta,2000h
    mov    eax,SIM1
    exit
main ENDP
END main
```

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 al programa en la directiva END.

8.6.4 Ejemplo: programa SumaArreglo

El programa *SumaArreglo*, que presentamos por primera vez en el capítulo 5, es un programa que puede separarse fácilmente en módulos. Para un breve repaso del diseño del programa, vamos a analizar el diagrama de estructura (figura 8-10). Los rectángulos sombreados se refieren a los procedimientos en la biblioteca de enlace del libro. El procedimiento **main** llama a **PedirEnteros**, 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. Eso es lo que hicimos para el programa *SumaArreglo*, que mostraremos en la siguiente sección.

FIGURA 8-10 Diagrama de estructura del programa SumaArreglo.



8.6.5 Creación de módulos mediante el uso de Extern

Vamos a mostrar dos versiones del programa SumaArreglo de varios módulos. Esta sección utilizará la tradicional directiva EXTERN para hacer referencia a las funciones en módulos separados. Más adelante, en la sección 8.6.6 implementaremos el mismo programa, utilizando las capacidades avanzadas de INVOKE, PROTO y PROC.

PedirEnteros *_pedir.asm* contiene el código fuente para el procedimiento PedirEnteros, el cual muestra indicadores pidiendo al usuario que introduzca tres enteros, recibe los valores de entrada mediante una llamada a ReadInt y los inserta en un arreglo:

```
TITLE Pedir enteros          (_pedir.asm)
INCLUDE Irvine32.inc
.code
;-----
PedirEnteros PROC
    ; Pide al usuario un arreglo de enteros y lo llena
    ; con la entrada del usuario.
    ; Recibe:
    ;     ptrIndicador:PTR BYTE           ; cadena del mensaje indicador
    ;     ptrArreglo:PTR DWORD           ; apuntador al arreglo
    ;     tamArreglo:DWORD              ; tamaño del arreglo
    ; Devuelve: nada
;
tamArreglo    EQU [ebp+16]
ptrArreglo    EQU [ebp+12]
ptrIndicador  EQU [ebp+8]

    enter    0,0
    pushad                           ; guarda todos los registros

    mov     ecx,tamArreglo
    cmp     ecx,0                     ; ¿tamaño del arreglo <= 0?
    jle     L2                         ; sí: termina
    mov     edx,ptrIndicador         ; dirección del indicador
    mov     esi,ptrArreglo
    mov     eax,[esi]
    add     esi,4                     ; siguiente entero
    loop    L1                         ; muestra la cadena
    call    ReadInt                  ; lee entero y lo coloca en EAX
    call    Crlf                      ; avanza a la siguiente línea de salida
    mov     [esi],eax                ; lo almacena en el arreglo
    add     esi,4                     ; siguiente entero
    L1:   call    WriteString        ; muestra la cadena
    call    ReadInt                  ; lee entero y lo coloca en EAX
    call    Crlf                      ; avanza a la siguiente línea de salida
    mov     [esi],eax                ; lo almacena en el arreglo
    add     esi,4                     ; siguiente entero
    add     esi,4                     ; siguiente entero
    loop    L1                         ; muestra la cadena
    L2:   popad                         ; restaura todos los registros
    leave                           ; restaura la pila
    ret     12                         ; restaura la pila
PedirEnteros ENDP
END
```

SumaArreglo El módulo *_sumarreg.asm* contiene el procedimiento SumaArreglo, el cual calcula la suma de los elementos del arreglo y devuelve un resultado en EAX:

```
TITLE Procedimiento SumaArreglo      (_sumarreg.asm)
INCLUDE Irvine32.inc
.code
SumaArreglo PROC
    ; Calcula la suma de un arreglo de enteros de 32 bits.
    ; Recibe:
```

```

;     ptrArreglo          ; apuntador al arreglo
;     tamArreglo          ; tamaño del arreglo (DWORD)
; Devuelve: EAX = suma
;-----
ptrArreglo EQU [ebp+8]
tamArreglo EQU [ebp+12]
    enter 0,0
    push  ecx           ; no mete EAX
    push  esi
    mov   eax,0          ; establece la suma a cero
    mov   esi,ptrArreglo
    mov   ecx,tamArreglo
    cmp   ecx,0          ; ¿el tamaño del arreglo <= 0?
    jle   L2             ; sí: termina
L1: add   eax,[esi]      ; agrega cada entero a suma
    add   esi,4          ; apunta al siguiente entero
    loop  L1             ; repite para el tamaño del arreglo
L2: pop   esi
    pop   ecx           ; devuelve la suma en EAX
    leave
    ret   8              ; restaura la pila
SumaArreglo ENDP
END

```

MostrarSuma El módulo _mostrar.asm contiene el procedimiento MostrarSuma, el cual muestra una etiqueta, seguida de la suma del arreglo:

```

TITLE Procedimiento MostrarSuma      (_mostrar.asm)

INCLUDE Irvine32.inc
.code
;-----
MostrarSuma PROC
; Muestra la suma en la consola.
; Recibe:
;     ptrIndicador        ; desplazamiento de la cadena indicadora
;     laSuma               ; la suma del arreglo (DWORD)
; Devuelve: nada
;-----
laSuma     EQU [ebp+12]
ptrIndicador EQU [ebp+8]
    enter 0,0
    push  eax
    push  edx
    mov   edx,ptrIndicador ; apuntador al indicador
    call  WriteString
    mov   eax,laSuma
    call  WriteInt         ; muestra EAX
    call  Crlf
    pop   edx
    pop   eax
    leave
    ret   8                ; restaura la pila
MostrarSuma ENDP
END

```

Módulo de arranque El módulo *Suma_main.asm* contiene el procedimiento de arranque (main), el cual contiene directivas EXTERN para los tres procedimientos externos. Para que el código fuente sea más amigable al usuario, la directiva EQU redefine los nombres de los procedimientos:

SumaArreglo	EQU SumaArreglo@0
PedirEnteros	EQU PedirEnteros@0
MostrarSuma	EQU MostrarSuma@0

Justo antes de cada llamada a los procedimientos, un comentario describe el orden de los parámetros. Este programa utiliza la convención STDCALL para el paso de parámetros:

```
TITLE Programa de suma de enteros (Suma_main.asm)

; Ejemplo con varios módulos:
; Este programa recibe como entrada varios enteros del usuario,
; los almacena en un arreglo, calcula la suma del
; arreglo y la muestra en pantalla.

INCLUDE Irvine32.inc

EXTERN PedirEnteros@0:PROC
EXTERN SumaArreglo@0:PROC, MostrarSuma@0:PROC

; Redefine los símbolos externos por conveniencia
SumaArreglo      EQU SumaArreglo@0
PedirEnteros     EQU PedirEnteros@0
MostrarSuma       EQU MostrarSuma@0

; modifique Cuenta para cambiar el tamaño del arreglo:
Cuenta = 3

.data
indicador1 BYTE  "Escriba un entero con signo: ",0
indicador2 BYTE  "La suma de los enteros es: ",0
arreglo    DWORD   Cuenta DUP(?)
suma       DWORD   ?

.code
main PROC
    call Clrscr
    ; PedirEnteros( addr indicador1, addr arreglo, Cuenta )
    push Cuenta
    push OFFSET arreglo
    push OFFSET indicador1
    call PedirEnteros

    ; sum = SumaArreglo( addr arreglo, Cuenta )
    push Cuenta
    push OFFSET arreglo
    call SumaArreglo
    mov suma,eax

    ; MostrarSuma( addr indicador2, suma )
    push suma
    push OFFSET indicador2
    call MostrarSuma

    call Crlf
    exit
main ENDP
END main
```

Los archivos de código fuente para este programa se almacenan en el directorio de programas de ejemplo, en una carpeta llamada cap08\SumaMod32_tradicional.

A continuación, veremos cómo cambiaría este programa, si se escribiera usando las directivas INVOKE y PROTO de Microsoft.

8.6.6 Creación de módulos mediante el uso de INVOKE y PROTO

Se pueden crear programas con varios módulos mediante el uso de las directivas avanzadas INVOKE, PROTO y la directiva extendida PROC (sección 8.5) de Microsoft. Su principal ventaja en comparación con el uso más tradicional de CALL y EXTERN es su habilidad para relacionar las listas de argumentos que pasa INVOKE con las listas correspondientes de los parámetros que declara PROC.

Vamos a recrear el programa SumaArreglo, usando las directivas INVOKE, PROTO y la directiva avanzada PROC. Un buen primer paso es crear un archivo de inclusión que contenga una directiva PROTO para cada procedimiento externo. Cada módulo incluirá este archivo (usando la directiva INCLUDE) sin incurrir en ninguna sobrecarga por el tamaño del código o en tiempo de ejecución. Si un módulo no llama a un procedimiento específico, el ensamblador ignora la correspondiente directiva PROTO. El código fuente de este programa se encuentra en la carpeta \cap08\SumMod32_avanzado.

El archivo de inclusión suma.inc He aquí el archivo de inclusión *suma.inc* para nuestro programa:

```
; (suma.inc)
INCLUDE Irvine32.inc

PedirEnteros PROTO,
    ptrIndicador:PTR BYTE,          ; cadena de mensaje indicador
    ptrArreglo:PTR DWORD,          ; apunta al arreglo
    tamArreglo:DWORD               ; tamaño del arreglo

SumaArreglo PROTO,
    ptrArreglo:PTR DWORD,          ; apunta al arreglo
    cuenta:DWORD                  ; tamaño del arreglo

MostrarSuma PROTO,
    ptrIndicador:PTR BYTE,          ; cadena de mensaje indicador
    laSuma:DWORD                   ; suma del arreglo
```

El módulo _pedir El archivo *_pedir.asm* utiliza la directiva PROC para declarar parámetros en el procedimiento PedirEnteros. Utiliza una directiva INCLUDE para copiar *suma.inc* a este archivo:

```
TITLE Pedir enteros           (_pedir.asm)

INCLUDE suma.inc              ; obtiene prototipos de procedimientos
.code
;-----
PedirEnteros PROC,
    ptrIndicador:PTR BYTE,          ; cadena de mensaje indicador
    ptrArreglo:PTR DWORD,          ; apuntador al arreglo
    tamArreglo:DWORD               ; tamaño del arreglo
;
; Pide al usuario un arreglo de enteros y lo llena
; con la entrada del usuario.
; Devuelve: nada
;-----
        pushad                  ; guarda todos los registros
        mov  ecx,tamArreglo
        cmp  ecx,0                 ; ¿el tamaño del arreglo <= 0?
```

```

jle L2 ; sí: termina
mov edx,ptrIndicador ; dirección del indicador
mov esi,ptrArreglo

L1: call WriteString ; muestra la cadena
    call ReadInt ; lee entero y lo coloca en EAX
    callCrLf ; avanza a la siguiente línea de salida
    mov [esi],eax ; lo almacena en el arreglo
    add esi,4 ; siguiente entero
    loop L1

L2: popad ; restaura todos los registros
    ret
PedirEnteros ENDP
END

```

Comparada con la versión anterior de PedirEnteros, ahora se omitieron las instrucciones **enter 0,0** y **leave**, ya que MASM las generará cuando encuentre la directiva PROC con los parámetros declarados. Además, la instrucción RET no necesita un parámetro constante (PROC se encarga de eso).

El módulo _sumarreg A continuación, el archivo *_sumarreg.asm* contiene el procedimiento SumaArreglo:

```

TITLE Procedimiento SumaArreglo      (_sumarreg.asm)
INCLUDE suma.inc
.code
;-----
SumaArreglo PROC,
    ptrArreglo:PTR DWORD,          ; apuntador al arreglo
    tamArreglo:DWORD              ; tamaño del arreglo
;
; Calcula la suma de un arreglo de enteros de 32 bits.
; Devuelve: EAX = suma
;-----
    push ecx                      ; no mete EAX
    push esi
    mov eax,0                      ; establece la suma a cero
    mov esi,ptrArreglo
    mov ecx,tamArreglo
    cmp ecx,0                      ; ¿el tamaño del arreglo <= 0?
    jle L2 ; sí: termina
L1: add eax,[esi]                 ; agrega cada entero a suma
    add esi,4                      ; apunta al siguiente entero
    loop L1 ; repite para el tamaño del arreglo
L2: pop esi
    pop ecx
    ret
SumaArreglo ENDP
END

```

El módulo _mostrar El archivo *_mostrar.asm* contiene el procedimiento MostrarSuma:

```

TITLE Procedimiento MostrarSuma(_mostrar.asm)
INCLUDE Suma.inc
.code

```

```

;-----[-----]
MostrarSuma PROC
    ptrIndicador:PTR BYTE,          ; cadena de mensaje indicador
    laSuma:DWORD                  ; la suma del arreglo
;
; Muestra la suma en la consola.
; Devuelve: nada
;-----[-----]
    push  eax
    push  edx
    mov   edx,ptrIndicador        ; apuntador al mensaje indicador
    call  WriteString
    mov   eax,laSuma
    call  WriteInt               ; muestra EAX
    call  Crlf
    pop   edx
    pop   eax
    ret
MostrarSuma ENDP
END

```

El modulo Suma_main El archivo *Suma_main.asm* (módulo de arranque) contiene el procedimiento main y llama a cada uno de los otros procedimientos. Utiliza INCLUDE para copiar los prototipos de los procedimientos de *suma.inc*:

```

TITLE Programa de suma de enteros           (Suma_main.asm)
INCLUDE suma.inc
Cuenta = 3
.data
indicador1 BYTE  "Escriba un entero con signo: ",0
indicador2 BYTE  "La suma de los enteros es: ",0
arreglo    DWORD  Cuenta DUP(?)
suma       DWORD  ?
.code
main PROC
    call Clrscr
    INVOKE PedirEnteros, ADDR indicador1, ADDR arreglo, Cuenta
    INVOKE SumaArreglo, ADDR arreglo, Cuenta
    mov   suma, eax
    INVOKE MostrarSuma, ADDR indicador2, suma
    call  Crlf
    exit
main ENDP
END main

```

Resumen Hemos mostrado dos formas de crear programas con varios módulos: en primer lugar, usando la directiva EXTERN más convencional y, en segundo lugar, usando las capacidades avanzadas de INVOKE, PROTO y PROC. Estas últimas directivas simplifican muchos detalles y están optimizadas para llamar a las funciones de la API de Windows. También ocultan una variedad de detalles, por lo que tal vez usted prefiera utilizar parámetros de pila explícitos junto con CALL y EXTERN.

8.6.7 Repaso de sección

1. (*Verdadero/Falso*): enlazar módulos OBJ es mucho más rápido que ensamblar archivos de código fuente ASM.
2. (*Verdadero/Falso*): separar un programa extenso en módulos cortos dificulta el mantenimiento del programa.

3. (*Verdadero/Falso*): en un programa con varios módulos, una instrucción END con una etiqueta ocurre sólo una vez, en el módulo de arranque.
4. (*Verdadero/Falso*): las directivas PROTO utilizan memoria, por lo que hay que tener cuidado de no incluir una directiva PROTO para un procedimiento, a menos que realmente se le vaya a llamar.

8.7 Resumen del capítulo

Hay dos tipos básicos de parámetros para los procedimientos: parámetros de registro y parámetros de pila. Las bibliotecas Irvine32 e Irvine16 utilizan parámetros de registro, los cuales están optimizados para agilizar la ejecución del programa. Por desgracia, tienden a crear un amontonamiento de código en los programas que hacen las llamadas. Los parámetros de pila son la alternativa. El programa que hace la llamada debe meter los argumentos de los procedimientos a la pila.

Un marco de pila (o registro de activación) es el área de la pila que se aparta para la dirección de retorno de un procedimiento, los parámetros que recibe, las variables locales y los registros que se guardan. El marco de pila se crea cuando el programa en ejecución empieza a ejecutar un procedimiento.

Cuando se mete una copia del argumento de un procedimiento a la pila, se *pasa por valor*. Cuando se mete la dirección de un argumento en la pila, se *pasa por referencia*; el procedimiento puede modificar la variable a través de su dirección. Los arreglos deben pasarse por referencia, para evitar tener que meter todos los elementos de un arreglo en la pila.

Para acceder a los parámetros de los procedimientos se utiliza el direccionamiento indirecto con el registro EBP. Las expresiones como [ebp+8] nos proporcionan un alto nivel de control sobre el direccionamiento de los parámetros de pila. La instrucción LEA devuelve el desplazamiento de cualquier tipo de operando indirecto. LEA es idealmente adecuada para usarse con los parámetros de pila.

La instrucción ENTER completa el marco de pila: guarda a EBP en la pila y reserva espacio para las variables locales. La instrucción LEAVE termina el marco de pila para un procedimiento, al invertir la acción de una instrucción ENTER previa.

Un procedimiento recursivo es uno que se llama a sí mismo, ya sea en forma directa o indirecta. La recursividad, que es la práctica de llamar procedimientos recursivos, puede ser una poderosa herramienta al trabajar con estructuras de datos que tengan patrones repetitivos.

La directiva LOCAL declara a una o más variables locales dentro de un procedimiento. Debe colocarse en la línea que sigue justo después de una directiva PROC. Las variables locales tienen distintas ventajas en comparación con las variables globales:

- Puede restringirse el acceso al nombre y al contenido de una variable local, sólo para el procedimiento que la contiene. Las variables locales son útiles cuando se depuran programas, ya que sólo un número limitado de instrucciones del programa pueden modificarlas.
- El tiempo de vida de una variable local está limitado al alcance de ejecución del procedimiento que la contiene. Las variables locales hacen un uso eficiente de la memoria, debido a que puede usarse el mismo espacio de almacenamiento para otras variables.
- Puede usarse el mismo nombre de variable en más de un procedimiento, sin que se produzca un conflicto de nombres.
- Las variables locales pueden usarse en procedimientos recursivos para almacenar valores en la pila. Si en vez de ellas se utilizaran variables globales, sus valores se sobrescribirían cada vez que el procedimiento se llamara a sí mismo.

La directiva INVOKE es un sustituto más poderoso para la instrucción CALL de Intel, ya que nos permite pasar varios argumentos. Puede usarse el operador ADDR para pasar un apuntador al llamar a un procedimiento mediante la directiva INVOKE.

La directiva PROC declara el nombre de un procedimiento con una lista de parámetros con nombre. La directiva PROTO crea un prototipo para un procedimiento existente. Un prototipo declara el nombre y la lista de parámetros de un procedimiento.

MASM utiliza la directiva .MODEL para determinar varias características importantes de un programa: su tipo de modelo de memoria, el esquema de nomenclatura de las funciones y la convención para el paso de parámetros. Los programas en modo de direccionamiento real que hemos mostrado hasta ahora utilizan el modelo pequeño de memoria, ya que éste mantiene a todo el código dentro de un solo segmento, y a todos los datos (incluyendo la pila) dentro de un solo segmento también. Los programas en modo protegido utilizan el modelo plano de memoria, en el cual todos los desplazamientos son de 32 bits, y el código y los datos pueden ser hasta de 4GB. Los especificadores de lenguaje más comunes son C y STDCALL.

Un programa de aplicación de cualquier tamaño es difícil de administrar cuando todo su código fuente se encuentra en el mismo archivo. Es más conveniente descomponer el programa en varios archivos de código fuente (llamados módulos), para que sea fácil ver y editar cada archivo.

8.8 Ejercicios de programación

Puede completar los siguientes ejercicios en modo protegido o en modo de direccionamiento real.

1. Intercambio de enteros

Cree un arreglo de enteros ordenados al azar. Usando el procedimiento **Intercambiar** de la sección 8.5.6 como herramienta, escriba un ciclo que intercambie cada par consecutivo de enteros en el arreglo.

2. Procedimiento DumpMem

Escriba un procedimiento de envoltura para el procedimiento **DumpMem** de la biblioteca de enlace, usando parámetros de pila. El nombre puede ser un poco distinto, como por ejemplo **DumpMemory**. A continuación se muestra un ejemplo de cómo debe llamarse:

```
INVOKE DumpMemory,OFFSET arreglo,LENGTHOF arreglo,TYPE arreglo
```

Escriba un programa de prueba que llame a su procedimiento varias veces, usando una variedad de tipos de datos.

3. Factorial no recursivo

Escriba una versión no recursiva del procedimiento **Factorial** (sección 8.3.2) que utilice un ciclo. Escriba un programa corto que evalúe en forma interactiva su procedimiento Factorial. Permita que el usuario introduzca el valor de n . Muestre el factorial calculado.

4. Comparación de factoriales

Escriba un programa que compare las velocidades en tiempo de ejecución del procedimiento recursivo **Factorial** de la sección 8.3.2 y del procedimiento Factorial no recursivo, escrito para el ejercicio de programación anterior. Use el procedimiento **GetMseconds** de la biblioteca de enlace del libro para medir y mostrar el número de milisegundos requeridos para llamar a cada procedimiento Factorial varios miles de veces en un solo instante.

5. Máximo común divisor

Escriba una implementación recursiva del algoritmo de Euclides para encontrar el máximo común divisor (MCD) de dos enteros. Hay descripciones de este algoritmo disponibles en libros de álgebra y en sitios Web. Nota: en los ejercicios de programación del capítulo 7 se proporcionó una versión no recursiva del problema del MCD.

Notas finales

1. Consulte la sección “Calls for Block-Structured Languages” (Llamadas a procedimientos para lenguajes estructurados por bloques), capítulo 6, del Manual para el desarrollador de software de la arquitectura Intel IA-32, Volumen 1.
2. NEARSTACK combina la pila y los datos del programa en un solo segmento físico, junto con los datos. FARSTACK utiliza distintos segmentos para código y datos, por lo que el registro DS contiene un valor distinto al del registro SS.

CADENAS Y ARREGLOS

- | | |
|--|--|
| <ul style="list-style-type: none">9.1 Introducción9.2 Instrucciones primitivas de cadenas<ul style="list-style-type: none">9.2.1 MOVSB, MOVSW y MOVS9.2.2 CMPSB, CMPSW y CMPSD9.2.3 SCASB, SCASW y SCASD9.2.4 STOSB, STOSW y STOSD9.2.5 LODSB, LODSW y LODSD9.2.6 Repaso de sección9.3 Procedimientos de cadenas seleccionados<ul style="list-style-type: none">9.3.1 Procedimiento Str_compare9.3.2 Procedimiento Str_length9.3.3 Procedimiento Str_copy9.3.4 Procedimiento Str_trim9.3.5 Procedimiento Str_ucase9.3.6 Programa de demostración de la biblioteca de cadenas9.3.7 Repaso de sección | <ul style="list-style-type: none">9.4 Arreglos bidimensionales<ul style="list-style-type: none">9.4.1 Ordenamiento de filas y columnas9.4.2 Operandos base-índice9.4.3 Operandos base-índice-desplazamiento9.4.4 Repaso de sección9.5 Búsqueda y ordenamiento de arreglos de enteros<ul style="list-style-type: none">9.5.1 Ordenamiento de burbuja9.5.2 Búsqueda binaria9.5.3 Repaso de sección9.6 Resumen del capítulo9.7 Ejercicios de programación |
|--|--|

9.1 Introducción

Si aprende a procesar cadenas y arreglos con eficiencia, podrá dominar el área más común de optimización de código. Los estudios han demostrado que la mayoría de los programas invierten el 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 le 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 o (Irvine16). Sus implementaciones son bastante similares al código que podríamos ver en una implementación de la biblioteca de cadenas estándar de C. La tercera parte de este capítulo muestra cómo manipular arreglos bidimensionales, usando modos de direccionamiento indirecto avanzados: base-índice y base-índice-desplazamiento. En la sección 4.4 presentamos el direccionamiento indirecto simple.

La última sección del capítulo, titulada Búsqueda y ordenamiento de arreglos enteros, es la más interesante. Aquí verá lo fácil que es implementar dos de los algoritmos de procesamiento de arreglos más comunes en las ciencias computacionales: el ordenamiento tipo burbuja y la búsqueda binaria. Es muy conveniente que estudie estos algoritmos en Java o C++, además del lenguaje ensamblador.

9.2 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 tabla 9-1 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.

Tabla 9-1 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, CMPSD	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 con el mismo valor, y no se pueden cambiar. Por otro lado, en el modo de direccionamiento real, los programadores de ASM manipulan con frecuencia a ES y DS.

En el modo de 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 En 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 Las instrucciones de primitiva de cadenas incrementan o decrementan a ESI y EDI, según el estado de la bandera Dirección (vea la tabla 9-2). 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 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.

Tabla 9-2 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

9.2.1 MOVSB, MOVSW y MOVSD

Las instrucciones MOVSB, MOVSW y MOVSD copian datos de la ubicación de memoria a la que apunta ESI, hasta la ubicación de memoria a la que apunta 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

Puede utilizar un prefijo de repetición con MOVSB, MOVSW y MOVSD. La bandera Dirección determina si ESI y EDI se van a incrementar o a decrementar. El tamaño del incremento o decrecimiento se muestra en la siguiente tabla:

Instrucción	Valor que se agrega o se resta a ESI y EDI
MOVSB	1
MOVSW	2
MOVSD	4

Ejemplo: copiar arreglo de dobles palabras Suponga 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(0FFFFFFFh)
destino DWORD 20 DUP(?)
.code
cld
mov ecx, LENGTHOF origen ; dirección = avance
mov esi, OFFSET origen ; establece contador REP
mov edi, OFFSET destino ; ESI apunta al origen
mov rep movsd ; EDI apunta al destino
; copia dobles palabras

```

9.2.2 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

Puede 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 a la que apunta EDI. Observe que el orden de los operandos en CMPS es opuesto 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 Suponga que desea 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, borre la bandera Dirección (dirección de avance), inicialice ECX como contador y utilice un prefijo repetido con CMPSD:

```
mov    esi,OFFSET origen
mov    edi,OFFSET destino
cld
mov    ecx,LENGTHOF origen           ; dirección = avance
repe   cmpsd                         ; contador de repetición
                                         ; 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 tres caracteres son idénticos, pero existe un carácter adicional en la segunda cadena.

El siguiente programa utiliza CMPSB 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 comparar dos cadenas
; de la misma longitud.

INCLUDE Irvine32.inc

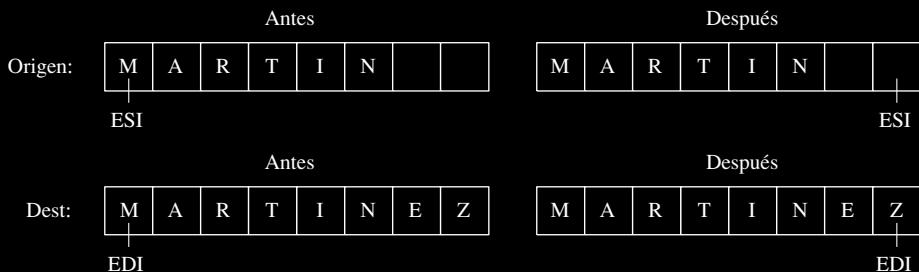
.data
origen BYTE "MARTIN "
dest   BYTE "MARTINEZ"
cad1   BYTE "La cadena de origen es mas chica",0dh,0ah,0
cad2   BYTE "La cadena de origen no es mas chica",0dh,0ah,0

.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”. En la figura 9-1, ESI y EDI quedan apuntando 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 quedado apuntando 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 llenar “MARTIN” con espacios a la derecha, para que fuera de la misma longitud que “MARTINEZ”. El proceso de llenar cadenas con espacios impone una restricción extraña para el manejo de las cadenas, que eliminamos del procedimiento **Str_compare** en la sección 9.3.1.

FIGURA 9–1 Comparación de dos cadenas mediante el uso de CMPSB.



9.2.3 SCASB, SCASW y 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 coincide con cada valor subsiguiente en memoria. El prefijo REPNE explora hasta que AL/AX/EAX coincide 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.

9.2.4 STOSB, STOSW y STOSD

Las instrucciones STOSB, STOSW y STOSD almacenan en memoria el contenido de AL/AX/EAX, respectivamente, en el desplazamiento al que apunta EDI. EDI se incrementa o decremente con base en el estado de la bandera Dirección. Cuando se utilizan con el prefijo REP, estas instrucciones son útiles para llenar 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

```

9.2.5 LODSB, LODSW y LODSD

Las instrucciones LODSB, LODSW y LODSD cargan un byte o palabra de la 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 esté 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            ; dirección = avance
    mov edi,esi                      ; índice de origen
    mov ecx,LENGTHOF arreglo         ; índice de destino
    mov edx,multiplicador           ; 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

```

9.2.6 Repaso de sección

1. En referencia a las instrucciones primitivas de cadenas, ¿qué registro de 32 bits se conoce como el *acumulador*?
2. ¿Qué instrucción compara un entero de 32 bits en el acumulador con el contenido de la memoria a la que apunta EDI?
3. ¿Qué registro índice utiliza la instrucción STOSD?
4. ¿Qué instrucción copia datos de la ubicación de memoria direccionada por EDI hacia AX?
5. ¿Qué hace el prefijo REPZ para una instrucción CMPSB?
6. ¿Qué valor de la bandera Dirección hace que los registros índice se desplacen hacia atrás en la memoria, al ejecutar primitivas de cadenas?
7. Cuando se utiliza un prefijo de repetición con STOSW, ¿qué valor se suma o se resta al registro índice?

8. ¿De qué manera la instrucción CMPS es ambigua?
9. *Reto:* cuando la bandera Dirección está en cero y SCASB encuentra un carácter coincidente, ¿hacia dónde apunta EDI?
10. *Reto:* al explorar un arreglo en busca de la primera coincidencia de un carácter específico, ¿qué prefijo de repetición es más adecuado?

9.3 Procedimientos de cadenas seleccionados

En esta sección demostraremos varios procedimientos de las bibliotecas Irvine32 e Irvine16, los cuales manipulan cadenas con terminación nula. Los procedimientos son muy similares a las funciones en la biblioteca estándar de C:

```
; Copia una cadena de origen a una cadena de destino
Str_copy PROTO,
    source:PTR BYTE,           ; cadena de origen
    target:PTR BYTE           ; cadena de destino

; Devuelve la longitud de una cadena (excluyendo el byte nulo) en EAX.
Str_length PROTO,
    pString:PTR BYTE          ; apuntador a la cadena

; Compara string1 con string2. Establece las banderas
; Cero y Acarreo de la misma forma que la instrucción CMP.
Str_compare PROTO,
    string1:PTR BYTE,
    string2:PTR BYTE

; Recorta un carácter dado a la derecha de una cadena.
; El segundo argumento es el carácter a recortar.
Str_trim PROTO,
    pString:PTR BYTE,          ; apunta a la cadena
    char:BYTE                  ; carácter a eliminar

; Convierte una cadena a mayúsculas.
Str_ucase PROTO,
    pString:PTR BYTE
```

9.3.1 Procedimiento Str_compare

El procedimiento **Str_compare** compara dos cadenas. El formato de la llamada es:

```
INVOKE Str_compare, ADDR cadena1, ADDR cadena2
```

Este procedimiento compara las cadenas en orden hacia delante, empezando en el primer byte. La comparación es sensible al uso de mayúsculas y minúsculas, ya que los códigos ASCII son distintos para las letras mayúsculas y las minúsculas. El procedimiento no devuelve un valor, pero las banderas Acarreo y Cero pueden interpretarse como se muestra en la tabla 9-3, usando los argumentos *cadena1* y *cadena2*.

Tabla 9-3 Banderas afectadas por el procedimiento Str_compare.

Relación	Bandera Acarreo	Bandera Cero	Bifurca si es verdadero
cadena1 < cadena2	1	0	JB
cadena1 == cadena2	0	1	JE
cadena1 > cadena2	0	0	JA

En la sección 6.2.7 encontrará una explicación de cómo CMP activa las banderas Acarreo y Cero. A continuación se muestra un listado del procedimiento **Str_compare**. Vea el programa *Comparar.asm* para una demostración (*Nota: en este texto se tradujeron los comentarios para facilitarle al lector la comprensión del código; en las bibliotecas Irvine32 e Irvine16 están en inglés*):

```
Str_compare PROC USES eax edx esi edi,
    string1:PTR BYTE,
    string2:PTR BYTE
;
; Compara dos cadenas.
; No devuelve nada, pero las banderas Cero y Acarreo se afectan
; de la misma forma que con la instrucción CMP.
;-----
    mov    esi,string1
    mov    edi,string2

L1:   mov    al,[esi]
    mov    dl,[edi]
    cmp    al,0           ; ¿final de string1?
    jne    L2             ; no
    cmp    dl,0           ; sí: ¿final de string2?
    jne    L2             ; no
    jmp    L3             ; sí, termina con ZF = 1

L2:   inc    esi           ; apunta al siguiente
    inc    edi
    cmp    al,dl          ; ¿los caracteres son iguales?
    je     L1             ; sí: continúa el ciclo
    jmp    L3             ; no: termina con las banderas activadas

L3:   ret
Str_compare ENDP
```

Podríamos haber usado la instrucción CMPSB al implementar **Str_compare**, pero tendríamos que conocer la longitud de la cadena más larga. Se hubieran requerido dos llamadas al procedimiento **Str_length**. En este caso en especial, es más fácil verificar los terminadores nulos en ambas cadenas dentro del mismo ciclo. CMPSB es más efectiva cuando tratamos con cadenas extensas o arreglos de longitudes conocidas.

9.3.2 Procedimiento Str_length

El procedimiento **Str_length** devuelve la longitud de una cadena en el registro EAX. Al llamarlo, hay que pasarle el desplazamiento de la cadena. Por ejemplo:

```
INVOKE Str_length, ADDR miCadena
```

He aquí la implementación del procedimiento:

```
Str_length PROC USES edi,
    pString:PTR BYTE           ; apuntador a la cadena
    mov edi,pString
    mov eax,0                  ; cuenta de caracteres

L1:   cmp BYTE PTR[edi],0      ; ¿final de la cadena?
    je L2                      ; sí: termina
    inc edi                    ; no: apunta al siguiente
    inc eax                    ; suma 1 a la cuenta
    jmp L1

L2:   ret
Str_length ENDP
```

Vea el programa *Longitud.asm* para una demostración de este procedimiento.

9.3.3 Procedimiento Str_copy

El procedimiento **Str_copy** copia una cadena con terminación nula, de una ubicación de origen a una ubicación de destino. Antes de llamar a este procedimiento, debemos asegurarnos de que el operando de destino sea lo bastante grande como para poder almacenar la cadena que se va a copiar. La sintaxis para llamar a **Str_copy** es:

```
INVOKE Str_copy, ADDR origen, ADDR destino
```

El procedimiento no devuelve valores. He aquí la implementación:

```
Str_copy PROC USES eax ecx esi edi,
    source:PTR BYTE,           ; cadena de origen
    target:PTR BYTE,           ; cadena de destino
;
; Copia una cadena del origen al destino.
; Requiere: la cadena de destino debe contener suficiente
;           espacio para guardar una copia de la cadena de origen.
;-----
    INVOKE Str_length,source   ; EAX = Longitud origen
    mov  ecx,eax               ; cuenta de REP
    inc  ecx                  ; suma 1 por el byte nulo
    mov  esi,source
    mov  edi,target
    cld
    rep  movsb                ; dirección = avance
    ret
    mov  eax,0                 ; copia la cadena
Str_copy ENDP
```

Vea el programa *CopiarCad.asm* para una demostración de este procedimiento.

9.3.4 Procedimiento Str_trim

El procedimiento **Str_trim** elimina todas las ocurrencias de un carácter a la derecha, seleccionado de una cadena con terminación nula. La sintaxis para llamar a este procedimiento es:

```
INVOKE Str_trim, ADDR cadena, car_a_eliminar
```

La lógica para este procedimiento es interesante, ya que hay que revisar una variedad de casos posibles (los cuales se muestran a continuación, con el carácter # a la derecha):

1. La cadena está vacía.
2. La cadena contiene otros caracteres seguidos de uno o más caracteres a la derecha, como en “Hola##”.
3. La cadena sólo contiene un carácter, el carácter que se va a eliminar, como en “#”.
4. La cadena no contiene el carácter que se desea eliminar, como en “Hola” o “H”.
5. La cadena contiene uno o más caracteres que se desean eliminar, seguidos de uno o más caracteres, como en “#H” o en “##Hola”.

Podemos usar el procedimiento **Str_trim** para eliminar todos los espacios (o cualquier otro carácter repetido) del final de una cadena. La manera más sencilla de truncar caracteres de una cadena es insertar un byte nulo justo antes de los caracteres que se desean retener. Cualquier carácter después del byte nulo se vuelve insignificante. He aquí el código fuente del procedimiento. El programa *Recortar.asm* prueba a **Str_trim**:

```
Str_trim PROC USES eax ecx edi,
    pString:PTR BYTE,          ; apunta a una cadena
    char:BYTE                 ; carácter a eliminar
;
; Elimina todas las ocurrencias de un carácter dado, de
; la parte final de una cadena.
; Devuelve: nada
;-----
```

```

    mov edi,pString
    INVOKE Str_length,edi          ; devuelve la longitud en EAX
    cmp eax,0                      ; ¿cadena de longitud cero?
    je L2                          ; sí: termina
    mov ecx,eax                   ; no: contador = longitud cadena
    dec eax
    add edi, eax                  ; EDI apunta al último carácter
    mov al,char                     ; carácter a recortar
    std                           ; dirección = avance
    repe scasb                    ; salta más allá del carácter a recortar
    jne L1                         ; ¿se eliminó el primer carácter?
    dec edi                        ; ajusta EDI: ZF=1 && ECX=0
    L1: mov BYTE PTR [edi+2],0     ; inserta byte nulo
    L2: ret
Str_trim ENDP

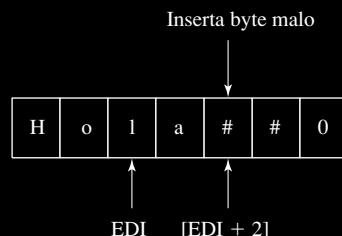
```

En todos los casos excepto uno, EDI se detiene 2 bytes detrás del carácter que deseamos sustituir con el carácter nulo. La tabla 9-4 muestra varios casos de prueba para cadenas no vacías. Con base en la primera definición de cadena de la tabla antes mencionada, la figura 9-2 muestra la posición de EDI cuando SCASB se detiene.

Tabla 9-4 Prueba del procedimiento Str_trim.

Definición de cadena	EDI cuando SCASB se detiene	Bandera Cero	ECX	Posición para guardar el carácter nulo
cad BYTE "Hola##",0	cad + 3	0	> 0	[edi + 2]
cad BYTE "#",0	cad - 1	1	0	[edi + 1]
cad BYTE "Hola",0	cad + 3	0	> 0	[edi + 2]
cad BYTE "H",0	cad - 1	0	0	[edi + 2]
cad BYTE "#H",0	cad + 0	0	> 0	[edi + 2]

FIGURA 9-2 Ejemplo de SCAS, después de encontrar una coincidencia.



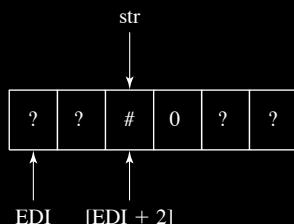
Cuando termina SCASB, se lleva a cabo una prueba especial para el único caso en el que la cadena contiene un solo carácter, y ese carácter es el que se va a eliminar. En este caso, EDI apunta sólo 1 byte adelante del carácter para sustituirlo con el carácter nulo (debido a que SCASB se detuvo porque ECX = 0, y no porque ZF = 1). Para compensar, decrementamos a EDI una vez antes de almacenar un byte nulo en [edi+2], como se muestra en la figura 9-3.

9.3.5 Procedimiento Str_ucase

El procedimiento Str_ucase convierte todos los caracteres de una cadena a mayúsculas. No devuelve un valor. Al llamar este procedimiento, hay que pasarle el desplazamiento de una cadena:

Invoke Str_ucase, ADDR miCadena

FIGURA 9-3 Inserción de un byte nulo, después de ejecutar SCAS.



He aquí la implementación del procedimiento:

```

Str_ucase PROC uses eax,esi
    pstring:PTR BYTE
; Convierte una cadena con terminación nula a mayúsculas.
; Devuelve: nada
;-----
    mov    esi,pString

L1:
    mov    al,[esi]           ; obtiene carácter
    cmp    al,0               ; ¿fin de la cadena?
    je     L3                ; sí: termina
    cmp    al,'a'             ; ¿debajo de "a"?
    jb    L2                ; no
    cmp    al,'z'             ; ¿encima de "z"?
    ja    L2                ; no
    and   BYTE PTR [esi],11011111b ; convierte el carácter
L2:   inc    esi              ; siguiente carácter
    jmp    L1
L3:   ret
Str_ucase ENDP

```

Vea el programa *Mayus.asm* para una demostración de este procedimiento.

9.3.6 Programa de demostración de la biblioteca de cadenas

El siguiente programa (DemoCad.asm) muestra ejemplos de cómo llamar a los procedimientos Str_trim, Str_ucase, Str_compare y Str_length de la biblioteca del libro:

```

TITLE Demostración de la biblioteca de cadenas (DemoCad.asm)

; Este programa demuestra los procedimientos de manejo de cadenas en
; la biblioteca de enlace del libro.

INCLUDE Irvine32.inc

.data
cadena_1 BYTE "abcde///",0
cadena_2 BYTE "ABCDE",0
msj0    BYTE "cadena_1 en mayusculas: ",0
msj1    BYTE "cadena_1 y cadena_2 son iguales",0
msj2    BYTE "cadena_1 es menor que cadena_2",0
msj3    BYTE "cadena_2 es menor que cadena_1",0
msj4    BYTE "La Longitud de cadena_2 es ",0
msj5    BYTE "cadena_1 despues de recortar: ",0

.code
main PROC

```

```
call  recortar_cadena
call  cambiar_mayusculas
call  comparar_cadenas
call  imprimir_longitud

exit
main ENDP

recortar_cadena PROC
; Elimina los caracteres a la derecha de cadena_1.

    INVOKE Str_trim, ADDR cadena_1, '/'
    mov   edx,OFFSET msj5
    call  WriteString
    mov   edx,OFFSET cadena_1
    call  WriteString
    call  Crlf

    ret
recortar_cadena ENDP

cambiar_mayusculas PROC
; Convierte cadena_1 a mayúsculas.

    mov   edx,OFFSET msj0
    call  WriteString
    INVOKE Str_ucase, ADDR cadena_1
    mov   edx,OFFSET cadena_1
    call  WriteString
    call  Crlf

    ret
cambiar_mayusculas ENDP

comparar_cadenas PROC
; Compara cadena_1 y cadena_2.

    INVOKE Str_compare, ADDR cadena_1, ADDR cadena_2
    .IF ZERO?
    mov   edx,OFFSET msj1
    .ELSEIF CARRY?
    mov   edx,OFFSET msj2      ; cadena 1 es menor que...
    .ELSE
    mov   edx,OFFSET msj3      ; cadena 2 es menor que...
    .ENDIF
    call  WriteString
    call  Crlf

    ret
comparar_cadenas ENDP

imprimir_longitud PROC
; Muestra la longitud de cadena_2.

    mov   edx,OFFSET msj4
    call  WriteString
    INVOKE Str_length, ADDR cadena_2
    call  WriteDec
    call  Crlf

    ret
imprimir_longitud ENDP
END main
```

Los caracteres a la derecha se eliminan de cadena_1 mediante la llamada a Str_trim. La cadena se convierte a mayúsculas mediante una llamada al procedimiento Str_ucase.

Resultados del programa He aquí los resultados del programa de Demostración de la biblioteca de cadenas:

```
cadena_1 despues de recortar: abcde
cadena_1 en mayusculas: ABCDE
cadena_1 y cadena_2 son iguales
La longitud de cadena_2 es 5
```

9.3.7 Repaso de sección

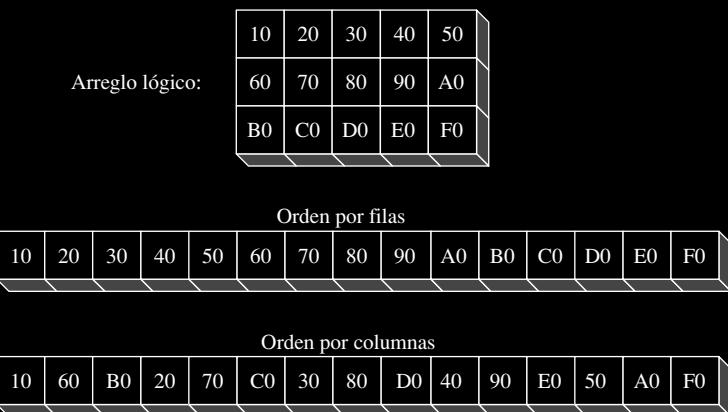
- (Verdadero/Falso): el procedimiento **Str_compare** se detiene cuando se llega al terminador nulo de la cadena más grande.
- (Verdadero/Falso): el procedimiento **Str_compare** no necesita utilizar a ESI y EDI para acceder a la memoria.
- (Verdadero/Falso): el procedimiento **Str_length** utiliza a SCASB para encontrar el terminador nulo al final de la cadena.
- (Verdadero/Falso): el procedimiento **Str_copy** evita que se copie una cadena en un área de memoria demasiado pequeña.
- ¿Qué configuración de la bandera Dirección se utiliza en el procedimiento **Str_trim**?
- Por qué el procedimiento **Str_trim** utiliza la instrucción JNE?
- ¿Qué ocurre en el procedimiento **Str_ucase** si la cadena de destino contiene un dígito?
- Reto: si el procedimiento **Str_length** usara SCASB, ¿qué prefijo de repetición sería el más apropiado?
- Reto: si el procedimiento **Str_length** usara SCASB, ¿cómo calcularía y devolvería la longitud de la cadena?

9.4 Arreglos bidimensionales

9.4.1 Ordenamiento de filas 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.

FIGURA 9–4 Orden por filas y por columnas.



Si implementa un arreglo bidimensional en lenguaje ensamblador, puede elegir cualquiera de los dos métodos de ordenamiento. En este capítulo utilizaremos el orden por filas. Si escribe subrutinas en lenguaje ensamblador para un lenguaje de alto nivel, debe seguir el orden especificado en su documentación.

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 ambos y le mostraremos ejemplos de cómo pueden utilizarse en forma efectiva.

9.4.2 Operando 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 base debe ser BX o BP. (Evite usar BP o EBP, excepto al direccionar la pila). El registro índice debe ser SI o DI. He aquí ejemplos 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 ax,[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 columna está en el registro índice. Por ejemplo, la siguiente tabla tiene tres filas y cinco columnas:

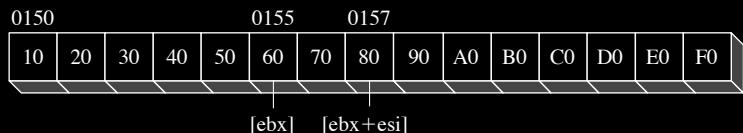
```
tablaB BYTE 10h, 20h, 30h, 40h, 50h
TamFila = ($ - tablaB)
        BYTE 60h, 70h, 80h, 90h, 0A0h
        BYTE 0B0h, 0C0h, 0D0h, 0E0h, 0F0h
```

La tabla está en orden por filas y el valor de TamFila constante lo calcula el ensamblador como el número de bytes en cada fila de la tabla. Suponga que deseamos localizar una entrada específica en la tabla, usando coordenadas de fila y de columna. Suponiendo que las coordenadas tienen base cero, la entrada en la fila 1, columna 2 contiene 80h. Establecemos EBX al desplazamiento de la tabla, sumamos (TamFila * indice_fila) para calcular el desplazamiento de la fila y establecemos ESI al índice de la columna:

```
indice_fila = 1
indice_columna = 2
mov ebx,OFFSET tablaB ; desplazamiento de la tabla
add ebx,TamFila * indice_fila ; desplazamiento de la fila
mov esi,indice_columna
mov al,[ebx + esi] ; AL = 80h
```

Suponga que el arreglo se encuentra en el desplazamiento 0150h. Entonces la dirección efectiva representada por EBX + ESI es 0157h. La figura 9-5 muestra cómo al sumar EBX y ESI se produce el desplazamiento del byte en tablaB[1,2]. Si la dirección efectiva apunta hacia fuera de la región de datos del programa, se produce un fallo de protección general.

FIGURA 9–5 Direcciónamiento de un arreglo con un operando base-índice.



Cálculo de la suma de una fila

El direccionamiento base índice simplifica muchas 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 llamado calc_suma_fila (vea *SumaFila.asm*) 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 tabla, EAX = índice de fila,
;          ECX = tamaño de 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 en la instrucción MOVZX.

Factores de escala

Si escribe código para un arreglo de valores tipo WORD, multiplique el operando por un factor de escala de 2. El siguiente ejemplo localiza el valor en la fila 1, columna 2.

```
tablaW WORD    10h, 20h, 30h, 40h, 50h
TamFilaw = ($ - tablaW)
        WORD    60h, 70h, 80h, 90h, 0A0h
        WORD    0B0h, 0C0h, 0D0h, 0E0h, 0F0h

.code
indice_fila = 1
indice_columna = 2
mov  ebx,OFFSET tablaW           ; desplazamiento de tabla
add  ebx, TamFilaw * indice_fila ; desplazamiento de fila
mov  esi, indice_columna
mov  ax,[ebx + esi*TYPE tablaW]   ; AX = 0080h
```

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:

```
tablaD DWORD  10h, 20h, ...etc.
.code
mov  eax,[ebx + esi*TYPE tablaD]
```

9.4.3 Operando 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[base + índice]

Desplazamiento puede ser el nombre de una 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 procesamiento 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:

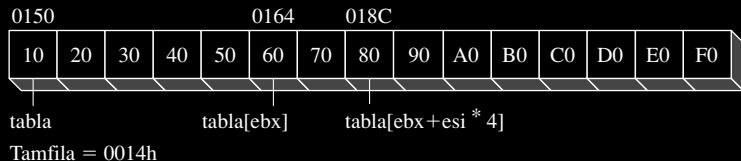
```
tablaD DWORD 10h, 20h, 30h, 40h, 50h
TamFila = ($ - tablaD)
        DWORD 60h, 70h, 80h, 90h, 0A0h
        DWORD 0B0h, 0C0h, 0D0h, 0E0h, 0F0h
```

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]
```

Suponga que la **tablaD** empieza en el desplazamiento 0150h. La figura 9-6 muestra las posiciones de EBX y ESI, relativas al arreglo. Los desplazamientos están en hexadecimal.

FIGURA 9-6 Ejemplo de base-índice-desplazamiento.



9.4.4 Repaso de sección

- En modo de 32 bits, ¿qué registros pueden usarse en un operando base-índice?
- Muestre un ejemplo de un operando base-índice en modo de 32 bits.
- Muestre un ejemplo de operando base-índice-desplazamiento en modo de 32 bits.
- Suponga que un arreglo bidimensional de dobles palabras tiene tres filas lógicas y cuatro columnas lógicas. Si ESI se utiliza como el índice de fila, ¿qué valor se sumaría a ESI para avanzar de una fila a la siguiente?
- Suponga que un arreglo bidimensional de dobles palabras tiene tres filas lógicas y cuatro columnas lógicas. Escriba una expresión, usando ESI y EDI, que dirija la tercera columna en la segunda fila. (La numeración para filas y columnas empieza en cero).
- En modo de direccionamiento real, ¿podría utilizar BP para dirigir un arreglo?
- En modo protegido, ¿podría utilizar EBP para dirigir un arreglo?

9.5 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

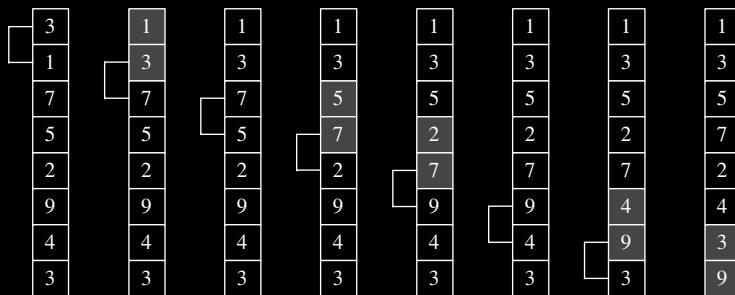
el mejor algoritmo para una aplicación específica que comprar una computadora más rápida. La mayoría de los estudiantes aprenden los métodos de búsqueda y ordenamiento usando lenguajes de alto nivel, como C++ y Java. El lenguaje ensamblador presenta una perspectiva diferente para el estudio de los algoritmos, 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.¹

La búsqueda y el ordenamiento nos dan la oportunidad de probar los modos de direccionamiento introducidos en este capítulo. 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.

9.5.1 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 figura 9-7 muestra el progreso de una ejecución (pasada) a través de un arreglo de enteros.

FIGURA 9-7 Primera ejecución a través de un arreglo (Ordenamiento de burbuja).



(Los valores sombreados se han intercambiado)

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). Suponga, 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 el arreglo 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 sería 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. Utilizaremos **N** para representar el tamaño del arreglo, **cx1** para representar el contador del ciclo externo y **cx2** para representar el contador del 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
}

```

Hemos omitido de manera intencional las cuestiones mecánicas, como guardar y restaurar el contador del ciclo externo. Observe que el contador del ciclo interno (**cx2**) se basa en el valor actual del contador del ciclo externo (**cx1**), que a su vez se decrementa en cada ejecución a través del arreglo.

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
;
; Ordena un arreglo de enteros de 32 bits con signo, en orden ascendente,
; usando el algoritmo de ordenamiento de burbuja.
; Recibe: apuntador a un arreglo, tamaño del arreglo
; Devuelve: nada
;-----
    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                   ; si [esi] <= [edi], no intercambia
    xchg   eax,[esi+4]           ; intercambia el par
    mov    [esi],eax
L3:   add    esi,4                ; mueve ambos apuntadores hacia delante
    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

```

9.5.2 Búsqueda binaria

Las búsquedas en los arreglos son algunas 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 del 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 los 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 en EAX. Este valor de retorno indica que se 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 número en el punto medio, restablece *último* a una posición más baja que el punto medio.
 4. Regresar al paso 1.

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 arreglo por un factor de n , el tiempo de búsqueda promedio se incrementa tan sólo 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 $\log_2(n + 1)$, redondeado al siguiente entero más grande. A continuación se muestra una implementación en C++ de una función de búsqueda binaria, diseñada para trabajar con un arreglo de enteros con signo:

```
int BusquedaBin( int valores[], const int valBusqueda, int cuenta )
{
    int primero = 0;
    int ultimo = cuenta - 1
    while( primero <= ultimo )
    {
        int medio = (ultimo + primero) / 2;
        if( valores[medio] < valBusqueda )
            primero = medio + 1;
        else if( valores[medio] > valBusqueda )
```

```

        ultimo = medio - 1;
    else
        return medio;           // éxito
    }
    return - 1;                // no se encontró
}

```

A continuación se muestra una implementación en lenguaje ensamblador del código de ejemplo en C++:

```

;-----[-----]
BusquedaBinaria PROC USES ebx edx esi edi,
    pArreglo:PTR DWORD,          ; apuntador a un arreglo
    Cuenta:DWORD,               ; tamaño del arreglo
    valBusqueda:DWORD           ; valor a buscar
LOCAL primero:DWORD,           ; primera posición
    ultimo:DWORD,               ; última posición
    medio:DWORD                 ; punto medio
;
; Busca un valor individual en un arreglo de enteros con signo.
; Recibe: Apuntador a un arreglo, tamaño del arreglo, valor a buscar.
; Devuelve: Si se encontró una coincidencia, EAX = la posición en el arreglo
; del elemento que coincide; en cualquier otro caso, EAX = -1.
;-----[-----]
        mov    primero,0          ; primero = 0
        mov    eax,Cuenta         ; ultimo = (cuenta - 1)
        dec    eax
        mov    ultimo,eax
        mov    edi,valBusqueda    ; EDI = valBusqueda
        mov    ebx,pArreglo        ; EBX apunta al arreglo
L1:   ; while primero <= ultimo
        mov    eax,primero
        cmp    eax,ultimo
        jg     L5                  ; termina la búsqueda
;
        medio = (ultimo + primero) / 2
        mov    eax,ultimo
        add    eax,primero
        shr    eax,1
        mov    medio,eax
;
; EDX = valores[medio]
        mov    esi,medio
        shl    esi,2                ; escala el valor medio por 4
        mov    edx,[ebx+esi]         ; EDX = valores[medio]
;
; if ( EDX < valBusqueda(EDI) )
        cmp    edx,edi
        jge    L2
;
        primero = medio + 1;
        mov    eax,medio
        inc    eax
        mov    primero,eax
        jmp    L4
;
; else if( EDX > valBusqueda(EDI) )
L2:   cmp    edx,edi              ; opcional
        jle    L3
;
        ultimo = medio - 1;

```

```

    mov    eax,medio
    dec    eax
    mov    ultimo,eax
    jmp    L4
; en cualquier otro caso, devuelve valor medio
L3:   mov    eax,medio           ; se encontró el valor
      jmp    L9               ; devuelve (medio)
L4:   jmp    L1               ; continúa el ciclo
L5:   mov    eax,-1            ; falló la búsqueda
L9:   ret
BusquedaBinaria ENDP

```

Programa de prueba

Para demostrar las funciones de ordenamiento de burbuja y búsqueda binaria que presentamos en este capítulo, vamos a escribir un programa corto de prueba que realiza los siguientes pasos, en secuencia:

- Llena un arreglo con enteros aleatorios.
- Muestra el arreglo en pantalla.
- Ordena el arreglo usando el ordenamiento de burbuja.
- Vuelve a mostrar el arreglo.
- Pide al usuario que introduzca un entero.
- Realiza una búsqueda binaria (en el arreglo) del entero que introdujo el usuario.
- Muestra los resultados de la búsqueda binaria.

Los procedimientos individuales se han colocado en archivos de código fuente separados, para facilitar su localización y la edición del código fuente. La tabla 9-5 presenta cada uno de los módulos y su contenido. La mayoría de los programas escritos por profesionales se dividen en módulos de código separados.

Tabla 9-5 Módulos en el programa de Ordenamiento de burbuja y Búsqueda binaria.

Módulo	Contenido
B_main.asm	Módulo principal: contiene los procedimientos main, MostrarResultados y PedirValBusqueda . Contiene el punto de entrada del programa y administra la secuencia general de las tareas
OrdBurbuja.asm	Procedimiento OrdenBurbuja : realiza un ordenamiento de burbuja en un arreglo de enteros de 32 bits con signo
BusquedaB.asm	Procedimiento BusquedaBinaria : realiza una búsqueda binaria en un arreglo de enteros de 32 bits con signo
LlenarArreg.asm	Procedimiento LlenarArreglo : llena un arreglo de enteros de 32 bits con signo con un rango de valores aleatorios
ImprArreg.asm	Procedimiento ImprimirArreglo : escribe el contenido de un arreglo de enteros de 32 bits con signo a la salida estándar

Los procedimientos en todos los módulos excepto *B_main* están escritos de tal forma que puedan utilizarse con facilidad en otros programas, sin necesidad de hacer modificaciones. Esto es muy conveniente, ya que podríamos ahorrar tiempo en el futuro al reutilizar el código existente. El mismo método se utiliza en las bibliotecas de enlace Irvine32 e Irvine16. A continuación se muestra un archivo de inclusión (*BusquedaB.inc*), que contiene prototipos de los procedimientos que se llaman desde el módulo principal (main):

```

; BusquedaB.inc - prototipos de los procedimientos usados en
; el programa OrdenBurbuja y BusquedaBinaria.

; Busca un entero en un arreglo de enteros de 32 bits
; con signo.
BusquedaBinaria PROTO,

```

```

pArreglo:PTR DWORD,           ; apuntador a arreglo
Cuenta:DWORD,                 ; tamaño del arreglo
valBusqueda:DWORD             ; valor a buscar

; Llena un arreglo con enteros aleatorios de 32 bits con signo
LlenarArreglo PROTO,
    pArreglo:PTR DWORD,       ; apuntador a un arreglo
    Cuenta:DWORD,             ; número de elementos
    RangoInferior:SDWORD,     ; rango inferior
    RangoSuperior:SDWORD      ; rango superior

; Escribe un arreglo de enteros de 32 bits con signo a la salida estándar
ImprimirArreglo PROTO,
    pArreglo:PTR DWORD,       ; apuntador a un arreglo
    Cuenta:DWORD

; Ordena el arreglo en sentido ascendente
OrdenBurbuja PROTO,
    pArreglo:PTR DWORD,       ; apuntador a un arreglo
    Cuenta:DWORD

```

A continuación se muestra un listado de *B_main.asm*, el módulo principal:

```

TITLE Ordenamiento de burbuja y Búsqueda binaria    (B_main.asm)

; Usa el ordenamiento de burbuja para ordenar un arreglo de
; enteros con signo, y realiza una búsqueda binaria.
; Módulo principal (Main), llama a BusquedaB.asm, OrdenBurb.asm, LlenarArreg.asm
; e ImprArreg.asm

INCLUDE Irvine32.inc
INCLUDE BusquedaB.inc          ; prototipos de los procedimientos

VALINF = -5000                  ; valor mínimo
VALSUP = +5000                  ; valor máximo
TAM_ARREGLO = 50                ; tamaño del arreglo

.data
arreglo DWORD TAM_ARREGLO DUP(?)

.code
main PROC
    call Randomize

    ; Llena un arreglo con enteros aleatorios con signo
    INVOKE LlenarArreglo, ADDR arreglo, TAM_ARREGLO, VALINF, VALSUP

    ; Muestra el arreglo
    INVOKE ImprimirArreglo, ADDR arreglo, TAM_ARREGLO
    call WaitMsg

    ; Realiza un ordenamiento de burbuja y vuelve a mostrar el arreglo
    INVOKE OrdenBurbuja, ADDR arreglo, TAM_ARREGLO
    INVOKE ImprimirArreglo, ADDR arreglo, TAM_ARREGLO

    ; Demuestra una búsqueda binaria
    call PedirValBusqueda        ; se devuelve en EAX
    INVOKE BusquedaBinaria,
        ADDR arreglo, TAM_ARREGLO, eax
    call MostrarResultados

exit
main ENDP

```

```

;-----.
PedirValBusqueda PROC
;
; Pide al usuario un entero con signo.
; Recibe: nada
; Devuelve: EAX = valor introducido por el usuario
;-----.
.data
indicador BYTE "Escriba un entero decimal con signo "
    BYTE "en un rango de -5000 a +5000"
    BYTE "a buscar en el arreglo: ",0

.code
    call Crlf
    mov edx,OFFSET indicador
    call WriteString
    call ReadInt
    ret
PedirValBusqueda ENDP
;-----.
MostrarResultados PROC
;
; Muestra el valor resultante de la búsqueda binaria.
; Recibe: EAX = número de posición a mostrar
; Devuelve: nada
;-----.
.data
msj1 BYTE "No se encontro el valor.",0
msj2 BYTE "Se encontro el valor en la posicion ",0
.code
.IF eax == -1
    mov edx,OFFSET msj1
    call WriteString
.ELSE
    mov edx,OFFSET msj2
    call WriteString
    call WriteDec
.ENDIF
    call Crlf
    call Crlf
    ret
MostrarResultados ENDP
END main

```

ImprimirArreglo A continuación se muestra un listado del módulo que contiene el procedimiento ImprimirArreglo:

```

TITLE Procedimiento ImprimirArreglo          (ImprArreg.asm)

INCLUDE Irvine32.inc

.code
;-----.
ImprimirArreglo PROC USES eax ecx edx esi,
    pArreglo:PTR DWORD,,; apuntador a un arreglo
    Cuenta:DWORD; número de elementos
;
; Escribe un arreglo de enteros decimales de 32 bits con signo

```

```

; salida estándar, separada por comas
; Recibe: apuntador a un arreglo, tamaño del arreglo
; Devuelve: nada
;-----
.data
coma BYTE ", ",0
.code
    mov    esi,pArreglo
    mov    ecx,Cuenta
    cld
    ; dirección = avance
L1: lodsd
    call  WriteInt
    mov   edx,OFFSET coma
    call  Writestring
    ; muestra una coma
    loop L1
    call  Crlf
    ret
ImprimirArreglo ENDP
END

```

LlenarArreglo A continuación se muestra un listado del módulo que contiene el procedimiento LlenarArreglo:

```

TITLE Procedimiento LlenarArreglo          (LlenarArreg.asm)
INCLUDE Irvine32.inc

.code
;-----
LlenarArreglo PROC USES eax edi ecx edx,
    pArreglo:PTR DWORD,           ; apuntador a un arreglo
    Cuenta:DWORD,                ; número de elementos
    RangoInferior:SDWORD,         ; rango inferior
    RangoSuperior:SDWORD          ; rango superior
;
; Llena un arreglo con una secuencia aleatoria de enteros de
; 32 bits con signo, entre RangoInferior y (RangoSuperior - 1).
; Devuelve: nada
;-----
    mov    edi,pArreglo          ; EDI apunta al arreglo
    mov    ecx,Cuenta            ; contador del ciclo
    mov    edx,RangoSuperior
    sub    edx,RangoInferior     ; EDX = rango absoluto (0..n)

L1: mov    eax,edx              ; obtiene rango absoluto
    call  RandomRange
    add    eax,RangoInferior     ; desvía el resultado
    stosd                           ; guarda EAX en [edi]
    loop L1
    ret
LlenarArreglo ENDP
END

```

9.5.3 Repaso de sección

- Si un arreglo ya estuviera en orden secuencial, ¿cuántas veces se ejecutaría el ciclo externo del procedimiento **OrdenBurbuja** de la sección 9.5.1?

2. En el procedimiento **OrdenBurbuja**, ¿cuántas veces se ejecuta el ciclo interno en la primera ejecución a través del arreglo?
3. En el procedimiento **OrdenBurbuja**, ¿el ciclo interno se ejecuta siempre el mismo número de veces?
4. Si se descubriera (a través de pruebas) que un arreglo de 500 enteros puede ordenarse en 0.5 segundos, ¿cuántos segundos se requerirían para ordenar un arreglo de 5000 enteros mediante el método de burbuja?
5. ¿Cuál es el número máximo de comparaciones que requiere el algoritmo de búsqueda binaria, si un arreglo contiene 127 elementos?
6. Dado un arreglo de n elementos, ¿cuál es el número máximo de comparaciones requeridas por el algoritmo de búsqueda binaria?
7. *Reto:* en el procedimiento **BusquedaBinaria** (sección 9.5.2), ¿por qué podría eliminarse la instrucción en la etiqueta L2 sin afectar los resultados?
8. *Reto:* en el procedimiento **BusquedaBinaria**, ¿cómo podría eliminarse la instrucción en la etiqueta L4?

9.6 Resumen del capítulo

Las instrucciones primitivas de cadenas son inusuales, ya que no requieren operandos tipo registro y están optimizadas para un acceso a memoria de alta velocidad. Éstas son:

- MOVS: mover datos de cadena.
- CMPS: comparar cadenas.
- SCAS: explorar cadena.
- STOS: almacenar datos de cadena.
- LODS: cargar el acumulador desde una cadena.

Cada instrucción de primitiva de cadena tiene un sufijo de B, W o D al manipular bytes, palabras y dobles palabras, respectivamente.

El prefijo de repetición REP repite una instrucción de primitiva de cadena con el incremento o decremento automático de los registros índice. Por ejemplo, cuando se utiliza REPNE con SCASB, explora los bytes de la memoria hasta que un valor en la memoria a la que apunta EDI coincide con el contenido del registro AL. La bandera Dirección determina si el registro índice se incrementa o se decrementa durante cada iteración de una instrucción primitiva de cadena.

Las cadenas y los arreglos son prácticamente lo mismo. Antes, una cadena consistía de un arreglo de valores ASCII de un solo byte, pero ahora las cadenas pueden contener caracteres Unicode de 16 bits. La única diferencia importante entre una cadena y un arreglo es que, por lo general, una cadena termina con un byte nulo (que contiene cero).

La manipulación de arreglos ocupa muchos recursos computacionales, ya que, por lo general, involucra un algoritmo de iteraciones. La mayoría de los programas invierten de un 80 a un 90 por ciento de su tiempo de ejecución en una pequeña fracción de su código. Como resultado, podemos optimizar nuestro software mediante una reducción del número y la complejidad de las instrucciones dentro de los ciclos. El lenguaje ensamblador es una excelente herramienta para la optimización de código, ya que se pueden controlar todos los detalles. Por ejemplo, podríamos optimizar un bloque de código mediante la sustitución de los registros por variables de memoria. O podríamos usar una de las instrucciones de procesamiento de cadenas que vimos en este capítulo, en vez de instrucciones MOV y CMP.

En este capítulo se introdujeron varios procedimientos útiles para el procesamiento de cadenas: El procedimiento **Str_copy** copia una cadena a otra. **Str_length** devuelve la longitud de una cadena. **Str_compare** compara dos cadenas. **Str_trim** elimina un carácter seleccionado del final de una cadena. **Str_ucase** convierte una cadena a mayúsculas.

Los operandos base-índice ayudan a manipular los arreglos bidimensionales (tablas). Podemos asignar a un registro base la dirección de la fila de una tabla, y apuntar un registro índice al desplazamiento de una columna dentro de la fila seleccionada. En modo de 32 bits, puede utilizarse cualquier registro de propósito general de 32 bits como registro base o índice. En modo de 16 bits, los registros base deben ser BX y BP; los

registros índice deben ser SI y DI. Los operandos base-índice-desplazamiento son similares a los operandos base-índice, excepto que también incluyen el nombre del arreglo:

[ebx + esi]	; base-índice
arreglo[ebx + esi]	; base-índice-desplazamiento

En este capítulo presentamos las implementaciones en lenguaje ensamblador de un ordenamiento de burbuja y una búsqueda binaria. Un ordenamiento de burbuja ordena los elementos de un arreglo en forma ascendente o descendente. Es efectivo para los arreglos que tienen sólo unos cuantos cientos de elementos, pero ineficiente para los arreglos más grandes. Una búsqueda binaria permite buscar con rapidez un solo valor en un arreglo ordenado. Es fácil de implementar en lenguaje ensamblador.

9.7 Ejercicios de programación

Los siguientes ejercicios pueden realizarse en modo de 32 bits o de 16 bits. Cada procedimiento de manejo de cadenas supone el uso de cadenas con terminación nula. Aún cuando no se solicita en forma explícita, debe escribir un programa corto de control para la solución de cada ejercicio, de manera que pueda probar su nuevo procedimiento.

1. Procedimiento Str_copy mejorado

El procedimiento **Str_copy** que se muestra en este capítulo no limita el número de caracteres a copiar. Cree una nueva versión (llamada **Str_copyN**) que reciba un parámetro de entrada adicional, para indicar el número máximo de caracteres a copiar.

2. Procedimiento Str_concat

Escriba un procedimiento llamado **Str_concat** para concatenar una cadena de origen al final de una cadena de destino. Debe existir suficiente espacio en la cadena de destino como para dar cabida a los nuevos caracteres. Pase apuntadores a las cadenas de origen y de destino. He aquí una llamada de ejemplo:

```
.data
cadDestino BYTE "ABCDE",10 DUP(0)
cadOrigen BYTE "FGH",0
.code
INVOKE Str_concat, ADDR cadDestino, ADDR cadOrigen
```

3. Procedimiento Str_remove

Escriba un procedimiento llamado **Str_remove** para eliminar *n* caracteres de una cadena. Pase un apuntador a la posición en la cadena en donde se van a eliminar los caracteres. Pase un entero que especifique el número de caracteres a eliminar. Por ejemplo, el siguiente código muestra cómo eliminar “xxxx” de **destino**:

```
.data
destino BYTE "abcxxxxdefghijklmop",0
.code
INVOKE Str_remove, ADDR [destino+3], 4
```

4. Procedimiento Str_find

Escriba un procedimiento llamado **Str_find** para buscar la primera ocurrencia que coincida de una cadena de origen dentro de una cadena de destino, y devuelva la posición en la que ocurrió la coincidencia. Los parámetros de entrada deben ser un apuntador a la cadena de origen y un apuntador a la cadena de destino. Si se encuentra una coincidencia, el procedimiento debe activar la bandera Cero y EAX debe apuntar a la posición coincidente en la cadena de destino. En caso contrario, la bandera Cero debe borrarse y EAX quedar indefinido. Por ejemplo, el siguiente código busca “ABC” y regresa con EAX apuntando a la “A” en la cadena de destino:

```
.data
destino BYTE "123ABC342432",0
origen BYTE "ABC",0
```

```
pos    DWORD ?
.code
INVOKE Str_Find, ADDR origen, ADDR destino
jnz  noSeEncontro
mov  pos,eax                      ; guarda el valor de la posición
```

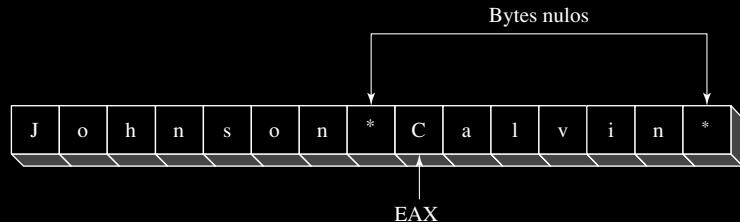
5. Procedimiento Str nextword

Escriba un procedimiento llamado **Str_nextword** que explore una cadena en busca de la primera ocurrencia de cierto carácter delimitador y lo sustituya con un byte nulo. Hay dos parámetros de entrada: un apuntador a la cadena y el carácter delimitador. Después de la llamada, si se encuentra el delimitador, debe activarse la bandera Cero y EAX debe contener el desplazamiento del siguiente carácter que está después del delimitador. En caso contrario, la bandera Cero debe borrarse y EAX quedar indefinido. El siguiente código de ejemplo pasa la dirección de **destino** y una coma como delimitador:

```
.data  
destino BYTE "Johnson,Calvin",0  
.code  
INVOKE Str_nextword, ADDR destino, ','  
jnz NoSeEncontro
```

En la figura 9-8, después de llamar a `Str_nextword`, EAX apunta al carácter que sigue después de la posición en la que se encontró la coma (y se sustituyó).

FIGURA 9–8 Ejemplo de Str nextword.



6. Construcción de una tabla de frecuencias

Escriba un procedimiento llamado **Obtener_frecuencias** para construir una tabla de frecuencias de caracteres. La entrada al procedimiento debe ser un apuntador a una cadena y un apuntador a un arreglo de 256 dobles palabras, todas inicializadas con cero. Cada posición del arreglo debe indexarse mediante su código ASCII correspondiente. Cuando el procedimiento regrese, cada entrada en el arreglo deberá contener una cuenta de las veces que ocurrió el carácter correspondiente en la cadena. Por ejemplo,

```
.data  
destino BYTE "AAEBDCFBBC",0  
tablaFrec DWORD 256 DUP(0)  
.code  
INVOKE Obtener_frecuencias, ADDR destino, ADDR tablaFrec
```

FIGURA 9-9 Ejemplo de tabla de frecuencias de caracteres.

Cadena de destino:	A	A	E	B	D	C	F	B	B	C	0
Código ASCII:	41	41	45	42	44	43	46	42	42	43	0
Tabla de frecuencias:	2	3	2	1	1	1	0	0	0	0	0
Índice:	41	42	43	44	45	46	47	48	49	4A	4B

La figura 9-9 muestra una imagen de la cadena y las entradas 41 (hexadecimal) a 4B en la tabla de frecuencias. La posición 41 contiene el valor 2, ya que la letra A (código ASCII 41h) ocurrió dos veces en la cadena. Se muestran cuentas similares para otros caracteres. Las tablas de frecuencia son útiles en la compresión de datos y otras aplicaciones en las que se involucra el procesamiento de caracteres. Por ejemplo, el algoritmo de codificación de Huffman almacena los caracteres que ocurren con más frecuencia, en menos bits que otros caracteres que ocurren con menos frecuencia.

7. Criba de Eratóstenes

La *Criba de Eratóstenes*, inventada por el matemático griego del mismo nombre, proporciona una rápida forma de encontrar todos los números primos dentro de un rango dado. El algoritmo implica la creación de un arreglo de bytes, en los que las posiciones se “marcan” insertando 1s de la siguiente manera: empezando con la posición 2 (que es un número primo), inserte un 1 en cada posición del arreglo que sea un múltiplo de 2. Después haga lo mismo para los múltiplos de 3, el siguiente número primo. Encuentre el siguiente número primo después de 3, que es 5, y marque todas las posiciones que sean múltiplos de 5. Proceda de esta forma hasta que se hayan encontrado todos los números primos. El resto de las posiciones del arreglo que están sin marca indican qué números son primos. Para este programa, cree un arreglo de 65,000 elementos y muestre todos los números primos entre 2 y 65,000. Declare el arreglo en un segmento de datos sin inicializar (vea la sección 3.4.11) y utilice STOSB para llenarlo con ceros. En modo de 32 bits, su arreglo puede ser mucho mayor.

8. Ordenamiento de burbuja

Agregue una variable al procedimiento **OrdenBurbuja** en la sección 9.5.1 que se active en 1 cada vez que se intercambie un par de valores dentro del ciclo interno. Use esta variable para salir del ordenamiento antes de que termine en forma normal, si descubre que no se realizaron intercambios durante una ejecución completa a través del arreglo. (Esta variable se conoce comúnmente como *bandera de intercambio*).

9. Búsqueda binaria

Vuelva a escribir el procedimiento de búsqueda binaria que se muestra en este capítulo, usando registros para medio, primero y último. Agregue comentarios para clarificar el uso de los registros.

10. Matriz de letras

Cree un procedimiento que genere una matriz de cuatro por cuatro, de letras mayúsculas elegidas al azar. Cuando elija las letras, debe haber por lo menos una probabilidad del 50% de que la letra elegida sea una vocal. Escriba un programa de prueba con un ciclo que llame a su procedimiento cinco veces, y que muestre cada matriz en la ventana de consola. A continuación se muestra un resultado de ejemplo para las primeras tres matrices:

```
D W A L  
S I V W  
U I O L  
L A I I  
  
K X S V  
N U U O  
O R Q O  
A U U T  
  
P O A Z  
A E A U  
G K A E  
I A G D
```

11. Matriz de letras/Conjuntos con vocales

Use la matriz de letras generada en el ejercicio de programación anterior como punto de inicio para este programa. Genere una matriz de letras aleatoria de cuatro por cuatro, en la que cada letra tenga una probabilidad del 50% de ser una vocal. Recorra cada fila, columna y diagonal de la matriz, generando conjuntos de letras.

Muestre sólo conjuntos de cuatro letras que contengan exactamente dos vocales. Por ejemplo, suponga que se generó la siguiente matriz:

P	O	A	Z
A	E	A	U
G	K	A	E
I	A	G	D

Entonces los conjuntos de cuatro letras que debe mostrar el programa son POAZ, GKAE, IAGD, PAGI, ZUED, PEAD y ZAKI. El orden de las letras dentro de cada conjunto no es importante.

12. Cálculo de la suma de la fila de un arreglo

Escriba un procedimiento llamado `calc_suma_fila` que calcule la suma de una sola columna en cualquier arreglo bidimensional de bytes, palabras o dobles palabras. El procedimiento debe tener los siguientes parámetros de pila: desplazamiento del arreglo, tamaño de la fila, tipo del arreglo, índice de la fila. Debe regresar la suma en EAX. Use parámetros de fila explícitos, no las instrucciones INVOKE ni PROC extendida. Use el direccionamiento base-índice con factores de escala (vea la sección 4.4.3). Escriba un programa para probar su procedimiento con arreglos de bytes, palabras y dobles palabras. Pida al usuario el índice de la fila y muestre la suma de la fila seleccionada.

Nota final

1. Donald, Knuth, *The Art of Computer Programming*, Volumen I: *Fundamental Algorithms*, Addison-Wesley, 1997.

10

ESTRUCTURAS Y MACROS

10.1 Estructuras

- 10.1.1 Definición de estructuras
- 10.1.2 Declaración de variables de estructura
- 10.1.3 Referencias a variables de estructura
- 10.1.4 Ejemplo: mostrar la hora del sistema
- 10.1.5 Estructuras que contienen estructuras
- 10.1.6 Ejemplo: paso del borracho
- 10.1.7 Declaración y uso de uniones
- 10.1.8 Repaso de sección

10.2 Macros

- 10.2.1 Generalidades
- 10.2.2 Definición de macros
- 10.2.3 Invocación de macros
- 10.2.4 Características adicionales de los macros
- 10.2.5 Uso de la biblioteca de macros del libro
- 10.2.6 Programa de ejemplo: envolturas
- 10.2.7 Repaso de sección

10.3 Directivas de ensamblado condicional

- 10.3.1 Comprobación de argumentos faltantes

- 10.3.2 Inicializadores de argumentos predeterminados
- 10.3.3 Expresiones booleanas
- 10.3.4 Directivas IF, ELSE y ENDIF
- 10.3.5 Las directivas IFIDN e IFIDNI
- 10.3.6 Ejemplo: suma de la fila de una matriz
- 10.3.7 Operadores especiales
- 10.3.8 Macrofunciones
- 10.3.9 Repaso de sección

10.4 Definición de bloques de repetición

- 10.4.1 Directiva WHILE
- 10.4.2 Directiva REPEAT
- 10.4.3 Directiva FOR
- 10.4.4 Directiva FORC
- 10.4.5 Ejemplo: lista enlazada
- 10.4.6 Repaso de sección

10.5 Resumen del capítulo

10.6 Ejercicios de programación

10.1 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 varios identificadores, pero éstos traslanan la misma área en memoria. En la sección 10.1.7 hablaremos sobre las uniones.

Las estructuras proporcionan una forma fácil de agrupar datos y pasarlo de un procedimiento a otro. Suponga que los parámetros de entrada para un procedimiento consisten en 20 unidades distintas de datos relacionados con una unidad de 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 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 STRUCT
    X WORD ?
    Y WORD ?                                ; desplazamiento 00
                                                ; 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.

10.1.1 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 que para las variables ordinarias. Las estructuras pueden contener casi cualquier número de campos:

```
nombre STRUCT
    declaraciones de los campos
nombre ENDS
```

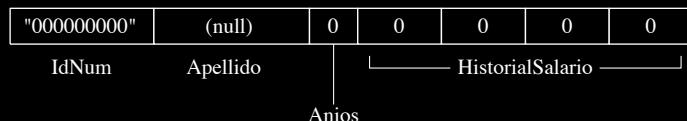
Inicializadores de campo Cuando los campos de una estructura tienen inicializadores, los valores se asignan cuando se crean las variables de la estructura. Podemos utilizar varios tipos de inicializadores de campos:

- **Indefinido:** el operador ? deja el contenido de los campos 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(?)
    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 tabla 10-1

presenta las alineaciones que usan los compiladores de 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 Empleado, usando ALIGN para colocar a **Anios** en un límite WORD y a **HistorialSalario** en un límite DWORD. Los tamaños de los campos aparecen como comentarios.

```
Empleado STRUCT
    numID     BYTE "00000000"          ; 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 0,0,0,0   ; 16
Empleado ENDS                      ; 60 en total
```

Tabla 10-1 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

10.1.2 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 llenan 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, utilice 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 Use 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 límites 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 <>
```

10.1.3 Referencias a variables de estructura

Las referencias a las variables de estructura y los nombres de estructura pueden hacerse utilizando los operadores TYPE y SIZEOF. Por ejemplo, vamos a regresar a la estructura **Empleado** que vimos antes:

```
Empleado STRUCT
    NumId     BYTE "00000000"           ; 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 0,0,0,0   ; 16
Empleado 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

El operador TYPE (sección 4.3) devuelve el número de bytes utilizados por el tipo de almacenamiento del identificador (BYTE, WORD, DWORD, etc.). El operador LENGTHOF devuelve la cuenta del número de elementos en un arreglo. El operador SIZEOF multiplica LENGTHOF por TYPE.

Referencias a los miembros

Las referencias a los miembros de estructuras con nombre requieren una variable de estructura como clasificador. 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],30000    ; segundo salario
```

Uso del operador OFFSET Puede 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. Suponga 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 del 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

```

Rendimiento de los miembros alineados de una estructura

Ya hemos visto que el procesador puede acceder con más eficiencia a los miembros de una estructura que estén alineados en forma apropiada. ¿Cuánto impacto tienen los campos desalineados en cuanto al rendimiento? Vamos a realizar una prueba simple, usando las dos versiones de la estructura Empleado que presentamos en este capítulo. Vamos a cambiar el nombre de la primera versión para poder utilizar ambas estructuras en el mismo programa:

```

EmpleadoMalo STRUCT
    NumId     BYTE "00000000"
    Apellido  BYTE 30 DUP(0)
    Anios     WORD 0
    HistorialSalario DWORD 0,0,0,0
EmpleadoMalo ENDS

Empleado STRUCT
    NumId     BYTE "00000000"
    Apellido  BYTE 30 DUP(0)
    ALIGN     WORD
    Anios     WORD 0
    ALIGN     DWORD
    HistorialSalario DWORD 0,0,0,0
Empleado ENDS

```

El siguiente código obtiene la hora del sistema, ejecuta un ciclo que accede a los campos de la estructura y calcula el tiempo transcurrido. La variable emp puede declararse como un objeto Empleado o Empleado-Malo.

```

.data
ALIGN DWORD
tiempoInicial DWORD ?          ; alinea tiempoInicial
emp Empleado <>                ; o: emp EmpleadoMalo <>
.code
    call GetMSeconds            ; obtiene tiempo inicial
    mov tiempoInicial,eax
    mov ecx,0FFFFFFFFFFh        ; contador de ciclos
L1:  mov emp.Anios,5
    mov emp.HistorialSalario,35000
    loop L1
    call GetMSeconds            ; obtiene tiempo inicial
    sub eax,tiempoInicial
    call WriteDec                ; muestra el tiempo transcurrido

```

En nuestro programa de prueba simple (*Struct1.asm*), el tiempo de ejecución usando la estructura Empleado alineada en forma correcta fue de 6141 milisegundos. El tiempo de ejecución utilizando la estructura

EmpleadoMalo fue de 6203 milisegundos. La diferencia de tiempos fue pequeña (62 milisegundos), tal vez debido a que la caché de memoria interna del procesador minimizó los problemas de alineación.

10.1.4 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: COORD y SYSTEMTIME:

```
COORD STRUCT
    X WORD ?
    Y WORD ?
COORD ENDS

SYSTEMTIME STRUCT
    wAnio WORD ?
    wMes WORD ?
    wDiaDeLaSemana WORD ?
    wDia WORD ?
    wHora WORD ?
    wMinuto WORD ?
    wSegundo WORD ?
    wMilisegundos WORD ?
SYSTEMTIME ENDS
```

Ambas estructuras están definidas en *SmallWin.inc*, un archivo que se encuentra en el directorio INCLUDE y al que hace referencia *Irvine32.inc*. Para obtener la hora del sistema (ajustada para su zona horaria local), llame a la función **GetLocalTime** de MS Windows y páselle la dirección de una 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
```

El archivo *SmallWin.inc*, creado por el autor, contiene definiciones de estructuras y prototipos de funciones adaptados de los archivos de encabezado de Microsoft Windows para los programadores de C y C++. Representa un pequeño subconjunto de las posibles funciones que pueden llamar los programas de aplicación.

Cuando un programa Win32 produce resultados en la 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
```

La constante STD_OUTPUT_HANDLE está definida en *SmallWin.inc*.

Para establecer la posición del cursor, llame a la función **SetConsoleCursorPosition** de MS Windows, pasándole el manejador de salida de consola y una variable de estructura COORD que contenga los 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 lo 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
    mov edx,OFFSET cadDosPuntos      ; ":"
    call WriteString
    movzx eax,horaSys.wMinute         ; minutos
    call WriteDec
    call WriteString
    movzx eax,horaSys.wSecond         ; segundos
    call WriteDec
    call CrLf
    call WaitMsg                      ; "Presione una tecla para continuar..."
    exit
main ENDP
END main
```

Este programa utiliza las siguientes definiciones de *SmallWin.inc* (*Irvine32.inc* las incluye de manera automática):

```
STD_OUTPUT_HANDLE EQU -11
SYSTEMTIME STRUCT ...
COORD STRUCT ...
GetStdHandle PROTO,
nStdHandle:DWORD
GetLocalTime PROTO,
lpSystemTime:PTR SYSTEMTIME
SetConsoleCursorPosition PROTO,
nStdHandle:DWORD,
coords:COORD
```

A continuación se muestran los resultados de ejemplo del programa, que se obtuvieron a las 12:16 p.m.:

12:16:35
Presione una tecla para continuar...

10.1.5 Estructuras que contienen estructuras

Las estructuras pueden contener instancias de otras estructuras. Por ejemplo, un **Rectangulo** 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.UpperLeft.X, 10
```

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
```

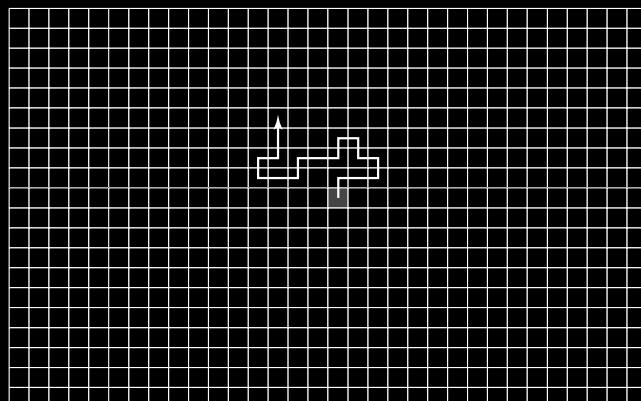
El operador OFFSET puede regresar apuntadores a campos individuales de una estructura, incluyendo los campos anidados:

```
mov edi,OFFSET rect2.InfDer
mov (COORD PTR [edi]).X, 50
mov edi,OFFSET rect2.InfDer.X
mov WORD PTR [edi], 50
```

10.1.6 Ejemplo: paso del borracho

A menudo, los libros de texto de programación contienen una versión del ejercicio “Paso del borracho”, en donde el programa simula la ruta que toma un profesor no muy sobrio, en su camino hacia el salón de clases. Mediante el uso de un generador de números aleatorios, podemos elegir una dirección para cada paso que dé el profesor. Por lo general, hay que comprobar que la persona no se desvíe y caiga a un lago del campus, pero no nos preocuparemos por eso. Suponga que el profesor empieza en el centro de una cuadrícula imaginaria, en donde cada cuadro representa un paso en dirección hacia el norte, sur, este u oeste. La persona sigue un camino aleatorio en la cuadrícula (figura 10-1).

FIGURA 10-1 Paso del borracho, ruta de ejemplo.



Nuestro programa utilizará una estructura COORD para llevar el rastro de cada paso en la ruta que toma el profesor. Los pasos se almacenan en un arreglo de objetos COORD:

```
MaxPasos = 50
PasoBorracho STRUCT
    ruta COORD MaxPasos DUP(<0,0>)
    rutasUsadas WORD 0
PasoBorracho ENDS
```

MaxPasos es una constante que determina el número total de pasos que da el profesor en la simulación. El campo **rutasUsadas** indica, al terminar el ciclo del programa, cuántos pasos dio el profesor. A medida que el profesor da cada paso, su posición se almacena en un objeto COORD y se inserta en la siguiente posición disponible en el arreglo **ruta**. El programa muestra las coordenadas en la pantalla. He aquí el listado completo del programa:

```
TITLE Paso del borracho                               (Paso.asm)

; Programa del Paso del borracho. El profesor empieza en las
; coordenadas 50,50 y deambula por el área inmediata.

INCLUDE Irvine32.inc
MaxPasos = 50
InicioX = 25
InicioY = 25

PasoBorracho STRUCT
    ruta COORD MaxPasos DUP(<0,0>)
    rutasUsadas WORD 0
PasoBorracho ENDS

MostrarPosicion PROTO xActual:WORD, yActual:WORD

.data
unPaso PasoBorracho <>

.code
main PROC
    mov    esi,OFFSET unPaso
    call   DarPasoBorracho
    exit
main ENDP

;-----
DarPasoBorracho PROC
LOCAL xActual:WORD, yActual:WORD
;
; Da un paso en direcciones aleatorias (norte, sur, este,
; oeste).
; Recibe:   ESI apunta a una estructura PasoBorracho
; Devuelve: la estructura se inicializa con valores aleatorios
;-----
    pushad
; Usa el operador OFFSET para obtener la dirección de la
; ruta, el arreglo de objetos COORD, y lo copia a EDI.
    mov    edi,esi
    add    edi,OFFSET PasoBorracho.ruta
    mov    ecx,MaxPasos           ; contador del ciclo
    mov    xActual,InicioX        ; ubicación X actual
```

```

        mov    yActual,InicioY           ; ubicación Y actual

OtraVez:
    ; Inserta la ubicación actual en el arreglo.
    mov    ax,xActual
    mov    (COORD PTR [edi]).X,ax
    mov    ax,yActual
    mov    (COORD PTR [edi]).Y,ax

    INVOKE MostrarPosicion, xActual, yActual

    mov    eax,4                   ; elige una ubicación (0-3)
    call   RandomRange

    .IF eax == 0                 ; Norte
        dec    yActual
    .ELSEIF eax == 1              ; Sur
        inc    yActual
    .ELSEIF eax == 2              ; Oeste
        dec    xActual
    .ELSE                          ; Este (EAX = 3)
        inc    xActual
    .ENDIF

    add    edi,TYPE COORD         ; apunta a la siguiente COORD
    loop   OtraVez

Meta:
    mov    (PasoBorracho PTR [esi]).rutasUsadas, MaxPasos
    popad
    ret
DarPasoBorracho ENDP

;-----
MostrarPosicion PROC xActual:WORD, yActual:WORD
; Muestra las posiciones X y Y actuales.
;-----
.data
cadComa BYTE ",",0
.code
    pushad
    movzx eax,xActual           ; posición X actual
    call   WriteDec
    mov    edx,OFFSET cadComa    ; cadena ","
    call   WriteString
    movzx eax,yActual           ; posición Y actual
    call   WriteDec
    call   Crlf
    popad
    ret
MostrarPosicion ENDP
END main

```

Procedimiento DarPasoBorracho Veamos de cerca el procedimiento **DarPasoBorracho**. Este procedimiento recibe un apuntador (ESI) a una estructura **PasoBorracho**. Mediante el uso del operador OFFSET, calcula el desplazamiento del arreglo **ruta** y lo copia a EDI:

```

mov    edi,esi
add    edi,OFFSET PasoBorracho.ruta

```

Las posiciones iniciales X y Y (InicioX e InicioY) del profesor se establecen en 25, al centro de una cuadrícula imaginaria de 50 por 50. El contador del ciclo se inicializa:

```
mov ecx,MaxPasos           ; contador del ciclo
mov xActual,InicioX        ; ubicación X actual
mov yActual,InicioY         ; ubicación Y actual
```

Al principio del ciclo, se inicializan las primeras dos entradas en el arreglo **ruta**:

OtraVez:

```
; Inserta la ubicación actual en el arreglo.
mov ax,xActual
mov (COORD PTR [edi]).X,ax
mov ax,yActual
mov (COORD PTR [edi]).Y,ax
```

Al final del recorrido, se inserta un contador en el campo **rutasUsadas**, indicando cuántos pasos se dieron:

Meta:

```
mov (PasoBorracho PTR [esi]).rutasUsadas, MaxPasos
```

En la versión actual del programa, **rutasUsadas** es igual a **MaxPasos**, pero eso podría cambiar si comprobamos los peligros como los lagos y edificios. Entonces, el ciclo terminaría antes de llegar a **MaxPasos**.

10.1.7 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* empiezan 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-unión
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 distintos para los mismos datos, e inicializa todos los campos con cero:

```
Entero UNION
    D DWORD 0
    W WORD 0
    B BYTE 0
Entero ENDS
```

Sea consistente Si se utilizan los inicializadores, deben tener valores consistentes. Suponga 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 los 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 puede 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 variables 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**. Observe 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 estructuras predefinidas. El campo **EventType** indica el tipo de registro que aparece en la unión. Cada estructura tiene una distribución y tamaño distintos, pero sólo se utiliza una a la vez:

```
INPUT_RECORD STRUCT
    EventType WORD ?
    ALIGN DWORD
    UNION Event
        KEY_EVENT_RECORD <>
        MOUSE_EVENT_RECORD <>
        WINDOW_BUFFER_SIZE_RECORD <>
        MENU_EVENT_RECORD <>
        FOCUS_EVENT_RECORD <>
    ENDS
INPUT_RECORD ENDS
```

A menudo, la API Win32 incluye la palabra RECORD al nombrar las estructuras.¹ 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
```

Encontrará el resto de las definiciones STRUCT de INPUT_RECORD en el archivo SmallWin.inc.

10.1.8 Repaso de sección

1. ¿Cuál es el propósito de la directiva STRUCT?
2. Cree una estructura llamada **MiEstruct** que contenga dos campos: **campo1**, una palabra individual, y **campo2**, un arreglo de 20 dobles palabras. Los valores iniciales de los campos pueden dejarse indefinidos.

La estructura creada en el ejercicio 2 (MiEstruct) se utilizará en los ejercicios del 3 al 11:

3. Declare una variable **MiEstruct** con valores predeterminados.
4. Declare una variable **MiEstruct** que inicialice el primer campo con cero.
5. Declare una variable **MiEstruct** que inicialice el segundo campo con un arreglo que contenga sólo ceros.
6. Declare una variable como un arreglo de 20 objetos **MiEstruct**.
7. Utilice el arreglo **MiEstruct** del ejercicio anterior para mover el **campo1** del primer elemento del arreglo a AX.
8. Utilice el arreglo **MiEstruct** del ejercicio anterior para usar ESI para indexar el tercer elemento del arreglo y mover AX a **campo1**. *Sugerencia:* utilice el operador PTR.
9. ¿Qué valor devuelve la expresión **TYPE MiEstruct**?
10. ¿Qué valor devuelve la expresión **SIZEOF MiEstruct**?
11. Escriba una expresión que devuelva el número de bytes en **campo2** de **MiEstruct**.
12. Suponga que se ha definido la siguiente estructura:

```
FacturaRenta STRUCT
    numFactura BYTE 5 DUP(' ')
    precioDiario WORD ?
    diasRentados WORD ?
FacturaRenta ENDS
```

Indique si cada una de las siguientes declaraciones es válida o no:

- a. **rentas** FacturaRenta <>
- b. FacturaRenta **rentas** <>
- c. marzo FacturaRenta<'12345',10,0>
- d. FacturaRenta <,10,0>
- e. actual FacturaRenta <,15,0>
13. Escriba una instrucción para obtener el campo **wHour** de una estructura SYSTEMTIME.
14. Utilice la siguiente estructura **Triangulo** para declarar una variable de estructura e inicializar sus vértices con (0,0), (5,0) y (7,6):

```
Triangulo STRUCT
    Vertice1 COORD <>
```

```

Vertice2 COORD <>
Vertice3 COORD <>
Triangulo ENDS

```

15. Declare un arreglo de estructuras **Triangulo**. Escriba un ciclo para inicializar el **Vertice1** de cada triángulo con coordenadas aleatorias en el rango (0..10,0..10).

10.2 Macros

10.2.1 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.

Declaración Los macros (procedimientos) se definen de manera directa al principio de un programa de código fuente, o se colocan en un archivo separado y se copian en un programa mediante una directiva INCLUDE. Los macros se expanden durante el paso de *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 de su código 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 le mostraremos cómo pasar argumentos a los macros, para que sean mucho más útiles.

10.2.2 Definición de macros

Un macro se define usando las directivas MACRO y ENDM. La sintaxis es

```

nombrermacro MACRO parámetro-1, parámetro-2...
    lista-instrucciones
ENDM

```

No hay una regla fija en relación con la sangría, pero le recomendamos 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 las directivas 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 los 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 llamada **mColocarCar** recibe un solo parámetro de entrada llamado **car** y lo muestra en la 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
```

10.2.3 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 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 sección anterior definimos al macro **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, mostrado 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
    push eax
    mov  al,al
    call WriteChar
    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 una 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.

Depuración de los programas que contienen macros

Depurar un programa que utilice macros puede ser un reto especial. Después de ensamblar un programa, hay que comprobar su archivo de listado (extensión .LST) para asegurarse de que cada macro se expanda en la forma esperada. Después, se inicia el programa en un depurador (como Visual Studio .NET). Se rastrea el programa en una ventana de desensamblado, usando la opción *mostrar código fuente* si la soporta el depurador. Cada llamada a un macro irá seguida del código generado por el macro. He aquí un ejemplo:

```

mEscribirEn 15,10,"Hola a todos"
    push  edx
    mov   dh,0Ah
    mov   dl,0Fh
    call  _Gotoxy@0 (401551h)
    pop   edx
    push  edx
    mov   edx,offset ??0000 (405004h)
    call  _WriteString@0 (401D64h)
    pop   edx

```

Los nombres de las funciones empiezan con el guion bajo (_), ya que la biblioteca Irvine32 utiliza la convención de llamadas STDCALL. En la sección 8.4.1 podrá consultar los detalles.

10.2.4 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 un macro, aparecen cada vez que ésta se expande. Si desea tener comentarios en el macro que no aparezcan en sus expansiones, debe empezarlos con doble signo de punto y coma (;;):

```
mColocarCar MACRO car:REQ
    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 a una variable llamada **cadena** y la inicializa con un arreglo de caracteres:

```
crearCadena MACRO texto
    .data
    cadena BYTE texto,0
ENDM
```

Suponga que invocamos al 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 nueva 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
  .data
  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 le 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 otros macros, se pasan directamente a sus macros anidados.

Use un enfoque modular al crear macros. Procure que sean cortos y simples, para poder combinarlos en macros más complejos. Esto le ayudará a reducir la cantidad de código duplicado en sus programas.

Ejemplo: mWriteLn El siguiente macro **mWriteLn** escribe una literal de cadena en la consola y le adjunta un fin de línea. Invoca al macro **mWrite** y llama al procedimiento **Crlf**:

```
mWriteLn MACRO texto
    mWrite  texto
    call    Crlf
ENDM
```

El parámetro **texto** se pasa directamente a **mWrite**. Suponga 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 invitado a un macro anidado:

```
1   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
```

10.2.5 Uso de la biblioteca de macros del libro

Los programas de ejemplo que acompañan este libro incluyen una biblioteca de macros pequeña pero útil, la cual podemos habilitar con sólo agregar la siguiente línea a nuestros programas, justo después de la directiva **INCLUDE** que ya tenemos:

```
INCLUDE Macros.inc
```

Algunos de los macros son envolturas alrededor de los procedimientos existentes en las bibliotecas Irvine32 e Irvine16, para facilitar el paso de parámetros. Otros macros proporcionan una nueva funcionalidad. La tabla 10-2 describe cada macro con detalle. Encontrará el código de ejemplo en *PruebaMacros.asm*.

Tabla 10-2 Los macros de la biblioteca Macros.inc.

Nombre del macro	Parámetros	Descripción
mDump	varName, useLabel	Muestra una variable, usando su nombre y atributos predeterminados
mDumpMem	address, itemCount, componentSize	Muestra un rango de memoria
mGotoxy	X, Y	Establece la posición del cursor en el búfer de la ventana de consola
mReadString	varName	Lee una cadena del teclado
mShow	itsName, format	Muestra una variable o registro en varios formatos
mShowRegister	regName, regValue	Muestra el nombre y contenido de un registro de 32 bits en hexadecimal
mWrite	Text	Escribe una literal de cadena en la ventana de consola
mWriteSpace	count	Escribe uno o más espacios en la ventana de consola
mWriteString	buffer	Escribe el contenido de una variable de cadena en la ventana de consola

mDumpMem

El macro mDumpMem muestra un bloque de memoria en la ventana de consola. Recibe una constante, registro o variable que contiene el desplazamiento de la memoria que queremos mostrar en pantalla. El segundo argumento debe ser el número de componentes de memoria a mostrar, y el tercer argumento es el tamaño de cada componente de memoria. (El macro llama al procedimiento de biblioteca DumpMem, y asigna los tres argumentos a ESI, ECX y EBX, respectivamente). Vamos a suponer la siguiente definición de datos:

```
.data
arreglo DWORD 1000h,2000h,3000h,4000h
```

La siguiente instrucción muestra el arreglo, usando sus atributos predeterminados:

```
mDumpMem OFFSET arreglo, LENGTHOF arreglo, TYPE arreglo
```

Resultado:

```
Dump of offset 00405004
-----
00001000 00002000 00003000 00004000
```

La siguiente instrucción muestra el mismo arreglo que una secuencia de bytes:

```
mDumpMem OFFSET arreglo, SIZEOF arreglo, TYPE BYTE
```

Resultado:

```
Dump of offset 00405004
-----
00 10 00 00 00 20 00 00 00 30 00 00 00 40 00 00
```

El siguiente código mete tres valores en la pila, establece los valores de EBX, ECX y ESI, y utiliza a mDumpMem para mostrar la pila:

```
mov eax,0AAAAAAAAh
push eax
mov eax,0BBBBBBBBh
push eax
mov eax,0CCCCCCCCCh
push eax
mov ebx,1
mov ecx,2
mov esi,3
mDumpMem esp, 8, TYPE DWORD
```

El vaciado resultante de la pila muestra que el macro ha metido a EBX, ECX y ESI en la pila. Después de esos valores están los tres enteros que metimos en la pila antes de invocar a mDumpMem:

```
Dump of offset 0012FFAC
-----
00000003 00000002 00000001 CCCCCCCC BBBB BBBB AAAAAAAA 7C816D4F
0000001A
```

Implementación He aquí el listado de código del macro:

```
mDumpMem MACRO address:REQ, itemCount:REQ, componentSize:REQ
;
; Muestra un vaciado de memoria, usando el procedimiento DumpMem.
; Recibe: desplazamiento de memoria, cuenta del número de elementos
; a mostrar, y el tamaño de cada componente de memoria.
; Evite pasar EBX, ECX y ESI como argumentos.
;
```

```

push ebx
push ecx
push esi
mov esi, address
mov ecx, itemCount
mov ebx, componentSize
call DumpMem
pop esi
pop ecx
pop ebx
ENDM

```

mDump

El macro mDump muestra la dirección y el contenido de una variable en hexadecimal. Recibe el nombre de una variable y (de manera opcional) un carácter, indicando que debe mostrarse una etiqueta al lado de la variable. El formato de visualización coincide en forma automática con el atributo de tamaño de la variable (BYTE, WORD o DWORD). El siguiente ejemplo muestra dos llamadas a mDump:

```

.data
tamDisco DWORD 12345h
.code
mDump tamDisco ; sin etiqueta
mDump tamDisco,Y ; muestra la etiqueta

```

Cuando se ejecuta el código, se produce el siguiente resultado:

```

Dump of offset 00405000
-----
00012345
Variable name: tamDisco
Dump of offset 00405000
-----
00012345

```

Implementación A continuación se muestra un listado del macro mDump, que a su vez llama a mDumpMem. Utiliza una nueva directiva llamada IFNB (*si no está en blanco*) para averiguar si el procedimiento que hizo la llamada pasó un argumento en el segundo parámetro (vea la sección 10.3):

```

; -----
; mDump MACRO varName:REQ, useLabel
;
; Muestra una variable, usando sus atributos conocidos
; Recibe: varName, el nombre de la variable.
; Si useLabel no está en blanco, se muestra el
; nombre de la variable.
; -----
; call Crlf
IFNB <useLabel>
    mWrite "Variable name: &varName"
ELSE mWrite " "
ENDIF
mDumpMem OFFSET varName, LENGTHOF varName, TYPE varName
ENDM

```

El & en **&varName** es un *operador de sustitución*, que permite insertar el valor del parámetro **varName** en la literal de cadena. Vea la sección 10.3.7 para más detalles.

mGotoxy

El macro **mGotoxy** coloca el cursor en una posición de fila y columna específicas en el búfer de la ventana de consola. Podemos pasarle valores inmediatos de 8 bits, operandos de memoria y valores de registros:

<code>mGotoxy 10,20</code>	<code>; valores inmediatos</code>
<code>mGotoxy fila,col</code>	<code>; operandos de memoria</code>
<code>mGotoxy ch,c1</code>	<code>; valores de registros</code>

Implementación He aquí un listado de código fuente del macro:

```
-----  
mGotoxy MACRO X:REQ, Y:REQ  
;  
; Establece la posición del cursor en la ventana de consola.  
; Recibe: las coordenadas X y Y (tipo BYTE). Evite  
;     pasar DH y DL como argumentos.  
-----  
    push  edx  
    mov   dh,Y  
    mov   dl,X  
    call  Gotoxy  
    pop   edx  
ENDM
```

Cómo evitar conflictos de registros Cuando los argumentos de un macro son registros, algunas veces pueden entrar en conflicto con los registros que los macros utilizan de manera interna. Por ejemplo, si llamamos a **mGotoxy** usando DH y DL, no se genera el código correcto. Para ver por qué, inspeccionemos el código expandido una vez que se han sustituido dichos parámetros:

```
1  push  edx  
2  mov   dh,dl          ;; fila  
3  mov   dl,dh          ;; columna  
4  call  Gotoxy  
5  pop   edx
```

Suponiendo que DL se pasa como el valor Y y DH como el valor X, la línea 2 sustituye a DH antes de que tengamos oportunidad de copiar el valor de la columna a DL en la línea 3.

Siempre que sea posible, las definiciones de los macros deben especificar qué registros no pueden usarse como argumentos.

mReadString

El macro **mReadString** recibe como entrada una cadena del teclado y la almacena en un búfer. En su interior, encapsula una llamada al procedimiento de biblioteca **ReadString**. Recibe el nombre del búfer:

```
.data  
primerNombre BYTE 30 DUP(?)  
.code  
mReadString primerNombre
```

He aquí el código fuente del macro:

```
-----  
mReadString MACRO varName:REQ  
;  
; Lee de la entrada estándar y coloca los datos en un búfer.  
; Recibe: el nombre del búfer. Evite pasar
```

```

;      ECX y EDX como argumentos.
;-----
push  ecx
push  edx
mov   edx,OFFSET varName
mov   ecx,SIZEOF varName
call  ReadString
pop   edx
pop   ecx
ENDM

```

mShow

El macro mShow muestra el nombre y contenido de cualquier registro o variable en un formato seleccionado por el procedimiento que hace la llamada. Recibe el nombre del registro, seguido de una secuencia opcional de letras que identifican el formato deseado. Use los siguientes códigos: H = hexadecimal, D = decimal sin signo, I = decimal con signo, B = binario, N = agrega una nueva línea. Pueden combinarse varios formatos de salida, y pueden especificarse varias líneas nuevas. El formato predeterminado es “HIN”. mShow es una útil herramienta de depuración; el procedimiento de biblioteca DumpRegs la utiliza mucho. Podemos insertar llamadas a mShow en cualquier programa, para mostrar los valores de los registros o variables importantes.

Ejemplo Las siguientes instrucciones muestran el registro AX en hexadecimal, decimal con signo, decimal sin signo y binario:

```

mov  ax,4096
mShow AX          ; opciones predeterminadas: HIN
mShow AX,DBN     ; decimal sin signo, binario, nueva linea

```

He aquí la salida:

```

AX = 1000h + 4096d
AX = 4096d 0001 0000 0000 0000b

```

Ejemplo Las siguientes instrucciones muestran a AX, BX, CX y DX en decimal sin signo, en la misma línea de salida:

```

; Insertan algunos valores de prueba y muestran cuatro registros:
mov  ax,1
mov  bx,2
mov  cx,3
mov  dx,4
mShow AX,D
mShow BX,D
mShow CX,D
mShow DX,DN

```

He aquí la salida correspondiente:

```

AX = 1d    BX = 2d    CX = 3d    DX = 4d

```

Ejemplo La siguiente llamada a mShow muestra el contenido de **midoblepalabra** en decimal sin signo, seguido de una nueva línea:

```

.data
midoblepalabra DWORD ?
.code
mShow midoblepalabra, DN

```

Implementación La implementación de mShow es demasiado extensa como para incluirla aquí, pero la encontrará en el archivo Macros.inc. Al implementar mShow, tuvimos que tener cuidado de mostrar los valores actuales de los registros antes de que las instrucciones dentro del mismo macro los modificara.

mShowRegister

El macro mShowRegister muestra el nombre y el contenido de un registro individual de 32 bits en hexadecimal. Recibe el nombre del registro, según se desee visualizar, seguido del registro en sí. La siguiente invocación a este macro especifica el nombre a visualizar como EBX:

```
mShowRegister EBX, ebx
```

Se produce el siguiente resultado:

```
EBX=7FFD9000
```

La siguiente invocación utiliza los signos < y > alrededor de la etiqueta, ya que contiene un espacio incrustado:

```
mShowRegister <Apuntador de pila>, esp
```

Se produce el siguiente resultado:

```
Apuntador de pila=0012FFC0
```

Implementación He aquí el código fuente del macro:

```
;-----
mShowRegister MACRO regName, regValue
LOCAL tempStr
;
; Muestra el nombre de un registro de 32 bits y su contenido.
; Recibe: el nombre del registro, el valor del registro.
;-----
.data
tempStr BYTE " &regName=",0
.code
    push eax
    ; Muestra el nombre de un registro
    push edx
    mov edx,OFFSET tempStr
    call WriteString
    pop edx
    ; Muestra el contenido de un registro
    mov eax,regValue
    call WriteHex
    pop eax
ENDM
```

mWriteSpace

El macro mWriteSpace escribe uno o más espacios en la ventana de la consola. Puede recibir de manera opcional un parámetro entero que especifique el número de espacios a escribir (el predeterminado es uno). Por ejemplo, la siguiente instrucción escribe cinco espacios:

```
mWriteSpace 5
```

Implementación He aquí el código fuente para mWriteSpace:

```
;-----
mWriteSpace MACRO count:=<1>
;
; Escribe uno o más espacios en la salida estándar.
; Recibe: un entero que especifica el número de espacios.
; Si cuenta está en blanco, se escribe un solo espacio.
;-----
LOCAL spaces
```

```
.data
spaces BYTE count DUP(' '),0
.code
    push  edx
    mov   edx,OFFSET spaces
    call  WriteString
    pop   edx
ENDM
```

La sección 10.3.2 explica cómo usar inicializadores predeterminados para los parámetros de los macros.

mWriteString

El macro **mWriteString** escribe el contenido de una variable de cadena en la ventana de consola. En su interior, simplifica las llamadas a **WriteString**, al permitirnos pasarle el nombre de una variable de cadena en la misma línea de instrucción. Por ejemplo:

```
.data
cad1 BYTE "Por favor escriba su nombre: ",0
.code
mWriteString cad1
```

Implementación La siguiente implementación de **mWriteString** guarda a EDX en la pila, lo llena con el desplazamiento de la cadena y lo saca de la pila, después de la llamada al procedimiento:

```
;-----
mWriteString MACRO buffer:REQ
;
; Escribe una variable de cadena en la salida estándar.
; Recibe: el nombre de la variable de cadena.
;-----
    push  edx
    mov   edx,OFFSET buffer
    call  WriteString
    pop   edx
ENDM
```

10.2.6 Programa de ejemplo: envolturas

Vamos a crear un programa corto llamado *Envolturas.asm*, que muestra todos los macros que ya hemos presentado como envolturas de procedimientos. Como cada macro oculta una gran parte del tedioso proceso de paso de parámetros, el programa es sorprendentemente compacto. Vamos a suponer que todos los macros que hemos mostrado hasta ahora se encuentran dentro del archivo *Macros.inc*:

```
TITLE Macros de envoltura de procedimientos (Envolturas.asm)
;
; Este programa demuestra los macros como envolturas
; para los procedimientos de la biblioteca. Contenido: mGotoxy, mWrite,
; mWriteString, mReadString y mDumpMem.

INCLUDE Irvine32.inc
INCLUDE Macros.inc ; definiciones de los macros

.data
arreglo DWORD 1,2,3,4,5,6,7,8
primerNombre BYTE 31 DUP(?)
apellidoPaterno BYTE 31 DUP(?)

.code
main PROC
    mGotoxy 0,0
    mWrite <"Programa de ejemplo de macros",0dh,0ah>
```

```
; Introduce el nombre del usuario.  
mGotoxy 0,5  
mWrite "Por favor introduzca su primer nombre: "  
mReadString primerNombre  
call Crlf  
  
mWrite "Por favor introduzca su apellido paterno: "  
mReadString apellidoPaterno  
call Crlf  
  
; Muestra el nombre del usuario.  
mWrite "Su nombre es "  
mWriteString primerNombre  
mWriteSpace  
mWriteString apellidoPaterno  
call Crlf  
  
; Muestra el arreglo de enteros.  
mDumpMem OFFSET arreglo,LENGTHOF arreglo, TYPE arreglo  
exit  
main ENDP  
END main
```

Resultados del programa A continuación se muestra un ejemplo de la salida del programa:

```
Programa de ejemplo de macros  
Por favor introduzca su primer nombre: Joe  
Por favor introduzca su apellido paterno: Smith  
Su nombre es Joe Smith  
Dump of offset 00404000  
-----  
00000001 00000002 00000003 00000004 00000005  
00000006 00000007 00000008
```

10.2.7 Repaso de sección

- (Verdadero/Falso): cuando se invoca a un macro, las instrucciones CALL y RET se insertan en forma automática en el programa ensamblado.
- (Verdadero/Falso): el preprocesador del ensamblador se encarga de la expansión de los macros.
- ¿Cuál es la principal ventaja de utilizar macros con parámetros, en comparación con usar macros sin parámetros?
- (Verdadero/Falso): mientras se encuentre en el segmento de código, la definición de un macro puede aparecer antes o después de las instrucciones que la invocan.
- (Verdadero/Falso): al sustituir un procedimiento extenso con un macro que contenga el código de ese procedimiento, por lo general, se incrementa el tamaño del código compilado de un programa si el macro se invoca varias veces.
- (Verdadero/Falso): un macro no puede contener definiciones de datos.
- ¿Cuál es el propósito de la directiva LOCAL?
- ¿Qué directiva muestra un mensaje en la consola durante el paso de ensamblado?
- Escriba un macro llamado **mImprimirCar** que muestre un carácter individual en la pantalla. Debe tener dos parámetros: el primero especifica el carácter que se va a mostrar, y el segundo especifica cuántas veces debe repetirse el carácter. He aquí una llamada de ejemplo:
`mImprimirCar 'X', 20`

10. Escriba un macro llamado **mGenAleatorio** que genere un entero aleatorio entre 0 y $n - 1$. El único parámetro debe ser n .

11. Escriba un macro llamado **mPedirEntero** que muestre un indicador y reciba un entero como entrada del usuario. Debe recibir una literal de cadena y el nombre de una variable tipo doble palabra. Llamada de ejemplo:

```
.data
valMin DWORD ?
.code
mPedirEntero "Escriba el valor minimo", valMin
```

12. Escriba un macro llamado **mEscribirEn** que posicione el cursor y escriba una literal de cadena en la ventana de la consola. *Sugerencia:* invoque a los macros **mGotoxy** y **mWrite**.

13. Muestre el código expandido que produce la siguiente instrucción que invoca al macro **mWriteString** de la sección 10.2.5:

```
mWriteString indicadorNombre
```

14. Muestre el código expandido que produce la siguiente instrucción que invoca al macro **mReadString** de la sección 10.2.5:

```
mReadString nombreCliente
```

15. *Reto:* escriba un macro llamado **mDumpMemx** que reciba un solo parámetro, el nombre de una variable. Su macro debe llamar al macro **mDumpMem**, pasarle el desplazamiento de la variable, el número de unidades y el tamaño de éstas. Demuestre una llamada al macro **mDumpMemx**.

10.3 Directivas de ensamblado condicional

Podemos utilizar una variedad 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 en este capítulo no deben confundirse con las directivas en tiempo de ejecución, como .IF y .ENDIF, que presentamos en la sección 6.7. Estas últimas evalúan expresiones con base en los valores en tiempo de ejecución almacenados en registros y variables.

La tabla 10-3 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.

10.3.1 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
```

Tabla 10-3 Directivas de ensamblado condicional.

Directiva	Descripción
IF <i>expresión</i>	Permite el ensamblado si el valor de <i>expresión</i> es verdadero (distinto de cero). Los posibles operadores relacionales son LT, GT, EQ, NE, LE y GE
IFB < <i>argumento</i> >	Permite el ensamblado si <i>argumento</i> está en blanco. El nombre del argumento debe ir encerrado entre los signos <>
IFNB < <i>argumento</i> >	Permite el ensamblado si <i>argumento</i> no está en blanco. El nombre del argumento debe ir encerrado entre los signos <>
IFIDN < <i>arg1</i> >,< <i>arg2</i> >	Permite el ensamblado si los dos argumentos son iguales (idénticos). Usa una comparación sensible a mayúsculas y minúsculas
IFIDNI < <i>arg1</i> >,< <i>arg2</i> >	Permite el ensamblado si los dos argumentos son iguales. Usa una comparación insensible a mayúsculas y minúsculas
IFDIF < <i>arg1</i> >,< <i>arg2</i> >	Permite el ensamblado si los dos argumentos no son iguales. Usa una comparación sensible a mayúsculas y minúsculas
IFDFI < <i>arg1</i> >,< <i>arg2</i> >	Permite el ensamblado si los dos argumentos no son iguales. Usa una comparación insensible a mayúsculas y minúsculas
IFDEF <i>nombre</i>	Permite el ensamblado si <i>nombre</i> está definido
IFNDEF <i>nombre</i>	Permite el ensamblado si <i>nombre</i> 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 <i>expresión</i>	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 de un macro inmediatamente, evitando que se expandan todas las instrucciones siguientes del macro

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
```

(En la sección 10.2.2 vimos que 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. A continuación se muestran los resultados en pantalla cuando se ensambla un programa con un parámetro faltante:

```
Assembling: Macro2.asm
-----
* Error: falta un parámetro en mWriteString
* (no se genero código)
-----
```

10.3.2 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

nombreparám := < *argumento* >

(Los espacios antes y después de los operadores son opcionales). Por ejemplo, el macro **mWriteLn** puede proporcionar una cadena que contenga un solo espacio como su 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 el argumento predeterminado, por lo que debemos insertar cuando menos un espacio entre las comillas.

10.3.3 Expresiones booleanas

El ensamblador nos permite utilizar los siguientes operadores relacionales en expresiones booleanas constantes que contengan a IF y otras directivas condicionales:

LT	Menor que
GT	Mayor que
EQ	Igual que
NE	Distinto de
LE	Menor o igual que
GE	Mayor o igual que

10.3.4 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 registros ni nombres de variables. Un formato de sintaxis utiliza sólo 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 **mGotoxy** 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 ERSS a 1. Dependiendo del valor de Y, podemos sumar 1 a ERSS. Por último, si ERSS 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 ERSS           ; constante local
ERSS = 0
IF (X LT 0) OR (X GT 79)
    ECHO Advertencia: El primer argumento para mGotoxy (X) esta fuera de rango.
    ECHO ****
    ERSS = 1
ENDIF
IF (Y LT 0) OR (Y GT 24)
    ECHO Advertencia: El segundo argumento para mGotoxy (Y) esta fuera de rango.
    ECHO ****
    ERSS = ERSS + 1
ENDIF
IF ERSS GT 0          ; si se encontraron errores,
    EXITM            ; ; sale del macro
ENDIF
push edx
mov dh,Y
mov dl,X
call Gotoxy
pop edx
ENDM
```

10.3.5 Las directivas IFIDN e IFIDNI

La directiva IFIDNI realiza una comparación insensible a mayúsculas y minúsculas 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 del **Bufer** 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 bufer,edx
```

10.3.6 Ejemplo: suma de la fila de una matriz

En la sección 9.4.2 vimos cómo calcular la suma de una fila individual en una matriz de bytes. Un ejercicio de programación en el capítulo 9 le pidió que generalizara el procedimiento para matrices de palabras y de dobles palabras. Aunque la solución para ese ejercicio es bastante extensa, vamos a ver si podemos usar un macro para simplificar la tarea. Primero, he aquí el procedimiento **calc_sum_fila** original que se muestra en el capítulo 9:

```

calc_suma_fila PROC uses ebx ecx edx esi
;
; Calcula la suma de una fila en una matriz de bytes.
; Recibe:   EBX = desplazamiento de tabla, EAX = índice de fila,
;           ECX = tamaño de 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

```

Vamos a empezar por cambiar PROC a MACRO, eliminar la instrucción RET y cambiar ENDP a ENDM. No hay un equivalente en los macros para la directiva USES, por lo que insertamos instrucciones PUSH y POP:

```

mCalc_suma_fila MACRO
    push ebx          ; guarda los registros que cambian
    push ecx
    push esi
    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 en la fila
    loop L1
    pop esi          ; restaura los registros cambiados

```

```

pop    ecx
pop    ebx
ENDM

```

A continuación, sustituimos los parámetros del macro por los parámetros de registro e inicializamos los registros dentro del macro:

```

mCalc_suma_fila MACRO indice, desplArreglo, tamFila
    push ebx                                ; guarda los registros que cambian
    push ecx
    push esi

    ; establece los registros requeridos
    mov eax,indice
    mov ebx,desplArreglo
    mov ecx,tamFila

    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 en la fila
        loop L1
    pop esi                                ; restaura los registros cambiados
    pop ecx
    pop ebx
ENDM

```

Ahora vamos a agregar un parámetro llamado **tipoEl** que especifique el tipo del arreglo (BYTE, WORD o DWORD):

```
mCalc_suma_fila MACRO indice, desplArreglo, tamFila, tipoEl
```

El parámetro tamFila, que se copia a ECX, indica en un momento dado el número de bytes en cada fila. Si vamos a usarlo como contador de ciclo, debe contener el número de *elementos* en cada fila. Por lo tanto, dividimos ECX entre 2 para los arreglos de 16 bits y entre 4 para los arreglos de dobles palabras. Una forma rápida de lograr esto es dividir **tipoEl** entre 2 y usarlo como contador de desplazamiento, desplazando ECX a la derecha:

```
shr ecx,(TYPE tipoEl / 2)                  ; byte=0, word=1, dword=2
```

TYPE tipoEl se convierte en el factor de escala en el operando base-índice de la instrucción MOVZX:

```
movzx edx,tipoEl PTR[ebx + esi*(TYPE tipoEl)]
```

MOVZX no se ensamblará si el operando derecho es una doble palabra, por lo que debemos usar el operador IFIDNI para crear una instrucción MOV separada cuando tipoEl es igual a DWORD:

```

IFIDNI <tipoEl>,<DWORD>
    mov edx,tipoEl PTR[ebx + esi*(TYPE tipoEl)]
ELSE
    movzx edx,tipoEl PTR[ebx + esi*(TYPE tipoEl)]
ENDIF

```

Por fin tenemos el macro terminado, recordando que debemos designar la etiqueta L1 como LOCAL:

```

;-----
mCalc_suma_fila MACRO indice, desplArreglo, tamFila, tipoEl
; Calcula la suma de una fila en un arreglo bidimensional.
;
; Recibe: índice de fila, desplazamiento del arreglo, número de bytes
; en cada fila de la tabla, y el tipo del arreglo (BYTE, WORD o DWORD).
;
```

```

; Devuelve: EAX = suma.
;-----
LOCAL L1
    push ebx           ; guarda los registros que cambian
    push ecx
    push esi

; establece los registros requeridos
    mov eax,indice
    mov ebx,desplArreglo
    mov ecx,tamFila

; calcula el desplazamiento de la fila.
    mul ecx           ; índice de fila * tamaño de fila
    add ebx,eax        ; desplazamiento de fila

; prepara el contador del ciclo.
    shr ecx,(TYPE tipoElt / 2) ; byte=0, word=1, dword=2
; inicializa el acumulador y el índice de columna
    mov eax,0          ; acumulador
    mov esi,0          ; índice de columna

L1:
    IFIDNI <tipoElt>, <DWORD>
        mov edx, tipoElt PTR[ebx + esi*(TYPE tipoElt)]
    ELSE
        movzx edx, tipoElt PTR[ebx + esi*(TYPE tipoElt)]
    ENDIF
    add eax,edx        ; lo suma al acumulador
    inc esi             ; siguiente byte en la fila
    loop L1

    pop esi            ; restaura los registros cambiados
    pop ecx
    pop ebx

ENDM

```

A continuación se muestran ejemplos de llamadas al macro, usando arreglos de bytes, palabras y dobles palabras. Vea el programa *row-sum.asm*:

```

.data
tablaB DWORD 10h, 20h, 30h, 40h, 50h
TamFilaB = ($ - tablaB)
                    DWORD 60h, 70h, 80h, 90h, 0A0h
                    DWORD 0B0h, 0C0h, 0D0h, 0E0h, 0F0h

tablaW DWORD 10h, 20h, 30h, 40h, 50h
TamFilaW = ($ - tablaW)
                    DWORD 60h, 70h, 80h, 90h, 0A0h
                    DWORD 0B0h, 0C0h, 0D0h, 0E0h, 0F0h

tablaD DWORD 10h, 20h, 30h, 40h, 50h
TamFilaD = ($ - tablaD)
                    DWORD 60h, 70h, 80h, 90h, 0A0h
                    DWORD 0B0h, 0C0h, 0D0h, 0E0h, 0F0h

indice DWORD ?
.code
mCalc_suma_fila indice, OFFSET tablaB, TamFilaB, BYTE
mCalc_suma_fila indice, OFFSET tablaW, TamFilaW, WORD
mCalc_suma_fila indice, OFFSET tablaD, TamFilaD, DWORD

```

10.3.7 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** (sección 10.2.5) 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 a 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 representaciones 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 al macro. Por ejemplo, al invocar a **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 de expansión (%) 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, suponga 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 del 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 linea 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 linea NUMLINEA: EAX no puede ser el segundo
```

Suponga que la siguiente llamada al macro ocurre en un programa en la línea 40:

```
MUL32 val1,eax,val3
```

Entonces se muestra el siguiente mensaje durante el ensamblado:

```
-----  
* Error en Linea 40: EAX no puede ser el segundo  
* argumento cuando se invoca al macro MUL32.  
-----
```

En el programa llamado *Macro3.asm* podrá ver una prueba del macro **Mul32**.

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 forma se interpretarían como delimitadores u otros operadores. Por ejemplo, el macro **mWrite** que presentamos en una sección anterior de este capítulo recibe una literal de cadena como su único argumento. Si le pasamos la siguiente cadena, el preprocesador la interpretará como tres argumentos de macro separados:

```
mWrite "Linea 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 un solo 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-literal: 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**. Observe el uso del operador de sustitución (&):

```
MostrarAdvertencia MACRO mensaje  
    mWrite "&mensaje"  
ENDM
```

A continuación, invitamos a **MostrarAdvertencia** y le pasamos la expresión %ValorYIncorrecto. El operador % evalúa (desreferencia) a **ValorYIncorrecto** y produce su 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
```

10.3.8 Macrofunciones

Una macrofunción es similar a un macroprocedimiento, en cuanto a que 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 símbolo dado; en caso contrario, devuelve falso (0):

```
EstaDefinido MACRO símbolo
    IFDEF símbolo
        EXITM <-1>                ;; Verdadero
    ELSE
        EXITM <0>                 ;; Falso
    ENDIF
ENDM
```

La directiva EXITM (salir de macro) detiene el resto de la expansión de la macro.

Llamada a una macrofunción Al llamar a una macrofunción, su lista de argumentos debe ir encerrada entre paréntesis. Por ejemplo, podemos llamar a la macro **EstaDefinido** y pasárle **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
```

Una macro como **EstaDefinido** puede ser útil cuando se diseñan programas para varios modelos 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 del 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 RealMode:

```
ML -c miProg.asm
```

Programa HolaNuevo El siguiente programa (*HolaNuevo.asm*) utiliza las macros que acabamos de describir, mostrando un mensaje en la pantalla:

```
TITLE MacroFunciones          (HolaNuevo.asm)
INCLUDE Macros.inc
IF IsDefined( RealMode )
    INCLUDE Irvine16.inc
ELSE
    INCLUDE Irvine32.inc
ENDIF

.code
main PROC
    Startup
    mWrite <"Este programa puede ensamblarse para ejecutarse ",0dh,0ah>
    mWrite <"tanto en modo Real como en modo Protegido.",0dh,0ah>
    exit
main ENDP
END main
```

Este programa puede ensamblarse en modo de direccionamiento real, usando *makeHola16.bat*, o en modo protegido, usando *make32.bat*.

10.3.9 Repaso de sección

1. ¿Cuál es el propósito de la directiva IFB?
2. ¿Cuál es el propósito de la directiva IFIDN?
3. ¿Qué directiva detiene el resto de la expansión de una macro?
4. ¿Qué diferencia hay entre IFIDNI e IFIDN?
5. ¿Cuál es el propósito de la directiva IFDEF?
6. ¿Qué directiva marca el final de un bloque condicional de instrucciones?
7. Muestre un ejemplo de un parámetro de macro que tenga un inicializador de argumento predeterminado.
8. Liste todos los operadores relacionales que pueden usarse en expresiones booleanas constantes.
9. Escriba un ejemplo corto que utilice las directivas IF, ELSE y ENDIF.
10. Escriba una instrucción en la que utilice la directiva IF para comprobar el valor del parámetro de macro constante Z; si Z es menor que cero, muestre un mensaje durante el ensamblado, indicando que el parámetro Z es inválido.
11. ¿Cuál es el propósito del operador & en la definición de una macro?
12. ¿Cuál es el propósito del operador ! en la definición de una macro?
13. ¿Cuál es el propósito del operador % en la definición de una macro?
14. Escriba una macro corta para demostrar el uso del operador & cuando el parámetro de la macro está incrustado en una cadena literal.
15. Suponga que se tiene la siguiente definición de la macro **mLocate**:

```
mLocate MACRO valX,valY
    IF valX LT 0                ;; ¿valX < 0?
        EXITM                   ;; de ser así, termina
    ENDIF
    IF valY LT 0                ;; ¿valY < 0?
        EXITM                   ;; de ser así, termina
    ENDIF
    mov   bx,0                  ;; página de video 0
    mov   ah,2                  ;; posiciona el cursor
    mov   dh,valY
```

```

    mov dl, valX
    int 10h           ;; llama al BIOS
ENDM

```

Muestre el código fuente generado por el preprocesador cuando se expande la macro mediante cada una de las siguientes instrucciones:

```

.data
fila BYTE 15
col BYTE 60
.code
mLocate -2,20
mLocate 10,20
mLocate col,fila

```

10.4 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 del 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.
- La directiva FORC repite un bloque de instrucciones iterando a través de una cadena de caracteres.

Cada una de estas directivas se demuestra en un programa de ejemplo llamado *Repeticion.asm*.

10.4.1 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
ENDM

```

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).

10.4.2 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 ubicacion, 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.

10.4.3 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 del 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
    Cursos 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 <>
```

10.4.4 Directiva FORC

La directiva FORC repite un bloque de instrucciones al 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 del 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. Observe que < y > deben ir precedidos por el operador de carácter-literal (!) para evitar que violen la sintaxis de la directiva FORC:

```
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	"\$"
00000003	251	BYTE	"%"
00000004	5E1	BYTE	"^"
00000005	261	BYTE	"&"
00000006	2A1	BYTE	"*"
00000007	3C1	BYTE	"<"
00000008	3E1	BYTE	">"

10.4.5 Ejemplo: lista enlazada

Es muy sencillo combinar la declaración de una estructura con la directiva REPEAT para indicar al ensamblador que cree una estructura de datos tipo lista enlazada. Cada nodo en una lista enlazada contiene un área de datos y un área de enlace:



En el área de datos, una o más variables pueden guardar datos únicos para cada nodo. En el área de enlace, un apuntador contiene la dirección del siguiente nodo en la lista. La parte del enlace del nodo final, por lo general, contiene un apuntador nulo. Vamos a crear un programa para crear y mostrar una lista enlazada simple. Primero, el programa define un nodo de la lista que tiene un entero individual (datos) y un apuntador al siguiente nodo:

```
NodoLista STRUCT
    DatosNodo DWORD ?
    ApuntSig DWORD ?           ; los datos del nodo
                                ; apuntador al siguiente nodo
NodoLista ENDS
```

A continuación, la directiva REPEAT crea varias instancias de objetos **NodoLista**. Para fines de prueba, el campo **DatosNodo** contiene una constante entera que varía de 1 a 15. Dentro del ciclo, incrementamos el contador e insertamos valores en los campos NodoLista:

```
CuentaTotalNodos = 15
NULL = 0
Contador = 0
.data
ListaEnlazada LABEL PTR NodoLista
```

```
REPT CuentaTotalNodos
    Contador = Contador + 1
    NodoLista <Contador, ($ + Contador * SIZEOF NodoLista)>
ENDM
```

La expresión $(\$ + Contador * \text{SIZEOF } \text{NodoLista})$ indica al ensamblador que debe multiplicar el contador por el tamaño de **NodoLista** y sumar su producto al contador de la ubicación actual. El valor se inserta en el campo **ApuntSig** en la estructura. (Es interesante observar que el valor del contador de ubicación (\$) permanece fijo en el primer nodo de la lista). La lista recibe un *nodo final* que marca su fin, en el cual el campo **ApuntSig** contiene el valor nulo (0):

NodoLista <0.0>

Cuando el programa recorre la lista, utiliza las siguientes instrucciones para obtener el campo **ApuntSig** y compararlo con NULL, de manera que pueda detectarse el final de la lista:

```
mov eax,(NodoLista PTR [esi]).ApuntSig  
cmp eax,NULL
```

Listado del programa A continuación se muestra el listado completo del programa. En main, un ciclo recorre la lista y muestra todos los valores de los nodos. En vez de utilizar un contador fijo para el ciclo, el programa comprueba la existencia del apuntador nulo en el nodo final y detiene la iteración al encontrarlo:

TITLE Creación de una lista enlazada (lista.asm)

INCLUDE Tryvne32.inc

```
NodoLista STRUCT  
    DatosNodo DWORD ?  
    ApuntSig  DWORD ?  
NodoLista ENDS
```

```
CuentaTotalNodos = 15  
NULL = 0  
Contador = 0
```

```
.data  
ListaEnlazada LABEL PTR NodoLista
```

```
REPT CuentaTotalNodos
    Contador = Contador + 1
;NodoLista <Contador, ($ + SIZEOF NodoLista)> ; PRUEBE ESTO
    NodoLista <Contador, ($ + Contador * SIZEOF NodoLista)>
```

ENDM
Nodo lista <0,0> : nodo final

Received by Dr. G. S. , Head of the Department of Zoology, University of Madras, Madras, India.

```
.code  
main PROC  
    mov    esi,OFFSET ListaEnlazada
```

: Muestra los enteros en los miembros DatosNodo.

, *maesra los*
SiguienteNodo:

: Verifica el nodo final.

mov eax, (Nod

cmp eax, NULL

je terminar

; Muestra los datos del nodo.

mov eax, (Nod

call WriteDec

call Crlf

; Obtiene apuntador al siguiente nodo.

```

    mov  esi,(NodoLista PTR [esi]).ApuntSig
    jmp  SiguienteNodo
terminar:
    exit
main ENDP
END main

```

10.4.6 Repaso de sección

1. Describa brevemente la directiva WHILE.
2. Describa brevemente la directiva REPEAT.
3. Describa brevemente la directiva FOR.
4. Describa brevemente la directiva FORC.
5. ¿Qué directiva de iteración sería la mejor herramienta para generar una tabla de búsqueda de caracteres?
6. Escriba las instrucciones que genera la siguiente macro:

```

FOR val,<100,20,30>
BYTE 0,0,0,val
ENDM

```

7. Suponga que se ha definido la siguiente macro **mRepetir**:

```

mRepetir MACRO car,cuenta
LOCAL L1
    mov cx,cuenta
L1: mov ah,2
    mov dl,char
    int 21h
    loop L1
ENDM

```

Escriba el código que genera el preprocesador, al expandir la macro **mRepetir** mediante cada una de las siguientes instrucciones (a, b y c):

```

mRepetir 'X',50           ; a
mRepetir AL,20            ; b
mRepetir valByte,valCuenta ; c

```

8. *Reto:* en el programa de ejemplo de Lista enlazada (sección 10.4.5), ¿cuál sería el resultado si el ciclo REPEAT se codificara de la siguiente manera?

```

REPEAT CuentaTotalNodos
    Contador = Contador + 1
    NodoLista <Contador, ($ + SIZEOF NodoLista)>
ENDM

```

10.5 Resumen del capítulo

Una *estructura* es una plantilla o patrón que se utiliza al crear tipos definidos por los usuarios. Muchas estructuras ya están definidas en la biblioteca de la API de MS Windows y se utilizan para transferir datos entre los programas de aplicación y la biblioteca. Las estructuras pueden contener un conjunto diverso de tipos de campos. Cada declaración de campo puede utilizar un inicializador de campo, el cual asigna un valor predeterminado al campo.

Las estructuras por sí solas no ocupan memoria, pero las variables de estructuras sí. El operador SIZEOF devuelve el número de bytes utilizados por una variable.

El operador punto (.) hace referencia al campo de una estructura mediante el uso de una variable de estructura o de un operando indirecto, como [esi]. Cuando un operando indirecto hace referencia al campo de una estructura, debemos usar el operador PTR para identificar el tipo de estructura, como en (COORD PTR [esi]).X.

Las estructuras pueden contener campos que también sean estructuras. En el programa Paso del borracho mostramos un ejemplo (sección 10.1.6), en donde la estructura **PasoBorracho** contenía un arreglo de estructuras COORD.

Por lo general, las macros se definen al principio de un programa, antes de los segmentos de datos y de código. Después, al llamar a una macro, el preprocesador inserta una copia del código de la macro en el programa, en la ubicación en la que se hizo la llamada.

Las macros pueden usarse de manera efectiva como *envolturas* alrededor de las llamadas a procedimientos, para simplificar el paso de parámetros y guardar los registros en la pila. Los macros como **mGotoxy**, **mDumpMem** y **mWriteString** son ejemplos de envolturas, ya que llaman a los procedimientos de la biblioteca de enlace del libro.

Un *macro procedimiento* (o *macro*) es un bloque con nombre de instrucciones en lenguaje ensamblador. Una *macro función* es similar, sólo que también devuelve un valor constante.

Las directivas de ensamblado condicional como IF, IFNB e IFIDNI pueden usarse para detectar argumentos faltantes, que estén fuera de rango o que sean del tipo incorrecto. La directiva ECHO muestra mensajes de error durante el ensamblado, con lo cual es posible alertar al programador sobre los errores en los argumentos que se pasan a las macros.

El operador de sustitución (&) resuelve las referencias ambiguas a los nombres de parámetros. El operador de expansión (%) expande las macros de texto y convierte las expresiones constantes en texto. El operador de texto-literal (<>) agrupa varios caracteres y texto en una sola literal. El operador de carácter-literal (!) obliga al preprocesador a tratar los operadores predefinidos como caracteres ordinarios.

Las directivas de bloques de repetición pueden reducir la cantidad de código repetitivo en los programas. Estas directivas son:

- WHILE repite un bloque de instrucciones con base en una expresión booleana.
- REPEAT repite un bloque de instrucciones con base en el valor de un contador.
- FOR repite un bloque de instrucciones al iterar a través de una lista de símbolos.
- FORC repite un bloque de instrucciones al iterar a través de una cadena de caracteres.

10.6 Ejercicios de programación

1. Macro mReadKey

Cree una macro que espere una pulsación de tecla y devuelva la tecla que se oprimió. La macro debe incluir parámetros para el código ASCII y el código de exploración del teclado. *Sugerencia:* llame a ReadKey de la biblioteca de enlace del libro. Escriba un programa para probar su macro. Por ejemplo, el siguiente código espera una tecla; cuando regresa, los dos argumentos contienen el código ASCII y el código de exploración:

```
.data
ascii BYTE ?
exploracion BYTE ?
.code
mReadKey ascii, exploracion
```

2. Macro mWriteStringAttr

(Requiere la lectura de la sección 15.3.3 o de la sección 11.1.11). Cree una macro que escriba una cadena con terminación nula a la consola, con un color de texto dado. Los parámetros de la macro deben incluir el nombre de la cadena y el color. *Sugerencia:* llame a SetTextColor de la biblioteca de enlace del libro. Escriba un programa para probar su macro con distintas cadenas, en distintos colores. He aquí una llamada de ejemplo:

```
.data
miCadena db "He aqui mi cadena",0
.code
mWriteString miCadena, white
```

3. Macro mMove32

Escriba una macro llamada **mMove32** que reciba dos operandos de memoria de 32 bits. La macro debe mover el operando de origen al operando de destino. Escriba un programa para probar su macro.

4. Macro mMult32

Cree una macro llamada **mMult32** que multiplique dos operandos de memoria de 32 bits y produzca un producto de 32 bits. Escriba un programa para probar su macro.

5. Macro mReadInt

Cree una macro llamada **mReadInt** que lea un entero con signo de 16 o 32 bits de la entrada estándar y devuelva su valor en un argumento. Use operadores condicionales para permitir que la macro se adapte al tamaño del resultado deseado. Escriba un programa que llame a la macro y le pase operandos de varios tamaños.

6. Macro mWriteInt

Cree una macro llamada **mWriteInt** que escriba un entero con signo a la salida estándar, mediante una llamada al procedimiento **WriteInt** de la biblioteca. El argumento que se pase a la macro puede ser un byte, una palabra o una doble palabra. Use operadores condicionales en la macro, para que ésta se adapte al tamaño del argumento. Escriba un programa para probar la macro, que le pase argumentos de distintos tamaños.

7. Macro mScroll

(Requiere la lectura de la sección 15.3.3.) Cree una macro llamada **mScroll** que muestre un rectángulo de color en la ventana de consola. Incluya los siguientes parámetros en la definición de la macro. Si **atrib** está en blanco, asuma el uso de caracteres color gris claro sobre un fondo negro:

FilaSI	Fila de la ventana superior izquierda
ColSI	Columna de la ventana superior izquierda
FilaID	Fila de la ventana inferior derecha
ColID	Columna de la ventana inferior derecha
atrib	Color de las líneas desplazadas

Escriba un programa para probar su macro.

8. Paso del borracho

Al probar el programa del Paso del borracho, tal vez se haya dado cuenta de que el profesor no parece deambular muy lejos del punto inicial. Esto sin duda se debe a que la probabilidad de que el profesor se mueva en cualquier dirección es la misma. Modifique el programa para que haya una probabilidad del 50% de que el profesor continúe caminando en la misma dirección que cuando dio el paso anterior. Debe haber una probabilidad del 10% de que invierta su dirección y una probabilidad del 20% de que voltee a la izquierda o a la derecha. Asigne una dirección inicial predeterminada antes de que empiece el ciclo.

9. Desplazamiento de varias dobles palabras

Use el programa de solución del ejercicio 4 en la sección 7.8 como punto inicial para este ejercicio. Cree una macro llamada **mDesplazarDoblesPalabras** para desplazar el contenido en **nombreArreglo**, ya sea a la izquierda o a la derecha (con base en la variable dirección), para un número especificado de bits:

```
mDesplazarDoblesPalabras MACRO,  
    nombreArreglo          ;; nombre del arreglo  
    direccion,              ;; D o I  
    numeroDeBits           ;; contador de desplazamientos
```

10. Instrucciones con tres operandos

Algunos conjuntos de instrucciones de computadora permiten el uso de instrucciones aritméticas con tres operandos. Dichas operaciones aparecen algunas veces en ensambladores virtuales simples, que se utilizan para presentar a los estudiantes el concepto del lenguaje ensamblador, o para utilizar un lenguaje intermedio en los compiladores. En las siguientes macros, suponga que EAX se reserva para las operaciones de la macro

y que no se preserva. Los demás registros modificados por la macro deben preservarse. Todos los parámetros son dobles palabras de memoria. Escriba macros para simular las siguientes operaciones:

- a. add3 destino, origen1, origen2
- b. sub3 destino, origen1, origen2 (destino = origen1 - origen2)
- c. mul3 destino, origen1, origen2
- d. div3 destino, origen1, origen2 (destino = origen1 / origen2)

Por ejemplo, las siguientes llamadas a macros implementan la expresión $x = (w + y) * z$:

```
.data  
temp DWORD ?  
.code  
add3 temp, w, y ; temp = w + y  
mul3 x, temp, z ; x = temp * z
```

Escriba un programa para probar sus macros; para ello implemente cuatro expresiones aritméticas, cada una de las cuales debe involucrar varias operaciones.

Nota final

1. Tal vez debido a que RECORD es el término utilizado en el antiguo lenguaje de programación COBOL, conocido para los diseñadores de Windows NT.

PROGRAMACIÓN EN MS-Windows

11.1	Programación de la consola Win32	11.2.2	La función MessageBox
11.1.1	Antecedentes	11.2.3	El procedimiento WinMain
11.1.2	Funciones de la consola Win32	11.2.4	El procedimiento WinProc
11.1.3	Visualización de un cuadro de mensaje	11.2.5	El procedimiento ErrorHandler
11.1.4	Entrada de consola	11.2.6	Listado del programa
11.1.5	Salida de consola	11.2.7	Repaso de sección
11.1.6	Lectura y escritura de archivos	11.3	Asignación dinámica de memoria
11.1.7	E/S de archivos en la biblioteca Irvine32	11.3.1	Programas PruebaMonton
11.1.8	Prueba de los procedimientos de E/S de archivos	11.3.2	Repaso de sección
11.1.9	Manipulación de ventanas de consola	11.4	Administración de memoria en la familia IA-32
11.1.10	Control del cursor	11.4.1	Direcciones lineales
11.1.11	Control del color de texto	11.4.2	Traducción de páginas
11.1.12	Funciones de hora y fecha	11.4.3	Repaso de sección
11.1.13	Repaso de sección	11.5	Resumen del capítulo
11.2	Escritura de una aplicación gráfica de Windows	11.6	Ejercicios de programación
11.2.1	Estructuras necesarias		

11.1 Programación de la consola Win32

Al leer este libro, algunas de las siguientes preguntas deben haber estado en su mente:

- ¿De qué manera manejan los programas de 32 bits la entrada-salida de texto?
- ¿Cómo se manejan los colores en el modo de consola de 32 bits?
- ¿Cómo funciona la biblioteca de enlace Irvine32?
- ¿Cómo se manejan las horas y fechas en MS-Windows?
- ¿Cómo puedo usar las funciones de MS-Windows para leer y escribir en archivos de datos?
- ¿Es posible escribir una aplicación Windows gráfica en lenguaje ensamblador?
- ¿De qué manera traducen los programas en modo protegido los segmentos y desplazamientos en direcciones físicas?
- Sé que la memoria virtual es buena. Pero ¿por qué es así?

Este capítulo responderá a estas y otras preguntas más, a medida que le mostremos los fundamentos de la programación de 32 bits bajo Microsoft Windows. La mayoría de la información que se presenta aquí está orientada a las aplicaciones de texto en modo de consola de 32 bits, ya que son mucho más fáciles de programar,

considerando un conocimiento de las estructuras y los parámetros de los procedimientos. La biblioteca de enlace Irvine32 se basa por completo en funciones de consola Win32, por lo que podemos comparar su código fuente con la información de este capítulo. Encontrará el código fuente en el sitio Web de este libro.

¿Por qué no escribir aplicaciones gráficas para MS-Windows? Si se escriben en lenguaje ensamblador o en C, los programas gráficos son extensos y detallados. Durante años, los programadores de C y C++ han trabajado con los detalles técnicos como los manejadores de dispositivos gráficos, la publicación de mensajes, la métrica de las fuentes, los mapas de bits de los dispositivos y los modos de asignación, con la ayuda de excelentes autores. Hay un grupo dedicado de programadores en lenguaje ensamblador con excelente sitios Web, que se dedican a la programación gráfica en Windows. Consulte el vínculo a *Assembly Language Sources (Fuentes de lenguaje ensamblador)* de la página Web de este libro.

Para no desanimar a los programadores gráficos, la sección 11.2 introduce la programación gráfica de 32 bits de una forma genérica. Es sólo un comienzo, pero tal vez se interese y desee profundizar más en el tema. En el resumen al final de este capítulo se incluye una lista de libros recomendados para ampliar su estudio.

En la superficie, los programas en modo de consola de 32 bits se ven y se comportan como los programas de MS-DOS de 16 bits que se ejecutan en modo de texto. No obstante, hay diferencias: los primeros se ejecutan en modo protegido de 32 bits, mientras que los programas de MS-DOS se ejecutan en modo de direccionamiento real. Utilizan distintas bibliotecas de funciones. Los programas Win32 llaman a las funciones de la misma biblioteca que utilizan las aplicaciones Windows gráficas. Los programas de MS-DOS utilizan interrupciones de BIOS y de MS-DOS que han existido desde la introducción de la IBM-PC.

Una *Interfaz de programación de aplicaciones* (API) es una colección de tipos, constantes y funciones que proporcionan la manera de manipular directamente los objetos a través de la programación. Por lo tanto, la API Win32 nos permite utilizar las funciones en la versión de 32 bits de MS-Windows.

SDK de la plataforma Win32 El *SDK de la plataforma* Microsoft (Kit de desarrollo de software) está muy relacionado con la API Win32. Este Kit es una colección de herramientas, bibliotecas, código de ejemplo y documentación para crear aplicaciones de MS-Windows. En el sitio Web de Microsoft encontrará la documentación completa en línea. Busque “*SDK de la plataforma*” en www.msdn.microsoft.com/mexico. El SDK de la plataforma es una descarga gratuita.

Tip: la biblioteca Irvine32 es compatible con las funciones de la API de Windows, por lo que puede llamar a ambas desde el mismo programa.

11.1.1 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 *make32.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 más 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 eventos 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 presentarle una variedad de funciones de la API Win32 y proporcionar unos cuantos ejemplos. Existen muchos detalles que no podemos cubrir aquí, debido a las limitaciones de espacio. Para averiguar más, haga clic en la Ayuda dentro de Microsoft Visual C++ Express, o visite el sitio Web de Microsoft MSDN (que se encuentra actualmente en www.msdn.microsoft.com/mexico). Al buscar funciones o identificadores, establezca el parámetro *Filtrado por a SDK de la plataforma*. Además,

en los programas de ejemplo que se proporcionan en el sitio web de libro, los archivos kernel32.txt y user32.txt ofrecen listas exhaustivas de nombres de funciones en las bibliotecas kernel32.lib y user32.lib.

Constantes A menudo, cuando lea la documentación para las funciones de la API Win32, encontrará nombres de constantes como TIME_ZONE_ID_UNKNOWN. En algunos casos, la constante ya estará definida en SmallWin.inc. Pero si no puede encontrarla ahí, busque en el sitio Web de nuestro 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, puede agregar lo siguiente a *SmallWin.h* o a su propio archivo de inclusión:

```
TIME_ZONE_ID_UNKNOWN = 0  
TIME_ZONE_ID_STANDARD = 1  
TIME_ZONE_ID_DAYLIGHT = 2
```

Conjuntos 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 de 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 los caracteres ANSI de 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 en este libro, redefinimos los nombres de 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 un 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 tamaño, etcétera). Estas funciones también permiten un control detallado del tamaño y la posición de la ventana, así como los colores del 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 (en la tabla 11-1 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 las letras LP es un *apuntador largo* (*long*) a algún otro objeto.

Archivo de inclusión SmallWin.inc

SmallWin.inc, creado por el autor, 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 libro. El archivo se encuentra en la carpeta en donde instaló los programas de ejemplo de este libro. Encontrará 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 sólo mostraremos una síntesis:

```
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
```

Tabla 11-1 Traducción de los tipos de MS Windows a MASM.

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 constante, 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 se pasa como parámetro a un procedimiento de ventana o función de devolución de llamada (callback)

El tipo HANDLE, un alias para DWORD, ayuda a nuestros prototipos de función a ser más consistentes con la documentación de Microsoft Win32:

```
HANDLE TEXTEQU <DWORD>
```

SmallWin.inc también incluye las definiciones de las estructuras utilizadas en las llamadas a Win32. Dos de ellas se muestran a continuación:

```
COORD STRUCT
    X WORD ?
    Y WORD ?
COORD ENDS

SYSTEMTIME STRUCT
    wYear WORD ?
    wMonth WORD ?
    wDayOfWeek WORD ?
    wDay WORD ?
    wHour WORD ?
    wMinute WORD ?
    wSecond WORD ?
    wMilliseconds WORD ?
SYSTEMTIME 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 consola de Win32 requieren recibir un manejador como primer argumento. Un *manejador* es un entero de 32 bits sin signo que identifica de 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 búfer 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 del 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
```

11.1.2 Funciones de la consola Win32

La tabla 11-2 contiene una referencia rápida al conjunto completo de funciones de consola Win32.¹ Encontrará una descripción completa de cada función en la biblioteca MSDN, en www.msdn.microsoft.com/mexico.

Tip: las funciones de la API Win32 no preservan EAX, EBX, ECX y EDX, por lo que debemos meter y sacar estos registros por nuestra cuenta.

Tabla 11-2 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 subprocessos
FillConsoleOutputAttribute	Establece los atributos de texto y color de fondo para un número especificado de celdas de caracteres
FillConsoleOutputCharacter	Escribe un carácter en el búfer de pantalla, un número especificado de veces
FlushConsoleInputBuffer	Vacía el búfer de entrada de la consola
FreeConsole	Desconecta el proceso que hizo la llamada de su 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 búfer 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
ReadConsoleOutputAttribute	Copia un número especificado de atributos de color de texto y de fondo, de las celdas consecutivas de un búfer de pantalla de consola
ReadConsoleOutputCharacter	Copia un número de caracteres de las celdas consecutivas de un búfer de pantalla de consola
ScrollConsoleScreenBuffer	Mueve un bloque de datos en un búfer de pantalla
SetConsoleActiveScreenBuffer	Establece el búfer de pantalla especificado para que sea el búfer de pantalla de consola que se muestra actualmente
SetConsoleCP	Establece la página 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

(Continúa)

Tabla 11-2 (Continuación)

Función	Descripción
SetConsoleCursorInfo	Establece el tamaño y la 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
SetConsoleOutputCP	Establece la página de código de salida utilizada por la consola asociada con el proceso que hizo la llamada
SetConsoleScreenBufferSize	Modifica el tamaño del búfer 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 búfer 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 de un búfer de pantalla de consola

11.1.3 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 cuadro
    lpCaption:PTR BYTE,   ; cadena, título del cuadro 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 de diálogo. El parámetro *uType* 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 iconos y la opción de botón predeterminada. Hay varias combinaciones de botones posibles:

- MB_OK
- MB_OKCANCEL
- MB_YESNO
- MB_YESNOCANCEL
- MB_RETRYCANCEL
- MB_ABORTRETRYIGNORE
- MB_CANCELTRYCONTINUE

Botón predeterminado Puede elegir qué botón se seleccionará de manera automática si el usuario oprime Intro. Las opciones son MB_DEFBUTTON1 (la predeterminada), MB_DEFBUTTON2, MB_DEFBUTTON3 y MB_DEFBUTTON4. Los botones están numerados a partir de la izquierda, empezando con 1.

Iconos Hay cuatro opciones de iconos disponibles. Algunas veces más de una constante produce el mismo ícono:

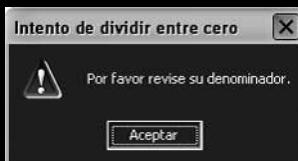
- Signo de alto: MB_ICONSTOP, MB_ICONHAND o MB_ICONERROR
- Signo de interrogación (?): MB_ICONQUESTION
- Símbolo de información (i): MB_ICONINFORMATION, MB_ICONASTERISK
- Signo de exclamación (!): MB_ICONEXCLAMATION, MB_ICONWARNING

Valor de retorno Si MessageBoxA falla, devuelve cero. En cualquier otro caso, devuelve un entero que especifica qué botón oprimió el usuario al cerrar el cuadro. Las opciones son IDABORT, IDCANCEL, IDCONTINUE, IDIGNORE, IDNO, IDOK, IDRETRY, IDTRYAGAIN e IDYES. Todas están definidas en SmallWin.inc.

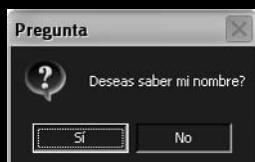
SmallWin.inc redefine a **MessageBoxA** como **MessageBox**, el cual parece un nombre más amigable para el usuario.

Programa de demostración

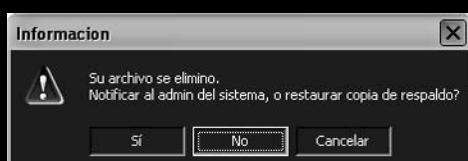
El siguiente programa (*MessageBox.asm*) demuestra algunas capacidades de la función **MessageBoxA**. La primera llamada a la función muestra un mensaje de advertencia:



La segunda llamada a la función hace una pregunta. Si el usuario selecciona el botón Sí, el programa utiliza el valor de retorno para seleccionar un curso de acción:



La tercera llamada a la función muestra tres botones y hace una pregunta para saber qué consideración sería segura:



Listado del programa He aquí el listado del programa. Como **MessageBox** es un alias para **MessageBoxA**, aquí se utiliza el nombre más simple:

```
TITLE Demostración de MessageBoxA      (MessageBox.asm)
INCLUDE Irvine32.inc
.data
```

```

leyendaA      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                  ; si no, se omite

; 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 desea que su ventana de cuadro de mensaje flote por encima de todas las demás ventanas en su escritorio, agregue la opción MB_SYSTEMMODAL a los valores que pase al último argumento (el parámetro *uType*).

11.1.4 Entrada de consola

Para estos momentos, ya hemos usado los procedimientos **ReadString** y **ReadChar** de la biblioteca de enlace del libro 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 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 búfer 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 (*Nota: se tradujeron los comentarios de las funciones para facilitar su comprensión al lector. Los archivos originales no se modificaron*) :

```

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 chars. a 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 le 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 desea que el búfer de entrada contenga una cadena con terminación nula, sustituya el byte que contiene 0Dh con un byte nulo. Esto es exactamente lo que hace el procedimiento **ReadString** de Irvine32.lib.

Nota: las funciones de la API Win32 no preservan los registros EAX, EBX, ECX y EDX.

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. Observe 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 búfer
    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, los caracteres de fin de línea que se insertan cuando el usuario oprime Intro. **bytesLeidos** es igual a 9:

Dump of offset 00404000

61 62 63 64 65 66 67 0D 0A

Comprobación de errores

Si una función de la 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 mensaje que explique el error. Para ello, hay que llamar a la función **FormatMessage**:

FormatMessage PROTO, dwFlags:DWORD, lpSource:DWORD, dwMsgID:DWORD, dwLanguageID:DWORD, lpBuffer:PTR BYTE, nSize:DWORD, va_list:DWORD	; da formato a un mensaje ; opciones de formato ; ubicación de definición de mensaje ; identificador de mensaje ; identificador de lenguaje ; apuntador al búfer que recibe la cadena ; tamaño del búfer ; apuntador a la lista de argumentos
---	--

Sus parámetros son algo complicados, así que tendrá 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:

- *dwFlags*, 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 a **GetLastError**.
- *dwLanguageID*, 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 búfer que recibe la cadena de mensaje con terminación nula. 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 para *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 ? ; apunta al mensaje de error
.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 asignado por **FormatMessage**:

```
INVOKE LocalFree, pMsjError
```

WriteWindowsMsg La biblioteca de enlace del libro contiene el siguiente procedimiento llamado **WriteWindowsMsg**, el cual encapsula los detalles relacionados con el manejo de mensajes:

```
;-----  
WriteWindowsMsg PROC USES eax edx  
;  
; Muestra una cadena que contiene el error más reciente  
; generado por MS-Windows.  
; Recibe: nada  
; Devuelve: nada  
; Última actualización: 6/10/05  
;-----  
.data  
EscribeMsjWindows_1 BYTE "Error ",0  
EscribeMsjWindows_2 BYTE ": ",0  
pMsjError DWORD ? ; apunta al mensaje de error  
IdMensaje DWORD ?  
.code  
    call GetLastError  
    mov IdMensaje,eax  
;  
; Muestra el número de error.  
    mov edx,OFFSET EscribeMsjWindows_1  
    call WriteString  
    call WriteDec ; muestra el número de error  
    mov edx,OFFSET EscribeMsjWindows_2  
    call WriteString  
;  
; Obtiene la cadena de mensaje correspondiente.  
    INVOKE FormatMessage, FORMAT_MESSAGE_ALLOCATE_BUFFER + \  
        FORMAT_MESSAGE_FROM_SYSTEM, NULL, IdMensaje, NULL,  
        ADDR pMsjError, NULL, NULL  
;  
; Muestra el mensaje de error generado por MS-Windows.  
    mov edx,pMsjError  
    call WriteString  
;  
; Libera la cadena de mensaje de error.  
    INVOKE LocalFree, pMsjError  
    ret  
WriteWindowsMsg ENDP
```

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 actualmente. 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 demás 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 desea aprender más acerca del proceso de introducción de datos mediante el teclado, lea el tema *About Keyboard Input (Acerca de la entrada del teclado)* en la documentación del SDK de la plataforma. Para una lista de constantes de teclas virtuales, consulte el archivo *VirtualKeys.inc* en el directorio \Examples\ch11 del archivo de códigos del sitio Web libro.

Procedimientos de teclado de Irvine32 La biblioteca Irvine32 tiene dos procedimientos relacionados:

- **ReadChar** espera a que se teclee un carácter ASCII en el teclado y devuelve ese 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 de EAX y EDX se sobrescriben.

En ReadKey, si AL contiene cero, el usuario puede haber oprimido una tecla especial (tecla de función, flecha del cursor, etcétera.). El registro AH contiene el código de exploración del teclado, el cual se puede relacionar con la lista de teclas en la página que está detrás de la portada de este libro. DX contiene el código de tecla virtual y EBX contiene información acerca de los estados de las teclas de control del teclado. En la tabla 11-3 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, por lo que no la mostraremos aquí. Puede consultarla en el archivo Irvine32.asm de la carpeta \Ejemplos\Lib32 del libro.

Tabla 11-3 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
SCROLLLOCK_ON	La luz SCROLL LOCK está encendida
SHIFT_PRESSED	Se oprimió la tecla MAYÚS

Programa de prueba de 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 Mayús está oprimida o no. Como mencionamos en el capítulo 5, 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                  ; retraso para el procesamiento de mensajes
      call   Delay
      call   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 cuáles 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 muestran en la tabla 11-4. Nuestro programa debe probar el valor devuelto en EAX, según lo indica la misma tabla.

Tabla 11-4 Prueba de las teclas con GetKeyState.

Tecla	Símbolo de tecla virtual	Bit que se evalúa en EAX
Bloq Num	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 Despl 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
```

11.1.5 Salida de consola

En los primeros capítulos tratamos de hacer la salida de la consola lo más sencilla posible. En el capítulo 5, el procedimiento **WriteString** en la biblioteca de enlace Irvine32 sólo requería un argumento, el desplazamiento de una cadena en EDX. Resulta que WriteString es en realidad una envoltura de una llamada más detallada a una función Win32 llamada **WriteConsole**.

No obstante, en este capítulo aprenderá 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 de las funciones de 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 del búfer de pantalla en la ventana de consola:

```
SMALL_RECT STRUCT
    Left WORD ?
    Top WORD ?
    Right WORD ?
    Bottom 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 le 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          (Consola1.asm)
; Este programa llama a las siguientes funciones de Consola Win32:
; GetStdHandle, ExitProcess, WriteConsole
INCLUDE Irvine32.inc
.data
```

```

finl EQU <0dh,0ah> ; secuencia de fin de línea
mensaje LABEL BYTE
    BYTE "Este programa es una simple demostracion de"
    BYTE "la salida en modo de consola, usa las funciones"
    BYTE "GetStdHandle y WriteConsole.",finl
tamanoMensaje 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
        tamanoMensaje, ; 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 demostracion de la salida en modo de 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.

11.1.6 Lectura y escritura de archivos

Función CrearArchivo

La función **CrearArchivo** crea un nuevo archivo o abre uno existente. Si tiene éxito, devuelve un manejador para el archivo abierto; en caso contrario, devuelve una constante especial llamada **INVALID_HANDLE_VALUE**. He aquí el prototipo:

```

CreateFile PROTO,
    lpFileName:PTR BYTE, ; crea un nuevo archivo
    dwDesiredAccess:DWORD, ; apuntador al nombre de archivo
                           ; modo de acceso

```

```

dwShareMode:DWORD,           ; modo de compartición
lpSecurityAttributes:DWORD,   ; apuntador a los atributos de seguridad
dwCreationDisposition:DWORD,  ; opciones de creación de archivo
dwFlagsAndAttributes:DWORD,   ; atributos de archivo
hTemplateFile:DWORD          ; manejador al archivo de plantilla

```

Los parámetros se describen en la tabla 11-5. El valor de retorno es cero si la función falla.

Tabla 11-5 Parámetros de CreateFile.

Parámetro	Descripción
lpFileName	Apunta a una cadena con terminación nula, que contiene un nombre de archivo parcial o completamente calificado (<i>unidad\ruta\nombrearchivo</i>)
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 qué 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 dispositivo para el archivo. Puede seleccionar uno de los valores de la tabla 11-6, o de un conjunto extenso de valores específicos de bandera que no se presentan aquí (busque *CreateFile* en la documentación del SDK de la plataforma).

Tabla 11-6 Opciones del parámetro dwDesiredAccess.

Valor	Significado
0	Especifica el acceso de consulta de dispositivo para el objeto. Una aplicación puede consultar los atributos de un dispositivo sin acceder a éste, o puede comprobar la existencia de un archivo
GENERIC_READ	Especifica el acceso de lectura al objeto. Los datos pueden 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. Seleccione uno de los valores en la tabla 11-7.

La tabla 11-8 presenta los valores que se utilizan con más frecuencia y que están permitidos en el parámetro *dwFlagsAndAttributes* (para una lista completa, busque *CreateFile* en la documentación del SDK de la plataforma). 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 a 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_ATTRIBUTE_HIDDEN + FILE_ATTRIBUTE_READONLY

```

Tabla 11-7 Opciones del parámetro dwCreationDisposition.

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 nuevo archivo. Si ya existe, la función sobrescribe el archivo, borra los atributos existentes y combina los atributos de archivo y las banderas especificadas por el parámetro <i>attributes</i> con la constante predefinida FILE_ATTRIBUTE_ARCHIVE. Requiere establecer el parámetro dwDesiredAccess a GENERIC_WRITE
OPEN_EXISTING	Abre el archivo. La función falla si el archivo no existe. Puede usarse 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 <i>CreationDisposition</i> 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

Tabla 11-8 Valores seleccionados de FlagsAndAttributes.

Atributo	Significado
FILE_ATTRIBUTE_ARCHIVE	El archivo debe archivarse. Las aplicaciones utilizan este atributo 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 otros atributos establecidos. Este atributo es válido sólo 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 usando para almacenamiento temporal

Ejemplos Los siguientes ejemplos son sólo con fines ilustrativos, para mostrar cómo se pueden crear y abrir archivos. Consulte la documentación en línea de Microsoft MSDN sobre **CreateFile** para aprender acerca de las muchas opciones disponibles:

- Abrir un archivo existente para lectura (entrada):

```
INVOKE CreateFile,
    ADDR nombrearchivo,
    GENERIC_READ,
    DO_NOT_SHARE,
    NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL ,
    0
; apuntador al nombre de archivo
; lee del archivo
; modo de compartición
; apuntador a los atributos de seguridad
; abre un archivo existente
; atributo normal de archivo
; no se utiliza
```

- Abrir un archivo existente para escritura (salida). Una vez abierto el archivo, podríamos escribir sobre los datos existentes o adjuntarle nuevos datos, desplazando el puntero del archivo hasta el final (vea SetFilePointer, sección 11.1.6):

```
INVOKE CreateFile,
    ADDR nombrearchivo,
```

```

    GENERIC_WRITE,           ; escribe en el archivo
    DO_NOT_SHARE,
    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

```

Las constantes llamadas DO_NOT_SHARE y NULL se definen en el archivo de inclusión *SmallWin.inc*, que *Irvine32.inc* incluye de manera automática.

Función CloseHandle

La función **CloseHandle** cierra un manejador de objetos abierto. Su prototipo es:

```

CloseHandle PROTO,
    hObject:HANDLE           ; manejador para el objeto

```

Podemos usar a **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 asincrónica

```

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 en 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 en el archivo. **ReadFile** también puede ejecutarse en modo asincrónico, 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
```

Vea el programa *AdjuntarArchivo.asm*.

11.1.7 E/S de archivos en la biblioteca Irvine32

La biblioteca Irvine32 contiene algunos procedimientos simplificados para la entrada/salida con archivos, que documentamos en el capítulo 5. Los procedimientos son envolturas de las funciones de la API Win32 que hemos descrito en este capítulo. El siguiente código fuente lista a CreateOutputFile, OpenInputFile, WriteToFile, ReadFromFile y CloseFile:

```
; -----
CreateOutputFile PROC
;
; Crea un nuevo archivo y lo abre en modo de salida.
; Recibe: EDX apunta al nombre del archivo.
```

```
; Devuelve: Si el archivo se creó con éxito, EAX
;   contiene un manejador de archivo válido. En caso contrario, EAX
;   es igual a INVALID_HANDLE_VALUE.
;-----
        INVOKE CreateFile,
            edx, GENERIC_WRITE, DO_NOT_SHARE, NULL,
            CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0
        ret
CreateOutputFile ENDP

;-----
OpenInputFile PROC
;
; Abre un archivo existente en modo de entrada.
; Recibe: EDX apunta al nombre del archivo.
; Devuelve: Si el archivo se abrió con éxito, EAX
;   contiene un manejador de archivo válido. En caso contrario,
;   EAX es igual a INVALID_HANDLE_VALUE.
; Última actualización: 6/8/2005
;-----
        INVOKE CreateFile,
            edx, GENERIC_READ, DO_NOT_SHARE, NULL,
            OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0
        ret
OpenInputFile ENDP

;-----
WriteToFile PROC
;
; Escribe un búfer en un archivo de salida.
; Recibe: EAX = manejador del archivo, EDX = desplazamiento del búfer,
;         ECX = número de bytes a escribir
; Devuelve: EAX = número de bytes que se escriben en el archivo.
; Si el valor devuelto en EAX es menor que
; el argumento que se pasó en ECX, es probable que ocurrió un error.
; Última actualización: 6/8/2005
;-----
.data
WriteToFile_1 DWORD ?           ; número de bytes escritos
.code
        INVOKE WriteFile,
            eax,                      ; escribe el búfer en el archivo
            edx,                      ; manejador del archivo
            ecx,                      ; apuntador al búfer
            ADDR WriteToFile_1,       ; número de bytes a escribir
            0,                         ; número de bytes escritos
            0                          ; bandera de ejecución traslapada
        mov  eax,WriteToFile_1       ; valor de retorno
        ret
WriteToFile ENDP

;-----
ReadFromFile PROC
;
; Lee un archivo de entrada y lo coloca en un búfer.
; Recibe: EAX = manejador del archivo, EDX = desplazamiento del búfer,
;         ECX = número de bytes a leer
; Devuelve: Si CF = 0, EAX = número de bytes leídos; si
;           CF = 1, EAX contiene el código de error del sistema devuelto
```

```
;     por la función GetLastError de la API Win32.
; Última actualización: 7/6/2005
;-----
;.data
ReadFromFile_1 DWORD ?
.code
    INVOKE ReadFile,
        eax,                      ; manejador del archivo
        edx,                      ; apuntador al búfer
        ecx,                      ; máximo de bytes a leer
        ADDR ReadFromFile_1,      ; número de bytes leídos
        0                         ; bandera de ejecución traslapada
    mov    eax,ReadFromFile_1
    ret
ReadFromFile ENDP
;-----
CloseFile PROC
;
; Cierra un archivo, usando su manejador como identificador.
; Recibe: EAX = manejador de archivo
; Devuelve: EAX = distinto de cero si el archivo se cierra
;           con éxito.
; Última actualización: 6/8/2005
;-----
    INVOKE CloseHandle, eax
    ret
CloseFile ENDP
```

11.1.8 Prueba de los procedimientos de E/S de archivos

Ejemplo: programa CrearArchivo

El siguiente programa crea un archivo en modo de salida, pide al usuario que introduzca texto, lo escribe en el archivo de salida, reporta el número de bytes escritos y cierra el archivo. Comprueba errores después de tratar de crear el archivo:

```
TITLE Creación de un archivo          (CrearArchivo.asm)
INCLUDE Irvine32.inc
TAM_BUFER = 501
.data
bufer BYTE TAM_BUFER DUP(?)
nombrearchivo    BYTE "salida.txt",0
manejadorArchivo HANDLE ?
longitudCadena  DWORD ?
bytesEscritos   DWORD ?
cad1 BYTE "No se puede crear el archivo",0dh,0ah,0
cad2 BYTE "Bytes escritos en el archivo [salida.txt]:" ,0
cad3 BYTE "Escriba hasta 500 caracteres y oprima "
        BYTE "[Intro]",0dh,0ah,0

.code
main PROC
; Crea un nuevo archivo de texto.
    mov    edx,OFFSET nombrearchivo
    call   CreateOutputFile
    mov    manejadorArchivo,eax
; Comprueba errores.
```

```

        cmp    eax, INVALID_HANDLE_VALUE      ; ¿se encontró un error?
        jne    archivo_ok                  ; no: salta
        mov    edx,OFFSET cad1            ; muestra el error
        call   WriteString
        jmp    terminar

archivo_ok:
; Pide al usuario que introduzca una cadena.
        mov    edx,OFFSET cad3            ; "Escriba hasta ...."
        call  WriteString
        mov    ecx,TAM_BUFER           ; Recibe una cadena como entrada
        mov    edx,OFFSET bufer
        call  ReadString
        mov    longitudCadena,eax       ; cuenta los caracteres introducidos

; Escribe el búfer en el archivo de salida.
        mov    eax,manejadorArchivo
        mov    edx,OFFSET bufer
        mov    ecx,longitudCadena
        call  WriteToFile
        mov    bytesEscritos,eax         ; guarda el valor de retorno
        call  CloseFile

; Muestra el valor de retorno.
        mov    edx,OFFSET cad2            ; "Bytes escritos"
        call  WriteString
        mov    eax,bytesEscritos
        call  WriteDec
        call  Crlf

terminar:
        exit
main    ENDP
END     main

```

Ejemplo: programa LeerArchivo

El siguiente programa abre un archivo en modo de entrada, lee su contenido y lo coloca en un búfer, y muestra el búfer en pantalla. Todos los procedimientos se llaman de la biblioteca Irvine32:

```

TITLE Lectura de un archivo          (LeerArchivo.asm)

; Abre, lee y muestra un archivo de texto, usando
; los procedimientos de Irvine32.lib.

INCLUDE Irvine32.inc
INCLUDE macros.inc
TAM_BUFER = 5000

.data
bufer BYTE TAM_BUFER DUP(?)
nombreArchivo    BYTE 80 DUP(0)
manejadorArchivo HANDLE ?

.code
main PROC

; Permite que el usuario introduzca un nombre de archivo.
    mWrite "Escriba un nombre de archivo de entrada: "
    mov    edx,OFFSET nombreArchivo
    mov    ecx,SIZEOF nombreArchivo

```

```

call ReadString

; Abre el archivo en modo de entrada.
    mov edx,OFFSET nombreArchivo
    call OpenInputFile
    mov manejadorArchivo,eax
; Comprueba errores.
    cmp eax,INVALID_HANDLE_VALUE      ; ¿error al abrir el archivo?
    jne archivo_ok                  ; si: no: salta
    mWrite <"No se puede abrir el archivo",0dh,0ah>
    jmp terminar                   ; y termina

archivo_ok:

; Lee el archivo y lo coloca en un búfer.
    mov edx,OFFSET bufer
    mov ecx,TAM_BUFER
    call ReadFromFile
    jnc comprobar_tamanio_bufer   ; ¿error al leer?
    mWrite "Error al leer el archivo" ; sí: muestra mensaje de error
    call WriteWindowsMsg
    jmp cerrar_archivo

comprobar_tamanio_bufer:
    cmp eax,TAM_BUFER           ; ¿el búfer es lo bastante grande?
    jb tam_buf_ok                ; sí
    mWrite <"Error: Bufer demasiado chico para el archivo",0dh,0ah>
    jmp terminar                 ; y termina

tam_buf_ok:
    mov bufer[eax],0             ; inserta terminador nulo
    mWrite "Tamaño del archivo: "
    call WriteDec                 ; muestra el tamaño del archivo
    call Crlf

; Muestra el búfer.
    mWrite <"Bufer:",0dh,0ah,0dh,0ah>
    mov edx,OFFSET bufer
    call WriteString              ; muestra el búfer
    call Crlf

cerrar_archivo:
    mov eax,manejadorArchivo
    call CloseFile

terminar:
    exit
main ENDP
END main

```

El programa reporta un error si no puede abrirse el archivo:

Escriba un nombre de archivo de entrada: loco.txt
 No se puede abrir el archivo

Reporta un error si no se puede leer del archivo. Por ejemplo, suponga que un error en el programa utiliza el manejador de archivo incorrecto al leer el archivo:

Escriba un nombre de archivo de entrada: archent.txt
 No se puede abrir el archivo. Error 6: The handle is invalid.

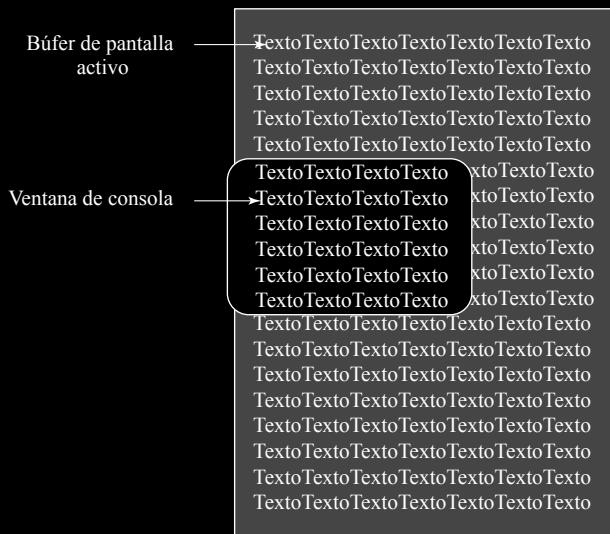
El búfer podría ser demasiado pequeño para guardar el archivo:

Escriba un nombre de archivo de entrada: archent.txt
Error: Bufer demasiado chico para el archivo

11.1.9 Manipulación de ventanas de consola

La API Win32 proporciona un control considerable sobre la ventana de consola y su búfer. La figura 11-1 muestra que el 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 una “mira”, mostrando parte del búfer.

FIGURA 11-1 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 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 la consola. *srWindow* devuelve las coordenadas de la ventana de consola relativa al búfer de pantalla. *drMaximumWindowSize* 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
```

La figura 11-2 presenta un ejemplo de los datos de la estructura que muestra el depurador de Microsoft Visual Studio.

FIGURA 11-2 Estructura CONSOLE_SCREEN_BUFFER_INFO.

Nombre	Valor	Tipo
infoConsola	{dwSize={...} dwCursorPosition={...} wAttributes=0 ...}	CONSOLE_SCREEN_BUFFER_INFO
dwSize	{X=0 Y=0 }	COORD
X	0	unsigned short
Y	0	unsigned short
dwCursorPosition	{X=0 Y=0 }	COORD
X	0	unsigned short
Y	0	unsigned short
wAttributes	0	unsigned short
srWindow	{Left=0 Top=0 Right=0 ...}	SMALL_RECT
Left	0	unsigned short
Top	0	unsigned short
Right	0	unsigned short
Bottom	0	unsigned short
dwMaximumWindowSize	{X=0 Y=0 }	COORD
X	0	unsigned short
Y	0	unsigned short

Función SetConsoleWindowInfo

La función **SetConsoleWindowInfo** nos permite establecer el tamaño y la 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 reposiciona la ventana de consola, logrando desplazar el texto hacia atrás. Utiliza la función *SetConsoleWindow*:

```

TITLE Desplazamiento de la ventana de consola          (Despl.asm)
INCLUDE Irvine32.inc

.data
mensaje BYTE ": Esta linea de texto se escribio "
        BYTE "en el bufer 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

```

Es mejor ejecutar este programa directamente desde el Explorador de Windows o desde un símbolo del sistema, en vez de hacerlo desde un entorno de edición integrado. En caso contrario, el editor podría afectar el

comportamiento y la apariencia de la ventana de consola. Debe oprimir una tecla dos veces al final: una vez para borrar el búfer de pantalla y otra para terminar el programa.

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 del búfer de pantalla
```

11.1.10 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 importante, 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 rellena en 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
```

11.1.11 Control del color de texto

Hay dos formas de controlar el color del texto en una ventana de consola. Podemos cambiar el color de texto actual llamando a **SetConsoleTextAttribute**, lo 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** (sección 11.1.9) 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 muestra en la sección 15.3.2.

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 16,17,18,19,20
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,
```

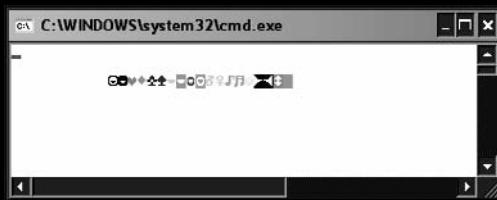
```

        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

```

La figura 11-3 muestra una captura de pantalla del resultado del programa, en donde los códigos de los caracteres del 1 al 20 se muestran como caracteres gráficos. Cada carácter está en un color distinto, aunque los colores no aparecen en la página impresa.

FIGURA 11-3 Resultado del programa EscribirColores.



11.1.12 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 podemos ver aquí un pequeño subconjunto de las funciones, pero puede consultar la documentación del SDK de la plataforma las funciones Win32 que se presentan en la tabla 11-9.

Estructura SYSTEMTIME La estructura SYSTEMTIME es utilizada por las funciones de la API de Windows relacionadas con la fecha y la hora:

```

SYSTEMTIME 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)
    wMilliseconds WORD ?   ; milisegundos (0-999)
SYSTEMTIME ENDS

```

El valor *wDayOfWeek* empieza con Domingo = 0, Lunes = 1, y así sucesivamente. El valor en *wMilliseconds* 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

```

Tabla 11-9 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 (<i>hora coordinada universal</i>) 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 periódicos a su reloj de la 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 la 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
SetSystemTimeAdjustment	Habilita o deshabilita los ajustes periódicos de la 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

Fuente: documentación del SDK de Windows en Microsoft MSDN.

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 continua durante 49.7 días. Puede 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 a 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 regresó 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 mas de 49.7 dias",0dh,0ah>
terminar:

    exit
main ENDP
END main
```

Función Sleep

Algunas veces los programas necesitan detenerse o retrasarse durante períodos breves. Aunque podríamos construir un ciclo de cálculo o un ciclo para mantener al procesador ocupado, el tiempo de ejecución de ese ciclo variaría 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,
    dwMilliseconds: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 en 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 como **GetLocalTime**, para que llene una estructura SYSTEMTIME.
2. Convertir la estructura SYSTEMTIME en una estructura FILETIME, llamando a 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, f1Time: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 f1Time
;
; Copia la estructura FILETIME en un entero de 64 bits.
    mov    esi,pDateTime
    mov    eax,f1Time.loDateTime
    mov    DWORD PTR [esi],eax
    mov    eax,f1Time.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 en la sección 7.5 para realizar operaciones aritméticas con fechas.

11.1.13 Repaso de sección

1. ¿Cuál es el comando del enlazador que especifica que el programa de destino es para la consola Win32?
2. (*Verdadero/Falso*): una función que termina con la letra W (como WriteConsoleW) está diseñada para trabajar con un conjunto de caracteres amplio (16 bits) como Unicode.
3. (*Verdadero/Falso*): unicode es el conjunto de caracteres nativo para Windows 98.
4. (*Verdadero/Falso*): la función **ReadConsole** lee la información del teclado del búfer de entrada.
5. (*Verdadero/Falso*): las funciones de entrada de consola Win32 pueden detectar cuando el usuario cambia el tamaño de la ventana de consola.
6. Mencione el tipo de datos de MASM que coincide con cada uno de los siguientes tipos estándar de MS-Windows:

BOOL
COLORREF
HANDLE
LPSTR
WPARAM

7. ¿Qué función Win32 devuelve un manejador para la entrada estándar?
8. ¿Qué función Win32 devuelve una cadena de texto del teclado y la coloca en un búfer?
9. Muestre una llamada de ejemplo a la función **ReadConsole**.
10. Describa la estructura COORD.
11. Muestre una llamada de ejemplo a la función **WriteConsole**.
12. Muestre una llamada de ejemplo a la función **CreateFile** para abrir un archivo existente en modo de lectura.
13. Muestre una llamada de ejemplo a la función **CreateFile** para crear un nuevo archivo con atributos normales, y que borre cualquier archivo existente que tenga el mismo nombre.
14. Muestre una llamada de ejemplo a la función **ReadFile**.
15. Muestre una llamada de ejemplo a la función **WriteFile**.
16. ¿Qué función Win32 mueve el apuntador del archivo a un desplazamiento especificado, relacionado con el principio de un archivo?
17. ¿Qué función Win32 cambia el título de la ventana de consola?
18. ¿Qué función Win32 nos permite cambiar las medidas del búfer de pantalla?
19. ¿Qué función Win32 nos permite cambiar el tamaño del cursor?
20. ¿Qué función Win32 nos permite cambiar el color de la salida de texto subsiguiente?
21. ¿Qué función Win32 nos permite copiar un arreglo de valores de atributos a celdas consecutivas del búfer de pantalla de la consola?
22. ¿Qué función Win32 nos permite detener un programa durante un número especificado de milisegundos?

11.2 Escritura de una aplicación gráfica de Windows

En esta sección le mostraremos cómo escribir una aplicación gráfica simple para Microsoft Windows. El programa crea y muestra una ventana principal, presenta cuadros de mensaje y responde a los eventos del ratón. La información que proporcionamos aquí sólo es una breve introducción; se requeriría cuando menos todo un capítulo completo para describir el funcionamiento de incluso la aplicación MS Windows más simple. Si desea más información, consulte la documentación del SDK de la plataforma. Otra excelente fuente de información es el libro de Charles Petzold, *Programming in Windows: The Definitive Guide to the Win32 API*.

La tabla 11-10 presenta las diversas bibliotecas y archivos de inclusión que utilizamos al crear este programa. Use el archivo de proyecto de Visual Studio, ubicado en la carpeta Ejemplos\Cap11\WinApp del libro, para generar y ejecutar el programa.

/SUBSYSTEM:WINDOWS sustituye a /SUBSYSTEM:CONSOLE, que utilizamos en capítulos anteriores. El programa llama a las funciones de dos bibliotecas estándar de MS Windows: kernel32.lib y user32.lib.

Tabla 11-10 Archivos requeridos para generar el programa WinApp.

Nombre de archivo	Descripción
WinApp.asm	Código 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 antes en este capítulo
user32.lib	Funciones adicionales de la API de MS Windows

Ventana principal El programa muestra una ventana principal que llena la pantalla. Aquí redujimos su tamaño para adaptarla a la página impresa (figura 11-4).

FIGURA 11-4 Ventana principal inicial, programa WinApp.



11.2.1 Estructuras necesarias

La estructura **POINT** especifica las coordenadas X y Y de un punto en la pantalla, medido en píxeles. Por ejemplo, puede usarse para localizar objetos de gráficos, ventanas y clics del ratón:

```
POINT STRUCT
    ptX  DWORD ?
    ptY  DWORD ?
POINT 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 **right** y **bottom** se almacenan valores similares:

```
RECT 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 **WNDCLASS** 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 STRUCT
  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 ícono 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.

11.2.2 La función MessageBox

La manera más fácil para que un programa muestre texto es colocarlo en un cuadro de mensaje que aparezca y espere 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 cuadro. *lpCaption* apunta a una cadena con terminación nula que aparecerá en la barra de leyenda del cuadro. *style* es un entero que describe tanto el ícono (opcional) como los botones (requeridos) del cuadro de diálogo. Los botones se identifican mediante constantes como MB_OK y MB_YESNO. Los íconos también se identifican mediante constantes como MB_ICONQUESTION. Al mostrar un cuadro de mensaje, podemos sumar las constantes para el ícono y los botones:

```

INVOKE MessageBox, hWnd, ADDR TextoPregunta,
                  ADDR TituloPregunta, MB_OK + MB_ICONQUESTION

```

11.2.3 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 el 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. Para leer más acerca de los mensajes, busque *Mensajes Windows (Windows Messages)* en la documentación del SDK de la plataforma.

11.2.4 El procedimiento WinProc

El procedimiento **WinProc** recibe y procesa todos los mensajes de eventos relacionados con una ventana. La mayoría de los eventos los inicia 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
    lParam: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:

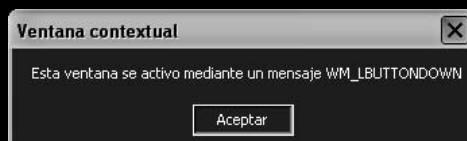
- WM_LBUTTONDOWN, que se genera cuando el usuario oprime el botón izquierdo del ratón.
- WM_CREATE, que indica que se acaba de crear la ventana principal.
- WM_CLOSE; que indica que la ventana principal de la aplicación está a punto de cerrar.

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 TerminaWinProc
```

El mensaje resultante que ve el usuario se muestra en la figura 11-5. Cualquier otro mensaje que no deseamos manejar se pasa a **DefWindowProc**, el manejador de mensajes predeterminado para MS Windows.

FIGURA 11-5 Ventana contextual, programa WinApp.



11.2.5 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 la memoria utilizada por la cadena de mensaje de error.

11.2.6 Listado del programa

No se asuste por el tamaño de este 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 mensaje contextuales.
; Gracias a Tom Joyce por crear un prototipo
; a partir del cual se derivó este programa.
.386
.model flat, STDCALL
INCLUDE GraphWin.inc

;===== DATOS =====
.data

TituloMsjCargaApp BYTE "Se cargo la aplicacion",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 recibio 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 ícono 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 TerminaWinProc
        .ELSEIF eax == WM_CREATE           ; ¿crear ventana?
            INVOKE MessageBox, hWnd, ADDR TextoMsjCargaApp,
                  ADDR TituloMsjCargaApp, MB_OK
            jmp TerminaWinProc
        .ELSEIF eax == WM_CLOSE            ; ¿cerrar ventana?
            INVOKE MessageBox, hWnd, ADDR MsjCerrar,
                  ADDR NombreVentana, MB_OK
            INVOKE PostQuitMessage,0
            jmp TerminaWinProc
        .ELSE                                ; ¿otro mensaje?
            INVOKE DefWindowProc, hWnd, localMsg, wParam, lParam
            jmp TerminaWinProc
        .ENDIF
TerminaWinProc:
        ret
WinProc ENDP
;-----
ErrorHandler PROC
; Muestra el mensaje de error apropiado del sistema.
;-----
.data
pMsjError  DWORD ?                      ; apuntador al mensaje de error
IDmensaje   DWORD ?                     ; apuntador al ID de error
.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, IDmensaje, NULL,
              ADDR pMsjError, 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

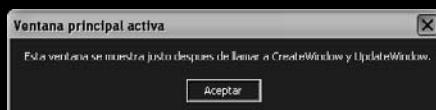
```

Ejecución del programa

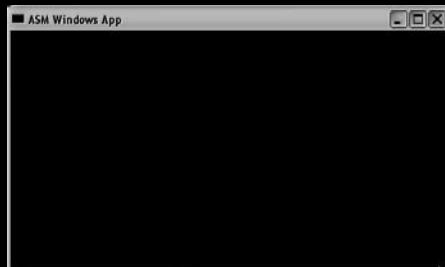
Cuando el programa se carga por primera vez, aparece el siguiente cuadro de mensaje:



Cuando el usuario hace clic en Aceptar para cerrar el cuadro de mensaje **Se cargo la aplicación**, aparece otro cuadro de mensaje:



Cuando el usuario cierra el cuadro de mensaje **Ventana principal activa**, aparece la ventana principal del programa:



Cuando el usuario hace clic con el ratón en cualquier parte de la ventana principal, aparece el siguiente cuadro de mensaje:



Cuando el usuario cierra este cuadro de mensaje y después hace clic en la X de la esquina superior derecha de la ventana principal, aparece el siguiente mensaje justo después de que se cierra la ventana:



Cuando el usuario cierra este cuadro de mensaje, el programa termina.

11.2.7 Repaso de sección

1. Describa una estructura **POINT**.
2. ¿Cómo se utiliza la estructura **WNDCLASS**?
3. En una estructura **WNDCLASS**, ¿cuál es el significado del campo *lpfnWndProc*?
4. En una estructura **WNDCLASS**, ¿cuál es el significado del campo *style*?
5. En una estructura **WNDCLASS**, ¿cuál es el significado del campo *hInstance*?
6. Cuando se hace una llamada a **CreateWindowEx**, ¿cómo se transmite la información de la apariencia de la ventana a la función?

7. Muestre un ejemplo de una llamada a la función **MessageBox**.
8. Mencione dos constantes de botones que pueden usarse al llamar a la función **MessageBox**.
9. Mencione dos constantes de iconos que pueden usarse al llamar a la función **MessageBox**.
10. Mencione cuando menos tres tareas que realiza el procedimiento **WinMain** (inicio).
11. Describa la función del procedimiento **WinProc** en el programa de ejemplo.
12. ¿Qué mensajes procesa el procedimiento **WinProc** en el programa de ejemplo?
13. Describa la función del procedimiento **ErrorHandler** en el programa de ejemplo.
14. ¿El cuadro de mensaje que se activa justo después de llamar a **CreateWindow** aparece antes o después de la ventana principal de la aplicación?
15. ¿El cuadro de mensaje que se activa mediante **WM_CLOSE** aparece antes o después de que se cierra la ventana principal?

11.3 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 tamano;  
cin >> tamano;                                // el usuario introduce el tamaño  
int arreglo[] = new int[tamano];
```

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 del montón de datos 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 lo bastante grande como para otorgar la petición.

Los programas en lenguaje ensamblador pueden realizar la asignación dinámica de dos maneras. En primer lugar, pueden realizar llamadas al sistema para obtener bloques de memoria del sistema operativo. En segundo lugar, pueden implementar sus propios administradores del montón de datos, que atiendan las peticiones para los objetos más pequeños. En esta sección le mostraremos cómo 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 en la tabla 11-11. 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. Para aprender más acerca de la administración de memoria, busque *Referencia de administración de memoria (Memory Management Reference)* en la documentación del SDK de la plataforma.

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
```

Tabla 11-11 Funciones relacionadas con el montón de datos.

Función	Descripción
GetProcessHeap	Devuelve en EAX un manejador de enteros de 32 bits al área del montón de datos existente 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 de 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 el tamaño de un bloque de memoria de un montón de pila. Si la función tiene éxito, el valor de retorno es un apuntador al 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 reasignado, 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)

Llamada de ejemplo:

```
.data
hMonton HANDLE ?
.code
INVOKE GetProcessHelp
.IF eax == NULL           ; no puede obtener el manejador
    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
    dwMaximumSize: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 el tamaño inicial del montón de datos, éste crece hasta el valor que se especifica en el parámetro *dwMaximumSize* (redondeado 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 de ejemplo a HeapCreate:

```
HEAP_START = 2000000          ; 2 MB
HEAP_MAX = 400000000         ; 400 MB
.data
hMonton = HANDLE ?           ; manejador para el montón
```

```
.code
INVOKE HeapCreate, 0, HEAP_START, 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 puede destruir el montón de datos, EAX es igual a NULL. A continuación se muestra una llamada de ejemplo, usando el procedimiento WriteWindowsMsg descrito en la sección 11.1.4:

```
.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 existente:

```
HeapAlloc PROTO,
    hHandle:HANDLE, ; manejador para el bloque del montón privado
    dwFlags:DWORD, ; banderas de control de asignación del 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.
- *dwFlags*, una doble palabra que contiene uno o más valores de bandera. Podemos establecerla 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 antes 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 una llamada de ejemplo:

```
INVOKE HeapFree, hHeap, 0, pArray
```

Manejo de errores Si encuentra un error al llamar a HeapCreate, HeapDestroy o GetProcessHeap, puede obtener los detalles llamando a la función **GetLastError** de la API. O puede llamar a la función **WriteWindowsMsg** de la biblioteca Irvine32. A continuación se muestra un ejemplo que llama 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 del sistema cuando falla, por lo que no podemos llamar a GetLastError o a WriteWindowsMsg.

11.3.1 Programas PruebaMonton

El siguiente ejemplo (*PruebaMonton1.asm*) utiliza la asignación dinámica de memoria para crear y llenar un arreglo de 1000 bytes:

```
Title Prueba del Montón #1                                (PruebaMonton1.asm)
INCLUDE Irvine32.inc

; Este programa utiliza la asignación dinámica de memoria para
; asignar y llenar un arreglo de bytes.

.data
TAM_ARREGLO = 1000
VAL_LLENAR EQU 0FFh

hMonton    DWORD ?           ; manejador para el montón de datos del proceso
pArreglo   DWORD ?           ; apuntador al bloque de memoria
nuevoMonton DWORD ?          ; manejador para el montón nuevo
cad1 BYTE "El tamaño del montón es: ",0

.code
main PROC
    INVOKE GetProcessHeap           ; obtiene el manejador para el montón del programa
    .IF eax == NULL                ; ¿falló?
        call WriteWindowsMsg
        jmp terminar
    .ELSE
        mov hMonton,eax            ; éxito
    .ENDIF

    call asignar_arreglo
    jnc arregloOk                 ; ¿falló (CF = 1)?
    call WriteWindowsMsg
    call CrLf
    jmp terminar

arregloOk:                      ; se puede llenar el arreglo
```

```
call llenar_arreglo
call mostrar_arreglo
call Crlf

; libera el arreglo
INVOKE HeapFree, hMonton, 0, pArreglo

terminar:

exit
main ENDP

;-----  
asignar_arreglo PROC USES eax
;  
; Asigna espacio para el arreglo en forma dinámica.  
; Recibe: nada  
; Devuelve: CF = 0 si la asignación tiene éxito.  
;  
INVOKE HeapAlloc, hMonton, HEAP_ZERO_MEMORY, TAM_ARREGLO
.IF eax == NULL
    stc                         ; regresa con CF = 1
.ELSE
    mov  pArreglo,eax          ; guarda el apuntador
    clc                         ; regresa con CF = 0
.ENDIF

ret
asignar_arreglo ENDP

;-----  
llenar_arreglo PROC USES ecx edx esi
;  
; Llena todas las posiciones del arreglo con un solo carácter.  
; Recibe: nada  
; Devuelve: nada  
;  
    mov  ecx,TAM_ARREGLO        ; contador del ciclo
    mov  esi,pArreglo           ; apuntador al arreglo
L1:   mov  BYTE PTR [esi],VAL_LLENAR      ; llena cada byte
    inc  esi                     ; siguiente ubicación
    loop L1

ret
llenar_arreglo ENDP

;-----  
mostrar_arreglo PROC USES eax ebx ecx esi
;  
; Muestra el arreglo
; Recibe: nada
; Devuelve: nada
;  
    mov  ecx,TAM_ARREGLO        ; contador del ciclo
    mov  esi,pArreglo           ; apunta al arreglo
L1:   mov  al,[esi]                 ; obtiene un byte
    mov  ebx,TYPE BYTE          ; lo muestra
    call WriteHexB
```

```

    inc  esi           ; siguiente ubicación
Loop  L1

    ret
mostrar_arreglo ENDP
END main

```

El siguiente ejemplo (*PruebaMonton2.asm*) utiliza la asignación dinámica de memoria para crear y llenar un arreglo de 1000 bytes:

```

Title Prueba del montón #2          (PruebaMonton2.asm)

INCLUDE Irvine32.inc

; Crea un montón y asigna varios bloques de memoria,
; expandiendo el montón hasta que falle.

.data
INICIO_MONTON = 2000000          ; 2 MB
MAX_MONTON = 400000000          ; 400 MB
TAM_BLOQUE = 500000             ; .5 MB

hMonton DWORD ?                ; manejador para el montón
pDatos DWORD ?                ; apuntador al bloque

cad1 BYTE 0dh,0ah,"Fallo la asignacion de memoria",0dh,0ah,0

.code
main PROC
    INVOKE HeapCreate, 0,INICIO_MONTON, MAX_MONTON

    .IF eax == NULL              ; ¿falló?
    call WriteWindowsMsg
    call Crlf
    jmp terminar
    .ELSE
    mov hMonton,eax            ; éxito
    .ENDIF

    mov ecx,2000                ; contador del ciclo

L1: call asignar_bloque        ; asigna un bloque
    .IF Carry?                ; ¿falló?
    mov edx,OFFSET cad1        ; muestra mensaje
    call WriteString
    jmp terminar
    .ELSE                      ; no: imprime un punto para
    mov al,'.'                 ; mostrar el progreso
    call WriteChar
    .ENDIF

;call liberar_bloque          ; habilita/deshabilita esta línea
loop L1

terminar:
    INVOKE HeapDestroy, hMonton ; destruye el montón
    .IF eax == NULL              ; ¿falló?
    call WriteWindowsMsg        ; sí: mensaje de error
    call Crlf
    .ENDIF

exit

```

```
main ENDP
asignar_bloque PROC USES ecx
    ; asigna un bloque y lo llena con ceros
    INVOKE HeapAlloc, hMonton, HEAP_ZERO_MEMORY, TAM_BLOQUE
    .IF eax == NULL
        stc
    ; regresa con CF = 1
    .ELSE
        mov pDatos, eax
        clc
    ; guarda el apuntador
    ; regresa con CF = 0
    .ENDIF
    ret
asignar_bloque ENDP
liberar_bloque PROC USES ecx
    INVOKE HeapFree, hMonton, 0, pDatos
    ret
liberar_bloque ENDP
END main
```

11.3.2 Repaso de sección

1. ¿Qué otro término existe para *asignación del montón de datos*, dentro del contexto de C, C++ y Java?
2. Describa la función GetProcessHeap.
3. Describa la función HeapAlloc.
4. Muestre una llamada de ejemplo a la función HeapCreate.
5. Al llamar a HeapDestroy, ¿cómo podemos identificar el bloque de memoria que se va a destruir?

11.4 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 en el capítulo 2, 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 o GS).
- Una *dirección lógica* es una combinación de un selector de segmento y un desplazamiento de 32 bits.

A lo largo de este libro hemos ignorado los registros de segmento, ya que los programas de usuario nunca los modifican en forma directa. Nos hemos enfocado sólo en los desplazamientos de datos de 32 bits. Sin embargo, desde el punto de vista del programador, los registros de segmento son importantes ya que contienen referencias indirectas a segmentos de memoria.

11.4.1 Direcciones lineales

Traducción de direcciones lógicas a direcciones lineales

Un sistema operativo multitareas permite que varios programas (tareas) se ejecuten en memoria al mismo tiempo. Cada programa tiene su propia área para datos. Suponga que cada uno de tres programas tiene una variable en el desplazamiento 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. En la sección 11.4.2 explicaremos el proceso de traducción de páginas.

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 (en 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 se genera una *dirección lineal* de 32 bits, como se muestra en la figura 11-6.

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.

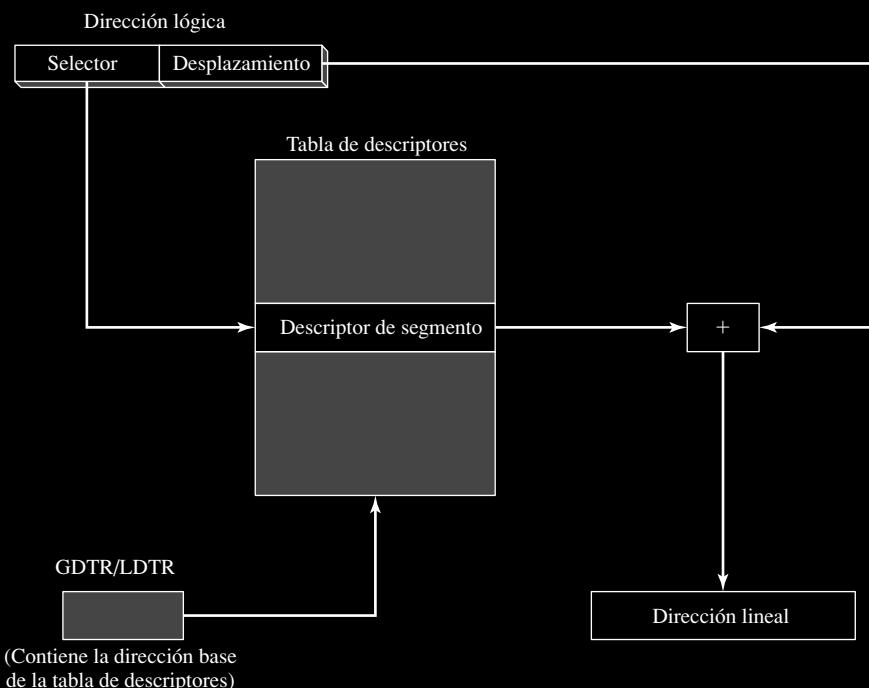
Paginación

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 la memoria. Para ello, el procesador carga al inicio sólo una parte de un programa en la 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 un conjunto de *tablas de páginas* para llevar la cuenta de las páginas utilizadas por todos los programas que se encuentran 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 el programa y genera un *fallo de página*. El sistema operativo copia la página requerida del disco a la memoria antes de que el programa pueda continuar. Desde el punto de vista de un programa de aplicación, los fallos de página y la traducción de páginas ocurren en forma automática.

Por ejemplo, en Windows 2000 podemos activar una herramienta llamada *Administrador de tareas* y ver la diferencia entre la memoria física y la memoria virtual. La figura 11-7 muestra una computadora con 1 GB de memoria física. La cantidad total de memoria virtual actualmente en uso es el área identificada como *Carga de transacciones* del Administrador de tareas. El límite de memoria virtual es de 633MB, considerablemente mucho mayor que el tamaño de memoria física de la computadora.

FIGURA 11-6 Conversión de una dirección lógica en una dirección lineal.



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 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 multitareas, a cada tarea o programa, por lo general, se le asigna su propia tabla de descriptores de segmentos, 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, como en la figura 11-8. Se muestran tres direcciones lógicas distintas, cada una de las cuales selecciona una entrada distinta en la LDT. En esta figura suponemos que la paginación está deshabilitada, por lo que el espacio de direcciones lineales también es el espacio de direcciones físicas.

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 4GB.

FIGURA 11-7 Ejemplo del Administrador de tareas de Windows.

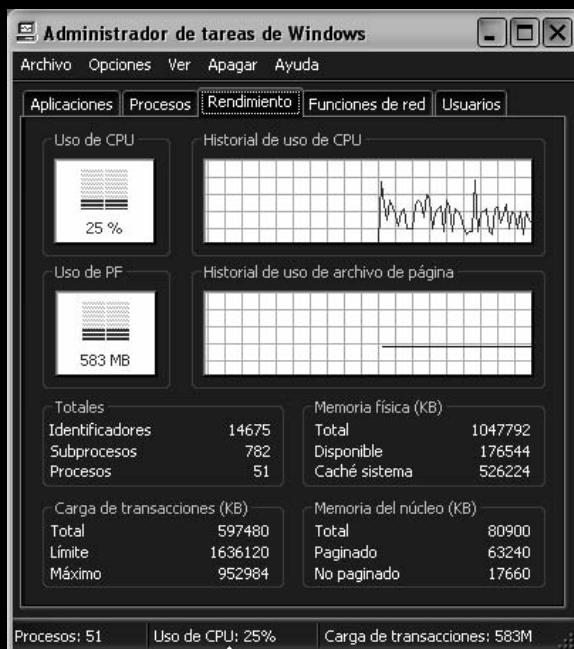
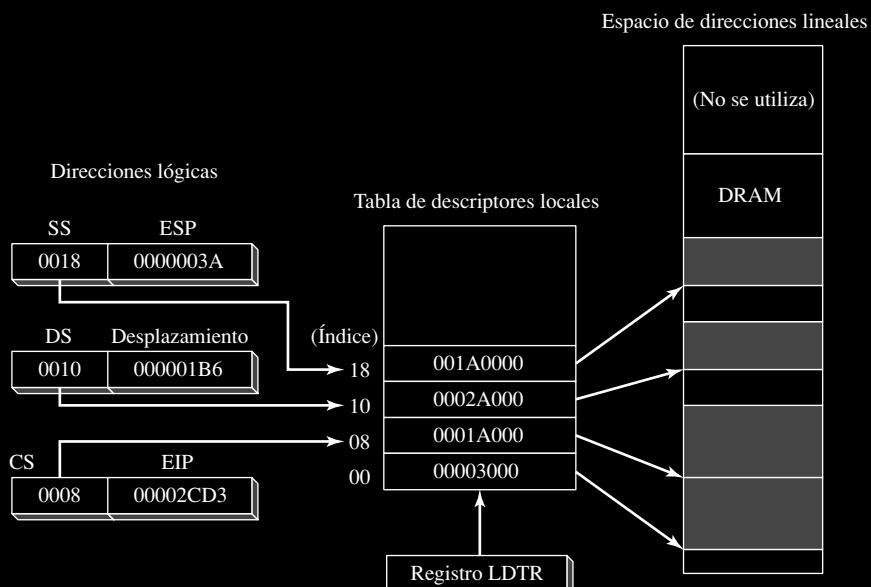


FIGURA 11-8 Indexamiento en una tabla de descriptores locales.



Nivel de privilegios: a cada segmento se le puede asignar un nivel de privilegios entre 0 y 3, en donde 0 es el que tiene más privilegio, por lo general, para el código del 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 pueden 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 está en cero, el límite del segmento se interpreta en unidades de bytes. Si el bit está activo, el límite del 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 dos formas, dependiendo de la bandera Granularidad:

- El número de bytes en el segmento, que 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.

11.4.2 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.² Se utilizan tres estructuras en el proceso:

- Directorio de página: un arreglo de hasta 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 en una dirección física, como se muestra en la figura 11-9:

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.

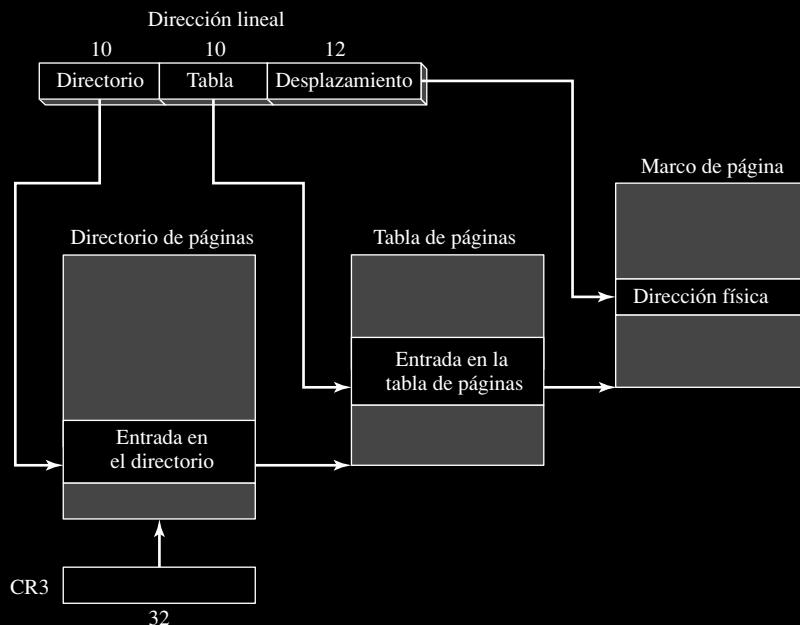
El sistema operativo tiene la opción de utilizar un solo directorio de páginas para todos los programas y tareas en ejecución, o un directorio de páginas por tarea, o una combinación de ambos.

Administrador de máquina virtual de MS Windows

Ahora que tenemos una idea general de la forma en que la familia IA-32 administra la memoria, podría ser interesante ver cómo administra MS Windows la memoria. El siguiente pasaje de texto proviene de la documentación del SDK de la plataforma:

El Administrador de máquina virtual (VMM) es el sistema operativo en modo protegido de 32 bits en el núcleo de MS Windows. Crea, ejecuta, monitorea y termina la ejecución de las máquinas virtuales. Administra la memoria, los procesos, las interrupciones y las excepciones. Trabaja con *dispositivos virtuales*, permitiéndoles interceptar las interrupciones y los fallos que controlan el acceso al hardware y al software instalado. La VMM y los dispositivos virtuales se ejecutan en un solo espacio de direcciones de modelo plano de 32 bits, en el nivel de privilegio 0. El sistema crea dos entradas en la tabla de descriptores globales (descriptores de segmento), una para el código y otra para los datos. Los segmentos son fijos en la dirección lineal 0. La VMM proporciona multitarea preferente con subprocesamiento múltiple. Ejecuta varias aplicaciones en forma simultánea, compartiendo el tiempo de la CPU entre las máquinas virtuales en las que se ejecutan las aplicaciones.

FIGURA 11-9 Traducción de una dirección lineal a una dirección física.



En el pasaje de texto anterior, podemos interpretar el término *máquina virtual* como lo que Intel llama un *proceso* o *tarea*. Consiste de código de programa, software de soporte, memoria y registros. A cada máquina virtual se le asigna su propio espacio de direcciones, espacio de puertos de E/S, tabla de vectores de interrupciones y tabla de descriptores globales. Las aplicaciones que se ejecutan en modo 8086 virtual se ejecutan en el nivel de privilegios 3. En MS Windows, los programas en modo protegido se ejecutan en los niveles de privilegios 0 y 3.

11.4.3 Repaso de sección

1. Defina los siguientes términos:
 - a. Multitareas.
 - b. Segmentación.
2. Defina los siguientes términos:
 - a. Selector de segmento.
 - b. Dirección lógica.
3. (*Verdadero/Falso*): un selector de segmento apunta a una entrada en una tabla de descriptores de segmento.
4. (*Verdadero/Falso*): un descriptor de segmento contiene la ubicación base de un segmento.
5. (*Verdadero/Falso*): un selector de segmento es de 32 bits.
6. (*Verdadero/Falso*): un descriptor de segmento no contiene información sobre el tamaño del segmento.
7. Describa una dirección lineal.
8. ¿Cómo se relaciona la paginación con la memoria lineal?
9. Si la paginación está deshabilitada, ¿cómo traduce el procesador una dirección lineal en una dirección física?
10. ¿Qué ventaja ofrece la paginación?
11. ¿Qué registro contiene la ubicación base de una tabla de descriptores globales?
12. ¿Qué registro contiene la ubicación base de una tabla de descriptores locales?
13. ¿Cuántas tablas de descriptores globales pueden existir?
14. ¿Cuántas tablas de descriptores locales pueden existir?

15. Mencione cuando menos cuatro campos en un descriptor de segmento.
16. ¿Qué estructuras están involucradas en el proceso de paginación?
17. ¿Qué estructura contiene la dirección base de una tabla de páginas?
18. ¿Qué estructura contiene la dirección base de un marco de página?

11.5 Resumen del capítulo

En la superficie, los programas en modo de consola de 32 bits se ven y se comportan como los programas MS-DOS de 16 bits que se ejecutan en modo de texto. Ambos tipos de programas leen de la entrada estándar y escriben en la salida estándar, soportan la redirección de la línea de comandos y pueden mostrar texto a color. No obstante, debajo de la superficie, los programas de consola Win32 y MS-DOS son bastante diferentes. Win32 se ejecuta en modo protegido de 32 bits, mientras que MS-DOS se ejecuta en modo de direccionamiento real. Los programas Win32 pueden llamar a las funciones de la misma biblioteca de funciones que utilizan las aplicaciones Windows gráficas. Los programas de MS-DOS están limitados a un conjunto más pequeño de interrupciones del BIOS y de MS-DOS, que han existido desde que se introdujo la IBM-PC.

En las funciones de la API de Windows se utilizan dos tipos de conjuntos de caracteres: el conjunto de caracteres ASCII/ANSI de 8 bits y una versión de 16 bits del conjunto de caracteres Unicode.

Los tipos de datos estándar de MS Windows que se utilizan en las funciones de la API deben traducirse en tipos de datos MASM (vea la tabla 11-1).

Los manejadores de consola son enteros de 32 bits que se utilizan para la entrada/salida en ventanas de consola. La función **GetStdHandle** obtiene un manejador de consola. Para la entrada de consola de alto nivel, llame a la función **ReadConsole**; para la salida de alto nivel, llame a **WriteConsole**. Al crear o abrir un archivo, llame a **CreateFile**. Al leer de un archivo, llame a **ReadFile**, y al escribir, a **WriteFile**. **CloseHandle** cierra un archivo. Para mover un apuntador de archivo, llame a **SetFilePointer**.

Para manipular el búfer de pantalla de consola, llame a **SetConsoleScreenBufferSize**. Para cambiar el color de texto, llame a **SetConsoleTextAttribute**. El programa EscribirColores en este capítulo demostró las funciones **WriteConsoleOutputAttribute** y **WriteConsoleOutputCharacter**.

Para obtener la hora del sistema, llame a **GetLocalTime**; para establecer la hora, llame a **SetLocalTime**. Ambas funciones utilizan la estructura **SYSTEMTIME**. El ejemplo de la función **GetDateTime** en este capítulo devuelve la fecha y hora como un entero de 64 bits, el cual especifica el número de intervalos de 100 nanosegundos que han transcurrido desde enero 1, 1601. Las funciones **TimerStart** y **TimerStop** pueden usarse para crear un cronómetro simple.

Al crear una aplicación gráfica de MS Windows, se llena una estructura **WNDCLASS** con información acerca de la clase de ventana principal del programa. Se crea un procedimiento **WinMain** que obtiene un manejador para el proceso actual, carga el ícono y el cursor del ratón, registra la ventana principal del programa, crea la ventana principal, muestra y actualiza las ventanas principales, y empieza un ciclo para recibir y despachar los mensajes.

El procedimiento **WinProc** es responsable de manejar los mensajes entrantes de Windows, que a menudo se activan mediante las acciones del usuario, como un clic del ratón o un tecleo. Nuestro programa de ejemplo procesa un mensaje **WM_LBUTTONDOWN**, un mensaje **WM_CREATE** y un mensaje **WM_CLOSE**. Muestra mensajes contextuales cuando se detectan estos eventos.

La asignación dinámica de memoria, o asignación del montón de datos, es una herramienta que podemos usar para reservar la memoria y liberarla para que nuestro programa la utilice. Los programas en lenguaje ensamblador pueden realizar una asignación dinámica de dos formas. En primer lugar, pueden realizar llamadas al sistema para obtener bloques de memoria del sistema operativo. En segundo lugar, pueden implementar sus propios administradores del montón de datos, para atender las peticiones de objetos más pequeños. A continuación se muestran las llamadas más importantes a la API Win32 para la asignación dinámica de memoria:

- **GetProcessHeap** devuelve un manejador entero de 32 bits para el área del montón de datos existente del programa.

- HeapAlloc asigna un bloque de memoria de un montón de datos.
- HeapCreate crea un nuevo montón de datos.
- HeapDestroy destruye un montón de datos.
- HeapFree libera un bloque de memoria de un montón de datos que antes estaba asignado.
- HeapReAlloc reasigna y cambia el tamaño de un bloque de memoria de un montón de datos.
- HeapSize devuelve el tamaño de un bloque de memoria previamente asignado.

La sección de administración de memoria de este capítulo se enfoca en dos temas principales: traducir las direcciones lógicas en direcciones lineales, y traducir las direcciones lineales en direcciones físicas.

El selector en una dirección lógica apunta a una entrada en una tabla de descriptores de segmento, que a su vez apunta a un segmento en la memoria lineal. El descriptor de segmento contiene información acerca del segmento, incluyendo su tamaño y el tipo de acceso. Hay dos tipos de tablas de descriptores: una sola tabla de descriptores globales (GDT) y una o más tablas de descriptores locales (LDT).

La paginación es una importante característica del procesador IA-32, que hace posible que una computadora ejecute una combinación de programas, que de otra forma no cabrían en la memoria. Para ello, el procesador carga al principio sólo una parte de un programa en la memoria, y mantiene el resto de las partes en el disco. El procesador utiliza un directorio de páginas, una tabla de páginas y un marco de páginas para generar la ubicación física de los datos. Un directorio de páginas contiene apuntadores a las tablas de páginas. Una tabla de páginas contiene apuntadores a las páginas.

Lectura Para leer más acerca de la programación en Windows, los siguientes libros le pueden ser de ayuda:

- Mark Russinovich y David Solomon, *Microsoft Windows Internals 4^a Edición.*, Microsoft Press, 2004.
- Barry Kauler, *Windows Assembly Language and System Programming*, CMP Books, 1997.
- Charles Petzold, *Programming Windows, 5^a Edición.*, Microsoft Press, 1998.

11.6 Ejercicios de programación

1. ReadString

Implemente su propia versión del procedimiento **ReadString**; utilice parámetros de pila. Debe recibir un apuntador a una cadena y un entero que indique el número máximo de caracteres a introducir. Debe devolver la cuenta (en EAX) del número de caracteres que se introdujeron. El procedimiento debe recibir como entrada una cadena de la consola e insertar un byte nulo al final de la misma (en la posición ocupada por 0Dh). Consulte la sección 11.1.4 para los detalles acerca de la función **ReadConsole** de Win32. Escriba un programa corto para probar su procedimiento.

2. Entrada/salida con cadenas

Escriba un programa que reciba como entrada la siguiente información del usuario, usando la función **ReadConsole** de Win32: primer nombre, apellido paterno, edad, número telefónico. Vuelva a mostrar la misma información con etiquetas y un formato atractivo, usando la función **WriteConsole** de Win32. No utilice procedimientos de la biblioteca Irvine32.

3. Limpiar la pantalla

Escriba su propia versión del procedimiento **Clscr** de la biblioteca para borrar la pantalla.

4. Relleno de pantalla aleatorio

Escriba un programa para llenar cada celda de la pantalla con un carácter aleatorio, en un color aleatorio. *Extra:* asigne una probabilidad del 50% de que el color de cualquier carácter sea rojo.

5. DibujarCuadro

Dibuje un cuadro en la pantalla, usando caracteres para dibujar líneas del conjunto de caracteres que se presenta en la solapa de la contraportada de este libro. *Sugerencia:* use la función **WriteConsoleOutputCharacter**.

6. Registros de estudiantes

Escriba un programa para crear un nuevo archivo de texto. Pida al usuario un número de identificación de estudiante, apellido paterno, primer nombre y fecha de nacimiento. Escriba esta información en el archivo. Reciba como entrada varios registros más de la misma manera y cierre el archivo.

7. Desplazamiento de la ventana de texto

Diseñe un programa para escribir 50 líneas de texto en el búfer de pantalla de consola. Numere cada línea. Mueva la ventana de consola a la parte superior del búfer, y comience a desplazar el texto hacia arriba, a una velocidad estable (dos líneas por segundo). Deje de desplazar la ventana de consola cuando ésta llegue al final del búfer.

8. Animación de bloques

Escriba un programa para dibujar un pequeño cuadro en la pantalla, usando varios bloques (código ASCII DBh) a color. Mueva el cuadro alrededor de la pantalla en direcciones generadas al azar. Use un valor de retraso fijo de 50 milisegundos. *Extra:* use un valor de retraso generado al azar, entre 10 y 100 milisegundos.

9. Fecha del último acceso de un archivo

Escriba un procedimiento llamado **FechaUltimoAcceso** para llenar una estructura SYSTEMTIME con la información de etiqueta de fecha y hora de un archivo. Debe recibir el desplazamiento de un nombre de archivo en EDX, y el desplazamiento de una estructura SYSTEMTIME en ESI. Si la función no puede encontrar el archivo, active la bandera Acarreo. Al implementar esta función, tendrá que abrir el archivo, obtener su manejador, pasar el manejador a **GetFileTime**, pasar su salida a **FileTimeToSystemTime** y cerrar el archivo. Escriba un programa de prueba para llamar a su procedimiento y que éste imprima la fecha del último acceso a un archivo específico. Ejemplo:

cap11_09.asm se acceso por ultima vez en: 6/16/2005

10. Leer un archivo extenso

Modifique el programa LeerArchivo.asm en la sección 11.1.8, para que pueda leer archivos más grandes que su búfer de entrada. Reduzca el tamaño del búfer a 1024 bytes. Use un ciclo para continuar la lectura y mostrar el archivo hasta que no se puedan leer más datos. Si planea mostrar el búfer con WriteString, recuerde insertar un byte nulo al final de los datos del búfer.

11. Lista enlazada

Avanzado: implemente una lista simplemente enlazada, usando las funciones de asignación dinámica de memoria que presentamos en este capítulo. Cada enlace deberá ser una estructura llamada Nodo (vea el capítulo 10), que contenga un valor entero y un apuntador al siguiente enlace en la lista. Use un ciclo para pedir al usuario todos los enteros que deseé escribir. A medida que se introduzca cada entero, asigne un objeto Nodo, inserte el entero en el Nodo y adjunte ese Nodo a la lista enlazada. Cuando se introduzca un valor de 0, detenga el ciclo. Por último, muestre toda la lista completa, de principio a fin. *Sólo debe intentar realizar este proyecto si ya ha creado anteriormente listas enlazadas en un lenguaje de alto nivel.*

Notas finales

1. Fuente: Documentación de Microsoft MSDN.
2. Los procesadores Pentium Pro y posteriores permiten una opción de dirección de 32 bits, pero no lo veremos en este libro.

INTERFAZ CON LENGUAJES DE ALTO NIVEL

- 12.1 Introducción
 - 12.1.1 Convenciones generales
 - 12.1.2 Repaso de sección
- 12.2 Código ensamblador en línea
 - 12.2.1 La directiva `_asm` en Microsoft Visual C++
 - 12.2.2 Ejemplo de cifrado de archivos
 - 12.2.3 Repaso de sección
- 12.3 Enlace con C/C++ en modo protegido
 - 12.3.1 Uso de lenguaje ensamblador para optimizar código en C++
 - 12.3.2 Llamadas a funciones en C y C++
 - 12.3.3 Ejemplo de tabla de multiplicación
- 12.3.4 Llamadas a funciones de la biblioteca de C
- 12.3.5 Programa de listado de directorios
- 12.3.6 Repaso de sección
- 12.4 Enlace con C/C++ en modo de direccionamiento real
 - 12.4.1 Enlace con Borland C++
 - 12.4.2 Ejemplo: LeerSector
 - 12.4.3 Ejemplo: enteros aleatorios grandes
 - 12.4.4 Repaso de sección
- 12.5 Resumen del capítulo
- 12.6 Ejercicios de programación

12.1 Introducción

La mayoría de los programadores no escriben aplicaciones de gran escala en lenguaje ensamblador, ya que se requeriría demasiado tiempo. En vez de ello, los lenguajes de alto nivel ocultan los detalles que de otra forma harían más lento el desarrollo de un proyecto. Sin embargo, el lenguaje ensamblador aún se utiliza bastante para configurar los dispositivos de hardware y optimizar tanto la velocidad como el tamaño del código de los programas.

En este capítulo nos enfocaremos en la *interfaz*, o conexión, entre el lenguaje ensamblador y los lenguajes de programación de alto nivel. En la primera sección mostraremos cómo escribir código de ensamblador en línea en C++. En la siguiente sección, enlazaremos módulos separados en lenguaje ensamblador con programas en C++. Se muestran ejemplos tanto en modo protegido como en modo de direccionamiento real. Por último, mostraremos cómo llamar funciones en C y C++ desde el lenguaje ensamblador.

12.1.1 Convenciones generales

Hay varias consideraciones generales que debemos tomar en cuenta al llamar a procedimientos en lenguaje ensamblador desde lenguajes de alto nivel.

En primer lugar, la *convención de nomenclatura* utilizada por un lenguaje se refiere a las reglas o características relacionadas con el nombramiento de las variables y los procedimientos. Por ejemplo, tenemos que responder a una pregunta importante: ¿El ensamblador o compilador altera los nombres de los identificadores que se colocan en archivos de código objeto y, de ser así, cómo lo hacen?

En segundo lugar, los *nombres de los segmentos* deben ser compatibles con los que se utilizan en el lenguaje de alto nivel.

En tercer lugar, el *modelo de memoria* utilizado por un programa (diminuto, pequeño, compacto, medio, grande, enorme o plano) determina el tamaño del segmento (16 o 32 bits), y si las llamadas y referencias serán cercanas (dentro del mismo segmento) o lejanas (entre distintos segmentos).

Convención de llamadas La *convención de llamadas* se refiere a los detalles de bajo nivel acerca de la forma en que se llaman los procedimientos. Hay que tener en cuenta los siguientes detalles:

- Qué registros deben preservar los procedimientos a los que se llama.
- El método utilizado para pasar argumentos: en registros, en la pila, en memoria compartida o mediante algún otro método.
- El orden en el que los programas que llaman a los procedimientos pasan los argumentos.
- Si se van a pasar los argumentos por valor o por referencia.
- La forma en que se restaura el apuntador de la pila después de la llamada a un procedimiento.
- La forma en que las funciones devuelven los valores a los programas que hacen llamadas.

Identificadores externos Al llamar a un procedimiento en lenguaje ensamblador desde un programa escrito en otro lenguaje, los identificadores externos deben tener convenciones de nomenclatura compatibles. Los *identificadores externos* son nombres que se han colocado en el archivo de código objeto de un módulo, de tal forma que el enlazador pueda poner los nombres a disposición de otros módulos del programa. El enlazador resuelve las referencias a los identificadores externos, pero sólo puede hacerlo si las convenciones de nomenclatura que se utilicen son consistentes.

Por ejemplo, suponga que un programa en C llamado *Main.c* llama a un procedimiento externo llamado **SumaArreglo**. Como se muestra en el siguiente diagrama, el compilador de C preserva de manera automática el uso de mayúsculas y minúsculas, y adjunta un guion bajo a la izquierda del nombre externo, cambiándolo a **_SumaArreglo**:



El módulo *Arreglo.asm*, escrito en lenguaje ensamblador, exporta el nombre del procedimiento **SumaArreglo** como **SUMAARREGLO**, ya que utiliza la opción de lenguaje Pascal en su directiva **.MODEL**. El enlazador no puede producir un programa ejecutable, ya que los dos nombres exportados son distintos.

Los compiladores para lenguajes de programación antiguos como COBOL y PASCAL, por lo general, solamente convierten los identificadores a mayúsculas. Lenguajes más recientes como C, C++ y Java preservan el uso de mayúsculas y minúsculas de los identificadores. Además, los lenguajes que soportan la sobrecarga de funciones (como C++) utilizan una técnica conocida como *decoración de nombres*, la cual agrega caracteres adicionales a los nombres de las funciones. Por ejemplo, una función llamada *MiSub(int n, double b)* podría exportarse como *MiSub#int#double*.

En un módulo de lenguaje ensamblador, podemos controlar la sensibilidad a mayúsculas y minúsculas eligiendo uno de los especificadores de lenguaje en la directiva **.MODEL** (vea los detalles en la sección 8.4.1).

Nombres de segmentos Al enlazar un procedimiento en lenguaje ensamblador con un programa escrito en un lenguaje de alto nivel, los nombres de los segmentos deben ser compatibles. En este capítulo, utilizaremos las directivas de segmento simplificadas de Microsoft **.CODE**, **.STACK** y **.DATA**, ya que son compatibles con los nombres de los segmentos producidos por los compiladores de Microsoft C++.

Modelos de memoria Un programa que hace la llamada a un procedimiento, y el procedimiento en sí, deben utilizar el mismo modelo de memoria. Por ejemplo, en modo de direccionamiento real podemos elegir los modelos pequeño (small), mediano (medium), compacto (compact), grande (large) y enorme (huge). En modo protegido, debemos usar el modelo plano (flat). En este capítulo mostraremos ejemplos de ambos modos.

12.1.2 Repaso de sección

1. ¿Qué quiere decir la *convención de nomenclatura* utilizada por un lenguaje?
2. ¿Qué modelos de memoria están disponibles en modo de direccionamiento real?
3. ¿Un procedimiento escrito en lenguaje ensamblador que utilice el especificador de lenguaje *Pascal* se enlazará con un programa en C++?
4. Cuando un programa en lenguaje de alto nivel llama a un procedimiento escrito en lenguaje ensamblador, ¿el programa que hace la llamada y el procedimiento llamado deben usar el mismo modelo de memoria?
5. Por qué es importante la sensibilidad a mayúsculas y minúsculas al llamar a los procedimientos en lenguaje ensamblador desde programas en C y C++?
6. ¿La convención de llamadas de un lenguaje incluye la preservación de ciertos registros por parte de los procedimientos?

12.2 Código ensamblador en línea

12.2.1 La directiva `_asm` en Microsoft Visual C++

El código ensamblador en línea es código fuente en lenguaje ensamblador que se inserta directamente en los programas en lenguajes de alto nivel. La mayoría de los compiladores de C/C++ soportan esta característica, al igual que Borland C++, Pascal y Delphi.

En esta sección demostraremos cómo escribir código ensamblador en línea para Microsoft Visual C++, que se ejecuta en modo protegido de 32 bits con el modelo plano de memoria. Otros compiladores de lenguajes de alto nivel soportan el código ensamblador en línea, pero la sintaxis exacta varía.

El código ensamblador en línea es una alternativa simple para el proceso de escribir código ensamblador en módulos externos. La ventaja principal de escribir código en línea es la simplicidad, ya que no hay que preocuparse por las cuestiones relacionadas con el enlace externo, los problemas de nomenclatura, y los protocolos para el paso de parámetros.

La desventaja principal de usar código ensamblador en línea es su falta de portabilidad. Esto es un problema cuando un programa en lenguaje de alto nivel debe compilarse para distintas plataformas de destino. Por ejemplo, el código ensamblador en línea que se ejecuta en un procesador Intel Pentium no se ejecutará en un procesador RISC. Hasta cierto grado, el problema puede resolverse insertando definiciones condicionales en el código fuente del programa para habilitar distintas versiones de las funciones para distintos sistemas de destino. Sin embargo, es fácil ver que el mantenimiento sigue siendo un problema. Por otra parte, una biblioteca de enlace de procedimientos externos en lenguaje ensamblador podría sustituirse con facilidad por una biblioteca de enlace similar, diseñada para una máquina de destino distinta.

La directiva `_asm` En Visual C++, la directiva `_asm` puede colocarse al principio de una sola instrucción, o puede marcar el comienzo de un bloque de instrucciones en lenguaje ensamblador (llamado *bloque asm*). La sintaxis es:

```
_asm instrucción
__asm {
    instrucción-1
    instrucción-2
    ...
    instrucción-n
}
```

(Hay dos caracteres de guion bajo antes de “asm”).

Comentarios Los comentarios se pueden colocar después de cualquier instrucción en el bloque `asm`, usando ya sea la sintaxis de lenguaje ensamblador, o la sintaxis de C/C++. El manual de Visual C++ sugiere evitar los comentarios estilo ensamblador, ya que podrían interferir con las macros en C, que se expanden en una sola línea lógica. He aquí ejemplos de comentarios permitidos:

```
mov esi,buf ; inicializa el registro índice
```

```
mov esi,buf // inicializa el registro índice
mov esi,buf /* inicializa el registro índice */
```

Características He aquí lo que podemos hacer al escribir código ensamblador en línea:

- Usar cualquier instrucción del conjunto de instrucciones.
- Usar los nombres de los registros como operandos.
- Hacer referencia a los parámetros de las funciones por su nombre.
- Hacer referencia a las etiquetas de código y las variables que se declararon fuera del bloque *asm*. Esto es importante, ya que las variables de funciones locales deben declararse fuera del bloque *asm*.
- Usar literales numéricas que incorporan la notación estilo ensamblador o de base estilo C. Por ejemplo, 0A26h y 0xA26 son equivalentes y ambos pueden utilizarse.
- Usar el operador PTR en instrucciones como **inc BYTE PTR [esi]**.
- Usar las directivas EVEN y ALIGN.

Limitaciones No podemos hacer lo siguiente al escribir código ensamblador en línea:

- Usar directivas de definición de datos como DB (BYTE) y DW (WORD).
- Usar operadores de ensamblador (excepto PTR).
- Usar STRUCT, RECORD, WIDTH y MASK.
- Usar directivas de macros, incluyendo MACRO, REPT, IRC, IRP y ENDM, u operadores de macros (<>, !, &, % y .TYPE).
- Hacer referencia a los segmentos por su nombre. Sin embargo, podemos usar los nombres de los registros de segmento como operandos.

Valores de registros No podemos hacer ninguna suposición acerca de los valores de los registros al principio de un bloque *asm*. El código que se ejecutó justo antes del bloque *asm* pudo haber modificado los registros justo antes del bloque *asm*. La palabra clave **_fastcall** en Microsoft Visual C++ hace que el compilador utilice registros para pasar los parámetros. Para evitar conflictos con los registros, no use **_fastcall** y **_asm** al mismo tiempo.

En general, podemos modificar a EAX, EBX, ECX y EDX en nuestro código en línea, ya que el compilador no espera que se preserven estos valores de una instrucción a otra. No obstante, si modificamos demasiados registros, tal vez sea imposible para el compilador optimizar por completo el código de C++ en el mismo procedimiento, ya que la optimización requiere el uso de registros.

Aunque no podemos usar el operador OFFSET, podemos obtener el desplazamiento de una variable mediante el uso de la instrucción LEA. Por ejemplo, la siguiente instrucción mueve el desplazamiento de **bufer** a ESI:

```
lea esi,buffer
```

Longitud, tipo y tamaño Podemos usar los operadores LENGTH, SIZE y TYPE con el ensamblador en línea. El operador LENGTH devuelve el número de elementos en un arreglo. El operador TYPE devuelve una de las siguientes opciones, dependiendo de su destino:

- El número de bytes utilizados por un tipo de C o C++, o una variable escalar.
- El número de bytes utilizados por una estructura.
- Para un arreglo, el tamaño de un solo elemento del arreglo.

El operador SIZE devuelve LENGTH * TYPE. El siguiente extracto de un programa demuestra los valores devueltos por el ensamblador en línea para varios tipos en C++.

El ensamblador en línea de Microsoft Visual C++ no soporta los operadores SIZEOF y LENGTHOF.

Uso de los operadores LENGTH, TYPE y SIZE

El siguiente programa contiene código ensamblador en línea que utiliza los operadores LENGTH, TYPE y SIZE para evaluar variables de C++. El valor devuelto por cada expresión se muestra como comentario en la misma línea:

```
struct Paquete {
    long codPostOrigen;           // 4
    long codPostDestino;          // 4
```

```

        float precioEnvio;           // 4
};

char miChar;
bool miBool;
short miShort;
int miInt;
long miLong;
float miFloat;
double miDouble;
Paquete miPaquete;

long double miLongDouble;
long miArregloLong[10];

__asm {
    mov eax,miPaquete.codPostDestino;
    mov eax,LENGTH miInt;          // 1
    mov eax,LENGTH miArregloLong; // 10
    mov eax,TYPE miChar;          // 1
    mov eax,TYPE miBool;          // 1
    mov eax,TYPE miShort;         // 2
    mov eax,TYPE miInt;          // 4
    mov eax,TYPE miLong;          // 4
    mov eax,TYPE miFloat;         // 4
    mov eax,TYPE miDouble;        // 8
    mov eax,TYPE miPaquete;       // 12
    mov eax,TYPE miLongDouble;    // 8
    mov eax,TYPE miArregloLong;   // 4
    mov eax,SIZE miLong;          // 4
    mov eax,SIZE miPaquete;       // 12
    mov eax,SIZE miArregloLong;   // 40
}

```

12.2.2 Ejemplo de cifrado de archivos

Vamos a escribir un programa corto para leer un archivo, cifrarlo y escribir su salida en otro archivo. La función **TraducirBufer** utiliza un bloque **asm** para definir instrucciones que iteran a través de un arreglo de caracteres, y aplicar un XOR a cada carácter con un valor predefinido. Las instrucciones en línea pueden hacer referencia a parámetros de funciones, variables locales y etiquetas de código. Como este ejemplo se compiló en Microsoft Visual C++ como aplicación de Consola Win32, el tipo de datos entero sin signo es de 32 bits:

```

void TraducirBufer( char * buf,
                     unsigned cuenta, unsigned char carCif )
{
    __asm {
        movesi,buf
        movecx,cuenta
        moval,carCif
L1:
        xor[esi],al
        incesi
        loopL1
    } // asm
}

```

Módulo de C++ El programa de inicio de C++ lee los nombres de los archivos de entrada y salida de la línea de comandos. Llama a **TraducirBufer** desde un ciclo que lee bloques de datos de un archivo, los cifra y escribe el búfer traducido en un nuevo archivo:

```
// CODIFICA.CPP - Copia y cifra un archivo.
```

```
#include <iostream>
#include <fstream>
#include "traducir.h"

using namespace std;

int main( int argcunt, char * args[] )
{
    // Lee archivos de entrada y salida de la línea de comandos.
    if( argcunt < 3 ) {
        cout << "Uso: codificar archent archsal" << endl;
        return -1;
    }
    const int TAMBUF = 2000;
    char bufer[TAMBUF];
    unsigned int cuenta;           // cuenta de caracteres
    unsigned char codigoCifrado;
    cout << "Codigo de cifrado [0-255]? ";
    cin >> codigoCifrado;

    ifstream archent( args[1], ios::binary );
    ofstream archsal( args[2], ios::binary );
    cout << "Leyendo" << args[1] << " y creando"
        << args[2] << endl;

    while ( !archent.eof() )
    {
        archent.read(bufer, TAMBUF);
        cuenta = archent.gcount();
        TraducirBufer(bufer, cuenta, codigoCifrado);
        archsal.write(bufer, cuenta);
    }
    return 0;
}
```

Es más fácil ejecutar este programa desde una ventana de comando del sistema, y pasarle los nombres de los archivos de entrada y salida. Por ejemplo, la siguiente línea de comandos lee archent.txt y produce codificado.txt:

```
codifica archent.txt codificado.txt
```

Archivo de encabezado El archivo de encabezado *traducir.h* contiene un solo prototipo de función para **TraducirBufer**:

```
void TraducirBufer (char * buf, unsigned cuenta,
                     unsigned char carCif);
```

Puede ver este programa en la carpeta \Ejemplos\cap12\VisualCPP\Codifica.

Sobrecarga de llamadas a procedimientos

Si analiza la ventana Desensamblador mientras depura este programa en un depurador, es interesante observar la sobrecarga que implica el proceso de llamar a un procedimiento y regresar de éste. Las siguientes instrucciones meten tres argumentos en la pila y llaman a **TraducirBufer**. En la ventana Desensamblador de Visual C++, activamos las opciones Mostrar código fuente y Mostrar nombres de símbolos:

```
; TraducirBufer(bufer, cuenta, codigoCifrado)
mov al,byte ptr [codigoCifrado]
push eax
mov ecx,dword ptr [cuenta]
push ecx
```

```
lea edx,[bufer]
push edx
call TraducirBufer (4159BFh)
add esp,0Ch
```

A continuación se muestra el código desensamblado de **TraducirBufer**. El compilador insertó de manera automática varias instrucciones para establecer EBP y guardar un conjunto estándar de registros que siempre se preservan, sin importar que el procedimiento los modifique o no:

```
push ebp
mov ebp,esp
sub esp,40h
push ebx
push esi
push edi

; El código en línea empieza aquí.
mov esi,dword ptr [buf]
mov ecx,dword ptr [cuenta]
mov al,byte ptr [carCif]
L1:
xor byte ptr [esi],al
inc esi
loop L1 (41D762h)
; Fin del código en línea.

pop edi
pop esi
pop ebx
mov esp,ebp
pop ebp
ret
```

Si desactivamos la opción *Mostrar nombres de símbolos* en la ventana Desensamblador del depurador, las tres instrucciones que mueven los parámetros hacia los registros aparecen así:

```
mov esi,dword ptr [ebp+8]
mov ecx,dword ptr [ebp+0Ch]
mov al,byte ptr [ebp+10h]
```

Se pidió al compilador que generara un destino *Debug*, el cual es código no optimizado, adecuado para una depuración interactiva. Si hubiéramos elegido un destino *Release*, el compilador hubiera generado código más eficiente (pero menos legible). En la sección 12.3.1 mostraremos código optimizado que generó el compilador.

Omisión de la llamada al procedimiento Las seis instrucciones en línea en la función **TraducirBufer** mostradas al principio de esta sección requieren un total de 18 instrucciones para ejecutarse. Si la función se llamara miles de veces, el tiempo requerido de ejecución podría ser considerable. Para evitar esta sobrecarga, vamos a insertar el código en línea en el ciclo que llama a **TraducirBufer**, creando así un programa más eficiente:

```
while (!archent.eof() )
{
    archent.read(bufer, TAMBUF );
    cuenta = archent.gcount();
    __asm {
        lea esi,bufer
        mov ecx,cuenta
        mov al,carCif
    L1:
        xor [esi],al
        inc esi
        loop L1
```

```

    } // asm
    archsal.write(bufer, cuenta);
}

```

Puede ver este programa en la carpeta *\Ejemplos\cap12\VisualCPP\Codifica_Enlinea*.

12.2.3 Repaso de sección

1. ¿Qué diferencia hay entre el código ensamblador en línea y un procedimiento de C++ en línea?
2. ¿Qué ventaja ofrece el código ensamblador en línea, en comparación con el uso de procedimientos externos en lenguaje ensamblador?
3. Muestre al menos dos formas de colocar comentarios en el código ensamblador en línea.
4. (Sí/No): ¿puede una instrucción en línea hacer referencia a etiquetas de código fuera del bloque __asm?
5. (Sí/No): ¿pueden utilizarse las directivas EVEN y ALIGN en el código ensamblador en línea?
6. (Sí/No): ¿puede usarse el operador OFFSET en el código ensamblador en línea?
7. (Sí/No): ¿pueden definirse variables con los operadores DW y DUP en el código ensamblador en línea?
8. Al utilizar la convención de llamadas **_fastcall**, ¿qué podría ocurrir si su código ensamblador en línea modifica los registros?
9. En vez de utilizar el operador OFFSET, ¿hay alguna otra forma de mover el desplazamiento de una variable a un registro índice?
10. ¿Qué valor devuelve el operador LENGTH cuando se aplica a un arreglo de enteros de 32 bits?
11. ¿Qué valor devuelve el operador SIZE cuando se aplica a un arreglo de enteros largos?

12.3 Enlace con C/C++ en modo protegido

Los programas escritos para los procesadores IA-32 que se ejecutan en modo protegido pueden algunas veces tener cuellos de botella que deben optimizarse para una eficiencia en tiempo de ejecución. Si son sistemas incrustados, deben tener limitaciones estrictas en cuanto al tamaño de la memoria. Con tales objetivos en mente, le mostraremos cómo escribir procedimientos externos en lenguaje ensamblador que pueden llamarse desde programas en C y C++ que se ejecutan en modo protegido. Dichos programas consisten cuando menos de dos módulos: El primero, escrito en lenguaje ensamblador, contiene el procedimiento externo; el segundo módulo contiene el código en C/C++ que inicia y termina el programa. Hay unos cuantos requerimientos específicos y características de C/C++ que afectan la manera en que debemos escribir el código ensamblador.

Argumentos Los programas en C/C++ pasan los argumentos de derecha a izquierda, como aparecen en la lista de argumentos. Una vez que el procedimiento regresa, el programa que hizo la llamada es responsable de limpiar la pila. Para ello, podemos sumar un valor al apuntador de la pila que sea igual al tamaño de los argumentos, o sacar un número adecuado de valores de la pila.

Nombres externos En el código fuente en lenguaje ensamblador, especifique la convención de llamadas de C en la directiva .MODEL y cree un prototipo para cada procedimiento que se llame desde un programa externo en C/C++:

```

.586
.model flat,C
AsmBuscarArreglo PROTO,
    valBusc:DWORD, apuntArreglo:PTR DWORD, cuenta:DWORD

```

Declaración de la función En un programa en C, use el calificador **extern** cuando declare un procedimiento externo en lenguaje ensamblador. Por ejemplo, así es como se declara **AsmBuscarArreglo**:

```
extern bool AsmBuscarArreglo( long n, long arreglo[], long cuenta );
```

Si el procedimiento se va a llamar desde un programa en C++, agregue un calificador “C” para evitar la decoración de nombres en C++:

```
extern "C" bool AsmBuscarArreglo( long n, long arreglo[], long cuenta );
```

La *decoración de nombres* es una técnica estándar del compilador de C++, que implica la modificación del nombre de una función con caracteres adicionales para indicar el tipo exacto de cada parámetro de la función. Se requiere en cualquier lenguaje que soporte la sobrecarga de funciones (dos funciones que tienen el mismo nombre, con distintas listas de parámetros). Desde el punto de vista del programador en lenguaje ensamblador, el problema con la decoración de nombres es que el compilador de C++ indica al enlazador que busque el nombre decorado en vez del original al momento de producir el archivo ejecutable.

12.3.1 Uso de lenguaje ensamblador para optimizar código en C++

Una de las formas en que podemos utilizar lenguaje ensamblador para optimizar programas escritos en otros lenguajes es buscar cuellos de botella relacionados con la velocidad. Los ciclos son buenos candidatos para la optimización, ya que cualquier instrucción adicional en un ciclo puede repetirse suficientes veces como para tener un efecto considerable en el rendimiento de un programa.

La mayoría de los compiladores de C/C++ tienen una opción de línea de comandos que genera en forma automática un listado en lenguaje ensamblador del programa en C/C++. Por ejemplo, en Microsoft Visual C++ el archivo de listado puede contener cualquier combinación de código fuente en C++, código ensamblador y código máquina, lo cual se muestra mediante las opciones en la tabla 12-1. Tal vez la más útil sea /FAs, que muestra cómo se traducen las instrucciones de C++ en lenguaje ensamblador.

Tabla 12-1 Opciones de línea de comandos de Visual C++ para la generación de código ASM.

Línea de comandos	Contenido del archivo de listado
/FA	Listado de sólo ensamblador
/FAC	Ensamblador con código máquina
/FAs	Ensamblador con código fuente
/FACs	Ensamblador, código máquina y código fuente

Ejemplo: BuscarArreglo

Vamos a crear un programa que muestre cómo un compilador de C++ de ejemplo genera código para una función llamada **BuscarArreglo**. Más adelante escribiremos una versión en lenguaje ensamblador de la función, tratando de escribir código más eficiente que el compilador de C++. La siguiente función **BuscarArreglo** (en C++) busca un valor individual en un arreglo de enteros largos:

```
bool BuscarArreglo( long valBuscar, long array[], long cuenta )
{
    for(int i = 0; i < cuenta; i++)
        array[i] *= valBuscar;

    return false;
}
```

Código de BuscarArreglo generado por Visual C++

Vamos a analizar el código fuente en lenguaje ensamblador generado por Visual C++ para la función **BuscarArreglo**, junto con el código fuente de la función en C++. Este procedimiento se compiló en un destino *Release*, sin optimización de código en efecto:

```
_i$282 = -4 ; tamaño = 4
_valBuscar$ = 8 ; tamaño = 4
_arreglo$ = 12 ; tamaño = 4
_cuenta$ = 16 ; tamaño = 4

_BuscarArreglo PROC NEAR
; 9      : {
```

```

push    ebp
mov     ebp,esp
push    ecx

; 10   : for(int i = 0; i < cuenta; i++)
        mov    DWORD PTR _i$282[ebp], 0
        jmp    SHORT $L283

$L284:
        mov    eax, DWORD PTR _i$282[ebp]
        add    eax, 1
        mov    DWORD PTR _i$282[ebp], eax

$L283:
        mov    ecx, DWORD PTR _i$282[ebp]
        cmp    ecx, DWORD PTR _cuenta$[ebp]
        jge    SHORT $L285

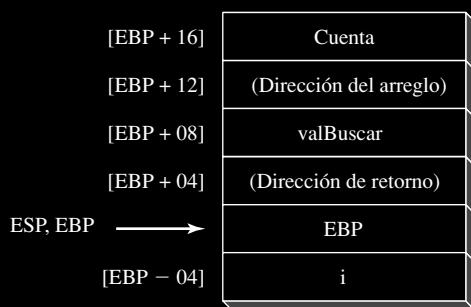
; 11   : arreglo[i] *= valBuscar;
        mov    edx, DWORD PTR _i$282[ebp]
        mov    eax, DWORD PTR _arreglo$[ebp]
        mov    ecx, DWORD PTR [eax+edx*4]
        imul  ecx, DWORD PTR _valBuscar$[ebp]
        mov    edx, DWORD PTR _i$282[ebp]
        mov    eax, DWORD PTR _arreglo$[ebp]
        mov    DWORD PTR [eax+edx*4], ecx
        jmp    SHORT $L284

$L285:
; 12   :
; 13   : return false;
        xor    al, al
; 14 : }
        mov    esp, ebp
        pop    ebp
        ret    0
_BuscarArreglo ENDP

```

Se metieron tres argumentos de 32 bits en la pila, en el siguiente orden: **cuenta**, **arreglo** y **valBuscar**. De estos tres, **arreglo** es el único que se pasa por referencia, ya que en C/C++ el nombre de un arreglo es un apuntador implícito al primer elemento del arreglo. El procedimiento guarda a EBP en la pila y crea espacio para la variable local **i**, metiendo una doble palabra extra en la pila (figura 12-1).

FIGURA 12-1 Marco de pila para la función BuscarArreglo.



Dentro del procedimiento, el compilador reserva espacio en la pila local para la variable **i**, metiendo a ECX (línea 9). El mismo espacio de almacenamiento se libera al final, cuando EBP se copia de vuelta a ESP (línea 14). Hay 14 instrucciones entre las etiquetas \$L284 y \$L285, lo cual constituye el cuerpo principal del ciclo.

Podemos escribir fácilmente un procedimiento en lenguaje ensamblador que sea más eficiente que el código que se muestra aquí.

Enlace de MASM con Visual C++

Vamos a crear una versión en lenguaje ensamblador de BuscarArreglo optimizada a mano, llamada **AsmBuscarArreglo**. Se aplican unos cuantos principios básicos a la optimización de código:

- Sacar del ciclo lo más que se pueda del procesamiento.
- Mover los parámetros de pila y las variables locales a los registros.
- Aprovechar las instrucciones especializadas de procesamiento de cadenas/arreglos (en este caso, SCASD).

Utilizaremos Microsoft Visual C++ (Visual Studio) para compilar el programa en C++ que hace la llamada y Microsoft MASM para ensamblar el procedimiento al que se llama. Visual C++ genera aplicaciones de 32 bits que se ejecutan sólo en modo protegido. Elegimos la Consola Win32 como el tipo de aplicación de destino para los ejemplos que se muestran aquí, aunque no hay una razón por la cual los mismos procedimientos no puedan funcionar en aplicaciones ordinarias de MS Windows. En Visual C++, las funciones devuelven valores de 8 bits en AL, valores de 16 bits en AX, valores de 32 bits en EAX, y valores de 64 bits en EDX: EAX. Las estructuras de datos más grandes (valores de estructuras, arreglos, etc.) se almacenan en una ubicación estática de datos, y se devuelve un apuntador a los datos en EAX.

Nuestro código en lenguaje ensamblador es un poco más legible que el código generado por el compilador de C++, ya que usamos nombres de etiquetas significativos y definimos constantes que simplifican el uso de los parámetros de pila. He aquí el listado completo del módulo:

```
TITLE Procedimiento BuscarArreglo      (AsmBuscarArreglo.asm)
.586
.model flat,C

AsmBuscarArreglo PROTO,
    valBusc:DWORD, apuntArreglo:PTR DWORD, cuenta:DWORD

.code
;-----
AsmBuscarArreglo PROC USES edi,
    valBusc:DWORD, apuntArreglo:PTR DWORD, cuenta:DWORD
;
; Realiza una búsqueda lineal de un entero de 32 bits
; en un arreglo de enteros. Devuelve un valor
; booleano en AL, indicando si se encontró el entero.
;-----
    verdadero = 1
    falso = 0

    mov    eax, valBusc           ; valor a buscar
    mov    ecx, cuenta            ; número de elementos
    mov    edi, apuntArreglo       ; apuntador al arreglo
    repne scasd                  ; realiza la búsqueda
    jz     devolverVerdadero      ; ZF = 1 si lo encuentra

devolverFalso:
    mov    al, falso
    jmp    short salir

devolverVerdadero:
    mov    al, verdadero

salir:
    ret
AsmBuscarArreglo ENDP
END
```

Comprobación del rendimiento de BuscarArreglo

Programa de prueba Es interesante comprobar el rendimiento de cualquier código en lenguaje ensamblador que usted escriba, en comparación con el código similar escrito en C++. Para ese fin, el siguiente programa de prueba en C++ recibe como entrada un valor a buscar y obtiene la hora del sistema antes y después de ejecutar un ciclo que llama a BuscarArreglo un millón de veces. La misma prueba se realiza en AsmBuscarArreglo. He aquí un listado del archivo de encabezado *buscarr.h*, con prototipos de funciones para el procedimiento en lenguaje ensamblador y la función en C++:

```
// buscarr.h

extern "C" {
    bool AsmBuscarArreglo( long n, long array[], long cuenta );
    // Versión en lenguaje ensamblador

    bool BuscarArreglo( long n, long array[], long cuenta );
    // Versión en C++
}
```

Módulo principal de C++ He aquí un listado de *main.cpp*, el programa de inicio que llama a BuscarArreglo y a AsmBuscarArreglo:

```
// main.cpp - Prueba de BuscarArreglo y AsmBuscarArreglo.

#include <iostream>
#include <time.h>
#include "buscarr.h"
using namespace std;

int main()
{
    // Llena un arreglo con enteros seudoaleatorios.
    const unsigned TAM_ARREGLO = 10000;
    const unsigned TAM_CICLO = 1000000;

    long array[TAM_ARREGLO];
    for(unsigned i = 0; i < TAM_ARREGLO; i++)
        array[i] = rand();

    long valBuscar;
    time_t tiempoInicial, tiempoFinal;
    cout << "Escriba el valor a buscar: ";
    cin >> valBuscar;
    cout << "Por favor espere. Esto tardara entre 10 y 30 segundos...\\n";

    // Prueba la función en C++:
    time( &tiempoInicial );
    bool encontro = false;

    for( int n = 0; n < TAM_CICLO; n++)
        encontro = BuscarArreglo( valBuscar, array, TAM_ARREGLO );

    time( &tiempoFinal );
    cout << "Tiempo transcurrido de CPP: " << long(tiempoFinal - tiempoInicial)
        << " segundos. Encontro = " << encontro << endl;

    // Prueba el procedimiento en lenguaje ensamblador:
    time( &tiempoInicial );
    encontro = false;

    for( int n = 0; n < TAM_CICLO; n++)
        encontro = AsmBuscarArreglo( valBuscar, array, TAM_ARREGLO );

    time( &tiempoFinal );
```

```

cout << "Tiempo transcurrido de ASM: " << long(tiempoFinal - tiempoInicial)
    << " segundos. Encontro = " << encontro << endl;
return 0;
}

```

Comparación entre código ensamblador y código en C++ sin optimizar Compilamos el programa en C++ a un destino Release (sin depuración) con la optimización de código desactivada. He aquí la salida, mostrando el peor caso (el valor no se encontró):

```

Escriba el valor a buscar: 55
Tiempo transcurrido de CPP: 53 segundos. Encontro = 0
Tiempo transcurrido de ASM: 14 segundos. Encontro = 0

```

Comparación entre código ensamblador y optimización del compilador A continuación, configuramos el compilador para optimizar el programa ejecutable para mejorar la velocidad, y ejecutamos el programa de prueba otra vez. He aquí los resultados, mostrando que el código ensamblador es considerablemente más rápido que el código en C++ optimizado por el compilador:

```

Escriba el valor a buscar: 55
Tiempo transcurrido de CPP: 20 segundos. Encontro = 0
Tiempo transcurrido de ASM: 14 segundos. Encontro = 0

```

Comparación entre apuntadores y subíndices

Los programadores que usaban compiladores de C antiguos observaron que el procesamiento de arreglos con apuntadores era más eficiente que el uso de subíndices. Por ejemplo, la siguiente versión de **BuscarArreglo** utiliza este método:

```

bool BuscarArreglo( long valBuscar, long arreglo[], long cuenta )
{
    long * p = arreglo;
    for(i = 0; i < cuenta, i++, p++)
        if( valBuscar == *p )
            return true;
    return false;
}

```

Al ejecutar esta versión de **BuscarArreglo** mediante el compilador de Visual C++ se produjo casi el mismo código en lenguaje ensamblador que la versión anterior que utiliza subíndices. Como los compiladores modernos son buenos para optimizar código, el uso de una variable apuntador no es más eficiente que el uso de un subíndice. He aquí el ciclo del código de destino de **BuscarArreglo** que produjo el compilador de C++:

```

$L176:
    cmp  esi, DWORD PTR [ecx]
    je   SHORT $L184
    inc  eax
    add  ecx, 4
    cmp  eax, edx
    jl   SHORT $L176

```

Invertiría bien su tiempo si estudiara la salida producida por un compilador de C++, para aprender acerca de las técnicas de optimización, el paso de parámetros y la implementación de código objeto. De hecho, muchos estudiantes de ciencias computacionales toman un curso de escritura de compiladores, que incluye dichos temas. También es importante considerar que los compiladores toman el caso general, ya que, por lo regular, no tienen un conocimiento específico acerca de las aplicaciones individuales o del hardware instalado. Algunos compiladores cuentan con una optimización especializada para un procesador específico, como el Pentium, lo cual puede mejorar en forma considerable la velocidad de los programas compilados. El lenguaje ensamblador codificado a mano puede aprovechar las instrucciones primitivas de cadenas de la familia IA-32, así como las características especializadas de hardware de las tarjetas de video, de sonido y de captura de datos.

12.3.2 Llamadas a funciones en C y C++

Podemos escribir programas en lenguaje ensamblador que llamen a funciones en C++. Existen por lo menos un par de razones para hacerlo:

- La entrada-salida es más flexible en C++, con su extensa biblioteca de flujos de entrada-salida (iostream). Esto es muy útil cuando se trabaja con números de punto flotante.
- C++ cuenta con amplias bibliotecas de matemáticas.

Al llamar funciones de la biblioteca estándar de C (o de C++), debemos iniciar el programa desde un procedimiento main() en C o C++ para que el código de inicialización de las bibliotecas pueda ejecutarse.

Prototipos de funciones

Las funciones en C++ que se llaman desde el código en lenguaje ensamblador deben definirse con las palabras clave “C” y **extern**. He aquí la sintaxis básica:

```
extern "C" nombreFunc( listaparam )
{ . . . }
```

He aquí un ejemplo:

```
extern "C" int pedirEntero()
{
    cout << "Escriba un entero:";
    //...
}
```

En vez de modificar la definición de cada función, es más fácil agrupar varios prototipos de funciones dentro de un bloque. Así, podemos omitir **extern** y “C” de las implementaciones de las funciones:

```
extern "C" {
    int pedirEntero();
    int mostrarEnt( int valor, unsigned anchSal );
    etc.
}
```

Módulo en lenguaje ensamblador

Uso de la biblioteca de vínculos Irvine32 Si su módulo en lenguaje ensamblador va llamar a procedimientos de la biblioteca de vínculos Irvine32, tenga en mente que utiliza la siguiente directiva .MODEL:

```
.model flat, STDCALL
```

Aunque STDCALL es compatible con la API Win32, no coincide con la convención de llamadas utilizada por los programas en C. Por lo tanto, debe agregar el calificador C a la directiva PROTO, al declarar funciones externas en C o C++ que el módulo en ensamblador vaya a llamar:

```
INCLUDE Irvine32.inc
pedirEntero PROTO C
mostrarEnt PROTO C, valor:SDWORD, anchSal:DWORD
```

El calificador C es requerido, ya que el enlazador debe hacer que coincidan los nombres de las funciones y listas de parámetros con las funciones exportadas por el módulo en C++. Además, el ensamblador debe generar el código adecuado para limpiar la pila después de las llamadas a las funciones, usando la convención de llamadas de C (vea la sección 8.4.1).

Los procedimientos en lenguaje ensamblador que sean llamados por el programa en C++ también deben utilizar el calificador C, de manera que el ensamblador utilice una convención de nomenclatura que el enlazador pueda reconocer. Por ejemplo, el siguiente procedimiento llamado **EstColorTextoSalida** tiene un solo parámetro tipo doble palabra:

```
EstColorTextoSalida PROC C,
color:DWORD
```

```
    .
    .
    .
    EstColorTextoSalida ENDP
```

Por último, si su código en ensamblador llama a otros procedimientos en lenguaje ensamblador, la convención de llamadas de C requiere que se eliminen los parámetros de la pila después de cada llamada a un procedimiento.

Uso de la directiva .MODEL Si su código en lenguaje ensamblador no llama a los procedimientos de la biblioteca Irvine32, puede indicar a la directiva .MODEL que utilice la convención de llamadas de C:

```
; (no incluir Irvine32.inc)
.586
.model flat,C
```

Ahora ya no tiene que agregar el calificador C a las directivas PROTO y PROC:

```
pedirEntero PROTO
mostrarEnt PROTO, valor:SDWORD, anchSal:DWORD
EstColorTextoSalida PROC,
    color:DWORD
    .
    .
    .
EstColorTextoSalida ENDP
```

Valores de retorno de las funciones

La especificación del lenguaje C++ no dice nada acerca de los detalles de implementación del código, por lo que no hay una forma estandarizada para que las funciones C++ regresen valores. Al escribir código en lenguaje ensamblador que llame a funciones en C++, revise la documentación de su compilador para ver cómo sus funciones devuelven los valores. La siguiente lista contiene varias posibilidades, pero no todas:

- Los enteros pueden regresarse en un solo registro, o en una combinación de registros.
- El programa que hace las llamadas puede reservar espacio en la pila para los valores de retorno de las funciones. Una función puede insertar los valores de retorno en la pila antes de regresar.
- Por lo general, los valores de punto flotante se meten en la pila de punto flotante del procesador antes de regresar de la función.

La siguiente lista muestra cómo las funciones de Microsoft Visual C++ devuelven los valores:

- Los valores **bool** y **char** se devuelven en AL.
- Los valores **short int** se devuelven en AX.
- Los valores **int** y **long int** se devuelven en EAX.
- Los apuntadores se devuelven en EAX.
- Los valores **float**, **double** y **long double** se meten en la pila de punto flotante como valores de 4, 8 y 10 bytes, respectivamente.

12.3.3 Ejemplo de tabla de multiplicación

Vamos a escribir una aplicación simple que pide al usuario un entero, lo multiplica por potencias ascendentes de 2 (desde 2^1 hasta 2^{10}), usando desplazamiento de bits, y vuelve a mostrar cada producto con espacios de relleno a la izquierda. Vamos a utilizar C++ para la entrada-salida. El módulo en lenguaje ensamblador contiene llamadas a tres funciones escritas en C++. El programa se inicia desde C++.

Módulo en lenguaje ensamblador

El módulo en lenguaje ensamblador contiene una función llamada **MostrarTabla**. Esta función llama a una función en C++ llamada **pedirEntero**, la cual recibe como entrada un entero por parte del usuario. Utiliza un ciclo para desplazar en forma repetida un entero llamado **valEnt** a la izquierda, y lo muestra en pantalla llamando a **mostrarEnt**.

```

; Función ASM que se llama desde C++.

INCLUDE Irvine32.inc

; Funciones externas en C++:
pedirEntero PROTO C
mostrarEnt PROTO C, valor:SDWORD, anchSal:DWORD

ANCHURA_SALIDA = 8
POTENCIA_FINAL = 10

.data
valEnt DWORD ?

.code
;-----
EstColorTextoSalida PROC C,
    color:DWORD
;
; Establece los colores del texto y borra la ventana
; de consola. Llama a funciones de la biblioteca Irvine32.
;-----
    mov    eax,color
    call   SetTextColor
    call   Clrscr
    ret
EstColorTextoSalida ENDP

;-----
MostrarTabla PROC C
;
; Recibe como entrada un entero n y muestra una
; tabla de multiplicación que varía de n * 2^1
; a n * 2^10.
;-----
    INVOKE pedirEntero          ; llama a la función en C++
    mov    valEnt,eax           ; guarda el entero
    mov    ecx,POTENCIA_FINAL  ; contador del ciclo

L1: push   ecx              ; guarda contador del ciclo
    shl    valEnt,1            ; multiplica por 2
    INVOKE mostrarEnt,valEnt,ANCHURA_SALIDA
    INVOKE nuevaLinea          ; salida CR/LF
    pop    ecx              ; restaura contador del ciclo
    loop   L1

    ret
MostrarTabla ENDP
END

```

En MostrarTabla, ECX debe meterse y sacarse antes de llamar a **mostrarEnt**, ya que las funciones en Visual C++ no guardan y restauran los registros de propósito general. La función **pedirEntero** devuelve su resultado en el registro EAX.

No se requiere que MostrarTabla use INVOKE al llamar a las funciones en C++. Puede lograrse el mismo resultado usando instrucciones PUSH y CALL. Así es como debe verse la llamada a **mostrarEnt**:

```

push  ANCHURA_SALIDA          ; mete el último argumento primero
push  valEnt

```

```

call  mostrarEnt           ; llama a la función
add   esp,8                ; limpia la pila

```

Debemos seguir la convención de llamadas del lenguaje C, en la cual los argumentos se meten en la pila en orden inverso, y el procedimiento que hace la llamada es responsable de eliminar los argumentos de la pila después de la llamada.

Programa de inicio en C++

Veamos ahora el módulo en C++ que inicia el programa. Su punto de entrada es **main()**, el cual asegura la ejecución del código de inicialización requerido en el lenguaje C++. Contiene prototipos de funciones para el procedimiento externo en lenguaje ensamblador y las tres funciones exportadas:

```

// main.cpp

// Demuestra las llamadas a funciones entre un programa
// en C++ y un módulo externo en lenguaje ensamblador.

#include <iostream>
#include <iomanip>
using namespace std;

extern "C" {
    // procedimientos ASM externos:
    void MostrarTabla();
    void EstColorTextoSalida( unsigned color );

    // funciones locales en C++:
    int pedirEntero();
    void mostrarEnt( int valor, int anchura );
}

// punto de entrada del programa
int main()
{
    EstColorTextoSalida( 0x1E );           // amarillo sobre azul
    MostrarTabla();                      // llama a procedimiento ASM
    return 0;
}

// Pide un entero al usuario.
int pedirEntero()
{
    int n;
    cout << "Escriba un entero entre 1 y 90,000:" ;
    cin >> n;
    return n;
}

// Muestra un entero con signo, con una anchura específica.
void mostrarEnt( int valor, int anchura )
{
    cout << setw(anchura) << valor;
}

```

Generación del proyecto Nuestro sitio Web (www.asmirvine.com) tiene un tutorial para generar proyectos combinados de C++/lenguaje ensamblador en Visual Studio.

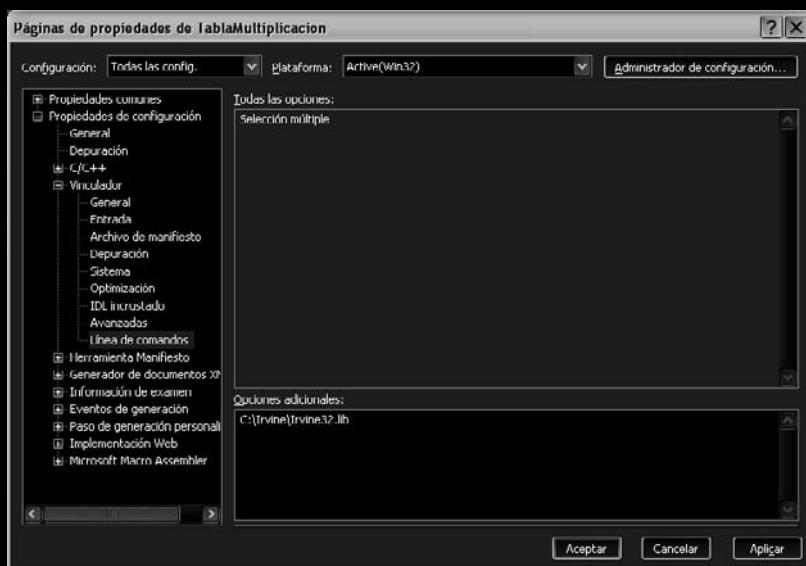
Resultado del programa He aquí un ejemplo de los resultados que genera el programa Tabla de multiplicación cuando el usuario introduce 90,000:

```
Escriba un entero entre 1 y 90,000: 90000
180000
360000
720000
1440000
2880000
5760000
11520000
23040000
46080000
92160000
```

Propiedades de Visual Studio

Si utiliza Visual Studio para generar programas que integran C++ con lenguaje ensamblador y hace llamadas a la biblioteca Irvine32, debe alterar ciertas configuraciones de los proyectos. Vamos a utilizar el programa Tabla de multiplicación como ejemplo. Seleccione *Propiedades* del menú Proyecto. En la lista desplegable *Configuración*, seleccione *Todas las configuraciones*. En *Propiedades de configuración*, haga doble clic en *Enlazador* y seleccione *Línea de comandos*. Agregue **C:\Irvine\Irvine32.lib** a la entrada *Opciones adicionales* (figura 12-2). Haga clic en Aceptar para cerrar la ventana Páginas de propiedades. Ahora Visual Studio podrá encontrar la biblioteca Irvine32.

FIGURA 12-2 Propiedad Línea de comandos del Enlazador.



La información aquí mostrada se probó en Visual Studio 2005, pero está sujeta a cambios. Por favor visite nuestro sitio Web (www.asmirvine.com) para ver las actualizaciones.

12.3.4 Llamadas a funciones de la biblioteca de C

El lenguaje C cuenta con una colección estandarizada de funciones, conocida como *Biblioteca estándar de C*. Las mismas funciones están disponibles para los programas en C++ y, por lo tanto, para los módulos en lenguaje ensamblador que se unen a los programas en C y C++. Los módulos en lenguaje ensamblador deben contener un prototipo para cada función en C que llamen. Por lo general, podemos encontrar los prototipos de las funciones en C accediendo al sistema de ayuda que proporciona el compilador de C++. Debemos

traducir los prototipos de las funciones en C a prototipos en lenguaje ensamblador, para poder llamarlas desde nuestros programas.

Función printf A continuación se muestra el prototipo en lenguaje C/C++ para la función **printf**, que muestra un apuntador a un carácter como su primer parámetro, seguido de un número variable de parámetros:

```
int printf(
    const char *formato [, argumento]...
);
```

La biblioteca de ayuda de su compilador de C/C++ contiene mucha documentación acerca de la función **printf**. El prototipo equivalente en lenguaje ensamblador cambia a `char *` por PTR BYTE, y cambia la lista de parámetros de longitud variable por el tipo VARARG:

```
printf PROTO C, pString:PTR BYTE, args:VARARG
```

Otra función útil es **scanf**, que recibe como entrada caracteres, números y cadenas de la entrada estándar (el teclado) y asigna los valores de entrada a variables:

```
scanf PROTO C, formato:PTR BYTE, args:VARARG
```

Visualización de números reales con formato, mediante la función printf

No es fácil escribir funciones en lenguaje ensamblador para dar formato y mostrar valores de punto flotante en pantalla. En vez de hacerlo usted mismo, puede aprovechar la función **printf** en lenguaje C/C++. Debe crear un módulo de inicio en C o C++ y enlazarlo con su código en lenguaje ensamblador. He aquí cómo establecer dicho programa en Visual C++ .NET:

1. Cree un programa de Consola Win32 en Visual C++. Cree un archivo llamado *main.cpp* e inserte una función **main** que llame a **asmMain** desde **main** en C++. También en **main**, inserte una declaración de por lo menos una variable de punto flotante. En el siguiente código, la instrucción que declara a **d** no tiene ningún otro efecto que forzar a Visual C++ a que cargue su biblioteca de punto flotante en tiempo de ejecución¹:

```
extern "C" void asmMain( );
int main( )
{
    double d = 3.5;           // carga la biblioteca de punto flotante
    asmMain( );
    return 0;
}
```

2. En la misma carpeta que *main.cpp*, cree un módulo en lenguaje ensamblador llamado *asmMain.asm*. Debe contener un procedimiento llamado **asmMain**, declarado con la convención de llamadas de C:

```
TITLE asmMain.asm
.386
.model flat,stdcall
.stack 2000
.code
asmMain PROC C
    ret
asmMain ENDP
END
```

3. Ensamble *asmMain.asm* (pero no lo enlace) para producir *asmMain.obj*.
4. Agregue *asmMain.obj* al proyecto de C++.
5. Genere y ejecute el proyecto. Si modifica *asmMain.asm*, ensámblelo de nuevo y vuelva a generar el proyecto antes de ejecutarlo otra vez.

Una vez que haya establecido su programa en forma adecuada, puede agregar código a asmMain.asm que llame a las funciones en lenguaje C/C++.

Visualización de valores de doble precisión El siguiente código en lenguaje ensamblador en **asmMain** imprime un valor REAL8, llamando a printf:

```
.data
double1 REAL8 1234567.890123
cadFormato BYTE "%.3f",0dh,0ah,0
.code
Invoke printf, ADDR cadFormato, double1
```

Ésta es la salida correspondiente:

```
1234567.890
```

La cadena de formato que se pasa a **printf** aquí es un poco distinta de lo que sería en C++. En vez de incrustar caracteres de escape como '\n', debemos insertar códigos ASCII (0dh, 0ah).

Los argumentos de punto flotante que se pasan a printf deben declararse con el tipo REAL8. Aunque es posible pasar valores de tipo REAL4, se requiere una cantidad considerable de programación astuta. Puede ver cómo su compilador de C++ hace esto si declara una variable de tipo float y la pasa a printf. Compile el programa y siga la ejecución del código desensamblado del programa con un depurador.

Varios argumentos La función printf acepta un número variable de argumentos, por lo que con la misma facilidad podemos dar formato y mostrar dos números en una llamada a la función:

```
TAB = 9
.data
formatoDos BYTE "%.2f",TAB,"%.3f",0dh,0ah,0
val1 REAL8 456.789
val2 REAL8 864.231
.code
Invoke printf, ADDR formatoDos, val1, val2
```

Ésta es la salida correspondiente:

```
456.79    864.231
```

[Vea el proyecto llamado **Ejemplo_Printf** (o **Printf_Example**) en la carpeta Ejemplos\Cap12\VisualCPP del código de este libro].

Escribir números reales con la función scanf

Podemos llamar a **scanf** para recibir como entrada valores de punto flotante de parte del usuario. El siguiente prototipo está definido en SmallWin.inc (se incluye en Irvine32.inc):

```
scanf PROTO C,
format:PTR BYTE, args:VARARG
```

Recibe el desplazamiento de una cadena de formato y los desplazamientos de una o más variables REAL4 o REAL8, para guardar los valores introducidos por el usuario. Ejemplos de llamadas:

```
.data
cadSimple BYTE "%f",0
cadDoble BYTE "%lf",0
simple1 REAL4 ?
```

```
doble1 REAL8 ?
.code
Invoke scanf, ADDR cadSimple, ADDR simple1
Invoke scanf, ADDR cadDoble, ADDR doble1
```

Debe invocar su código en lenguaje ensamblador desde un programa de inicio en C o C++.

12.3.5 Programa de listado de directorios

Vamos a escribir un programa corto para borrar la pantalla, mostrar el directorio actual del disco y pedir al usuario que introduzca el nombre de un archivo. Tal vez quiera extender este programa para que abra y muestre el archivo seleccionado.

Módulo auxiliar (stub) de C++ El módulo de C++ sólo contiene una llamada a **asm_main**, por lo que podemos llamarlo *módulo auxiliar*:

```
// main.cpp
// módulo auxiliar: inicial el programa en lenguaje ensamblador
extern "C" void asm_main();           // proc asm de inicio
void main()
{
    asm_main();
}
```

Módulo ASM El módulo de lenguaje ensamblador contiene los prototipos de funciones, varias cadenas y una variable **nombreArchivo**. Llama a la función **system** dos veces y le pasa los comandos “cls” y “dir”. Después llama a **printf**, muestra un indicador para pedir el nombre de un archivo y llama a **scanf**, para que el usuario pueda introducir el nombre. No hace llamadas a la biblioteca Irvine32, por lo que podemos establecer la directiva .MODEL a la convención del lenguaje C:

```
; Programa ASM iniciado desde C+          (asmMain.asm)
.586
.MODEL flat,C
; Funciones de la biblioteca estándar de C:
system PROTO, pCommand:PTR BYTE
printf PROTO, pString:PTR BYTE, args:VARARG
scanf  PROTO, pFormat:PTR BYTE,pBuffer:PTR BYTE, args:VARARG
fopen  PROTO, mode:PTR BYTE, filename:PTR BYTE
fclose PROTO, pfile:DWORD
TAM_BUFER = 5000
.data
cad1 BYTE "cls",0
cad2 BYTE "dir/w",0
cad3 BYTE "Escriba el nombre de un archivo: ",0
cad4 BYTE "%s",0
cad5 BYTE "no se puede abrir el archivo",0dh,0ah,0
cad6 BYTE "El archivo se abrio ",0dh,0ah,0
cadModo BYTE "r",0
nombreArchivo BYTE 60 DUP(0)
apuntBuf DWORD ?
apuntArch DWORD ?

.code
asm_main PROC
; borra la pantalla, muestra el directorio del disco
Invoke system,ADDR cad1
Invoke system,ADDR cad2
```

```

; pide un nombre de archivo
INVOKE printf, ADDR cad3
INVOKE scanf, ADDR cad4, ADDR nombreArchivo

; trata de abrir el archivo
INVOKE fopen, ADDR nombreArchivo, ADDR cadModo
mov apuntArch, eax

.IF eax == 0           ; ¿no se puede abrir el archivo?
INVOKE printf, ADDR cad5
jmp terminar
.ELSE
INVOKE printf, ADDR cad6
.ENDIF
.
; Cierra el archivo
INVOKE fclose, apuntArch

terminar:
ret                   ; regresa a main de C++
asm_main ENDP
END

```

La función `scanf` requiere dos argumentos: el primero es un apuntador a una cadena de formato (“`%s`”) y el segundo es un apuntador a la variable de cadena de entrada (**nombreArchivo**). No vamos a explicar las funciones estándar de C, ya que hay mucha documentación en Web. Una excelente referencia es la obra de Brian W. Kernighan y Dennis M. Ritchie, The C Programming Language, 2^a Ed., Prentice Hall, 1988.

12.3.6 Repaso de sección

1. ¿Cuáles son las dos palabras clave de C++ que deben incluirse en la definición de una función, si ésta se va a llamar desde un módulo en lenguaje ensamblador?
2. ¿En qué forma la convención de llamadas usada por la biblioteca Irvine32.lib no es compatible con la convención de llamadas usada por los lenguajes C y C++?
3. Por lo general, ¿cómo devuelven las funciones en C++ los valores de punto flotante?
4. ¿Cómo devuelve una función en Microsoft Visual C++ un **short int**?
5. ¿Cuál es una declaración PROTO válida en lenguaje ensamblador para la función `printf()` estándar de C?
6. Cuando se llame la siguiente función en lenguaje C, ¿el argumento `x` se meterá en primer o en último lugar en la pila?
`void MiSub (x, y, z);`
7. ¿Cuál es el propósito del especificador “C” en la declaración `extern`, en los procedimientos que se llaman desde C++?
8. ¿Por qué es importante la decoración de nombres al llamar a los procedimientos externos en lenguaje ensamblador desde C++?
9. En este capítulo, cuando se utilizó un compilador optimizador de C++, ¿qué diferencias relacionadas con la generación de código ocurrieron entre el ciclo codificado con subíndices de arreglo y el ciclo codificado con variables apuntadores?

12.4 Enlace con C/C++ en modo de direccionamiento real

Muchas aplicaciones de sistemas incrustados se siguen escribiendo para entornos de 16 bits, usando los procesadores 8086 y 8088. Además, algunas aplicaciones utilizan procesadores de 32 bits que se ejecutan en modo de direccionamiento real. Por lo tanto, es importante para nosotros mostrar ejemplos de subrutinas en lenguaje ensamblador que se llaman desde C y C++ en entornos de modo real.

Los programas de ejemplo en esta sección utilizan la versión de 16 bits de Borland C++ 5.01 y utilizan Windows 98 (ventana de MS-DOS) como el sistema operativo de destino, con un modelo pequeño (small) de

memoria. Vamos a utilizar Borland TASM 4.0 como ensamblador para estos ejemplos, ya que es probable que la mayoría de los usuarios de Borland C++ utilicen Turbo Assembler en vez de MASM. También crearemos aplicaciones en modo real de 16 bits usando Borland C++ 5.01 y presentaremos programas en los modelos pequeño y grande de memoria, mostrando cómo llamar a procedimientos cercanos (near) y lejanos (far).

12.4.1 Enlace con Borland C++

Valores de retorno de las funciones En Borland C++, las funciones devuelven valores de 16 bits en AX y valores de 32 bits en DX:AX. Las estructuras de datos más grandes (valores de estructuras, arreglos, etc.) se almacenan en una ubicación estática de datos, y se devuelve un apuntador a los datos en AX. En los modelos de programas mediano, grande y enorme, se devuelve un apuntador de 32 bits en DX:AX.

Establecer un proyecto En el entorno integrado de desarrollo de Borland C++ (IDE), cree un nuevo proyecto. Cree un módulo de código fuente (archivo CPP) y escriba el código para el programa principal en C++. Cree el archivo ASM que contiene el procedimiento que planea llamar. Use TASM para ensamblar el programa en un módulo objeto, ya sea desde la línea de comandos de DOS o desde el IDE de Borland C++, usando su capacidad de transferencia. El nombre de archivo (menos la extensión) debe tener ocho caracteres o menos; en caso contrario, el enlazador de 16 bits no reconocerá su nombre.

Si ensambló el módulo ASM por separado, agregue el archivo objeto creado por el ensamblador al proyecto de C++. Invoque el comando MAKE o BUILD desde el menú. Este comando compila el archivo CPP y, si no hay errores, enlaza los dos módulos de código objeto para producir un programa ejecutable. Sugerencia: limite el nombre del archivo CPP de código fuente a ocho caracteres, o de lo contrario el programa Turbo Debugger para DOS no podrá encontrarlo cuando quiera depurar el programa.

Depuración El compilador Borland C++ no permite que el depurador de DOS se ejecute desde el IDE. En vez de ello, debe ejecutar Turbo Debugger para DOS ya sea desde el símbolo de DOS o desde el escritorio de Windows. Si utiliza el comando de menú File/Open (Archivo/Abrir) del depurador, seleccione el archivo ejecutable creado por el enlazador de C++. El archivo de código fuente de C++ deberá aparecer de inmediato, y podrá empezar a rastrear y ejecutar el programa.

Cómo guardar los registros Los procedimientos en ensamblador que llame Borland C++ deben preservar los valores de BP, DS, SS, SI, DI y la bandera Dirección.

Tamaños de almacenamiento Un programa de Borland C++ de 16 bits utiliza tamaños de almacenamiento específicos para todos sus tipos de datos. Éstos son únicos para esta implementación específica, y deben ajustarse para cada compilador de C++. Consulte la tabla 12-2.

Tabla 12-2 Tipos de datos de Borland C++ en aplicaciones de 16 bits.

Tipo de C++	Bytes de almacenamiento	Tipo de ASM
char, unsigned char	1	byte
int, unsigned int, short int	2	word
enum	2	word
long, unsigned long	4	dword
float	4	dword
double	8	qword
long double	10	tbyte
apuntador cercano (near)	2	word
apuntador lejano (far)	4	dword

12.4.2 Ejemplo: LeerSector

Debe ejecutarse en MS-DOS, Windows 95, 98 o Millennium. Vamos a empezar con un programa de Borland C++ que llama a un procedimiento externo en lenguaje ensamblador llamado **LeerSector**. Por lo general, los compiladores de C++ no incluyen funciones de biblioteca para leer sectores de disco, ya que dichos detalles son demasiado dependientes del hardware, por lo que sería impráctico implementar bibliotecas para todas las computadoras posibles. Los programas en lenguaje ensamblador pueden leer con facilidad los sectores de disco, llamando a la función 7305h de INT 21h (en la sección 14.4 podrá consultar los detalles). Entonces, nuestra tarea actual es crear la interfaz entre el lenguaje ensamblador y C++ para combinar los puntos fuertes de ambos lenguajes.

El ejemplo **LeerSector** requiere el uso de un compilador de 16 bits, ya que implica llamadas a las interrupciones de MS-DOS (es posible llamar a las interrupciones de 16 bits desde programas de 32 bits, para ello consulte el libro de Barry Kauler cuya bibliografía se encuentra al final de este capítulo²). La última versión de Visual C++ que produce programas de 16 bits es la versión 1.5. Otros compiladores que producen código de 16 bits son Turbo C y Turbo Pascal, ambos de Borland.

Ejecución del programa Primero vamos a demostrar la ejecución del programa. Cuando se inicia el programa en C++, el usuario selecciona el número de la unidad, el sector inicial y el número de sectores a leer. Por ejemplo, este usuario desea leer los sectores del 0 al 9 de la unidad A:

```
Programa para mostrar sectores.
Escriba el numero de la unidad [1=A, 2=B, 3=C, 4=D, 5=E,...]: 1
Número de sector inicial a leer: 0
Número de sectores a leer: 20
```

Esta información se pasa al procedimiento en lenguaje ensamblador, el cual lee los sectores y los coloca en un búfer. El programa en C++ empieza a mostrar el búfer, un sector a la vez. A medida que se muestra cada sector, los caracteres que no son ASCII se sustituyen por puntos. Por ejemplo, a continuación se muestra la visualización del sector 0 de la unidad A:

```
Leyendo sectores 0 - 20 de Unidad 1
Sector 0 -----
.<.(P3j2IHC.....@.....)Y..MYDISK      FAT12    .3.
....{...x..v..V.U."....N.....|.E..F..E.8N$}"....w.r...:f..
|f;..W.u.....V....s.3..F...f..F..V....F....v.`.F..V..  ....^...H...F
..N.a....#.r98-t.`....}.at9Nt... ;.r.....}.....t.<.t.....
...}.....}.....^..f.....}.E..N....F..V.....r....p..B.-`fj.RP.Sj
.j...t....3..v....v.B....v.....V$..d.ar.@u.B.^..Iuw....'..I
nvalid system disk...Disk I/O error...Replace the disk, and then
press any key....IOSYMSDOS    SYS...A....~....@....U.
```

Los sectores continúan mostrándose, uno por uno, hasta que se haya mostrado todo el búfer completo.

Programa en C++ que llama a LeerSector

Ahora mostraremos el programa completo en C++ que llama al procedimiento **LeerSector**:

```
// SectorMain.cpp - Llama al procedimiento LeerSector
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
const int TAM_SECTOR = 512;
extern "C" LeerSector( car * bufer, long sectorInicial,
                      int numUnidad, int numSectores );
void MostrarBufer( const car * bufer, long sectorInicial,
```

```

        int numSectores )
{
    int n = 0;
    long ultimo = sectorInicial + numSectores;
    for(long sNum = sectorInicial; sNum < ultimo; sNum++)
    {
        cout << "\nSector " << sNum
            << " -----"
            << "-----\n";
        for(int i = 0; i < TAM_SECTOR; i++)
        {
            char car = bufer[n++];
            if( unsigned(car) < 32 || unsigned(car) > 127)
                cout << '.';
            else
                cout << car;
        }
        cout << endl;
        getch(); // pausa - espera a que se presione una tecla
    }
}

int main()
{
    char * bufer;
    long sectorInicial;
    int numUnidad;
    int numSectores;

    system("CLS");
    cout << "Programa para mostrar sectores.\n\n"
        << "Escriba el numero de la unidad [1=A, 2=B, 3=C, 4=D, 5=E,...]: ";
    cin >> numUnidad;
    cout << "Numero de sector inicial a leer: ";
    cin >> sectorInicial;
    cout << "Numero de sectores a leer: ";
    cin >> numSectores;
    bufer = new char[numSectores * TAM_SECTOR];

    cout << "\n\nLeyendo sectores " << sectorInicial << " - "
        << (sectorInicial + numSectores) << " de Unidad "
        << numUnidad << endl;

    LeerSector( bufer, sectorInicial, numUnidad, numSectores );
    MostrarBufer( bufer, sectorInicial, numSectores );
    system("CLS");
    return 0;
}

```

En la parte superior del listado encontramos la declaración, o prototipo, de la función **LeerSector**:

```
extern "C" LeerSector( car bufer[], long sectorInicial,
                        int numUnidad, int numSectores );
```

El primer parámetro, *bufer*, es un arreglo de caracteres que guarda los datos del sector, después de leerlo del disco. El segundo parámetro, *sectorInicial*, es el número del sector inicial que se va a leer. El tercer parámetro, *numUnidad*, es el número de la unidad de disco. El cuarto parámetro, *numSectores*, especifica el número de sectores a leer. El primer parámetro se pasa por referencia y todos los demás parámetros se pasan por valor.

En **main** se pide al usuario el número de unidad, sector inicial y número de sectores. El programa también asigna almacenamiento en forma dinámica para el búfer que almacena los datos del sector:

```
cout << "Programa para mostrar sectores.\n\n"
      << "Escriba el numero de la unidad [1=A, 2=B, 3=C, 4=D, 5=E,...]: ";
cin >> numUnidad;
cout << "Numero de sector inicial a leer: ";
cin >> sectorInicial;
cout << "Numero de sectores a leer: ";
cin >> numSectores;
bufer = new char[numSectores * TAM_SECTOR];
```

Esta información se pasa al procedimiento externo **LeerSector**, el cual rellena el búfer con sectores del disco:

```
LeerSector( bufer, sectorInicial, numUnidad, numSectores );
```

Este búfer se pasa a **MostrarBufer**, un procedimiento en el programa en C++ que muestra cada sector en formato de texto ASCII:

```
MostrarBufer( bufer, sectorInicial, numSectores );
```

Módulo en lenguaje ensamblador

A continuación se muestra el módulo en lenguaje ensamblador que contiene el procedimiento **LeerSector**. Como ésta es una aplicación en modo real, debe aparecer la directiva .386 antes de la directiva .MODEL, para indicar al ensamblador que debe crear segmentos de 16 bits:

```
TITLE Lectura de sectores de disco          (LeerSec.asm)
;
; El procedimiento LeerSector se llama desde una aplicación
; de 16 bits en modo real, escrita en Borland C++ 5.01.
; Puede leer discos FAT12, FAT16 y FAT32 bajo
; MS-DOS y Windows 95/98/Me.

Public _LeerSector
.model small
.386

ESDiscos STRUC
    sectorInic    DD ?           ; número de sector inicial
    numSectores   DW 1           ; número de sectores
    despBufer     DW ?           ; desplazamiento del búfer
    segBufer      DW ?           ; segmento del búfer
ESDiscos ENDS

.data
estrucDisco ESDiscos <>

.code
-----
_LLeerSector PROC NEAR C
ARG apuntBufer:WORD, sectorInicial:DWORD, numeroUnidad:WORD, \
     numSectores:WORD
;
; Lee n sectores de una unidad de disco especificada.
; Recibe: apuntador al búfer que almacenará el sector,
;         los datos, número de sector inicial, número de unidad
;         y número de sectores.
; Devuelve: nada
-----
```

```

enter 0,0
pusha
mov eax,sectorInicial
mov estrucDisco.sectorInic,eax
mov ax,numSectores
mov estrucDisco.numSectores,ax
mov ax,apuntBufer
mov estrucDisco.despBufer,ax
push ds
pop es
estrucDisco.segBufer

mov ax,7305h          ; ABSDiskReadWrite
mov cx,0FFFFh          ; siempre tiene este valor
mov dx,numeroUnidad    ; número de unidad
mov bx,OFFSET estrucDisco ; número de sector
mov si,0               ; modo de lectura
int 21h                ; lee el sector del disco
popa
leave
ret
_LeerSector ENDP
END

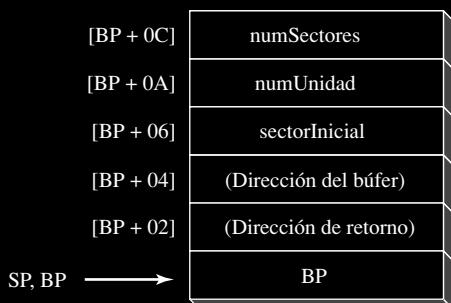
```

Como se utilizó Borland Turbo Assembler para codificar este ejemplo, usamos la palabra clave ARG de Borland para especificar los argumentos del procedimiento. La directiva ARG nos permite especificar los argumentos en el mismo orden que la declaración correspondiente de la función en C++:

ASM:	<code>_LeerSector PROC NEAR C</code>
	<code>ARG apuntBufer:word, sectorInicial:dword, \</code>
	<code>numeroUnidad:word, numSectores:word</code>
C++:	<code>extern "C" LeerSector(char bufer[],</code>
	<code>long sectorInicial, int numUnidad,</code>
	<code>int numSectores);</code>

Los argumentos se meten en la pila en orden inverso, siguiendo la convención de llamadas de C. El más alejado de EBP es **numSectores**, el primer parámetro que se mete en la pila, el cual se muestra en el marco de pila de la figura 12-3. **SectorInicial** es una doble palabra de 32 bits y ocupa las ubicaciones [bp+6] a [bp+09] en la pila. El programa se compiló para el modelo pequeño de memoria, por lo que **bufer** se pasa como un apuntador cercano de 16 bits.

FIGURA 12-3 Procedimiento LeerSector, marco de pila.



12.4.3 Ejemplo: enteros aleatorios grandes

Para mostrar un ejemplo útil de la llamada a una función externa desde Borland C++, podemos llamar a **LongRand**, una función en lenguaje ensamblador que devuelve un entero seudoaleatorio de 32 bits sin signo.

Esto es útil, ya que la función rand() estándar en la biblioteca de Borland C++ sólo devuelve un entero entre 0 y RAND_MAX (32,767). Nuestro procedimiento devuelve un entero entre 0 y 4,294,967,295.

Este programa está compilado en el modelo grande de memoria, lo cual permite que los datos sean mayores de 64K, y se requiere el uso de valores de 32 bits para la dirección de retorno y los valores de los apuntadores de datos. La declaración de la función externa en C++ es:

```
extern "C" unsigned long LongRandom();
```

El listado del programa principal se muestra a continuación. Este programa asigna espacio de almacenamiento para un arreglo llamado **arregloAleat**. Utiliza un ciclo para llamar a **LongRandom**, inserta cada número en el arreglo y lo escribe en la salida estándar:

```
// main.cpp
// Llama a la función externa LongRandom, escrita en
// lenguaje ensamblador, que devuelve un entero aleatorio
// de 32 bits sin signo. Se compila en el modelo grande (Large) de memoria.

#include <iostream.h>
extern "C" unsigned long LongRandom();
const int TAM_ARREGLO = 500;

int main()
{
    // Asigna espacio para el arreglo y rellena con enteros
    // aleatorios sin signo de 32 bits, y muestra.

    unsigned long * arregloAleat = new unsigned long[TAM_ARREGLO];

    for(unsigned i = 0; i < TAM_ARREGLO; i++)
    {
        arregloAleat[i] = LongRandom();
        cout << arregloAleat[i] << ',';
    }
    cout << endl;
    return 0;
}
```

La función LongRandom El módulo en lenguaje ensamblador que contiene la función **LongRandom** es una simple adaptación del procedimiento **Random32** de la biblioteca de enlaces del libro:

```
; Módulo del procedimiento LongRandom          (longrand.asm)

.model large
.386
Public _LongRandom
.data
semilla dd 12345678h

; Devuelve un entero seudoaleatorio sin signo de 32 bits
; en DX:AX, en el rango de 0 - FFFFFFFFh.
.code
_LongRandom PROC far, C
    mov  eax, 343FDh
    mul  semilla
    xor  edx,edx
    add  eax, 269EC3h
    mov  semilla, eax
    ror  eax,8           ; guarda la semilla para la siguiente llamada
                        ; rota a la derecha sacando dígito menos
                        ; significativo
    shld edx,eax,16      ; copia los 16 bits superiores de EAX a DX
```

```

    ret
__LongRandom  endp
end

```

La instrucción ROR ayuda a eliminar los patrones recurrentes cuando se generan enteros aleatorios pequeños. Borland C++ espera que el valor de retorno de la función de 32 bits se encuentre en los registros DX: AX, por lo que copiamos los 16 bits superiores de EAX a DX con la instrucción SHLD, que parece estar convenientemente diseñada para esta tarea.

12.4.4 Repaso de sección

1. ¿Qué registros y banderas deben preservar los procedimientos en lenguaje ensamblador que se llaman desde Borland C++?
2. En Borland C++, ¿cuántos bytes se utilizan para los siguientes tipos? 1) int, 2) enum, 3) float, 4) double.
3. En el módulo LeerSector de esta sección, si no se utilizara la directiva ARG, ¿cómo codificaría la siguiente instrucción?

```
        mov  eax,sectorInicial
```
4. En la función **LongRandom** que se muestra en esta sección, ¿qué pasaría con la salida si se eliminara la instrucción ROR?

12.5 Resumen del capítulo

El lenguaje ensamblador es la herramienta perfecta para optimizar partes selectas de una aplicación grande escrita en algún lenguaje de alto nivel. También es una buena herramienta para personalizar ciertos procedimientos para un hardware específico. Estas técnicas requieren uno de dos métodos:

- Escribir código ensamblador en línea, incrustado dentro del código en lenguaje de alto nivel.
- Enlazar los procedimientos en lenguaje ensamblador con el código en lenguaje de alto nivel.

Ambos métodos tienen sus méritos y sus limitaciones. En este capítulo presentamos ambos.

La convención de nomenclatura utilizada por un lenguaje se refiere a la manera en que se asignan nombres a los segmentos y módulos, así como a las reglas o características relacionadas con el nombramiento de variables y procedimientos. El modelo de memoria utilizado por un programa determina si las llamadas y referencias serán cercanas (dentro del mismo segmento) o lejanas (entre distintos segmentos).

Al llamar a un procedimiento en lenguaje ensamblador desde un programa escrito en otro lenguaje, cualquier identificador que se comparta entre los dos lenguajes debe ser compatible. También debemos usar nombres de segmentos en el procedimiento que sean compatibles con el programa que hace la llamada. El escritor de un procedimiento utiliza la convención de llamadas del lenguaje de alto nivel para determinar cómo recibir los parámetros. La convención de llamadas afecta a la hora de determinar si el procedimiento que se llamó o el programa que hizo la llamada es el que debe restaurar el apuntador de la pila.

En Visual C++, la directiva **asm** se utiliza para escribir código ensamblador en línea en un programa de código fuente en C++. En este capítulo, se utilizó un programa de cifrado de archivos para demostrar el lenguaje ensamblador en línea.

Este capítulo mostró cómo enlazar procedimientos en lenguaje ensamblador con programas en Microsoft Visual C++ que se ejecutan en modo protegido, y programas en Borland C++ que se ejecutan en modo de direccionamiento real.

Al llamar a las funciones de la biblioteca estándar de C (C++), debemos crear un programa auxiliar en C o C++ que contenga una función **main()**. Al iniciarse **main()**, la biblioteca en tiempo de ejecución del compilador se inicializa de manera automática. Desde **main()** podemos llamar a un procedimiento de inicio en el módulo de lenguaje ensamblador. Este módulo puede llamar a cualquier función de la biblioteca estándar de C.

Se escribió un procedimiento en lenguaje ensamblador llamado **BuscarArreglo** y se llamó desde un programa en Visual C++. Comparamos el archivo de código fuente en lenguaje ensamblador generado por el compilador, con el código ensamblado a mano, en nuestro esfuerzo por aprender más acerca de las técnicas

de optimización de código. El programa LeerSector mostró un programa en Borland C++ ejecutándose en modo de direccionamiento real, el cual llama a un procedimiento en lenguaje ensamblador para leer los sectores de un disco.

12.6 Ejercicios de programación

1. Ejemplo MultArreglo

Use el ejemplo BuscarArreglo de la sección 12.3.1 como modelo para este ejercicio. Escriba un procedimiento en lenguaje ensamblador llamado MultArreglo, que multiplique un arreglo de dobles palabras por un entero. Escriba la misma función en C++. Cree un programa de prueba que llame a ambas versiones de MultArreglo desde ciclos y que compare sus tiempos de ejecución.

2. LeerSector, visualización hexadecimal

Requiere un compilador de C++ en modo real de 16 bits, que se ejecute en MS-DOS, Windows 95, 98 o Millennium. Agregue un nuevo procedimiento al programa en C++ de la sección 12.4.2, que llame al procedimiento LeerSector. Este nuevo procedimiento debe mostrar cada sector en hexadecimal. Use iomanip::setfill() para llenar cada byte de salida con un cero a la izquierda.

3. Procedimiento ArregloLongRandom

Use el procedimiento LongRandom de la sección 12.4.3 como punto de inicio para crear un procedimiento llamado ArregloLongRandom que rellene un arreglo con enteros aleatorios de 32 bits sin signo. Debe recibir un apuntador a un arreglo desde un programa en C o C++, junto con una cuenta que indique el número de elementos del arreglo a llenar:

```
extern "C" void ArregloLongRandom( unsigned long * bufer,  
                                    unsigned cuenta );
```

4. Procedimiento TraducirBufer externo

Escriba un procedimiento externo en lenguaje ensamblador que realice el mismo tipo de cifrado que se muestra en el procedimiento TraducirBufer en línea de la sección 12.2.2. Ejecute el programa compilado en el depurador y juzgue si esta versión se ejecuta más rápido que el programa Codifica.cpp de la sección 12.2.2.

5. Programa de números primos

Escriba un procedimiento en lenguaje ensamblador que devuelva un valor de 1 si el entero de 32 bits que se pasa en el registro EAX es primo, y 0 si EAX no es primo. Llame a este procedimiento desde un programa en lenguaje de alto nivel. Permita que el usuario introduzca números muy grandes y haga que su programa muestre un mensaje para cada uno de ellos, indicando si es primo o no.

6. Procedimiento BuscarArregloInv

Modifique el procedimiento BuscarArreglo de la sección 12.3.1. Llame a su función BuscarArregloInv, y deje que busque en forma invertida, desde el final del arreglo. Devuelva el índice del primer valor que coincide, o -1 si no se encuentra el valor.

Notas finales

1. En la versión de Visual C++ que probamos, debe incluir por lo menos una instrucción que utilice valores de punto flotante para forzar a Visual C++ a que cargue el módulo de punto flotante. En caso contrario, se produce el error de “no se cargó el módulo de punto flotante”.
2. Vea la obra de Barry Kauler, Windows Assembly Language and System Programming, CMP Books, 1997.

PROGRAMACIÓN EN MS-DOS de 16 bits

- 13.1 MS-DOS y la IBM-PC
 - 13.1.1 Organización de la memoria
 - 13.1.2 Redirección de entrada-salida
 - 13.1.3 Interrupciones de software
 - 13.1.4 Instrucción INT
 - 13.1.5 Codificación para los programas de 16 bits
 - 13.1.6 Repaso de sección
- 13.2 Llamadas a funciones de MS-DOS (INT 21h)
 - 13.2.1 Funciones de salida selectas
 - 13.2.2 Ejemplo de programa: Hola programador
 - 13.2.3 Funciones de entrada selectas
 - 13.2.4 Funciones de fecha/hora
 - 13.2.5 Repaso de sección
- 13.3 Servicios estándar de E/S de archivos de MS-DOS
 - 13.3.1 Crear o abrir un archivo (716Ch)
 - 13.3.2 Cerrar manejador de archivo (3Eh)
 - 13.3.3 Mover apuntador de archivo (42h)
 - 13.3.4 Obtener la fecha y hora de la creación de un archivo
 - 13.3.5 Procedimientos de biblioteca selectos
 - 13.3.6 Ejemplo: leer y copiar un archivo de texto
 - 13.3.7 Leer la cola de comandos de MS-DOS
 - 13.3.8 Ejemplo: crear un archivo binario
 - 13.3.9 Repaso de sección
- 13.4 Resumen del capítulo
- 13.5 Ejercicios del capítulo

13.1 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 de 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 conocerá la organización básica de la memoria de MS-DOS, cómo activar las llamadas a funciones de MS-DOS (conocidas como *interrupciones*) y cómo realizar operaciones básicas de entrada-salida en el nivel del sistema operativo. Todos los programas en este capítulo se ejecutan en modo de direccionamiento real, debido a que utilizan la instrucción INT. Las interrupciones se diseñaron en un principio para ejecutarse bajo MS-DOS en modo de direccionamiento real. Es posible llamar a las interrupciones en modo protegido, pero las técnicas para hacerlo no se cubrirán en este libro.

Los programas en modo de direccionamiento real tienen las siguientes características:

- Sólo pueden direccionar 1 megabyte de memoria.
- Sólo puede ejecutarse un programa a la vez (una sola tarea) en una sola sesión.
- No es posible la protección de los 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 a las computadoras les fascinó la PC, ya que era una herramienta ideal 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 podían provocar los programas de aplicaciones. En contraste, los sistemas de minicomputadoras disponibles en esa época podían manejar varios usuarios y evitaban que los programas de aplicaciones sobrescribieran los datos unos 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 minicomputadoras, en especial cuando se conectaban varias PCs en red.

13.1.1 Organización de la memoria

En el modo de 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 figura 13-1 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 de software.

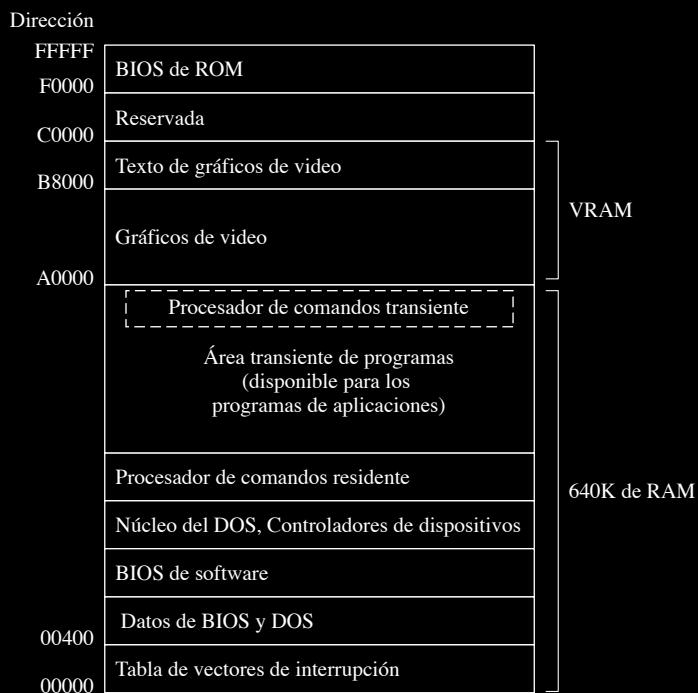
Justo encima de la tabla de vectores se encuentra el *área de datos del BIOS y de MS-DOS*. Después le sigue el *BIOS de software*, que incluye los procedimientos para administrar la mayoría de los dispositivos de E/S, incluyendo el teclado, el disco duro, la pantalla de video, los puertos seriales y 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 búferes 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 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 9FFFF. 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 de gráficos. Cuando el video se encuentra en modo de texto a color, la ubicación de memoria B8000 almacena todo el texto que se muestra en la 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.

FIGURA 13-1 Mapa de memoria de MS-DOS.



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 de IBM, y utilizan el área de datos del BIOS de 00400 a 004FF.

13.1.2 Redirección de entrada-salida

A lo largo de este capítulo haremos referencias 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 del 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 u 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 llamado *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 tabla 13-1 se muestra el conjunto completo de nombres de dispositivos.

Tabla 13-1 Nombres de dispositivos estándar de MS-DOS.

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

13.1.3 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 del 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.

13.1.4 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 la instrucción INT, deben insertarse uno o más parámetros en los registros. Por lo menos, debe moverse 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.

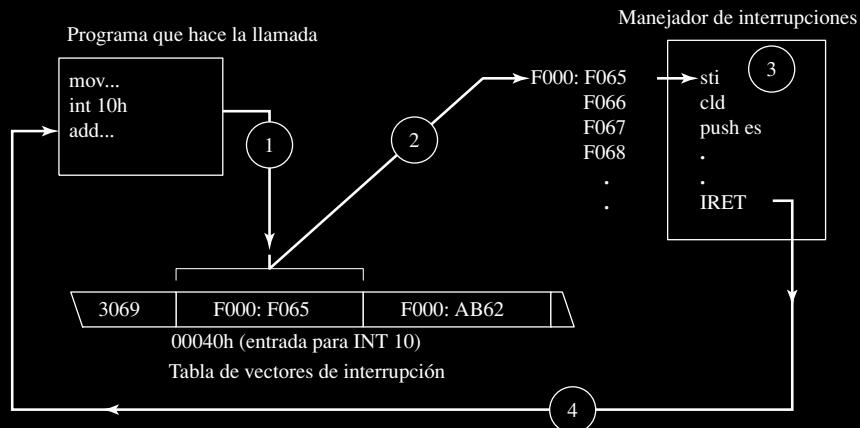
Vectorización de interrupciones

La CPU procesa la instrucción INT mediante el uso de la tabla de vectores de interrupción 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 figura 13-2 ilustra los pasos que realiza la CPU cuando un programa invoca a la instrucción INT:

- **Paso 1:** el operando de la 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.

FIGURA 13-2 Proceso de vectorización de interrupciones.



Interrupciones comunes

Las interrupciones de software llaman a *rutinas de servicio 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 vacío 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*.

13.1.5 Codificación para los programas de 16 bits

Los programas diseñados para MS-DOS deben ser aplicaciones de 16 bits que se ejecuten en modo de direccionamiento real. Las aplicaciones en modo de direccionamiento real utilizan segmentos de 16 bits y siguen el esquema de direccionamiento segmentado descrito en la sección 2.3.1. Si utiliza un procesador de 32 bits, puede usar los registros de propósito general de 32 bits para datos, incluso en el 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. Recomendamos el modelo pequeño (Small), el cual mantiene el código en un segmento y la pila más los datos en otro:
.MODEL smpmall
- 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 los 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 INT 21h:

```
mov ah, 4ch ; termina el proceso  
int 21h ; interrupción 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.
- Al ensamblar programas de 16 bits hay que utilizar el archivo *make16.bat* (procesamiento por lotes). Este archivo crea un vínculo con *Irvine16.lib* y ejecuta el vinculador anterior de 16 bits de Microsoft (versión 5.6).
- Los programas en modo de 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 Millenium. 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 si incluimos el archivo *Irvine16.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 como **exit**:

```
.MODEL small, stdcall  
.STACK 200h  
.386  
exit EQU <.EXIT>
```

13.1.6 Repaso de sección

1. ¿Cuál es la ubicación de memoria más alta en la que se puede cargar un programa de aplicación?
2. ¿Qué ocupa los 1024 bytes inferiores de la memoria?
3. ¿Cuál es la ubicación inicial del área de datos de BIOS y MS-DOS?
4. ¿Cuál es el nombre del área de memoria que contiene los procedimientos de bajo nivel que utiliza la computadora para la entrada-salida?
5. Muestre un ejemplo de cómo redirigir la salida de un programa a la impresora.
6. ¿Cuál es el nombre del dispositivo de MS-DOS para la primera impresora paralela?
7. ¿Qué es una rutina de servicio de interrupción?
8. Cuando se ejecuta la instrucción INT, ¿cuál es la primera tarea que realiza la CPU?
9. ¿Cuáles son los cuatro pasos que realiza la CPU cuando un programa invoca a una instrucción INT? *Sugerencia:* vea la figura 13-2.
10. Cuando termina una rutina de servicio de interrupción, ¿cómo continúa la ejecución un programa de aplicación?
11. ¿Qué número de interrupción se utiliza para los servicios de video?
12. ¿Qué número de interrupción se utiliza para la hora del día?
13. ¿Qué desplazamiento dentro de la tabla de vectores de interrupción contiene la dirección del manejador de interrupciones INT 21h?

13.2 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*, 2^a Edición, Microsoft Press, 1988. Una lista más extensa y actualizada, conocida como *Ralf Brown's Interrupt List*, se encuentra en línea. Visite nuestro sitio Web para obtener detalles acerca de esto.

Para cada función INT 21h que describiremos en este capítulo, presentaremos los parámetros de entrada y valores de retorno necesarios, agregaremos 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, el registro del segmento de datos, por lo general, se establece con el área de datos del programa. Si por alguna razón éste no es el caso, hay que usar el operador SEG para establecer a 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
```

El primer programa en lenguaje ensamblador para procesadores Intel que escribí (aproximadamente en 1983) mostraba un “*” en la pantalla:

```
mov ah,2
mov dl,'*'
int 21h
```

La gente decía que el lenguaje ensamblador era difícil, pero ¡esto era alentador! Después hubo algunos detalles más que tuve que aprender antes de escribir programas que no fueran triviales.

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 de direccionamiento real que presentamos en este libro, nos hemos basado en la definición de una macro en la biblioteca Irvine16 llamada **exit**. Ésa se define así:

```
exit TEXTEQU <.EXIT>
```

En otras palabras, **exit** es un alias o sustituto para .EXIT (la directiva de MASM que termina un programa). Creamos el símbolo **exit** con el fin de que usted pudiera usar un solo comando para terminar los programas de 16 y 32 bits. En los 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 ; llamada 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. Pueden usarse otros códigos de retorno entre 1 y 255 para indicar resultados adicionales que tengan un significado específico para cada programa. Por ejemplo, ML.EXE, el Microsoft Assembler, devuelve 0 si un programa se ensambla en forma adecuada, y un valor distinto de cero si no es así.

El apéndice C contiene una lista bastante extensa de interrupciones del BIOS y de MS-DOS.

13.2.1 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 del capítulo 15.

Filtrado de los caracteres de control Todas las funciones en esta sección *filtran*, o interpretan los 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 tabla 13-2 contiene una lista de los caracteres de control que es probable que nos encontremos.

Tabla 13-2 Caracteres ASCII de control.

Código ASCII	Descripción
08h	Retroceso (se mueve una columna a la izquierda)
09h	Tabulación horizontal (avanza <i>n</i> columnas hacia delante)
0Ah	Avance de línea (se mueve a 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 INT 21h 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 delante
Recibe	AH = 2 DL = valor del carácter
Devuelve	Nada
Llamada de ejemplo	<pre>mov ah, 2 mov dl, 'A' int 21h</pre>

Función 5 de INT 21h	
Descripción	Escribe un solo carácter en la impresora
Recibe	AH = 5 DL = valor del carácter
Devuelve	Nada
Llamada de ejemplo	<code>mov ah, 5 ; selecciona la salida de impresora mov dl, "Z" ; carácter a imprimir int 21h ; llamada a MS-DOS</code>
Notas	MS-DOS espera hasta que la impresora esté lista para imprimir el carácter. Puede terminar la espera oprimiendo las teclas Ctrl-Inter. La salida predeterminada es al puerto de impresora LPT1

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	<code>mov ah, 6 mov dl, "A" int 21h</code>
Notas	A diferencia de otras funciones de INT 21h, ésta no filtra (interpreta) los caracteres ASCII de control

Función 9 de INT 21h	
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	<code>.data cadena BYTE "Esta es una cadena\$" .code mov ah,9 mov dx,OFFSET cadena int 21h</code>
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 del arreglo
Devuelve	AX = número de bytes escritos
Llamada de ejemplo	<pre>.data mensaje "Hola, programador" .code mov ah,40h mov bx,1 mov cx,LENGTHOF mensaje mov dx,OFFSET mensaje int 21h</pre>

13.2.2 Ejemplo de programa: Hola programador

A continuación se muestra un programa simple, 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 ax,@data                      ; inicializa DS
    mov ds,ax
    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 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 enseguida de la directiva END:

```
TITLE Programa Hola programador          (Hola2.asm)
.MODEL small
.STACK 100h
.386
.data
mensaje BYTE "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 búfer
    int 21h

    .EXIT
main ENDP
END

```

13.2.3 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. Para una lista más completa, consulte el apéndice C. Como se muestra en la siguiente tabla, la función 1 de INT 21h lee un solo carácter 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	<pre> mov ah,1 int 21h mov car,al </pre>
Notas	Si no hay un carácter presente en el búfer de entrada, el programa espera. Esta función envía (echo) el carácter a la 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 se realiza ninguna otra acción:

Función 6 de INT 21h	
Descripción	Lee un carácter de la entrada estándar sin esperar
Recibe	AH = 6 DSL = FFh
Devuelve	Si ZF = 0, AL contiene el código ASCII del carácter
Llamada de ejemplo	<pre> mov ah,6 mov dl,0FFh int 21h jz saltar mov car,AL saltar: </pre>
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 estar entre 0 y 128):

```
cuenta = 80
TECLADO STRUCT
    maxEntrada BYTE cuenta          ; caracteres máximos a introducir
    cuentaEntrada BYTE ?           ; cuenta actual de entrada
    bufer 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. Puede utilizarse la tecla retroceso para borrar caracteres y retroceder el cursor. El usuario termina la entrada oprimiendo la tecla Intro u oprimiendo Ctrl-Inter. 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 cuántos caracteres se introdujeron, sin contar la tecla Intro. La siguiente tabla describe la función 0Ah:

Función 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	<pre>.data datosTec1 TECLADO <> .code mov ah,0Ah mov dx,OFFSET datosTec1 int 21h</pre>

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	<pre>mov ah,08h int 21h cmp al,0 je saltar ; (introduce el carácter) saltar:</pre>
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 le puede dar un buen uso, si ejecutamos un programa desde el símbolo del 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,VALXOR             ; cifra el carácter
    mov ah,6                  ; escribe en la salida
    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. Puede usar cualquier valor entre 0 y 255 en este contexto, aunque si utiliza 0 no se producirá ningún cifrado. Desde luego que el cifrado es débil, pero podría ser suficiente como para desalentar al usuario promedio al tratar de manejarlo. Cuando ejecute el programa en el símbolo del sistema, indique el nombre del archivo de entrada (y de salida, si lo hay). A continuación se muestran dos ejemplos:

cifrar < archent.txt	Entrada desde archivo (archent.txt), salida a la consola
cifrar < archent.txt > archsal.txt	Entrada desde 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 del teclado cuando el manejador del dispositivo en BX es igual a cero:

Función 3Fh de INT 21h	
Descripción	Lee un arreglo de bytes de un archivo o dispositivo
Recibe	AH = 3Fh BX = manejador de dispositivo/archivo (0 = teclado) CX = máximo de bytes a leer DS: DX = dirección de búfer de entrada
Devuelve	AX = número de bytes que se leyeron
Llamada de ejemplo	<pre>.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</pre>
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 su programa funciona en la forma esperada, debe vaciar el búfer de entrada, un carácter a la vez, después de llamar a la función 3Fh. El siguiente código se encarga de ello (vea el programa *Teclado.asm* para una demostración completa):

```
;-----
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              ; llamada a MS-DOS
    cmp unByte,0Ah        ; ¿llegó al fin de línea?
    jne L1                ; no: lee otro
    popa
    ret
VaciarBufer ENDP
```

13.2.4 Funciones de fecha/hora

Muchas aplicaciones populares de software muestran 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 2Bh de INT 21h establece la fecha del sistema. La función 2Ch de INT 21h obtiene la hora del sistema, y la función 2Dh 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	<pre>mov ah, 2Ah int 21h mov anio,cx mov mes,dh mov dia,dl mov diaSemana,a1</pre>

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	<pre>mov ah,2Bh mov cx,anio mov dh,mes mov dl,dia int 21h cmp al,0 jne fallo</pre>
Notas	Es probable que no funcione si ejecuta 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	<pre>mov ah, 2Ch int 21h mov horas, ch mov minutos, cl mov segundos, dh</pre>

Función 2Dh 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	<pre>mov ah, 2Dh mov ch, horas mov cl, minutos mov dh, segundos int 21h cmp al, 0 jne fallo</pre>
Notas	Es probable que no funcione si ejecuta Windows NT, 2000 o XP con un perfil de usuario restringido

Ejemplo: mostrar la hora y la fecha

El siguiente programa (*FechaHora.asm*) muestra la fecha y hora del sistema. El código es un poco más largo de lo esperado, ya que el programa inserta ceros a la izquierda en las horas, minutos y segundos:

```
TITLE Muestra la fecha y la hora          (FechaHora.asm)

Include Irvine16.inc
Escribir PROTO car:BYTE
.data
cad1 BYTE "Fecha: ",0
cad2 BYTE ", Hora: ",0
```

```
.code
main PROC
    mov    ax,@data
    mov    ds,ax

    ; Muestra la fecha:
    mov    dx,OFFSET cad1
    call   WriteString
    mov    ah,2Ah                ; obtiene la fecha del sistema
    int    21h
    movzx  eax,dh                ; mes
    call   WriteDec
    INVOKE Escribir,'-'
    movzx  eax,d1                ; dia
    call   WriteDec
    INVOKE Escribir,'-'
    movzx  eax,cx                ; año
    call   WriteDec

    ; Muestra la hora:
    mov    dx,OFFSET cad2
    call   WriteString
    mov    ah,2Ch                ; obtiene la hora del sistema
    int    21h
    movzx  eax,ch                ; horas
    call   EscribirDecRelleno
    INVOKE Escribir,':'
    movzx  eax,cl                ; minutos
    call   EscribirDecRelleno
    INVOKE Escribir,':'
    movzx  eax,dh                ; segundos
    call   EscribirDecRelleno
    call   Crlf

    exit
main ENDP

;-----
Escribir PROC car:BYTE
; Muestra un solo caracter.
;-----
    push  eax
    push  edx
    mov   ah,2                  ; función para escribir un caracter
    mov   dl,car
    int   21h
    pop   edx
    pop   eax
    ret
Escribir ENDP

;-----
EscribirDecRelleno PROC
; Muestra el entero sin signo en EAX, rellena
; hasta dos posiciones de digito con cero a la izquierda.
;-----
    .IF eax < 10
```

```

push  eax
push  edx
mov   ah,2          ; escribe un cero a la izquierda
mov   dl,'0'
int   21h
pop   edx
pop   eax
.ENDIF
call  WriteDec      ; escribe decimal sin signo
ret               ; usando el valor en EAX
EscribirDecRelleno ENDP
END main

```

Resultados de ejemplo:

Fecha: 12-8-2006, Hora: 23:01:23

13.2.5 Repaso de sección

1. ¿Qué registro guarda el número de la función cuando se hace una llamada a INT 21h?
2. ¿Qué función INT 21h termina un programa?
3. ¿Qué función INT 21h escribe un solo carácter a la salida estándar?
4. ¿Qué función INT 21h escribe una cadena que se termina con un \$, a la salida estándar?
5. ¿Qué función INT 21h escribe un bloque de datos a un archivo o dispositivo?
6. ¿Qué función INT 21h lee un solo carácter de la entrada estándar?
7. ¿Qué función INT 21h lee un bloque de datos del dispositivo de entrada estándar?
8. Si desea obtener la fecha del sistema, mostrarla y después modificarla, ¿qué funciones INT 21h se requieren?
9. ¿Qué funciones INT 21h que se mostraron en este capítulo tal vez no funcionen en Windows NT, 2000 o XP, con un perfil de usuario restringido?
10. ¿Qué función INT 21h utilizaría para ver si en el búfer de entrada estándar hay un carácter esperando a ser procesado?

13.3 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 los que podemos mostrar en este libro. La tabla 13-3 muestra unas cuantas de las funciones que se utilizan con más frecuencia.

Tabla 13-3 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 del 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 redirección en el símbolo del 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 tabla 13-4 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. Busque *Windows 9x* en la documentación del SDK de la plataforma.

Tabla 13-4 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 especificó 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
1A	Tipo de medios desconocido
1B	No se encontró el sector
1C	Se agotó el papel de la impresora
1D	Fallo al escribir
1E	Fallo al leer
1F	Fallo general

13.3.1 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 de archivos. 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 = sólo lectura, 2 = oculto, 3 = sistema, 8 = ID de volumen, 20h = archivo) DX = acción (1 = abrir, 2 = truncar, 10h = crear) DS:SI = segmento/desplazamiento del nombre del archivo DI = sugerencia de alias (opcional)
Devuelve	Si la creación/apertura tuvo éxito, CF = 0, AX = manejador del archivo y CX = acción realizada. Si la creación/apertura falló, CF = 1
Llamada de ejemplo	<pre> 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 Nombreachivo int 21h jc fallo mov manejador,ax ; manejador del archivo mov accionRealizada,cx ; acción realizada </pre>
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_REPLACE_OPENED. Todas estas constantes están definidas en Irvine16.inc

Ejemplos adicionales El siguiente código crea un nuevo 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
  
```

El siguiente código trata de crear un nuevo archivo. Falla (con la bandera Acarreo activa) si ya existe:

```

    mov ax,716Ch           ; Abrir/Crear extendida
    mov bx,2               ; lectura-escritura
    mov cx,0               ; atributo normal
  
```

```

mov dx,10h           ; acción: crear
mov si,OFFSET ArchivoNuevo
int 21h
jc fallo
mov manejador,ax    ; manejador del archivo
mov accionRealizada,cx ; acción realizada para abrir el archivo

```

13.3.2 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 del archivo
Recibe	AH = 3Eh 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

13.3.3 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	<pre> mov ah,42h mov al,0 ; método: desplazamiento ; desde el principio ; BX = manejador ; CX,desplazamientoSup ; MOV DX,desplazamientoInf ; int 21h </pre>
Notas	El desplazamiento devuelto del apuntador del archivo en DX:AX siempre es relativo al principio del archivo

13.3.4 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. Para aprender acerca de los formatos de fecha y hora empaquetados de MS-DOS, vea la sección 14.3.1. Para ver un ejemplo de cómo extraer los campos de fecha/hora, vea la sección 7.3.4.

Función 5706h de INT 21h	
Descripción	Obtiene la fecha y hora de la 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	<pre>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</pre>
Notas	El archivo ya debe estar abierto. El valor <i>milisegundos</i> indica el número de intervalos 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

13.3.5 Procedimientos de biblioteca selectos

A continuación se muestran 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 (*Nota: se tradujeron los comentarios en estos ejemplos de código para facilitar su comprensión al lector. Los procedimientos originales están en inglés*):

```
;-----
; 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 seleen (CX-1) caracteres.
; Bug fixed 9/22/03: return valve was tuolarge by 1
;-----
        push cx                      ; guarda los registros
        push si
        push cx
        mov  si,dx
        dec  cx
        mov  ah,1
        int  21h
        cmp  al,0Dh
        je   L2
        push cx
        mov  si,dx
        dec  cx
        mov  al,' '
        mov  dx,si
        int  21h
        jc   L1
        pop  cx
        pop  si
        pop  cx
        ret
```

```

    mov [si],al           ; no: guarda el carácter
    inc si               ; incrementa el apuntador al búfer
    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               ; se agregó el 9/22/03
    pop si               ; restaura los registros
    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
; Last update: 08/02/2002
;-----

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

```

13.3.6 Ejemplo: leer y copiar un archivo de texto

En este capítulo presentamos antes la función 3Fh de INT 21h, en el contexto de una operación de lectura de la entrada estándar. Esta función también puede leer un archivo si el manejador en BX identifica a uno que se haya abierto en modo de entrada. Cuando la función 3Fh regresa, AX indica el número de bytes que se leyeron del archivo. Cuando se llega al final del archivo, el valor devuelto en AX es menor que el número de bytes solicitados (en CX).

También presentamos la función 40h de INT 21h, en el contexto de una operación de escritura a la salida estándar (manejador de dispositivo 1). En vez de ello, el manejador en BX puede hacer referencia a un archivo abierto. La función actualiza de manera automática el apuntador de posición del archivo, por lo que la siguiente llamada a la función 40h empieza a escribir en donde se quedó la llamada anterior.

El programa *LeerArchivo.asm* que veremos demuestra varias funciones de INT 21h que presentamos en esta sección:

- Función 716Ch: crea un nuevo archivo o abre uno existente.
- Función 3Fh: lee de un archivo o dispositivo.
- Función 40h: escribe a un archivo o dispositivo.
- Función 3Eh: cierra el manejador del archivo.

El siguiente programa abre un archivo de texto en modo de entrada, lee no más de 5,000 bytes del archivo, lo muestra en la consola, crea un nuevo archivo y copia los datos al nuevo archivo:

```

TITLE Lee un archivo de texto      (LeerArchivo.asm)
; Lee, muestra y copia un archivo de texto.
INCLUDE Irvine16.inc

.data
TamBuf =    5000
archent   BYTE "mi_archivo_texto.txt",0
archsal    BYTE "mi_archivo_salida.txt",0
manejadorEnt WORD ?
manejadorSal WORD ?
bufer     BYTE TamBuf DUP(?)
bytesLeidos WORD ?

.code
main PROC
    mov ax,@data
    mov ds,ax

; Abre el archivo de entrada
    mov ax,716Ch          ; abrir o crear extendido
    mov bx,0               ; modo = sólo lectura
    mov cx,0               ; atributo normal
    mov dx,1               ; acción: abrir
    mov si,OFFSET archent
    int 21h                ; llamada a MS-DOS
    jc terminar            ; termina si hay error
    mov manejadorEnt,ax

; Lee el archivo de entrada
    mov ah,3Fh              ; lee archivo o dispositivo
    mov bx,manejadorEnt      ; manejador del archivo
    mov cx,TamBuf            ; máximo de bytes a leer
    mov dx,OFFSET bufer       ; apuntador al búfer
    int 21h
    jc terminar            ; termina si hay error
    mov bytesLeidos,ax

; Muestra el búfer
    mov ah,40h              ; escribe en archivo o dispositivo
    mov bx,1                 ; manejador de salida de consola
    mov cx,bytesLeidos        ; número de bytes
    mov dx,OFFSET bufer       ; apuntador al búfer
    int 21h
    jc terminar            ; termina si hay error

; Cierra el archivo
    mov ah,3Eh              ; función: cerrar archivo
    mov bx,manejadorEnt      ; manejador de archivo de entrada
    int 21h
    jc terminar            ; termina si hay error

; Crea el archivo de salida
    mov ax,716Ch          ; crear o abrir extendido
    mov bx,1               ; modo = sólo escritura
    mov cx,0               ; atributo normal
    mov dx,12h              ; acción: crear/truncar
    mov si,OFFSET archsal
    int 21h                ; llamada a MS-DOS
    jc terminar            ; termina si hay error
    mov manejadorSal,ax     ; guarda el manejador

; Escribe búfer en nuevo archivo
    mov ah,40h              ; escribe en archivo o dispositivo

```

```

    mov  bx,manejadorSal      ; manejador de archivo de salida
    mov  cx,bytesLeidos      ; número de bytes
    mov  dx,OFFSET bufer      ;apuntador al búfer
    int  21h
    jc   terminar            ; termina si hay error

; Cierra el archivo
    mov  ah,3Eh               ; función: cerrar archivo
    mov  bx,manejadorSal      ; manejador de archivo de salida
    int  21h                  ; llamada a MS-DOS

terminar:
    call Crlf
    exit
main ENDP
END main

```

13.3.7 Leer la cola de comandos de MS-DOS

En los programas que mostraremos a continuación, pasaremos con frecuencia información a los programas en la línea de comandos. Suponga que tenemos que pasar el nombre *archivo1.doc* a un programa llamado *atrib.exe*. La línea de comandos de MS-DOS sería:

```
atrib 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 memoria, en el 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). En la sección 16.3.1 hablaremos sobre el prefijo de segmento de programa. Consulte también la sección 2.3.1 para ver cómo funciona el direccionamiento segmentado en modo de direccionamiento real.

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 *atrib.exe*, el contenido hexadecimal de la cola de comandos sería el siguiente:

Desplazamiento:	80 81 82 83 84 85 86 87 88 89 8A 8B																								
Contenido:	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0A</td><td>20</td><td>46</td><td>49</td><td>4C</td><td>45</td><td>31</td><td>2E</td><td>44</td><td>4F</td><td>43</td><td>0D</td> </tr> <tr> <td>F</td><td>I</td><td>L</td><td>E</td><td>I</td><td>.</td><td>D</td><td>O</td><td>C</td><td></td><td></td><td></td></tr> </table>	0A	20	46	49	4C	45	31	2E	44	4F	43	0D	F	I	L	E	I	.	D	O	C			
0A	20	46	49	4C	45	31	2E	44	4F	43	0D														
F	I	L	E	I	.	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.

Para establecer los parámetros de la línea de comandos en CodeView, seleccione *Set Runtime Arguments...* del menú *Run*. Oprima F10 para ejecutar la primera instrucción del programa, abra una ventana de memoria, seleccione *Memory* del menú *Options* y escriba ES:0x80 en el campo *Address Expression*.

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 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 asignar a DX el desplazamiento del búfer en el que se va a copiar la cola de comandos. A menudo, los

programas 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 del 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 SCASB 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 delante
mov a1,20h ; 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 ; guarda los registros generales
    pusha
    mov ah,62h ; obtiene 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 ; sí: termina
    cld ; explora en dirección hacia delante
    mov a1,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 utilizar mejor la dirección de segmento en 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=0 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

```

13.3.8 Ejemplo: creación un archivo binario

Un *archivo binario* recibe ese nombre debido a que los datos almacenados en él son sólo una imagen binaria de los datos de un programa. Por ejemplo, suponga que su programa creó y llenó un arreglo de dobles palabras:

```
miArreglo DWORD 50 DUP(?)
```

Si quisiera escribir este arreglo en un archivo de texto, tendría que convertir cada entero en una cadena y escribirlo por separado. Una manera más eficiente de almacenar estos datos sería tan sólo escribir una imagen binaria de *miArreglo* en un archivo. Un arreglo de 50 dobles palabras utiliza 200 bytes de memoria, y esa cantidad de espacio de almacenamiento en el disco es exactamente lo que utilizaría el archivo.

El siguiente programa *ArchBin.asm* llena un arreglo con enteros aleatorios, los muestra en la pantalla, escribe los enteros en un archivo binario y cierra ese archivo. Después vuelve a abrir el archivo, lee los enteros y los muestra en la pantalla:

```

TITLE Programa de archivos binarios      (ArchBin.asm)
; Este programa crea un archivo binario que contiene
; un arreglo de dobles palabras. Después lee el archivo
; de vuelta y muestra los valores.

INCLUDE Irvine16.inc

.data
miArreglo DWORD 50 DUP(?)

nombreArchivo BYTE "archivo arreglo binario.bin",0
manejadorArchivo WORD ?
cadComa        BYTE ", ",0

; Establezca CreateFile a cero si sólo desea
; leer y mostrar el archivo binario existente.
CreateFile = 1

.code
main PROC
    mov ax,@data
    mov ds,ax

    .IF CreateFile EQ 1
        call LlenarElArreglo
        call MostrarElArreglo
        call CrearElArchivo
    .ENDIF

```

```
    call WaitMsg
    call CrLf
.ENDIF
    call LeerElArchivo
    call MostrarElArreglo

terminar:
    call CrLf
    exit
main ENDP

;-----
LeerElArchivo PROC
;
; Abre y lee el archivo binario.
; Recibe: nada.
; Devuelve: nada
;-----
    mov ax,716Ch          ; abrir archivo extendido
    mov bx,0               ; modo: sólo lectura
    mov cx,0               ; atributo: normal
    mov dx,1               ; abre archivo existente
    mov si,OFFSET nombreArchivo ; nombre del archivo
    int 21h               ; llamada a MS-DOS
    jc terminar           ; termina si hay error
    mov manejadorArchivo,ax ; guarda el manejador

; Lee el archivo de entrada y después cierra el archivo.
    mov ah,3Fh             ; lee archivo o dispositivo
    mov bx,manejadorArchivo ; manejador del archivo
    mov cx,SIZEOF miArreglo ; máximo de bytes a leer
    mov dx,OFFSET miArreglo ; apuntador al búfer
    int 21h
    jc terminar           ; termina si hay error
    mov ah,3Eh             ; función: cerrar archivo
    mov bx,manejadorArchivo ; manejador del archivo de salida
    int 21h               ; llamada a MS-DOS

terminar:
    ret
LeerElArchivo ENDP

;-----
MostrarElArreglo PROC
;
; Muestra el arreglo de dobles palabras.
; Recibe: nada.
; Devuelve: nada
;-----
    mov CX,LENGTHOF miArreglo
    mov si,0

L1:
    mov eax,miArreglo[si]      ; obtiene un número
    call WriteHex              ; muestra el número
    mov edx,OFFSET cadComa     ; muestra una coma
    call WriteString            ; siguiente posición del arreglo
    add si,TYPE miArreglo
    loop L1
    ret

MostrarElArreglo ENDP
```

```

;-----
LlenarElArreglo PROC
;
; Llena el arreglo con enteros aleatorios.
; Recibe: nada.
; Devuelve: nada
;-----
    mov    CX,LENGTHOF miArreglo
    mov    si,0
L1:
    mov    eax,1000          ; genera enteros aleatorios
    call   RandomRange      ; entre 0 - 999 en EAX
    mov    miArreglo[si],eax ; los almacena en el arreglo
    add    si,TYPE miArreglo ; siguiente posición del arreglo
    loop   L1
    ret
LlenarElArreglo ENDP

;-----
CrearElArchivo PROC
;
; Crea un archivo que contiene datos binarios.
; Recibe: nada.
; Devuelve: nada
;-----
    mov    ax,716Ch          ; crea el archivo
    mov    bx,1                ; modo: sólo escritura
    mov    cx,0                ; archivo normal
    mov    dx,12h              ; acción: crear/truncar
    mov    si,OFFSET nombreArchivo ; nombre del archivo
    int    21h
    jc     terminar          ; termina si hay error
    mov    manejadorArchivo,ax ; guarda el manejador

; Escribe el arreglo de enteros en el archivo.
    mov    ah,40h              ; escribe en archivo o dispositivo
    mov    bx,manejadorArchivo ; manejador del archivo de salida
    mov    cx,SIZEOF miArreglo ; número de bytes
    mov    dx,OFFSET miArreglo ; apuntador al búfer
    int    21h
    jc     terminar          ; termina si hay error

; Cierra el archivo.
    mov    ah,3Eh              ; función: cerrar archivo
    mov    bx,manejadorArchivo ; manejador del archivo de salida
    int    21h
;-----
terminar:
    ret
CrearElArchivo ENDP
END main

```

Hay que destacar que el proceso de escribir todo el arreglo se realiza con una sola llamada a la función 40h de INT 21h. No hay necesidad de un ciclo:

```

mov    ah,40h          ; escribe en archivo o dispositivo
mov    bx,manejadorArchivo ; manejador del archivo de salida
mov    cx,SIZEOF miArreglo ; número de bytes
mov    dx,OFFSET miArreglo ; apuntador al búfer
int    21h

```

Lo mismo aplica cuando se lee el archivo y se colocan los datos de vuelta en el arreglo. Una sola llamada a la función 3Fh de INT 21h se encarga del trabajo:

```
mov ah,3Fh ; lee archivo o dispositivo
mov bx,manejadorArchivo ; manejador del archivo
mov cx,SIZEOF miArreglo ; máximo de bytes a leer
mov dx,OFFSET miArreglo ; apuntador al búfer
int 21h
```

13.3.9 Repaso de sección

1. Mencione los cinco manejadores de dispositivos estándar de MS-DOS.
2. Después de llamar a una función de E/S de MS-DOS, ¿qué bandera indica que se produjo un error?
3. Al llamar a la función 716Ch para crear un archivo, ¿qué argumentos se requieren?
4. Muestre un ejemplo de cómo abrir un archivo existente en modo de entrada.
5. Al llamar a la función 716Ch para leer un arreglo binario de un archivo que ya se encuentra abierto, ¿qué valores se requieren para los argumentos?
6. ¿Cómo comprobamos el fin de archivo al leer un archivo de entrada usando la función 3Fh de INT 21h?
7. Al llamar a la función 3Fh, ¿qué diferencia hay entre leer un archivo y leer datos del teclado?
8. Si deseamos leer un archivo de acceso aleatorio, ¿qué función INT 21h nos permite saltar directamente a un registro específico en medio del archivo?
9. Escriba un segmento de código corto para posicionar el apuntador a 50 bytes del principio de un archivo. Suponga que el archivo ya se encuentra abierto, y que BX contiene el manejador del archivo.

13.4 Resumen del capítulo

En este capítulo vimos la organización básica de memoria de MS-DOS, cómo activar las llamadas a las funciones de MS-DOS, y cómo realizar operaciones básicas de entrada-salida al nivel del sistema operativo.

Al dispositivo de entrada estándar y al dispositivo de salida estándar se les conoce en conjunto como la *consola*, en la que se utiliza el teclado para la entrada y la pantalla de video para la salida.

Una *interrupción de software* es una llamada a un procedimiento del sistema operativo. La mayoría de estos procedimientos, conocidos como *manejadores de interrupciones*, proporcionan la capacidad de entrada-salida a los programas de aplicaciones.

La instrucción INT (llamada a un procedimiento de interrupción) mete las banderas de la CPU y la dirección de retorno de 32 bits (CS e IP) en la pila, deshabilita las demás interrupciones y llama a un manejador de interrupción. La CPU procesa la instrucción INT mediante el uso de la *tabla de vectores de interrupción*, una tabla que contiene direcciones de segmento-desplazamiento de 32 bits de manejadores de interrupciones.

Los programas diseñados para MS-DOS deben ser aplicaciones de 16 bits que se ejecuten en modo de direccionamiento real. Las aplicaciones en modo de direccionamiento real usan segmentos de 16 bits y utilizan el direccionamiento segmentado.

La directiva .MODEL especifica el modelo de memoria que utilizará nuestro programa. La directiva .STACK asigna una pequeña cantidad de espacio de almacenamiento local para nuestro programa. En el modo de direccionamiento real, las entradas en la pila son de 16 bits, de manera predeterminada. Para habilitar el uso de los registros de 32 bits se utiliza la directiva .386.

Una aplicación de 16 bits que contenga variables debe asignar a DS la ubicación del segmento de datos, antes de poder acceder a las variables.

Todo programa debe incluir una instrucción que termine el programa y regrese al sistema operativo. Una manera de hacer esto es mediante la directiva .EXIT. Otra forma es llamar a la función 4Ch de INT 21h.

Cualquier programa en modo de direccionamiento real puede acceder a los puertos de hardware, los vectores de interrupción y al sistema de memoria al ejecutarse en MS-DOS, Windows 95, 98 y Millenium. Por otro lado, este tipo de acceso sólo se otorga a los programas en modo de núcleo y a los controladores de dispositivos en Windows NT, 2000 y XP.

Al ejecutarse un programa, cualquier texto adicional en su línea de comandos se almacena de manera automática en el área de la cola de comandos de MS-DOS de 128 bytes, en el desplazamiento 80h de un segmento especial de memoria conocido como *prefijo de segmento de programa* (PSP). El procedimiento **GetCommandTail** de la biblioteca Irvine16 devuelve una copia de la cola de comandos. En la sección 16.3.1 hablaremos sobre el prefijo de segmento de programa.

A continuación se muestran algunas interrupciones del BIOS de uso frecuente:

- INT 10h, Servicios de video: procedimientos que muestran rutinas para controlar la posición del cursor, escribir texto a color, desplazar la pantalla y mostrar 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: un procedimiento que obtiene el número de pulsaciones de reloj desde que el equipo se encendió, o establece el contador a un nuevo valor.
- INT 1Ch, Interrupción de temporizador del usuario: un procedimiento vacío que se ejecuta 18.2 veces por segundo.

A continuación se presenta una variedad de funciones importantes de MS-DOS (INT 21h):

- INT 21h, Servicios de MS-DOS. Procedimientos que ofrecen entrada-salida, manejo de archivos y administración de la memoria. También se conocen como llamadas a funciones de MS-DOS.
- INT 21h cuenta con soporte para cerca de 200 funciones distintas, las cuales se identifican mediante un número de función que se coloca en el registro AH.
- La función 4Ch de INT 21h termina el programa actual (conocido como proceso).
- Las funciones 2 y 6 de INT 21h escriben un solo carácter en la salida estándar.
- La función 5 de INT 21h escribe un solo carácter en la impresora.
- La función 9 de INT 21h escribe una cadena en la salida estándar.
- La función 40h de INT 21h escribe un arreglo de bytes en un archivo o dispositivo.
- La función 1 de INT 21h lee un solo carácter de la entrada estándar.
- La función 6 de INT 21h lee un carácter de la entrada estándar sin esperar.
- La función 0Ah de INT 21h lee una cadena con búfer de la entrada estándar.
- La función 0Bh de INT 21h obtiene el estado del búfer de entrada estándar.
- La función 3Fh de INT 21h lee un arreglo de bytes de un archivo o dispositivo.
- La función 2Ah de INT 21h obtiene la fecha del sistema.
- La función 2Bh de INT 21h establece la fecha del sistema.
- La función 2Ch de INT 21h obtiene la hora del sistema.
- La función 2Dh de INT 21h establece la hora del sistema.
- La función 716Ch de INT 21h crea un archivo o abre uno existente.
- La función 3Eh de INT 21h cierra un manejador de archivo.
- La función 42h de INT 21h mueve el apuntador de posición de un archivo.
- La función 5706h de INT 21h obtiene la fecha y hora de creación de un archivo.
- La función 62h de INT 21h devuelve la porción correspondiente al segmento de la dirección del prefijo de segmento del programa.

Los siguientes programas de ejemplo demostraron cómo aplicar las funciones de MS-DOS:

- El programa *FechaHora.asm* muestra la fecha y hora del sistema.
- El programa *LeerArchivo.asm* abre un archivo de texto en modo de entrada, lee el archivo, lo muestra en la consola, crea uno nuevo y copia los datos al nuevo archivo.
- El programa *ArchBin.asm* llena un arreglo con enteros aleatorios, muestra los enteros en la pantalla, los escribe en un archivo binario y cierra el archivo. Después vuelve a abrirlo, lee los enteros y los muestra en la pantalla.

Un archivo binario recibe ese nombre debido a que los datos que se almacenan en éste son una imagen binaria de los datos de un programa.

13.5 Ejercicios del capítulo

Los siguientes ejercicios deben realizarse en modo de direccionamiento real. No utilice funciones de la biblioteca Irvine16. Use las llamadas a funciones de INT 21h para toda la entrada-salida, a menos que un ejercicio indique específicamente otra cosa.

1. Leer un archivo de texto

Abra un archivo en modo de entrada, lea su contenido y muéstrela en la pantalla, en hexadecimal. El búfer de entrada debe ser pequeño (un valor aproximado de 256 bytes), de manera que el programa utilice un ciclo para repetir la llamada a la función 3Fh todas las veces que sea necesario, hasta que se haya procesado todo el archivo.

2. Copiar un archivo de texto

Modifique el programa **LeerArchivo** de la sección 13.3.6, de manera que pueda leer un archivo de cualquier tamaño. Suponiendo que el búfer sea más pequeño que el archivo de entrada, utilice un ciclo para leer todos los datos. Use un tamaño de búfer de 256 bytes. Muestre los mensajes de error apropiados si la bandera Acarreo se activa después de cualquier llamada a una función INT 21h.

3. Establecer la fecha

Escriba un programa para mostrar la fecha actual y pedir al usuario una nueva fecha. Si el usuario escribe una fecha que no está en blanco, úsela para actualizar la fecha del sistema.

4. Conversión a mayúsculas

Escriba un programa que utilice las funciones de INT 21h para recibir como entrada letras minúsculas del teclado y convertirlas a mayúsculas. Muestre sólo las letras en mayúscula.

5. Fecha de la creación de un archivo

Escriba un procedimiento que muestre la fecha de la creación de un archivo, junto con su nombre. Debe recibir un apuntador al nombre del archivo en el registro DX. Escriba un programa de prueba para demostrar el procedimiento con varios nombres de archivos distintos, incluyendo los nombres de archivos extendidos. Si no puede encontrarse un archivo, muestre el mensaje de error apropiado.

6. Programa para buscar coincidencias de texto

Escriba un programa que abra un archivo de texto, el cual debe contener hasta 60K bytes, y que realice la búsqueda de una cadena sin sensibilidad a mayúsculas y minúsculas. La cadena y el nombre de archivo pueden ser una entrada del usuario. Muestre cada línea del archivo en el que aparezca la cadena y anteponga a cada línea un número. Revise el procedimiento **Str_find** de los ejercicios de programación de la sección 9.7. Su programa debe ejecutarse en modo de direccionamiento real.

7. Cifrado de archivos mediante el uso de XOR

Mejore el programa de cifrado de archivos de la sección 6.3.4, de la siguiente manera:

- Pida al usuario el nombre de un archivo de texto simple y un archivo de texto cifrado.
- Abra el archivo de texto simple en modo de entrada, y el archivo de texto cifrado en modo de salida.
- Permita que el usuario introduzca un solo código entero de cifrado (de 1 a 255).
- Lea el archivo de texto simple en un búfer, y aplique un OR exclusivo a cada byte con el código de cifrado.
- Escriba el búfer en el archivo de texto cifrado.

El único procedimiento que puede llamar de la biblioteca de vínculos del libro es **ReadInt**. Todas las demás operaciones de entrada/salida deben realizarse mediante el uso de INT 21h. El mismo código que escriba también puede usarse para descifrar el archivo de texto cifrado, con lo cual se producirá el archivo de texto simple original.

8. Procedimiento ContarPalabras

Escriba un programa para contar las palabras en un archivo de texto. Pida al usuario el nombre de un archivo y muestre la cuenta de palabras en la pantalla. El único procedimiento que puede llamar de la biblioteca de vínculos del libro es **WriteDec**. Todas las demás operaciones de entrada/salida deben realizarse mediante el uso de INT 21h.

FUNDAMENTOS DE LOS DISCOS

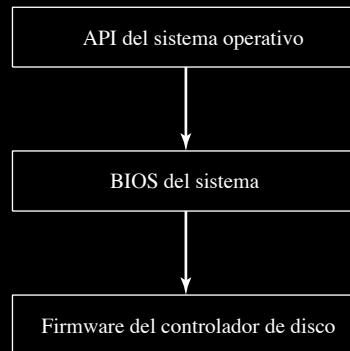
14.1 Sistemas de almacenamiento en disco	14.4 Lectura y escritura de sectores de disco (7305h)
14.1.1 Pistas, cilindros y sectores	14.4.1 Programa para visualización de sectores
14.1.2 Particiones de disco (volúmenes)	14.4.2 Repaso de sección
14.1.3 Repaso de sección	
14.2 Sistemas de archivos	14.5 Funciones de archivo a nivel de sistema
14.2.1 FAT12	14.5.1 Obtener el espacio libre del disco (7303h)
14.2.2 FAT16	14.5.2 Crear subdirectorio (39h)
14.2.3 FAT32	14.5.3 Eliminar subdirectorio (3Ah)
14.2.4 NTFS	14.5.4 Establecer el directorio actual (3Bh)
14.2.5 Áreas principales del disco	14.5.5 Obtener el directorio actual (47h)
14.2.6 Repaso de sección	14.5.6 Obtener y establecer atributos de archivo (7143h)
14.3 Directorio de disco	14.5.7 Repaso de sección
14.3.1 Estructura de directorios de MS-DOS	14.6 Resumen del capítulo
14.3.2 Nombres de archivos extensos en MS Windows	14.7 Ejercicios de programación
14.3.3 Tabla de asignación de archivos (FAT)	
14.3.4 Repaso de sección	

14.1 Sistemas de almacenamiento en disco

En este capítulo presentaremos los fundamentos de los sistemas de almacenamiento en disco. También mostraremos cómo se relaciona el almacenamiento en disco con el almacenamiento en disco a nivel de BIOS en las computadoras basadas en Intel. Por último, le mostraremos cómo interactúa MS Windows con los programas de aplicación para proporcionar el acceso a los archivos y directorios. En la sección 2.5 se mencionó por primera vez el BIOS del sistema. La interacción entre los niveles virtuales de una computadora se vuelve aparente si se considera el almacenamiento en disco (figura 14-1):

- En el nivel más 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 lecturas de sectores, escrituras 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.

FIGURA 14-1 Niveles virtuales de acceso al disco.



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 platos, lados, pistas, cilindros y sectores. En el nivel del BIOS del sistema, el almacenamiento de disco se describe en términos de clústeres y sectores. En el nivel del OS, el almacenamiento en disco se describe en términos de directorios y archivos.

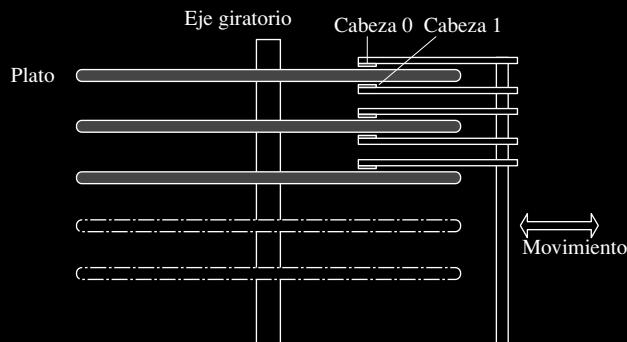
Programas en lenguaje ensamblador Los programas a nivel de usuario escritos en lenguaje ensamblador pueden acceder de manera directa al BIOS del sistema en MS-DOS, Windows 95, 98 y Millenium. Por ejemplo, tal vez tenga que almacenar y obtener los datos almacenados en un formato no convencional, recuperar datos perdidos o realizar diagnósticos en el hardware del disco. En este capítulo le 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 la manipulación de unidades y directorios.

Si utiliza Windows NT, 2000 o XP, los programas a nivel de usuario sólo pueden acceder al sistema de discos usando la API Win32. Esta 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.

14.1.1 Pistas, cilindros y sectores

Una unidad de disco ordinaria, como la que se muestra en la figura 14-2, está compuesta de 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.

FIGURA 14-2 Elementos físicos de un disco duro.

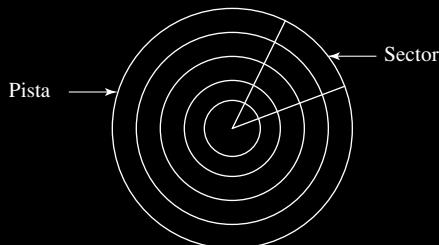


A la superficie de un disco se le da formato en forma de bandas concéntricas invisibles llamadas *pistas*, en las que los datos se almacenan de manera magnética. Una unidad de disco duro típica de 3.5" puede contener miles de pistas. Al proceso de desplazar las cabezas de lectura/escritura de una pista a otra se le conoce como *búsqueda*. El *tiempo promedio de búsqueda* es un tipo de medida de la velocidad de un disco. Otra medida es RPM (revoluciones 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 avanzamos 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 porción de 512 bytes de una pista, como se muestra en la figura 14-3. 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.

FIGURA 14-3 Pistas y sectores de disco.



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 superficie.
- 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 cuyos 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 pueda 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 *números de sectores lógicos*. Estos números de sectores lógicos siempre se enumeran en forma secuencial, empezando en cero.

14.1.2 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 de varios sistemas de archivos. Una unidad puede tener 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 asigna a una letra de unidad (C, D, E, etcétera). Las particiones lógicas no pueden tener capacidad de inicio. Es posible dar formato a cada partición lógica o del sistema con un sistema de archivos distintos.

Por ejemplo, suponga que a una unidad de disco duro de 20GB se le asignó 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 esta última partición en dos particiones lógicas de 2GB y 8GB, para después darles formato con varios sistemas de archivos como FAT16, FAT32 o NTFS (en la siguiente sección de este capítulo hablaremos sobre los detalles de estos sistemas de archivos). 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 multiinicio 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 participación 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 comparten datos almacenados en la misma partición lógica. Por ejemplo, todas las versiones 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.

Herramientas: puede usar el programa FDISK.EXE en MS-DOS y Windows 98 para crear y eliminar particiones, pero no preserva los datos. Mejor aún, Windows 2000 y XP tienen una herramienta llamada Administrador de discos, que cuenta con la capacidad de crear, eliminar y cambiar particiones de tamaño sin destruir los datos. También hay programas para particionar elaborados por terceros, como *PartitionMagic* de Symantec, que permite cambiar de tamaño y mover particiones sin destruir los datos.

Ejemplo de inicio dual En la figura 14-4, la herramienta *Administración de discos* de Windows 2000 muestra las seis particiones en una sola unidad de disco. La figura que aparece es para un sistema que arranca tanto en Windows 98 como en Windows 2000. Hay dos particiones principales, cuyos nombres arbitrarios son SYSTEM 98 y WIN2000-A. Sólo puede haber una partición activa en un momento dado. Cuando está activa se le llama *partición del sistema*.

En la misma figura, la partición del sistema es actualmente WIN2000-A, asignada a la unidad C. Observe que la partición del sistema inactiva no tiene letra de unidad. Si reiniciáramos la computadora e hicieráramos que arranca desde SYSTEM 98, esta partición se convertiría en la unidad C y la partición WIN2000-A estaría inactiva.

Mientras tanto, la partición extendida se ha dividido en cuatro particiones lógicas, dos de las cuales no tienen formato, mientras que a las otras dos, llamadas BACKUP y DATA_1, se les dio formato con el sistema de archivos FAT32.

FIGURA 14-4 Herramienta Administración de discos de Windows 2000.

Volume	Layout	Type	File System	Status	Capacity	Free Space	% Free
	Partition	Basic		Healthy	5.13 GB	5.13 GB	100 %
	Partition	Basic		Healthy	2.01 GB	2.01 GB	100 %
BACKUP (E:)	Partition	Basic	FAT32	Healthy	7.80 GB	4.84 GB	62 %
DATA_1 (D:)	Partition	Basic	FAT32	Healthy	7.80 GB	2.66 GB	34 %
SYSTEM 98	Partition	Basic	FAT32	Healthy	1.95 GB	1.12 GB	57 %
WIN2000-A (C:)	Partition	Basic	NTFS	Healthy (System)	3.91 GB	1.43 GB	36 %

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.

14.1.3 Repaso de sección

1. (*Verdadero/Falso*): una pista se divide en varias unidades llamadas *sectores*.
2. (*Verdadero/Falso*): un sector consiste en varias pistas.
3. Un(a) _____ consiste en todas las pistas accesibles desde una sola posición de las cabezas de lectura/escritura de una unidad de disco duro.
4. (*Verdadero/Falso*): los sectores físicos siempre son de 512 bytes, ya que el fabricante los marca en el disco.
5. En FAT32, ¿cuántos bytes utiliza un sector lógico?
6. ¿Por qué los archivos se guardan en un principio en cilindros adyacentes?
7. Cuando el espacio de almacenamiento de un archivo se fragmenta, ¿qué significa esto en términos de cilindros y las operaciones de *búsqueda* que realiza la unidad?
8. Otro nombre para una unidad de disco es unidad _____.
9. ¿Qué es lo que mide el *tiempo promedio de búsqueda* de una unidad?
10. ¿Qué es un *formato de bajo nivel*?
11. ¿Qué contiene el *registro de inicio maestro*?
12. ¿Cuántas particiones primarias puede haber activas al mismo tiempo?
13. Cuando una partición primaria está activa, se le conoce como la partición _____.

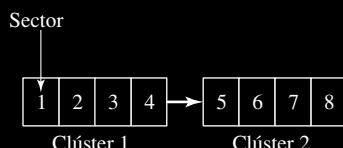
14.2 Sistemas de archivos

Todos los sistemas operativos tienen cierto tipo de sistema 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 aún 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 del tamaño de su partición de disco. La figura 14-5 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. La sección 14.3.3 explica la FAT con mayor detalle.

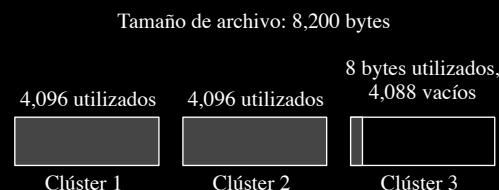
FIGURA 14-5 Ejemplo de cadenas de clústeres.



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 figura 14-6 muestra un archivo de 8200 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. Imagine lo que resultaría si nuestro archivo de 8200 bytes se almacenara en un volumen que tuviera clústeres de 32KB. En ese caso, se desperdiciarían

24568 bytes ($32768 - 8200$). En los volúmenes que tienen una gran cantidad de archivos pequeños, es mejor usar tamaños pequeños para los clústeres.

FIGURA 14-6 Cadena de clústeres que muestra el desperdicio de espacio.



Ejemplo en Windows 2000/XP En la tabla 14-1 se muestran los tamaños de clúster estándar y los tipos de sistemas de archivos, para las 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.

Tabla 14-1 Tamaños de particiones y clústeres (Más de 1GB).

Tamaño del volumen	Clúster de FAT16	Clúster de FAT32	Clúster de NTFS ^a
1.25GB-2GB	32KB	4KB	2KB
2GB-4GB	64KB ^b	4KB	4KB
4GB-8GB	ns (<i>no se soporta</i>)	4KB	4KB
8GB-16GB	ns	8KB	4KB
16GB-32GB	ns	16KB	4KB
32GB-2TB	ns	ns ^c	4KB

^a Tamaños predeterminados en NTFS. Pueden modificarse a la hora de dar formato al disco.

^b Los clústeres de 64KB con FAT16 sólo se soportan en Windows 2000 y XP.

^c Hay un parche de software disponible, que permite a Windows 98 dar formato a unidades mayores de 32GB.

14.2.1 FAT12

El sistema de archivos FAT12 se utilizó por primera vez en los discos duros 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.

14.2.2 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 inficiente en los volúmenes de más de 1GB, ya que FAT16 utiliza tamaños de clúster 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 y 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.

14.2.3 FAT32

El sistema de archivos FAT32 se introdujo con la versión OEM2 de Windows 95, y 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 clúster más pequeño 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.

14.2.4 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 esparcirse a través de varios discos duros.
- El tamaño de clúster predeterminado es de 4KB, para los discos de más de 2GB.
- Soporta los nombres de archivo 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 posibles (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 varias unidades.

La tabla 14-2 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.

Tabla 14-2 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

14.2.5 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 formato al volumen. Por ejemplo, la asignación de sectores en un disquete de 3.5 pulgadas y 1.44MB se muestra en la tabla 14-3.

Tabla 14-3 Asignación de sectores en un disquete de 1.44MB.

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 tabla 14-4 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.

Tabla de asignación de archivos (FAT) La tabla de asignación de archivos es bastante compleja, por lo que hablaremos con más detalle sobre ella en la sección 14.3.3.

Tabla 14-4 Distribución del registro de inicio de MS-DOS.

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 32 MB
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 de 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 de 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 pueden ser otros nombres de directorios o referencias a archivos. Una entrada de directorio que hace referencia a un archivo contiene su nombre, tamaño, atributos y número de clúster de inicio que utiliza el archivo.

Área de datos El *área de datos* del disco es en donde se almacenan los archivos y subdirectorios.

14.2.6 Repaso de sección

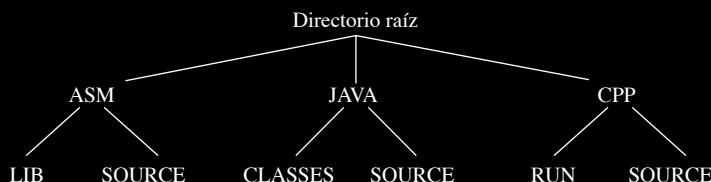
- (Verdadero/Falso): un sistema de archivos asigna sectores lógicos a los clústeres.
- (Verdadero/Falso): el número del clúster inicial de un archivo se almacena en la *tabla de parámetros de disco*.
- (Verdadero/Falso): todos los sistemas excepto NTFS requieren el uso de por lo menos un clúster para almacenar un archivo.
- (Verdadero/Falso): el sistema de archivos FAT32 permite establecer permisos individuales de usuario para los directorios, pero no para los archivos.
- (Verdadero/Falso): Linux no soporta el sistema de archivos FAT32.
- En Windows 98, ¿cuál es el volumen FAT16 más grande que se permite?
- Suponga que el registro de inicio de su volumen de disco está corrupto. ¿Qué sistema(s) de archivos proporcionarían soporte para una copia de respaldo del registro de inicio?

8. ¿Qué sistema(s) de archivos de MS Windows soportan los nombres de archivos Unicode de 16 bits?
9. ¿Qué sistema(s) de archivos de MS Windows soportan los *volumenes reflejados*, en donde los mismos datos se escriben en forma simultánea en varias unidades?
10. Suponga que necesita mantener un registro de las últimas diez modificaciones realizadas en un archivo. ¿Qué sistema(s) de archivos soporta(n) esta característica?
11. Si tiene un volumen de disco de 20GB y desea tener un tamaño de clúster $\leq 8KB$ (para evitar desperdiciar el espacio), ¿qué sistema(s) de archivos podría utilizar?
12. ¿Cuál es el volumen de disco FAT32 más grande que soporta clústeres de 4KB?
13. Describa las cuatro áreas (en orden) de un disquete de 1.44MB.
14. En una unidad de disco con formato de MS-DOS, ¿cómo podría determinar el número de sectores que utiliza cada clúster?
15. *Reto:* si un disco tiene un tamaño de clúster de 8KB, ¿cuántos bytes de espacio desperdiciado habrá cuando se almacene un archivo de 8200 bytes?
16. *Reto:* explique cómo NTFS almacena los archivos dispersos. Para responder a esta pregunta, tendrá que visitar el sitio Web de Microsoft MSDN y buscar la información.

14.3 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 el *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 directorios en los niveles inferiores (figura 14-7).

FIGURA 14-7 Ejemplo de un árbol de directorios del 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 la *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 le puede anteponer una especificación de unidad.

14.3.1 Estructura de directorios de MS-DOS

Si tratáramos 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 archivo extendidos disponible en MS Windows.

Cada entrada de directorio de MS-DOS es de 32 bytes y contiene los campos que se muestran en la tabla 14-5. 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 tabla 14-6 se muestran los posibles valores de estado. 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.

Tabla 14-5 Entrada de directorio de MS-DOS.

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

Tabla 14-6 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
4nh	Primera entrada de nombre de archivo extenso (contiene la primera parte del nombre): si el byte de atributo = 0Fh, esto marca la última de varias entradas que contienen un solo nombre de archivo extenso. El dígito n indica el número de entradas que utiliza el nombre de archivo

Campo atributo

El campo *atributo* identifica el tipo del archivo. El campo está asignado por bits y, por lo general, contiene una combinación de uno de los valores que se muestran en la figura 14-8. 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 contiene el nombre de un subdirectorio. La *etiqueta de volumen* identifica 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 del directorio. El bit *sólo lectura* evita que el archivo se elimine o se modifique de cualquier forma. Por último, un valor de atributo de 0Fh indica que la entrada actual del directorio es para un nombre de archivo extendido.

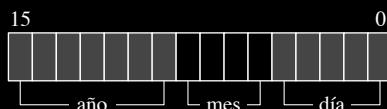
FIGURA 14–8 Campos de bytes de atributos de un archivo.



Fecha y hora

El campo *etiqueta de fecha* (figura 14–9) indica la fecha en que se creó 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 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.

FIGURA 14–9 Campo de etiqueta de fecha de un archivo.



El campo *etiqueta de hora* (figura 14–10) 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 en la figura 14–11 indica las 14:02:40 horas.

FIGURA 14–10 Campo de etiqueta de hora de un archivo.

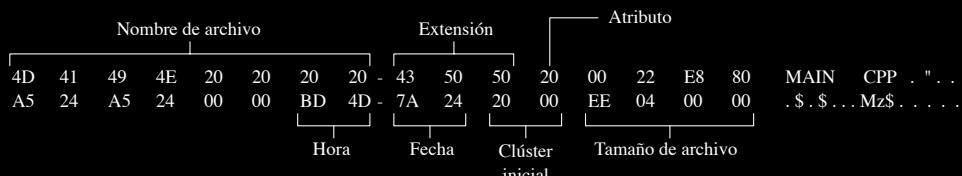


FIGURA 14–11 Ejemplo de etiqueta de hora.



Ejemplo de entrada de directorio de un archivo Examinemos la entrada para un archivo llamado MAIN.CPP (figura 14-12). Este archivo tiene un atributo normal, y su bit de archivo (20h) está activo, lo cual muestra que se ha modificado. Su número de clúster inicial es 0020h, su tamaño es de 0000004EEh bytes, el campo *Hora* es igual a 4DBDh (9:45:58) y el campo *Fecha* es igual a 247Ah (Marzo 26, 1998).

FIGURA 14-12 Ejemplo de entrada de directorio de un archivo.



En esta figura, la hora, fecha y número de clúster inicial son valores de 16 bits, almacenados en orden little 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.

14.3.2 Nombres de archivos extensos en MS Windows

En MS Windows, a un nombre de archivo mayor que 8 + 3 caracteres, o a un nombre de archivo 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. El dígito inferior indica el número de entradas de directorio que va a utilizar el nombre de archivo extenso. Las entradas subsiguientes cuentan en orden descendente desde $n - 1$ hasta 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 serán los bytes de estado iguales a 02h y 01h, como puede verse en la siguiente tabla:

Byte 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

Ejemplo Para ilustrar esto, vamos a utilizar un archivo que tiene el nombre de archivo de 26 caracteres ABCDEFGHIJKLMNOPQRSTUVWXYZ.TXT y guardarlo como archivo de texto en el directorio raíz de la unidad A. A continuación, vamos a ejecutar DEBUG.EXE del símbolo del sistema y cargar los sectores del directorio en la memoria, en el desplazamiento 100. Esto va seguido de una D (comando de vaciado)¹:

L 100 0 13 5 (carga los sectores 13h - 17h)
D 100 (vacía el desplazamiento 100 en la pantalla)

Windows crea tres entradas de directorio para este archivo, como se muestra en la figura 14-13.

Vamos a empezar con la entrada en 01C0h. El primer byte, que contiene 01, marca esta entrada como la última de una secuencia de entradas de nombre de archivo extenso. Va seguida de los primeros 13 caracteres del nombre “ABCDEFGHIJKLM”. Cada carácter Unicode es de 16 bits, y se almacena en orden little endian. Observe que el byte de atributo en el desplazamiento 0B es igual a 0F, lo cual indica que es una entrada de nombre de archivo extendido (MS-DOS ignora de manera automática cualquier nombre de archivo que tenga este atributo).

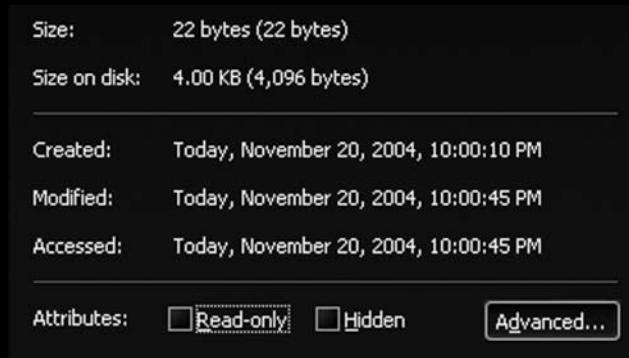
La entrada en 01A0h contiene los últimos 13 caracteres del nombre de archivo extenso, que son "NOPQRSTUVWXYZ.TXT".

FIGURA 14-13 Entrada de directorio para un nombre de archivo extenso.

Primera entrada extensa																	
Última entrada extensa																	
Atributo (entrada extensa)																	
01A0	42	4E	00	4F	00	50	00	51	00	52	00	0F	00	27	53	00	BN.O.P.Q.R...S.
01B0	54	00	55	00	56	00	2E	00	54	00	00	00	58	00	54	00	T.U.V...T...X.T.
01C0	01	41	00	42	00	43	00	44	00	45	00	0F	00	27	46	00	.A.B.C.D.E...F.
01D0	47	00	48	00	49	00	4A	00	4B	00	00	00	4C	00	4D	00	G.H.I.J.K...L.M.
01E0	41	42	43	44	45	46	7E	31	54	58	54	20	00	AF	78	62	ABCDEF~1TXT ..xb
01F0	2F	2B	30	2B	00	00	59	B9	30	2B	02	00	52	01	00	00	/+0+..Y.0+..R...
Fecha de creación			Fecha de último acceso			Fecha de última modificación			Tamaño del archivo		Hora de creación						
			Hora de última modificación			Primer clúster											

En el desplazamiento 01E0h, el nombre de archivo corto generado de manera automática se crea a partir de las primeras seis letras del nombre de archivo extenso, seguido de ~1, y seguido de los primeros tres caracteres después del último punto en el nombre original. Estos caracteres son códigos ASCII de 1 byte. La entrada de nombre de archivo corto también contiene la fecha y hora de creación, la fecha del último acceso, la fecha y hora de la última modificación, el número de clúster inicial, y el tamaño del archivo. La figura 14-4 muestra la información que aparece en el cuadro de diálogo *Propiedades* del Explorador de Windows, la cual coincide con los datos del directorio sin formato.

FIGURA 14-14 Cuadro de diálogo Propiedades de archivo.



14.3.3 Tabla de asignación de archivos (FAT)

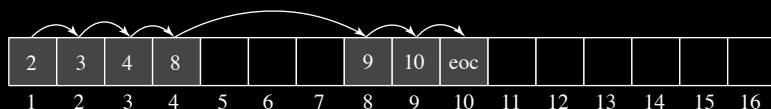
Los sistemas de archivos FAT12, FAT16 y FAT32 utilizan una tabla conocida como la *tabla de asignación de archivos* (FAT) para llevar el registro de la ubicación de cada archivo en el disco. La FAT asigna los clústeres de disco, mostrando a qué archivo específico pertenecen. Cada entrada corresponde a un número de clúster, y cada clúster contiene uno o más sectores. En otras palabras, la 10^a entrada en la FAT indica el 10^{vo} clúster en el disco, la 11^a entrada identifica el 11^{vo} clúster, y así sucesivamente.

Cada archivo se representa en la FAT como una lista enlazada, llamada *cadena de clústeres*. Cada entrada en la FAT contiene un entero que identifica a la siguiente entrada. En la figura 14-15 se muestran dos cadenas de clústeres, una para **Archivo1** y la otra para **Archivo2**. **Archivo1** ocupa los clústeres 1, 2, 3, 4, 8, 9 y 10. **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.

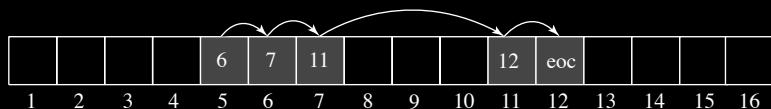
Cuando se crea un archivo, el sistema operativo busca la primera entrada de clúster disponible en la FAT. Se producen huecos cuando no hay suficientes clústeres contiguos para contener el archivo completo. En el diagrama anterior, esto le ocurrió tanto a **Archivo1** como a **Archivo2**. Cuando se modifica 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 las distintas pistas para localizar todos los clústeres de un archivo. La mayoría de los sistemas operativos cuentan con una herramienta de desfragmentación de discos integrada.

FIGURA 14-15 Ejemplo: dos cadenas de clústeres.

Archivo1: número de clúster inicial = 1, tamaño = 7 clústeres



Archivo2: número de clúster inicial = 5, tamaño = 5 clústeres



14.3.4 Repaso de sección

- (Verdadero/Falso): una especificación de archivo incluye una ruta de directorio y el nombre del archivo.
- (Verdadero/Falso): la lista principal de archivos en un disco se llama *directorio base*.
- (Verdadero/Falso): una entrada de directorio de un archivo contiene el número de sector inicial de ese archivo.
- (Verdadero/Falso): el campo de fecha de MS-DOS en una entrada de directorio debe sumarse al 1980.
- ¿Cuántos bytes utiliza una entrada de directorio de MS-DOS?
- Mencione los siete campos básicos de una entrada de directorio de MS-DOS (no incluya el campo *reservado*).
- En una entrada de nombre de archivo de MS-DOS, identifique los seis valores posibles del byte de estado.
- Muestre el formato del campo de etiqueta de hora en una entrada de directorio de MS-DOS.
- Cuando se almacena un nombre de archivo extenso en un directorio de volumen (en MS Windows), ¿cómo se identifica la primera entrada del nombre de archivo extenso?
- Si un nombre de archivo tiene 18 caracteres, ¿cuántas entradas de nombre de archivo extenso se requieren?
- MS Windows agregó dos nuevos campos de fecha a la entrada de directorio de un archivo de MS-DOS original. ¿Cuáles son sus nombres?
- Reto:* ilustre los enlaces de la tabla de asignación de archivos para un archivo que utiliza los clústeres 2, 3, 7, 6, 4, 8, en ese orden.

14.4 Lectura y escritura de sectores de disco (7305h)

La función 7305h de INT 21h (lectura y escritura absolutas 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 direcccionamiento real de 16 bits. No trataremos de llamar a INT 21h (ni a cualquier otra interrupción) desde el modo protegido, debido a las complejidades implicadas.

La función 7305h funciona en los sistemas de archivos 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 de disco podría ignorar con facilidad los permisos de compartición de archivos 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 bufer   ; segmento del búfer
ESDISCO ENDS
```

A continuación se muestran 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 escribir 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
```

14.4.1 Programa para visualización de sectores

Vamos a dar un buen uso a lo que hemos aprendido sobre los sectores, escribiendo un programa que lee y muestra los sectores de disco individuales en formato ASCII. A continuación se muestra el seudocódigo:

```

Pedir número de sector inicial y número de unidad
do while (tecleo <> ESC)
    Mostrar encabezado
    Leer un sector
    If error de MS-DOS then terminar
    Mostrar un sector
    Esperar tecleo
    Incrementar número de sector
end do

```

Listado del programa He aquí un listado completo del programa *Sector.asm* de 16 bits. Se ejecuta en modo de direccionamiento real en Windows 95, 98 y Me, pero no en Windows NT, 2000 o XP, debido a que tienen una seguridad más estricta sobre el acceso al disco:

```

TITLE Programa para visualización de sectores      (Sector.asm)
; Demuestra la función 7305h de INT 21h (ABSDiskReadWrite)
; Este programa en modo real lee y muestra sectores del disco.
; Trabaja en los sistemas de archivos FAT16 & FAT32 que se ejecutan
; en Windows 95, 98 y Millenium.

INCLUDE Irvine16.inc

EstablecerCursor PROTO, fila:BYTE, col:BYTE
FIN_LINEA EQU <0dh,0ah>
TECLA_ESC = 1Bh
FILA_DATOS = 5
COL_DATOS = 0
TAM_SECTOR = 512
MODO_LECTURA = 0                                ; para la función 7505h

ESDiscos STRUCT
    sectorInicial DWORD ?                      ; número del sector inicial
    numSectores WORD 1                          ; número de sectores
    despBufer WORD OFFSET bufer                ; desplazamiento del búfer
    segBufer WORD SEG bufer                   ; segmento del búfer
ESDiscos ENDS

.data
numeroUnidad BYTE ?
estrucDisco ESDiscos <>
bufer BYTE TAM_SECTOR DUP(0),0                 ; un sector

fila_actual     BYTE  ?
col_actual      BYTE  ?

; Recursos de cadena
lineaCad        BYTE FIN_LINEA,79 DUP(0C4h),FIN_LINEA,0
encabezadoCad   BYTE "Programa de visualización de sectores (Sector.exe)"
                  BYTE FIN_LINEA,FIN_LINEA,0
pedirSectorCad  BYTE "Escriba el numero de sector inicial: ",0
pedirUnidadCad  BYTE "Escriba el numero de la unidad (1=A, 2=B, "
                  BYTE "3=C, 4=D, 5=E, 6=F): ",0
noSePuedeLeerCad BYTE FIN_LINEA,"*** No se puede leer el sector. "
                  BYTE "Oprima cualquier tecla...", FIN_LINEA, 0
leyendoSectorCad \
                  BYTE "Oprima Esc para terminar, o cualquier tecla para continuar..."
                  BYTE FIN_LINEA,FIN_LINEA,"Leyendo sector: ",0

.code
main PROC
    mov    vax,@data
    mov    ds,ax

```

```

call Clrscr
mov dx,OFFSET encabezadoCad      ; muestra bienvenida
call Writestring                 ; pide al usuario...
call PedirNumeroSector

L1: call Clrscr
    call LeerSector               ; lee un sector
    jc L2                         ; termina si hay error
    call MostrarSector
    call ReadChar
    cmp al,TECLA_ESC              ; ¿se oprimió Esc?
    je L3                          ; sí: termima
    inc estrucDisco.sectorInicial ; siguiente sector
    jmp L1                          ; repite el ciclo

L2: mov dx,OFFSET noSePuedeLeerCad ; mensaje de error
    call Writestring
    call ReadChar

L3: call Clrscr
    exit
main ENDP
;-----
PedirNumeroSector PROC
;
; Pide al usuario el número de sector inicial
; y el número de la unidad. Inicializa el campo sectorInicial
; de la estructura ESDisco, así como de la variable
; numeroUnidad.
;-----
pusha
mov dx,OFFSET pedirSectorCad
call WriteString
call ReadInt
mov estrucDisco.sectorInicial,eax
call Crlf
mov dx,OFFSET pedirUnidadCad
call WriteString
call ReadInt
mov numeroUnidad,al
call Crlf
popa
ret
PedirNumeroSector ENDP
;-----
LeerSector PROC
;
; Lee un sector y lo coloca en el búfer de entrada.
; Recibe: DL = Número de la unidad
; Requiere: Debe inicializarse la estructura ESDisco.
; Devuelve: Si CF=0, la operación tuvo éxito;
;           en caso contrario, CF=1 y AX contiene un
;           código de error.
;-----
pusha
mov ax,7305h                      ; ABSDiskReadWrite
mov cx,-1                           ; siempre es -1
mov bx,OFFSET estrucDisco          ; número de sector

```

```

        mov    si,MODO_LECTURA          ; modo de lectura
        int    21h                      ; lee el sector del disco
        popa
        ret
LeerSector ENDP

;-----
MostrarSector PROC
;
; Muestra los datos del sector en <bufer>, usando las llamadas
; a la función INT 10h del BIOS. Esto evita filtrar los códigos
; ASCII de control.
; Recibe: nada. Devuelve: nada.
; Requiere: el búfer debe contener los datos del sector.
;-----
        mov    dx,OFFSET encabezadoCad   ; muestra el encabezado
        call   WriteString
        mov    eax,estrucDisco.sectorInicial ; muestra el número del sector
        call   WriteDec
        mov    dx,OFFSET lineaCad        ; línea horizontal
        call   Writestring
        mov    si,OFFSET bufer           ; apunta al búfer
        mov    fila_actual,FILA_DATOS   ; establece fila, columna
        mov    col_actual,COL_DATOS
        INVOKE EstablecerCursor,fila_actual,col_actual

        mov    cx,TAM_SECTOR           ; contador del ciclo
        mov    bh,0                     ; página de video 0
L1:   push  cx                     ; guarda el contador del ciclo
        mov    ah,0Ah
        mov    al,[si]                 ; muestra el carácter
        mov    cx,1                     ; obtiene el byte del búfer
        int    10h
        call   MoverCursor
        inc    si                     ; apunta al siguiente byte
        pop    cx                     ; restaura el contador del ciclo
        loop  L1                     ; repite el ciclo
        ret
MostrarSector ENDP

;-----
MoverCursor PROC
;
; Avanza el cursor a la siguiente columna, comprueba la posibilidad
; de que el texto pase a la siguiente línea en la pantalla.
;-----
        cmp    col_actual,79           ; ¿última columna?
        jae    L1                     ; sí: avanza a la siguiente fila
        inc    col_actual             ; no: incrementa la columna
        jmp    L2
L1:   mov    col_actual,0           ; siguiente fila
        inc    fila_actual
L2:   INVOKE EstablecerCursor,fila_actual,col_actual
        ret
MoverCursor ENDP

;-----
EstablecerCursor PROC USES dx,

```

```

fila:BYTE, col:BYTE
;
; Establece la posición del cursor de la pantalla
;-----
    mov    dh, fila
    mov    dl, col
    call   Gotoxy
    ret
EstablecerCursor ENDP
END main

```

El núcleo del programa es el procedimiento **LeerSector**, el cual lee cada sector del disco, usando la función 7305h de INT 21h. Los datos del sector se colocan en un búfer, y éste se muestra mediante el procedimiento **MostrarSector**.

Uso de INT 10h La mayoría de los sectores contienen datos binarios, y si se utilizara INT 21h para mostrarlos, se filtrarían los caracteres ASCII de control. Por ejemplo, los caracteres de Tabulación y Nueva línea harían que la pantalla se separara. En vez de ello, es mejor utilizar la función 9 de INT 10h, la cual muestra los códigos ASCII del 0 al 31 como caracteres gráficos. En la sección 15.4 se describe la función INT 10h. Como la función 9 no hace que avance el cursor, debe escribirse el código adicional para desplazar el cursor una columna a la derecha después de mostrar cada carácter. El procedimiento **EstablecerCursor** simplifica la implementación del procedimiento **Gotoxy** en la biblioteca Irvine16.

Variaciones Pueden crearse variaciones interesantes en el programa de Visualización de sectores. Por ejemplo, podemos pedir al usuario un rango de números de sectores a visualizar. Cada sector puede mostrarse en hexadecimal. Podemos dejar que el usuario se desplace hacia delante y hacia atrás por los sectores, usando las teclas AvPág y RePág. Algunas de estas mejoras aparecen en los ejercicios del capítulo.

14.4.2 Repaso de sección

1. (*Verdadero/Falso*): se pueden leer sectores de un disco duro usando la función 7305h de INT 21h en Windows Me, pero no en Windows XP.
2. (*Verdadero/Falso*): la función 7305h de INT 21h lee uno o más sectores de disco sólo en modo protegido.
3. ¿Qué parámetros de entrada requiere la función 7305h de INT 21h?
4. En el Programa para visualización de sectores (sección 14.4.1), ¿por qué se utiliza la interrupción 10h para mostrar caracteres?
5. *Reto*: en el Programa para visualización de sectores (sección 14.4.1), ¿qué pasaría si el número de sector inicial estuviera fuera de rango?

14.5 Funciones de archivo a nivel de sistema

En el modo de direccionamiento real, INT 21h proporciona los servicios del sistema (tabla 14-7) para crear y cambiar 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. En el apéndice C encontrará una lista más detallada de las interrupciones de MS-DOS y sus descripciones.

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 características como los nombres de archivo extensos 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 originalmente en MS-DOS.

Tabla 14-7 Servicios de disco de INT 21h selectos.

Número de función	Nombre de función
0Eh	Establecer 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 archivo
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
56hh	Cambiar el nombre del archivo
57h	Obtener/establecer la fecha y hora del archivo
59h	Obtener la información de error extendida

14.5.1 Obtener el espacio libre del disco (7303h)

La función 7303h de INT 21h 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:

```
ExtGetDskFreSpcStruc STRUC
    StructSize      WORD  ?
    Level          WORD  ?
    SectorsPerCluster  DWORD ?
    BytesPerSector   DWORD ?
    AvailableClusters  DWORD ?
    TotalClusters     DWORD ?
    AvailablePhysSectors  DWORD ?
    TotalPhysSectors   DWORD ?
    AvailableAllocationUnits  DWORD ?
    TotalAllocationUnits  DWORD ?
    Rsvd           DWORD 2 DUP (?)
```

```
ExtGetDskFreSpcStruc ENDS
```

En el archivo *Irvine16.inc* encontrará una copia de esta estructura. La siguiente lista contiene una breve descripción de cada campo:

- **StructSize:** un valor de retorno que representa el tamaño de la estructura **ExtGetDskFreSpcStruc** en bytes. Cuando se ejecuta la función 7303h de INT 21h (Get_ExtFreeSpace), coloca el tamaño de la estructura en este miembro.
- **Level:** un valor de nivel de entrada y retorno. Este campo debe inicializarse con cero.
- **SectorsPerCluster:** el número de sectores dentro de cada clúster.
- **BytesPerSector:** el número de bytes en cada sector.
- **AvailableClusters:** el número de clústeres disponibles.
- **TotalClusters:** el número total de clústeres en el volumen.
- **AvailablePhysSectors:** 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.
- **AvailableAllocationUnits:** el número de unidades de asignación disponibles en el volumen, sin ajuste para compresión.

- **TotalAllocationUnits:** el número total de unidades de asignación en el volumen, sin ajuste para compresión.
- **Rsvd:** 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 **ExtGetDskFreSpcStruc**.
- CX debe contener el tamaño de la variable **ExtGetDskFreSpcStruc**.
- 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 bandera Acarreo. Después de llamar a la función, los siguientes tipos de cálculos podrían ser útiles:

- Para averiguar qué tan grande es el volumen en kilobytes, use la fórmula (TotalClusters * SectorsPerCluster * BytesPerSector)/1024.
- Para averiguar cuánto espacio libre hay en el volumen, en kilobytes, la fórmula es (AvailableClusters * SectorsPerCluster * BytesPerSector)/1024.

Programa de espacio libre en el disco

El siguiente programa utiliza la función 7303h de INT 21h para obtener la información de espacio libre en un volumen de disco tipo FAT. Muestra tanto el tamaño del volumen como el espacio libre. Se ejecuta en Windows 95, 98 y Millenium, pero no en Windows NT, 2000 o XP:

```
TITLE Espacio libre en el disco                                (EspacioDisco.asm)
INCLUDE Irvine16.inc

.data
bufer ExtGetDskFreSpcStruc <>
nombreUnidad BYTE "C:\",0
cad1 BYTE "Tamaño del volumen (KB): ",0
cad2 BYTE "Espacio libre (KB): ",0
cad3 BYTE "Falló la llamada a la función.",0dh,0ah,0

.code
main PROC
    mov ax,@data
    mov ds,ax
    mov es,ax

    mov bufer.Level,0           ; debe ser cero
    mov di,OFFSET bufer        ; ES:DI apunta al búfer
    mov cx,SIZEOF bufer        ; tamaño del búfer
    mov dx,OFFSET DriveName   ; apuntador al nombre de la unidad
    mov ax,7303h                ; obtiene el espacio libre en el disco
    int 21h
    jc error                   ; falló si CF = 1

    mov dx,OFFSET cad1         ; tamaño del volumen
    call WriteString
    call CalcTamVolumen
    call WriteDec
    call Crlf

    mov dx,OFFSET cad2         ; espacio libre
    call WriteString
```

```

    call CalcLibreVolumen
    call WriteDec
    call Crlf
    jmp quit
error:
    mov dx,OFFSET cad3
    call WriteString
quit:
    exit
main ENDP

;-----
CalcTamVolumen PROC
;
; Calcula y devuelve el tamaño del volumen de disco, en kilobytes.
; Recibe:     variable búfer, una estructura ExtGetDskFreSpcStruc
; Devuelve:   EAX = tamaño del volumen
; Comentarios: (SectorsPerCluster * 512 * TotalClusters) / 1024
;
        mov eax,bufer.SectorsPerCluster
        shl eax,9                                ; mult por 512
        mul bufer.TotalClusters
        mov ebx,1024
        div ebx                                     ; devuelve kilobytes
        ret
CalcTamVolumen ENDP

;-----
CalcLibreVolumen PROC
;
; Calcula y devuelve el número de kilobytes disponibles
; en el volumen dado.
; Recibe:     variable búfer, una estructura ExtGetDskFreSpcStruc
; Devuelve:   EAX = espacio disponible, en kilobytes
; Comentarios: (SectorsPerCluster * 512 * AvailableClusters) / 1024
;
        mov eax,bufer.SectorsPerCluster
        shl eax,9                                ; mult por 512
        mul bufer.AvailableClusters
        mov ebx,1024
        div ebx                                     ; devuelve kilobytes
        ret
CalcLibreVolumen ENDP
END main

```

14.5.2 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 retorno de error posibles son 3 y 5. El error 3 (*no se encontró la ruta*) indica que una parte del nombre de la ruta no existe. Suponga que pedimos a MS-DOS que cree el directorio ASMPROGNUEVO, pero la ruta ASMPROG no existe. Esto generaría un error 3. El error 5 (*acceso denegado*) indica que el subdirectorio propuesto ya existe o que el primer directorio en la ruta es el directorio raíz y ya está lleno.

14.5.3 Eliminar subdirectorio (3Ah)

La función 3Ah de INT 21h elimina un directorio. Recibe un apuntador a la unidad deseada y la ruta en DS:DX. Si se omite el nombre de la unidad, se asume que se va a utilizar la unidad predeterminada. El siguiente código elimina el directorio \ASM de la unidad C:

```
.data
nombreruta BYTE 'C:\ASM',0
.code
mov ah,3Ah           ; elimina el 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*).

14.5.4 Establecer el directorio actual (3Bh)

La función 3Bh de INT 21h 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:\ASMPROGS:

```
.data
nombreruta BYTE "C:\ASM\PROGS",0
.code
mov ah,3Bh           ; establece el directorio actual
mov dx,OFFSET nombreruta
int 21h
jc mostrar_error
```

14.5.5 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 raíz 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:\ASMPROGS, la cadena devuelta por MS-DOS es “ASMPROGS”:

```
.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
```

14.5.6 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 tabla 14-8. El atributo _A_NORMAL debe utilizarse solo, pero los demás atributos pueden combinarse mediante el operador +.

Tabla 14-8 Atributos de archivo (definidos en *Irvine16.inc*).

Valor	Significado
_A_NORMAL (0000h)	Se puede leer el archivo o escribir en él. Este valor es válido únicamente 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 establece los atributos de un archivo a sólo lectura y oculto:

```
mov ax,7143h
mov b1,1
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 atributos se devuelven en CX como una combinación de potencias de 2. Use la instrucción TEST para evaluar los atributos individuales. Por ejemplo,

```
text cx,_A_RDONLY
jnz archivoSoloLectura ; el archivo es de sólo lectura
```

El atributo _A_ARCH puede aparecer con cualquiera de los demás atributos.

14.5.7 Repaso de sección

1. ¿Qué función INT 21h utilizaría para obtener el tamaño de clúster de una unidad de disco?
2. ¿Qué función INT 21h utilizaría para averiguar cuántos clústeres libres hay en la unidad C?
3. ¿Qué funciones INT 21h llamaría si quisiera crear un directorio llamado D:\apps y convertirlo en el directorio actual?
4. ¿Qué función INT 21h llamaría si quisiera hacer un archivo de sólo lectura?

14.6 Resumen del capítulo

En el nivel del sistema operativo, no es útil conocer la geometría exacta de un disco (ubicaciones físicas) o la información específica de su marca. El BIOS, que en este caso equivale al firmware del controlador de disco, actúa como un agente entre el hardware del disco y el sistema operativo.

A la superficie de un disco se le da formato en forma de bandas concéntricas conocidas como *pistas*, en las que los datos se almacenan de manera magnética. El *tiempo promedio de búsqueda* mide el tiempo promedio que se invierte en pasar de una pista a otra. El rendimiento de un disco puede medirse en RPM (revoluciones por minuto), así como la *tasa de transferencia de datos* (cantidad de datos transferidos hacia y desde la unidad en 1 segundo).

Un *cilindro* se refiere a todas las pistas a las que se puede acceder desde una sola posición de las cabezas de lectura/escritura. Con el tiempo, a medida que los archivos se espacien más a lo largo de un disco, se fragmentan y ya no se almacenan en cilindros adyacentes.

Un *sector* es una porción de 512 bytes de una pista. El fabricante marca los sectores físicos en forma magnética (invisible) en el disco, usando lo que se conoce como formato de bajo nivel.

La *geometría física del disco* describe la estructura de un disco para que el BIOS del sistema pueda leerla. Una unidad física de disco duro individual puede dividirse en una o más unidades lógicas llamadas particiones, o volúmenes. Una unidad puede tener varias particiones. Una partición extendida puede subdividirse en un número ilimitado de particiones lógicas. Cada partición lógica aparece como una letra de unidad separada y puede tener un sistema de archivos distinto al de las demás particiones. Cada una de las particiones primarias puede contener un sistema operativo con capacidad de inicio.

El *registro maestro de inicio* (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 las 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 de inicio, que a su vez carga el sistema operativo.

Un sistema de archivos lleva el registro de la ubicación, el tamaño y los atributos de cada archivo en el disco. Proporciona un mapa en el que se asignan los sectores lógicos a los clústeres, la unidad básica de almacenamiento para todos los archivos y directorios, y un mapa de asignación de 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. Para hacer referencia a una cadena de clústeres, se utiliza una tabla de asignación de archivos (FAT) que mantiene un registro de todos los clústeres que utiliza un archivo.

Los siguientes sistemas de archivos se utilizan en los sistemas IA-32:

- El sistema de archivos FAT12 se utilizó por primera vez en los disquetes de la IBM-PC.
- El sistema de archivos FAT16 es el único formato disponible para las unidades de disco que se utilizan en MS-DOS.
- El sistema de archivos FAT32 se introdujo con la versión OEM2 de Windows 95, y se mejoró en Windows 98.
- El sistema de archivos NTFS lo soportan Windows NT, 2000 y XP.

Todo disco en los sistemas de archivos tipo FAT y NTFS tiene un *directorio raíz*, el cual es la lista principal de archivos en el disco. El directorio raíz también puede contener los nombres de otros directorios, llamados subdirectorios.

MS-DOS y Windows utilizan una tabla llamada *tabla de asignación de archivos* (FAT) para llevar un registro de la ubicación de cada archivo en el disco. La FAT asigna clústeres de disco específicos a los archivos. Cada entrada corresponde a un número de clúster, y cada clúster se asocia con uno o más sectores.

En el modo de direccionamiento real, INT 21h proporciona funciones (tabla 14-7) para crear y modificar directorios, modificar los atributos de los archivos, buscar archivos que coincidan, etcétera. Estas funciones tienden a estar menos disponibles en los lenguajes de alto nivel.

El programa de Visualización de sectores lee y muestra cada sector del disquete en la unidad A.

El programa Espacio libre en disco muestra el tamaño del volumen de disco seleccionado y la cantidad de espacio libre.

14.7 Ejercicios de programación

Los siguientes ejercicios deben compilarse y ejecutarse en modo de direccionamiento real. Asegúrese de realizar una copia de seguridad de cualquier disco que se vea afectado por estos programas, o cree un disco temporal para usarlo mientras los prueba. *Bajo ninguna circunstancia deberá ejecutar los programas en un disco fijo, ¡sino hasta que los haya depurado con mucho cuidado!*

1. Establecer la unidad de disco predeterminada

Escriba un procedimiento que pida al usuario una unidad de disco duro (A, B, C o D) y después establezca la unidad predeterminada a la elección del usuario.

2. Espacio en disco

Escriba un procedimiento llamado **Obtener_TamDisco** que devuelva la cantidad de espacio de datos total en una unidad de disco seleccionada. *Entrada:* AL = número de unidad (0 = A, 1 = B, 2 = C, ...). *Salida:* DX:AX = espacio de datos, en bytes.

3. Espacio libre en disco

Escriba un procedimiento llamado **Obtener_EspaciolibreDisco** que devuelva la cantidad de espacio libre en una unidad de disco seleccionada. *Entrada:* DS:DX apunta a una cadena que contiene el especificador de la unidad. *Salida:* EDX: EAX = espacio libre en disco, en bytes. Escriba un programa para probar este procedimiento y mostrar el resultado de 64 bits en hexadecimal.

4. Mostrar atributos de un archivo

Escriba un procedimiento llamado **MostrarAtributosArchivo** que reciba el desplazamiento de un nombre de archivo en DX y muestre los atributos del archivo en la ventana de consola. Los atributos a buscar son normal, oculto, sólo lectura y sistema. *Sugerencia:* use la función 7143h de INT 21h.

Escriba un programa que llame a **MostrarAtributosArchivo** y le pase el nombre de un archivo. Antes de ejecutar su programa, establezca los atributos del archivo desde el Explorador de Windows, haciendo clic con el botón derecho en el nombre del archivo, seleccionando Propiedades y haciendo clic en las opciones Oculto y Sólo lectura. De manera alternativa, puede ejecutar el comando Attrib desde el símbolo del sistema de Windows. Ejecute su programa y verifique que sea correcta la visualización de los atributos. Resultados de ejemplo:

```
atributos de temp.txt: Oculto Solo-lectura
```

5. Espacio libre del disco, en clústeres

Modifique el programa de Espacio libre en el disco de la sección 14.5.1, de manera que muestre la siguiente información:

Especificacion de unidad:	"C:\\"
Bytes por sector:	512
Sectores por cluster:	8
Numero total de clusteres:	999999
Numero de clusteres disponibles:	99999

6. Mostrar el número de sector

Utilice el programa de Visualización de sectores (sección 14.4.1) como punto de inicio para mostrar una cadena al principio de la pantalla, que indique el especificador de la unidad y el número de sector actual (en hexadecimal).

7. Visualización de sectores en hexadecimal

Use el programa de Visualización de sectores (sección 14.4.1) como punto de inicio; agregue código que permita al usuario oprimir F2 para mostrar el sector actual en hexadecimal, con 24 bytes en cada línea. El desplazamiento del primer byte en cada línea deberá mostrarse al principio de la misma. La pantalla será de 22 líneas de alto con una línea parcial al final. A continuación se muestra un ejemplo de las primeras dos líneas, para mostrar la distribución:

```
0000 17311625 25425B75 279A4909 200D0655 D7303825 4B6F9234  
0018 273A4655 25324B55 273A4959 293D4655 A732298C FF2323DB  
(etc.)
```

Nota final

1. Consulte el tutorial sobre DEBUG en el sitio Web del libro.

PROGRAMACIÓN A NIVEL DEL BIOS

- 15.1 Introducción
 - 15.1.1 Área de datos del BIOS
- 15.2 Entrada de teclado mediante INT 16h
 - 15.2.1 Cómo funciona el teclado
 - 15.2.2 Funciones de INT 16h
 - 15.2.3 Repaso de sección
- 15.3 Programación de VIDEO con INT 10h
 - 15.3.1 Fundamentos
 - 15.3.2 Control del color
 - 15.3.3 Funciones de video de INT 10h
 - 15.3.4 Ejemplos de procedimientos de la biblioteca
 - 15.3.5 Repaso de sección
- 15.4 Dibujo de gráficos mediante INT 10h
 - 15.4.1 Funciones de INT 10h relacionadas con píxeles
- 15.4.2 Programa DibujarLinea
- 15.4.3 Programa de coordenadas cartesianas
- 15.4.4 Conversión de coordenadas cartesianas a coordenadas de pantalla
- 15.4.5 Repaso de sección
- 15.5 Gráficos de mapas de memoria
 - 15.5.1 Modo 13h: 320 × 200, 256 colores
 - 15.5.2 Programa de gráficos de mapas de memoria
 - 15.5.3 Repaso de sección
- 15.6 Programación del ratón
 - 15.6.1 Funciones Int 33h para el ratón
 - 15.6.2 Programa para rastrear el ratón
 - 15.6.3 Repaso de sección
- 15.7 Resumen del capítulo
- 15.8 Ejercicios del capítulo

15.1 Introducción

Leer este capítulo es como retroceder en la historia. Cuando apareció la primera IBM-PC, muchos programadores (incluyéndome a mí) querían saber cómo meterse a la caja y trabajar directamente con el hardware de la computadora. Peter Norton descubrió con rapidez todo tipo de información útil y secreta, lo cual lo llevó a publicar su reconocido libro *Inside the IBM-PC*. En un gesto de generosidad, IBM publicó todo el código fuente en lenguaje ensamblador para el BIOS de la IBM PC/XT (aún conservo una copia). Los diseñadores pioneros de juegos como Michael Abrash (autor de *Quake* y *Doom*) aprendieron a optimizar el software de gráficos y sonido, usando su conocimiento sobre el hardware de la PC¹. Ahora usted puede unirse a este distinguido grupo y trabajar detrás de las cámaras, debajo de MS-DOS y Windows, al nivel del BIOS (*sistema básico de entrada-salida*). ¿Es obsoleta esta información? Definitivamente no, si trabaja en aplicaciones de sistemas incrustados o si quiere aprender acerca del diseño del BIOS de la computadora.

Todos los programas en este capítulo son aplicaciones en modo real de 16 bits. Puede desarrollar y ejecutar los programas que mostramos en este capítulo en cualquier versión de Microsoft Windows. Aquí aprenderá cosas útiles, como:

- Lo que ocurre cuando se oprime una tecla en el teclado, y en dónde terminan los caracteres.

- Cómo verificar el búfer del teclado, para ver si hay caracteres en espera, y cómo borrar del búfer los tecleos anteriores.
- Cómo leer las teclas que no son ASCII, como las teclas de función y las flechas del cursor.
- Cómo mostrar texto a color y por qué los colores se basan en el sistema de mezcla de colores RGB de la pantalla de video.
- Cómo dividir la pantalla en paneles de colores y desplazar cada uno de ellos por separado.
- Cómo dibujar gráficos de mapas de bits a 256 colores.
- Cómo detectar los movimientos y los clics del ratón.

15.1.1 Área de datos del BIOS

El área de datos del BIOS, que se muestra parcialmente en la tabla 15-1, contiene los datos del sistema que utilizan las rutinas de servicio del BIOS de ROM. Por ejemplo, el búfer de escritura adelantada 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.

Tabla 15-1 Área de datos del BIOS, en el segmento 0040h.

Desplazamiento Hex	Descripción
0000-0007	Direcciones de puertos, COM1-COM4
0008-000F	Direcciones de puertos, LPT1-LPT4
0010-0011	Lista del hardware instalado
0012	Bandera de inicialización
0013-0014	Tamaño de memoria, en kilobytes
0015-0016	Memoria en el canal de E/S
0017-0018	Banderas de estado del teclado
0019	Almacenamiento alternativo de entrada de teclas
001A-001B	Apuntador del búfer del teclado (inicio)
001C-001D	Apuntador del búfer del 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

15.2 Entrada de teclado mediante INT 16h

En la sección 2.5 diferenciamos los distintos niveles de entrada-salida disponibles para los programas en lenguaje ensamblador. En este capítulo tendrá 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 tiene mucha flexibilidad y control.

El BIOS maneja la entrada del teclado mediante llamadas a la interrupción 16h. Las rutinas de BIOS no permiten la redirección, pero facilitan la lectura de las 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, el cual se muestra en la solapa de este libro. 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:

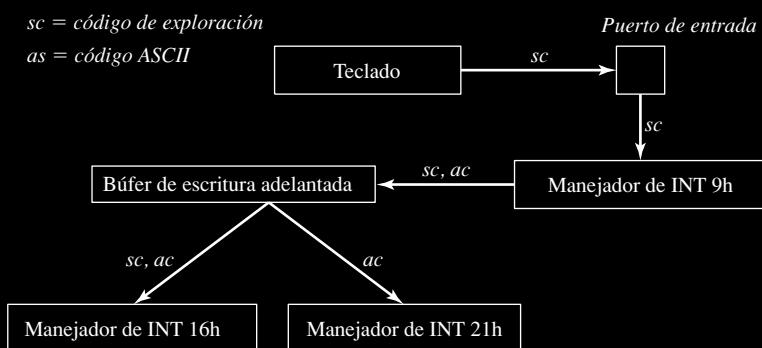
Teclas	Código ASCII
Ins, Supr, AvPág, RePág, Inicio, Fin, Flecha arriba, Flecha abajo, Flecha izquierda, Flecha derecha	E0h
Teclas de función (F1 – F12)	00h

15.2.1 Cómo funciona el teclado

La entrada del teclado sigue una ruta 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* (vea la figura 15-1). Pueden mantenerse hasta 15 tecleo 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 diseñado 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 adelantada del teclado (si el código de exploración no tiene un código ASCII que coincida, el código ASCII de la tecla en el búfer de escritura adelantada es igual a cero o E0h).

FIGURA 15-1 Secuencia del procesamiento de tecleo.



Una vez que el código de exploración y el código ASCII se encuentran seguros en el búfer 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 a nivel 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 obtenga 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. En la sección 13.2.3 explicamos la entrada del teclado mediante INT 21h.

15.2.2 Funciones de INT 16h

INT 16h presenta varias ventajas concisas en comparación con INT 21h, para el manejo del teclado. En primer lugar, 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 el establecimiento de la velocidad de repetición y obtener el estado de las banderas del teclado. La *velocidad de repetición* 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 un retraso 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: 0 = más rápida (30/seg), 1Fh = más lenta (2/seg)
Devuelve	Nada
Llamada de ejemplo	<pre>mov ax,0305h mov bh,1 ; retraso de repetición de 500ms mov bl,0Fh ; velocidad de repetición int 16h</pre>

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	<pre>mov ah,5 mov ch,3Bh ; código de exploración para la tecla F1 mov cl,0 ; código ASCII int 16h</pre>

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 en 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	<pre>mov ah,10h int 16h mov codigoExpl,ah mov codigoASCII,al</pre>
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 la tecla 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 a **DumpRegs** muestra todos los registros, pero sólo necesitamos ver AH (código de exploración) y AL (código ASCII). Por ejemplo, cuando el usuario oprime la tecla de función F1, éste es el resultado que se muestra (3B00h):

EAX=00003800	EBX=00000000	ECX=000000FF	EDX=000005D6
ESI=00000000	EDI=00002000	EBP=0000091E	ESP=00002000
EIP=0000000F	EFL=00003202	CF=0 SF=0 ZF=0 OF=0 AF=0	PF=0

Comprobar 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 la siguiente tecla disponible, si la hay. Puede utilizar esta función dentro de un ciclo para llevar a cabo otras tareas del programa. Observe que la función no elimina la tecla del búfer de escritura adelantada. En la siguiente tabla podrá consultar los detalles:

Función 11h de INT 16h	
Descripción	Comprueba el búfer del teclado
Recibe	AH = 11h
Devuelve	Sí hay una tecla en espera, ZF = 0, AH = código de exploración, AL = código ASCII; en caso contrario, ZF = 1
Llamada de ejemplo	<pre>mov ah,11h int 16h jz NoHayTeclaEnEspera ; no hay tecla en el búfer mov codigoExpl,ah mov codigoASCII,al</pre>
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 haya notado que los programas de procesamiento de palabras 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	<pre>mov ah,12h int 16h mov banderasTeclado,ax</pre>
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 tabla 15.2, son particularmente interesantes debido a que nos dicen mucho acerca de lo que el usuario está haciendo con el teclado. ¿Está oprimiendo la tecla de 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.

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 comprueban con frecuencia el búfer del teclado, para ver si se oprimieron teclas de flechas y otras teclas 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.

Tabla 15-2 Valores de la bandera del teclado.^a

Bit	Descripción
0	Se oprimió 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 activó la tecla Bloq Núm
6	Se activó la tecla Bloq Mayús
7	Se activó la tecla Insert
8	Se oprimió la tecla Ctrl izquierda
9	Se oprimió la techa 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

^aFuente: Ray Duncan, *Advanced MS-DOS Programming*, 2^a edición, Microsoft Press, 1988. pp. 586-587.

Mediante el uso de las funciones de 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:

```

    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 ; ¿fué la tecla para salir?
    je    terminar        ; sí: termina ahora (ZF=1)
    jmp   L1              ; no: comprueba el búfer otra vez

noHayTecla:
    or    al,1            ; no se oprimió ninguna tecla
                        ; borra la bandera Cero
terminar:
    pop  ax
    ret
BorrarTeclado ENDP
END main

```

El programa muestra un punto en la pantalla cada 300 milisegundos. Al probarlo, oprima cualquier secuencia de teclas aleatorias, las cuales se ignoran y se eliminan del búfer de escritura adelantada. El programa se detendrá tan pronto como se oprima ESC.

15.2.3 Repaso de sección

1. ¿Qué interrupción (16h o 21h) es mejor para leer la entrada del usuario que incluye teclas de funciones y otras teclas extendidas?
2. ¿En qué parte de la memoria se mantienen los caracteres que se reciben del teclado, mientras esperan a que los programas de aplicaciones los procesen?
3. ¿Qué operaciones realiza la rutina de servicio INT 9h?
4. ¿Qué función de INT 16h mete teclas en el búfer del teclado?
5. ¿Qué función de INT 16h elimina la siguiente tecla disponible del búfer del teclado?
6. ¿Qué función de INT 16h examina el búfer del teclado y devuelve el código de exploración y el código ASCII de la primera entrada disponible?
7. ¿La función 11h de INT 16h elimina un carácter del búfer del teclado?
8. ¿Qué función de INT 16h nos proporciona el valor del byte de la bandera del teclado?
9. ¿Qué bit en el byte de la bandera del teclado indica que se oprimió la tecla Bloq Despl?
10. Escriba instrucciones para recibir como entrada el byte de la bandera del teclado y repetir un ciclo hasta que se oprima la tecla Ctrl.

11. *Reto:* el procedimiento **BorrarTeclado** de la sección 15.2.2 comprueba sólo un código de exploración del teclado. Suponga que su programa tiene que comprobar varios códigos de exploración (por ejemplo, las cuatro flechas del cursor). Sin escribir código, sugiera las modificaciones que podría hacer al procedimiento para que esto sea posible.

15.3 Programación de VIDEO con INT 10h

15.3.1 Fundamentos

Tres niveles de acceso

Cuando un programa de aplicación necesita escribir caracteres en la pantalla, en modo de texto, puede elegir de 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 texto en la pantalla de video. La entrada/salida puede redirigirse fácilmente hacia otros dispositivos, como una impresora o un disco. La salida es bastante lenta, y no podemos controlar el color del texto.
- **Acceso a nivel del 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 mueven directamente a la RAM de video, por lo que la ejecución es instantánea. La salida no puede redirigirse. Durante la era del MS-DOS, los programas procesadores de palabras y las hojas electrónicas de cálculo utilizaban 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 del 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 deben 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, cree un acceso directo al archivo EXE del programa. Después abra el cuadro de diálogo Propiedades para el acceso directo, seleccione Opciones y *Modo de pantalla completa* en el grupo Uso de la ficha Pantalla (Display Options).
- Abra una ventana de Símbolo del sistema desde el menú Inicio, y oprima Alt-Intro para cambiar al modo de pantalla completa. Use el comando CD (cambiar directorio), navegue hasta el directorio de su archivo EXE y ejecute el programa escribiendo su nombre. Alt-Intro es un *comutador*, por lo que si lo oprime 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 integrado en el BIOS genera 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 fuentes 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 el 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 izquierdo de la pantalla, columna 0. Cada columna es la anchura de una celda de carácter.

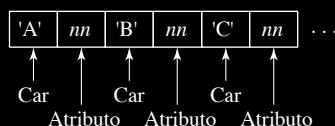
Fuentes Los caracteres se generan desde una tabla residente en memoria de fuentes de caracteres. El BIOS permite a los programas reescribir las tablas de caracteres en tiempo de ejecución, por lo que pueden mostrarse fuentes personalizadas.

Páginas de texto de video La memoria de video en modo de 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 las otras páginas 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, pero no la veremos en este capítulo). 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:

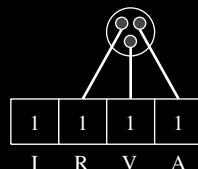


Destello Los caracteres en la pantalla de video pueden destellar. Para ello, el controlador de video invierte los colores de texto y de fondo de un carácter, a una velocidad predeterminada. De manera predeterminada, cuando una PC inicia en el modo de MS-DOS, el destello está habilitado. Es posible desactivarlo 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 estando en MS Windows.

15.3.2 Control del color

Mezcla de colores primarios

Cada píxel 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 píxel. Por lo tanto, todos los colores de texto disponibles se pueden representar 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 píxel blanco:



Al mezclar los tres colores primarios (tabla 15-3) pueden generarse nuevos colores. Además, al encender el bit de intensidad podemos hacer los colores mezclados más brillantes.

Tabla 15-3 Ejemplo de mezcla de colores.

Mezcle estos colores primarios...	Para obtener este color	Agregue el bit de intensidad
rojo + verde + azul	gris oscuro	blanco
verde + azul	cyan	cyan claro
rojo + azul	magenta	magenta claro
rojo + verde	café	amarillo
(ningún color)	negro	gris oscuro

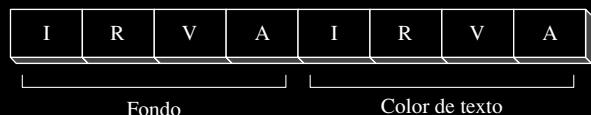
Los colores primarios estilo MS-DOS y los colores mixtos se compilan en una lista de todos los posibles colores de 4 bits, como se muestra en la tabla 15-4. Cada color en la columna de la derecha tiene activado su bit de intensidad.

Tabla 15-4 Codificación de texto a color de cuatro bits.

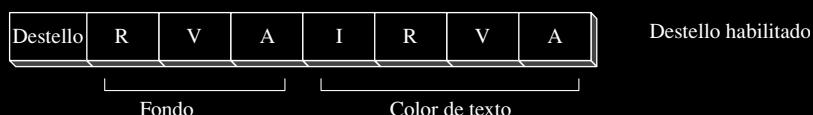
IRVA	Color	IRVA	Color
0000	negro	1000	gris
0001	azul	1001	azul claro
0010	verde	1010	verde claro
0011	cyan	1011	cyan claro
0100	rojo	1100	rojo claro
0101	magenta	1101	magenta claro
0110	café	1110	amarillo
0111	gris claro	1111	blanco

Byte 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 del color de fondo controla el destello del carácter. Cuando este bit se activa, el carácter destella:



Cuando está habilitado el destello, sólo los colores de baja intensidad en la columna izquierda de la tabla 15-4 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 000000111 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), use el operador SHL de ensamblador para desplazar los bits del color de fondo cuatro posiciones a la izquierda, y aplique un OR entre estos bits y el color del 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
cafe = 110b
mov bh,((cafe SHL 4) OR azul) ; 01100001
```

Las fuentes y los 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, a menos que cambiamos a modo de pantalla completa. Lo mismo se aplica para la visualización de gráficos mediante INT 10h.

15.3.3 Funciones de video de INT 10h

La tabla 15-5 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 (sección 15.4).

Tabla 15-5 Funciones selectas de INT 10h.

Número de función	Descripción
0	Establece la pantalla de video a uno de los modos de texto o de gráficos
1	Establece las líneas del cursor, con lo 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 píxel de gráficos en la pantalla, en modo de gráficos (vea el apéndice C)
0Dh	Lee el color de un píxel de gráficos individual, en una ubicación dada (vea el apéndice C)
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 (vea el apéndice C)

Es conveniente preservar los registros de propósito general (mediante PUSH) antes de llamar 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 tabla 15-6 presenta los modos de texto que se utilizan con más frecuencia:

Tabla 15-6 Modos de texto de video reconocidos por INT 10h.

Modo	Resolución (columnas x filas)	Número de colores
0	40 × 25	16
1	40 × 25	16
2	80 × 25	16
3	80 × 25	16
7 ^a	80 × 25	2
14h	132 × 25	16

^a Monitor monocromático.

Es conveniente obtener el modo de video actual (función 0Fh de INT 10h) y guardarla en una variable, antes de asignarle un nuevo valor. Después puede restaurar el modo de video original cuando termine su programa. La siguiente tabla muestra cómo establecer el modo de video.

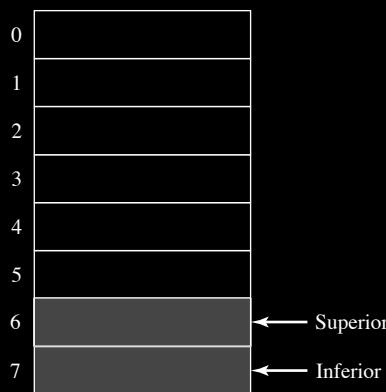
Función 0 de INT 10h	
Descripción	Establece el modo de video
Recibe	AH = 0 AL = modo de video
Devuelve	Nada
Llamada de ejemplo	<code>mov ah,0 mov al,3 ; modo de video 3 (texto a color) int 10h</code>
Notas	La pantalla se borra de manera automática, a menos que se active el bit superior en AL antes de 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 01h de INT 10h	
Descripción	Establece las líneas del cursor
Recibe	AH = 01h CH = línea superior CL = línea inferior
Devuelve	Nada
Llamada de ejemplo	<code>mov ah,1 mov cx,0607h ; tamaño predeterminado de cursor a color int 10h</code>
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:



Establecer posición del cursor (02h)

La función 2 de INT 10h localiza el cursor en una fila y columna específicas en la página de video de su elección, como puede verse en la siguiente tabla.

Función 02h de INT 10h	
Descripción	Establece la posición del cursor
Recibe	AH = 2 DH, DL = valores de fila, columna BH = página de video
Devuelve	Nada
Llamada de ejemplo	<pre>mov ah,2 mov dh,10 ; fila 10 mov dl,20 ; columna 20 mov bh,0 ; página de video 0 int 10h</pre>
Notas	Para los modos de 80 × 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 qué opción se seleccionó del menú.

Función 03h de INT 10h	
Descripción	Obtiene la posición y el tamaño del cursor
Recibe	AH = 3 BH = página de video
Devuelve	CH, CL = líneas de exploración inicial y final del cursor DH, DL = fila, columna de la ubicación del cursor
Llamada de ejemplo	<pre>mov ah,3 mov bh,0 ; página de video 0 int 10h mov cursor,CX mov posicion,DX</pre>

Mostrar y ocultar el cursor Es útil poder ocultar de manera temporal el cursor al mostrar menús, escribir en forma continua en la pantalla o leer la entrada del ratón. Para ocultar el cursor, podemos establecer el valor de su línea superior a un valor ilegal (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 ; establece 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).

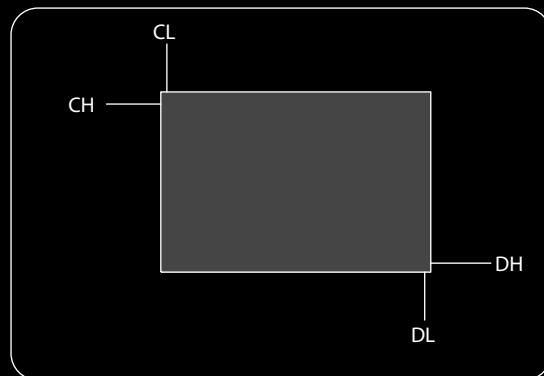
Desplazar ventana hacia arriba (06h)

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 numeradas de 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 figura 15-2, los registros CH/CL definen la fila y la columna de la esquina superior izquierda, y DH/DL definen la fila y columna de la esquina inferior derecha. Esta función no tiene un efecto predecible sobre la posición del cursor.

A medida que se desplaza una ventana hacia arriba, su línea inferior se sustituye por una línea en blanco. Si todas las líneas 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 líneas a 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 inferior derecha de la ventana
Devuelve	Nada
Llamada de ejemplo	<pre>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 ; llamada al BIOS</pre>

FIGURA 15-2 Definición de una ventana mediante el uso de 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 escribe texto dentro de la ventana mediante una llamada a una función del DOS, el texto utilizará los mismos colores de texto y de fondo. El siguiente programa (*VenTexto.asm*) demuestra esta técnica:

```
TITLE Ventana de texto a color          (VenTexto.asm)
; Muestra una ventana a color y escribe texto en su interior.

INCLUDE Irvine16.inc
.data
mensaje BYTE "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 desplazar 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 carácter 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 la pantalla* (*screen scraping*). Los programas podrían convertir el texto a voz para los usuarios con discapacidad auditiva.

Función 08h de INT 10h	
Descripción	Lee el carácter y el atributo en la posición actual del cursor
Recibe	AH = 8 BH = página de video
Devuelve	AL = código ASCII del carácter AH = atributo del carácter
Llamada de ejemplo	<pre> mov ah,8 mov bh,0 ; página de video 0 int 10h mov car,al ; guarda el carácter mov atrib,ah ; guarda el atributo </pre>

Escribir carácter y atributo (09h)

La función 9 de INT 10h escribe un 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 de gráficos especiales del BIOS que coinciden con los códigos ASCII del 1 al 31.

Función 09h de INT 10h	
Descripción	Escribe el 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	<pre>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 int 10h</pre>
Notas	No avanza el cursor después de escribir el carácter. Puede llamarse en modos de texto y de gráficos

La *cuenta de repetición* en CX especifica cuántas veces se va a repetir el carácter (no debe repetirse más allá de la línea actual en 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 0h 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	<pre>mov ah,0Ah mov al,'A' ; carácter ASCII mov bh,0 ; página de video 0 mov cx,1 ; cuenta de repetición int 10h</pre>
Notas	No avanza el cursor

Obtener información del modo de video (0Fh)

La función 0Fh de INT 10h devuelve información acerca del modo actual de video, incluyendo el número del modo, el número de columnas en la pantalla y el número de la página activa de video, como podemos ver en la siguiente tabla. Esta función es útil al principio de un programa, cuando queremos guardar el modo de video actual y cambiar a un nuevo modo. Al terminar el programa, podemos restablecer el modo de video (con la función 0 de INT 10h) al valor guardado.

Función 0h 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 píxeles) BH = página activa de video
Llamada de ejemplo	<pre>mov ah,0Fh int 10h mov modov,al ; guarda el modo mov columnas,ah ; guarda las columnas mov pagina,bh ; guarda la página</pre>
Notas	Trabaja en modos 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 intensidad o el destello del carácter. En la siguiente tabla podrá 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	<pre>mov ah,10h mov al,3 mov bl,1 ; habilita el destello int 10h</pre>
Notas	Cambia el texto en la pantalla, entre el modo de destello y el modo de alta intensidad. En MS Windows, el destello sólo puede ocurrir cuando la aplicación se ejecuta en modo de pantalla completa

Escribir cadena en modo de 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 (vea el programa *CadColor2.asm* en la carpeta cap15 de los códigos de este libro). 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 (vea 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	<pre>.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 ; longitud cadena mov dh,fila ; fila inicial mov dl,columna ; columna inicial mov bp,OFFSET cadenaColor ; ES:BP apunta a la cadena ipnt 10h</pre>
Notas	<p>Puede llamarse cuando el adaptador de pantalla está en modo de texto o de gráficos. Valores del modo de escritura:</p> <ul style="list-style-type: none"> • 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 códigos de caracteres y bytes de atributos, alternando uno y uno; la posición del cursor no se actualiza después de la escritura. • 3 = la cadena contiene códigos de caracteres y bytes de atributos, alternando uno y uno; la posición del cursor se actualiza después de la escritura.

Ejemplo: mostrar una cadena a color

El siguiente programa (*CadColor.asm*) muestra una cadena en la consola, usando un color distinto para cada carácter. Debe ejecutarse en modo de pantalla completa si desea ver que los caracteres destellen. El destello está habilitado de manera predeterminada, pero puede eliminar la llamada a **HabilitarDestello** y ver la misma cadena en un fondo gris oscuro:

```
TITLE Ejemplo de cadenas a color           (CadColor.asm)

INCLUDE Irvine16.inc
.data
ATRIB_SUP = 10000000b
cadena BYTE "ABCDEFGHIJKLMOP"
color  BYTE 1

.code
main PROC
```

```

        mov  ax,@data
        mov  ds,ax
        call ClrScr
        call HabilitarDestello
        mov  cx,SIZEOF cadena
        mov  si,OFFSET cadena

L1: push  cx          ; guarda el contador del ciclo
    mov  ah,9          ; escribe carácter/atributo
    mov  al,[si]        ; carácter a mostrar
    mov  bh,0          ; página de video 0
    mov  bl,color      ; atributo
    or   bl,ATRIB_SUP ; establece bit de destello e intensidad
    mov  cx,1          ; lo muestra una vez
    int  10h
    mov  cx,1          ; avanza el cursor a la
    call AvanzarCursor ; siguiente columna de la pantalla
    inc  color         ; siguiente color
    inc  si            ; siguiente caracter
    pop  cx            ; restaura el contador del ciclo
Loop L1

    call Crlf
    exit
main ENDP

;-----
HabilitarDestello PROC
;
; Habilita el destello (usando el bit superior de los atributos
; de color). En MS Windows, esto sólo funciona si
; el programa se ejecuta en modo de pantalla completa.
; Recibe: nada.
; Devuelve: nada
;-----
    push ax
    push bx
    mov ax,1003h        ; habilita el destello o el modo de intensidad
    mov bl,1             ; se habilita el destello
    int 10h
    pop bx
    pop ax
    ret
HabilitarDestello ENDP

```

El procedimiento AvanzarCursor puede usarse en cualquier programa que llame a las funciones de texto INT 10h.

```

;-----
AvanzarCursor PROC
;
; Avanza el cursor n columnas a la derecha.
; Recibe: CX = número de columnas
; Devuelve: nada
;-----
    pusha
L1: push  cx          ; guarda el contador del ciclo
    mov  ah,3          ; obtiene la posición del cursor
    mov  bh,0          ; y la coloca en DH, DL
    int  10h          ; ¡cambia el registro CX!

```

```

inc  dl          ; incrementa la columna
mov  ah,2        ; establece la posición del cursor
int  10h
pop  cx          ; restaura el contador del ciclo
loop L1          ; siguiente columna

popa
ret
AvanzarCursor ENDP
END main

```

15.3.4 Ejemplos de procedimientos de la biblioteca

Vamos a ver dos procedimientos útiles pero simples, de la biblioteca de vínculos Irvine16: **Gotoxy** y **Clrsqr** (*Nota: en estos ejemplos se tradujeron los comentarios para facilitar su comprensión al lector. Los procedimientos originales están en inglés*).

Procedimiento Gotoxy

El procedimiento **Gotoxy** establece la posición del cursor en la página de video 0:

```

;-----
Gotoxy PROC
;
; Establece la posición del cursor en la página de video 0.
; Recibe: DH,DL = fila, columna
; Devuelve: nada
;
pusha
mov   ah,2          ; establece la posición del cursor
mov   bh,0          ; página de video 0
int  10h
popa
ret
Gotoxy ENDP

```

Procedimiento Clrsqr

El procedimiento **Clrsqr** borra la pantalla y posiciona el cursor en la fila 0, columna 0 en la página de video 0:

```

;-----
Clrsqr PROC
;
; Borra la pantalla (página de video 0) y posiciona el cursor
; en la fila 0, columna 0.
; Receives: nada
; Devuelve: nada
; Actualmente sólo afecta a 25 filas y 80 columnas
;
pusha
mov   ax,0600h      ; desplaza la ventana hacia arriba
mov   cx,0           ; esquina superior izquierda (0,0)
mov   dx,184Fh       ; esquina inferior derecha (24,79)
mov   bh,7           ; atributo normal
int  10h            ; llamada al BIOS
mov   ah,2           ; posiciona el cursor en 0,0
mov   bh,0           ; página de video 0
mov   dx,0           ; fila 0, columna 0
int  10h
popa
ret
Clrsqr ENDP

```

15.3.5 Repaso de sección

1. ¿Cuáles son los tres niveles de acceso para la pantalla de video que se mencionan al principio de esta sección?
2. ¿Qué nivel de acceso produce los resultados con más rapidez?
3. ¿Cómo se ejecuta un programa en modo de pantalla completa?
4. Cuando una computadora se inicia en MS-DOS, ¿cuál es el modo de video predeterminado?
5. ¿Qué información contiene cada posición en la pantalla de video para un solo carácter?
6. ¿Qué rayos de electrones se requieren para generar cualquier color en una pantalla de video?
7. Muestre la asignación de los colores de texto y de fondo en el byte de atributos de video.
8. ¿Qué función de INT 10h posiciona el cursor en la pantalla?
9. ¿Qué función de INT 10h desplaza una ventana rectangular hacia arriba?
10. ¿Qué función de INT 10h escribe un carácter y un atributo en la posición actual del cursor?
11. ¿Qué función de INT 10h establece el tamaño del cursor?
12. ¿Qué función de INT 10h obtiene el modo de video actual?
13. ¿Qué parámetros se requieren para establecer la posición del cursor mediante INT 10h?
14. ¿Cómo es posible ocultar el cursor?
15. ¿Qué parámetros se requieren para desplazar una ventana hacia arriba?
16. ¿Qué parámetros se requieren para escribir un carácter y un atributo en la posición actual del cursor?
17. ¿Qué función INT 10h establece los modos de destello e intensidad?
18. ¿Qué valores deben moverse a AH y AL para borrar la pantalla mediante la función 6 de INT 10h?
19. *Reto:* si tiene un perro, ¿cree que podría sorprenderle que usted pasara varias horas seguidas observando una pantalla de computadora en color gris?

15.4 Dibujo de gráficos mediante INT 10h

La función 0Ch de INT 10h dibuja un píxel individual en modo de gráficos. Podría utilizarlo para dibujar formas y líneas complejas, pero es demasiado lento. Para aprender los fundamentos, empezaremos con esta función y más adelante le mostraremos cómo dibujar gráficos, escribiendo directamente en la RAM de video.

Puede dibujar texto en la pantalla usando 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 muestran en la tabla 15-7. Cada modo puede establecerse mediante el uso de la función 0 de INT 10h (establecer modo de video).

Tabla 15-7 Modos gráficos de video reconocidos por INT 10h.

Modo	Resolución (columnas X filas, en píxeles)	Número de colores
6	640 × 200	2
0Dh	320 × 200	16
0Eh	640 × 200	16
0Fh	640 × 350	2
10h	640 × 350	16
11h	640 × 480	2
12h	640 × 480	16
13h	320 × 200	256
6Ah	800 × 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 pantalla 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.

15.4.1 Funciones de INT 10h relacionadas con píxeles

Escribir pixel de gráficos (0Ch)

La función 0Ch de INT 10h, como se muestra en la siguiente tabla, dibuja un píxel 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 la memoria de video después de calcular el número de colores por píxel, la resolución horizontal, etcétera.

Función 0Ch de INT 10h	
Descripción	Escribe un píxel de gráficos
Recibe	AH = 0Ch AL = valor del píxel BH = página de video CX = coordenada x DX = coordenada y
Devuelve	Nada
Llamada de ejemplo	<pre>mov ah,0Ch mov al,valorPixel mov bh,paginaVideo mov cx,coord_x mov dx,coord_y int 10h</pre>
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 pixel 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 la columna, y devuelve el valor del píxel 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	<pre>mov ah,0Dh mov bh,0; ; página de video 0 mov cx,coord_x mov dx,coord_y int 10h mov valorPixel,al</pre>
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

15.4.2 Programa DibujarLinea

El programa *DibujarLinea* cambia a modo de gráficos mediante el uso de INT 10h, escribe el nombre del programa en texto y dibuja una línea recta horizontal. Si lo ejecuta en MS Windows, cambie la ventana de la consola al modo de pantalla completa, oprimiendo Alt-Intro.² A continuación se muestra el listado del programa completo:

```
TITLE Programa DibujarLinea           (DibujarLinea.asm)
; Este programa dibuja texto y una línea recta en modo de gráficos.

INCLUDE Irvine16.inc

;----- Constantes de modo de video -----
Modo_06 = 6; 640 X 200, 2 colores
Modo_0D = 0Dh; 320 X 200, 16 colores
Modo_0E = 0Eh; 640 X 200, 16 colores
Modo_0F = 0Fh; 640 X 350, 2 colores
Modo_10 = 10h; 640 X 350, 16 colores
Modo_11 = 11h; 640 X 480, 2 colores
Modo_12 = 12h; 640 X 480, 16 colores
Modo_13 = 13h; 320 X 200, 256 colores
Modo_6A = 6Ah; 800 X 600, 16 colores

.data
guardaModo BYTE ?           ; guarda el modo de video actual
xActual    WORD 100          ; número de columna (coordenada X)
yActual    WORD 100          ; número de fila (coordenada Y)
COLOR = 1001b                ; color de línea (cyan)

tituloProg BYTE "Pixel1.asm"
TITULO_FILA = 5
TITULO_COLUMNNA = 14

; Cuando use un modo de 2 colores, establezca COLOR a 1 (blanco)

.code
main PROC
    mov ax,@data
    mov ds,ax

    ; Guarda el modo de video actual.
    mov ah,0Fh
    int 10h
    mov guardaModo,al

    ; Cambia a un modo de gráficos.
    mov ah,0           ; establece el modo de video
    mov al,Modo_6A
    int 10h

    ; Escribe el nombre del programa, como texto.
    mov ax,SEG tituloProg   ; obtiene el segmento de tituloProg
    mov es,ax              ; lo almacena en ES
    mov bp,OFFSET tituloProg
    mov ah,13h              ; función: escribe una cadena
    mov al,0                ; modo: sólo códigos de caracteres
    mov bh,0                ; página de video 0
    mov bl,7                ; atributo = normal
    mov cx,SIZEOF tituloProg ; longitud de cadena
    mov dh,TITULO_FILA      ; fila (en celdas de caracteres)
    mov dl,TITULO_COLUMNNA  ; columna (en celdas de caracteres)
    int 10h
```

```

; Dibuja una línea recta.
LongitudLinea = 100

mov dx,yActual
mov cx,LongitudLinea ; contador del ciclo

L1:
push cx
mov ah,0Ch ; escribe un píxel
mov al,COLOR ; color del píxel
mov bh,0 ; página de video 0
mov cx,xActual
int 10h
inc xActual
;inc color ; habilite para ver una línea de varios colores
pop cx
Loop L1

; Espera un tecleo.
mov ah,0
int 16h

; Restaura el modo de video inicial.
mov ah,0 ; establece el modo de video
mov al,guardaModo ; modo de video guardado
int 10h
exit
main ENDP
END main

```

Cambiar el modo de video Puede probar distintos modos de gráficos con sólo modificar una instrucción del programa, que selecciona el modo de video 6Ah:

```

mov ah,0 ; establece el modo de video
mov al,modo_64 ; modifique este valor para obtener distintos modos
int 10h ; llama a la rutina de BIOS

```

15.4.3 Programa de coordenadas cartesianas

El programa *Coordenadas cartesianas* dibuja los ejes X y Y de un sistema de coordenadas cartesianas, con el punto de intersección en las ubicaciones X = 400 y Y = 300 en la pantalla. Hay dos procedimientos importantes, **DibujarLineaHorizontal** y **DibujarLineaVertical**, que podrían insertarse sin problema en otros programas de gráficos. El programa establece el adaptador de video al Modo 6Ah (800 × 600, 16 colores).

```

TITLE Coordenadas cartesianas (Pixel2.asm)

; Este programa cambia al modo de gráficos de 800 X 600 y
; dibuja los ejes X, Y de un sistema de coordenadas cartesianas.
; Cambie el modo de pantalla completa antes de ejecutar este programa.
; Las constantes de colores están definidas en Irvine16.inc.

INCLUDE Irvine16.inc

Modo_6A = 6Ah ; 800 X 600, 16 colores
X_ejeY = 300
X_ejeX = 50
X_longEje = 700

Y_ejeX = 400
Y_ejeY = 30
Y_longEje = 540

```

```

.data
guardaModo BYTE ?

.code
main PROC
    mov ax,@data
    mov ds,ax

; Guarda el modo de video actual
    mov ah,0Fh           ; obtiene el modo de video
    int 10h
    mov guardaModo,al

; Cambia a un modo de gráficos
    mov ah,0             ; establece el modo de video
    mov al,Modo_6A        ; 800 X 600, 16 colores
    int 10h

; Dibuja el eje X
    mov cx,X_ejeX        ; coord X del inicio de la línea
    mov dx,X_ejeY        ; coord Y del inicio de la línea
    mov ax,X_longEje     ; longitud de la línea
    mov bl,white          ; color de la línea (vea IRVINE16.inc)
    call DibujaLineaHorizont ; dibuja la línea ahora

; Dibuja el eje Y
    mov cx,Y_ejeX        ; coord X del inicio de la línea
    mov dx,Y_ejeY        ; coord Y del inicio de la línea
    mov ax,Y_longEje     ; longitud de la línea
    mov bl,white          ; color de la línea
    call DibujarLineaVertical ; dibuja la línea ahora

; Espera un tecleo
    mov ah,10h            ; espera una tecla
    int 16h

; Restaura el modo de video inicial
    mov ah,0               ; establece el modo de video
    mov al,guardaModo      ; modo de video guardado
    int 10h

    exit
main endp

;-----
DibujaLineaHorizont PROC
;
; Dibuja una línea horizontal que empieza en la posición X,Y
; con una longitud y color dados.
; Recibe: CX = coordenada X, DX = coordenada Y,
;         AX = longitud y BL = color
; Devuelve: nada
;-----
.data
xActual WORD ?

.code
    pusha
    mov xActual,cx          ; guarda la coordenada X
    mov cx,ax                ; contador del ciclo

```

```

DLH1:
    push  cx                      ; guarda el contador del ciclo
    mov   al,b1                   ; color
    mov   ah,0Ch                  ; dibuja un píxel
    mov   bh,0                     ; página de video
    mov   cx,xActual              ; obtiene la coordenada X
    int   10h
    inc   xActual                ; mueve 1 píxel a la derecha
    pop   cx                      ; restaura el contador del ciclo
    Loop  DLH1

    popa
    ret
DibujaLineaHorizont ENDP

;-----
DibujarLineaVertical PROC
;
; Dibuja una línea vertical que empieza en la posición X,Y
; con una longitud y color dados.
; Recibe: CX = coordenada X, DX = coordenada Y,
;          AX = longitud, BL = color
; Devuelve: nada
;-----
.data
yActual WORD ?

.code
    pusha
    mov   yActual,dx             ; guarda coordenada Y
    mov   xActual,cx             ; guarda coordenada X
    mov   cx,ax                  ; contador del ciclo

DLV1:
    push  cx                      ; guarda el contador del ciclo
    mov   al,b1                   ; color
    mov   ah,0Ch                  ; función: dibuja un píxel
    mov   bh,0                     ; establece la página de video
    mov   cx,xActual              ; establece coordenada X
    mov   dx,yActual              ; establece coordenada Y
    int   10h
    inc   yActual                ; mueve 1 píxel hacia abajo
    pop   cx                      ; restaura el contador del ciclo
    Loop  DLV1

    popa
    ret
DibujarLineaVertical ENDP
END main

```

15.4.4 Conversión de coordenadas cartesianas a coordenadas de pantalla

Los puntos en un gráfico cartesiano no corresponden a las coordenadas absolutas que utiliza el sistema de gráficos del BIOS. En los dos programas de ejemplo anteriores, es evidente que las coordenadas de pantalla empiezan en $sx = 0$, $sy = 0$ en la esquina superior izquierda de la pantalla. Los valores sx crecen a la derecha, y los valores sy crecen hacia la parte inferior de la pantalla. Puede utilizar las siguientes fórmulas para convertir las coordenadas cartesianas X, Y a las coordenadas de pantalla sx, sy :

$$sx = (sXOrig + X) \quad sy = (sYOrig - Y)$$

en donde $sXOrig$ y $sYOrig$ son las coordenadas de pantalla del origen del sistema de coordenadas cartesianas. En el Programa de coordenadas cartesianas (sección 15.4.3), nuestras líneas se intersectan en $sXOrig = 400$ y $sYOrig = 300$, colocando el origen en medio de la pantalla. Si utilizamos los cuatro puntos en la figura 15-3 para probar las fórmulas de conversión, la tabla 15-8 muestra los resultados de los cálculos.

FIGURA 15-3 Coordenadas de prueba para las fórmulas de conversión.

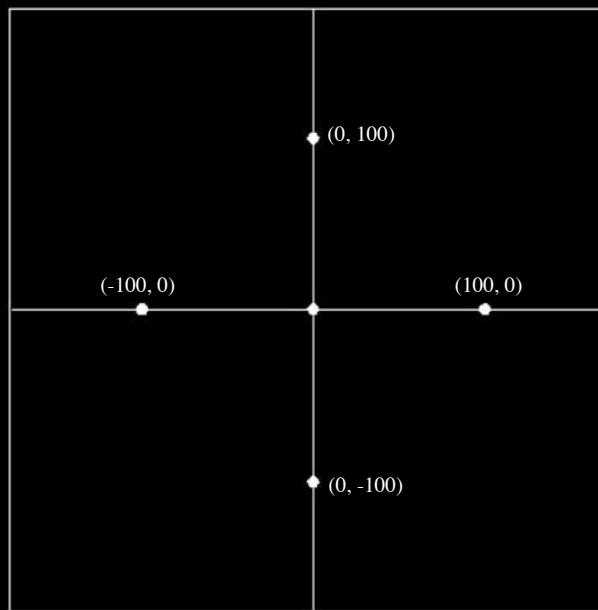


Tabla 15-8 Prueba de las fórmulas de conversión.

Cartesiana (X,Y)	(400 + X, 300 - Y)	Pantalla (sx, sy)
(0,100)	(400 + 0, 300 - 100)	(400, 200)
(100, 0)	(400 + 100, 300 - 0)	(500, 300)
(0, -100)	(400 + 0, 300 - (-100))	(400, 400)
(-100, 0)	(400 + (-100), 300 - 0)	(300, 300)

15.4.5 Repaso de sección

1. ¿Qué función de INT 10h dibuja un solo píxel en la pantalla de video?
2. Al utilizar INT 10h para dibujar un solo píxel, ¿qué valores deben colocarse en los registros AL, BH, CX y DX?
3. ¿Cuál es la principal desventaja de dibujar píxeles mediante el uso de INT 10h?
4. Escriba instrucciones de ASM para establecer el adaptador de video al modo 11h.
5. ¿Qué modo de video es de 800×600 píxeles, a 16 colores?
6. ¿Cuál es la fórmula para convertir una coordenada X cartesiana en coordenadas de píxel de pantalla? Use la variable sx para la columna de la pantalla, y use $sXOrig$ para la columna de la pantalla en la que se encuentra el punto de origen cartesiano (0,0).
7. Si un punto de origen cartesiano se encuentra en las coordenadas de pantalla $sy = 250$, $sx = 350$, convierta los siguientes puntos cartesianos en la forma (X,Y) a coordenadas de pantalla (sx, sy):

- a. (0, 100)
- b. (25, 25)
- c. (-200, -150)

15.5 Gráficos de mapas de memoria

Ya hemos visto cómo el proceso de dibujar píxeles mediante INT 10h es demasiado lento, excepto para la salida de gráficos más rudimentaria. Se ejecuta una cantidad considerable de código cada vez que el BIOS dibuja un píxel. Ahora podemos mostrarle 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.

15.5.1 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 una manera similar. El último byte en el arreglo se asigna al píxel en la esquina inferior derecha de la pantalla. ¿Por qué utilizar todo un byte para cada píxel? Porque el byte contiene una referencia a uno de 256 valores distintos de color.

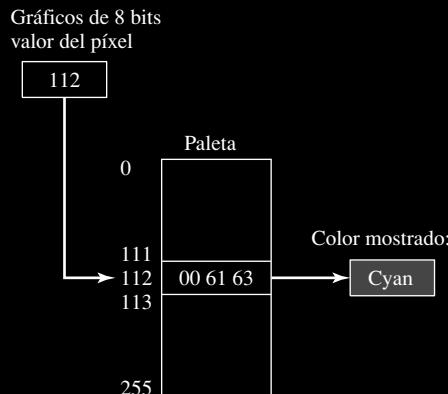
Instrucción OUT Los valores de píxel y de color se transmiten al hardware del adaptador de video mediante el uso de la instrucción OUT (salida a puerto). La dirección de puerto de 16 bits se asigna a DX; y el valor que se envía al puerto se encuentra en AL, AX o EAX. Por ejemplo, la paleta de color de video se encuentra en la dirección de puerto 3C8h. Las siguientes instrucciones envían el valor 20h al puerto:

<code>mov dx, 3c8h</code>	<code>; dirección de puerto</code>
<code>mov al, 20h</code>	<code>; valor a enviar</code>
<code>out dx, al</code>	<code>; envía el valor al puerto</code>

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* (figura 15-4). 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) color 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.

FIGURA 15-4 Conversión de índices de color de píxel a colores de pantalla.



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 (el máximo). De hecho, y como lo demuestra 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 crean estableciendo todos menos un nivel de color a cero. Para obtener un color claro, se incrementan los otros dos colores en cantidades iguales. He aquí algunas variaciones 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 y verde claro se crean de una manera similar. Desde luego que 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	lavanda

15.5.2 Programa de gráficos de mapas de memoria

El programa *Gráficos de mapas de memoria* que presentamos a continuación dibuja una fila de 10 píxeles en la pantalla, usando la asignación directa de memoria en el modo 13h. El procedimiento principal llama a procedimientos que establecen el modo de video al modo 13h, establecen el color de fondo de la pantalla, dibujan algunos píxeles de color, y restauran el adaptador de video a su modo inicial. Los dos puertos de salida controlan la paleta de colores de video. El valor que se envía al puerto 3C8h indica qué entrada de la paleta de video se planea modificar. Después, los valores de los colores se envían al puerto 3C9h. He aquí el listado del programa:

```
; Gráficos de mapas de memoria, Modo 13           (Modo13.asm)
INCLUDE Irvine16.inc
PUERTO_PALETA_VIDEO = 3C8h
PUERTO_SELECCION_COLOR = 3C9h
```

```
INDICE_COLOR = 1
FONDO_INDICE_PALETA = 0
ESTABLECER_MODO_VIDEO = 0
OBTENER_MODO_VIDEO = 0Fh
SEGMENTO_VIDEO = 0A000h
ESPERAR_TECLEO = 10h
MODO_13 = 13h

.data
guardaModo BYTE ? ; modo de video guardado
valX WORD ? ; coordenada x
valY WORD ? ; coordenada y
msj BYTE "Bienvenido al Modo 13!",0

.code
main PROC
    mov ax,@data
    mov ds,ax
    call EstablecerModoVideo
    call EstablecerFondoPantalla

    ; Muestra un mensaje de bienvenida.
    mov edx,OFFSET msj
    call WriteString
    call Dibujar_pixeles
    call RestaurarModoVideo
    exit
main ENDP

;-----
EstablecerFondoPantalla PROC
;
; Establece el color de fondo de la pantalla. El indice
; de paleta de video 0 es el color de fondo.
;
    mov dx,PUERTO_PALETA_VIDEO
    mov al,FONDO_INDICE_PALETA
    out dx,al

; Establece el color de fondo de la pantalla a azul oscuro.

    mov dx,PUERTO_SELECCION_COLOR
    mov al,0 ; rojo
    out dx,al
    mov al,0 ; verde
    out dx,al
    mov al,35 ; azul (intensidad 35/63)
    out dx,al

    ret
EstablecerFondoPantalla endp

;-----
EstablecerModoVideo PROC
;
; Guarda el modo de video actual, cambia a un
; nuevo modo y apunta a ES al segmento de video.
;
```

```
    mov ah,OBTENER_MODO_VIDEO
    int 10h
    mov guardaModo,al           ; lo guarda
    mov ah,ESTABLECER_MODO_VIDEO
    mov al,MODO_13               ; al modo 13h
    int 10h
    push SEGMENTO_VIDEO          ; dirección del segmento de video
    pop es                        ; ES apunta al segmento de video
    ret
EstablecerModoVideo ENDP

;-----
RestaurarModoVideo PROC
;
; Espera a que se oprima una tecla y restaura
; el modo de video a su valor original.
;-----
    mov ah,ESPERAR_TECLEO
    int 16h
    mov ah,ESTABLECER_MODO_VIDEO ; restablece el modo de video
    mov al,guardaModo            ; al modo guardado
    int 10h
    ret
RestaurarModoVideo ENDP

;-----
Dibujar_pixeles PROC
;
; Establece los colores individuales de la paleta y dibuja
; varios píxeles.
;-----
; Cambia el color en el índice 1 a blanco (63,63,63).
    mov dx,PUERTO_PALETA_VIDEO
    mov al,1                     ; establece el índice de paleta 1
    out dx,al
    mov dx,PUERTO_SELECCION_COLOR
    mov al,63                    ; rojo
    out dx,al
    mov al,63                    ; verde
    out dx,al
    mov al,63                    ; azul
    out dx,al
; Calcula el desplazamiento del búfer de video del primer píxel.
; El método es específico para el modo 13h, que es de 320 X 200.
    mov valX,160                 ; mitad de la pantalla
    mov valY,100
    mov ax,320                   ; 320 para el modo de video 13h
    mul valY                      ; coordenada y
    add ax,valX                  ; coordenada x
; Coloca el índice de color en el búfer de video.
    mov cx,10                     ; dibuja 10 píxeles
    mov di,ax                      ; AX contiene el desplazamiento del búfer
```

; Dibuja ahora los píxeles. De manera predeterminada, el ensamblador supone
; que DI es un desplazamiento de la dirección de segmento en DS. La
; redefinición de segmento ES:[DI] indica a la CPU que debe usar mejor la
; dirección de segmento en ES. ES apunta actualmente a la VRAM.

DP1:

```
    mov    BYTE PTR es:[di],INDICE_COLOR
    add    di,5           ; se mueve 5 píxeles a la derecha
    loop   DP1
    ret
Dibujar_píxeles ENDP
END main
```

Este programa es bastante fácil de implementar, ya que los píxeles se encuentran en la misma línea de la pantalla. Por otro lado, para dibujar una línea vertical podríamos sumar 320 a cada valor de DI, para avanzar a la siguiente fila de píxeles. O podríamos dibujar una línea diagonal con pendiente de -1 , sumando 321 a DI. El proceso de dibujar líneas arbitrarias entre dos puntos cualesquiera se maneja mejor mediante el *Algoritmo de Bresenham*, el cual se explica con detalle en muchos sitios Web.

15.5.3 Repaso de sección

- (Verdadero/Falso): el modo de video 13h asigna los píxeles de la pantalla como un arreglo bidimensional de bytes, en donde cada byte corresponde a dos píxeles.
- (Verdadero/Falso): en el modo de video 13h, cada fila de la pantalla utiliza 320 bytes de almacenamiento.
- Explique en una oración cómo el modo de video 13h establece los colores de los píxeles.
- ¿Cómo se utiliza el índice de colores en el modo de video 13h?
- En el modo de video 13h, ¿qué contiene cada elemento de la paleta de colores?
- ¿Cuáles son los tres valores RGB para el color gris oscuro?
- ¿Cuáles son los tres valores RGB para el color blanco?
- ¿Cuáles son los tres valores RGB para el color rojo brillante?
- Reto*: muestre cómo establecer el color de fondo de la pantalla en el modo de video 13h a verde.
- Reto*: muestre cómo establecer el color de fondo de la pantalla en el modo de video 13h a blanco.

15.6 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 dispositivos. 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* (¿cómo cree que idearon este nombre?). Un mickey representa aproximadamente 1/200 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.³ También hay un umbral de doble velocidad, cuyo valor predeterminado es de 64 mickeys por segundo.

15.6.1 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 recibe el número de función en el registro AX, en vez de AH (que es la norma para las interrupciones 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 muestran 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 = número de botones del ratón; en caso contrario, AX = 0
Llamada de ejemplo	<pre>mov ax,0 int 33h cmp ax,0 je RatonNoDisponible mov numeroDeBotones,bx</pre>
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, muestran y ocultan el puntero 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 del 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	<pre>mov ax,1 int 33h</pre>
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	Oculta el puntero del ratón
Recibe	AX = 2
Devuelve	Nada
Llamada de ejemplo	<pre>mov ax,2 int 33h</pre>
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 3 de 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	<pre>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, cx</pre>
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 convertir las coordenadas de píxel 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:

$$C = \text{int}(P/M)$$

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 de carácter 12: $C = \text{int}(100/8)$.

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	<pre>mov ax, 4 mov cx, 200 ; posición X mov dx, 100 ; posición Y int 33h</pre>
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 píxel 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:

$$P = C \times M$$

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 desea colocar el ratón en la celda del carácter 12, la coordenada X del píxel que se encuentra más a la izquierda de esa 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 un 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 botón 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 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	<pre>mov ax,5 mov bx,0 ; ID del botón int 33h test ax,1 ; ¿se oprimió el botón izquierdo? jz saltar ; no - salta mov coord_X,cx ; sí - guarda las coordenadas mov coord_Y,dx</pre>
Notas	El estado del botón 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 la programación controlada por eventos, un evento de *clic* 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 información acerca de si se soltó un botón
Recibe	AX = 6 BX = ID del botón (0 = izquierdo, 1 = derecho, 2 = central)
Devuelve	AX = estado del botón BX = contador de veces que se soltó el botón CX = coordenada X del último botón que se soltó DX = coordenada Y del último botón que se soltó

Función 6 de INT 33h	
Llamada de ejemplo	<pre>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 ; sí - guarda las coordenadas mov coord_Y,dx</pre>
Notas	El estado del botón del ratón 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 máxima (en píxeles)
Devuelve	Nada
Llamada de ejemplo	<pre>mov ax,7 mov cx,100 ; establece el rango de X a mov dx,700 ; (100,700) int 33h</pre>

Función 8 de INT 33h	
Descripción	Establece los límites verticales
Recibe	AX = 8 CX = coordenada Y mínima (en píxeles) DX = coordenada Y máxima (en píxeles)
Devuelve	Nada
Llamada de ejemplo	<pre>mov ax,8 int 33h mov cx,100 ; establece el rango Y a mov dx,500 ; (100,500) int 33h</pre>

Funciones varias del ratón

Hay otras funciones de INT 33h que son útiles para configurar el ratón y controlar su comportamiento. No hablaremos en detalle sobre estas funciones, pero se presentan en la tabla 15-9.

Tabla 15-9 Varias funciones del ratón.

Función	Descripción	Parámetros 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 área de exclusión del ratón (evita que el ratón entre en un rectángulo)	Recibe: CX, DX = coordenadas X, Y de la esquina superior 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	Recibe: BX = velocidad horizontal (mickeys por segundo), CX = velocidad vertical (mickeys por segundo), DX = umbral de doble velocidad en mickeys por segundo
AX = IBh	Obtiene 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 error; 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)

15.6.2 Programa para rastrear el ratón

Hemos escrito un programa simple para *rastrear el ratón*, el cual rastrea el movimiento del cursor del ratón de texto. Las coordenadas X y Y se actualizan en forma continua en la esquina inferior derecha de la pantalla, y cuando el usuario oprime el botón izquierdo, se muestra la posición del ratón en la esquina inferior izquierda de la pantalla. He aquí el código fuente:

```

TITLE Rastrear el ratón (Raton.asm)
; Demuestra las funciones básicas de ratón, disponibles a través de INT 33h.
; En el modo DOS estándar, cada posición de carácter en la ventana de
; DOS equivale a 8 unidades de ratón.

INCLUDE Irvine16.inc

OBTENER_ESTADO_RATON = 0
MOSTRAR_PUNTERO_RATON = 1
OCULTAR_PUNTERO_RATON = 2
OBTENER_TAMANIO_CURSOR = 3
OBTENER_INFO_OPRIMIR_BOTON = 5
OBTENER_POSICION_Y_ESTADO_RATON = 3
teclaESC = 1Bh

.data
saludo    BYTE “[Mouse.exe] Oprima Esc para salir”,0
lineaEstado BYTE “Boton izquierdo: “
            BYTE “Posicion del raton: “,0
espacios  BYTE “”,”,0
coordX WORD 0 ; posición X actual
coordY WORD 0 ; posición Y actual

```

```

oprimX WORD 0 ; pos X del último botón oprimido
oprimY WORD 0 ; pos Y del último botón oprimido

; Muestra las coordenadas.
filaEstado BYTE ?
colEstado BYTE 15
colBotonOprim BYTE 20
colEstado2 BYTE 60
colCoord BYTE 65

.code
main PROC
    mov ax,@data
    mov ds,ax
    call Clrscr

; Obtiene las coordenadas X/Y de la pantalla.
    call GetMaxXY ; DH = filas, DL = columnas
    dec dh ; calcula el valor de la fila de estado
    mov filaEstado,dh

; Oculta el cursor de texto y muestra el ratón.
    call OcultarCursor
    mov dx,OFFSET saludo
    call WriteString
    call MostrarPunteroRaton

; Muestra la información de estado en la línea inferior de la pantalla.
    mov dh,filaEstado
    mov dl,0
    call Gotoxy
    mov dx,OFFSET lineaEstado
    call Writestring

; Ciclo: muestra las coordenadas del ratón, comprueba si se oprimió el
; botón izquierdo o una tecla (Esc).

L1: call MostrarPosicionRaton
    call OprimioBotonIzq ; comprueba si se oprimió el botón
    mov ah,11h ; ¿ya se oprimió una tecla?
    int 16h
    jz L2 ; no, continúa el ciclo
    mov ah,10h ; elimina tecla del búfer
    int 16h
    cmp al,teclaESC ; sí. ¿Es la tecla ESC?
    je salir ; sí, sale del programa
L2: jmp L1 ; no, continúa el ciclo

; Oculta el ratón, restaura el cursor de texto, borra
; la pantalla y espera un tecleo.
salir:
    call OcultarPunteroRaton
    call MostrarCursor
    call Clrscr
    call WaitMsg
    exit
main ENDP

;-----
;-----ObtenerPosicionRaton PROC USES ax
;

```

```

; Obtiene la posición actual del ratón y el estado de los botones.
; Recibe: nada
; Devuelve: BX = estado de los botones (0 = se oprimió botón izq,
;           (1 = se oprimió botón der, 2 = se oprimió botón centr)
;           CX = coordenada X
;           DX = coordenada Y
;-----.
    mov  ax,OBTENER_POSICION_Y_ESTADO_RATON
    int  33h
    ret
ObtenerPosicionRaton ENDP

;-----.
OcultarCursor PROC USES ax cx
;
; Oculta el cursor de texto estableciendo el valor de su
; línea superior a un valor ilegal.
; Recibe: nada. Devuelve: nada
;-----.
    mov  ah,OBTENER_TAMANIO_CURSOR
    int  10h
    or   ch,30h                 ; establece fila superior al val ilegal
    mov  ah,1                   ; establece el tamaño del cursor
    int  10h
    ret
OcultarCursor ENDP

;-----.
MostrarCursor PROC USES ax cx
;
; Muestra el cursor de texto, estableciendo el tamaño al predeterminado.
; Recibe: nada. Devuelve: nada
;-----.
    mov  ah,OBTENER_TAMANIO_CURSOR
    int  10h
    mov  ah,1                   ; establece el tamaño del cursor
    mov  cx,0607h               ; tamaño predeterminado
    int  10h
    ret
MostrarCursor ENDP

;-----.
OcultarPunteroRaton PROC USES ax
;
; Oculta el puntero del ratón.
; Recibe: nada. Devuelve: nada
;-----.
    mov  ax,OCULTAR_PUNTERO_RATON
    int  33h
    ret
OcultarPunteroRaton ENDP

;-----.
MostrarPunteroRaton PROC USES ax
;
; Hace visible el puntero del ratón.
; Recibe: nada. Devuelve: nada
;-----.
    mov  ax,MOSTRAR_PUNTERO_RATON

```

```
int 33h
ret
MostrarPunteroRaton ENDP

;-----
OprimioBotonIzq PROC
;
; Comprueba la última vez que se oprimió el botón izquierdo
; del ratón y muestra su ubicación.
; Recibe: nada. Devuelve: nada
;
pusha
mov ax,OBTENER_INFO_OPRIMIR_BOTON
mov bx,0 ; especifica el botón izquierdo
int 33h

; Sale del procedimiento si las coordenadas no han cambiado.
cmp cx,oprimX ; ¿misma coordenada X?
jne L1 ; no: continúa
cmp dx,oprimY ; ¿misma coordenada Y?
je L2 ; yes: sale

; Las coordenadas cambiaron, por lo que deben guardarse.
L1: mov oprimX,cx
    mov oprimY,dx

; Posiciona el cursor, borra los números anteriores.
    mov dh,filaEstado ; fila de la pantalla
    mov dl,colEstado ; columna de la pantalla
    call Gotoxy
    push dx
    mov dx,OFFSET espacios
    call WriteString
    pop dx

; Muestra las coordenadas en donde se oprimió el botón del ratón.
    call Gotoxy
    mov ax,coordX
    call WriteDec
    mov dl,colBotonOprim
    call Gotoxy
    mov ax,coordY
    call WriteDec

L2: popa
    ret
OprimioBotonIzq ENDP

;-----
EstablecerPosicionRaton PROC
;
; Establece la posición del ratón en la pantalla.
; Recibe: CX = coordenada X
;           DX = coordenada Y
; Devuelve: nada
;
    mov ax,4
    int 33h
    ret
EstablecerPosicionRaton ENDP
```

```

;-----[-----]
MostrarPosicionRaton PROC
;
; Obtiene y muestra las coordenadas del ratón en
; la parte inferior de la pantalla.
; Recibe: nada
; Devuelve: nada
;-----[-----]

    pusha
    call ObtenerPosicionRaton

; Sale del procedimiento si no han cambiado las coordenadas.
    cmp cx,coordX           ; ¿misma coordenada X?
    jne L1                   ; no: continúa
    cmp dx,coordY           ; ¿misma coordenada Y?
    je L2                   ; sí: sale

; Guarda las nuevas coordenadas X y Y.
L1:   mov coordX,cx
      mov coordY,dx

; Posiciona el cursor, borra los números anteriores.
    mov dh,filaEstado        ; fila de la pantalla
    mov dl,colEstado2         ; columna de la pantalla
    call Gotoxy
    push dx
    mov dx,OFFSET espacios
    call WriteString
    pop dx

; Muestra las coordenadas del ratón.
    call Gotoxy
    mov ax,coordX
    call WriteDec
    mov dl,colCoord           ; columna de la pantalla
    call Gotoxy
    mov ax,coordY
    call WriteDec

L2:   popa
      ret
MostrarPosicionRaton ENDP
END main

```

Comportamientos variados El comportamiento del programa cambia un poco, dependiendo de dos factores: (1) qué versión de MS Windows esté ejecutando, y (2) si lo ejecuta en una ventana de consola o en el modo de pantalla completa. Por ejemplo, en Windows XP la ventana de consola tiene un valor predeterminado de 50 líneas de texto verticales. Al ejecutarlo en modo de pantalla completa, el cursor del ratón es un bloque sólido; sus coordenadas parecen cambiar un píxel a la vez, mientras que el ratón salta de un carácter al siguiente sólo cuando se desplaza 8 píxeles en sentido horizontal o 16 píxeles en sentido vertical. En el modo de ventana de consola, el cursor del ratón es un apuntador; sus coordenadas cambian 8 píxeles a la vez en sentido horizontal, y 16 píxeles a la vez en sentido vertical.

15.6.3 Repaso de sección

1. ¿Qué función de INT 33h restablece el ratón y obtiene su estado?
2. Escriba instrucciones de ASM para restablecer el ratón y obtener su estado.
3. ¿Qué función de INT 33h muestra y oculta el puntero del ratón?

4. Escriba instrucciones de ASM para ocultar el puntero del ratón.
5. ¿Qué función de INT 33h obtiene la posición y estado del ratón?
6. Escriba instrucciones de ASM para obtener la posición del ratón y almacenarla en las variables **ratonX** y **ratonY**.
7. ¿Qué función de INT 33h establece la posición del ratón?
8. Escriba instrucciones de ASM para establecer el puntero del ratón a X = 100 y Y = 400.
9. ¿Qué función de INT 33h obtiene la información acerca de si se oprimió uno de los botones del ratón?
10. Escriba instrucciones de ASM para saltar a la etiqueta **Boton1** cuando se oprima el botón izquierdo del ratón.
11. ¿Qué función de INT 33h obtiene información acerca de si se soltó uno de los botones del ratón?
12. Escriba instrucciones de ASM para obtener la posición del ratón en el punto en el que se soltó el botón derecho, y almacenar la posición en las variables **ratonX** y **ratonY**.
13. Escriba instrucciones de ASM para establecer los límites verticales del ratón a 200 y 400.
14. Escriba instrucciones de ASM para establecer los límites horizontales del ratón a 300 y 600.
15. *Reto:* suponga que desea que el puntero del ratón apunte a la esquina superior izquierda de la celda de caracteres ubicada en la fila 10, columna 20 en modo de texto. ¿Qué valores de X y Y tendrá que pasar a la función 4 de INT 33h, suponiendo 8 píxeles horizontales por carácter y 16 píxeles verticales por carácter?
16. *Reto:* suponga que desea que el puntero del ratón apunte a la mitad de la celda de caracteres ubicada en la fila 15, columna 22 en modo de texto. ¿Qué valores de X y Y tendrá que pasar a la función 4 de INT 33h, suponiendo 8 píxeles horizontales por carácter y 16 píxeles verticales por carácter?
17. *Reto:* ¿quién inventó el ratón de computadora, en qué año y en dónde?

15.7 Resumen del capítulo

Trabajar al nivel del BIOS le brinda más control sobre los dispositivos de entrada-salida del que puede tener en el nivel de MS-DOS. En este capítulo le mostramos cómo programar el teclado mediante INT 16h, la pantalla de video mediante INT 10h, y el ratón mediante INT 33h.

En especial, INT 16h es útil para leer teclas extendidas del teclado, como las teclas de función y las flechas del cursor.

El hardware del teclado trabaja con los manejadores INT 9h, INT 16h e INT 21h de manera que la entrada del teclado esté disponible para los programas. Este capítulo contiene un programa que sondea el teclado y termina un ciclo cuando se oprime Esc.

Los colores se producen en la pantalla de video usando la síntesis aditiva de los colores primarios. Los bits de color se asignan al byte de atributos de video.

Hay un amplio rango de funciones útiles de INT 10h que pueden controlar la pantalla de video a nivel del BIOS. Este capítulo contiene un programa de ejemplo para desplazar una ventana a color y escribir texto en medio de ella.

Puede dibujar gráficos a color mediante el uso de INT 10h. Este capítulo contiene dos programas de ejemplo que le muestran cómo hacer esto. Puede usar una fórmula simple para convertir las coordenadas cartesianas a coordenadas de la pantalla (ubicaciones de los píxeles).

Un programa de ejemplo con su documentación muestra cómo dibujar gráficos a color de alta velocidad, escribiendo directamente a la memoria de video.

Hay varias funciones de INT 33h para manipular y leer el ratón. Un programa de ejemplo rastrea tanto los movimientos del ratón como los clics de sus botones.

Para más información No es fácil encontrar información acerca de las funciones del BIOS, ya que muchos de los libros buenos de referencia se han dejado de imprimir. He aquí mis favoritos:

- Ralf, Brown y Jim, Kyle, *PC Interrupts. A Programmer's Reference to BIOS, DOS and Third-Party Calls*, Addison-Wesley, 1991.

- Ray, Duncan, *IBM ROM BIOS*, Microsoft Press, 1998.
- Ray, Duncan, *Advanced MS-DOS Programming*, 2a. edición, Microsoft Press, 1988.
- Frank van Gilluwe. *The Undocumented PC: A Programmer's Guide to I/O, CPUs, and Fixed Memory Areas*, Addison-Wesley, 1996.
- Thom, Hogan. *Programmer's PC Sourcebook: Reference Tables for IBM PCs and Compatibles, Ps/2 Systems, Eisa-Based Systems, Ms-DOS Operating System Through Version*, Microsoft Press, 1991.
- Jim, Kyle. *DOS 6 Developer's Guide*, SAMS, 1993.
- Muhammad Ali, Mazidi, and Janice Gillispie Mazidi, *The 80x86 IBM PC & Compatible Computers*, 4a. edición, Volúmenes I y II, Prentice-Hall, 2002.

En el sitio Web de este libro encontrará vínculos hacia muchas fuentes adicionales de información, incluyendo la lista actual de Ralf Brown de las interrupciones de MS-DOS y del BIOS.

15.8 Ejercicios del capítulo

Los siguientes ejercicios deben realizarse en el modo de direccionamiento real:

1. Tabla ASCII

Mediante el uso de INT 10h, muestre todos los 256 caracteres del conjunto de caracteres ASCII extendidos de IBM (dentro de la solapa del libro). Muestre 32 columnas por línea, con un espacio después de cada carácter.

2. Ventana de texto desplazable

Defina una ventana de texto que tenga un tamaño aproximado a tres cuartas partes de la pantalla de video. Deje que el programa lleve a cabo las siguientes acciones, en secuencia:

- Dibujar una cadena de caracteres aleatorios en la línea superior de la ventana (puede llamar al procedimiento Random_range de la biblioteca Irvine16).
- Desplazar la ventana una línea hacia abajo.
- Detener el programa durante un tiempo aproximado de 200 milisegundos (puede llamar a la función **Delay** de la biblioteca Irvine16).
- Dibujar otra línea de texto aleatorio.
- Continuar desplazándose y dibujando hasta que se hayan mostrado 50 líneas.

Este programa y sus diversas mejoras recibieron un sobrenombre de mis estudiantes de lenguaje ensamblador, de acuerdo con una popular película en la que los personajes interactúan en un mundo virtual. No puedo mencionar su nombre aquí, pero es probable que usted lo averigüe para cuando complete los programas.

3. Desplazamiento de columnas a color

Use el ejercicio **Ventana de texto desplazable** como punto de partida y realice las siguientes modificaciones:

- La cadena aleatoria sólo deberá tener caracteres en las columnas 0, 3, 6, 9, . . . , 78. Las otras columnas deberán estar en blanco. Esto creará el efecto de columnas a medida que avanza hacia abajo.
- Cada columna debe tener un color distinto.

4. Desplazamiento de columnas en distintas direcciones

Use el ejercicio **Ventana de texto desplazable** como punto de partida y realice la siguiente modificación: Antes de que empiece el ciclo, elija al azar cada columna para que se desplace hacia arriba o hacia abajo. Deberá continuar en la misma dirección durante el tiempo que se ejecute el programa. *Sugerencia:* defina cada columna como una ventana desplazable separada.

5. Dibujo de un rectángulo mediante el uso de INT 10h

Use las herramientas para dibujar píxeles de INT 10h y cree un procedimiento llamado **DibujarRectangulo**, que reciba parámetros de entrada que especifiquen la ubicación de la esquina superior izquierda y la esquina inferior derecha, y el color. Escriba un programa de prueba corto para dibujar varios rectángulos de distintos tamaños y colores.

6. Trazo de una función mediante el uso de INT 10h

Use las herramientas para dibujar píxeles de INT 10h y trace la línea determinada por la ecuación $Y = 2(X^2)$.

7. Línea en el Modo 13

Modifique el programa de Gráficos de mapas de memoria en la sección 15.5.2, de manera que dibuje una sola línea vertical.

8. Modo 13, múltiples líneas

Modifique el programa de Gráficos de mapas de memoria en la sección 15.5.2, de manera que dibuje una serie de 10 líneas verticales, cada una de ellas en un color distinto.

9. Programa para dibujar cuadros

Las aplicaciones de MS-DOS en la década de 1980 y a principios de 1990 mostraban, por lo general, cuadros y marcos, usando los caracteres para dibujo de líneas en modo de texto. Este ejercicio de programación reproducirá esas técnicas. Escriba un procedimiento para dibujar un marco de una sola línea en cualquier parte de la pantalla. Use los siguientes códigos ASCII extendidos de la tabla que se encuentra en la solapa de este libro: C0h, BFh, B3h, C4h, D9h y DAh. El único parámetro de entrada del procedimiento debe ser un apuntador a una estructura MARCO:

```
MARCO STRUCT
    Izquierdo BYTE ?           ; lado izquierdo
    Superior BYTE ?           ; linea superior
    Derecho BYTE ?            ; lado derecho
    Inferior BYTE ?           ; linea inferior
    ColorMarco BYTE ?         ; color del cuadro
MARCO ENDS
```

Escriba un programa para probar su procedimiento, que reciba apuntadores a diversos objetos MARCO.

Notas finales

1. Un excelente ejemplo es la obra de Michael Abrash, *The Zen of Code Optimization*, Coriolis Group Books, 1994.
2. Tal vez tenga problemas al ejecutar *Pixel1.asm* y *Pixel2.asm* en MS Windows, en computadoras que tengan una cantidad relativamente baja de RAM de video. Si esto representa un problema, cambie a otro modo o inicie en modo MS-DOS puro.
3. De Ray Duncan, *Advanced MS-DOS Programming*, 2^a edición, Microsoft Press, 1988, p. 601.

PROGRAMACIÓN EXPERTA EN MS-DOS

16.1	Introducción	
16.2	Definición de segmentos	
16.2.1	Directivas de segmento simplificadas	
16.2.2	Definiciones explícitas de segmentos	
16.2.3	Redefiniciones de segmentos	
16.2.4	Combinación de segmentos	
16.2.5	Repasso de sección	
16.3	Estructura de un programa en tiempo de ejecución	
16.3.1	Prefijo de segmento del programa	
16.3.2	Programas COM	
16.3.3	Programas EXE	
16.3.4	Repasso de sección	
16.4	Manejo de interrupciones	
16.4.1	Interrupciones de hardware	
16.4.2	Instrucciones de control de interrupciones	
16.4.3	Escritura de un manejador de interrupciones personalizado	
16.4.4	Programas TSR (Terminar y permanecer residente)	
16.4.5	Aplicación: El programa No_reinicio	
16.4.6	Repasso de sección	
16.5	Control de hardware mediante el uso de puertos de E/S	
16.5.1	Puertos de entrada-salida	
16.5.2	Programa de sonido de PC	
16.6	Resumen del capítulo	

16.1 Introducción

Este capítulo le será de utilidad si planea ser un ingeniero que trabaje al nivel de hardware en los procesadores Intel. Le ayudará también si desea conocer las sorprendentes cosas que los expertos de MS-DOS podrían hacer con recursos muy limitados hace unos cuantos años. Obtendrá cimientos útiles si planea convertirse en un programador a nivel del sistema. Es un capítulo acerca de los recursos y la programación del sistema MS-DOS. He aquí lo que veremos:

- Le mostraremos cómo obtener la mayor flexibilidad de las directivas .MODEL, .CODE, .STACK y relacionadas.
- Le mostraremos cómo definir segmentos a partir de cero, mediante las directivas explícitas de segmento.
- Demostraremos un programa en el modelo extenso de memoria, que tiene varios segmentos de código y de datos.
- Explicaremos la estructura en tiempo de ejecución de los programas COM y EXE, incluyendo los encabezados EXE.
- Elaboraremos un mapa del Prefijo de segmento de programa (PSP) y le mostraremos cómo encontrar la cadena del entorno de MS-DOS.

- Le enseñaremos cómo sustituir los manejadores de interrupciones existentes con sus propios manejadores. Demostraremos esto mediante la escritura de un manejador de interrupciones Ctrl-Inter (también conocido como *rutina de servicio de interrupción*, o ISR).
- Explicaremos cómo funcionan las interrupciones de hardware y presentaremos los diversos niveles de *petición de interrupción* (IRQ) utilizados por el Controlador de interrupciones programable (PIC) Intel 8259.
- Escribiremos un programa TSR (*terminar y permanecer residente*) para interceptar la combinación de teclas Ctrl-Alt-Supr. Si aprende a hacer esto, podrá unirse a la categoría de los expertos de MS-DOS.
- Le mostraremos cómo escribir datos de hardware directamente a los puertos de salida, y cómo utilizar los puertos para monitorear el estado del hardware, controlar su comportamiento y leer datos de entrada de los dispositivos de hardware.

Si ha estado cerca de programadores experimentados durante algunos años, es probable que haya escuchado muchos de los términos de la siguiente lista. ¿Ha observado cómo los expertos de los viejos tiempos utilizan términos como IRQ, TSR, PSP y 8259 en sus conversaciones? Ahora podrá saber de lo que hablan.

16.2 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), pues hicieron que la primera semana de clases transcurriera con mucha más facilidad. Sin embargo, también quedó claro que los programadores expertos tal vez preferirían la flexibilidad en vez de la simplicidad, y se apegarían a la forma tradicional de hacer las cosas. Si llegó a este capítulo (y comprendió todos los capítulos anteriores), ahora está listo 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 de que satisfagan sus necesidades.

16.2.1 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 puede acceder directamente a través de DS o SS.

Las directivas .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 también está limitado a 64Kb. Cuando .FARDATA y .FARDATA? se utilizan en los modelos pequeño 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. Puede 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 alinea 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 nombre 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
```

Recuerde que si llama a los procedimientos de la biblioteca de vínculos de 16 bits del libro, su 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 msg
    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 mostrarle 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
msg1 db "Primer mensaje",0dh,0ah,0
msg2 db "Segundo mensaje",0dh,0ah,"$"

.code
main PROC
    mov ax,@data
    mov ds,ax
    mov dx,OFFSET msg1           ; llamada NEAR
    call WriteString
    call Mostrar                 ; llamada FAR
    .exit
main ENDP

.code OtroCodigo
Mostrar PROC FAR
    mov ah,9
    mov dx,offset msg2
    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**. Observe que este procedimiento debe tener un modificador FAR para indicar al ensamblador que debe generar el 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'

16.2.2 Definiciones explícitas de segmentos

Hay ocasiones en las que tal vez sea preferible crear definiciones explícitas de segmentos. Por ejemplo, tal vez quiera definir varios segmentos de datos con búferes de memoria adicionales. O tal vez vaya a enlazar su programa con una biblioteca de objetos que utilice sus propios nombres de segmentos propietarios. Por último, tal vez esté escribiendo un procedimiento que deba llamarse desde un compilador de un lenguaje de alto nivel, que no utilice los nombres de segmento 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 pueda utilizarse. 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 único 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 cómo 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 predeterminado 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. Observe 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 los 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 procesador 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 en un 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, el 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 que 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 empiece en la misma dirección que cualquier otro segmento COMMON con el mismo nombre. En realidad, los segmentos se traslanan 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 de segmento (DS), ya que **bandera_teclado** no se encuentra en el segmento de datos estándar. En la sección 16.2.3 explicaremos las redefiniciones de segmentos.

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 sensible a mayúsculas y minúsculas, encerrada entre comillas sencillas. Los segmentos con el mismo tipo de clase se cargan juntos, aunque pueden estar en un 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. Debe incluir la etiqueta de este tipo si planea utilizar 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 del nombre de un segmento:

```
ASSUME regseg : nombreseg
```

ASSUME en realidad no modifica el valor de un registro de segmento. Eso debe hacerse en tiempo de ejecución, usando instrucciones para asignar valores de segmento a los registros de segmento. Su 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 miCódigo y SS se asocia con miPila:

```
ASSUME cs:miCódigo, 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 le 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 data1
    mov bx,val2             ; segmento dato2

    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 utilizaron 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 mismos valores (desplazamientos iniciales), pero distintos atributos de segmento:

Name	Type	Value	Attr
val1	Word	0000	datos1
val2	Word	0000	datos2

16.2.3 Redefiniciones de segmentos

Una *redefinición de segmento* 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, puede usarse para acceder a una variable en 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 el 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 dx,ax
mov al,var1
.

.

ASSUME ds:data            ; usa el segmento de datos predeterminado
mov ax,data
mov ds,ax
```

16.2.4 Combinación de segmentos

Los programas extensos deben dividirse en módulos separados para simplificar su edición y depuración. Incluso hasta el código fuente ubicado en distintos módulos puede combinarse en el mismo segmento. Sólo hay que usar el mismo nombre 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 del libro, usando directivas de segmento simplificadas.

Si utiliza un tipo de alineación BYTE, cada segmento va justo después del anterior. Si se utiliza 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 (SEGД), y un segmento de pila (SEGP). El módulo principal (main) contiene los tres segmentos; SEGC y SEGД tienen un tipo de combinación PUBLIC. Se utiliza un tipo de alineación BYTE para SEGC, con lo que se 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
    var1 WORD 1000h
segd ends

segp SEGMENT STACK 'STACK'          ; segmento de pila
    BYTE 100h dup('S')
segp ENDS
END main

```

Submódulo:

```

TITLE Ejemplo de segmentos      (submódulo, SEG2A.ASM)

PUBLIC subrutina_1, var2

segc SEGMENT BYTE PUBLIC 'CODE'
ASSUME cs:segc, ds:segd

subrutina_1 PROC             ; se llama desde MAIN
    mov  ah,9
    mov  dx,OFFSET msj
    int  21h
    ret
subrutina_1 ENDP
segc ENDS

segd SEGMENT WORD PUBLIC 'DATA'

var2 WORD  2000h            ; se accesa desde MAIN
msj  BYTE   'Ahora esta en Subrutina_1'
    BYTE   0Dh,0Ah,'$'

segd ENDS
END

```

El enlazador creó el siguiente archivo MAP, en el cual se muestra un segmento de código, un segmento de datos y un segmento de pila:

Start	Stop	Length	Name	Class
00000H	0001BH	0001CH	CSEG	Code
0001CH	00035H	0001AH	DSEG	DATA
00040H	0013FH	00100H	SSEG	STACK

Program entry point at 0000:0000

16.2.5 Repaso de sección

1. ¿Cuál es el propósito de las directivas SEGMENT y ENDS?
2. ¿Qué valor devuelve el operador SEG?
3. Explique la función de la directiva ASSUME.
4. En una definición de segmento, ¿cuáles son los posibles tipos de alineación?
5. En una definición de segmento, ¿cuáles son los posibles tipos de combinación?
6. ¿Qué tipo de alineación es más eficiente para un procesador IA-32?

7. ¿Cuál es el propósito del tipo de combinación en una definición de segmento?
8. ¿Cómo se define un segmento en una dirección absoluta, como 40h?
9. ¿Cuál es el propósito de la opción de tipo de clase en una definición de segmento?
10. Escriba una instrucción que utilice una redefinición de segmento.
11. En el siguiente ejemplo, suponga que segA empieza en la dirección 1A060h. ¿Cuál será la dirección inicial del *tercer* segmento, también llamado segA?

```

segA SEGMENT COMMON
    var1 WORD ?
    var2 BYTE ?
segA ends

pila SEGMENT STACK
    BYTE 100h dup(0)
pila ends

segA SEGMENT COMMON
    var3 WORD 3000h
    var4 BYTE 40h
segA ends

```

16.3 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 programa *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 de BAT se le conoce como archivo de procedimiento por lotes, que es un archivo de texto que contiene comandos de MS-DOS que deben ejecutarse como si se hubieran 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 agote 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 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.

16.3.1 Prefijo de segmento de programa

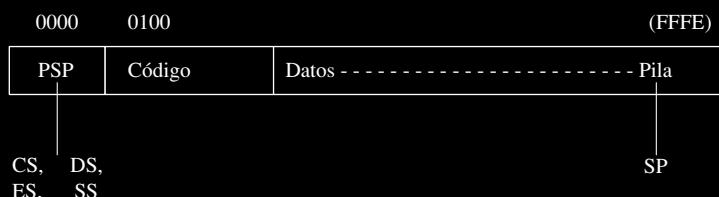
MS-DOS crea un bloque especial de 256 bytes al principio de un programa, a medida que se carga en memoria; a este bloque se le conoce como el *Prefijo de segmento del programa*. La estructura del Prefijo de segmento del programa (PSP) se muestra en la tabla 16-1.

Tabla 16-1 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

16.3.2 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 las ubicaciones desde 0 hasta 0FFh:

```

TITLE Programa Hola en formato COM  (HolaCom.asm)

.model tiny
.code
org 100h
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 formato 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,'$'

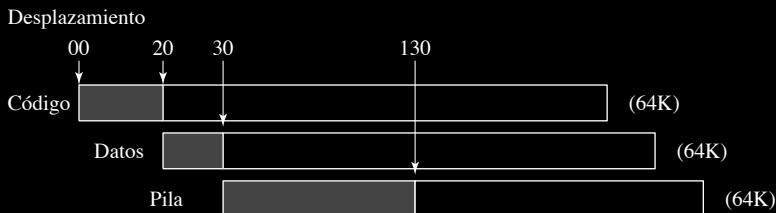
inicio:
    mov ah,9
    mov dx,OFFSET mensaje_hola
    int 21h
    mov ax,4C00h
    int 21h
main ENDP
END main
```

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 necesite el espacio o no. Los programas COM no se diseñaron para ejecutarse en un entorno multitarea.

16.3.3 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 dirección 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 con el 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 traslanan:



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 100h 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 (de 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 programa **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 llamado *prog1.exe* es:

```
exehdr prog1 /max 400
```

Exehdr puede mostrar importantes estadísticas acerca de un programa. A continuación se muestran 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 de 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 contenga 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 la memoria.

16.3.4 Repaso de sección

1. Cuando se escribe un comando en el símbolo de MS-DOS, ¿qué ocurre si el comando no es un comando interno de MS-DOS?
2. Cuando MS-DOS ejecuta un comando, ¿busca los archivos BAT antes de los archivos EXE en el directorio actual?
3. ¿Qué son los programas transientes?
4. ¿Cuál es el nombre del área de 256 bytes al inicio de un programa transiente?

5. ¿En dónde guarda un programa transiente la dirección de segmento de la cadena de entorno actual?
6. ¿Qué es un programa COM?
7. ¿Qué modelo(s) de memoria utilizan los programas COM?
8. ¿Qué modificador de línea de comandos del enlazador se requiere al crear un programa COM?
9. ¿Cuál es la limitación de memoria de un programa COM?
10. Cuando se ejecuta un programa COM, ¿qué tan eficiente es su uso de la memoria?
11. ¿Cuántos segmentos de programa puede contener un programa COM?
12. ¿Cuáles son los valores iniciales de todos los registros de segmento en un programa COM?
13. ¿Cuál es el propósito de la directiva ORG?
14. Cuando se almacena en disco, las dos partes principales de un programa EXE son el *encabezado* y el módulo _____.
15. ¿Hacia dónde apuntan DS y ES cuando se carga un programa EXE?
16. ¿Qué es lo que determina la cantidad de memoria que se asigna a un programa EXE?
17. ¿Cuál es el propósito del programa **exehdr**?
18. Si quisiera conocer el número de entradas de reubicación en un archivo EXE, ¿en dónde buscaría?

16.4 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 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 su computadora se inicia en modo de MS-DOS. Puede hacer esto usando Windows 95 y 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 OS permitiera que dos programas que se ejecutan al mismo tiempo modificaran las configuraciones internas en el mismo dispositivo de hardware, los resultados serían cuando menos impredecibles.

Existen varias razones para escribir un manejador de interrupciones. Tal vez quiera que su programa 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 especial de teclas activas.

Podemos sustituir uno de los manejadores de interrupciones predeterminados de MS-DOS para ofrecer 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 estándar para que un programa se recupere.

Puede sustituir el manejador de errores críticos de MS-DOS o el manejador de Ctrl-Inter con su propio manejador. El manejador de errores críticos predeterminado de MS-DOS hace que un programa aborte y regrese a MS-DOS. Su propio 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 tareas.

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 tabla 16-2 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.

Tabla 16-2 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 le 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.

16.4.1 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 los almacena en un búfer.

En ocasiones, los programas deben deshabilitar las interrupciones de hardware al realizar operaciones delicadas en los registros de segmento y la 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 distintos en una PC, incluyendo los que se presentan en la tabla 16-3. 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 la más baja. Una interrupción de un nivel más bajo no puede interrumpir a una de un nivel más alto que se encuentre en progreso. Por ejemplo, si el puerto de comunicaciones 1 (COM1) tratara de interrumpir el manejador de interrupciones del teclado, tendría que esperar hasta que este último terminara. Además, dos o más peticiones de interrupción simultáneas se procesan de acuerdo con sus niveles de prioridad. La programación de interrupciones se maneja mediante el PIC 8259.

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 las interrupciones externas no se encuentran desabilitadas, la CPU hace lo siguiente, en secuencia:

1. Mete el registro Flags en la pila.
2. Borra la bandera 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.

Tabla 16-3 Asignaciones de IRQ (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)

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. Despues 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 al registro Flags de la pila. El control regresa al programa que se estaba ejecutando cuando ocurrió la interrupción.

16.4.2 Instrucciones de control de interrupciones

La CPU tiene una bandera llamada *bandera 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 despues 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 apropiada, 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, suponga que su código se interrumpió cuando estaba en el proceso de modificar los valores de SS y SP. Su registro SS podría apuntar a un nuevo segmento de pila, mientras que su apuntador de pila tendría que actualizarse:

```
mov ax,mipila ; restablece SS
    mov ss,ax
```

```
; ¡¡¡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 podrían 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.

16.4.3 Escritura de un manejador de interrupciones personalizado

Uno podría preguntarse para qué existe la tabla de vectores de interrupción. Desde luego que 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 vectores 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 interrupciones 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 los 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               ; llamada 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_rtn  ; 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 de INT9)

```

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 se esté ejecutando. Esto puede dejar al programa en un estado inestable, ya que 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-Break (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 oprime ESC para terminar."
        BYTE 0dh,0ah,0

.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 interrupciones
    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 = vector de 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 oprime Esc.

En algunos sistemas, tal vez tenga 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 haya estado en progreso cuando se oprimió Ctrl-Inter. En general, podemos llamar a cualquier interrupción de 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 000Eh, en el prefijo de segmento del programa.

16.4.4 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 utilería 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 programas 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

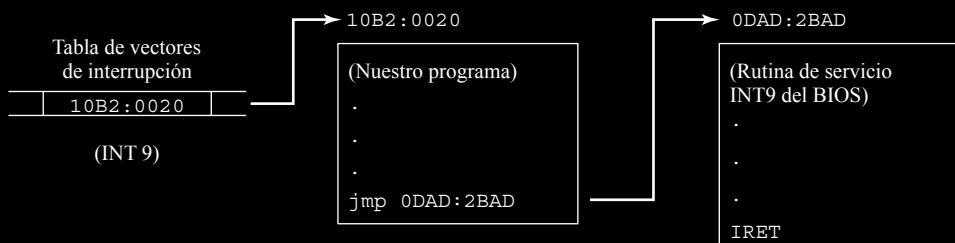
Suponga que escribimos una rutina de servicio de interrupciones que puede inspeccionar cada carácter que se escribe en el teclado y que puede almacenarlo en la ubicación 10B2:0020. Para instalar la ISR, obtenemos el vector actual de INT 9 de la tabla de vectores de interrupción, lo guardamos y sustituimos la entrada en la tabla con la dirección de nuestra ISR.

Cuando se oprime una tecla, se transfiere un byte individual al controlador del teclado hacia el puerto de teclado de la computadora, y se activa una interrupción de hardware. El PIC 8259 pasa el número de interrupción a la CPU y 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 encadenamiento se muestra en la figura 16-1. 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.

FIGURA 16-1 Vectorización de una interrupción.

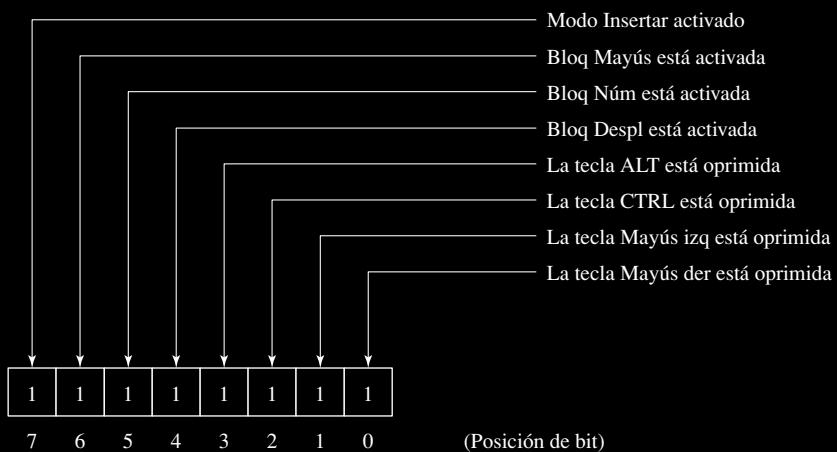


16.4.5 Aplicación: el programa No_reinicio

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 sólo funciona si 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 antes de reempezar es la ubicación del byte de estado del teclado que mantiene MS-DOS en la memoria inferior, y se muestra en la figura 16-2. 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 significa cada bit, cuando es igual a 1.

FIGURA 16-2 Byte de la bandera de estado del teclado.



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 activas 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 teclado se filtra 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 protegido. 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: debe iniciar la computadora en modo de 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 **manejador_int9**, se deja en memoria y es a 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. Ensamble el código, enlácelo
; y conviértalo en un programa COM incluyendo el comando /T
; en la línea de comandos de Microsoft LINK.
; Reinicie 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          ; sí: permite que el sistema se reinicie

L4:  and  ah,NOT tecla_ctrl    ; no: apaga el bit para CTRL
      mov  es:[di],ah      ; almacena bandera_tec

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,2509h          ; establece el vector de interrupción, INT 9
        mov  dx,offset manejador_int9
        int  21h

        mov  ax,3100h          ; 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, al siguiente párrafo
        int  21h                ; ejecuta la función de MS-DOS

END inicio

```

Primero vamos a ver 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, que 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...)

```

Tenga 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 comprueban 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_tec1 ; 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             ; sí: permite que el sistema se reinicie
```

Si el usuario no ha oprimido 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 bandera_tec1
```

Por último, ejecutamos un salto lejano a la rutina existente de BIOS INT 9h, almacenada en la variable **interrup9_ant**. Esto permite procesar todas las teclas presionadas normales, lo cual es vital para la operación básica de la computadora:

```
jmp cs:[interrup9_ant] ; salta a la rutina de INT 9
```

16.4.6 Repaso de sección

1. ¿Qué acción predeterminada lleva a cabo el *manejador de errores críticos*?
2. ¿Qué contiene cada una de las entradas de la tabla de vectores de interrupción?
3. ¿En qué dirección se almacena el vector de interrupciones para INT 10h?
4. ¿Qué chip controlador genera las interrupciones de hardware?
5. ¿Qué instrucción deshabilita las interrupciones de hardware?
6. ¿Qué instrucción habilita las interrupciones de hardware?
7. ¿Qué nivel de IRQ tiene la mayor prioridad, 0 o 15?
8. De acuerdo con lo que sabe acerca de los niveles de IRQ, si un programa se encuentra en el proceso de crear un archivo en disco y oprime una tecla, ¿cuándo cree que se colocará la tecla en el búfer del teclado, antes o después de haber creado el archivo?
9. Cuando se oprime una tecla, ¿qué interrupción de hardware se ejecuta?
10. Cuando termina un manejador de interrupciones, ¿cómo continúa la CPU la ejecución en donde se encontraba antes de activar la interrupción?
11. ¿Qué funciones de MS-DOS obtienen y establecen los vectores de interrupción?

12. Explique la diferencia entre un manejador de interrupciones y un programa residente en memoria.
13. Describa un programa TSR.
14. ¿Cómo puede eliminarse un programa TSR de la memoria?
15. Si un programa residente en memoria sustituye uno de los vectores de interrupción, ¿cómo puede seguir aprovechando algunas funciones en el manejador existente de la interrupción?
16. ¿Qué función de MS-DOS termina un programa y deja parte del programa residente en la memoria?
17. En el programa No_reinicio, ¿qué combinación de tecla reinicia la computadora?

16.5 Control de hardware mediante el uso de puertos 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 con 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 numeraciones específicas, conocidas como *puertos*. Los puertos son conexiones, o compuertas, entre la CPU y otros dispositivos, como el teclado, las bocinas, el módem y la tarjeta de sonido.

16.5.1 Puertos de entrada-salida

Cada puerto de entrada-salida tiene un número específico entre 0 y FFFFh. Por ejemplo, para controlar la bocina se utiliza un puerto, haciendo que el cono de la misma se mueva con rapidez hacia dentro y hacia fuera. 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 de 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 contraste, 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           ; recibe doble palabra del puerto
out dx,eax           ; envía doble palabra al mismo puerto
```

16.5.2 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, mani-

pulando 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, y se imprime el estado de nuevo.

Nuestro programa de sonido no producirá sonidos en una computadora portátil si su altavoz está conectado directamente a la tarjeta de sonido, en vez de estar conectado al puerto del 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.

INCLUDE Irvine16.inc      ; programa en modo real de 16 bits

altavoz      = 61h          ; dirección del puerto del altavoz
temporizador = 42h          ; dirección del puerto del temporizador
retraso1     = 500           ; retraso entre las notas
retraso2     = 0D000h

.code
main PROC
    in    al,altavoz        ; obtiene el estado del altavoz
    push ax                 ; guarda el estado
    or    al,00000011b       ; establece los 2 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       ; ciclo exterior
L3:  push cx
    mov   cx,retraso2       ; ciclo interior
L3a: loop L3a
    pop   cx
    loop L3
    sub   al,1                ; eleva el tono
    jnz   L2                  ; reproduce otro tono

    pop   ax                 ; 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  altavoz, 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 tenga que ajustar los valores de **retraso1** y **retraso2**:

```

        mov    cx,retraso1
L3:  push   cx                      ; ciclo exterior
      mov    cx,retraso2
L3a:                         ; ciclo interior
      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          ; obtiene el estado original
and  al,11111100b ; borra los 2 bits inferiores
out   altavoz,al  ; apaga el altavoz

```

16.6 Resumen del capítulo

En ocasiones, los programadores necesitan crear definiciones explícitas de segmentos, en especial cuando se adaptan a las bibliotecas de código existentes que utilizan sus propios nombres de segmentos. Las directivas SEGMENT y ENDS definen el inicio y el fin de un segmento, respectivamente. Cuando el segmento que se va a definir se combina con otro segmento, su *tipo de alineación* indica al enlazador cuántos bytes debe omitir. El *tipo de combinación* indica al enlazador cómo combinar segmentos que tienen el mismo nombre. El tipo de clase de un segmento proporciona otra manera más de combinar segmentos. Pueden combinarse varios segmentos si se les da el mismo nombre y se especifica un tipo de combinación PUBLIC.

La directiva ASSUME permite al ensamblador calcular los desplazamientos de las etiquetas y las variables en tiempo de ensamblado. Un prefijo de redefinición de segmento instruye al procesador para que utilice un registro de segmento distinto al segmento predeterminado para la instrucción actual.

El procesador de comandos de MS-DOS interpreta cada comando escrito en un símbolo del sistema. Los programas con las extensiones COM y EXE se llaman *programas transientes*. Se cargan en memoria y se ejecutan, y después se libera la memoria que ocupan. MS-DOS crea un bloque especial de 256 bytes al principio de un programa transiente, llamado *Prefijo de segmento del programa*.

Hay dos tipos de programas transientes, que se identifican en base a la extensión que utilizan: COM y EXE. Un programa COM es una imagen binaria sin modificaciones de un programa en lenguaje ensamblador. Un programa EXE se almacena en disco con un encabezado EXE, seguido de un módulo de carga que contiene al programa en sí. MS-DOS utiliza el área de encabezado de un programa EXE para calcular correctamente las direcciones de los segmentos y de otros componentes.

Los *manejadores de interrupciones* (rutinas de servicio de interrupciones) simplifican la entrada/salida, así como las tareas básicas del sistema. También puede sustituir los manejadores de interrupciones predeterminados con su propio código, para ofrecer servicios más completos o personalizados. La tabla de vectores de interrupción se encuentra en los primeros 1024 bytes de RAM (ubicaciones 0:0 a 0:03FF). Cada entrada en la tabla es una dirección tipo segmento-desplazamiento de 32 bits, que apunta a una rutina de servicio de interrupciones.

Una interrupción de hardware se genera mediante el Controlador de interrupciones programable (PIC), el cual indica a la CPU que debe suspender la ejecución del programa actual y ejecutar una rutina de servicio de interrupciones. Las interrupciones de hardware permiten que la CPU detecte los eventos importantes en segundo plano antes de perder datos esenciales. Las interrupciones pueden activarse mediante un número de dispositivos diferentes, cada uno de los cuales tiene una prioridad con base en su *nivel de petición de interrupciones* (IRQ).

La bandera Interrupción controla la forma en la que la CPU responde a las interrupciones externas (de hardware). Si se activa la bandera Interrupción, se habilitan las interrupciones; si se borra la bandera, se deshabilitan las interrupciones. La instrucción STI (establecer interrupción) habilita las interrupciones; la instrucción CLI (borrar interrupción) deshabilita las interrupciones.

Un programa TSR (*terminar y permanecer residente*) deja una parte de sí mismo en la memoria. El uso más común para los programas TSR es para los manejadores de interrupciones instalados que permanecen en memoria hasta que la computadora se reinicia, o cuando el TSR se elimina mediante un desinstalador especial.

El programa No_reinicio que presentamos en este capítulo es un programa TSR que evita que el sistema se reinicie mediante las teclas Ctrl-Alt-Supr usuales.

Los sistemas IA-32 ofrecen dos tipos de entrada-salida de hardware: por *asignación de memoria y basada en puerto*. Cuando se utiliza la *E/S por asignación de memoria*, un programa puede escribir datos a una dirección específica de memoria, y los datos se transfieren al dispositivo de salida. La *E/S basada en puerto* requiere que las instrucciones IN y OUT lean y escriban datos en ubicaciones con numeraciones específicas, llamadas *puertos*.

El puerto de control del altavoz (número 61h) enciende y apaga el altavoz, para lo cual manipula el chip de *Interfaz periférica programable* Intel 8255. El programa Demo de altavoz muestra cómo generar sonidos mediante la reproducción de una serie de notas ascendentes.

PROCESAMIENTO DE PUNTO FLOTANTE y codificación de instrucciones

- | | |
|---|---|
| 17.1 Representación binaria de punto flotante | 17.2.8 Sincronización de excepciones |
| 17.1.1 Representación de punto flotante binaria IEEE | 17.2.9 Ejemplos de código |
| 17.1.2 El exponente | 17.2.10 Aritmética de modo mixto |
| 17.1.3 Números de punto flotante binarios normalizados | 17.2.11 Enmascaramiento y desenmascaramiento de excepciones |
| 17.1.4 Creación de la representación IEEE | 17.2.12 Repaso de sección |
| 17.1.5 Conversión de fracciones decimales a reales binarios | 17.3 Codificación de instrucciones Intel |
| 17.1.6 Repaso de sección | 17.3.1 Formato de instrucciones IA-32 |
| 17.2 Unidad de punto flotante | 17.3.2 Instrucciones de un solo byte |
| 17.2.1 Pila de registros FPU | 17.3.3 Movimiento inmediato a un registro |
| 17.2.2 Redondeo | 17.3.4 Instrucciones en modo de registro |
| 17.2.3 Excepciones de punto flotante | 17.3.5 Prefijo de tamaño de operando del procesador IA-32 |
| 17.2.4 Conjunto de instrucciones de punto flotante | 17.3.6 Instrucciones en modo de memoria |
| 17.2.5 Instrucciones aritméticas | 17.3.7 Repaso de sección |
| 17.2.6 Comparación de valores de punto flotante | 17.4 Resumen del capítulo |
| 17.2.7 Lectura y escritura de valores de punto flotante | 17.5 Ejercicios de programación |

17.1 Representación binaria de punto flotante

Un número decimal de punto flotante contiene tres componentes: un signo, una mantisa y un exponente. Por ejemplo, en el número -1.23154×10^5 , el signo es negativo, la mantisa es 1.23154 y el exponente es 5.

Busque la documentación de Intel IA-32. Para aprovechar este capítulo al máximo, obtenga una copia electrónica gratuita del *Manual para el desarrollador de software de la arquitectura Intel IA-32 (IA-32 Intel Architecture Software Developer's Manual)*, Volúmenes 1 y 2. Escriba la dirección www.intel.com en su explorador Web y busque *manuales IA-32*.

17.1.1 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 tabla 17.1 describe sus características.¹

Tabla 17-1 Formatos binarios de punto flotante de IEEE.

Precisión simple	32 bits: 1 bit para el signo, 8 bits para el exponente y 23 bits para la parte fraccional de la mantisa. Rango normalizado aproximado: 2^{-126} a 2^{127} . También se le conoce como <i>real corto</i>
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 <i>real largo</i>
Precisión doble extendida	80 bits: 1 bit 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 <i>real extendido</i>

Como los tres formatos son muy similares, nos enfocaremos en el formato de precisión simple (figura 17-1). Los 32 bits se ordenan con el bit más significativo (MSB) a la izquierda. El segmento que se marca como *fracción* indica la parte fraccional de la mantisa. Como podría esperar, los bytes individuales se almacenan en memoria en el orden little endian (LSB en la dirección inicial).

FIGURA 17-1 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 la 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 binario, 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:

$$123.154 = (1 \times 10^2) + (2 \times 10^1) + (3 \times 10^0) + (1 \times 10^{-1}) + (5 \times 10^{-2}) + (4 \times 10^{-3})$$

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

$$11.1011 = (1 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) + (1 \times 2^{-4})$$

Otra forma de expresar los valores a la derecha del punto binario es presentarlos 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 patrón 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 tabla 17-2 muestra ejemplos adicionales de cómo traducir la 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. Para una referencia rápida, la tabla 17-3 presenta ejemplos de números binarios de punto flotante, junto con sus fracciones decimales equivalentes y valores decimales.

Tabla 17-2 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

Tabla 17-3 Fracciones binarias y decimales.

Binario	Fracción decimal	Valor decimal
.1	1/2	.5
.01	1/4	.25
.001	1/8	.125
.0001	1/16	.0625
.00001	1/32	.03125

La precisión de la mantisa

El continuo completo de números reales no puede representarse en ningún formato de punto flotante que tenga un número finito de bits. Por ejemplo, suponga que un formato de punto flotante simplificado tiene mantisas de 5 bits. No habría forma de representar los valores que se encuentren entre los números 1.1111 y 10.0000 binarios. Por ejemplo, el valor binario 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.

17.1.2 El exponente

Los exponentes de precisión simple se almacenan como enteros sin signo de 8 bits con una desviación de 127. El exponente actual del número debe sumarse a 127. Considere el valor binario 1.101×2^5 . Después de que se suma el exponente actual (5) a 127, el exponente con desviación (132) se almacena en la representación del número. La tabla 17-4 muestra ejemplos de exponentes en decimal con signo, después en decimal con desviación y, por último, en binario sin signo. 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 posible del exponente no pueda ocasionar un desbordamiento.

Tabla 17-4 Ejemplos de exponentes representados en binario.

Exponente (E)	Con desviación (E + 127)	Binario
+5	132	10000100
0	127	01111111
-10	117	01110101
+127	254	11111110
-126	1	00000001
-1	126	01111110

17.1.3 Números de punto flotante binarios normalizados

La mayoría de los números binarios de punto flotante se almacenan en formato *normalizado*, 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	1.1101×2^3
.000101	1.01×2^{-4}
1010001.	1.010001×2^6

Valores sin normalizar Podríamos decir que *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, llenando con ceros si es necesario.

17.1.4 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 17-1 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 1.101×2^0 se representa de la siguiente manera:

- Bit de signo: 0
- Exponente: 0111111
- Fracción: 10100000000000000000000000000000

El exponente con desviación (0111111) es la representación binaria del 127 decimal. Todas las mantisas normalizadas tienen un 1 a la izquierda del punto binario, por lo que no hay necesidad de codificar el bit de forma explícita. En la tabla 17-5 se muestran ejemplos adicionales.

La especificación IEEE incluye varias codificaciones de números reales 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.

Tabla 17-5 Ejemplos de codificaciones de bits de precisión simple.

Valor binario	Exponente con desviación	Signo, Exponente, Fracción
-1.11	127	1 01111111 11000000000000000000000000000000
+1101.101	130	0 10000010 10110100000000000000000000000000
-.00101	124	1 01111100 01000000000000000000000000000000
+100111.0	132	0 10000100 00111000000000000000000000000000
+.0000001101011	120	0 01111000 10101100000000000000000000000000

La unidad Intel de punto flotante utiliza los números indefinidos como respuestas para algunas operaciones inválidas con números de punto flotante.

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 distintos 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. Suponga que la FPU calcula un resultado de 1.010111×2^{-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 en 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:

$$\begin{aligned}1.0101110000000000001111 &\times 2^{-129} \\0.1010111000000000000111 &\times 2^{-128} \\0.0101011100000000000011 &\times 2^{-127} \\0.0010101110000000000001 &\times 2^{-126}\end{aligned}$$

En este ejemplo se produjo cierta pérdida de precisión en el significando, como resultado del desplazamiento del punto decimal.

Infinito positivo y negativo El infinito positivo ($+\infty$) representa al máximo número real positivo, y el infinito negativo ($-\infty$) representa al máximo número real negativo. Podemos comparar a los infinitos con otros valores: $-\infty$ es menor que $+\infty$, $-\infty$ es menor que cualquier número finito, y $+\infty$ es mayor que cualquier número finito. Cualquiera de los infinitos puede representar una condición de desbordamiento de punto flotante. El resultado de un cálculo no puede normalizarse, ya que su exponente sería demasiado grande como para poder representarse mediante el número disponible 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 depuración. Un programa es libre de codificar cualquier información que deseé en un NaN. La unidad de punto flotante no trata de realizar operaciones con los NaNs. El manual de la familia Intel IA-32 detalla un conjunto de reglas que determinan los resultados de las instrucciones cuando se utilizan combinaciones de los dos tipos de NaNs como operandos.²

Codificaciones específicas Hay varias codificaciones especiales para los valores que encontramos a menudo en las operaciones de punto flotante; estas codificaciones se muestran en la tabla 17-6. Las posiciones de bit marcadas con la letra *x* pueden ser 1 o 0. QNaN es un NaN silencioso, y SNaN es un NaN de señalización.

Tabla 17-6 Codificaciones específicas de precisión simple.

^a El campo de la mantisa de un NaN empieza con 0, pero por lo menos uno de los bits restantes debe ser 1.

17.1.5 Conversión de fracciones decimales a reales binarios

Cuando una fracción decimal puede representarse como una suma de fracciones en la forma $(1/2 + 1/4 + 1/8 + \dots)$, es muy sencillo descubrir el real binario correspondiente. En la tabla 17-7, la mayoría de las fracciones en la columna izquierda no se encuentran en un formato que se traduzca fácilmente a binario. Sin embargo, pueden escribirse como en la segunda columna.

Tabla 17-7 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 aproximarnos a una fracción así mediante una suma de fracciones, cuyos denominadores sean potencias de 2. ¡Imagine cómo se verían afectados los valores monetarios como \$39.95!

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 numerador 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 el 1010 binario. Si realizamos la división larga binaria, encontraremos que el cociente es el 0.1 binario:

$$\begin{array}{r}
 & .1 \\
 1 & 0 & 1 & 0 & \left[\begin{array}{r} 0 & 1 & 0 & 1 & 0 \\ -1 & 0 & 1 & 0 \\ \hline 0 \end{array} \right]
 \end{array}$$

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 larga binaria*.³

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 el 1010 binario (10 decimal):

$$\begin{array}{r}
 & .0\ 0\ 1\ 1\ 0\ 0\ 1\ 1 \text{ (etc.)} \\
 \hline
 1\ 0\ 1\ 0 & \left| \begin{array}{r} 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ 1\ 0\ 1\ 0 \\ \hline 1\ 1\ 0\ 0 \\ 1\ 0\ 1\ 0 \\ \hline 1\ 0\ 0\ 0 \\ 1\ 0\ 1\ 0 \\ \hline 1\ 1\ 0\ 0 \\ 1\ 0\ 1\ 0 \\ \hline \end{array} \right. \\
 & \text{etc.}
 \end{array}$$

El primer cociente lo bastante grande para usar es 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 0.2 no puede representarse mediante un número finito de bits. La mantisa codificada de precisión simple es 0011001100110011001.

Conversión de valores de precisión simple a decimales

A continuación se muestran los pasos sugeridos para convertir un valor IEEE de precisión 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 (127 decimal), con lo que se produce el exponente sin desviación. Luego se convierte el 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 10000010 01011000000000000000000000000000) a decimal

1. El número es positivo.
2. El exponente sin desviación es 00000011 binario, o 3 decimal.
3. Si combinamos el signo, exponente y mantisa, el número binario es $+1.01011 \times 2^3$.
4. El número binario denormalizado es +1010.11.
5. El valor decimal es $+10\frac{3}{4}$, o +10.75.

17.1.6 Repaso de sección

1. ¿Por qué el formato real de precisión simple no permite un exponente de -127 ?
2. ¿Por qué el formato real de precisión simple no permite un exponente de $+128$?
3. En el formato IEEE de doble precisión, ¿cuántos bits se reservan para la parte fraccional de la mantisa?

4. En el formato IEEE de precisión simple, ¿cuántos bits se reservan para el exponente?
5. Exprese el valor binario de punto flotante 1101.01101 como una suma de fracciones decimales.
6. Explique por qué el 0.2 decimal no puede representarse exactamente mediante un número finito de bits.
7. Normalice el valor binario 11011.01011
8. Normalice el valor binario 0000100111101.1
9. Muestre la codificación IEEE de precisión simple para el número +1110.011 binario.
10. ¿Cuáles son los dos tipos de *Nan*s?
11. Convierta la fracción 5/8 en un real binario.
12. Convierta la fracción 17/32 en un real binario.
13. Convierta el valor decimal +10.75 a un real IEEE de precisión simple.
14. Convierta el valor decimal -76.0625 a un real IEEE de precisión simple.

17.2 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 matemáticas 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 hardware de punto flotante se integró en la CPU principal y se le llamó *Unidad de punto flotante* (FPU).

17.2.1 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 que 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 figura 17-2 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 documentados en los textos de introducción a las ciencias computacionales y en Internet, por lo que no los explicaremos aquí. La tabla 17-8 contiene algunos ejemplos de expresiones equivalentes.

Tabla 17-8 Ejemplos de infijo a postfijo.

Infijo	Postfijo
$A + B$	$A \ B \ +$
$(A - B) / D$	$A \ B \ - \ D \ /$
$(A + B) * (C + D)$	$A \ B \ + \ C \ D \ + \ *$
$((A + B) / C) * (E - F)$	$A \ B \ + \ C \ / \ E \ F \ - \ *$

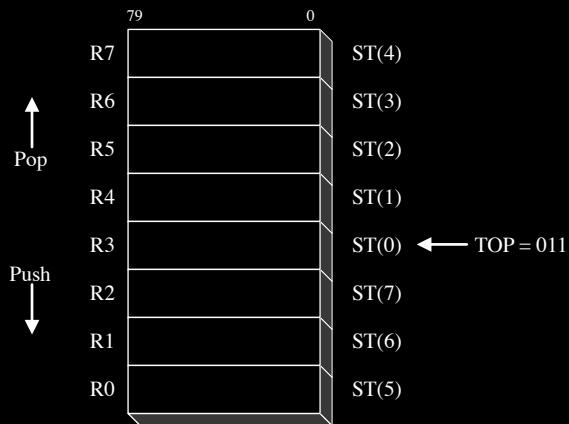
FIGURA 17-2 Evaluación de la expresión postfijo $5 \ 6 * 4 -$.

Izquierda a derecha	Pila	Acción		
5	<table border="1"><tr><td>5</td></tr></table>	5	ST (0) Push 5	
5				
5 6	<table border="1"><tr><td>5</td></tr><tr><td>6</td></tr></table>	5	6	ST (1) ST (0) Push 6
5				
6				
5 6 *	<table border="1"><tr><td>30</td></tr></table>	30	Multiplica ST(1) por ST(0) y saca a ST(0) de la pila	
30				
5 6 * 4	<table border="1"><tr><td>30</td></tr><tr><td>4</td></tr></table>	30	4	ST (1) ST (0) Push 4
30				
4				
5 6 * 4 -	<table border="1"><tr><td>26</td></tr></table>	26	Resta ST(0) de ST(1) y saca a ST(0) de la pila	
26				

Registros 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 (vea la figura 17-3). 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 figura 17-3 TOP es igual al número 011 binario, con lo cual identifica a R3 como la parte superior de la pila. Esta ubicación de la pila se conoce también como ST(0) (o simplemente ST) al escribir instrucciones de punto flotante. El último registro es ST(7).

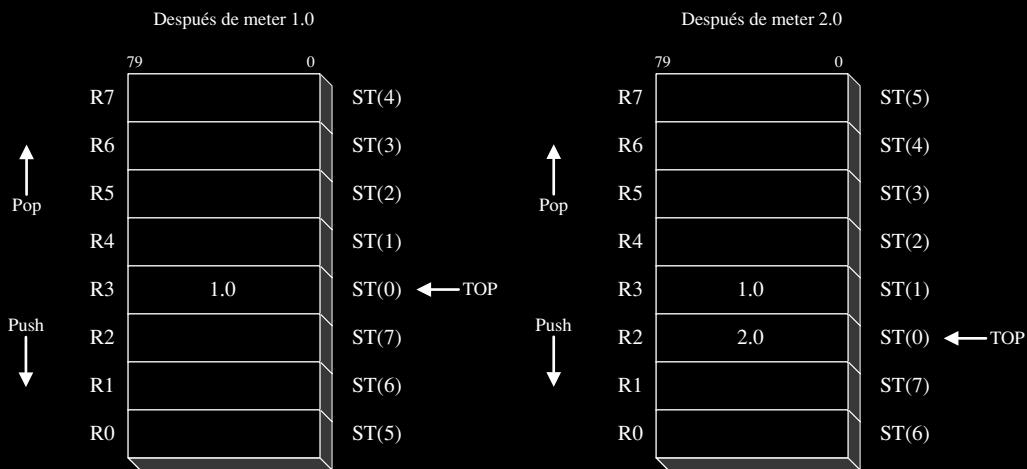
FIGURA 17-3 Pila de registros de datos de punto flotante.



Como podríamos esperar, una operación *push* (también conocida como *cargar*) decremente a TOP en 1 y copia un operando en el registro identificado como ST(0). Si TOP es igual a 0 antes de una 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 sobrescriben los datos existentes en la pila de registros, se genera una *excepción de punto flotante*. La figura 17-4 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 referiremos 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.

FIGURA 17-4 La pila de la FPU, después de meter 1.0 y 2.0.



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 resultado de una operación aritmética en memoria, traduce el resultado a uno de los siguientes formatos: entero, entero largo, de precisión simple (real corto), de precisión doble (real largo), o decimal codificado en binario empaquetado.

Registros de propósito especial

La FPU tiene seis registros de *propósito especial* (vea la figura 17-5):

- **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 usarlo 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 excepciones.
- **Registro de etiqueta:** indica el contenido de cada registro en la pila de registros de datos de la FPU. Utiliza dos bits por registro para indicar si el registro contiene un número válido, cero 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 registros de propósito especial para preservar la información de estado al cambiar de una tarea a otra. En el capítulo 2 mencionamos la preservación del estado, cuando explicamos cómo realiza la CPU la multitarea.

17.2.2 Redondeo

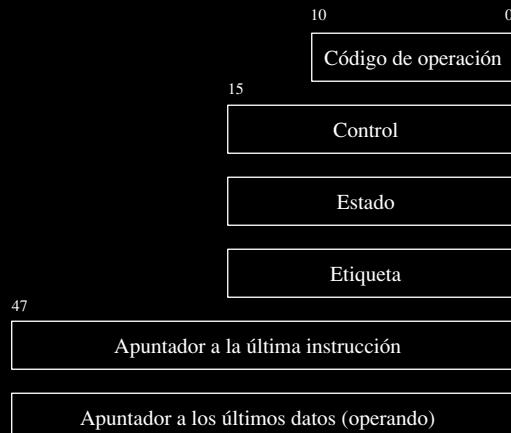
La FPU trata de generar un resultado infinitamente preciso de un cálculo de punto flotante. En muchos casos esto es imposible, ya que el operando de destino tal vez no pueda representar de manera precisa el resultado calculado. Por ejemplo, suponga que cierto formato de almacenamiento sólo permite tres bits fraccionales. Nos permitiría almacenar valores como 1.011 o 1.101, pero no 1.0101. Suponga 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 redondearlo hacia abajo, restándole .0001:

- 1.0111 --> 1.100
- 1.0111 --> 1.011

Si el resultado preciso fuera negativo, al sumarle $- .0001$, el resultado redondeado se movería cerca de $-\infty$. Al restarle $- .0001$ se movería más cerca de cero como de $+\infty$:

- (a) $-1.0111 \rightarrow -1.100$
- (b) $-1.0111 \rightarrow -1.011$

FIGURA 17-5 Registros de propósito especial de la FPU.



La FPU nos permite seleccionar uno de cuatro métodos de redondeo:

- *Redondeo al número par 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 $-\infty$* : el resultado redondeado es menor o igual al resultado infinitamente preciso.
- *Redondeo hacia $+\infty$* : 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 control de la FPU La palabra de control de la FPU contiene dos bits llamados *campo RC*, que especifican el método de redondeo a utilizar. Los valores de los campos son:

- 00 binario: redondea al número par más cercano (predeterminado).
- 01 binario: redondea hacia infinito negativo.
- 10 binario: redondea hacia infinito positivo.
- 11 binario: redondea hacia cero (trunca).

Redondear al número par más cercano es el método predeterminado, y se considera como el más apropiado y preciso para la mayoría de los programas de aplicaciones. La tabla 17-9 muestra cómo se aplicarían los cuatro métodos de redondeo al número $+1.0111$ binario. De manera similar, la tabla 17-10 muestra los posibles redondeos del número -1.0111 binario.

Tabla 17-9 Ejemplo: redondeo de $+1.0111$.

Método	Resultado preciso	Redondeado
Redondea al número par más cercano	1.0111	1.100
Redondea hacia $-\infty$	1.0111	1.011
Redondea hacia $+\infty$	1.0111	1.100
Redondea hacia cero	1.0111	1.011

Tabla 17-10 Ejemplo: redondeo de -1.0111.

Método	Resultado preciso	Redondeado
Redondea al número más cercano (par)	-1.0111	-1.100
Redondea hacia $-\infty$	-1.0111	-1.100
Redondea hacia $+\infty$	-1.0111	-1.011
Redondea hacia cero	-1.0111	-1.011

17.2.3 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 detectan 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 correspondientes. 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 los 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 ocurrieron desde la última vez que se borraron las excepciones. Al completarse una secuencia de cálculos, podemos comprobar si ocurrió alguna excepción.

17.2.4 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 puede ejercer 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 la 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 y \log_{10}) en la pila. Los registros de propósito general como EAX, EBX, ECX y EDX no pueden ser operandos (la única excepción es FSTSW, 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 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. Recomendamos 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 tabla 17-11. Deberá utilizar estos tipos cuando defina 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 1.212342342234234243E+864
.code
fld valGrande ; carga la variable en la pila
```

Tabla 17-11 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 tipos de operandos de memoria que MOV. He aquí algunos ejemplos:

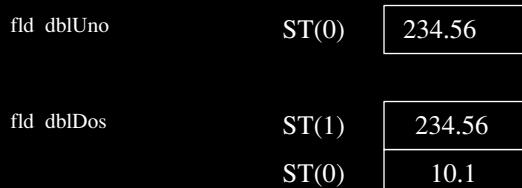
```
.data
arreglo REAL8 10 DUP(?)
.code
```

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-índice
fld arreglo[ebx+esi]	; base-índice-desplazamiento
fld arreglo[ebx+esi*TYPE arreglo]	; base-índice-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 decremente, 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 doble precisión y lo carga en ST(0). Se preserva el signo del operando de destino. Demostraremos su uso en la sección 17.2.10 (Aritmética en modo mixto). FILD soporta los mismos 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 $\log_2 10$ en la pila de registros.
- La instrucción FLDL2E mete $\log_2 e$ en la pila de registros.
- La instrucción FLDPI mete π en la pila de registros.
- La instrucción FLDLG2 mete $\log_{10} 2$ en la pila de registros.
- La instrucción FLDLN2 mete $\log_e 2$ 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, 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 la pila. Las siguientes instrucciones almacenan a ST(0) en la memoria. Vamos a suponer que ST(0) es igual a 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, debemos usar la instrucción FSTP.

FSTP La instrucción FSTP (almacena 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 a 234.56, antes de ejecutar las siguientes instrucciones:

```
fstp dblTres ; 10.1
fstp dblCuatro ; 234.56
```

Después de la ejecución, los dos valores se eliminaron 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 (almacena 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. En la sección 17.2.10 demostraremos su uso (Aritmética de modo mixto). FIST soporta los mismos tipos de operandos de memoria que FST.

17.2.5 Instrucciones aritméticas

En la tabla 17-12 se presentan las operaciones aritméticas básicas. Todas las instrucciones aritméticas soporan los mismos tipos de operandos que FLD (cargar) y FST (almacenar), por lo que los operandos pueden ser indirectos, indexados, de base-índice, etcétera.

Tabla 17-12 Instrucciones básicas de aritmética 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:

```
FADD4
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:

fadd	Antes:	ST(1)	234.56
		ST(0)	10.1
	Después:	ST(0)	244.66

Operandos de registro Empezando con el mismo contenido de la pila, la siguiente ilustración demuestra cómo se suma ST(0) a ST(1):

fadd st(1), st(0)	Antes:	ST(1)	234.56
		ST(0)	10.1
	Después:	ST(1)	244.66
		ST(0)	10.1

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 fadd REAL PTR[esi]	; ; ; ST(0) += miSimple ; 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:

faddp st(1), st(0)	Antes:	ST(1)	234.56
		ST(0)	10.1
	Después:	ST(0)	244.66

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:

```
FSUB5
FSUB m32pf
FSUB m64pf
FSUB ST(i), ST(j)
FSUB ST(i), ST(0)
```

La operación de FSUB es similar a la de FADD, sólo que resta en vez de sumar. Por ejemplo, la forma sin operandos de FSUB resta ST(0) de ST(1). El resultado se almacena temporalmente en ST(1). Después, ST(0) se saca de la pila, 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:

```
FMUL6
FMUL m32pf
FMUL m64pf
FMUL ST(0), ST(j)
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 misma sintaxis que FADD y FSUB:

```
FDIV7
FDIV m32pf
FDIV m64pf
FDIV ST(i), ST(j)
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 obtener detalles, consulte el manual del 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
```

17.2.6 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 FCOM (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 <i>m32pf</i>	Compara ST(0) con <i>m32pf</i>
FCOM <i>m64pf</i>	Compara ST(0) con <i>m64pf</i>
FCOM ST(<i>i</i>)	Compara ST(0) con ST(<i>i</i>)

La instrucción FCOMP lleva a cabo las mismas operaciones con los mismos tipos de operandos, 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 resultados de comparar valores de punto flotante (tabla 17-13). 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.

Tabla 17-13 Códigos de condición establecidos por FCOM, FCOMP, FCOMPP.

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 ^a	1	1	1	(Ninguno)

^a Si se genera una excepción de operando aritmético inválido (debido a operandos inválidos) y la excepción está enmascarada, C3, C2 y C0 se establecen de acuerdo con la fila marcada *Sin orden*.

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 17-13 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 Asuma 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 del 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 compara 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 Pentium II). FCOMI tiene la siguiente sintaxis:

```
FCOMI ST(0),ST(1)
```

Vamos a rescribir nuestro ejemplo de código anterior (comparar X con Y) usando FCOMI:

```
.code
; if( X < Y )
;   N = 1
    fld  Y           ; ST(0) = Y
    fld  X           ; ST(0) = X, ST(1) = Y
    fcomi ST(0),ST(1) ; compara a ST(0) con ST(1)
    jnb  L1          ; ¿ST(0) no es < ST(1)? Salta
    mov  N,1          ; N = 1
L1:
```

La instrucción FCOMI tomó el lugar de tres instrucciones en la versión anterior, pero requirió una instrucción FLD más. La instrucción FCOMI no acepta operandos de memoria.

Comparación de igualdad

Casi todos los libros de texto de programación para principiantes advierten al lector que no debe 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 matemáticos, debería ser igual a cero pero los resultados son bastante distintos (aproximadamente $4.4408921E-016$). Utilizaremos los siguientes datos y mostraremos la pila de la FPU después de cada paso en la tabla 17-14:

```
val1 REAL8 2.0
```

Tabla 17-14 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 y 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 lenguaje en ensamblador para hacer esto, usando épsilon como la máxima diferencia que pueden 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 tabla 17-5 rastrea el progreso del programa, mostrando la pila después de la ejecución de cada una de las primeras cuatro instrucciones.

Tabla 17-15 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 redefiniéramos a val3 como mayor que épsilon, no sería igual a val2:

val3 REAL8 1.001E-12 ; no es igual

17.2.7 Lectura y escritura de valores de punto flotante

En las bibliotecas de vínculos del libro se incluyen dos procedimientos para operaciones de entrada-salida de punto flotante, que creó William Barret de la Universidad Estatal de San José:

- **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 la 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
3.5E005
-3.5E+5
3.5E-4
+3.5E-4
```

ShowFPUStruct Otro útil procedimiento, que escribió James Brink de la Universidad Pacific Lutheran, muestra la pila de la FPU. Se llama sin parámetros:

```
call ShowFPUStruct
```

Programa de ejemplo El siguiente programa de ejemplo mete dos valores de punto flotante en la pila de la FPU, los muestra, recibe dos valores del usuario, los multiplica y muestra su producto:

```
TITLE Prueba de E/S con valores de punto flotante de 32 bits (pruebaFlotante32.asm)
INCLUDE Irvine32.inc
INCLUDE macros.inc

.data
primero REAL8 123.456
segundo REAL8 10.0
tercero REAL8 ?

.code
main PROC
    finit           ; inicializa la FPU
```

```

; Mete dos valores de punto flotante y muestra la pila de la FPU.
fld primero
fld segundo
call ShowFPUStack

; Recibe dos valores de punto flotante y muestra su producto.
mWrite "Escriba un numero real: "
call ReadFloat

mWrite "Escriba un numero real: "
call ReadFloat

fmul ST(0),ST(1)           ; multiplica

mWrite "Su producto es: "
call WriteFloat
call Crlf

exit
main ENDP
END main

```

Ejemplo de entrada/salida (la entrada del usuario se muestra en negritas):

```

----- FPU Stack -----
ST(0): +1.0000000E+001
ST(1): +1.2345600E+002
Escriba un numero real: 3.5
Escriba un numero real: 4.2
Su producto es: 1+1.4700000E+001

```

17.2.8 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 *concurrente*, 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
fld 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
```

17.2.9 Ejemplos de código

En esta sección veremos algunos 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 ST(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) *= valC
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 ; índice 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 al siguiente elemento
    loop   L1
    call   WriteFloat ; muestra la suma en ST(0)
```

Suma de raíces cuadradas

La instrucción FSQRT sustituye el número en ST(0) con su raíz cuadrada. El siguiente código calcula la suma de dos raíces 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 $(\text{arreglo}[0] * \text{arreglo}[1]) + (\text{arreglo}[2] * \text{arreglo}[3])$. A este cálculo algunas veces se le conoce como *producto punto*. La tabla 17-16 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

```

Tabla 17-16 Cálculo de un producto punto $(6.0 * 2.0) + (4.5 * 3.2)$.

Instrucción	Pila de la FPU
fld arreglo	ST(0): +6.000000E+000
fmul [arreglo+4]	ST(0): +1.200000E+001
fld [arreglo+8]	ST(0): +4.500000E+000 ST(1): +1.200000E+001
fmul [arreglo+12]	ST(0): +1.440000E+001 ST(1): +1.200000E+001
fadd	ST(0): +2.640000E+001

17.2.10 Aritmética de modo mixto

Hasta este punto hemos realizado operaciones aritméticas que involucran sólo números reales. A menudo, las aplicaciones realizan operaciones aritméticas en modo mixto, combinando enteros y reales. Las instrucciones de aritmética 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 (de manera 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 (bits 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 palabra de control
and   palabCtrl,001111111111b   ; restablece el redondeo al predeterminado
fldcw palabCtrl      ; carga palabra de control
```

17.2.11 Enmascaramiento y desenmascaramiento de excepciones

Las excepciones se enmascararon de manera predeterminada (sección 17.2.3), de manera que cuando se genera una excepción de punto flotante, el procesador asigna un valor predeterminado al resultado y continúa realizando su trabajo en silencio. 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 desenmascara la excepción en la palabra de control de la FPU, el procesador trata de ejecutar un manejador de excepciones apropiado. El desenmascaramiento se logra borrando el bit apropiado en la palabra de control de la FPU (tabla 17-17). Suponga que deseamos desenmascarar la excepción de división 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 división 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,111111111111011b   ; desenmascara división entre cero
fldcw palabCtrl      ; la carga de vuelta en la FPU
```

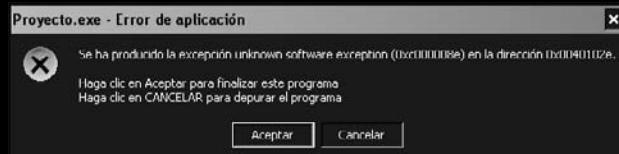
Tabla 17-17 Campos en la palabra de control de la FPU.

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

Ahora, si ejecutamos código que divide entre cero, se genera una excepción desenmascarada:

```
fild val1
fdiv val2 ; división entre cero
fst val2
```

Tan pronto como la instrucción FST empieza a ejecutarse, MS Windows muestra el siguiente cuadro de diálogo:



Enmascaramiento de excepciones Para enmascarar una excepción, se activa 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
```

17.2.12 Repaso de sección

- Escriba una instrucción para cargar un duplicado de ST(0) en la pila de la FPU.
- Si ST(0) se posiciona en el registro absoluto R6 de la pila de registros, ¿cuál es la posición de ST(2)?
- Mencione por lo menos tres registros de propósito especial de la FPU.
- Cuando la segunda letra de una instrucción de punto flotante es B, ¿qué tipo de operando se indica?
- ¿Qué instrucciones aceptan operandos inmediatos?
- ¿Cuál es el tipo de datos más grande que permite la instrucción FLD, y ¿cuántos bytes contiene?
- ¿Qué diferencia hay entre las instrucciones FSTP y FST?
- ¿Qué instrucción cambia el signo de un número?
- ¿Qué tipos de operandos pueden usarse con la instrucción FADD?

10. ¿Qué diferencia hay entre las instrucciones FISUB y FSUB?
11. En los procesadores anteriores a la familia P6, ¿qué instrucción compara dos valores de punto flotante?
12. Escriba una secuencia de dos instrucciones para mover las banderas de estado de la FPU al registro EFLAGS.
13. ¿Qué instrucción carga un operando entero en ST(0)?
14. ¿Qué campo en la palabra de control de la FPU nos permite cambiar el modo de redondeo del procesador?
15. Dado un resultado preciso de 1.010101101, redondéelo a una mantisa de 8 bits, usando el método de redondeo predeterminado de la FPU.
16. Dado un resultado preciso de –1.010101101, redondéelo a una mantisa de 8 bits, usando el método de redondeo predeterminado de la FPU.
17. Escriba instrucciones para implementar el siguiente código de C++:


```
double B = 7.8;
double M = 3.6;
double N = 7.1;
double P = -M * (N + B);
```
18. Escriba instrucciones para implementar el siguiente código de C++:


```
int B = 7;
double N = 7.1;
double P = sqrt(N) + B;
```

17.3 Codificación de instrucciones Intel

Para comprender el lenguaje ensamblador por completo, debe 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 direccionamiento 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 de los 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 una 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, aritméticas, 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 lenguaje ensamblador y sus operandos en código máquina. *Decodificar* una instrucción significa convertir una instrucción en código máquina a lenguaje ensamblador. ¡Por lo menos, nuestra explicación acerca de los procesos de codificación y decodificación de las instrucciones Intel le ayudará a apreciar más a los autores de MASM!

17.3.1 Formato de instrucciones IA-32

El formato general de instrucciones de máquina de IA-32 (figura 17-6) 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 variante 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 tabla 17-18 describe el campo Mod, y la tabla 17-19 describe el campo R/M para las aplicaciones de 16 bits, cuando MOD = 10 binario.

- El **byte de índice 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.

FIGURA 17-6 Formato de instrucciones IA-32.

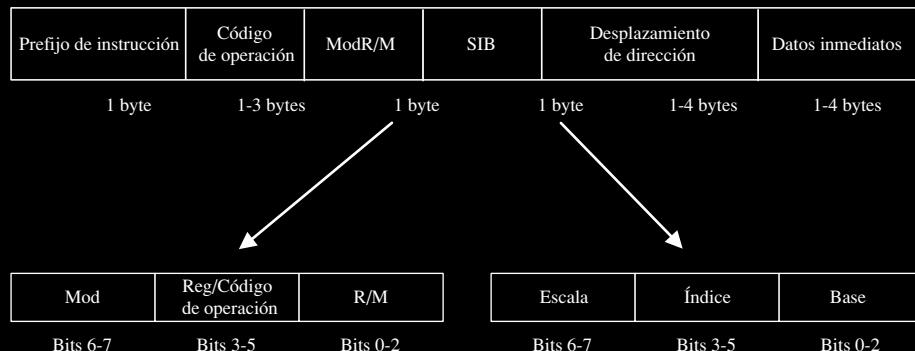


Tabla 17-18 Valores del campo Mod.

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

Tabla 17-19 Valores del campo R/M de 16 bits (para Mod = 10).

R/M	Dirección efectiva
000	[BX + SI] + D16 ^a
001	[BX + DI] + D16
010	[BP + SI] + D16
011	[BP + DI] + D16
100	[SI] + D16
101	[DI] + D16
110	[BP] + D16
111	[BX] + D16

^a D16 indica un desplazamiento de 16 bits.

17.3.2 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 del código de operación, cuyo valor está predeterminado por el conjunto de instrucciones del procesador. La tabla 17-20 presenta algunas instrucciones comunes de un solo byte. Parecería 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.

Tabla 17-20 Instrucciones de un solo byte.

Instrucción	Código de operación
AAA	37
AAS	3F
CBW	98
LODSB	AC
XLAT	D7
INC DX	42

17.3.3 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 codificación de una instrucción MOV que mueve una palabra inmediata en 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 inmediata, el byte inferior primero. Los números de los registros que se utilizan en los códigos de operación se presentan en la tabla 17-21. Todos los valores numéricos en los siguientes ejemplos son hexadecimales:

Tabla 17-21 Números 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 registro 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 (consulte la tabla 17-21).
3. El operando inmediato (0001) se adjunta 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 para BX es 3, por lo que se suma 3 a B8 para producir el código de operación **BB**.
3. Los bytes del operando inmediato son **34 12**.

Para que practique y se familiarice con este proceso, le sugerimos ensamblar a mano algunas instrucciones MOV inmediatas, y después compruebe sus resultados inspeccionando el código generado por MASM en un archivo de listado de código fuente.

17.3.4 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 tabla 17-22 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.

Tabla 17-22 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 Mod 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 al 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 tabla 17-23 presenta algunos ejemplos más que utilizan operandos tipo registro de 8 bits y 16 bits.

Tabla 17-23 Ejemplos de codificaciones de instrucciones MOV, operandos de registro.

Instrucción	Código de operación	mod	reg	r/m
<code>mov ax,dx</code>	8B	11	000	010
<code>mov al,dl</code>	8A	11	000	010
<code>mov cx,dx</code>	8B	11	001	010
<code>mov cl,dl</code>	8A	11	001	010

17.3.5 Prefijo de tamaño de operando del procesador IA-32

Ahora vamos a poner nuestra atención en la codificación de instrucciones para los procesadores Intel de 32 bits (IA-32). Algunas instrucciones en lenguaje máquina, generadas para los procesadores IA-32, empiezan con un

prefijo de tamaño de operando (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 operandos 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 las instrucciones MOV que presentamos antes en la tabla 17-23. 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, mostramos su codificación:

```
.model small
.286
.code
.stack 100h
main PROC
    mov ax,dx          ; 8B C2
    mov al,d1          ; 8A C2
```

No utilizamos el archivo *Irvine16.inc*, ya que está orientado al procesador 386.

Vamos a ensamblar las 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,d1        ; 8A C2
```

17.3.6 Instrucciones en modo de memoria

Si el byte Mod R/M se utilizara sólo para identificar los operandos 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 de operandos mediante el byte Mod R/M. La tabla 17-24 presenta los bytes Mod R/M (en hexadecimal) para Mod 00 (puede consultar la tabla completa en el *Manual del desarrollador de software para la arquitectura Intel IA-32*, Vol. 2A). He aquí cómo 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 **Dirección efectiva**.

Suponga que queremos codificar **MOV AX,[SI]**; los bits Mod son 00 y los bits R/M son 100 binario. Sabemos de la tabla 17-19 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

El byte 04 hexadecimal aparece en la columna marcada AX, en la fila 5.

El byte Mod R/M para **MOV [SI],AL** es el mismo (04h), ya que el registro AL también es el registro número 000. Vamos a codificar la instrucción **MOV[SI],AL**. El código de operación para mover datos desde un registro de 8 bits es **88**. El byte Mod R/M es 04h y la instrucción de máquina es **88 04**.

Tabla 17-24 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]

Ejemplos de la instrucción MOV

Todos los formatos de las instrucciones y los códigos de operación para las instrucciones MOV de 8 y 16 bits se muestran en la tabla 17-25. Las tablas 17-26 y 17-27 proporcionan información suplementaria acerca de las abreviaciones utilizadas en la tabla 17-25. Use estas tablas como referencias cuando ensamble instrucciones MOV a mano. Para obtener más detalles, consulte el *Manual del desarrollador de software para la arquitectura Intel IA-32*, Vol. 2A.

Tabla 17-25 Códigos de operación de la instrucción MOV.

Código de operación	Instrucción	Descripción
88/r	MOV eb,rb	Mueve un registro de byte al byte EA
89/r	MOV ew,rw	Mueve un registro de palabra a la palabra EA
8A/r	MOV rb,eb	Mueve el byte EA al registro de byte
8B/r	MOV rw,ew	Mueve la palabra EA al registro de palabra
8C/0	MOV ew,ES	Mueve ES a la palabra EA
8C/1	MOV ew,CS	Mueve CS a la palabra EA
8C/2	MOV ew,SS	Mueve SS a la palabra EA
8C/3	MOV DS,ew	Mueve DS a la palabra EA
8E/0	MOV ES,mw	Mueve una palabra de memoria a ES
8E/0	MOV ES, rw	Mueve un registro de palabra a ES
8E/2	MOV SS,mw	Mueve una palabra de memoria a SS

(Continúa)

Tabla 17-25 (Continuación)

Código de operación	Instrucción	Descripción
8E/2	MOV SS,rw	Mueve una palabra de registro a SS
8E/3	MOV DS,mw	Mueve una palabra de memoria a DS
8E/3	MOV DS,rw	Mueve un registro de palabra a DS
A0 dw	MOV AL,xb	Mueve variable de byte (desplazamiento dw) a AL
A1 dw	MOV AL,xw	Mueve variable de palabra (desplazamiento dw) a AX
A2 dw	MOV xb,AL	Mueve AL a variable de byte (desplazamiento dw)
A3 dw	MOV xw,AX	Mueve AX a registro de palabra (desplazamiento dw)
B0 +rb db	MOV rb,db	Mueve byte inmediato a registro de byte
B8 +rw dw	MOV rw,dw	Mueve palabra inmediata a registro de palabra
C6 /0 db	MOV eb,db	Mueve byte inmediato a byte EA
C7 /0 dw	MOV ew,dw	Mueve palabra inmediata a palabra EA

Tabla 17-26 Clave para los códigos de operación de las instrucciones.

/n:	Un byte Mod R/M va después del código de operación, posiblemente seguido de los campos inmediato y desplazamiento. El dígito n (0-7) es el valor del campo reg del byte Mod R/M
/r:	Un byte Mod R/M va después del código de operación, posiblemente seguido de los campos inmediato y desplazamiento
db:	Un operando de byte inmediato va después del código de operación y de los bytes Mod R/M
dw:	Un operando de palabra inmediato va después del código de operación y de los bytes Mod R/M
+rb:	Un código de registro (0-7) para un registro de 8 bits, que se suma al byte hexadecimal anterior para formar un código de operación de 8 bits
+rw:	Un código de registro (0-7) para un registro de 16 bits, que se suma al byte hexadecimal anterior para formar un código de operación de 8 bits

Tabla 17-27 Clave para los operandos de las instrucciones.

db	Un valor con signo entre -128 y +127. Si se combina con un operando de palabra, se extiende el signo de este valor
dw	Un valor de palabra inmediato que es un operando de la instrucción
eb	Un operando del tamaño de un byte, ya sea registro o memoria
ew	Un operando del tamaño de una palabra, ya sea registro o memoria
rb	Un registro de 8 bits identificado por el valor (0-7)
rw	Un registro de 16 bits identificado por el valor (0-7)
xb	Una variable de memoria de un byte simple, sin un registro base o índice
xw	Una variable de memoria de una palabra simple, sin registro base o índice

La tabla 17-28 contiene algunos ejemplos adicionales de instrucciones MOV que podemos ensamblar a mano y comparar con el código máquina que se muestra en la tabla. Estamos suponiendo que **miPalabra** empieza en el desplazamiento 0102h.

Tabla 17-28 Ejemplos de instrucciones MOV, con código máquina.

Instrucción	Código máquina	Modo de direccionamiento
mov ax,miPalabra	A1 02 01	directo (optimizado para AX)
mov miPalabra,bx	89 1E 02 01	directo
mov [di],bx	89 ID	Indexado
mov [bx+2],ax	89 47 02	base-desp
mov [bx+si].ax	89 00	base-indexado
mov word ptr [bx+di+2],1234h	C7 41 02 34 12	base-indexado-desp

17.3.7 Repaso de sección

1. Proporcione los códigos de operación para las siguientes instrucciones MOV:

```
.data
miByte    BYTE ?
miPalabra WORD ?
.code
mov  ax,@data
mov  ds,ax
mov  ax,bx
mov  bl,al
mov  al,[si]
mov  miByte,al
mov  miPalabra,ax
```

; a.
; b.
; c.
; d.
; e.
; f.

2. Proporcione los códigos de operación para las siguientes instrucciones MOV:

```
.data
miByte    BYTE ?
miPalabra WORD ?
.code
mov  ax,@data
mov  ds,ax
mov  es,ax
mov  d1,b1
mov  b1,[di]
mov  ax,[si+2]
mov  al,miByte
mov  dx,miPalabra
```

; a.
; b.
; c.
; d.
; e.
; f.

3. Proporcione los bytes Mod R/M para las siguientes instrucciones MOV:

```
.data
arreglo WORD 5 DUP(?)
.code
mov  ax,@data
mov  ds,ax
mov  d1,b1
mov  b1,[di]
mov  ax,[si+2]
mov  ax,arreglo[si]
mov  arreglo[di],ax
```

; a.
; b.
; c.
; d.
; e.
; f.

4. Proporcione los bytes Mod R/M para las siguientes instrucciones MOV:

```
.data
arreglo WORD 5 DUP(?)
.code
mov ax,@data
mov ds,ax
mov BYTE PTR arreglo,5      ; a.
mov dx,[bp+5]                ; b.
mov [di],bx                  ; c.
mov [di+2],dx                ; d.
mov arreglo[si+2],ax          ; e.
mov arreglo[bx+di],ax          ; f.
```

5. Ensamble las siguientes instrucciones a mano y escriba los bytes de lenguaje máquina hexadecimales para cada instrucción etiquetada. Suponga que **val1** se encuentra en el desplazamiento 0. En donde se utilicen valores de 16 bits, los bytes deberán aparecer en orden little endian:

```
.data
val1 BYTE 5
va12 WORD 256
.code
mov ax,@data
mov ds,ax
mov al,val1                 ; a.
mov cx,va12                  ; b.
mov dx,OFFSET val1           ; c.
mov dl,2                     ; d.
mov bx,1000h                  ; e.
                                ; f.
```

17.4 Resumen del capítulo

Un número binario de punto flotante contiene tres componentes: un signo, una mantisa y un exponente. Los procesadores Intel utilizan formatos de almacenamiento binario de punto flotante especificados en el estándar 754-1985 para Aritmética binaria de punto flotante, producido por la organización IEEE:

- Un valor de 32 bits con precisión simple utiliza 1 bit para el signo, 8 bits para el exponente, y 23 bits para la parte fraccional de la mantisa.
- Un valor de 64 bits con precisión doble utiliza 1 bit para el signo, 11 bits para el exponente, y 52 bits para la parte fraccional de la mantisa.
- Un valor de 80 bits con precisión doble extendida utiliza 1 bit para el signo, 16 bits para el exponente, y 63 bits para la parte fraccional de la mantisa.

Si el bit de signo es igual a 1, el número es negativo; si el bit es 0, el número es positivo.

La mantisa de un número de punto flotante consiste en los dígitos decimales a la izquierda y derecha del punto decimal.

No todos los números reales entre 0 y 1 pueden representarse mediante números de punto flotante en una computadora, ya que sólo hay un número finito de bits disponibles.

Los números finitos normalizados son todos los valores finitos distintos de cero que pueden codificarse en un número real normalizado, entre cero e infinito. El infinito positivo ($+\infty$) representa el máximo número real positivo, y el infinito negativo ($-\infty$) representa el máximo número real negativo. Los *Nans* son patrones de bits que no representan números de punto flotante válidos.

El procesador Intel 8086 se diseñó para manejar sólo la aritmética con enteros, por lo que Intel produjo un chip *coprocesador de punto flotante* 8087 separado, que se insertaba en la tarjeta madre de la computadora, junto con el 8086. Con la llegada del Intel486, se integraron las operaciones de punto flotante en la CPU principal y se le cambió el nombre a *Unidad de punto flotante* (FPU).

La FPU tiene ocho registros de 80 bits que pueden direccionarse por separado, llamados R0 a R7 y ordenados en forma de una pila de registros. Los operandos de punto flotante se almacenan en la pila de la FPU en formato real extendido, mientras se utilizan en los cálculos. También se utilizan operandos de memoria en los cálculos. Cuando la FPU almacena el resultado de una operación aritmética en memoria, traduce el resultado a uno de los siguientes formatos: entero, entero largo, precisión simple, precisión doble o decimal codificado en binario.

Los nemáticos de las instrucciones Intel de punto flotante empiezan con la letra F para distinguirlas de las instrucciones de la CPU. La segunda letra de una instrucción (comúnmente, B o I) indica cómo se va a interpretar un operando de memoria: B indica un operando decimal codificado en binario (BCD), y I indica un operando entero binario. Si no se especifica ninguna de las dos, se asume que el operando de memoria está en formato de número real.

El procesador 8086 fue el primero de una línea de procesadores que utilizan un diseño de *Computadora con un conjunto complejo de instrucciones* (CISC). El conjunto de instrucciones es extenso e incluye una amplia variedad de operaciones de direccionamiento de memoria, de desplazamiento, aritméticas, de movimiento de datos y lógicas.

Codificar una instrucción significa convertir una instrucción en lenguaje ensamblador y sus operandos en código máquina. *Decodificar* una instrucción significa convertir una instrucción en código máquina a una instrucción en lenguaje ensamblador y sus operandos.

El formato de instrucciones de máquina de la familia IA-32 contiene un byte de prefijo opcional, un código de operación, un byte Mod R/M opcional, bytes inmediatos opcionales, y bytes de desplazamiento de memoria opcionales. Pocas instrucciones contienen todos los campos. El byte de prefijo redefine el tamaño de operando predeterminado para el procesador de destino. El byte de código de operación contiene el código de operación único de la instrucción. El campo Mod R/M identifica el modo de direccionamiento y los operandos. En las instrucciones que utilizan operandos tipo registro, el byte Mod R/M contiene un identificador de 3 bits para cada operando tipo registro.

17.5 Ejercicios de programación

1. Comparación de punto flotante

Implemente el siguiente código de C++ en lenguaje ensamblador. Sustituya las llamadas a WriteString por las llamadas a la función printf():

```
double X;
double Y;
if( X < Y )
    printf("X es menor\n");
else
    printf("X no es menor\n");
```

Use las rutinas de la biblioteca Irvine32 para la salida de consola, en vez de llamar a la función printf de la biblioteca estándar de C. Ejecute el programa varias veces, asignando un rango de valores a X y Y para probar la lógica de su programa.

2. Mostrar binario de punto flotante

Escriba un procedimiento que reciba un valor binario de punto flotante con precisión simple y lo muestre en el siguiente formato: signo: mostrar + o -; mantisa: binario de punto flotante, con el prefijo “1.”; exponente: mostrarlo en decimal, sin desviación, precedido por la letra E y el signo del exponente. Ejemplo:

```
.data
Ejemplo REAL4 -1.75
```

Resultado en pantalla:

```
-1.11000000000000000000000000000000 E+0
```

3. Establecer los modos de redondeo

Requiere conocimiento sobre las macros. Escriba una macro para establecer el modo de redondeo de la FPU. El único parámetro de entrada es un código de dos letras:

- RE: redondea al número par más cercano.
- RD: redondea hacia infinito negativo.
- RU: redondea hasta infinito positivo.
- RZ: redondea hacia cero (truncar).

Ejemplo de llamadas a la macro (no debe importar el uso de mayúsculas y minúsculas):

```
mRedondeo Re
mRedondeo rd
mRedondeo RU
mRedondeo rZ
```

Escriba un programa corto de prueba que utilice la instrucción FIST (guardar entero) para probar cada uno de los posibles modos de redondeo.

4. Evaluación de expresiones

Escriba un programa para evaluar la siguiente expresión aritmética:

$$((A + B) / C) * ((D - A) + E)$$

Asigne valores de prueba a las variables y muestre el valor resultante.

5. Área de un círculo

Escriba un programa que pida al usuario el radio de un círculo. Calcule y muestre en pantalla el área del círculo. Use los procedimientos ReadFloat y WriteFloat de la biblioteca del libro. Use la instrucción FLDPI para cargar a π en la pila de registros.

6. Fórmula cuadrática

Pida al usuario los coeficientes a, b y c de un polinomio en la forma $ax^2 + bx + c$. Calcule y muestre las raíces reales del polinomio, usando la *fórmula cuadrática*. Si cualquier raíz es imaginaria, muestre un mensaje apropiado.

7. Mostrar valores de estado de los registros

El registro de Etiqueta (sección 17.2.1) indica el tipo de contenido en cada registro de la FPU, usando 2 bits para cada uno (figura 17-7). Puede cargar la palabra Etiqueta llamando a la instrucción FSTENV, la cual llena la siguiente estructura en modo protegido (definida en *Irvine32.inc*):

```
FPU_ENVIRON STRUCT
    controlWord     WORD ?
    ALIGN DWORD
    statusWord      WORD ?
    ALIGN DWORD
    tagWord         WORD ?
    ALIGN DWORD
    instrPointerOffset   DWORD ?
    instrPointerSelector  DWORD ?
    operandPointerOffset  DWORD ?
    operandPointerSelector WORD ?
    WORD ?           ; no se utiliza
FPU_ENVIRON ENDS
```

Hay una estructura con el mismo nombre definida en *Irvine16.inc*, con un formato ligeramente distinto para la programación en modo de direccionamiento real.

Escriba un programa que meta dos o más valores en la pila de la FPU, muestre la pila llamando a ShowFPUStack, muestre el valor de Etiqueta de cada registro de datos de la FPU, y muestre el número de registro correspondiente a ST(0). Para este último, llame a la instrucción FSTSW para guardar la palabra de estado en una variable entera de 16 bits, y extraiga el indicador TOP de la pila de los bits 11 al 13. Use la siguiente salida de ejemplo como guía:

```
----- FPU Stack -----
ST(0): +1.5000000E+000
ST(1): +2.0000000E+000

R0  esta vacio
R1  esta vacio
R2  esta vacio
R3  esta vacio
R4  esta vacio
R5  esta vacio
R6  es valido
R7  es valido

ST(0) = R6
```

De los resultados de ejemplo podemos ver que ST(0) es R6 y, por lo tanto, ST(1) es R7. Ambos contienen números de punto flotante válidos.

FIGURA 17-7 Valores de la palabra Etiqueta.

15	R7	R6	R5	R4	R3	R2	R1	R0	0
Valores de ETIQUETA:									
00 = válido									
01 = cero									
10 = especial (Nan, no soportado, infinito o denormalizado)									
11 = vacío									

Notas finales

1. *Manual del desarrollador de software para la arquitectura Intel IA-32*, Vol. 1, Capítulo 4. Vea también www.grouper.ieee.org/groups/754/
2. *Manual del desarrollador de software para la arquitectura Intel IA-32*, Vol. 1, Sección 4.8.3.
3. De Harvey Nice, de la Universidad DePaul.
4. MASM usa una instrucción FADD sin parámetros para realizar la misma operación que la instrucción FADDP sin parámetros de Intel.
5. MASM usa una instrucción FSUB sin parámetros para realizar la misma operación que la instrucción FSUBP sin parámetros de Intel.
6. MASM usa una instrucción FMUL sin parámetros para realizar la misma operación que la instrucción FMULP sin parámetros de Intel.
7. MASM usa una instrucción FDIV sin parámetros para realizar la misma operación que la instrucción FDIVP sin parámetros de Intel.

A

REFERENCIA DE MASM

A.1 Introducción

A.2 Palabras reservadas de MASM

A.3 Nombres de registros

A.4 Microsoft Assembler (ML)

A.5 Directivas de MASM

A.6 Símbolos

A.7 Operadores

A.8 Operadores en tiempo de ejecución

A.1 Introducción

Los manuales de Microsoft MASM 6.11 se imprimieron por última vez en 1992, y consistían en tres volúmenes:

- *Guía del programador.*
- *Referencia.*
- *Entorno y herramientas.*

Por desgracia, estos manuales ya no están disponibles desde hace muchos años, pero Microsoft provee copias electrónicas de ellos (archivos de MS-Word) en su paquete *SDK de la plataforma*. En definitiva, los manuales son artículos de colección.

La información en este capítulo se tomó de los capítulos 1 a 3 del manual de *Referencia (Reference)*, con actualizaciones del archivo *leame.txt* (*readme.txt*) de MASM 6.14.

Notación de sintaxis A lo largo de este apéndice utilizamos una notación de sintaxis consistente. Las palabras en letras mayúsculas indican una palabra reservada de MASM que puede aparecer en nuestro programa, ya sea en mayúsculas o en minúsculas. En el siguiente ejemplo, DATA es una palabra reservada:

.DATA

Las palabras en cursiva indican un término o categoría definidos. En el siguiente ejemplo, *número* se refiere a una constante entera:

ALIGN [[*número*]]

Cuando se utilizan dobles corchetes `[[..]]` para encerrar a un elemento, significa que éste es opcional. En el siguiente ejemplo, *texto* es opcional:

`[[texto]]`

Cuando aparece un separador vertical `|` entre los elementos en una lista de dos o más elementos, debemos seleccionar uno de ellos. El siguiente ejemplo indica una elección entre NEAR y FAR:

`NEAR | FAR`

Tres puntos suspensivos `(. . .)` indican la repetición del último elemento en una lista. En el siguiente ejemplo, la coma que va después de un *inicializador* debe repetirse varias veces:

`[[nombre]] BYTE inicializador [[, inicializador]] . . .`

A.2 Palabras reservadas de MASM

\$	PARITY?
?	PASCAL
@B	QWORD
@F	REAL4
ADDR	REAL8
BASIC	REAL10
BYTE	SBYTE
C	SDWORD
CARRY?	SIGN?
DWORD	STDCALL
FAR	SWORD
FAR16	SYSCALL
FORTRAN	TBYTE
FWORD	VARARG
NEAR	WORD
NEAR16	ZERO?
OVERFLOW?	

A.3 Nombres de los registros

AH	CR0	DR1	EBX	SI
AL	CR2	DR2	ECX	SP
AX	CR3	DR3	EDI	SS
BH	CS	DR6	EDX	ST
BL	CX	DR7	ES	TR3
BP	DH	DS	ESI	TR4
BX	DI	DX	ESP	TR5
CH	DL	EAX	FS	TR6
CL	DR0	EBP	GS	TR7

A.4 Microsoft Assembler (ML)

El programa ML (*ML.EXE*) ensambla uno o más archivos de código fuente en lenguaje ensamblador. La sintaxis es:

```
ML [[ opciones ]] nombrearchivo [[ [[ opciones ]] nombrearchivo ]] . . . [[ /link opcionesenlazador ]]
```

El único parámetro requerido es por lo menos un *nombrearchivo*, el nombre de un archivo de código fuente escrito en lenguaje ensamblador. Por ejemplo, el siguiente comando ensambla el archivo de código fuente *SumaResta.asm* y produce el archivo de objeto *SumaResta.obj*:

```
ML -c SumaResta.asm
```

El parámetro *opciones* consiste en cero o más opciones de la línea de comandos, cada uno de los cuales inicia con una barra diagonal (/) o guión corto (-). Si se usan varias opciones, deben separarse por un espacio cuando menos. La tabla A-1 lista el conjunto completo de opciones de la línea de comandos. Estas opciones son sensibles al uso de mayúsculas y minúsculas.

Tabla A-1 Opciones de la línea de comandos de ML.

Opción	Acción
/AT	Habilita el soporte para el modelo diminuto (tiny) de memoria. Habilita los mensajes de error para las instrucciones de código que violan los requerimientos de los archivos de formato .COM. Observe que esto no es equivalente a la directiva .MODEL TINY
/B! <i>nombrearchivo</i>	Selecciona un enlazador alternativo
/c	Sólo ensambla, sin vincular
/coff	Genera un archivo de código objeto en <i>Formato de archivo de objetos comunes de Microsoft</i>
/Cp	Preserva el uso de mayúsculas y minúsculas de todos los identificadores del usuario
/Cu	Asigna todos los identificadores a letras mayúsculas
/Cx	Preserva el uso de mayúsculas y minúsculas en los símbolos públicos y externos (predeterminado)
/D <i>símbolo</i> [= <i>valor</i>]	Define una macro de texto con el nombre dado. Si se omite <i>valor</i> , se deja en blanco. Varios símbolos (tokens) separados por espacios deben encerrarse entre comillas
/EP	Genera un listado de código fuente preprocesado (lo envía a STDOUT). Vea /Sf
/Fn <i>númhex</i>	Establece el tamaño de la pila a <i>númhex</i> bytes (es lo mismo que /link /STACK <i>número</i>). El valor debe expresarse en notación hexadecimal. Debe haber un espacio entre /F y <i>númhex</i>
/F! <i>nombrearchivo</i>	Nombra el archivo ejecutable
/Fl[[<i>nombrearchivo</i>]]	Genera un listado de código ensamblado. Vea /Sf
/Fm[[<i>nombrearchivo</i>]]	Crea un archivo de mapa del enlazador
/Fo <i>nombrearchivo</i>	Nombra a un archivo de objeto
/FPi	Genera correcciones del emulador para la aritmética de punto flotante (sólo en lenguaje mixto)
/Fr[[<i>nombrearchivo</i>]]	Genera un archivo del Explorador de origen (.SBR)
/FR[[<i>nombrearchivo</i>]]	Genera una forma extendida de un archivo .SBR (Explorador de origen)
/Gc	Especifica el uso de las convenciones de nomenclatura y de llamadas a funciones estilo FORTRAN o Pascal

Opción	Acción
/Gd	Especifica el uso de las convenciones de nomenclatura y de llamadas a funciones estilo C
/Gz	Usa las conexiones de llamadas de STDCALL
/H <i>número</i>	Restringe los nombres externos a una cantidad de caracteres significativos determinada por <i>número</i> . El valor predeterminado es 31 caracteres
/help	Llama a QuickHelp para obtener ayuda sobre ML
/I <i>nombreruta</i>	Establece la ruta para un archivo de inclusión. Se permite un máximo de 10 opciones /I
/link	Opciones y bibliotecas del enlazador
/nologo	Suprime los mensajes para un ensamblado con éxito
/omf	Genera un archivo OMF (Formato de módulo de objetos de Microsoft). Este formato es requerido por el Microsoft Linker de 16 bits anterior (LINK16.EXE)
/Sa	Activa el listado de toda la información disponible
/Sc	Agrega los tiempos de las instrucciones al archivo de listado
/Sf	Agrega el listado de la primera pasada a un archivo de listado
/Sg	Hace que el código ensamblador generado por MASM aparezca en el archivo de listado de código fuente. Por ejemplo, utilice esta opción si desea ver cómo funcionan las directivas .IF y .ELSE
/SI <i>anchura</i>	Establece la anchura de línea del listado de código fuente, en caracteres por línea. El rango es de 60 a 255 o 0. El valor predeterminado es 0. Igual que PAGE <i>anchura</i>
/Sn	Desactiva la tabla de símbolos al producir un listado
/Sp <i>longitud</i>	Establece la longitud de página del listado de código fuente, en líneas por página. El rango es de 10 a 255 o 0. El valor predeterminado es 0. Igual que PAGE <i>longitud</i>
/Ss <i>texto</i>	Especifica texto para el listado de código fuente. Igual que SUBTITLE <i>texto</i>
/St <i>texto</i>	Especifica el título para el listado de código fuente. Igual que TITLE <i>texto</i>
/Sx	Activa las condicionales falsas en el listado
/Ta <i>nombrearchivo</i>	Ensambla un archivo de código fuente cuyo nombre no termina con la extensión .ASM
/w	Igual que /W0
/Wnivel	Establece el nivel de advertencias, en donde <i>nivel</i> = 0, 1, 2 o 3
/WX	Devuelve un código de error si se generan advertencias
/X	Ignora la ruta de entorno de INCLUDE
/Zd	Genera información de número de línea en el archivo de código objeto
/Zf	Hace públicos a todos los símbolos
/Zi	Genera información de CodeView en un archivo de código objeto
/Zm	Habilita la opción M510 para máxima compatibilidad con MASM 5.1
/Zp[<i>alineación</i>]]	Empaquesta estructuras en el límite de byte especificado. La <i>alineación</i> puede ser 1, 2 o 4
/Zs	Sólo realiza una comprobación de sintaxis
/?	Muestra un resumen de la sintaxis de la línea de comandos de ML
/error Reporte	Reporta los errores internos del ensamblador a Microsoft

A.5 Directivas de MASM

nombre = expresión

Asigna el valor numérico de *expresión* a *nombre*. El símbolo puede redefinirse después.

.186

Habilita el ensamblado de instrucciones para el procesador 80186; deshabilita el ensamblado de instrucciones que se introdujeron con los procesadores posteriores. También habilita las instrucciones del 8087.

.286

Habilita el ensamblado de instrucciones no privilegiadas para el procesador 80286; deshabilita el ensamblado de instrucciones que se introdujeron con los procesadores posteriores. También habilita las instrucciones del 80287.

.286P

Habilita el ensamblado de todas las instrucciones (incluyendo las privilegiadas) para el procesador 80286; deshabilita el ensamblado de instrucciones que se introdujeron con los procesadores posteriores. También habilita las instrucciones del 80287.

.287

Habilita el ensamblado de instrucciones para el coprocesador 80287; deshabilita el ensamblado de instrucciones que se introdujeron con los coprocesadores posteriores.

.386

Habilita el ensamblado de instrucciones no privilegiadas para el procesador 80386; deshabilita el ensamblado de instrucciones que se introdujeron con los procesadores posteriores. También habilita las instrucciones del 80387.

.386P

Habilita el ensamblado de todas las instrucciones (incluyendo las privilegiadas) para el procesador 80386; deshabilita el ensamblado de instrucciones que se introdujeron con los procesadores posteriores. También habilita las instrucciones del 80387.

.387

Habilita el ensamblado de instrucciones para el coprocesador 80387.

.486

Habilita el ensamblado de instrucciones no privilegiadas para el procesador 80486.

.486P

Habilita el ensamblado de todas las instrucciones (incluyendo las privilegiadas) para el procesador 80486.

.586

Habilita el ensamblado de instrucciones no privilegiadas para el procesador Pentium.

.586P

Habilita el ensamblado de todas las instrucciones (incluyendo las privilegiadas) para el procesador Pentium.

.686

Habilita el ensamblado de instrucciones no privilegiadas para el procesador Pentium Pro.

.686P

Habilita el ensamblado de todas las instrucciones (incluyendo las privilegiadas) para el procesador Pentium Pro.

.8086

Habilita el ensamblado de instrucciones para el 8086 (y las instrucciones idénticas para el 8088); deshabilita el ensamblado de instrucciones que se introdujeron con los procesadores posteriores. También habilita las instrucciones del 8087. Éste es el modo predeterminado para los procesadores.

.8087

Habilita el ensamblado de instrucciones para el 8087; deshabilita el ensamblado de instrucciones que se introdujeron con los procesadores posteriores. Éste es el modo predeterminado para los coprocesadores.

ALIAS <alias> = <nOMBRE-actual>

Asigna el nombre anterior de una función a un nuevo nombre. *Alias* es el nombre alternativo o alias, y *nOMBRE-actual* es el nombre actual de la función o procedimiento. Los signos < > son obligatorios. La

directiva ALIAS puede usarse para crear bibliotecas que permitan al enlazador (LINK) asignar una función anterior a una nueva función.

ALIGN [[número]]

Alinea la siguiente variable o instrucción en un byte que sea un múltiplo de *número*.

.ALPHA

Ordena los segmentos alfabéticamente.

ASSUME registroseg: nombre [, registroseg:nombre] ...

ASSUME registrodatos:tipo [, registrodatos:tipo] ...

ASSUME registro:ERROR [, registro:ERROR] ...

ASSUME [[registro:]] NOTHING [, registro:NOTHING] ...

Habilita la comprobación de errores para los valores de los registros. Después de que una directiva ASSUME entra en efecto, el ensamblador vigila en caso de que haya cambios a los valores de los registros dados. ERROR genera un error si se utiliza el registro. NOTHING elimina la comprobación de errores de los registros. Podemos combinar distintos tipos de directivas ASSUME en una instrucción.

.BREAK [.IF condición]

Genera código para terminar un bloque .WHILE o .REPEAT si *condición* es verdadera.

[[nombre]] BYTE inicializador [, inicializador] ...

Asigna e inicializa de manera opcional un byte de almacenamiento para cada *inicializador*. También puede usarse como un especificador de tipo, en cualquier parte en que un tipo sea legal.

nombre CATSTR [[elementotexto1 [, elementotexto2]] ...]

Concatena elementos de texto. Cada elemento de texto puede ser una cadena literal, una constante precedida por un % o la cadena devuelta por una macrofunción.

.CODE [[nombre]]

Al utilizarse con .MODEL, indica el inicio de un segmento de código llamado *nombre* (el nombre de segmento predeterminado es _TEXT para los modelos diminuto, pequeño, compacto y plano, o *módulo*_TEXT para los demás modelos).

COMM definición [, definición] ...

Crea una variable comunal con los atributos especificados en *definición*. Cada *definición* tiene la siguiente forma:

[[tipolenguaje]] [[NEAR | FAR]] etiqueta:tipo [:cuenta]

La *etiqueta* es el nombre de la variable. El *tipo* puede ser cualquier especificador de tipo (BYTE, WORD, etcétera) o un entero que especifique el número de bytes. La *cuenta* especifica el número de objetos de datos (uno es el predeterminado).

COMMENT delimitador [texto]

[[texto]]

[[texto]] delimitador [texto]

Trata a todo el *texto* entre, o en la misma línea que los delimitadores, como si fuera un comentario.

.CONST

Al utilizarse con .MODEL, inicia un segmento de datos constante (con el nombre de segmento CONST). Este segmento tiene el atributo de sólo lectura.

.CONTINUE [.IF condición]

Genera código para saltar a la parte superior de un bloque .WHILE o .REPEAT si *condición* es verdadera.

.CREF

Habilita el listado de símbolos en la porción correspondiente al símbolo de la tabla de símbolos y el archivo del explorador.

.DATA

Al utilizarse con .MODEL, inicia un segmento de datos cercano para los datos inicializados (nombre de segmento _DATA).

.DATA?

Al utilizarse con **.MODEL**, inicia un segmento de datos cercano para los datos sin inicializar (nombre de segmento _BSS).

.DOSSEG

Ordena los segmentos de acuerdo con la convención de segmentos de MS-DOS: CODE primero, después los segmentos que no estén en DGROUP, y luego los segmentos en DGROUP. Los segmentos en DGROUP siguen este orden: los segmentos que no estén en BSS o STACK, después el segmento BSS, y por último los segmentos STACK. Se utiliza sobre todo para asegurar el soporte de CodeView en los programas de MASM independientes. Es igual que **DOSSEG**.

DOSSEG

Idéntica a **.DOSSEG**, que es la forma que se prefiere.

DB

Puede usarse para definir datos como **BYTE**.

DD

Puede usarse para definir datos como **DWORD**.

DF

Puede usarse para definir datos como **FWORD**.

DQ

Puede usarse para definir datos como **QWORD**.

DT

Puede usarse para definir datos como **TBYTE**.

DW

Puede usarse para definir datos como **WORD**.

[[nombre]] DWORD inicializador [, inicializador]]...

Asigna e inicializa de manera opcional una doble palabra (4 bytes) de almacenamiento para cada *inicializador*. También puede usarse como especificador de tipo, en cualquier parte en donde un tipo sea legal.

ECHO mensaje

Muestra *mensaje* en el dispositivo de salida estándar (de manera predeterminada, la pantalla). Es igual que **%OUT**.

.ELSE

Vea **.IF**.

ELSE

Marca el inicio de un bloque alternativo dentro de un bloque condicional. Vea **IF**.

ELSEIF

Combina a **ELSE** e **IF** en una instrucción. Vea **IF**.

ELSEIF2

Bloque **ELSEIF** que se evalúa en cada pasada del ensamblador, si **OPTION:SETIF2** es **TRUE**.

END [[dirección]]

Marca el final de un módulo y, de manera opcional, establece el punto de entrada del programa a *dirección*.

.ENDIF

Vea **.IF**

ENDIF

Vea **IF**.

ENDM

Termina una macro o bloque de repetición. Vea **MACRO**, **FOR**, **FORC**, **REPEAT** o **WHILE**.

nombre ENDP

Marca el final del procedimiento *nombre*, que se empezó antes con **PROC**. Vea **PROC**.

nombre ENDS

Marca el final de un segmento, estructura o *nombre* de unión que haya empezado antes con **SEGMENT**, **STRUCT**, **UNION** o una directiva de segmento simplificada.

.ENDW

Vea **.WHILE**.

nombre EQU expresión

Asigna el valor numérico de *expresión* a *nombre*. El *nombre* no puede redefinirse después.

nombre EQU <texto>

Asigna el *texto* especificado a *nombre*. A este *nombre* se le puede asignar un *texto* distinto después. Vea **TEXTEQU**.

.ERR [[mensaje]]

Genera un error.

.ERR2 [[mensaje]]

Bloque **.ERR** que se evalúa en cada pasada del ensamblador si **OPTION:SETIF2** es **TRUE**.

.ERRB <elementotexto> [[, mensaje]]

Genera un error si *elementotexto* está en blanco.

.ERRDEF nombre [[, mensaje]]

Genera un error si *nombre* es una etiqueta, variable o símbolo que se definió antes.

ERRDIF[[I]] <elementotexto1>, <elementotexto2> [[, mensaje]]

Genera un error si los elementos de texto son distintos. Si se da **I**, la comparación es insensible al uso de mayúsculas y minúsculas.

.ERRE expresión [[, mensaje]]

Genera un error si *expresión* es falsa (0).

.ERRIDN[[I]] <elementotexto1>, <elementotexto2> [[, mensaje]]

Genera un error si los elementos de texto son idénticos. Si se da **I**, la comparación es insensible al uso de mayúsculas y minúsculas.

.ERRNB <elementotexto> [[, mensaje]]

Genera un error si *elementotexto* no está en blanco.

.ERRNDEF nombre [[, mensaje]]

Genera un error si no se ha definido *nombre*.

.ERRNZ expresión [[, mensaje]]

Genera un error si *expresión* es verdadera (distinta de cero).

EVEN

Alinea la siguiente variable o instrucción con un byte par.

.EXIT [[expresión]]

Genera el código de terminación. Devuelve la *expresión* opcional a la interfaz del sistema (shell).

EXITM [[elementotexto]]

Termina la expansión del bloque actual de repetición o de macro, y empieza a ensamblar la siguiente instrucción fuera del bloque. En una macro función, *elementotexto* es el valor que se devuelve.

EXTERN [[tipolenguaje]] nombre [(idalt)] :tipo [[, [[tipolenguaje]] nombre [(idalt)] :tipo]...]

Define una o más variables externas, etiquetas o símbolos que llamaron a *nombre*, cuyo tipo es *tipo*. El *tipo* puede ser **ABS**, que importa a *nombre* como una constante. Es igual que **EXTRN**.

EXTERNDEF [[tipolenguaje]] nombre:tipo [[, [[tipolenguaje]] nombre:tipo]...]

Define a una o más variables externas, etiquetas o símbolos llamados *nombre*, cuyo tipo sea *tipo*. Si *nombre* ya está definido en el módulo, se trata como **PUBLIC**. Si se hace referencia a *nombre* en el módulo, se trata como **EXTERN**. Si no se hace referencia a *nombre*, se ignora. El *tipo* puede ser **ABS**, que importa a *nombre* como una constante. Por lo general, se utiliza en los archivos de inclusión.

EXTRN

Vea **EXTERN**.

.FARDATA [[*nombre*]]

Al utilizarse con **.MODEL**, inicia un segmento de datos lejano para los datos inicializados (nombre de segmento FAR_DATA o *nombre*).

.FARDATA?[[*nombre*]]

Al utilizarse con **.MODEL**, inicia un segmento de datos lejano para los datos sin inicializar (nombre de segmento FAR_BSS o *nombre*).

FOR parámetro [[:REQ | :=default]],<argumento [[, argumento]]...>

instrucciones

ENDM

Marca un bloque que se repetirá una vez para cada *argumento*, en donde el *argumento* actual sustituirá a *parámetro* en cada repetición. Igual que **IRP**.

FORC

parámetro,<cadena> instrucciones

ENDM

Marca un bloque que se repetirá una vez para cada carácter en una *cadena*, en donde el carácter actual sustituirá a *parámetro* en cada repetición. Igual que **IRPC**.

[[*nombre*]] FWORD *inicializador* [[, *inicializador*]...]

Asigna e inicializa de manera opcional 6 bytes de almacenamiento para cada *inicializador*. También puede utilizarse como especificador de tipo, en cualquier parte en donde un tipo sea legal.

GOTO *etiquetamacro*

Las transferencias se ensamblan en la línea marcada *:etiquetamacro*. **GOTO** se permite sólo dentro de bloques **MACRO**, **FOR**, **FORC**, **REPEAT** y **WHILE**. La etiqueta debe ser la única directiva en la línea y debe ir precedida por un signo de dos puntos.

***nombre* GROUP *segmento* [[, *segmento*]...]**

Agrega los *segmentos* especificados al grupo llamado *nombre*. Esta directiva no tiene efecto cuando se utiliza en la programación en el modelo plano de 32 bits, y producirá un error si se utiliza con la opción de línea de comandos /coff.

.IF *condición1*

instrucciones

[.ELSEIF *condición2*

instrucciones]]

[.ELSE

instrucciones]]

.ENDIF

Genera código para evaluar la *condición1* (por ejemplo, AX > 7) y ejecuta las *instrucciones* si esa condición es verdadera. Si le sigue una directiva **.ELSE**, sus instrucciones se ejecutan si la condición original era falsa. Observe que las condiciones se evalúan en tiempo de ejecución.

IF *expresión1*

instruccionesif

[ELSEIF *expresión2*

instruccioneselseif]]

[ELSE

instruccioneselse]]

.ENDIF

Permite el ensamblado de *instruccionesif* si *expresión1* es verdadera (distinta de cero), o de *instruccioneselseif* si *expresión1* es falsa(0) y *expresión2* es verdadera. Las siguientes directivas pueden sustituirse por **ELSEIF**: **ELSEIFB**, **ELSEIFDEF**, **ELSEIFDIF**, **ELSEIFDIFI**, **ELSEIFE**, **ELSEIFIDN**,

ELSEIFIDNI, ELSEIFNB y ELSEIFNDEF. De manera opcional, ensambla *instruccioneselse* si la expresión anterior es falsa. Observe que las expresiones se evalúan en tiempo de ensamblado.

IF2 expresión

El bloque **IF** se evalúa en cada pasada del ensamblador si **OPTION:SETIF2** es **TRUE**. Vea **IF** para la sintaxis completa.

IFB elementotexto

Permite el ensamblado si *elementotexto* está en blanco. Vea **IF** para la sintaxis completa.

IFDEF nombre

Permite el ensamblado si *nombre* es una etiqueta, variable o símbolo que se haya definido antes. Vea **IF** para la sintaxis completa.

IFDIF[I**] elementotexto1, elementotexto2**

Permite el ensamblado si los elementos de texto son distintos. Si se incluye **I**, la comparación es insensible a mayúsculas y minúsculas. Vea **IF** para una sintaxis completa.

IFE expresión

Permite el ensamblado si *expresión* es falsa (0). Vea **IF** para la sintaxis completa.

IFIDN[I**] elementotexto1, elementotexto2**

Permite el ensamblado si los elementos de texto son idénticos. Si se incluye **I**, la comparación es insensible a mayúsculas y minúsculas. Vea **IF** para la sintaxis completa.

IFNB elementotexto

Permite el ensamblado si *elementotexto* no está en blanco. Vea **IF** para la sintaxis completa.

IFNDEF nombre

Permite el ensamblado si *nombre* no se ha definido. Vea **IF** para la sintaxis completa.

INCLUDE nombrearchivo

Inserta código fuente del archivo de código fuente que proporciona *nombrearchivo* en el archivo de código fuente actual durante el ensamblado. El *nombrearchivo* debe ir encerrado entre los signos < y > si incluye una barra diagonal inversa, punto y coma, símbolo mayor que, menor que, comilla sencilla o comilla doble.

INCLUDELIB nombrebiblioteca

Informa al enlazador que el módulo actual debe incluirse con *nombrebiblioteca*. El *nombrebiblioteca* debe ir encerrado entre los signos < y > si incluye una barra diagonal inversa, punto y coma, símbolo mayor que, menor que, comilla sencilla o comilla doble.

nombre INSTR [posición,] elementotexto1, elementotexto2

Encuentra la primera ocurrencia de *elementotexto2* en *elementotexto1*. La *posición* inicial es opcional. Cada elemento de texto puede ser una cadena literal, una constante precedida por un %, o la cadena devuelta por una macro función.

Invoke expresión [, argumentos]

Llama al procedimiento en la dirección proporcionada por *expresión*, y le pasa los argumentos en la pila o en registros, de acuerdo con las convenciones de llamadas estándar del tipo de lenguaje. Cada argumento que se pasa al procedimiento puede ser una expresión, un par de registros, o una expresión de dirección (precedida por **ADDR**).

IRP

Vea **FOR**.

IRPC

Vea **FORC**.

nombre LABEL tipo

Crea una nueva etiqueta, asignando el valor del contador de la ubicación actual y el *tipo* dado a *nombre*.

nombre LABEL [NEAR | FAR | PROC] PTR [tipo]

Crea una nueva etiqueta, asignándole el valor del contador de la ubicación actual y el *tipo* dado a *nombre*.

.K3D

Permite el ensamblado de instrucciones K3D.

.LALL

Vea **.LISTMACROALL**

.LFCOND

Vea **.LISTIF**.

.LIST

Inicia el listado de instrucciones. Es la opción predeterminada.

.LISTALL

Inicia el listado de todas las instrucciones. Es equivalente a la combinación de **.LIST**, **.LISTIF** y **.LISTMACROALL**.

.LISTIF

Inicia el listado de instrucciones en bloques condicionales falsos. Igual que **.LFCOND**.

.LISTMACRO

Inicia el listado de instrucciones de expansión de macros que generan código o datos. Es la opción predeterminada. Igual que **.XALL**.

.LISTMACROALL

Inicia el listado de todas las instrucciones en las macros. Igual que **.LALL**.

LOCAL *nombrelocal* [[, *nombrelocal*]...]

Dentro de una macro, **LOCAL** define etiquetas que son únicas para cada instancia de la macro.

LOCAL *etiqueta* [[:*cuenta*] [[:*tipo*] [[:*etiqueta*] [[:*cuenta*] [[:*tipo*]]]...]

Dentro de una definición de un procedimiento (**PROC**), **LOCAL** crea variables basadas en la pila que existen mientras dure el procedimiento. La *etiqueta* puede ser una variable simple o un arreglo que contenga elementos *cuenta*.

nombre MACRO [[:*parámetro* [[:REQ | :=*predeterminado* | :VARARG]]...]

instrucciones

ENDM [[:*valor*]]

Marca un macro bloque llamado *nombre* y establece receptáculos de *parámetros* para los argumentos que se pasan cuando se hace una llamada a la macro. Una macro función devuelve *valor* a la instrucción que hizo la llamada.

.MMX

Habilita el ensamblado de las instrucciones MMX.

.MODEL *modelomemoria* [[, *tipolenguaje*] [[, *opciónpila*]]

Inicializa el modelo de memoria del programa. El *modelomemoria* puede ser **TINY**, **SMALL**, **COMPACT**, **MEDIUM**, **LARGE**, **HUGE** o **FLAT**. El *tipolenguaje* puede ser **C**, **BASIC**, **FORTRAN**, **PASCAL**, **SYSCALL** o **STDCALL**. La *opciónpila* puede ser **NEARSTACK** o **FARSTACK**.

NAME *nombremódulo*

Se ignora.

.NO87

Impide el ensamblado de todas las instrucciones de punto flotante.

.NOCREF [[:*nombre*[[, *nombre*]...]]

Suprime el listado de los símbolos en la tabla de símbolos y en el archivo del explorador. Si se especifican nombres, sólo se suprimen los nombres dados. Igual que **.XREF**.

.NOLIST

Suprime el listado del programa. Igual que **.XLIST**.

.NOLISTIF

Suprime el listado de los bloques condicionales cuya condición se evalúe como falsa (0). Ésta es la opción predeterminada. Igual que **.SFCOND**.

.NOLISTMACRO

Suprime el listado de las expansiones de macros. Igual que **.SALL**.

OPTION listaopciones

Habilita y deshabilita características del ensamblador. Las opciones disponibles incluyen **CASEMAP**, **DOTNAME**, **NODOTNAME**, **EMULATOR**, **NOEMULATOR**, **EPILOGUE**, **EXPR16**, **EXPR32**, **LANGUAGE**, **LJMP**, **NOLJMP**, **M510**, **NOM510**, **NOKEYWORD**, **NOSIGNEXTEND**, **OFFSET**, **OLDMACROS**, **NOOLDMACROS**, **OLDSTRUCTS**, **NOOLDSTRUCTS**, **PROC**, **PROLOGUE**, **READONLY**, **NOREADONLY**, **SCOPED**, **NOSCOPED**, **SEGMENT** y **SETIF2**.

ORG expresión

Establece el contador de ubicación a *expresión*.

%OUT

Vea **ECHO**.

[[nombre]] OWORD inicializador [, inicializador]]...

Asigna e inicializa de manera opcional una palabra octal (16 bytes) de almacenamiento para cada *inicializador*. También puede usarse como especificador de tipo, en cualquier parte en donde un tipo sea legal. Este tipo de datos lo utilizan principalmente las instrucciones SIMD de flujo continuo; contiene un arreglo de cuatro reales de 4 bytes.

PAGE [[longitud]], anchura]]

Establece la *longitud* y *anchura* de los caracteres del listado del programa. Si no se proporcionan argumentos, genera un avance de página.

PAGE⁺

Incrementa el número de sección y restablece el número de página a 1.

POPCONTEXT contexto

Restaura parte o todo el *contexto* actual (guardado por la directiva **PUSHCONTEXT**). El *contexto* puede ser **ASSUMES**, **RADIX**, **LISTING**, **CPU** o **ALL**.

etiqueta **PROC** [[*distancia*]] [[**tipolenguaje**]] [[*visibilidad*]] [[<*argprólogo*>]]

[[**USES** *listaregs*]] [[, **parámetro** [:*etiqueta*]]]]...

instrucciones

etiqueta **ENDP**

Marca el inicio y el final de un bloque de procedimientos llamado *etiqueta*. Las instrucciones en el bloque pueden llamarse mediante la instrucción **CALL** o la directiva **VOKE**.

etiqueta **PROTO** [[*distancia*]] [[**tipolenguaje**]] [[, [**parámetro**] :*etiqueta*]]...

Crea el prototipo de una función.

PUBLIC [[tipolenguaje**]] *nombre* [, [[**tipolenguaje**]] *nombre*]]...**

Hace que cada variable, etiqueta o símbolo absoluto que se especifique como *nombre* esté disponible para todos los demás módulos en el programa.

PURGE *nombremacro* [, *nombremacro*]]...

Elimina las macros especificadas de la memoria.

PUSHCONTEXT contexto

Guarda parte de, o todo el *contexto* actual; las suposiciones de los registros de segmento, el valor de la base, las banderas de listado y cref, o los valores del procesador/coprocesador. El *contexto* puede ser **ASSUMES**, **RADIX**, **LISTING**, **CPU** o **ALL**.

[[*nombre*]] QWORD *inicializador* [, *inicializador*]]...

Asigna e inicializa de manera opcional 8 bytes de almacenamiento para cada *inicializador*. También puede usarse como un especificador de tipo, en cualquier parte en donde un tipo sea legal.

.RADIX expresión

Establece la base predeterminada, en el rango de 2 a 16, con el valor de *expresión*.

nombre REAL4 inicializador [, inicializador]... . . .

Asigna e inicializa de manera opcional un número de punto flotante con precisión simple (4 bytes) para cada *inicializador*.

nombre REAL8 inicializador [, inicializador]... . . .

Asigna e inicializa de manera opcional un número de punto flotante con precisión doble (8 bytes) para cada *inicializador*.

nombre REAL10 inicializador [, inicializador]... . . .

Asigna e inicializa de manera opcional un número de punto flotante de 10 bytes para cada *inicializador*.

nombreregistro RECORD nombrecampo:anchura [= expresión] [, nombrecampo:anchura [= expresión]]... . . .

Declara un tipo de registro que consiste en los campos especificados. El *nombrecampo* nombra el campo, *anchura* especifica el número de bits, y *expresión* proporciona su valor inicial.

.REPEAT

instrucciones

.UNTIL condición

Genera código que repite la ejecución del bloque de *instrucciones* hasta que *condición* sea verdadera.

.UNTILCXZ, que se hace verdadera cuando CX es cero, puede sustituirse por .UNTIL. La *condición* es opcional con .UNTILCXZ.

REPEAT expresión

instrucciones

ENDM

Marca un bloque que se va a repetir un número de veces determinado por *expresión*. Igual que REPT.

REPT

Vea REPEAT.

.SALL

Vea .NOLISTMACRO.

nombre SBYTE inicializador [, inicializador]... . . .

Asigna e inicializa de manera opcional un byte con signo de almacenamiento para cada *inicializador*.

También puede usarse como un especificador de tipo, en cualquier parte en donde un tipo sea legal.

nombre SDWORD inicializador [, inicializador]... . . .

Asigna e inicializa de manera opcional una doble palabra con signo (4 bytes) de almacenamiento para cada *inicializador*. También puede usarse como un especificador de tipo, en cualquier parte en donde un tipo sea legal.

nombre SEGMENT [[READONLY]] [[alineación]] [[combinación]] [[uso]] [[‘clase’]]

instrucciones

nombre ENDS

Define un segmento de programa llamado *nombre* con los atributos de segmento *alineación* (BYTE, WORD, DWORD, PARA, PAGE), *combinación* (PUBLIC, STACK, COMMON, MEMORY, AT dirección, PRIVATE), *uso* (USE16, USE32, FLAT) y *clase*.

.SEQ

Ordena los segmentos en forma secuencial (el orden predeterminado).

.SFCOND

Vea .NOLISTIF.

nombre SIZESTR elementotexto

Busca el tamaño de un elemento de texto.

.STACK [[tamaño]]

Al utilizarse con .MODEL, define un segmento de pila (con el nombre de segmento STACK). El *tamaño* opcional especifica el número de bytes para la pila (valor predeterminado: 1024). La directiva .STACK cierra en forma automática la instrucción de la pila.

.STARTUP

Genera el código de inicio del programa.

STRUC

Vea **STRUCT**.

nombre STRUCT [[*alineación*]] [[, **NONUNIQUE**]]

declaracionescampos

nombre ENDS

Declara un tipo de estructura con las *declaracionescampos* especificadas. Cada campo debe ser una definición válida de datos. Igual que **STRUC**.

nombre SUBSTR *elementotexto*, *posición* [[, *longitud*]]

Devuelve una subcadena de *elementotexto*, empezando en la *posición* indicada. El *elementotexto* puede ser una cadena literal, una constante precedida por %, o la cadena devuelta por una macrofunción.

SUBTITLE *texto*

Define el subtítulo del listado. Igual que **SUBTTL**.

SUBTTL

Vea **SUBTITULO**.

nombre SWORD *inicializador* [[, *inicializador*]] . . .

Asigna e inicializa de manera opcional una palabra con signo (2 bytes) de almacenamiento para cada *inicializador*. También puede usarse como un especificador de tipo, en cualquier parte en donde un tipo sea legal.

[[**nombre**]] **TBYTE** *inicializador* [[, *inicializador*]] . . .

Asigna e inicializa de manera opcional 10 bytes de almacenamiento para cada *inicializador*. También puede usarse como un especificador de tipo, en cualquier parte en donde un tipo sea legal.

nombre TEXTEQU [[*elementotexto*]]

Asigna *elementotexto* a *nombre*. El *elementotexto* puede ser una cadena literal, una constante precedida por un %, o la cadena devuelta por una macrofunción.

.TFCOND

Activa o desactiva el listado de bloques condicionales falsos.

TITLE *texto*

Define el título del listado del programa.

nombre TYPEDEF *tipo*

Define un nuevo tipo llamado *nombre*, el cual es equivalente a *tipo*.

nombre UNION [[*alineación*]] [[, **NONUNIQUE**]]

declaracionescampos

[[*nombre*]] ENDS

Declara una unión de uno o más tipos de datos. Las *declaracionescampos* deben ser definiciones válidas de datos. Omita la etiqueta **ENDS** *nombre* en las definiciones **UNION** anidadas.

.UNTIL

Vea **.REPEAT**.

.UNTILCXZ

Vea **.REPEAT**.

.WHILE *condición*

Instrucciones

.ENDW

Genera código que ejecuta el bloque de *instrucciones* mientras *condición* sea verdadera.

WHILE *expresión*

instrucciones

ENDM

Repite el ensamblado de *instrucciones* de bloques, siempre y cuando *expresión* sea verdadera.

[[*nombre*]] WORD *inicializador* [[, *inicializador*]] . . .

Asigna e inicializa de manera opcional una palabra (2 bytes) de almacenamiento para cada *inicializador*.

También puede usarse como un especificador de tipo, en cualquier parte en donde un tipo sea legal.

.XALL

Vea **.LISTMACRO**

.XREF

Vea **.NOCREF**.

.XLIST

Vea **.NOLIST**.

.XMM

Habilita el ensamblado de las instrucciones de la Extensión SIMD de flujo continuo de Internet.

A.6 Símbolos

\$

El valor actual del contador de ubicación.

?

En las declaraciones de datos, un valor que el ensamblador asigna pero no inicializa.

@@:

Define una etiqueta de código que se puede reconocer sólo entre *etiqueta1* y *etiqueta2*, en donde *etiqueta1* es el inicio del código o la etiqueta @@: anterior, y *etiqueta2* es el final del código o la siguiente etiqueta @@:. Vea **@B** y **@F**.

@B

La ubicación de la etiqueta @@: anterior.

@Catstr(*cadena1* [[, *cadena2* . . .]])

Macrofunción que concatena una o más cadenas. Devuelve una cadena.

@code

El nombre del segmento de código (macro de texto).

@CodeSize

0 para los modelos **TINY**, **SMALL**, **COMPACT** y **FLAT**, y 1 para los modelos **MEDIUM**, **LARGE** y **HUGE** (igualación numérica).

@Cpu

Una máscara de bit que especifica el modo del procesador (igualación numérica).

@CurSeg

El nombre del segmento actual (macro de texto).

@data

El nombre del grupo de datos predeterminado. Se evalúa como DGROUP para todos los modelos excepto **FLAT**. Se evalúa como **FLAT** en el modelo de memoria **FLAT** (macro de texto).

@DataSize

0 para los modelos **TINY**, **SMALL**, **MEDIUM** y **FLAT**; 1 para los modelos **COMPACT** y **LARGE**; y 2 para el modelo **HUGE** (igualación numérica).

@Date

La fecha del sistema en el formato mm/dd/aa (macro de texto).

@Environ(*varent*)

Valor de la variable de entorno *varent* (macrofunción).

@F

La ubicación de la siguiente etiqueta **@@**:

@fardata

El nombre del segmento definido por la directiva **.FARDATA** (macro de texto).

@fardata?

El nombre del segmento definido por la directiva **.FARDATA?** (macro de texto).

@FileCur

El nombre del archivo actual (macro de texto).

@FileName

El nombre base del archivo principal que se va a ensamblar (macro de texto).

@InStr(*posición*], *cadena1*, *cadena2*)

Macrofunción que busca la primera ocurrencia de *cadena2* en *cadena1*, empezando en la *posición* dentro de *cadena1*. Si la *posición* no aparece, la búsqueda empieza al principio de *cadena1*. Devuelve un entero de posición o 0 si no se encuentra *cadena2*.

@Interface

Información acerca de los parámetros de lenguaje (igualación numérica).

@Line

El número de líneas de código fuente en el archivo actual (igualación numérica).

@Modelo

1 para el modelo **TINY**, 2 para el modelo **SMALL**, 3 para el modelo **COMPACT**, 4 para el modelo **MEDIUM**, 5 para el modelo **LARGE**, 6 para el modelo **HUGE**, y 7 para el modelo **FLAT** (igualación numérica).

@SizeStr(*cadena*)

Macrofunción que devuelve la longitud de la cadena dada. Devuelve un entero.

@stack

DGROUP para pilas cercanas o STACK para pilas lejanas (macro de texto).

@SubStr(*cadena*, *posición* [, *longitud*])

Macrofunción que devuelve una subcadena que inicia en *posición*.

@Time

La hora del sistema en formato hh:mm:ss de 24 horas (macro de texto).

@Version

610 en MASM 6.1 (macro de texto).

@WordSize

Dos para un segmento de 16 bits o 4 para un segmento de 32 bits (igualación numérica).

A.7 Operadores

expresión1* + *expresión2

Devuelve el resultado de *expresión1* más *expresión2*.

expresión1* - *expresión2

Devuelve el resultado de *expresión1* menos *expresión2*.

expresión1* * *expresión2

Devuelve el resultado de *expresión1* por *expresión2*.

expresión1* / *expresión2

Devuelve el resultado de *expresión1* entre *expresión2*.

-*expresión*

Invierte el signo de *expresión*.

expresión1[expresión2]

Devuelve el resultado de *expresión1* más [*expresión2*].

segmento:expresión

Redefine el segmento predeterminado de *expresión* con *segmento*. El *segmento* puede ser un registro de segmento, nombre de grupo, nombre de segmento o expresión de segmento. La *expresión* debe ser una constante.

expresión.campo [[.campo]]...

Devuelve *expresión* más el desplazamiento de *campo* dentro de su estructura o unión.

[registro].campo [[.campo]]...

Devuelve el valor en la ubicación a la que apunta *registro*, más el desplazamiento de *campo* dentro de su estructura o unión.

<texto>

Trata a *texto* como un elemento literal individual.

“textο”

Trata a “*textο*” como una cadena.

‘textο’

Trata a ‘*textο*’ como una cadena.

!carácter

Trata a *carácter* como literal, en vez de un operador o símbolo.

:textο

Trata a *textο* como un comentario.

;;textο

Trata a *textο* como un comentario en una macro que sólo aparece en la definición de la macro. El listado no muestra *textο* en donde se expande la macro.

%expresión

Trata el valor de *expresión* en el argumento de una macro como texto.

&parámetro&

Sustituye *parámetro* con su correspondiente valor de argumento.

ABS

Vea la directiva EXTERNDEF.

ADDR

Vea la directiva INVOKE.

expresión1 AND expresión2

Devuelve el resultado de una operación AND a nivel de bits para *expresión1* y *expresión2*.

cuenta DUP (valorinicial [[, valorinicial]]...)

Especifica el número de *cuenta* de declaraciones de *valorinicial*.

expresión1 EQ expresión2

Devuelve verdadero (-1) si *expresión1* es igual a *expresión2*, y devuelve falso (0) si no lo es.

expresión1 GE expresión2

Devuelve verdadero (-1) si *expresión1* es mayor o igual a *expresión2*, y devuelve falso (0) si no lo es.

expresión1 GT expresión2

Devuelve verdadero (-1) si *expresión1* es mayor que *expresión2*, y devuelve falso (0) si no lo es.

HIGH expresión

Devuelve el byte superior de *expresión*.

HIGHWORD expresión

Devuelve la palabra superior de *expresión*.

expresión1 LE expresión2

Devuelve verdadero (-1) si *expresión1* es menor o igual a *expresión2*, y devuelve falso (0) si no lo es.

LENGTH *variable*

Devuelve el número de elementos de datos en la *variable* que crea el primer inicializador.

LENGTHOF *variable*

Devuelve el número de objetos de datos en *variable*.

LOW *expresión*

Devuelve el byte inferior de *expresión*.

LOWWORD *expresión*

Devuelve la palabra baja de *expresión*.

LROFFSET *expresión*

Devuelve el desplazamiento de *expresión*. Igual que **OFFSET**, pero genera un desplazamiento resuelto por el cargador, lo cual permite a Windows reubicar los segmentos de código.

expresión1* LT *expresión2

Devuelve verdadero (-1) si *expresión1* es menor que *expresión2*, y devuelve falso (0) si no lo es.

MASK {*nombrecamporegistro* | *registro*}

Devuelve una máscara de bits en la que se activan los bits en *nombrecamporegistro* o *registro*, y todos los demás bits se borran.

expresión1* MOD *expresión2

Devuelve el valor entero del residuo (módulo) al dividir *expresión1* entre *expresión2*.

expresión1* NE *expresión2

Devuelve verdadero (-1) si *expresión1* no es igual a *expresión2*, y devuelve falso (0) si lo es.

NOT *expresión*

Devuelve *expresión* con todos los bits invertidos.

OFFSET *expresión*

Devuelve el desplazamiento de *expresión*.

OPATTR *expresión*

Devuelve una palabra que define el modo y el alcance de *expresión*. El byte inferior es idéntico al byte devuelto por **.TYPE**. El byte superior contiene información adicional.

expresión1* OR *expresión2

Devuelve el resultado de una operación OR a nivel de bits para *expresión1* y *expresión2*.

tipo* PTR *expresión

Obliga a que la *expresión* se trate como si tuviera el *tipo* especificado.

[[*distancia*]] PTR *tipo*

Especifica un apuntador a *tipo*.

SEG *expresión*

Devuelve el segmento de *expresión*.

expresión* SHL *cuenta

Devuelve el resultado de desplazar los bits de la *expresión* a la izquierda, un número de bits especificado por *cuenta*.

SHORT *etiqueta*

Establece el tipo de *etiqueta* a corto. Todos los saltos a *etiqueta* deben ser cortos (dentro del rango de -128 a +127 bytes, desde la instrucción de salto hasta *etiqueta*).

expresión* SHR *cuenta

Devuelve el resultado de desplazar los bits de *expresión* a la derecha, un número de bits especificado por *cuenta*.

SIZE *variable*

Devuelve el número de bytes en *variable* que asigna el primer inicializador.

SIZEOF {*variable* | *tipo*}

Devuelve el número de bytes en *variable* o *tipo*.

THIS tipo

Devuelve un operando del *tipo* especificado, cuyos valores de desplazamiento y segmento son iguales al valor del contador de ubicación actual.

.TYPE expresión

Vea OPATTR.

TYPE expresión

Devuelve el tipo de *expresión*.

WIDTH {nombrecamporegistro | registro}

Devuelve la anchura en bits del *nombrecamporegistro* o *registro* actual.

expresión1 XOR expresión2

Devuelve el resultado de una operación XOR a nivel de bits para *expresión1* y *expresión2*.

A.8 Operadores en tiempo de ejecución

Los siguientes operadores se utilizan sólo dentro de los bloques .IF, .WHILE o .REPEAT, y se evalúan en tiempo de ejecución, no en tiempo de ensamblado:

expresión1 == expresión2

Es igual a.

expresión1 != expresión2

No es igual a.

expresión1 > expresión2

Es mayor que.

expresión1 >= expresión2

Es mayor o igual a.

expresión1 < expresión2

Es menor que.

expresión1 <= expresión2

Es menor que o igual que.

expresión1 || expresión2

OR lógico.

expresión1 && expresión2

AND lógico.

expresión1 & expresión2

AND a nivel de bits.

!expresión

Negación lógica.

CARRY?

Estado de la bandera Acarreo.

OVERFLOW?

Estado de la bandera Desbordamiento.

PARITY?

Estado de la bandera Paridad.

SIGN?

Estado de la bandera Signo.

ZERO?

Estado de la bandera Cero.

El conjunto de instrucciones IA-32

- B.1 Introducción
 - B.1.1 Banderas
 - B.1.2 Descripciones y formatos de las instrucciones
- B.2 Detalles del conjunto de instrucciones (que no son de punto flotante)
- B.3 Instrucciones de punto flotante

B.1 Introducción

Este apéndice es una guía rápida para las instrucciones IA-32 de uso más común. No cubre las instrucciones en modo de sistema o las instrucciones que, por lo general, se utilizan sólo en el código del núcleo del sistema operativo, o en los controladores de dispositivos en modo protegido.

B.1.1 Banderas

La descripción de cada instrucción contiene una serie de cuadros que describen cómo la instrucción afecta a las banderas de estado de la CPU. Cada bandera se identifica mediante una sola letra:

O	Desbordamiento	S	Signo	P	Paridad
D	Dirección	Z	Cero	C	Acarreo
I	Interrupción	A	Acarreo auxiliar		

Dentro de los cuadros, la siguiente notación muestra cómo cada instrucción afecta a las banderas:

1	Activa la bandera.
0	Borra la bandera.
?	Puede cambiar la bandera a un valor indeterminado.
(en blanco)	La bandera no se cambia.
*	Cambia la bandera, de acuerdo a ciertas reglas específicas asociadas con la misma.

Por ejemplo, el siguiente diagrama de las banderas de la CPU se tomó de una de las descripciones de las instrucciones:

O	D	I	S	Z	A	P	C
?			?	?	*	?	*

En este diagrama podemos ver que las banderas Desbordamiento, Signo, Cero y Paridad cambiarán a valores desconocidos. Las banderas Acarreo auxiliar y Acarreo se modificarán de acuerdo a las reglas asociadas con las banderas. Las banderas Dirección e Interrupción no se cambiarán.

B.1.2 Descripciones y formatos de las instrucciones

Al hacer una referencia a los operandos de origen y de destino, utilizamos el orden natural de los operandos en todas las instrucciones del Intel 80x86, en donde el primer operando es el destino y el segundo es el origen. Por ejemplo, en la instrucción MOV al destino se le asignará una copia de los datos en el operando de origen:

MOV destino, origen

Puede haber muchos formatos disponibles para una sola instrucción. La tabla B-1 contiene una lista de símbolos que se utilizan en los formatos de las instrucciones. En las descripciones de cada instrucción, utilizamos la notación “(IA-32)” para indicar que una instrucción, o una de sus variantes, sólo está disponible en los procesadores de la familia IA-32 (del Intel386 en adelante). De manera similar, la notación “(80286)” indica que debe usarse por lo menos un procesador 80286.

Las notaciones de los registros como (E)CX, (E)SI, (E)DI, (E)SP, (E)BP, y (E)IP hacen la diferencia entre los procesadores IA-32 que utilizan los registros de 32 bits y los primeros procesadores que utilizaban registros de 16 bits.

Tabla B-1 Símbolos que se utilizan en los formatos de las instrucciones.

Símbolo	Descripción
<i>reg</i>	Un registro general de 8, 16 o 32 bits de la siguiente lista: AH, AL, BH, BL, CH, CL, DH, DL, AX, BX, CX, DX, SI, DI, BP, SP, EAX, EBX, ECX, EDX, ESI, EDI, EBP y ESP
<i>reg8, reg16, reg32</i>	Un registro general, identificado por su número de bits
<i>regseg</i>	Un registro de segmento de 16 bits (CS, DS, ES, SS, FS, GS)
<i>acum</i>	AL, AX o EAX
<i>mem</i>	Un operando de memoria, usando cualquiera de los modos de direccionamiento de memoria estándar
<i>mem8, mem16, mem32</i>	Un operando de memoria, identificado por su número de bits
<i>etiquetacorta</i>	Una ubicación en el segmento de código dentro de un rango de -127 a +128 bytes de la ubicación actual
<i>etiquetacercana</i>	Una ubicación en el segmento de código actual, identificado por una etiqueta
<i>etiquetalejana</i>	Una ubicación en un segmento de código externo, identificado por una etiqueta
<i>Inm</i>	Un operando inmediato
<i>inm8, inm16, inm32</i>	Un operando inmediato, identificado por su número de bits
<i>instrucción</i>	Una instrucción en lenguaje ensamblador del 80x86

B.2 Detalles del conjunto de instrucciones (que no son de punto flotante)

AAD	<p>Ajuste ASCII antes de la división</p> <table style="margin-left: auto; margin-right: auto;"> <tr> <th>O</th><th>D</th><th>I</th><th>S</th><th>Z</th><th>A</th><th>P</th><th>C</th></tr> <tr> <td>?</td><td></td><td></td><td>*</td><td>*</td><td>?</td><td>*</td><td>?</td></tr> </table>	O	D	I	S	Z	A	P	C	?			*	*	?	*	?
O	D	I	S	Z	A	P	C										
?			*	*	?	*	?										
	<p>Convierte los dígitos BCD desempaquetados en AH y AL a un solo valor binario, como preparación para la instrucción DIV.</p> <p>Formato de la instrucción:</p> <pre>AAD</pre>																

AAS	<p>Ajuste ASCII después de la resta</p> <table style="margin-left: auto; margin-right: auto;"> <tr> <th>O</th><th>D</th><th>I</th><th>S</th><th>Z</th><th>A</th><th>P</th><th>C</th></tr> <tr> <td>?</td><td></td><td></td><td>?</td><td>?</td><td>*</td><td>?</td><td>*</td></tr> </table>	O	D	I	S	Z	A	P	C	?			?	?	*	?	*
O	D	I	S	Z	A	P	C										
?			?	?	*	?	*										
	<p>Ajusta el resultado en AX después de una operación de resta. Si AL > 9, AAS decrementa a AH y activa las banderas Acarreo y Acarreo auxiliar.</p> <p>Formato de la instrucción:</p> <pre>AAS</pre>																

ADC	Suma con acarreo																
	<table style="margin: auto; border-collapse: collapse;"> <thead> <tr> <th>O</th><th>D</th><th>I</th><th>S</th><th>Z</th><th>A</th><th>P</th><th>C</th></tr> </thead> <tbody> <tr> <td>*</td><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td></tr> </tbody> </table>	O	D	I	S	Z	A	P	C	*			*	*	*	*	*
O	D	I	S	Z	A	P	C										
*			*	*	*	*	*										

ADD**Suma**

O	D	I	S	Z	A	P	C
*			*	*	*	*	*

Un operando de origen se suma a un operando de destino, y la suma se almacena en el destino. Los operandos deben tener el mismo tamaño.

Formatos de la instrucción:

ADD *reg, reg*
ADD *mem, reg*
ADD *reg, mem*

ADD *reg, imm*
ADD *mem, imm*
ADD *acum, imm*

AND**AND lógico**

O	D	I	S	Z	A	P	C
0			*	*	?	*	0

A cada bit en el operando de destino se le aplica un AND con el bit correspondiente en el operando de origen.

Formatos de la instrucción:

AND *reg, reg*
AND *mem, reg*
AND *reg, mem*

AND *reg, imm*
AND *mem, imm*
AND *acum, imm*

BOUND**Comprobar límites de arreglo (80286)**

O	D	I	S	Z	A	P	C

Verifica que el valor de un índice con signo se encuentre dentro de los límites del arreglo. En el procesador 80286, el operando de destino puede ser cualquier registro de 16 bits que contenga el índice a comprobar. El operando de origen debe ser un operando de memoria de 32 bits, en el que las palabras superior e inferior contienen los límites superior e inferior del valor del índice. En la familia IA-32, el destino puede ser un registro de 32 bits y el origen puede ser un operando de memoria de 64 bits.

Formatos de la instrucción:

BOUND *reg16, mem32*

BOUND *reg32, mem64*

**BSF,
BSR****Exploración de bit (IA-32)**

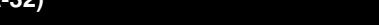
O	D	I	S	Z	A	P	C
?			?	?	?	?	?

Explora un operando en busca del primer bit activado. Si se encuentra el bit, se borra la bandera Cero, y al operando de destino se le asigna el número de bit (índice) del primer bit activo que se haya encontrado. Si no se encontró un bit, ZF = 1. BSF explora desde el bit 0 hasta el bit más alto, y BSR empieza en el bit más alto y explora en orden descendente, hasta el bit 0.

Formatos de la instrucción (se aplican a BSF y BSR):

BSF *reg16/r/m16*

BOUND *reg32,r/m32*

BSWAP	Intercambiar byte (IA-32)  <p>Invierte el orden de los bytes de un registro de destino de 32 bits. Formato de la instrucción: <code>BSWAP reg32</code></p>
-------	---

BT, BTC, BTR, BTS	<p>Pruebas de bit (IA-32)</p> <table style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>O</th><th>D</th><th>I</th><th>S</th><th>Z</th><th>A</th><th>P</th><th>C</th></tr> </thead> <tbody> <tr> <td>?</td><td></td><td></td><td>?</td><td>?</td><td>?</td><td>?</td><td>*</td></tr> </tbody> </table> <hr/> <p>Copia un bit especificado (n) en la bandera Acarreo. El operando de destino contiene el valor en el que se encuentra el bit, y el operando de origen indica la posición del bit dentro del destino. BT copia el bit n a la bandera Acarreo. BTC copia el bit n a la bandera Acarreo y complementa el bit n en el operando de destino. BTR copia el bit n en la bandera Acarreo y borra el bit n en el destino. BTS copia el bit n en la bandera Acarreo y activa el bit n en el destino.</p> <p>Formatos de la instrucción:</p> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;"> $BT \ r/m16, \#nm8$ $BT \ r/m32, \#nm8$ </td><td style="width: 50%;"> $BT \ r/m16, r16$ $BT \ r/m32, r32$ </td></tr> </table>	O	D	I	S	Z	A	P	C	?			?	?	?	?	*	$BT \ r/m16, \#nm8$ $BT \ r/m32, \#nm8$	$BT \ r/m16, r16$ $BT \ r/m32, r32$
O	D	I	S	Z	A	P	C												
?			?	?	?	?	*												
$BT \ r/m16, \#nm8$ $BT \ r/m32, \#nm8$	$BT \ r/m16, r16$ $BT \ r/m32, r32$																		

CALL	<p>Llamar a un procedimiento</p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td></tr> </table>	O	D	I	S	Z	A	P	C	<input type="text"/>							
O	D	I	S	Z	A	P	C										
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>										
	<p>Mete la ubicación de la siguiente instrucción en la pila y se transfiere a la ubicación de destino. Si el procedimiento es cercano (en el mismo segmento), sólo se mete el desplazamiento de la siguiente instrucción; en caso contrario, se meten el segmento y el desplazamiento.</p> <p>Formatos de la instrucción:</p> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;"> <code>CALL etiquetacercana</code> <code>CALL etiquetalejana</code> <code>CALL reg</code> </td> <td style="width: 50%;"> <code>CALL mem16</code> <code>CALL mem32</code> </td> </tr> </table>	<code>CALL etiquetacercana</code> <code>CALL etiquetalejana</code> <code>CALL reg</code>	<code>CALL mem16</code> <code>CALL mem32</code>														
<code>CALL etiquetacercana</code> <code>CALL etiquetalejana</code> <code>CALL reg</code>	<code>CALL mem16</code> <code>CALL mem32</code>																

CBW	<p>Convertir byte a palabra</p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td></tr> </table> <p>Extiende el bit de signo en AL a lo largo del registro AH. Formato de la instrucción:</p> <pre>CBW</pre>	O	D	I	S	Z	A	P	C	<input type="text"/>							
O	D	I	S	Z	A	P	C										
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>										

CDQ**Convertir doble palabra a palabra cuádruple (IA-32)**

O	D	I	S	Z	A	P	C

Extiende el bit de signo en EAX a lo largo del registro EDX.

Formato de la instrucción:

CDQ

CLC**Borrar bandera Acarreo**

O	D	I	S	Z	A	P	C
							0

Borra la bandera Acarreo, asignándole un cero.

Formato de la instrucción:

CLC

CLD**Borrar bandera Dirección**

O	D	I	S	Z	A	P	C
	0						

Borra la bandera Dirección, asignándole un cero. Las instrucciones de primitivas de cadena incrementan (E)SI y (E)DI de manera automática.

Formato de la instrucción:

CLD

CLI**Borrar bandera Interrupción**

O	D	I	S	Z	A	P	C
		0					

Borra la bandera Interrupción, asignándole un cero. Esto deshabilita las interrupciones de hardware enmascarables hasta que se ejecute una instrucción STI.

Formato de la instrucción:

CLI

CMC**Complementar la bandera Acarreo**

O	D	I	S	Z	A	P	C
							*

Cambia el valor actual la bandera Acarreo, de 0 a 1 y viceversa.

Formato de la instrucción:

CMC

CMP	Comparar															
	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td>*</td><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td></tr> </table>	O	D	I	S	Z	A	P	C	*			*	*	*	*
O	D	I	S	Z	A	P	C									
*			*	*	*	*	*									
Compara el destino con el origen, realizando una resta implícita entre el origen y el destino.																
Formatos de la instrucción:																
CMP <i>reg, reg</i> CMP <i>mem, reg</i> CMP <i>reg, mem</i>				CMP <i>reg, imm</i> CMP <i>mem, imm</i> CMP <i>acum, imm</i>												

CMPS, CMPSB, CMPSW, CMPSD	Comparar cadenas															
	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td>*</td><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td></tr> </table>	O	D	I	S	Z	A	P	C	*			*	*	*	*
O	D	I	S	Z	A	P	C									
*			*	*	*	*	*									
Compara las cadenas en la memoria direccionada por DS:(E)SI y ES:(E)DI. Realiza una resta implícita entre el destino y el origen. CMPSB compara bytes, CMPSW compara palabras y CMPSD compara dobles palabras (en procesadores IA-32). (E)SI y (E)DI se incrementan o decrementan de acuerdo con el tamaño del operando y el estado de la bandera Dirección. Si la bandera Dirección está activa, (E)SI y (E)DI se decrementan; en caso contrario (E)SI y (E)DI se incrementan.																
Formatos de la instrucción (se omitieron de manera intencional los formatos que utilizan operandos explícitos):																
CMPSB CMPSD				CMPSW												

CMPXCHG	Comparar e intercambiar															
	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td>*</td><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td></tr> </table>	O	D	I	S	Z	A	P	C	*			*	*	*	*
O	D	I	S	Z	A	P	C									
*			*	*	*	*	*									
Compara el destino con el acumulador (AL, AX o EAX). Si son iguales, el origen se copia al destino; en caso contrario, el destino se copia al acumulador.																
Formatos de la instrucción:																
CMPXCHG <i>reg, reg</i>				CMPXCHG <i>mem, reg</i>												

CWD	Convierte palabra a doble palabra															
	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>	O	D	I	S	Z	A	P	C							
O	D	I	S	Z	A	P	C									
Extiende el bit de signo en AX hasta el registro DX.																
Formato de la instrucción:																
CWD																

DAA**Ajuste decimal después de la suma**

O	D	I	S	Z	A	P	C
?			*	*	*	*	*

Ajusta la suma binaria en AL después de haber sumado dos valores BCD empaquetados. Convierte la suma a dos dígitos BCD en AL.

Formato de la instrucción:

DAA

DAS**Ajuste decimal después de la resta**

O	D	I	S	Z	A	P	C
?			*	*	*	*	*

Convierte el resultado binario de una operación de resta a dos dígitos BCD empaquetados en AL.

Formato de la instrucción:

DAS

DEC**Decrementar**

O	D	I	S	Z	A	P	C
*			*	*	*	*	

Resta 1 de un operando. No afecta a la bandera Acarreo.

Formatos de la instrucción:

DEC reg

DEC mem

DIV**Dividir entero sin signo**

O	D	I	S	Z	A	P	C
?			?	?	?	?	?

Realiza una división de enteros sin signo de 8, 16 o 32 bits. Si el divisor es de 8 bits, el dividendo es AX, el cociente es AL y el residuo es AH. Si el divisor es de 16 bits, el dividendo es DX:AX, el cociente es AX y el residuo es DX. Si el divisor es de 32 bits, el dividendo es EDX:EAX, el cociente es EAX y el residuo es EDX.

Formatos de la instrucción:

DIV reg

DIV mem

ENTER**Crear marco de pila (80286)**

O	D	I	S	Z	A	P	C

Crea un marco de pila para un procedimiento que recibe parámetros de pila y utiliza variables de pila locales. El primer operando indica el número de bytes a reservar para las variables de pila locales. El segundo operando indica el nivel de anidamiento del procedimiento (debe establecerse en 0 para C, Basic y FORTRAN).

Formato de la instrucción:

ENTER inm16, inm8

HLT	<p>Detener</p> <table style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr> <td style="text-align: center;">O</td><td style="text-align: center;">D</td><td style="text-align: center;">I</td><td style="text-align: center;">S</td><td style="text-align: center;">Z</td><td style="text-align: center;">A</td><td style="text-align: center;">P</td><td style="text-align: center;">C</td></tr> <tr> <td style="border: 1px solid black; width: 1em;"></td><td style="border: 1px solid black; width: 1em;"></td></tr> </table> <p>Detiene la CPU hasta que ocurra una interrupción de hardware. (Nota: la bandera Interrupción debe activarse con la instrucción STI para que puedan ocurrir interrupciones de hardware.)</p> <p>Formato de la instrucción:</p> <pre>HLT</pre>	O	D	I	S	Z	A	P	C								
O	D	I	S	Z	A	P	C										

IDIV	<p>Dividir entero con signo</p> <table style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr> <td style="text-align: center;">O</td><td style="text-align: center;">D</td><td style="text-align: center;">I</td><td style="text-align: center;">S</td><td style="text-align: center;">Z</td><td style="text-align: center;">A</td><td style="text-align: center;">P</td><td style="text-align: center;">C</td></tr> <tr> <td style="border: 1px solid black; width: 1em;"></td><td style="border: 1px solid black; width: 1em;"></td></tr> </table> <p>Realiza una operación de división de enteros con signo en EDX:EAX, DX:AX o AX. Si el divisor es de 8 bits, el dividendo es AX, el cociente es AL y el residuo es AH. Si el divisor es de 16 bits, el dividendo es DX:AX, el cociente es AX y el residuo es DX. Si el divisor es de 32 bits, el dividendo es EDX:EAX, el cociente es EAX y el residuo es EDX. Por lo general, se utiliza CBW o CWD antes de la operación IDIV para extender el signo del dividendo.</p> <p>Formatos de la instrucción:</p> <pre>IDIV reg</pre> <pre>IDIV mem</pre>	O	D	I	S	Z	A	P	C								
O	D	I	S	Z	A	P	C										

IMUL	<p>Multiplicar entero con signo</p> <table style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr> <td style="text-align: center;">O</td><td style="text-align: center;">D</td><td style="text-align: center;">I</td><td style="text-align: center;">S</td><td style="text-align: center;">Z</td><td style="text-align: center;">A</td><td style="text-align: center;">P</td><td style="text-align: center;">C</td></tr> <tr> <td style="border: 1px solid black; width: 1em;"></td><td style="border: 1px solid black; width: 1em;"></td></tr> </table> <p>Realiza una multiplicación de enteros con signo en AL, AX o EAX. Si el multiplicador es de 8 bits, el multiplicando es AL y el producto es AX. Si el multiplicador es de 16 bits, el multiplicando es AX y el producto es DX:AX. Si el multiplicador es de 32 bits, el multiplicando es EAX y el producto es EDX:EAX. Las banderas Acarreo y Desbordamiento se activan si un producto de 16 bits se extiende hacia AH, si un producto de 32 bits se extiende hacia DX, o si un producto de 64 bits se extiende hacia EDX.</p> <p>Formatos de la instrucción:</p> <pre>IMUL r/m8</pre> <pre>IMUL r/,32</pre> <p>Dos operandos:</p> <pre>IMUL r16/m16</pre> <pre>IMUL r32/m32</pre> <pre>IMUL r16/inm16</pre> <p>Tres operandos:</p> <pre>IMUL r16,r/m16,inm8</pre> <pre>IMUL r32,r/m32,inm8</pre> <pre>IMUL r16,r/m16,inm16</pre> <pre>IMUL r32,r/m32,inm32</pre>	O	D	I	S	Z	A	P	C								
O	D	I	S	Z	A	P	C										

IN Entrada desde un puerto <table style="margin-left: auto; margin-right: auto;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td></tr> </table>	O	D	I	S	Z	A	P	C	<input type="text"/>	<p>Recibe un byte o palabra de un puerto y lo coloca en AL o AX. El operando de origen es la dirección de un puerto, que se expresa como una constante de 8 bits o como una dirección de 16 bits en DX. En el procesador IA-32, una doble palabra puede recibirse desde un puerto y colocarse en EAX.</p> <p>Formatos de la instrucción:</p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td>IN <i>acum, imm</i></td><td>IN <i>acum,DX</i></td></tr> </table>	IN <i>acum, imm</i>	IN <i>acum,DX</i>									
O	D	I	S	Z	A	P	C														
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>														
IN <i>acum, imm</i>	IN <i>acum,DX</i>																				
INC Incrementar <table style="margin-left: auto; margin-right: auto;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td>*</td><td><input type="text"/></td><td><input type="text"/></td><td>*</td><td>*</td><td>*</td><td>*</td><td><input type="text"/></td></tr> </table>	O	D	I	S	Z	A	P	C	*	<input type="text"/>	<input type="text"/>	*	*	*	*	<input type="text"/>	<p>Suma 1 a un operando tipo registro o de memoria.</p> <p>Formatos de la instrucción:</p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td>INC <i>reg</i></td><td>INC <i>mem</i></td></tr> </table>	INC <i>reg</i>	INC <i>mem</i>		
O	D	I	S	Z	A	P	C														
*	<input type="text"/>	<input type="text"/>	*	*	*	*	<input type="text"/>														
INC <i>reg</i>	INC <i>mem</i>																				
INS, INSB, INSW, INSD Entrada desde un puerto a una cadena (80286) <table style="margin-left: auto; margin-right: auto;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td></tr> </table>	O	D	I	S	Z	A	P	C	<input type="text"/>	<p>Recibe una cadena a la que apunta ES:(E)DI desde un puerto. El número de puerto se especifica en DX. Para cada valor recibido, (E)DI se ajusta de la misma forma que en LODSB y las instrucciones de primitivas de cadena similares. Puede utilizarse el prefijo REP con esta instrucción.</p> <p>Formatos de la instrucción:</p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td>INS <i>dest,DX</i></td><td>REP INSB <i>dest,DX</i></td></tr> <tr> <td>REP INSW <i>dest,DX</i></td><td>REP INSD <i>dest,DX</i></td></tr> </table>	INS <i>dest,DX</i>	REP INSB <i>dest,DX</i>	REP INSW <i>dest,DX</i>	REP INSD <i>dest,DX</i>							
O	D	I	S	Z	A	P	C														
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>														
INS <i>dest,DX</i>	REP INSB <i>dest,DX</i>																				
REP INSW <i>dest,DX</i>	REP INSD <i>dest,DX</i>																				
INT Interrupción <table style="margin-left: auto; margin-right: auto;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td><input type="text"/></td><td><input type="text"/></td><td>0</td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td></tr> </table>	O	D	I	S	Z	A	P	C	<input type="text"/>	<input type="text"/>	0	<input type="text"/>	<p>Genera una interrupción de software, que a su vez llama a una subrutina del sistema operativo. Borra la bandera Interrupción y mete las banderas, CS e IP en la pila antes de bifurcar hacia la rutina de interrupción.</p> <p>Formatos de la instrucción:</p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td>INT <i>imm</i></td><td>INT 3</td></tr> </table>	INT <i>imm</i>	INT 3						
O	D	I	S	Z	A	P	C														
<input type="text"/>	<input type="text"/>	0	<input type="text"/>																		
INT <i>imm</i>	INT 3																				
INTO Interrupción por desbordamiento <table style="margin-left: auto; margin-right: auto;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td><input type="text"/></td><td><input type="text"/></td><td>*</td><td>*</td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td><td><input type="text"/></td></tr> </table>	O	D	I	S	Z	A	P	C	<input type="text"/>	<input type="text"/>	*	*	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<p>Genera la interrupción interna 4 de la CPU si se activa la bandera Desbordamiento. MS-DOS no realiza ninguna acción si se llama a INT 4, pero puede sustituirse por una rutina escrita por el usuario.</p> <p>Formato de la instrucción:</p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td>INTO</td></tr> </table>	INTO			
O	D	I	S	Z	A	P	C														
<input type="text"/>	<input type="text"/>	*	*	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>														
INTO																					

IRET	<p>Retorno de interrupción</p> <table style="margin-left: auto; margin-right: auto; border-collapse: collapse; text-align: center;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td> </tr> <tr> <td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td> </tr> </table>	O	D	I	S	Z	A	P	C	*	*	*	*	*	*	*	*
O	D	I	S	Z	A	P	C										
*	*	*	*	*	*	*	*										
	<p>Regresa de una rutina de manejo de interrupciones. Saca el contenido de la pila y lo coloca en (E)IP, CS y las banderas.</p> <p>Formato de la instrucción:</p> <p style="text-align: center;">IRET</p>																

Jcondición	<p>Salto condicional</p> <table style="margin-left: auto; margin-right: auto; border-collapse: collapse; text-align: center;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td> </tr> <tr> <td> </td><td> </td><td> </td><td> </td><td> </td><td> </td><td> </td><td> </td> </tr> </table>	O	D	I	S	Z	A	P	C								
O	D	I	S	Z	A	P	C										
	<p>Salta a una etiqueta si una condición de bandera especificada es verdadera. Si se usa un procesador anterior al IA-32, la etiqueta debe estar en el rango de -128 a +127 bytes, a partir de la ubicación actual. En los procesadores IA-32, el desplazamiento de la etiqueta puede ser un valor positivo o negativo de 32 bits. En la tabla B-2 podrá ver una lista de nemáticos.</p> <p>Formato de la instrucción:</p> <p style="text-align: center;"><i>Jcondición etiqueta</i></p>																

Tabla B-2 Nemáticos de salto condicional.

Nemónico	Comentario	Nemónico	Comentario
JA	Salta si es mayor	JE	Salta si es igual
JNA	Salta si no es mayor	JNE	Salta si no es igual
JAE	Salta si es mayor o igual	JZ	Salta si es cero
JNAE	Salta si no es mayor o igual	JNZ	Salta si no es cero
JB	Salta si es menor	JS	Salta si hay signo
JNB	Salta si no es menor	JNS	Salta si no hay signo
JBE	Salta si es menor o igual	JC	Salta si hay acarreo
JNBE	Salta si no es menor o igual	JNC	Salta si no hay acarreo
JG	Salta si es mayor	JO	Salta si hay desbordamiento
JNG	Salta si no es mayor	JNO	Salta si no hay desbordamiento
JGE	Salta si es mayor o igual	JP	Salta si hay paridad
JNGE	Salta si no es mayor o igual	JPE	Salta si la paridad es igual
JL	Salta si es menor	JNP	Salta si no hay paridad
JNL	Salta si no es menor	JPO	Salta si la paridad es impar
JLE	Salta si es menor o igual	JNLE	Salta si no es menor o igual

**JCXZ,
JECXZ****Salta si CX es cero**

O	D	I	S	Z	A	P	C

Salta a una etiqueta corta si el registro CX es igual a cero. La etiqueta corta debe estar en el rango de -128 a +127 bytes, a partir de la siguiente instrucción. En el procesador IA-3, JECXZ salta si ECX es igual a cero.

Formatos de la instrucción:

JCXZ *etiquetacorta*JECXZ *etiquetacorta***JMP****Salta incondicionalmente a la etiqueta**

O	D	I	S	Z	A	P	C

Salta a una etiqueta de código. Un salto corto se encuentra dentro de los -128 a +127 bytes a partir de la ubicación actual. Un salto cercano está dentro del mismo segmento de código, y un salto lejano se encuentra fuera del segmento actual.

Formatos de la instrucción:

JMP *etiquetacorta*
JMP *etiquetacercana*
JMP *etiquetalejana*JMP *reg16*
JMP *mem16*
JMP *mem32***LAHF****Cargar AH de las banderas**

O	D	I	S	Z	A	P	C

Las siguientes banderas se copian a AH: Signo, Cero, Acarreo auxiliar, Paridad y Acarreo.

Formato de la instrucción:

LAHF

**LDS,
LES,
LFS,
LGS,
LSS****Cargar apuntador lejano**

O	D	I	S	Z	A	P	C

Carga el contenido de un operando de memoria tipo doble palabra en un registro de segmento y en el registro de destino especificado. En los procesadores anteriores al IA-3, LDS carga en DS, LES carga en ES. En el procesador IA-32, LFS carga en FS, LGS carga en GS y LSS carga en SS.

Formato de la instrucción (es igual para LDS, LES, LFS, LGS, LSS):

LDS *reg,mem*

LEA	Cargar dirección efectiva <table style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr> <td style="text-align: center; padding: 2px;">O</td><td style="text-align: center; padding: 2px;">D</td><td style="text-align: center; padding: 2px;">I</td><td style="text-align: center; padding: 2px;">S</td><td style="text-align: center; padding: 2px;">Z</td><td style="text-align: center; padding: 2px;">A</td><td style="text-align: center; padding: 2px;">P</td><td style="text-align: center; padding: 2px;">C</td></tr> <tr> <td style="border: 1px solid black; width: 10px;"></td><td style="border: 1px solid black; width: 10px;"></td></tr> </table>	O	D	I	S	Z	A	P	C								
O	D	I	S	Z	A	P	C										
	<p>Calcula y carga la dirección efectiva de 16 o 32 bits de un operando de memoria. Es similar a MOV.. OFFSET, con la excepción de que sólo LEA puede obtener una dirección que se calcule en tiempo de ejecución.</p> <p>Formato de la instrucción:</p> <p style="margin-left: 40px;">LEA <i>reg, mem</i></p>																

LEAVE	Salir de procedimiento de alto nivel <table style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr> <td style="text-align: center; padding: 2px;">O</td><td style="text-align: center; padding: 2px;">D</td><td style="text-align: center; padding: 2px;">I</td><td style="text-align: center; padding: 2px;">S</td><td style="text-align: center; padding: 2px;">Z</td><td style="text-align: center; padding: 2px;">A</td><td style="text-align: center; padding: 2px;">P</td><td style="text-align: center; padding: 2px;">C</td></tr> <tr> <td style="border: 1px solid black; width: 10px;"></td><td style="border: 1px solid black; width: 10px;"></td></tr> </table>	O	D	I	S	Z	A	P	C								
O	D	I	S	Z	A	P	C										
	<p>Termina el marco de pila de un procedimiento. Esto invierte la acción de la instrucción ENTER al inicio de un procedimiento, restaurando (E)SP y (E)BP a sus valores originales.</p> <p>Formato de la instrucción:</p> <p style="margin-left: 40px;">LEAVE</p>																

LOCK	Bloquea el bus del sistema <table style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr> <td style="text-align: center; padding: 2px;">O</td><td style="text-align: center; padding: 2px;">D</td><td style="text-align: center; padding: 2px;">I</td><td style="text-align: center; padding: 2px;">S</td><td style="text-align: center; padding: 2px;">Z</td><td style="text-align: center; padding: 2px;">A</td><td style="text-align: center; padding: 2px;">P</td><td style="text-align: center; padding: 2px;">C</td></tr> <tr> <td style="border: 1px solid black; width: 10px;"></td><td style="border: 1px solid black; width: 10px;"></td></tr> </table>	O	D	I	S	Z	A	P	C								
O	D	I	S	Z	A	P	C										
	<p>Evita que otros procesadores se ejecuten durante la siguiente instrucción. Esta instrucción se usa cuando otro procesador pueda modificar un operando de memoria accesado en ese momento por la CPU.</p> <p>Formato de la instrucción:</p> <p style="margin-left: 40px;">LOCK <i>instrucción</i></p>																

LDS, LODSB, LODSW, LOSDS	Carga el acumulador desde la cadena <table style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr> <td style="text-align: center; padding: 2px;">O</td><td style="text-align: center; padding: 2px;">D</td><td style="text-align: center; padding: 2px;">I</td><td style="text-align: center; padding: 2px;">S</td><td style="text-align: center; padding: 2px;">Z</td><td style="text-align: center; padding: 2px;">A</td><td style="text-align: center; padding: 2px;">P</td><td style="text-align: center; padding: 2px;">C</td></tr> <tr> <td style="border: 1px solid black; width: 10px;"></td><td style="border: 1px solid black; width: 10px;"></td></tr> </table>	O	D	I	S	Z	A	P	C								
O	D	I	S	Z	A	P	C										
	<p>Carga un byte o palabra de memoria direccionada por DS:(E)SI en el acumulador (AL, AX o EAX). Si se utiliza LODS, debe especificarse el operando de memoria. LODSB carga un byte en AL, LODSW carga una palabra en AX y LODSB en el IA-32 carga una doble palabra en EAX. (E)SI se incrementa o decremente de acuerdo con el tamaño del operando y el estado de la bandera Dirección. Si la bandera Dirección (DF) = 1, (E)SI se decrementa; si DF = 0, (E)SI se incrementa.</p> <p>Formatos de la instrucción:</p> <table style="margin-left: 40px; border-collapse: collapse;"> <tr> <td style="border: 1px solid black; padding: 2px;">LODS</td> <td style="border: 1px solid black; padding: 2px;"><i>mem</i></td> <td style="border: 1px solid black; padding: 2px;">LODSB</td> </tr> <tr> <td style="border: 1px solid black; padding: 2px;">LODS</td> <td style="border: 1px solid black; padding: 2px;"><i>regseg:mem</i></td> <td style="border: 1px solid black; padding: 2px;">LODSW</td> </tr> <tr> <td style="border: 1px solid black; padding: 2px;">LODS</td> <td style="border: 1px solid black; padding: 2px;"></td> <td style="border: 1px solid black; padding: 2px;"></td> </tr> </table>	LODS	<i>mem</i>	LODSB	LODS	<i>regseg:mem</i>	LODSW	LODS									
LODS	<i>mem</i>	LODSB															
LODS	<i>regseg:mem</i>	LODSW															
LODS																	

**LOOP,
LOOPW****Ciclo**

O	D	I	S	Z	A	P	C

Decrementa (E)CX y salta a una etiqueta corta si (E)CX es mayor que cero. El destino debe estar entre -128 a +127 a partir de la ubicación actual. En los procesadores IA-32, ECX se utiliza como el contador de ciclo predeterminado.

Formatos de la instrucción:

LOOP *etiquetacorta*

LOOPW *etiquetacorta*

LOOPD**Ciclo (IA-32)**

O	D	I	S	Z	A	P	C

Decrementa ECX y salta a una etiqueta corta si ECX es mayor que Cero. El destino debe estar entre -128 y +127 bytes a partir de la ubicación actual.

Formato de la instrucción:

LOOPD *etiquetacorta*

**LOOPE,
LOOPZ****Salta si es igual (Cero)**

O	D	I	S	Z	A	P	C

Decrementa (E)CX y salta a una etiqueta corta si (E)CX > 0 y se activa la bandera Cero.

Formatos de la instrucción:

LOOPE *etiquetacorta*

LOOPZ *etiquetacorta*

**LOOPNE,
LOOPNZ****Salta si no es igual (Cero)**

O	D	I	S	Z	A	P	C

Decrementa (E)CX y salta a una etiqueta corta si (E)CX > 0 y se borra la bandera Cero.

Formatos de la instrucción:

LOOPNE *etiquetacorta*

LOOPNZ *etiquetacorta*

MOV**Mover**

O	D	I	S	Z	A	P	C

Copia un byte o palabra de un operando de origen a un operando de destino.

Formatos de la instrucción:

MOV *reg, reg*

MOV *reg, inm*

MOV *mem, reg*

MOV *mem, inm*

MOV *reg, mem*

MOV *mem16, regseg*

MOV *reg16, regseg*

MOV *regseg, mem16*

MOV *regseg, reg16*

MOV\$, MOVSB, MOVSW, MOVSD	<p>Mover cadena</p> <table style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr> <td style="text-align: center;">O</td><td style="text-align: center;">D</td><td style="text-align: center;">I</td><td style="text-align: center;">S</td><td style="text-align: center;">Z</td><td style="text-align: center;">A</td><td style="text-align: center;">P</td><td style="text-align: center;">C</td></tr> <tr> <td style="border: 1px solid black; width: 10px;"></td><td style="border: 1px solid black; width: 10px;"></td></tr> </table> <p>Copia un byte o palabra de la memoria direccionada por DS:(E)SI a la memoria direccionada por ES:(E)DI. MOV\$ requiere que se especifiquen ambos operandos. MOVSB copia un byte, MOVSW copia una palabra, y en el IA-32, MOVSD copia una doble palabra. (E)SI y (E)DI se incrementan o decrementan de acuerdo con el tamaño del operando y el estado de la bandera Dirección. Si la bandera Dirección (DF) = 1, (E)SI y (E)DI se decrementan; si DF = 0, (E)SI y (E)DI se incrementan.</p> <p>Formatos de la instrucción:</p> <pre>MOVSB MOVSW MOVSD MOV\$ dest, origen MOV\$ ES:dest, regseg:origen</pre>	O	D	I	S	Z	A	P	C								
O	D	I	S	Z	A	P	C										

MOVSX	<p>Mover con extensión de signo</p> <table style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr> <td style="text-align: center;">O</td><td style="text-align: center;">D</td><td style="text-align: center;">I</td><td style="text-align: center;">S</td><td style="text-align: center;">Z</td><td style="text-align: center;">A</td><td style="text-align: center;">P</td><td style="text-align: center;">C</td></tr> <tr> <td style="border: 1px solid black; width: 10px;"></td><td style="border: 1px solid black; width: 10px;"></td></tr> </table> <p>Copia un byte o palabra de un operando de origen a un registro de destino y extiende el signo hacia la mitad superior del destino. Esta instrucción se utiliza para copiar un operando de 8 o 16 bits en un destino más grande.</p> <p>Formatos de la instrucción:</p> <table style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr> <td style="text-align: center;">MOV\$X reg32, reg16</td><td style="text-align: center;">MOV\$X reg32, mem16</td></tr> <tr> <td style="text-align: center;">MOV\$X reg16, reg8</td><td style="text-align: center;">MOV\$X reg16, m8</td></tr> </table>	O	D	I	S	Z	A	P	C									MOV\$X reg32, reg16	MOV\$X reg32, mem16	MOV\$X reg16, reg8	MOV\$X reg16, m8
O	D	I	S	Z	A	P	C														
MOV\$X reg32, reg16	MOV\$X reg32, mem16																				
MOV\$X reg16, reg8	MOV\$X reg16, m8																				

MOVZX	<p>Mover con extensión de ceros</p> <table style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr> <td style="text-align: center;">O</td><td style="text-align: center;">D</td><td style="text-align: center;">I</td><td style="text-align: center;">S</td><td style="text-align: center;">Z</td><td style="text-align: center;">A</td><td style="text-align: center;">P</td><td style="text-align: center;">C</td></tr> <tr> <td style="border: 1px solid black; width: 10px;"></td><td style="border: 1px solid black; width: 10px;"></td></tr> </table> <p>Copia un byte o palabra de un operando de origen a un registro de destino y lo extiende con ceros hacia la mitad superior del destino. Esta instrucción se utiliza para copiar un operando de 8 o 16 bits en un destino más grande.</p> <p>Formatos de la instrucción:</p> <table style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr> <td style="text-align: center;">MOV\$X reg32, reg16</td><td style="text-align: center;">MOV\$X reg32, mem16</td></tr> <tr> <td style="text-align: center;">MOV\$X reg16, reg8</td><td style="text-align: center;">MOV\$X reg16, m8</td></tr> </table>	O	D	I	S	Z	A	P	C									MOV\$X reg32, reg16	MOV\$X reg32, mem16	MOV\$X reg16, reg8	MOV\$X reg16, m8
O	D	I	S	Z	A	P	C														
MOV\$X reg32, reg16	MOV\$X reg32, mem16																				
MOV\$X reg16, reg8	MOV\$X reg16, m8																				

MUL	<p>Multiplicar entero sin signo</p> <table style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr> <td style="text-align: center;">O</td><td style="text-align: center;">D</td><td style="text-align: center;">I</td><td style="text-align: center;">S</td><td style="text-align: center;">Z</td><td style="text-align: center;">A</td><td style="text-align: center;">P</td><td style="text-align: center;">C</td></tr> <tr> <td style="border: 1px solid black; width: 10px;"></td><td style="border: 1px solid black; width: 10px;"></td></tr> </table> <p>Multiplica AL, AX o EAX por un operando de origen. Si el origen es de 8 bits, se multiplica por AL y el producto se almacena en AX. Si el origen es de 16 bits, se multiplica por AX y el producto se almacena en DX:AX. Si el origen es de 32 bits, se multiplica por EAX y el producto se almacena en EDX:EAX.</p> <p>Formatos de la instrucción:</p> <table style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr> <td style="text-align: center;">MUL reg</td><td style="text-align: center;">MUL mem</td></tr> </table>	O	D	I	S	Z	A	P	C									MUL reg	MUL mem
O	D	I	S	Z	A	P	C												
MUL reg	MUL mem																		

NEG**Negar**

O	D	I	S	Z	A	P	C
*			*	*	*	*	*

Calcula el complemento a dos del operando de destino y almacena el resultado en el destino.

Formatos de la instrucción:

NEG *reg*

NEG *mem*

NOP**Ninguna operación**

O	D	I	S	Z	A	P	C

Esta instrucción no hace nada, pero puede usarse dentro de un ciclo de sincronización o para alinear una instrucción subsiguiente en un límite de palabra.

Formato de la instrucción:

NOP

NOT**Not**

O	D	I	S	Z	A	P	C

Realiza una operación NOT lógica sobre un operando, invirtiendo cada uno de sus bits.

Formatos de la instrucción:

NOT *reg*

NOT *mem*

OR**OR inclusivo**

O	D	I	S	Z	A	P	C
0			*	*	?	*	0

Realiza una operación OR booleana (a nivel de bits) entre cada bit coincidente en el operando de destino, y cada bit en el operando de origen.

Formatos de la instrucción:

OR *reg, reg*

OR *reg, imm*

OR *mem, reg*

OR *mem, imm*

OR *reg, mem*

OR *acum, imm*

OUT**Salida a un puerto**

O	D	I	S	Z	A	P	C

En procesadores anteriores al IA-32, esta instrucción envía un byte o palabra del acumulador a un puerto. La dirección del puerto puede ser una constante si se encuentra en el rango 0-FFh, o DX puede contener una dirección de puerto entre 0 y FFFFh. En un procesador IA-32, puede enviarse una doble palabra a un puerto.

Formatos de la instrucción:

OUT *imm8, acum*

OUT *DX, acum*

**OUTS,
OUTSB,
OUTSW,
OUTSD****Enviar cadena a un puerto (80286)**

O	D	I	S	Z	A	P	C

Envía una cadena a la que apunta ES:(E)DI a un puerto. El número de puerto se especifica en DX. Para cada valor que se envía, (E)DI se ajusta de la misma forma que LODSB y las instrucciones de primitivas de cadena similares. Puede usarse el prefijo REP con esta instrucción.

Formatos de la instrucción:

OUTS *dest*,DX
OUTSW *dest*,DX

REP OUTSB *dest*,DX
REP OUTSD *dest*,DX

POP**Sacar de la pila**

O	D	I	S	Z	A	P	C

Copia una palabra o doble palabra en la ubicación actual del apuntador de pila al operando de destino, y suma 2 (o 4) a (E)SP.

Formatos de la instrucción:

POP *reg16/r32*
POP *mem16/mem32*

POP *regseg*

**POPA,
POPAD****Sacar todos**

O	D	I	S	Z	A	P	C

Saca 16 bytes de la parte superior de la pila y los coloca en los ocho registros de propósito general, en el siguiente orden: DI, SI, BP, SP, BX, DX, CX, AX. El valor para SP se descarta, por lo que SP no se reasigna. POPA saca hacia registros de 16 bits y POPAD en el IA-32 saca hacia registros de 32 bits.

Formatos de la instrucción:

POPA

POPAD

**POPF,
POPFD****Sacar banderas de la pila**

O	D	I	S	Z	A	P	C
*	*	*	*	*	*	*	*

POPF saca la parte superior de la pila y la coloca en el registro FLAGS de 16 bits. POPFD en el IA-32 saca la parte superior de la pila y la coloca en el registro EFLAGS de 32 bits.

Formatos de la instrucción:

POPF

POPFD

PUSH**Meter en la pila**

Resta 2 de (E)SP y copia el operando de origen en la ubicación de la pila a la que apunta (E)SP. Del 80186 en adelante, puede meterse un valor inmediato en la pila.

Formatos de la instrucción:

PUSH *reg16/reg32*
PUSH *mem16/mem32*

PUSH *regseg*
PUSH *inm16/inm32*

**PUSHA,
PUSHAD****Meter todos (80286)**

Mete los siguientes registros de 16 bits en la pila, en orden: AX, CX, DX, BX, SP, BP, SI y DI. La instrucción PUSHAD para el IA-32 mete EAX, ECX, EDX, EBX, ESP, EBP, ESI y EDI.

Formatos de la instrucción:

PUSHA

PUSHAD

**PUSHF,
PUSHFD****Meter banderas**

PUSHF mete el registro FLAGS de 16 bits en la pila. PUSHFD mete el registro EFLAGS de 32 bits en la pila (IA-32).

Formatos de la instrucción:

PUSHF

PUSHFD

**PUSHW,
PUSHD****Meter en la pila**

PUSHW mete una palabra de 16 bits en la pila, y en el IA-32, PUSHD mete una doble palabra de 32 bits en la pila.

Formatos de la instrucción:

PUSH *reg16/reg32*
PUSH *mem16/mem32*

PUSH *regseg*
PUSH *inm16/inm32*

RCL	Rotar acarreo a la izquierda <table style="margin-left: auto; margin-right: auto;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td>*</td><td></td><td></td><td></td><td></td><td></td><td></td><td>*</td></tr> </table>	O	D	I	S	Z	A	P	C	*							*
O	D	I	S	Z	A	P	C										
*							*										
	Rota el operando de destino a la izquierda, usando el operando de origen para determinar el número de rotaciones. La bandera Acarreo se copia al bit más inferior, y el bit más superior se copia a la bandera Acarreo. El operando <i>imm8</i> debe ser 1 al utilizar el procesador 8086/8088. Formatos de la instrucción: <table style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">RCL <i>reg, imm8</i></td><td style="text-align: right;">RCL <i>mem, imm8</i></td></tr> <tr> <td style="text-align: center;">RCL <i>reg, CL</i></td><td style="text-align: right;">RCL <i>mem, CL</i></td></tr> </table>	RCL <i>reg, imm8</i>	RCL <i>mem, imm8</i>	RCL <i>reg, CL</i>	RCL <i>mem, CL</i>												
RCL <i>reg, imm8</i>	RCL <i>mem, imm8</i>																
RCL <i>reg, CL</i>	RCL <i>mem, CL</i>																

RCR	Rotar acarreo a la derecha <table style="margin-left: auto; margin-right: auto;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td>*</td><td></td><td></td><td></td><td></td><td></td><td></td><td>*</td></tr> </table>	O	D	I	S	Z	A	P	C	*							*
O	D	I	S	Z	A	P	C										
*							*										
	Rota el operando de destino a la derecha, usando el operando de origen para determinar el número de rotaciones. La bandera Acarreo se copia al bit más superior, y el bit más inferior se copia a la bandera Acarreo. El operando <i>imm8</i> debe ser 1 al utilizar el procesador 8086/8088. Formatos de la instrucción: <table style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">RCR <i>reg, imm8</i></td><td style="text-align: right;">RCR <i>mem, imm8</i></td></tr> <tr> <td style="text-align: center;">RCR <i>reg, CL</i></td><td style="text-align: right;">RCR <i>mem, CL</i></td></tr> </table>	RCR <i>reg, imm8</i>	RCR <i>mem, imm8</i>	RCR <i>reg, CL</i>	RCR <i>mem, CL</i>												
RCR <i>reg, imm8</i>	RCR <i>mem, imm8</i>																
RCR <i>reg, CL</i>	RCR <i>mem, CL</i>																

REP	Repetir cadena <table style="margin-left: auto; margin-right: auto;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>	O	D	I	S	Z	A	P	C								
O	D	I	S	Z	A	P	C										
	Repite una instrucción de primitiva de cadena, usando (E)CX como contador. (E)CX se decremente cada vez que se repite la instrucción, hasta que (E)CX = 0. Formato (se muestra con MOVS): <table style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">REP MOVS <i>dest, origin</i></td></tr> </table>	REP MOVS <i>dest, origin</i>															
REP MOVS <i>dest, origin</i>																	

REPcondición	Repetir cadena condicionalmente <table style="margin-left: auto; margin-right: auto;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>	O	D	I	S	Z	A	P	C								
O	D	I	S	Z	A	P	C										
	Repite una instrucción de primitiva de cadena hasta que (E)CX = 0 y mientras sea verdadera una condición de bandera. REPZ (REPE) se repite mientras la bandera Cero esté activa, y REPZ (REPNE) se repite mientras la bandera Cero esté en cero. Sólo deben usarse SCAS y CMPS con REP condición, ya que son las únicas primitivas de cadena que modifican la bandera Cero. Formatos utilizados con SCAS: <table style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">REP SCAS <i>dest</i></td><td style="text-align: right;">REPNE SCAS <i>dest</i></td></tr> <tr> <td style="text-align: center;">REPZ SCASB</td><td style="text-align: right;">REPNE SCASB</td></tr> <tr> <td style="text-align: center;">REPE SCASW</td><td style="text-align: right;">REPNEZ SCASW</td></tr> </table>	REP SCAS <i>dest</i>	REPNE SCAS <i>dest</i>	REPZ SCASB	REPNE SCASB	REPE SCASW	REPNEZ SCASW										
REP SCAS <i>dest</i>	REPNE SCAS <i>dest</i>																
REPZ SCASB	REPNE SCASB																
REPE SCASW	REPNEZ SCASW																

**RET,
RETN,
RETF****Retornar de un procedimiento**

O	D	I	S	Z	A	P	C

Saca una dirección de retorno de la pila. RETN (retorno cercano) saca sólo la parte superior de la pila y la coloca en (E)IP. En modo de direccionamiento real, RETF (retorno lejano) saca la pila y la coloca primero en (E)IP y después en CS. RET puede ser cercana o lejana, dependiendo del atributo especificado o implicado por la directiva PROC. Un operando inmediato opcional de 8 bits indica a la CPU que debe sumar un valor a (E)SP después de sacar la dirección de retorno.

Formatos de la instrucción:

RET
RETN
RETF

RET *imm8*
RETN *imm8*
RETF *imm8*

ROL**Rotar a la izquierda**

O	D	I	S	Z	A	P	C
*							*

Rota el operando de destino a la izquierda, usando el operando de origen para determinar el número de rotaciones. El bit superior se copia en la bandera Acarreo y se mueve hacia la posición del bit más inferior. El operando *imm8* debe ser 1 al utilizar el procesador 8086/8088.

Formatos de la instrucción:

ROL *reg, imm8*
ROL *reg, CL*

ROL *mem, imm8*
ROL *mem, CL*

ROR**Rotar a la derecha**

O	D	I	S	Z	A	P	C
*							*

Rota el operando de destino a la derecha, usando el operando de origen para determinar el número de rotaciones. El bit inferior se copia en la bandera Acarreo y en la posición del bit superior. El operando *imm8* debe ser 1 al utilizar el procesador 8086/8088.

Formatos de la instrucción:

ROR *reg, imm8*
ROR *reg, CL*

ROR *mem, imm8*
ROR *mem, CL*

SAHF**Almacena AH en el registro Flags**

O	D	I	S	Z	A	P	C
			*	*	*	*	*

Copia AH en los bits del 0 al 7 del registro Flags.

Formato de la instrucción:

SAHF

SAL**Desplazamiento aritmético a la izquierda**

O	D	I	S	Z	A	P	C
*			*	*	?	*	*

Desplaza cada bit en el operando de destino a la izquierda, usando el operando de origen para determinar el número de desplazamientos. El bit superior se copia en la bandera Acarreo y el bit inferior se llena con un cero. El operando imm8 debe ser 1 al utilizar el procesador 8086/8088.

Formatos de la instrucción:

SAL *reg, imm8*
SAL *reg, CL*

SAL *mem, imm8*
SAL *mem, CL*

SAR**Desplazamiento aritmético a la derecha**

O	D	I	S	Z	A	P	C
*			*	*	?	*	*

Desplaza cada bit en el operando de destino a la derecha, usando el operando de origen para determinar el número de desplazamientos. El bit inferior se copia en la bandera Acarreo y el bit superior retiene su valor anterior. Este desplazamiento se utiliza a menudo con los operandos con signo, ya que preserva el signo del número. El operando imm8 debe ser 1 al utilizar el procesador 8086/8088.

Formatos de la instrucción:

SAR *reg, imm8*
SAR *reg, CL*

SAR *mem, imm8*
SAR *mem, CL*

SBB**Resta con préstamo**

O	D	I	S	Z	A	P	C
*			*	*	*	*	*

Resta el operando de origen al operando de destino y después resta la bandera Acarreo al destino.

Formatos de la instrucción:

SBB *reg, reg*
SBB *mem, reg*
SBB *reg, mem*

SBB *reg, imm*
SBB *mem, imm*

**SCAS,
SCASB,
SCASW,
SCASD****Explorar cadena**

O	D	I	S	Z	A	P	C
*			*	*	*	*	*

Explora una cadena en memoria a la que apunta ES:(E)DI, en busca de un valor que coincida con el acumulador. SCAS requiere que se especifiquen los operandos. SCASB explora en busca de un valor de 16 bits que coincida con AX, y SCASD explora en busca de un valor de 32 bits que coincida con EAX. (E)DI se incrementa o decremente de acuerdo con el tamaño del operando y el estado de la bandera Dirección. Si DF = 1, (E)DI se decrementa; si DF = 0, (E)DI se incrementa.

Formatos de la instrucción:

SCASB
SCASD
SCAS ES:*dest*

SCASW

SETcondición**Activar condicionalmente**

O	D	I	S	Z	A	P	C

Si la condición dada de la bandera es verdadera, se asigna el valor 1 al byte especificado por el operando de destino. Si la condición de la bandera es falsa, se asigna un valor de 0 al destino. En la tabla B-2 se listan los valores posibles para condición.

Formatos de la instrucción:

SETcond reg8

SETcond mem8

SHL**Desplazar a la izquierda**

O	D	I	S	Z	A	P	C
*			*	*	?	*	*

Desplaza cada bit en el operando de destino a la izquierda, usando el operando de origen para determinar el número de desplazamientos. El bit superior se copia en la bandera Acarreo, y el bit inferior se llena con un cero (en forma idéntica a SAL). El operando *imm8* debe ser 1 si se utiliza el procesador 8086/8088.

Formatos de la instrucción:

SHL reg, imm8
SHL reg, CL

SHL mem, imm8
SHL mem, CL

SHLD**Desplazar a la izquierda con doble precisión (IA-32)**

O	D	I	S	Z	A	P	C
*			*	*	?	*	*

Desplaza los bits del segundo operando hacia el primer operando. El tercer operando indica el número de bits a desplazar. Las posiciones abiertas por el desplazamiento se llenan con los bits más significativos del segundo operando. Éste siempre debe ser un registro, y el tercer operando puede ser un valor inmediato o el registro CL.

Formatos de la instrucción:

SHLD reg16, reg16, imm8
SHLD reg32, reg32, imm8
SHLD reg16, reg16, CL
SHLD reg32, reg32, CL

SHLD mem16, reg16, imm8
SHLD mem32, reg32, imm8
SHLD mem16, reg16, CL
SHLD mem32, reg32, CL

SHR**Desplazar a la derecha**

O	D	I	S	Z	A	P	C
*			*	*	?	*	*

Desplaza cada bit en el operando de destino a la derecha, usando el operando de origen para determinar el número de desplazamientos. El bit superior se llena con un cero, y el bit inferior se copia en la bandera Acarreo. El operando *imm8* debe ser 1 si se utiliza el procesador 8086/8088.

Formatos de la instrucción:

SHR reg, imm8
SHR reg, CL

SHR mem, imm8
SHR mem, CL

SHRD**Desplazar a la derecha con doble precisión (IA-32)**

O	D	I	S	Z	A	P	C
*			*	*	?	*	*

Desplaza los bits del segundo operando hacia el primer operando. El tercer operando indica el número de bits a desplazar. Las posiciones abiertas por el desplazamiento se llenan con los bits menos significativos del segundo operando. Éste siempre debe ser un registro, y el tercer operando puede ser un valor inmediato o el registro CL.

Formatos de la instrucción:

SHRD reg16, reg16, inm8
 SHRD reg32, reg32, inm8
 SHRD reg16, reg16, CL
 SHRD reg32, reg32, CL

SHRD mem16, reg16, inm8
 SHRD mem32, reg32, inm8
 SHRD mem16, reg16, CL
 SHRD mem32, reg32, CL

STC**Activar bandera Acarreo**

O	D	I	S	Z	A	P	C
							1

Establece la bandera Acarreo.

Formato de la instrucción:

STC

STD**Activar bandera Dirección**

O	D	I	S	Z	A	P	C
	1						

Establece la bandera Dirección, haciendo que (E)SI y/o (E)DI se decrementen mediante instrucciones de primitivas de cadena. Por ende, el procesamiento de las cadenas será desde las direcciones superiores, hasta las direcciones inferiores.

Formato de la instrucción:

STD

STI**Activar bandera Interrupción**

O	D	I	S	Z	A	P	C
		1					

Establece la bandera Interrupción, la cual habilita las interrupciones enmascarables. Las interrupciones se deshabilitan de manera automática cuando ocurre una interrupción, por lo que un procedimiento manejador de interrupciones las rehabilita de manera inmediata, usando STI.

Formato de la instrucción:

STI

**STOS,
STOSB,
STOSW,
STOSD**
Almacenar datos de cadena

O	D	I	S	Z	A	P	C

Almacena el acumulador en la ubicación de memoria direccionada por ES:(E)DI. Si se utiliza STOS, debe especificarse un operando de destino. STOSB copia AL a la memoria, STOSW copia AX a la memoria, y STOSD para el IA-32 copia EAX a la memoria. (E)DI se incrementa o decremente de acuerdo con el tamaño del operando y el estado de la bandera Dirección. Si DF = 1, (E)DI se decrementa; si DF = 0, (E)DI se incrementa.

Formatos de la instrucción:

STOSB

STOSD

 STOS *mem*

 STOS ES:*dest*

STOSW

SUB
Restar

O	D	I	S	Z	A	P	C
*			*	*	*	*	*

Resta el operando de destino del operando de origen.

Formatos de la instrucción:

 SUB *reg, reg*

 SUB *mem, reg*

 SUB *reg, mem*

 SUB *reg, imm*

 SUB *mem, imm*

 SUB *acum, imm*
TEST
Probar

O	D	I	S	Z	A	P	C
0			*	*	?	*	0

Prueba bits individuales en el operando de destino, comparándolos con los del operando de origen. Realiza una operación AND lógica que afecta a las banderas, pero no al operando de destino.

Formatos de la instrucción:

 TEST *reg, reg*

 TEST *mem, reg*

 TEST *reg, mem*

 TEST *reg, imm*

 TEST *mem, imm*

 TEST *acum, imm*
WAIT
Esperar al coprocesador

O	D	I	S	Z	A	P	C

Suspende la ejecución de la CPU hasta que el coprocesador termine la instrucción actual.

Formato de la instrucción:

WAIT

XADD**Intercambiar y sumar (Intel486)**

O	D	I	S	Z	A	P	C
*			*	*	*	*	*

Suma el operando de origen al operando de destino. Al mismo tiempo, el valor de destino original se mueve al operando de origen.

Formatos de la instrucción:

XADD *reg, reg*

XADD *mem, reg*

XCHG**Intercambiar**

O	D	I	S	Z	A	P	C

Intercambia el contenido de los operandos de origen y de destino.

Formatos de la instrucción:

XCH *reg, reg*

XCH *reg, mem*

XCH *mem, reg*

**XLAT,
XLATB****Traducir byte**

O	D	I	S	Z	A	P	C

Usa el valor en AL como índice en una tabla a la que apunta DS:BX. El byte al que apunta el índice se mueve hacia AL. Puede especificarse un operando para proporcionar una redefinición de segmento. Se puede usar XLATB en vez de XLAT.

Formatos de la instrucción:

XLAT

XLAT *mem*

XLAT *reg:seg:mem*

XLATB

XOR**OR exclusivo**

O	D	I	S	Z	A	P	C
0			*	*	?	*	0

Se aplica un OR exclusivo a cada bit en el operando de origen con su bit correspondiente en el destino. El bit de destino es 1 sólo cuando los bits de origen y de destino son distintos.

Formatos de la instrucción:

XOR *reg, reg*

XOR *mem, reg*

XOR *reg, mem*

XOR *reg, imm*

XOR *mem, imm*

XOR *acum, imm*

B.3 Instrucciones de punto flotante

La tabla B-3 contiene una lista de todas las instrucciones de punto flotante de los procesadores IA-32, con descripciones breves y formatos de los operandos. Por lo general, las instrucciones se agrupan por función en vez de que sólo se ordenen de manera alfabética. Por ejemplo, la instrucción FIADD sigue justo después de FADD y de FADDP, ya que realiza la misma operación pero con conversión de enteros.

Para obtener toda la información completa acerca de las instrucciones de punto flotante IA-32, consulte el *Manual del desarrollador de software para la arquitectura Intel IA-32 (IA-32 Intel Architecture Software Developer's Manual)*, Vol. 2A. La palabra *pila* en esta tabla se refiere a la pila de registros de la FPU. (La tabla B-1 presenta muchos de los símbolos utilizados para describir los formatos y operandos de las instrucciones de punto flotante.)

Tabla B-3 Instrucciones de punto flotante IA-32.

Instrucción	Descripción	
F2XMI	Calcular $2^x - 1$. Sin operandos	
FABS	Valor absoluto. Borra el bit de signo de ST(0). Sin operandos	
FADD	Sumar punto flotante. Suma los operandos de origen y de destino, almacena la suma en el operando de destino. Formatos:	
	FADD	Suma ST(0) a ST(1), y saca de la pila
	FADD m32pf	Suma m32pf a ST(0)
	FADD m64pf	Suma m64pf a ST(0)
	FADD ST(0),ST(i)	Suma ST(i) a ST(0)
	FADD ST(i),ST(0)	Suma ST(0) a ST(i)
FADDP	Sumar punto flotante y sacar. Realiza la misma operación que FADD, después saca de la pila. Formato: FADDP ST(i),ST(0)	Suma ST(0) a ST(i)
FIADD	Convertir entero a punto flotante y sumar. Suma los operandos de destino y de origen, almacena la suma en el operando de destino. Formatos:	
	FIADD m32ent	Suma m32ent a ST(0)
	FIADD m16ent	Suma m16ent a ST(0)
FBLD	Cargar decimal codificado en binario. Convierte el operando de origen BCD al formato de punto flotante con precisión doble extendida y lo mete en la pila. Formato: FBLD m80bcd	Mete m80bcd en la pila de registros
FBSTP	Almacenar entero BCD y sacar. Convierte el valor en el registro ST(0) a un entero BCD empaquetado de 18 dígitos, almacena el resultado en el operando de destino y saca de la pila de registros. Formato: FBSTP m80bcd	Mete ST(0) en m80bcd y saca de la pila
FCHS	Cambiar signo. Complementa el signo de ST(0). Sin operandos	
FCLEX	Borrar excepciones. Borra las banderas de excepciones de punto flotante (PE, UE, OE, ZE, DE e IE), la bandera de estado de resumen de excepciones (ES), la bandera de fallo de pila (SF), y la bandera ocupado (B) en la palabra de estado de la FPU. Sin operandos. FNCLEX realiza la misma operación, sin comprobar las excepciones de punto flotante desenmascaradas que haya pendientes	
FCMOVcc	Movimiento condicional de punto flotante. Prueba las banderas de estado en EFLAGS, mueve el operando de origen (segundo operando) al operando de destino (primer operando) si la condición de prueba dada es verdadera. Formatos:	
	FCMOVB ST(0),ST(i)	Mueve si es menor
	FCMOVE ST(0),ST(i)	Mueve si es igual
	FCMOVEBE ST(0),ST(i)	Mueve si es menor o igual
	FCMOVU ST(0),ST(i)	Mueve si está desordenado
	FCMOVNB ST(0),ST(i)	Mueve si no es menor
	FCMOVNE ST(0),ST(i)	Mueve si no es igual
	FCMOVNBE ST(0),ST(i)	Mueve si no es menor o igual
	FCMOVNU ST(0),ST(i)	Mueve si no está desordenado

Instrucción	Descripción										
FCOM	<p>Comparar valores de punto flotante. Compara ST(0) con el operando de origen y establece las banderas de código de condición C0, C2 y C3 en la palabra de estado de la FPU, de acuerdo con los resultados. Formatos:</p> <table> <tr> <td>FCOM <i>m32pf</i></td><td>Compara ST(0) con <i>m32pf</i></td></tr> <tr> <td>FCOM <i>m64pf</i></td><td>Compara ST(0) con <i>m64pf</i></td></tr> <tr> <td>FCOM ST(<i>i</i>)</td><td>Compara ST(0) con ST(<i>i</i>)</td></tr> <tr> <td>FCOM</td><td>Compara ST(0) con ST(1)</td></tr> </table> <p>FCOMP realiza la misma operación que FCOM y después saca de la pila. FCOMPP hace lo mismo que FCOM y después saca de la pila dos veces. FUCOM, FUCOMP y FUCOMPP son iguales que FCOM, FCOMP y FCOMPP, respectivamente, excepto que comprueban valores desordenados</p>	FCOM <i>m32pf</i>	Compara ST(0) con <i>m32pf</i>	FCOM <i>m64pf</i>	Compara ST(0) con <i>m64pf</i>	FCOM ST(<i>i</i>)	Compara ST(0) con ST(<i>i</i>)	FCOM	Compara ST(0) con ST(1)		
FCOM <i>m32pf</i>	Compara ST(0) con <i>m32pf</i>										
FCOM <i>m64pf</i>	Compara ST(0) con <i>m64pf</i>										
FCOM ST(<i>i</i>)	Compara ST(0) con ST(<i>i</i>)										
FCOM	Compara ST(0) con ST(1)										
FCOMI	<p>Comparar valores de punto flotante y establecer EFLAGS. Realiza una comparación desordenada de los registros ST(0) y ST(<i>i</i>), y establece las banderas de estado (ZF, PF, CF) en el registro EFLAGS de acuerdo con los resultados. Formato:</p> <table> <tr> <td>FCOMI ST(0),ST(<i>i</i>)</td><td>Compara ST(0) con ST(<i>i</i>)</td></tr> </table> <p>FCOMIP realiza la misma tarea que FCOMI, y después saca de la pila. FUCOMI y FUCOMIP realizan comprobaciones para valores desordenados</p>	FCOMI ST(0),ST(<i>i</i>)	Compara ST(0) con ST(<i>i</i>)								
FCOMI ST(0),ST(<i>i</i>)	Compara ST(0) con ST(<i>i</i>)										
FCOS	Coseno. Calcula el coseno de ST(0) y almacena el resultado en ST(0). La entrada debe estar en radianes. Sin operandos										
FDECSTP	Decrementar apuntador de la parte superior de la pila. Resta 1 al campo TOP de la palabra de estado de la FPU, con lo que se rota la pila de manera eficaz. Sin operandos										
FDIV	<p>Dividir punto flotante y sacar. Divide el operando de destino entre el operando de origen y almacena el resultado en la ubicación de destino. Formatos:</p> <table> <tr> <td>FDIV</td><td>ST(1) = ST(1) / ST(0), y saca de la pila</td></tr> <tr> <td>FDIV <i>m32pf</i></td><td>ST(0) = ST(0) / <i>m32pf</i></td></tr> <tr> <td>FDIV <i>m64pf</i></td><td>ST(0) = ST(0) / <i>m64pf</i></td></tr> <tr> <td>FDIV ST(0),ST(<i>i</i>)</td><td>ST(0) = ST(0) / ST(<i>i</i>)</td></tr> <tr> <td>FDIV ST(<i>i</i>),ST(0)</td><td>ST(<i>i</i>) = ST(<i>i</i>) / ST(0)</td></tr> </table>	FDIV	ST(1) = ST(1) / ST(0), y saca de la pila	FDIV <i>m32pf</i>	ST(0) = ST(0) / <i>m32pf</i>	FDIV <i>m64pf</i>	ST(0) = ST(0) / <i>m64pf</i>	FDIV ST(0),ST(<i>i</i>)	ST(0) = ST(0) / ST(<i>i</i>)	FDIV ST(<i>i</i>),ST(0)	ST(<i>i</i>) = ST(<i>i</i>) / ST(0)
FDIV	ST(1) = ST(1) / ST(0), y saca de la pila										
FDIV <i>m32pf</i>	ST(0) = ST(0) / <i>m32pf</i>										
FDIV <i>m64pf</i>	ST(0) = ST(0) / <i>m64pf</i>										
FDIV ST(0),ST(<i>i</i>)	ST(0) = ST(0) / ST(<i>i</i>)										
FDIV ST(<i>i</i>),ST(0)	ST(<i>i</i>) = ST(<i>i</i>) / ST(0)										
FDIVP	<p>Dividir punto flotante y sacar. Igual que FDIV, y después saca de la pila. Formato:</p> <table> <tr> <td>FDIVP ST(<i>i</i>),ST(0)</td><td>ST(<i>i</i>) = ST(<i>i</i>) / ST(0), y saca de la pila</td></tr> </table>	FDIVP ST(<i>i</i>),ST(0)	ST(<i>i</i>) = ST(<i>i</i>) / ST(0), y saca de la pila								
FDIVP ST(<i>i</i>),ST(0)	ST(<i>i</i>) = ST(<i>i</i>) / ST(0), y saca de la pila										
FIDIV	<p>Convertir entero en punto flotante y dividir. Después de convertir, realiza la misma operación que FDIV. Formatos:</p> <table> <tr> <td>FIDIV <i>m32ent</i></td><td>ST(0) = ST(0) / <i>m32ent</i></td></tr> <tr> <td>FIDIV <i>m16ent</i></td><td>ST(0) = ST(0) / <i>m16ent</i></td></tr> </table>	FIDIV <i>m32ent</i>	ST(0) = ST(0) / <i>m32ent</i>	FIDIV <i>m16ent</i>	ST(0) = ST(0) / <i>m16ent</i>						
FIDIV <i>m32ent</i>	ST(0) = ST(0) / <i>m32ent</i>										
FIDIV <i>m16ent</i>	ST(0) = ST(0) / <i>m16ent</i>										
FDIVR	<p>Dividir a la inversa. Divide el operando de origen entre el operando de destino y almacena el resultado en la ubicación de destino. Formatos:</p> <table> <tr> <td>FDIVR</td><td>ST(0) = ST(0) / ST(1), y saca de la pila</td></tr> <tr> <td>FDIVR <i>m32pf</i></td><td>ST(0) = <i>m32pf</i> / ST(0)</td></tr> <tr> <td>FDIVR <i>m64pf</i></td><td>ST(0) = <i>m64pf</i> / ST(0)</td></tr> <tr> <td>FDIVR ST(0),ST(<i>i</i>)</td><td>ST(0) = ST(<i>i</i>) / ST(0)</td></tr> <tr> <td>FDIVR ST(<i>i</i>),ST(0)</td><td>ST(<i>i</i>) = ST(0) / ST(1)</td></tr> </table>	FDIVR	ST(0) = ST(0) / ST(1), y saca de la pila	FDIVR <i>m32pf</i>	ST(0) = <i>m32pf</i> / ST(0)	FDIVR <i>m64pf</i>	ST(0) = <i>m64pf</i> / ST(0)	FDIVR ST(0),ST(<i>i</i>)	ST(0) = ST(<i>i</i>) / ST(0)	FDIVR ST(<i>i</i>),ST(0)	ST(<i>i</i>) = ST(0) / ST(1)
FDIVR	ST(0) = ST(0) / ST(1), y saca de la pila										
FDIVR <i>m32pf</i>	ST(0) = <i>m32pf</i> / ST(0)										
FDIVR <i>m64pf</i>	ST(0) = <i>m64pf</i> / ST(0)										
FDIVR ST(0),ST(<i>i</i>)	ST(0) = ST(<i>i</i>) / ST(0)										
FDIVR ST(<i>i</i>),ST(0)	ST(<i>i</i>) = ST(0) / ST(1)										
FDIVRP	<p>Dividir a la inversa y sacar. Realiza la misma operación que FDIVR, y después saca de la pila. Formato:</p> <table> <tr> <td>FDIVRP ST(<i>i</i>),ST(0)</td><td>ST(<i>i</i>) = ST(0) / ST(<i>i</i>), y saca de la pila</td></tr> </table>	FDIVRP ST(<i>i</i>),ST(0)	ST(<i>i</i>) = ST(0) / ST(<i>i</i>), y saca de la pila								
FDIVRP ST(<i>i</i>),ST(0)	ST(<i>i</i>) = ST(0) / ST(<i>i</i>), y saca de la pila										
FIDIVR	<p>Convertir entero en punto flotante y dividir a la inversa. Después de convertir, realiza la misma operación que FDIVR. Formatos:</p> <table> <tr> <td>FIDIVR <i>m32ent</i></td><td>ST(0) = <i>m32ent</i> / ST(0)</td></tr> <tr> <td>FIDIVR <i>m16ent</i></td><td>ST(0) = <i>m16ent</i> / ST(0)</td></tr> </table>	FIDIVR <i>m32ent</i>	ST(0) = <i>m32ent</i> / ST(0)	FIDIVR <i>m16ent</i>	ST(0) = <i>m16ent</i> / ST(0)						
FIDIVR <i>m32ent</i>	ST(0) = <i>m32ent</i> / ST(0)										
FIDIVR <i>m16ent</i>	ST(0) = <i>m16ent</i> / ST(0)										
FFREE	<p>Liberar registro de punto flotante. Deja el registro vacío, usando la palabra Tag. Formato:</p> <table> <tr> <td>FFREE ST(<i>i</i>)</td><td>ST(<i>i</i>) = vacía</td></tr> </table>	FFREE ST(<i>i</i>)	ST(<i>i</i>) = vacía								
FFREE ST(<i>i</i>)	ST(<i>i</i>) = vacía										

(Continúa)

Tabla B-3 (Continuación)

Instrucción	Descripción	
FICOM	Comparar entero. Compara el valor en ST(0) con un operando de origen entero y establece las banderas de código de condición C0, C2 y C3 de acuerdo con los resultados. El operando de origen entero se convierte a punto flotante antes de la comparación. Formatos: FICOM <i>m32ent</i> Compara ST(0) con <i>m32ent</i> FICOM <i>m16ent</i> Compara ST(0) con <i>m16ent</i> FICOMP realiza la misma operación que FICOM y después saca de la pila	
FILD	Convertir entero a punto flotante y cargar en pila de registros. Formatos: FILD <i>m16ent</i> Mete <i>m16ent</i> en pila de registros FILD <i>m32ent</i> Mete <i>m32ent</i> en pila de registros FILD <i>m64ent</i> Mete <i>m64ent</i> en pila de registros	
FINCSTP	Incrementar apuntador de la parte superior de la pila. Suma 1 al campo TOP de la palabra de estado de la FPU. Sin operandos	
FINIT	Inicializar unidad de punto flotante. Establece los registros de control, de estado, de etiqueta, apuntador de instrucciones y apuntador de datos a sus estados predeterminados. La palabra de control se establece a 037FH (redondear al número más cercano, todas las excepciones enmascaradas, precisión de 64 bits). La palabra de estado se borra (no se activan banderas de excepción, TOP = 0). Los registros de datos en la pila de registros permanecen sin cambio, pero se marcan como vacíos. Sin operandos. FNINIT realiza la misma operación sin comprobar las excepciones de punto flotante no enmascaradas que haya pendientes	
FIST	Almacenar entero en operando de memoria. Almacena ST(0) en un operando de memoria entero con signo, y lo redondea de acuerdo con el campo RC en la palabra de control de la FPU. Formatos: FIST <i>m16ent</i> Almacena ST(0) en <i>m16ent</i> FIST <i>m32ent</i> Almacena ST(0) en <i>m32ent</i> FISTP realiza la misma operación que FIST, y después saca de la pila de registros. Tiene un formato adicional: FIST <i>m64ent</i> Almacena ST(0) en <i>m64ent</i> , y saca de la pila	
FISTTP	Almacenar entero con truncamiento. Realiza la misma operación que FIST, pero trunca automáticamente el entero y saca de la pila. Formatos: FISTTP <i>m16ent</i> Almacena ST(0) en <i>m16ent</i> , y saca de la pila FISTTP <i>m32ent</i> Almacena ST(0) en <i>m32ent</i> , y saca de la pila FISTTP <i>m64ent</i> Almacena ST(0) en <i>m64ent</i> , y saca de la pila	
FLD	Cargar valor de punto flotante en la pila de registros. Formatos: FLD <i>m32pf</i> Mete <i>m32pf</i> en la pila de registros FLD <i>m64pf</i> Mete <i>m64pf</i> en la pila de registros FLD <i>m80pf</i> Mete <i>m80pf</i> en la pila de registros FLD ST(i) Mete ST(i) en la pila de registros	
FLD1	Carga +1.0 en la pila de registros. Sin operandos	
FLDL2T	Carga log ₂ 10 en la pila de registros. Sin operandos	
FLDL2E	Carga log _e 2 en la pila de registros. Sin operandos	
FLDPi	Carga pi en la pila de registros. Sin operandos	
FLDLG2	Carga log ₁₀ 2 en la pila de registros. Sin operandos	
FLDLN2	Carga log _e 2 en la pila de registros. Sin operandos	
FLDZ	Carga +0.0 en la pila de registros. Sin operandos	
FLDCW	Carga la palabra de control de la FPU desde un valor de memoria de 16 bits. Formato: FLDCW <i>m2bytes</i> Carga la palabra de control de la FPU desde <i>m2bytes</i>	
FLDENV	Carga el entorno de la FPU desde memoria en la FPU. Formato: FLDENV <i>m14/28bytes</i> Carga el entorno de la FPU desde memoria	

Instrucción	Descripción	
FMUL	Multiplicar punto flotante. Multiplica los operandos de origen y de destino, y almacena el producto en el operando de destino. Formatos:	
	FMUL FMUL <i>m32pf</i> FMUL <i>m64pf</i> FMUL ST(0),ST(i) FMUL ST(i),ST(0)	ST(1) = ST(1) * ST(0), y saca de la pila ST(0) = ST(0) * <i>m32pf</i> ST(0) = ST(0) * <i>m64pf</i> ST(0) = ST(0) * ST(i) ST(i) = ST(i) * ST(0)
FMULP	Multiplicar punto flotante y sacar. Realiza la misma operación que FMUL, y después saca de la pila. Formato: FMULP ST(i),ST(0)	ST(i) = ST(i) * ST(0), y saca de la pila
FIMUL	Convertir entero y multiplicar. Convierte el operando de origen en un número de punto flotante, lo multiplica por ST(0) y almacena el producto en ST(0). Formatos: FIMUL <i>m16ent</i> FIMUL <i>m32ent</i>	
FNOP	Ninguna operación. Sin operandos	
FPATAN	Arcotangente parcial. Sustituye ST(1) con arctan(ST(1)/ST(0)) y saca de la pila de registros. Sin operandos	
FPREM	Residuo parcial. Sustituye ST(0) con el residuo que se obtiene al dividir ST(0) entre ST(1). Sin operandos. FPREM1 es similar, ya que sustituye ST(0) con el residuo IEEE que se obtiene al dividir ST(0) entre ST(1)	
FPTAN	Tangente parcial. Sustituye ST(0) con su tangente y mete 1.0 en la pila de la FPU. La entrada debe estar en radianes. Sin operandos	
FRNDINT	Redondear a entero. Redondea ST(0) al valor entero más cercano. Sin operandos	
FRSTOR	Restaurar el estado de la FPU x87. Carga el estado de la FPU (entorno de operación y pila de registros) del área de memoria especificada por el operando de origen. Formato: FRSTOR <i>m94/108bytes</i>	
FSAVE	Almacenar el estado de la FPU x87. Almacena el estado actual de la FPU (entorno de operación y pila de registros) en la memoria especificada por el operando de destino, y después reinicializa la FPU. Formato: FSAVE <i>m94/108bytes</i>	
	FNSAVE realiza la misma operación sin comprobar las excepciones de punto flotante desenmascaradas que haya pendientes	
FSCALE	Escalar. Trunca el valor en ST(1) a un valor entero y lo suma al exponente del operando de destino ST(0). Sin operandos	
FSIN	Seno. Sustituye ST(0) con su seno. La entrada debe estar en radianes. Sin operandos	
FSINCOS	Seno y coseno. Calcula el seno y el coseno de ST(0). La entrada debe estar en radianes. Sustituye ST(0) con el seno y mete el coseno en la pila de registros. Sin operandos	
FSQRT	Raíz cuadrada. Sustituye ST(0) con su raíz cuadrada. Sin operandos	
FST	Almacenar valor de punto flotante. Formatos: FST <i>m32pf</i> FST <i>m64pf</i> FST ST(i)	Copia ST(0) a <i>m32pf</i> Copia ST(0) a <i>m64pf</i> Copia ST(0) a ST(i)
	FSTP realiza la misma operación que FST, y después saca de la pila. Tiene un formato adicional: FSTP <i>m80pf</i>	Copia ST(0) a <i>m80pf</i> , y saca de la pila
FSTCW	Almacenar palabra de control de la FPU. Formato: FLDCW <i>m2bytes</i>	Almacena palabra de control de FPU en <i>m2bytes</i>
	FNSTCW realiza la misma operación sin comprobar las excepciones de punto flotante desenmascaradas que haya pendientes	

(Continúa)

Tabla B-3 (Continuación)

Instrucción	Descripción	
FSTENV	Almacenar entorno de la FPU. Almacena el entorno de la FPU en una estructura m14bytes o m28bytes, dependiendo de si el procesador está en modo real o en modo protegido. Formato: FSTENV <i>opmem</i>	Almacenar entorno de la FPU en <i>opmem</i> FNSTENV realiza la misma operación sin comprobar las excepciones de punto flotante desenmascaradas que haya pendientes
FSTSW	Almacenar palabra de estado de la FPU. Formatos: FSTSW <i>m2bytes</i> FSTSW AX	Almacena palab. estado FPU en <i>m2bytes</i> Almacena palab. estado FPU en registro AX FNSTSW realiza la misma operación sin comprobar las excepciones de punto flotante desenmascaradas que haya pendientes
FSUB	Restar número de punto flotante. Resta el operando de origen del operando de destino y almacena la diferencia en la ubicación de destino. Formatos: FSUB FSUB <i>m32pf</i> FSUB <i>m64pf</i> FSUB ST(0),ST(i) FSUB ST(i),ST(0)	$ST(0) = ST(1) - ST(0)$, y saca de la pila $ST(0) = ST(0) - m32pf$ $ST(0) = ST(0) - m64pf$ $ST(0) = ST(0) - ST(i)$ $ST(i) = ST(i) - ST(0)$
FSUBP	Restar punto flotante y sacar. La instrucción FSUBP realiza la misma operación que FSUB, y después saca de la pila. Formato: FSUBP ST(i),ST(0)	$ST(i) = ST(i) - ST(0)$, y saca de la pila
FISUB	Convertir entero en punto flotante y restar. Convierte el operando de origen a punto flotante, lo resta de ST(0) y almacena el resultado en ST(0). Formatos: FISUB <i>m16ent</i> FISUB <i>m32ent</i>	$ST(0) = ST(0) - m16ent$ $ST(0) = ST(0) - m32ent$
FSUBR	Restar punto flotante a la inversa. Resta el operando de destino del operando de origen y almacena la diferencia en la ubicación de destino. Formatos: FSUBR FSUBR <i>m32pf</i> FSUBR <i>m64pf</i> FSUBR ST(0),ST(i) FSUBR ST(i),ST(0)	$ST(0) = ST(1) - ST(0)$, y saca de la pila $ST(0) = m32pf - ST(0)$ $ST(0) = m64pf - ST(0)$ $ST(0) = ST(i) - ST(0)$ $ST(i) = ST(0) - ST(i)$
FSUBRP	Restar punto flotante a la inversa y sacar. La instrucción FSUBRP realiza la misma operación que FSUB, y después saca de la pila. Formato: FSUBRP ST(i),ST(0)	$ST(i) = ST(0) - ST(i)$, y saca de la pila
FISUBR	Convertir entero y restar punto flotante a la inversa. Después de convertir en punto flotante, realiza la misma operación que FSUBR. Formatos: FISUBR <i>m16ent</i> FISUBR <i>m32ent</i>	
FTST	Probar. Compara ST(0) con 0.0 y establece las banderas de código de condición en la palabra de estado de la FPU. Sin operandos	
FWAIT	Esperar. Espera a que se completen todos los manejadores de excepciones de punto flotante pendientes. Sin operandos	
FXAM	Examinar. Examina ST(0) y establece las banderas de código de condición en la palabra de estado de la FPU. Sin operandos	
FXCH	Intercambiar contenido de registros. Formatos: FXCH ST(i) FXCH	Intercambia ST(0) y ST(i) Intercambia ST(0) y ST(1)

Instrucción	Descripción
FXRSTOR	Restaurar estado de FPU x87, Tecnología MMX, SSE y SSE2. Vuelve a cargar los registros de la FPU, la tecnología MMX, XMM y MXCSR de la imagen de memoria especificada en el operando de origen. Formato: <i>FXRSTOR m512bytes</i>
FXSAVE	Guardar estado de FPU x87, Tecnología MMX, SSE y SSE2. Guarda el estado actual de los registros de la FPU, la tecnología MMX, XMM y MXCSR en la imagen de memoria especificada en el operando de destino. Formato: <i>FXRSAVE m512bytes</i>
FXTRACT	Extraer exponente y mantisa. Separa el origen en ST(0) en su exponente y su mantisa, almacena el exponente en ST(0), y mete la mantisa en la pila de registros. Sin operandos
FYL2X	Calcular $y * \log_2 x$. El registro ST(1) contiene el valor de y, y ST(0) contiene el valor de x. La pila se saca, por lo que el resultado se deja en ST(0). Sin operandos
FYL2XP1	Calcular $y * \log_2(x + 1)$. El registro ST(1) contiene el valor de y, y ST(0) contiene el valor de x. La pila se saca, por lo que el resultado se deja en ST(0). Sin operandos

C

INTERRUPCIONES del BIOS y de MS-DOS

- C.1 Introducción
- C.2 Interrupciones de la PC
- C.3 Funciones de la interrupción 21H (Servicios de MS-DOS)
- C.4 Funciones de la interrupción 10H (BIOS de video)
- C.5 Funciones INT 16h del BIOS de teclado
- C.6 Funciones del ratón (INT 33h)

C.1 Introducción

Este apéndice presenta agrupados algunos de los números de interrupciones de uso más común:

- Lista general de interrupciones de la PC, que corresponden a la tabla de vectores de interrupción almacenada en los primeros 1024 bytes de memoria.
- Funciones INT 21h de MS-DOS.
- Funciones INT 10h del BIOS de video.
- Funciones INT 16h del BIOS de teclado.
- Funciones INT 33h del ratón.

La documentación de las interrupciones de la PC es una enorme tarea, debido a las distintas versiones de MS-DOS, así como de los diversos extensores de DOS y los controladores del hardware de la PC. La fuente definitiva de información sobre las interrupciones es la Lista de interrupciones de Ralf Brown (Ralf Brown's Interrupt List), disponible en varios formatos en Web. Mi formato favorito es la versión de HTML, que está disponible en www.ctyme.com/rbrown.htm. Los URLs Web cambian con frecuencia, por lo que es conveniente que revise el sitio Web de nuestro libro para obtener los vínculos actualizados a esta lista y a otros sitios Web sobre lenguaje ensamblador.

C.2 Interrupciones de la PC

Tabla C-1 Lista general de números de interrupciones de la PC.^a

Operación	Descripción
0	<i>Error de división.</i> Generada por la CPU: se activa cuando hay un intento de dividir entre cero
1	<i>Paso individual.</i> Generada por la CPU: se activa cuando se activa la bandera Trap de la CPU
2	<i>Interrupción no enmascarable.</i> Hardware externo: se activa cuando ocurre un error en la memoria
3	<i>Punto de interrupción.</i> Generada por la CPU: se activa cuando se ejecuta la instrucción 0CCh (INT 3)
4	<i>Desbordamiento detectado por INTO.</i> Generada por la CPU: se activa cuando se ejecuta la instrucción INTO y se activa la bandera Desbordamiento
5	<i>Imprimir pantalla.</i> Se activa mediante la instrucción INT 5 o al oprimir las teclas Mayús-ImprPant
6	<i>Código de operación inválido (80286+)</i>
7	<i>Extensión del procesador no disponible (80286+)</i>
8	IRQ0: <i>interrupción del temporizador del sistema.</i> Actualiza el reloj del BIOS 18.2 veces por segundo. Para su propia programación, vea INT 1Ch
9	IRQ1: <i>interrupción de hardware del teclado.</i> Se activa cuando se oprime una tecla. Lee la tecla del puerto del teclado y la almacena en el búfer de escritura adelantada
0A	IRQ2: <i>controlador de interrupciones programable</i>
0B	IRQ3: comunicaciones seriales (COM2)
0C	IRQ4: comunicaciones seriales (COM1)
0D	IRQ5: disco fijo
0E	IRQ6: <i>interrupción del disquete.</i> Se activa cuando hay una búsqueda de disco en progreso
0F	IRQ7: <i>impresora paralela</i>
10	<i>Servicios de video.</i> Rutinas para manipular la pantalla de video (vea la lista completa en la tabla C-3)
11	<i>Revisión de equipo.</i> Devuelve una palabra que muestra todos los periféricos conectados al sistema
12	<i>Tamaño de memoria.</i> Devuelve la cantidad de memoria (en bloques de 1024 bytes) en AX
13	<i>Servicios de disco flexible.</i> Restablece el controlador de disco, obtiene el estado del acceso más reciente al disco, lee y escribe en sectores físicos, y da formato a un disco
14	<i>Servicios de puerto asíncrono (serial).</i> Inicializa y lee o escribe en el puerto de comunicaciones asíncronas, y devuelve el estado del puerto
15	Controlador de casete
16	<i>Servicios de teclado.</i> Lee e inspecciona la entrada del teclado (vea la lista completa en la tabla C-4)
17	<i>Servicios de impresora.</i> Inicializa, imprime y devuelve el estado de la impresora
18	<i>BASIC de ROM.</i> Ejecuta el lenguaje BASIC de casete en la ROM
19	<i>Cargador de arranque (bootstrap loader).</i> Reinicio para MS-DOS

(Continúa)

Tabla C-1 (*Continuación*)

Operación	Descripción
1A	<i>Hora del día.</i> Obtiene el número de pulsaciones del temporizador desde que se encendió la máquina, o establece el contador a un nuevo valor. Las pulsaciones ocurren 18.2 veces por segundo
1B	<i>Interrupción del teclado.</i> Este manejador de interrupciones se ejecuta mediante INT 9h, al oprimir CTRL-INTER
1C	<i>Interrupción del temporizador del usuario.</i> Rutina vacía, que se ejecuta 18.2 veces por segundo. Usted puede utilizarla en sus propios programas
1D	<i>Parámetros de video.</i> Apunta a una tabla que contiene inicialización e información para el chip controlador de video
1E	<i>Parámetros de disquete.</i> Apunta a una tabla que contiene información de inicialización para el controlador de disquete
1F	<i>Tabla de gráficos.</i> Fuente de gráficos de 8 × 8. La tabla se mantiene en la memoria de todos los caracteres de gráficos extendidos, con códigos ASCII mayores a 127
20	<i>Terminar programa.</i> Termina un programa COM (es mejor usar la función 4Ch de INT 21h)
21	<i>Servicios de MS-DOS</i> (vea la lista completa en la tabla C-2)
22	<i>Dirección de terminación de MS-DOS.</i> Apunta a la dirección del programa o proceso padre. Cuando termina el programa actual, ésta es la dirección de retorno
23	<i>Dirección de interrupción de MS-DOS.</i> MS-DOS salta aquí cuando se oprime CTRL-INTER
24	<i>Dirección de error crítico de MS-DOS.</i> DOS salta a esta dirección cuando hay un error crítico en el programa actual, como un error de medios de disco
25	<i>Lectura absoluta de disco</i> (obsoleta)
26	<i>Escritura absoluta de disco</i> (obsoleta)
27	<i>Terminar y permanecer residente</i> (obsoleta)
28-FF	(Reservada)
33	<i>Ratón de Microsoft.</i> Funciones para rastrear y controlar el ratón
34-3E	Emulación de punto flotante
3F	Administrador superpuesto (Overlay Manager)
40-41	<i>Servicios de disco fijo.</i> Controlador de disco fijo
42-5F	Reservada: usos especializados
60-6B	Disponible para que la utilicen los programas de aplicaciones
6C-7F	Reservada: usos especializados
80-F0	Reservada: utilizada por ROM BASIC
F1-FF	Disponible para los programas de aplicaciones

^a Fuentes de información: Ray Duncan, *Advanced MS-DOS*, 2^a edición, Microsoft Press, 1998. *Ralf Brown's Interrupt List*, disponible en Web.

C.3 Funciones de la interrupción 21H (Servicios de MS-DOS)

Hay tantos servicios de MS-DOS disponibles a través de INT 21h, que no es posible documentarlos todos aquí. En vez de ello, la tabla C-2 muestra un resumen de las generalidades acerca de las funciones de uso común.

Tabla C-2 Funciones de la interrupción 21h (Servicios de MS-DOS).

Operación	Descripción
1	<i>Leer carácter de la entrada estándar.</i> Si no hay un carácter listo, espera la entrada. Devuelve: AL = carácter
2	<i>Escribir carácter a la salida estándar.</i> Recibe: DL = carácter
3	<i>Leer carácter de entrada auxiliar estándar</i> (puerto serial)
4	<i>Escribir carácter en salida auxiliar estándar</i> (puerto serial)
5	<i>Escribir carácter en impresora.</i> Recibe: DL = carácter
6	<i>Dirigir entrada/salida de consola.</i> Si DL = FFh, lee un carácter en espera de la entrada estándar. Si DL es cualquier otro valor, escribe el carácter que hay en DL a la salida estándar
7	<i>Dirigir entrada de carácter sin eco.</i> Espera un carácter de la entrada estándar. Devuelve: AL = carácter
8	<i>Entrada de carácter sin eco.</i> Espera un carácter del dispositivo de entrada estándar. Devuelve: AL = carácter. El carácter no se imprime (eco). Puede terminarse mediante Ctrl-Inter
9	<i>Escribir cadena a salida estándar.</i> Recibe: DS:DX = dirección de la cadena
0A	<i>Entrada de teclado con búfer.</i> Lee una cadena de caracteres del dispositivo de entrada estándar. Recibe: DS:DX apunta una estructura predefinida del teclado
0B	<i>Comprobar estado de la entrada estándar.</i> Verifica si hay un carácter de entrada en espera. Devuelve: AL = 0FFh si el carácter está listo; en caso contrario, AL = 0
0C	<i>Borrar búfer del teclado e invocar a la función de entrada.</i> Borra el búfer de entrada de la consola y después ejecuta una función de entrada. Recibe: AL = función deseada (1, 6, 7, 8 o 0Ah)
0E	<i>Seleccionar unidad predeterminada.</i> Recibe: DL = número de unidad (0 = A, 1 = B, etcétera)
0F-18	Funciones de archivo FCB (obsoleta)
19	<i>Obtener unidad predeterminada actual.</i> Devuelve: AL = número de unidad (0 = A, 1 = B, etcétera)
1A	<i>Establecer dirección de transferencia de disco.</i> Recibe: DS:DX contiene la dirección del área de transferencia del disco
25	<i>Establecer vector de interrupción.</i> Establece una entrada en la Tabla de vectores de interrupción a una nueva dirección. Recibe: DS:DX apunta a la rutina de manejo de interrupciones que se inserta en la tabla; AL = el número de interrupción
26	<i>Crear nuevo prefijo de segmento del programa.</i> Recibe: DX = dirección de segmento para el nuevo PSP
27-29	Funciones de archivo FCB (obsoleta)
2A	<i>Obtener fecha del sistema.</i> Devuelve: AL = Día de la semana (0-6, en donde Domingo = 0), CX = año, DH = mes, y DL = día
2B	<i>Establecer fecha del sistema.</i> Recibe: CX = año, DH = mes, y DL = día. Devuelve: AL = 0 si la fecha es válida
2C	<i>Obtener hora del sistema.</i> Devuelve: CH = hora, CL = minutos, DH = segundos, y DL = centésimas de segundos
2D	<i>Establecer hora del sistema.</i> Recibe: CH = hora, CL = minutos, DH = segundos, y DL = centésimas de segundos. Devuelve: AL = 0 si la hora es válida
2E	<i>Establecer bandera de verificación.</i> Recibe: AL = nuevo estado de la bandera de Verificación de MS-DOS (0 = apagada, 1 = encendida), DL = 00h

(Continúa)

Tabla C-2 (Continuación)

Operación	Descripción
2F	<i>Obtener dirección de transferencia de disco</i> (DTA). Devuelve: ES:BX = dirección
30	<i>Obtener número de versión de MS-DOS</i> . Devuelve: AL = número mayor de versión, AH = número menor de versión, BH = número de serie OEM, BL:CX = número de serie de usuario de 24 bits
31	<i>Terminar y permanecer residente</i> . Termina el programa o proceso actual, dejando parte de sí mismo en la memoria. Recibe: AL = código de retorno y DX = número solicitado de párrafos
32	<i>Obtener bloque de parámetros de unidad de MS-DOS</i> . Recibe: DL = número de unidad. Devuelve: AL = estado; DS:BX apunta al bloque de parámetros de la unidad
33	<i>Comprobación de interrupción extendida</i> . Indica si MS-DOS está comprobando o no que se oprime Ctrl-Inter
34	<i>Obtener dirección de bandera INDOS</i> (Sin documentar)
35	<i>Obtener vector de interrupción</i> . Recibe: AL = número de interrupción. Devuelve: ES:BX = segmento/desplazamiento del manejador de interrupciones
36	<i>Obtener espacio libre en disco</i> . (sólo FAT16). Recibe: DL = número de unidad (0 = predeterminada, 1 = A, etcétera). Devuelve: AX = sectores por clúster, o FFFFh si el número de unidad es inválido; BX = número de clústeres disponibles, CX = bytes por sector y DX = clústeres por unidad
37	<i>Obtener carácter de conmutación</i> (Sin documentar)
38	Obtener o establecer información regional ^a
39	<i>Crear subdirectorio</i> . Recibe: DS:DX apunta a una cadena ASCIIZ con la ruta y el nombre de directorio. Devuelve: AX = código de error si se activa la bandera Acarreo
3A	<i>Eliminar subdirectorio</i> . Recibe: DS:DX apunta a una cadena ASCIIZ con la ruta y el nombre de directorio. Devuelve: AX = código de error si se activa la bandera Acarreo
3B	<i>Cambiar directorio actual</i> . Recibe: DS:DX apunta a una cadena ASCIIZ con la nueva ruta del directorio. Devuelve: AX = código de error si se activa la bandera Acarreo
3C	<i>Crear o truncar archivo</i> . Crea un nuevo archivo o trunca un archivo anterior a cero bytes. Abre el archivo en modo de salida. Recibe: DS:DX apunta a una cadena ASCIIZ con el nombre del archivo, y CX = atributo del archivo. Devuelve: AX = código de error si se activa la bandera Acarreo; en caso contrario AX = el manejador del nuevo archivo
3D	<i>Abrir archivo existente</i> . Abre un archivo en modo de entrada, salida o entrada-salida. Recibe: DS:DX apunta a una cadena ASCIIZ con el nombre del archivo y AL = el código de acceso (0 = leer, 1 = escribir, 2 = leer/escribir). Devuelve: AX = código de error si se activa la bandera Acarreo, en caso contrario AX = el manejador del nuevo archivo
3E	<i>Cerrar manejador de archivo</i> . Cierra el archivo o dispositivo especificado por un manejador de archivo. Recibe: BX = manejador de archivo de la operación abrir o crear previa. Devuelve: Si la bandera Acarreo se activa, AX = código de error
3F	<i>Leer de archivo o dispositivo</i> . Lee un número especificado de bytes de un archivo o dispositivo. Recibe: BX = manejador del archivo, DS:DX apunta a un búfer de entrada y CX = número de bytes a leer. Devuelve: Si se activa la bandera Acarreo, AX = código de error; en caso contrario, AX = número de bytes leídos
40	<i>Escribir en archivo o dispositivo</i> . Escribe un número específico de bytes a un archivo o dispositivo. Recibe: BX = manejador del archivo, DS:DX apunta a un búfer de salida y CX = el número de bytes a escribir. Devuelve: si se activa la bandera Acarreo, AX = código de error; en caso contrario, AX = número de bytes escritos
41	<i>Eliminar archivo</i> . Elimina un archivo de un directorio especificado. Recibe: DS:DX apunta a una cadena ASCIIZ con el nombre del archivo. Devuelve: AX = código de error si se activa la bandera Acarreo

Operación	Descripción
42	<i>Mover apuntador de archivo.</i> Mueve el apuntador de lectura/escritura de un archivo, de acuerdo con el método especificado. Recibe: CX:DX = distancia (bytes) que se va a mover el apuntador del archivo, AL = código del método, BX = manejador del archivo. Los códigos de los métodos son: 0 = mover desde el inicio del archivo, 1 = mover hasta la ubicación actual más un desplazamiento, y 2 = mover hasta el final del archivo más un desplazamiento. Devuelve: AX = código de error si se activa la bandera Acarreo
43	<i>Obtener/Establecer atributo de archivo.</i> Obtiene o establece el atributo de un archivo. Recibe: DS:DX = apuntador a una ruta y nombre de archivo ASCIIZ, CX = atributo y AL = código de función (1 = establecer atributo, 0 = obtener atributo). Devuelve: AX = código de error si se activa la bandera Acarreo
44	<i>Control de E/S para dispositivos.</i> Obtiene o establece la información de un dispositivo asociado con un manejador de dispositivo abierto, o envía una cadena de control al manejador del dispositivo, o recibe una cadena de control desde el manejador del dispositivo
45	<i>Duplicar manejador de archivo.</i> Devuelve un nuevo manejador de archivo para un archivo que ya se encuentre abierto. Recibe: BX = manejador del archivo. Devuelve: AX = código de error si se activa la bandera Acarreo
46	<i>Forzar duplicado de manejador de archivo.</i> Obliga a que el manejador en CX se refiera al mismo archivo, en la misma posición que el manejador en BX. Recibe: BX = manejador de archivo existente y CX = segundo manejador de archivo. Devuelve: AX = código de error si se activa la bandera Acarreo
47	<i>Obtener directorio actual.</i> Obtiene el nombre de ruta completa del directorio actual. Recibe: DS:SI apunta a un área de 64 bytes para guardar la ruta de directorio, y DL = número de unidad. Devuelve: Un búfer en DS:SI se llena con la ruta, y AX = código de error si se activa la bandera Acarreo
48	<i>Asignar memoria.</i> Asigna un número solicitado de párrafos de memoria, medidos en bloques de 16 bytes. Recibe: BX = número de párrafos solicitados. Devuelve: AX = segmento del bloque asignado y BX = tamaño del mayor bloque disponible (en párrafos), y AX = código de error si se activa la bandera Acarreo
49	<i>Liberar memoria asignada.</i> Libera la memoria que asignó previamente la función 48h. Recibe: ES = segmento del bloque a liberar. Devuelve: AX = código de error si se activa la bandera Acarreo
4A	<i>Modificar bloques de memoria.</i> Modifica los bloques de memoria asignados para contener un nuevo tamaño de bloque. El bloque se reducirá o crecerá. Recibe: ES = segmento del bloque y BX = número solicitado de párrafos. Devuelve: AX = código de error si se activa la bandera Acarreo y BX = número máximo de bloques disponibles
4B	<i>Cargar o ejecutar programa.</i> Crea un prefijo de segmento de programa para otro programa, lo carga en la memoria y lo ejecuta. Recibe: DS:DX apunta a una cadena ASCIIZ con la unidad, ruta y nombre de archivo del programa; ES:BX apunta a un bloque de parámetro y AL = valor de la función. Valores de función en AL: 0 = cargar y ejecutar el programa; 3 = cargar pero no ejecutar (superponer el programa). Devuelve: AX = código de error si se activa la bandera Acarreo
4C	<i>Terminar proceso.</i> La manera usual de terminar un programa y regresar a MS-DOS o al programa que hizo la llamada. Recibe: AL = código de retorno de 8 bits, que puede consultarse mediante la función 4Dh de DOS, o mediante el comando ERRORLEVEL en un archivo de procesamiento por lotes
4D	<i>Obtener código de retorno del proceso.</i> Obtiene el código de retorno de un proceso o programa, generado por la llamada a la función 31h o por la llamada a la función 4Ch. Devuelve: AL = código de 8 bits devuelto por el programa, AH = tipo de salida generada: 0 = terminación normal, 1 = terminación mediante CTRLINTER, 2 = terminación mediante un error crítico de dispositivo, y 3 = terminación mediante una llamada a la función 31h
4E	<i>Buscar primer archivo que coincida.</i> Busca el primer nombre de archivo que coincida con una especificación de archivo dada. Recibe: DS:DX apunta a una unidad, ruta y especificación de archivo ASCIIZ; CX = Atributo de archivo a usar en la búsqueda. Devuelve: AX = código de error si se activa la bandera Acarreo; en caso contrario, se llena la DTA actual con el nombre de archivo, atributo, fecha y tamaño. Por lo general, se hace una llamada a la función 1Ah de DOS (establecer DTA) antes de esta función

(Continúa)

Tabla C-2 (Continuación)

Operación	Descripción
4F	<i>Buscar siguiente archivo que coincida.</i> Busca el siguiente nombre de archivo que coincide con una especificación de archivo dada. Esta función siempre se llama después de la función 4Eh de DOS. Devuelve: AX = código de error si se activa la bandera Acarreo; en caso contrario, la DTA actual se llena con la información del archivo
54	<i>Obtener bandera Verificación.</i> Devuelve: AH = Bandera Verificación para la E/S de disco (0 = desactivada; 1 = activada)
56	<i>Cambiar de nombre/mover archivo.</i> Cambia el nombre a un archivo, o lo mueve hacia otro directorio. Recibe: DS:DX apunta a una cadena ASCIIZ que especifica la unidad, ruta y nombre de archivo actual; ES:DI apunta a la nueva ruta y nombre de archivo. Devuelve: AX = código de error si se activa la bandera Acarreo
57	<i>Obtener/Establecer fecha/hora de archivo.</i> Obtiene o establece la etiqueta de fecha y hora para un archivo. Recibe: AL = 0 para obtener la fecha/hora, o AL = 1 para establecer la fecha/hora; BX = manejador del archivo, CX = nueva hora del archivo, y DX = nueva fecha del archivo. Devuelve: AX = código de error si se activa la bandera Acarreo; en caso contrario, CX = hora actual del archivo y DX = fecha actual del archivo
58	<i>Obtener o establecer estrategia de asignación de memoria^a</i>
59	<i>Obtener información de error extendida.</i> Devuelve información adicional acerca de un error de MS-DOS, incluyendo la clase de error, el lugar, y la acción recomendada. Recibe: BX = número de versión de MS-DOS (cero para la versión 3.xx). Devuelve: AX = código de error extendido, BH = clase de error, BL = acción sugerida y CH = lugar
5A	<i>Crear archivo temporal.</i> Genera un nombre de archivo único en un directorio especificado. Recibe: DS:DX apunta a un nombre de ruta ASCIIZ que termina con una barra diagonal inversa (\); CX = atributo de archivo deseado. Devuelve: AX = código de error si se activa la bandera Acarreo; en caso contrario, DS:DX apunta a la ruta con el nuevo nombre de archivo que se le adjuntó
5B	<i>Crear nuevo archivo.</i> Trata de crear un nuevo archivo, pero falla si el nombre de archivo ya existe. Esto nos evita tener que sobrescribir un archivo existente. Recibe: DS:DX apunta a una cadena ASCIIZ con la ruta y el nombre de archivo. Devuelve: AX = código de error si se activa la bandera Acarreo
5C-61	Se omitió
62	<i>Obtener dirección del segmento de prefijo del programa (PSP).</i> Devuelve: BX = el valor de segmento del PSP del programa actual
7303h	<i>Obtener espacio libre en disco.</i> Llena una estructura que contiene información detallada sobre el espacio en disco. Recibe: AX = 1 7303h, ES:DI apunta a una estructura ExtGetDskFreSpcStruc, CX = tamaño de la estructura ExtGetDskFreSpcStruc, DS:DX apunta a una cadena con terminación nula que contiene el nombre de la unidad. Devuelve: La estructura ExtGetDskFreSpcStruc se llena con información del disco. Vea los detalles en la sección 14.5.1
7305h	<i>Lectura y escritura en disco absoluta.</i> Lee sectores individuales o grupos de sectores del disco. No funciona en Windows NT, 2000 y XP. Recibe: AX = 7305h, DS:BX = segmento/desplazamiento de una variable de la estructura DISKIO, CX = 0FFFFh, DL = número de unidad (0 = predeterminada, 1 = A, 2 = B, 3 = C, etcétera), SI = bandera de lectura/escritura. Vea los detalles en la sección 14.4

^a Para obtener detalles, vea: Ray Duncan, Advanced MS-DOS, 2^a edición, Microsoft Press, 1998; *Ralf Brown's Interrupt List*, disponible en Web.

C.4 Funciones de la interrupción 10H (BIOS de video)

Tabla C-3 Funciones de la interrupción 10h (BIOS de video).

Operación	Descripción
0	<i>Establecer modo de video.</i> Establece la pantalla de video a monocromo, texto, gráficos o modo de color. Recibe: AL = modo de pantalla
1	<i>Establecer líneas del cursor.</i> Identifica las líneas de exploración inicial y final para el cursor. Recibe: CH = línea inicial del cursor y CL = línea final del cursor
2	<i>Establecer posición del cursor.</i> Posiciona el cursor en la pantalla. Recibe: BH = página de video, DH = fila y DL = columna
3	<i>Establecer posición del cursor.</i> Obtiene la posición y el tamaño en pantalla del cursor. Recibe: BH = página de video. Devuelve: CH = línea inicial del cursor, CL = línea final del cursor, DH = fila del cursor y DL = columna del cursor
4	<i>Leer lápiz luminoso.</i> Lee la posición y el estado del lápiz luminoso. Devuelve: CH = fila de píxel, BX = columna de píxel, DH = fila de caracteres y DL = columna de caracteres
5	<i>Establecer página de visualización.</i> Establece la página de video a visualizar. Recibe: AL = número de página deseado
6	<i>Desplazar ventana hacia arriba.</i> Desplaza una ventana en la página de video actual hacia arriba, sustituyendo las líneas desplazadas con espacios en blanco. Recibe: AL = número de líneas a desplazar, BH = atributo para las líneas desplazadas, CX = fila y columna de la esquina superior izquierda, y DX = fila y columna inferior derecha
7	<i>Desplazar ventana hacia abajo.</i> Desplaza una ventana en la página de video actual hacia abajo, sustituyendo las líneas desplazadas con espacios en blanco. Recibe: AL = número de líneas a desplazar, BH = atributo para las líneas desplazadas, CX = fila y columna de la esquina superior izquierda, y DX = fila y columna inferior derecha
8	<i>Leer carácter y atributo.</i> Lee el carácter y su atributo en la posición actual del cursor. Recibe: BH = página de pantalla. Devuelve: AH = byte de atributo, y AL = código de carácter ASCII
9	<i>Escribir carácter y atributo.</i> Escribe un carácter y su atributo en la posición actual del cursor. Recibe: AL = carácter ASCII, BH = página de video, y CX = factor de repetición
0A	<i>Escribir carácter.</i> Escribe sólo un carácter (sin atributo) en la posición actual del cursor. Recibe: AL = carácter ASCII, BH = página de video, BL = atributo, y CX = factor de repetición
0B	<i>Establecer paleta de colores.</i> Selecciona un grupo de colores disponibles para el adaptador de color o EGA. Recibe: AL = modo de pantalla y BH = página activa de pantalla
0C	<i>Escribir píxel de gráficos.</i> Escribe un píxel de gráficos cuando se encuentra en modo de gráficos. Recibe: AL = valor de píxel, CX = coordenada X, y DX = coordenada Y
0D	<i>Leer píxel de gráficos.</i> Lee el color de un solo píxel de gráficos en una ubicación dada. Recibe: CX = coordenada X y DX = coordenada Y
0E	<i>Escribir carácter.</i> Escribe un carácter en la pantalla y avanza el cursor. Recibe: AL = código de carácter ASCII, BH = página de video, y BL = atributo o color
0F	<i>Obtener modo de video actual.</i> Obtiene el modo de video actual. Devuelve: AL = modo de video y BH = página de video activa
10	<i>Establecer paleta de video.</i> (EGA solamente) Establece el registro de la paleta de video, color del borde o bit de intensidad/destello. Recibe: AL = código de función (00 = establecer registro de paleta, 01 = establecer color del borde, 02 = establecer paleta y color del borde, 03 = establecer/restablecer bit de destello/intensidad), BH = color, y BL = registro de paleta a establecer. Si AL = 2, ES:DX apunta a una lista de colores
11	<i>Generador de caracteres.</i> Selecciona el tamaño de carácter para la pantalla EGA. Por ejemplo, se utiliza una fuente de 8 por 8 para la pantalla de 43 líneas, y se utiliza una fuente de 8 por 14 para la pantalla de 25 líneas
12	<i>Función de selección alternativa.</i> Devuelve información técnica acerca de la pantalla EGA
13	<i>Escribir cadena.</i> (PC/AT solamente) Escribe una cadena de texto en la pantalla de video. Recibe: AL = modo, BH = página, BL = atributo, CX = longitud de cadena, DH = fila, DL = columna, y ES:BP apunta a la cadena (no funciona en la IBM-PC o PC/XT)

C.5 Funciones INT 16h del BIOS de teclado

Tabla C-4 Funciones de la interrupción 16h del BIOS del teclado.

Operación	Descripción
03h	<i>Establecer velocidad de repetición.</i> Recibe: AH = 03h, AL = 5, BH = retraso de repetición, y BL = velocidad de repetición. Los valores de retraso en BH son 0 = 250 ms; 1 = 500 ms; 2 = 750 ms; 3 = 1000 ms. La velocidad de repetición en BL varía de 0 (más veloz) a 1Fh (más lenta). Devuelve: nada
05h	<i>Meter tecla en búfer.</i> Mete un carácter del teclado y su correspondiente código de exploración en el búfer de escritura adelantada del teclado. Recibe: AH = 05h, CH = código de exploración, y CL = código de carácter. Si el búfer de escritura adelantada ya está lleno, se activará la bandera Acarreo y AL = 1. Devuelve: nada
10	<i>Esperar tecla.</i> Espera un carácter de entrada y un código de exploración del teclado. Recibe: AH = 10h. Devuelve: AH = código de exploración. AL = carácter ASCII. (La función 00h duplica esta función, usando un tipo anterior de teclado)
11	<i>Comprobar búfer del teclado.</i> Averigua si hay un carácter esperando en el búfer de escritura adelantada del teclado. Recibe: AH = 01h. Devuelve: SI hay una tecla esperando, el código de exploración se devuelve en AH y su código ASCII se devuelve en AL, y la bandera Cero se borra (el carácter permanecerá en el búfer de entrada). Si no hay tecla esperando, se activa la bandera Cero. (La función 01h duplica esta función, usando un tipo anterior de teclado)
12	<i>Obtener banderas del teclado.</i> Devuelve el byte de la bandera del teclado que está almacenado en la RAM inferior. Recibe: AH = 12h. Devuelve: las banderas del teclado en AX. (La función 02h duplica esta función, usando un tipo anterior de teclado)

C.6 Funciones del ratón (INT 33h)

Las funciones INT 33h del ratón reciben su número de función en el registro AX. Si desea obtener más información acerca de estas funciones, vea la sección 15.6. Para las funciones adicionales del ratón, vea la tabla 15-9.

Tabla C-5 Funciones INT 33h del ratón.

Operación	Descripción
0000h	<i>Restablecer ratón y obtener estado.</i> Recibe: AX = 0000h. Restablece el ratón y confirma que está disponible. El ratón (si se encuentra) se centra en la pantalla, su página de visualización se establece a la página de video 0, su apuntador se oculta y sus proporciones de mickeys a píxeles y velocidad se establecen a los valores predeterminados. El rango de movimiento del ratón se establece a toda el área de la pantalla
0001h	<i>Mostrar puntero del ratón.</i> Recibe: AX = 0001h. Devuelve: nada. El controlador del ratón mantiene una cuenta del número de veces que se llama a esta función
0002h	<i>Ocultar puntero del ratón.</i> Recibe: AX = 0002h. Devuelve: nada. La posición del ratón se sigue rastreando cuando está invisible
0003h	<i>Obtener posición y estado del ratón.</i> Recibe: AX = 0003h. Devuelve: BX = estado del botón del ratón, CX = coordenada X (en píxeles), DX = coordenada Y (en píxeles)
0004h	<i>Establecer posición del ratón.</i> Recibe: AX = 0004h, CX = coordenada X (en píxeles), DX = coordenada Y (en píxeles). Devuelve: nada
0005h	<i>Obtener información de botones oprimidos del ratón.</i> Recibe: AX = 0005h, BX = ID del botón (0 = izquierdo, 1 = derecho, 2 = central). Devuelve: AX = estado del botón, BX = contador de botones oprimidos del ratón, CX = coordenada X del último botón oprimido, DX = coordenada Y del último botón oprimido
0006h	<i>Obtener información de botones liberados del ratón.</i> Recibe: AX = 0006h, BX = ID del botón (0 = izquierdo, 1 = derecho, 2 = central). Devuelve: AX = estado del botón, BX = contador de botones liberados del ratón, CX = coordenada X del último botón liberado, DX = coordenada Y del último botón liberado
0007h	<i>Establecer límites horizontales.</i> Recibe: AX = 0007h, CX = coordenada X mínima (en píxeles), DX = coordenada X máxima (en píxeles). Devuelve: nada
0008h	<i>Establecer límites verticales.</i> Recibe: AX = 0008h, CX = coordenada Y mínima (en píxeles), DX = coordenada Y máxima (en píxeles). Devuelve: nada

D

RESPUESTAS A LAS PREGUNTAS DE REPASO

D.1 Conceptos básicos

D.1.1 Bienvenido al lenguaje ensamblador

1. Un ensamblador convierte los programas en código fuente de lenguaje ensamblador a lenguaje máquina. Un enlazador combina los archivos individuales creados por un ensamblador en un solo programa ejecutable.
2. El lenguaje ensamblador es una herramienta útil para saber de qué manera los programas de aplicaciones se comunican con el sistema operativo de la computadora a través de los manejadores de interrupciones, llamadas al sistema, y áreas de memoria comunes. También nos ayuda a comprender de qué manera el sistema operativo carga y ejecuta los programas de aplicaciones.
3. En una relación de *uno a varios*, una sola instrucción se expande en varias instrucciones de lenguaje ensamblador o de lenguaje máquina.
4. Se dice que un lenguaje cuyos programas de código fuente pueden compilarse y ejecutarse en una amplia variedad de sistemas computacionales es *portable*.
5. No. Cada lenguaje ensamblador se basa en una familia de procesadores o en una computadora específica.
6. Algunos ejemplos de aplicaciones con sistemas incrustados son los sistemas de combustible y encendido de los automóviles, los sistemas de control de aire acondicionado, los sistemas de seguridad, los sistemas de control de vuelo, las computadoras portátiles, los módems, las impresoras y demás periféricos de computadora inteligentes.
7. Los *controladores de dispositivos* son programas que traducen los comandos generales del sistema operativo en referencias específicas a los detalles de hardware que sólo el fabricante conoce.
8. C++ no permite asignar un apuntador de cierto tipo a un apuntador de otro tipo. El lenguaje ensamblador no tiene dicha restricción con los apuntadores.
9. Aplicaciones adecuadas para el lenguaje ensamblador: controlador de dispositivos de hardware y sistemas incrustados, y los juegos de computadora que requieren un acceso directo al hardware.
10. En lenguaje de alto nivel tal vez no proporcione un acceso directo al hardware. Aun cuando lo haga, por lo general, se deben usar técnicas de codificación difíciles, lo cual puede ocasionar problemas de mantenimiento.
11. El lenguaje ensamblador tiene una estructura formal mínima, por lo que los programadores con varios niveles de experiencia deben imponer una estructura. Esto conlleva a dificultades para mantener el código existente.
12. El código para la expresión $X = (Y * 4) + 3$ es:

```
mov eax,Y          ; mueve Y a EAX  
mov ebx,4          ; mueve 4 a EBX
```

```

imul ebx ; EAX = EAX * EBX
add eax,3 ; suma 3 a EAX
mov X,eax ; mueve EAX a X

```

D.1.2 Concepto de máquina virtual

1. Las computadoras se construyen en niveles, de manera que cada nivel representa un nivel de traducción, de un conjunto de instrucciones de mayor nivel a un conjunto de instrucciones de menor nivel.
2. Porque es demasiado detallado y consiste sólo de números. Es difícil de entender para los humanos.
3. Verdadero.
4. Un programa N1 completo se convierte en un programa N0 mediante un programa N0 diseñado en específico para este propósito. Después, el programa N0 resultante se ejecuta directamente en el hardware de la computadora.
5. El modo de operación virtual86 del IA-32 emula la arquitectura del procesador Intel 8086/8088 utilizado en la IBM-PC original.
6. El código byte de Java es un lenguaje de bajo nivel que se ejecuta con rapidez en tiempo de ejecución, mediante un programa conocido como máquina virtual de Java (JVM).
7. Lógica digital, microarquitectura, arquitectura del conjunto de instrucciones, sistema operativo, lenguaje ensamblador, lenguaje de alto nivel.
8. Los comandos específicos de la microarquitectura son a menudo un secreto propietario. Además, la programación en microcódigo no es práctica, ya que con frecuencia se requieren tres o cuatro microinstrucciones para llevar a cabo una sola operación primitiva.
9. Arquitectura del conjunto de instrucciones.
10. Niveles 2 y 3.

D.1.3 Representación de datos

1. El bit menos significativo (bit 0).
2. El bit más significativo (el bit de mayor numeración).
3. (a) 248 (b) 202 (c) 240
4. (a) 53 (b) 150 (c) 204
5. (a) 00010001 (b) 101000000 (c) 00011110
6. (a) 110001010 (b) 110010110 (c) 100100001
7. (a) 2 (b) 4 (c) 8
8. (a) 16 (b) 32 (c) 64
9. (a) 7 (b) 9 (c) 16
10. (a) 12 (b) 16 (c) 22
11. (a) CF57 (b) 5CAD (c) 93EB
12. (a) 35DA (b) CEA3 (c) FEDB
13. (a) 1110 0101 1011 0110 1010 1110 1101 0111
 (b) 1011 0110 1001 0111 1100 0111 1010 0001
 (c) 0010 0011 0100 1011 0110 1101 1001 0010
14. (a) 0000 0001 0010 0110 1111 1001 1101 0100
 (b) 0110 1010 1100 1101 1111 1010 1001 0101
 (c) 1111 0110 1001 1011 1101 1100 0010 1010
15. (a) 58 (b) 447 (c) 16534

16. (a) 98 (b) 457 (c) 27227
17. (a) FFE6 (b) FE3C
18. (a) FFE0 (b) FFC2
19. (a) +31915 (b) -16093
20. (a) +32667 (b) -32208
21. (a) -75 (b) +42 (c) -16
22. (a) -128 (b) -52 (c) -73
23. (a) 11111011 (b) 11011100 (c) 11110000
24. (a) 10111000 (b) 10011110 (c) 11100110
25. 58h y 88d.
26. 4Dh y 77d.
27. Para manejar conjuntos de caracteres internacionales que requieren más de 256 códigos.
28. $2^{256} - 1$
29. $+2^{255} - 1$

D.1.4 Operaciones booleanas

1. (NOT X) OR Y.

2. X AND Y.

3. V.

4. F.

5. V.

6. Tabla de verdad:

A	B	$A \vee B$	$\neg(A \vee B)$
F	F	F	V
F	V	V	F
V	F	V	F
V	V	V	F

7. Tabla de verdad:

A	B	$\neg A$	$\neg B$	$\neg A \wedge \neg B$
F	F	V	V	V
F	V	V	F	F
V	F	F	V	F
V	V	F	F	F

8. 16, o (2^4).

9. 2 bits, produciendo los siguientes valores: 00, 01, 10, 11.

D.2 Arquitectura del procesador IA-32

D.2.1 Conceptos generales

1. Unidad de control, Unidad aritmética-lógica y reloj.
2. Buses de datos, de dirección y de control.
3. La memoria convencional se encuentra fuera de la CPU y responde con más lentitud a las solicitudes de acceso. Los registros están fijos dentro de la CPU.
4. Obtener, decodificar, ejecutar.
5. Obtener operandos de memoria, almacenar operandos de memoria.
6. Durante el paso de obtención.
7. Ejecución de las etapas del procesador en paralelo, lo cual hace posible la ejecución traslapada de las instrucciones de máquina.
8. 10 ciclos de reloj.
9. 12 ciclos ($5 + (8 - 1)$).
10. Un procesador superescalar contiene dos o más canalizaciones de ejecución.
11. 15 ciclos de reloj ($5 + 10$).
12. La sección 2.1.4 menciona el nombre de archivo, tamaño y ubicación inicial en el disco. La mayoría de los directorios también almacenan la fecha y hora de la última modificación del archivo.
13. El OS ejecuta una bifurcación (como un GOTO) a la primera instrucción de máquina en el programa.
14. La CPU ejecuta varias tareas (programas) al cambiar rápidamente de un programa al siguiente. Esto da la impresión de que todos los programas se ejecutan al mismo tiempo.
15. El programador del OS determina cuánto tiempo debe asignar a cada tarea, y cambia de una tarea a otra.
16. El contador del programa, las variables de la tarea, y los registros de la CPU (incluyendo las banderas de estado).
17. 3.33×10^{-10} , que es $1.0/3.0 \times 10^9$.

D.2.2 Arquitectura del procesador IA-32

1. Modo de direccionamiento real, modo protegido y modo de administración del sistema.
2. EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP.
3. CS, DS, SS, ES, FS, GS.
4. Contador de ciclo.
5. EBP.
6. Las más comunes: Acarreo, Signo, Cero, Desbordamiento. Las menos comunes: Acarreo auxiliar, Paridad.
7. Acarreo.
8. Desbordamiento.
9. Signo.
10. Unidad de punto flotante.
11. 80 bits.
12. El Intel 80386.
13. El Pentium.
14. El Pentium.
15. CISC significa *conjunto complejo de instrucciones*: una extensa colección de instrucciones, algunas de las cuales realizan operaciones sofisticadas que podrían ser características de un lenguaje de alto nivel.

16. El término RISC significa *conjunto reducido de instrucciones*: un pequeño conjunto de instrucciones simples (atómicas) que pueden combinarse en operaciones más complejas.

D.2.3 Administración de memoria del procesador IA-32

1. 4GB (0 a FFFFFFFFh).
2. 1MB (0 a FFFFh).
3. Lineal (absoluto).
4. 09600h.
5. 0CFF0h.
6. 32 bits.
7. El registro SS.
8. Tabla de descriptores locales.
9. Tabla de descriptores globales.
10. El tamaño total de todos los programas cargados en la memoria puede exceder la cantidad de memoria física instalada en la computadora.
11. Desde luego que ésta es una pregunta abierta. Es un hecho que MS-DOS tuvo que ejecutarse primero en los procesadores 8086/8088, que sólo soportaban el modo de direccionamiento real. Cuando surgieron procesadores más recientes que soportaban el modo Protegido, quizás Microsoft deseaba que MS-DOS siguiera ejecutándose en los procesadores anteriores. De no ser así, los clientes con computadoras antiguas no se actualizarían a las nuevas versiones de MS-DOS.
12. Las siguientes direcciones tipo segmento-desplazamiento apuntan a la misma dirección lineal: 0640:0100, y 0630:0200.

D.2.4 Componentes de una microcomputadora IA-32

1. SRAM significa RAM estática; se utiliza en la memoria caché de la CPU.
2. Pentium.
3. El 8259 es un chip controlador de interrupciones, que en ocasiones se le llama PIC; programa las interrupciones de hardware e interrumpe a la CPU.
4. En la tarjeta de video o en la tarjeta madre (área especial de memoria).
5. Un rayo de electrones ilumina los puntos de fósforo en la pantalla, llamados píxeles. Empezando desde la parte superior de la pantalla, el cañón dispara electrones de izquierda a derecha en una fila horizontal, se apaga brevemente y regresa al lado izquierdo de la pantalla para empezar una nueva fila. El retrazado horizontal se refiere al periodo durante el cual el cañón está apagado, al pasar de una fila a otra. Cuando se dibuja la última fila, el cañón se apaga (retrazado vertical) y se desplaza hasta la esquina superior izquierda de la pantalla para empezar de nuevo.
6. RAM dinámica, RAM estática, RAM de video, y RAM de CMOS.
7. RAM estática.
8. La computadora puede consultar a un dispositivo conectado a través del USB para averiguar su nombre y tipo de dispositivo, y el tipo de controlador que soporta. La computadora también puede suspender la energía para cada dispositivo. Ninguna de estas capacidades es posible con los puertos seriales y paralelos.
9. Upstream y Downstream.
10. UART 16550 (transmisor receptor asíncrono universal).

D.2.5 Sistema de entrada-salida

1. El nivel del programa de aplicación.

2. Las funciones del BIOS se comunican directamente con el hardware del sistema. Son independientes del sistema operativo.
3. Se están inventando nuevos dispositivos todo el tiempo, con capacidades que, por lo general, no se anticiparon al momento de escribir el BIOS.
4. El nivel del BIOS.
5. Los niveles del sistema operativo, BIOS y hardware.
6. A menudo, los programas de juegos tratan de aprovechar las características más recientes en las tarjetas de sonido especializadas. Hay que destacar que las aplicaciones de juegos en MS-DOS eran más propensas a hacer esto que los juegos que se ejecutan en MS Windows. De hecho, Windows NT, 2000 y XP evitan que las aplicaciones accedan directamente al hardware del sistema.
7. No. El mismo BIOS funcionaría para ambos sistemas operativos. Muchos propietarios de computadoras instalan dos o tres sistemas operativos en la misma computadora. En definitiva ¡no querrían cambiar el BIOS del sistema cada vez que reiniciaran la computadora!

D.3 Fundamentos del lenguaje ensamblador

D.3.1 Elementos básicos del lenguaje ensamblador

1. h, q, o, d, b, r, t, y.
2. No (se requiere un cero a la izquierda).
3. No (tienen la misma precedencia).
4. Expresión: 10 MOD 3.
5. Constante de número real: +3.5E-02.
6. No, también pueden encerrarse entre comillas dobles.
7. Directivas.
8. 247 caracteres.
9. Verdadero.
10. Verdadero.
11. Falso.
12. Verdadero.
13. Etiqueta, nemónico, operando(s), comentario.
14. Verdadero.
15. Verdadero.
16. Ejemplo de código:

```
Comment !
Este es un comentario
Este tambien es un comentario
!
```

17. Debido a que las direcciones codificadas en las instrucciones se tendrían que actualizar cada vez que se insertaran nuevas variables antes de las ya existentes.

D.3.2 Ejemplo: suma y resta de enteros

1. La directiva INCLUDE copia las definiciones necesarias y la información de configuración del archivo de texto *Irvine32.inc*. Los datos de este archivo se insertan en el flujo de datos leído por el ensamblador.
2. La directiva .CODE marca el inicio del segmento de código.

3. código, datos y pila.
4. Llamando al procedimiento **DumpRegs**.
5. La instrucción **exit**.
6. La directiva **PROC**.
7. La directiva **ENDP**.
8. Marca la última línea del programa a ensamblar, y la etiqueta enseguida de END identifica el punto de entrada del programa (en donde empieza la ejecución).
9. PROTO declara el nombre de un procedimiento al que el programa actual llama.

D.3.3 Ensamblado, enlazado y ejecución de programas

1. Archivos de código objeto (.OBJ) y de listado (.LST).
2. Verdadero.
3. Verdadero.
4. Cargador.
5. Ejecutable (.EXE) y mapa (.MAP).
6. La opción /FI.
7. La opción /Zi.
8. Indica al enlazador que debe producir una aplicación de *Consola Win32*.
9. Hay demasiados como para mencionarlos aquí, pero puede ver sus nombres abriendo el archivo Kernel32.lib; para ello utilice el editor TextPad que se proporciona en el CD-ROM del libro. El archivo se mostrará en hexadecimal. Avance hasta el desplazamiento 1840h y busque los diversos nombres de funciones que se presentan desde ese punto en adelante.
10. /ENTRY establece la dirección inicial del programa (el punto de entrada). Por ejemplo, suponga que desea que su programa empiece a ejecutarse en el procedimiento Inicio. La línea de comandos del enlazador sería:

```
Link /ENTRY:Inicio
```

Ésta es una pregunta difícil, ya que no podemos encontrar la respuesta en el apéndice A. En vez de ello, puede leer acerca de las opciones de línea de comandos del enlazador de 32 bits, visitando el sitio Web de MSDN y buscando *referencia enlazador (linker)*.

D.3.4 Definición de datos

1. var1 SWORD ?
2. var2 BYTE ?
3. var3 SBYTE ?
4. var4 QWORD ?
5. SDWORD
6. var5 SDWORD -2147483648
7. wArreglo WORD 10, 20, 30
8. miCcolor BYTE "azul",0
9. dArreglo DWORD 50 DUP(?)
10. miCadenaPrueba BYTE 500 DUP("TEST")
11. bArreglo BYTE 20 DUP(0)
12. 21h, 43h, 65h, 87h.

D.3.5 Constantes simbólicas

1. RETROCESO = 08h
2. SegundosEnDia = 24 * 60 * 60
3. TamArreglo = (\$ - miArreglo)
4. TamArreglo = (\$ - miArreglo) / TYPE DWORD
5. PROCEDURE TEXTEQU <PROC>
6. Ejemplo de código:

```
Ejemplo TEXTEQU <"Esta es una cadena">
MiCadena BYTE Ejemplo
```
7. EstablecerESI TEXTEQU <mov esi, OFFSET miArreglo>

D.4 Transferencias de datos, direccionamiento y aritmética

D.4.1 Instrucciones de transferencia de datos

1. Registro, inmediato y memoria.
2. Falso.
3. Falso.
4. Verdadero.
5. Un operando de registro o memoria de 32 bits.
6. Un operando inmediato de 16 bits (constante).
7. (a) no es válida (b) válida (c) no es válida (d) no es válida (e) no es válida (g) válida (h) no es válida.
8. (a) FCh (b) 01h
9. (a) 1000h (b) 3000h (c) FFF0h (d) 4000h
10. (a) 00000001h (b) 00001000h (c) 00000002h (d) FFFFFFFFCh

D.4.2 Suma y resta

1. inc val2
2. sub eax, val3
3. Código:

```
mov ax, val4
sub val2, ax
```
4. CF = 0, SF = 1.
5. CF = 1, SF = 1.
6. Anote los siguientes valores de bandera:
 - (a) CF = 1, SF = 0, ZF = 1, OF = 0
 - (b) CF = 0, SF = 1, ZF = 0, OF = 1
 - (c) CF = 0, SF = 1, ZF = 0, OF = 0
7. Ejemplo de código:

```
mov ax, val2
neg ax
add ax, bx
sub ax, val4
```
8. No.

9. Sí.
 10. Sí (por ejemplo, mov al, -128... seguida de ... neg al).
 11. No.
 12. Establecer las banderas Acarreo y Desbordamiento al mismo tiempo:

```
mov al,80h  
add al,80h
```

13. Establecer la bandera Cero después de INC y DEC para indicar un desbordamiento sin signo:

```
mov al,OFFh  
inc al  
jz ocurrio_desbordamiento  
mov bl,1  
dec bl  
jz ocurrio_desbordamiento
```

14. Restar 3 de 4 (sin signo). El acarreo que sale del MSB se invierte y se coloca en la bandera Acarreo:

$$\begin{array}{r}
 1 \leftarrow \\
 \boxed{\begin{array}{ccccccccc} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}} \quad (4) \\
 + \boxed{\begin{array}{ccccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \end{array}} \quad (-3) \\
 \hline
 \boxed{\begin{array}{ccccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}} \quad (1)
 \end{array}$$

```
mov al,4  
sub al,3 ; CF = 0
```

D.4.3 Operadores y directivas relacionadas con los datos

1. Falso.
 2. Falso.
 3. Verdadero.
 4. Falso.
 5. Verdadero.
 6. Directiva de datos:

```
.data  
ALIGN 2  
misBytes BYTE 10h, 20h, 30h, 40h  
etc.
```

7. (a) 1 (b) 4 (c) 4 (d) 2 (e) 4 (f) 8 (g) 5
8. mov dx, WORD PTR misBytes
9. mov al, BYTE PTR misPalabras +1
10. mov eax, DWORD PTR misBytes

- #### 11. Directiva de datos:

```
misPalabrasD LABEL DWORD  
misPalabras WORD 3 DUP (?) ,2000h  
.data  
mov eax,misPalabrasD
```

12. Directiva de datos:

```
misBytesW LABEL WORD  
misBytes BYTE 10h,20h,30h,40h  
.code  
mov ax,misBytesW
```

D.4.4 Direcciónamiento indirecto

1. Falso.
2. Verdadero.
3. Falso.
4. Falso.
5. Verdadero (se requiere el operador PTR).
6. Verdadero.
7. (a) 10h (b) 40h (c) 003Bh (d) 3 (e) 3 (f) 2
8. (a) 2010h (b) 003B008Ah (c) 0 (d) 0 (e) 0044h

D.4.5 Instrucciones JMP y LOOP

1. Verdadero.
2. Falso.
3. 4,294,967,296 veces.
4. Falso.
5. Verdadero.
6. CX.
7. ECX.
8. Falso (-128 a +127 bytes a partir de la ubicación actual).
9. ¡Es un truco! El programa no se detiene, ya que la primera instrucción LOOP decrementa ECX a cero. La segunda instrucción LOOP decrementa ECX a FFFFFFFFh, haciendo que el ciclo exterior se repita.
10. Inserte la siguiente instrucción en la etiqueta L1: push ecx. Inserte también la siguiente instrucción antes de la segunda instrucción LOOP: pop ecx. (Una vez que haya sumado estas instrucciones, el valor final de eax es 1Ch).

D.5 Procedimientos

D.5.1 Introducción

No hay preguntas de repaso.

D.5.2 Enlace con una biblioteca externa

1. Falso (si contiene código objeto).

2. Ejemplo de código:

```
MiProc PROTO
```

3. Ejemplo de código:

```
call miProc
```

4. Irvine32.lib.
5. Kernel32.lib.
6. Kernel32.lib es una biblioteca de vínculos dinámicos que forma parte fundamental del sistema operativo MS Windows.
7. %1.

D.5.3 La biblioteca de enlace del libro

1. Procedimiento RandomRange.
2. Procedimiento WaitMsg.
3. Ejemplo de código:

```
    mov  eax,700  
    call Delay
```

4. Procedimiento WriteDec.
5. Procedimiento Gotoxy.
6. INCLUDE Irvine32.inc.
7. Instrucciones PROTO (prototipos de procedimientos) y definiciones de constantes. (También hay macros de texto, pero en este capítulo no se mencionaron).
8. ESI contiene la dirección inicial de los datos, ECX contiene el número de unidades de datos y EBX contiene el tamaño de la unidad de datos (byte, palabra o doble palabra).
9. EDX contiene el desplazamiento de un arreglo de bytes, y ECX contiene el número máximo de caracteres a leer.
10. Acarreo, Signo, Cero y Desbordamiento, y EFL muestra los bits de bandera en hexadecimal.
11. Ejemplo de código:

```
.data  
cad1 BYTE "Escriba número de identificación: ",0  
cadID BYTE 15 DUP(?)  
.code  
    mov  edx,OFFSET cad1  
    call WriteString  
    mov  edx,OFFSET cadID  
    mov  ecx,(SIZEOF cadID) - 1  
    call ReadString
```

D.5.4 Operaciones de la pila

1. SS y ESP.
2. La pila en tiempo de ejecución es el único tipo de pila que la CPU maneja en forma directa. Por ejemplo, contiene las direcciones de retorno de los procedimientos a los que se llamó.
3. UEPS significa “último en entrar, primero en salir”. El último valor que se mete a la pila es el primer valor que se saca de la misma.
4. ESP se decrementa por 4.
5. Verdadero.
6. Falso (se pueden meter valores de 16 y de 32 bits).
7. Verdadero.
8. Falso (sí puede, del procesador 80186 en adelante).
9. PUSHAD.

10. PUSHFD.
11. POPFD.
12. El método de NASM permite al programador especificar qué registros se van a meter. Por otro lado, PUSHAD no tiene esa flexibilidad. Esto se vuelve importante cuando un procedimiento debe guardar varios registros y al mismo tiempo regresar un valor al procedimiento que lo llamó en el registro EAX. En este tipo de situación, EAX no puede meterse y sacarse, ya que se perdería el valor de retorno.
13. Equivalente a PUSH EAX:

```
sub esp,4
mov [esp],eax
```

D.5.5 Definición y uso de los procedimientos

1. Verdadero.
2. Falso.
3. La ejecución continuaría más allá del final del procedimiento, tal vez hasta el inicio de otro procedimiento. Por lo general, este tipo de error de programación es muy difícil de detectar.
4. *Recibe* indica los parámetros de entrada que se proporcionan al procedimiento, cuando se hace su llamada. *Devuelve* indica qué valor, si lo hay, produce el procedimiento cuando regresa al procedimiento que lo llamó.
5. Falso (mete el desplazamiento de la instrucción que va *después* de la llamada).
6. Verdadero.
7. Verdadero.
8. Falso (no hay operador NESTED).
9. Verdadero.
10. Falso.
11. Verdadero (también recibe una cuenta del número de elementos de un arreglo).
12. Verdadero.
13. Falso.
14. Falso.
15. Las siguientes instrucciones tendrían que modificarse:

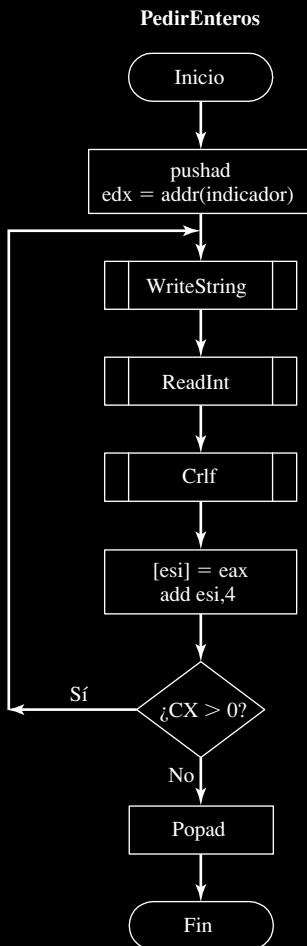
add eax[esi]	se convierte en -->	add ax,[esi]
add esi,4	se convierte en -->	add esi,2

D.5.6 Diseño de programas mediante el uso de procedimientos

1. Descomposición funcional, o diseño arriba-abajo.
2. Clrscr, WriteString, ReadInt y WriteInt.
3. Un programa maestro contiene todos sus procedimientos importantes, pero se encuentran vacíos o casi vacíos.
4. Falso (si recibe un apuntador a un arreglo).
5. Las siguientes instrucciones tendrían que modificarse:

mov esi, eax	se convierte en -->	mov [esi],ax
add esi,4	se convierte en -->	add esi,2

6. Diagrama de flujo del procedimiento PedirEnteros:



D.6 Procesamiento condicional

D.6.1 Introducción

No hay preguntas de repaso.

D.6.2 Instrucciones booleanas y de comparación

1. (a) 00101101 (b) 01001000 (c) 01101111 (d) 10100011
2. (a) 85h (b) 34h (c) BFh (d) AEh
3. (a) CF=0, ZF=0, SF=0
(b) CF=0, ZF=0, SF=0
(c) CF=1, ZF=0, SF=1

4. and ax, 00FFh
5. or ax,0FF00h
6. xor eax,0FFFFFFFh
7. test eax,1 ; (se activa el bit inferior si eax es impar)
8. or al,000100000b
9. and al,00001111b
10. Ejemplo de código:

```
.data
valMem DWORD ?
.code
mov al,BYTE PTR valMem
xor al,BYTE PTR valMem+1
xor al,BYTE PTR valMem+2
xor al,BYTE PTR valMem+3
```

D.6.3 Saltos condicionales

1. JA, JNBE, JAE, JNB, JB, JNAE, JBE, JNA.
2. JG, JNLE, JGE, JNL, JL, JNGE, JLE, JNG.
3. JCXZ.
4. Sí.
5. No (JB usa operandos sin signo, mientras que JL usa operandos con signo).
6. JBE.
7. JL.
8. No (8109h es negativo y 26h es positivo).
9. Sí.
10. Sí (la representación sin signo de -42 se compara con 26).
11. Código:

```
cmp dx,cx
jbe L1
```
12. Código:

```
cmp ax,cx
jg L2
```
13. Código:

```
and al,11111100b
jz L3
jmp L4
```
14. La instrucción XOR en la secuencia de tres instrucciones siempre borra la bandera Acarreo. La instrucción BTC puede borrar o no la bandera Acarreo, dependiendo del valor en semáforo.

D.6.4 Instrucciones de saltos condicionales

1. Falso.
2. Verdadero.
3. Verdadero.
4. Ejemplo de código:

```
.data
```

```

arreglo SWORD 3,5,14,-3,-6,-1,-10,10,30,40,4
centinela SWORD 0
.code
main PROC
    mov esi,OFFSET arreglo
    mov ecx,LENGTHOF arreglo
siguiente:
    test WORD PTR [esi],8000h      ; prueba el bit de signo
    pushfd                         ; mete banderas en la pila
    add esi,TYPE arreglo
    popfd                          ; saca banderas de la pila
    loopz siguiente                 ; continua ciclo mientras
ZF=1
    jz terminar                    ; no encuentre ninguno
    sub esi,TYPE arreglo          ; ESI apunta al valor

```

5. Si no se encontrara un valor que coincidiera, ESI terminaría apuntando más allá del final del arreglo. Esto podría ocasionar corrupción en los datos si ESI se desreferenciara y se usara para modificar la memoria.

D.6.5 Estructuras condicionales

Vamos a suponer que todos los valores son sin signo en esta sección.

1. Ejemplo de código:

```

cmp bx,cx
jna siguiente
mov X,1
siguiente:

```

2. Ejemplo de código:

```

cmp dx,cx
jnbe L1
mov X,1
jmp siguiente
L1: mov X,2
siguiente:

```

3. Ejemplo de código:

```

cmp val1,cx
jna L1
cmp cx,dx
jna L1
mov X,1
jmp siguiente
L1: mov X,2
siguiente:

```

4. Ejemplo de código:

```

cmp bx,cx
ja L1
cmp bx,val1
ja L1
mov X,2
jmp siguiente
L1: mov X,1
siguiente:

```

5. Ejemplo de código:

```
cmp bx,cx ; ¿bx > cx?
```

```

jna L1 ; no: prueba condición después de OR
cmp bx,dx ; sí: ¿es bx > dx?
jna L1 ; no: prueba condición después de OR
jmp L2 ; sí: establece X a 1
-----OR(dx > ax) -----
L1: cmp dx,ax ; ¿dx > ax?
    jna L3 ; no: establece X a 2
    L2: mov X,1 ; sí: establece X a 1
        jmp siguiente ; y termina
    L3: mov X,2 ; establece X a 2
Siguiente:

```

6. Los futuros cambios en la tabla alterarán el valor de NumeroDeEntradas. Podríamos olvidar actualizar la constante en forma manual, pero el ensamblador puede ajustar correctamente un valor calculado.

7. Ejemplo de código:

```

.data
suma DWORD 0
ejemplo DWORD 50
arreglo DWORD 10,60,20,33,72,89,45,65,72,18
TamArreglo = ($ - arreglo) / TYPE arreglo

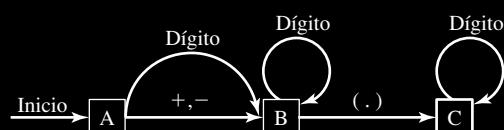
.code
    mov    eax,0           ; suma
    mov    edx,ejemplo      ; índice
    mov    esi,0
    mov    ecx,TamArreglo

L1:   cmp    esi,ecx
    jnl    L5
    cmp    arreglo[esi*4],edx
    jng    L4
    add    eax,arreglo[esi*4]
L4:   inc    esi
    jmp    L1
L5:   mov    suma,eax

```

D.6.6 Aplicación: máquinas de estado finito

1. Un gráfico dirigido (también conocido como *diágrafo*).
2. Cada nodo es un estado.
3. Cada flanco es una transición de un estado a otro, producida por alguna entrada.
4. Estado C.
5. Un número infinito de dígitos.
6. La FSM entra en un estado de error.
7. No. La FSM propuesta permitiría que un entero con signo consistiera de sólo un signo positivo (+) o negativo (-). La FSM en la sección 6.6.2 no permite eso.
8. FSM que reconoce números reales sin exponentes:



D.6.7 Directivas de decisión

No hay preguntas de repaso.

D.7 Aritmética de enteros

D.7.1 Introducción

No hay preguntas de repaso.

D.7.2 Instrucciones de desplazamiento y rotación

1. ROL.

2. RCR.

3. SAR.

4. RCL.

5. Ejemplo de código:

```
shr al,1           ; desplaza AL hacia la bandera Acarreo
jnc siguiente      ; ¿se activó la bandera Acarreo?
or al,80h          ; sí: activa el bit superior
siguiente:         ; no: no hace nada
```

6. La bandera Acarreo recibe el bit inferior de AX (antes del desplazamiento).

7. shl eax,4

8. shr ebx,2

9. ror dl,4 (o: rol dl,4)

10. shld dx,ax,1

11. (a) 6Ah (b) EAh (c) FDh (d) A9h

12. (a) 9Ah (b) 6Ah (c) 0A9h (d) 3Ah

13. Ejemplo de código:

```
shr ax,1          ; desplaza AX hacia bandera Acarreo
rcr bx,1          ; desplaza bandera Acarreo hacia BX
;Usando SHRD:
shrd bx,ax,1
```

14. Ejemplo de código:

```
mov ecx,32        ; contador del ciclo
mov bl,0          ; cuenta los bits que son '1'
L1: shr eax,1     ; lo desplaza hasta la bandera Acarreo
jnc L2            ; ¿se activó la bandera Acarreo?
inc bl            ; sí: suma a la cuenta de bits
L2: loop L1       ; continúa el ciclo
; si BL es impar, borra la bandera de paridad
; si BL es par, establece la bandera de paridad
shr bl,1          ; 
jc  impar          ; 
mov bh,0          ; 
or  bh,0           ; PF = 1
jmp siguiente      ; 
impar:             ; 
    mov bh,1
```

or bh,1 ; PF = 0
siguiente:

D.7.3 Aplicaciones de desplazamiento y rotación

- Para este problema, tenemos que empezar con el byte de mayor orden y avanzar hasta el byte inferior:

```
arregloBytes BYTE 81h,20h,33h
.code
shr arregloBytes+2,1
rcr arregloBytes+1,1
rcr arregloBytes,1
```

- Para este problema, tenemos que empezar con la palabra de menor orden y avanzar hasta la palabra superior:

```
arregloBytes WORD 810Dh,0C064hh,93ABh
.code
shl arregloBytes+2,1
rcl arregloBytes+1,1
rcl arregloBytes+4,1
```

- El multiplicador (24) puede factorizarse en $16 * 8$:

```
mov ebx,eax ; guarda una copia de eax
shl eax,4 ; multiplica por 16
shl ebx,3 ; multiplica por 8
add eax,ebx ; suma los productos
```

- Como explica la sugerencia, el multiplicador (21) puede factorizarse en $16 * 4 + 1$:

```
mov ebx,eax ; guarda una copia de eax
mov ecx,eax ; guarda otra copia de eax
shl eax,4 ; multiplica por 16
shl ebx,2 ; multiplica por 4
add eax,ebx ; suma los productos
add eax,ecx ; suma el valor original de eax
```

- Hay que cambiar la instrucción en la etiqueta L1 a shr eax,1

- Vamos a suponer que la palabra de hora está en el registro DX:

```
shr dx,5 ; (ceros a la izquierda opcionales)
and d1,00111111b ; guarda en variable
```

D.7.4 Instrucciones de multiplicación y división

- El producto se almacena en registros que tienen el doble del tamaño del multiplicador y del multiplicando. Por ejemplo, si multiplicamos 0FFh por 0FFh, el producto (FE01h) cabe fácilmente dentro de 16 bits.
- Cuando el producto se ajusta completamente dentro del registro inferior del producto, IMUL extiende el signo del producto hacia el registro superior. Por otro lado, MUL extiende el producto con ceros.
- Con IMUL, las banderas Acarreo y Desbordamiento se activan cuando la mitad superior del producto no es una extensión de signo de la mitad inferior del producto.
- EAX.
- AX.
- AX.
- Ejemplo de código:

```
mov ax,dividendoInferior
 cwd ; extiende el signo del dividendo
```

```
    mov  bx, divisor
    idiv bx
```

8. DX = 0002h, AX = 2200h.

9. AX = 0306h.

10. EDX = 0, EAX = 00012340h.

11. La instrucción DIV producirá un desbordamiento de división, por lo que no se pueden determinar los valores de AX y DX.

12. Ejemplo de código:

```
    mov  ax,3
    mov  bx,-5
    imul bx
    mov  val1,ax           ; producto

    // solución alternativa:
    mov  al,3
    mov  bl,-5
    imul bl
    mov  val1,ax           ; producto
```

13. Ejemplo de código:

```
    mov  ax,-276
    cwd
    mov  bx,10             ; extiende el signo de AX hacia DX
    idiv bx
    mov  val1,ax           ; cociente
```

14. Implemente la expresión sin signo $\text{val1} = (\text{val2} * \text{val3}) / (\text{val4} - 3)$:

```
    mov eax,val2
    mul val3
    mov ebx,val4
    sub ebx,3
    div ebx
    mov val1,eax
```

(Puede sustituir cualquier registro de propósito general por EBX en este ejemplo).

15. Implemente la expresión con signo $\text{val1} = (\text{val2} / \text{val3}) * (\text{val1} + \text{val2})$:

```
    mov eax,val2
    cdq
    idiv val3             ; extiende EAX hacia EDX
                           ; EAX = cociente
    mov ebx,val1
    add ebx,val2
    imul ebx
    mov val1,eax           ; 32 bits inferiores del producto
```

(Puede sustituir cualquier registro de propósito general por EBX en este ejemplo).

D.7.5 Suma y resta extendidas

1. La instrucción ADC suma un operando de origen y la bandera Acarreo a un operando de destino.
2. La instrucción SBB resta un operando de origen y la bandera Acarreo de un operando de destino.
3. EAX = C0000000h, EDX = 00000010h.
4. EAX = F0000000h, EDX = 000000FFH.
5. DX = 0016H.

6. Al corregir este ejemplo, es más fácil reducir el número de instrucciones. Podemos usar un solo registro (ESI) para indexar las tres variables. ESI debe establecerse en cero antes del ciclo, ya que los enteros se almacenan en orden little endian, en donde sus bytes de menor orden van primero:

```

    mov ecx,8          ; contador del ciclo
    mov esi,0          ; usa el mismo reg índice
    clc               ; borra bandera Acarreo
    sup:
        mov al,byte ptr val1[esi]      ; obtiene el primer número
        sbb al,byte ptr val2[esi]      ; resta el segundo
        mov byte ptr resultado[esi],al ; almacena el resultado
        inc esi                      ; avanza al siguiente par
    loop sup

```

Desde luego que podemos reducir con facilidad el número de iteraciones del ciclo, sumando dobles palabras en vez de bytes.

D.7.6 Aritmética ASCII y con decimales desempaquetados

1. Ejemplo de código:
`or ax,3030h`
2. Ejemplo de código:
`and ax,0F0Fh`
3. Ejemplo de código:
`and ax,0F0Fh ; lo convierte en desempaquetado
aad`
4. Ejemplo de código:
`aam`
5. Ejemplo de código (muestra el valor binario en AX):
`sal16 PROC
 aam
 or ax,3030h
 push eax
 mov al,ah
 call WriteChar
 pop eax
 call WriteChar
 ret
sal16 ENDP`
6. Después de AAA, AX sería igual a 0108h. Intel dice: Primero, si el dígito inferior de AL es mayor que 9 o se activa la bandera Acarreo auxiliar, hay que sumar 6 a AL y 1 a AH. Después en todos los casos, AND AL con 0Fh. Seudocódigo:
`IF ((AL AND 0FH) > 9) OR (AcarreoAux = 1) THEN
 sumar 6 a AL
 sumar 1 a AH
END IF
AND A1 con 0FH;`

D.7.7 Aritmética con decimales empaquetados

1. Cuando la suma de una adición con decimales empaquetados es mayor que 99, DAA activa la bandera Acarreo. Por ejemplo,

```

    mov al,56h
    add al,92h          ; AL = E8h
    daa               ; AL = 48h, CF = 1

```

2. Cuando se resta un entero decimal empaquetado extenso de uno pequeño, DAS activa la bandera Acarreo. Por ejemplo,

```
mov al,56h
sub al,92h
das
; AL = C4h
; AL = 64h, CF = 1
```

3. $n + 1$ bytes.

4. Suponga que $AL = 3Dh$, $AF = 0$, y $CF = 0$. Como el dígito inferior (D) es > 9 , restamos 6 a D. Ahora AL es igual a $37h$. Como el dígito superior izquierdo (3) es ≤ 9 y $CF = 0$, no es necesario ningún otro ajuste. DAS produce $AL = 37h$.

D.8 Procedimientos avanzados

D.8.1 Introducción

No hay preguntas de repaso.

D.8.2 Marcos de pila

1. Verdadero.
2. Verdadero.
3. Verdadero.
4. Falso.
5. Verdadero.
6. Verdadero.
7. Parámetros por valor y parámetros por referencia.
8. Ejemplo de código:

```
mov esp,ebp
pop ebp
```

9. EAX
10. Pasa una constante entera a la instrucción RET. Esta constante se suma al apuntador de pila, justo después que la instrucción RET ha sacado de la pila la dirección de retorno del procedimiento.
11. Diagrama del marco de pila:



12. LEA puede devolver el desplazamiento de un operando indirecto; en especial, es útil para obtener el desplazamiento de un parámetro de pila.
13. Cuatro bytes.
14. Ejemplo de código:

```
SumarTres PROC
; modelado en base al procedimiento SumarDos en la sección 8.4.3:
push ebp
```

```

mov    ebp,esp
mov    eax,[ebp + 16]; 10h
add    eax,[ebp + 12]; 20h
add    eax,[ebp + 8] ; 30h
pop    ebp
ret    12
SumarTres ENDP

```

15. Se extiende con ceros hacia EAX y se mete en la pila.
16. Declaración: LOCAL pArreglo:PTR DWORD
17. Declaración: LOCAL bufer[20]:BYTE
18. Declaración: LOCAL pwArreglo: PTR WORD
19. Declaración: LOCAL miByte:SBYTE
20. Declaración: LOCAL miArreglo[20]:DWORD
21. La convención de llamadas de C, ya que especifica que los argumentos deben meterse en la pila en orden inverso, con lo cual se puede crear un procedimiento/función con un número variable de parámetros. El último parámetro que se mete en la pila puede ser una cuenta que especifique el número de parámetros que ya se encuentran en la pila. Por ejemplo, en el siguiente diagrama, el valor de cuenta se encuentra en [EBP + 8]:

10h	[EBP + 20]
20h	[EBP + 16]
30h	[EBP + 12]
3	[EBP + 8]
(direc retorno)	[EBP + 4]
EBP	<-ESP

D.8.3 Recursividad

1. Falso.
2. Termina cuando n es igual a cero.
3. Las siguientes instrucciones se ejecutan después de que termina cada llamada recursiva:

```

DevolverFact:
    mov    ebx,[ebp+8]
    mul    ebx
L2:   pop    ebp
    ret    4

```

4. El valor calculado excedería el rango de una doble palabra sin signo y se pasaría más allá de cero. La salida parecería menor a 12 factorial.
5. $12!$ Usa 156 bytes de almacenamiento en la pila. *Fundamento:* de la figura 8-1, podemos ver que cuando $n = 0$, se utilizan 12 bytes de la pila (3 entradas). Cuando $n = 1$, se utilizan 24 bytes. Cuando $n = 2$, se utilizan 36 bytes. Por lo tanto, la cantidad de espacio de almacenamiento requerido para $n!$ es $(n + 1) * 12$.
6. Un algoritmo de Fibonacci recursivo utiliza los recursos del sistema de forma ineficiente, debido a que cada llamada a la función de Fibonacci con un valor de n genera llamadas a la función para todos los números de Fibonacci entre 1 y $n - 1$. He aquí el pseudocódigo para generar los primeros 20 valores:

```

for(int i = 1; i <= 20; i++)

```

```
    imprimir( fibonacci(i) );
int fibonacci(int n)
{
    if( n == 1 )
        return 1;
    elseif( n == 2 )
        return 2;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

D.8.4 Directiva .MODEL

1. Un segmento de código y un segmento de datos. Todo el código y los datos son cercanos, lo que significa que sólo necesitan desplazamientos de 16 bits.
2. Se usa en modo protegido. Todos los desplazamientos son de 32 bits y tanto el código como los datos pertenecen al mismo segmento.
3. La opción C preserva el uso de mayúsculas y minúsculas de los identificadores y antepone un guión bajo a los nombres externos. La opción PASCAL convierte todos los identificadores a mayúsculas.

D.8.5 INVOKE, ADDR, PROC y PROTO (opcional)

1. Verdadero.
2. Falso.
3. Falso.
4. Verdadero.
5. Falso.
6. Verdadero.
7. Verdadero.
8. Declaración:

```
ArregloMult PROC apunt1:PTR DWORD,
               apunt2:PTR DWORD,
               cuenta:DWORD           ; (puede ser byte, palabra o doble palabra)
```

9. Declaración:

```
ArregloMult PROTO apunt1:PTR DWORD,
               apunt2:PTR DWORD,
               cuenta:DWORD           ; (puede ser byte, palabra o doble palabra)
```

10. Utiliza parámetros de entrada-salida.
11. Es un parámetro de salida.

D.8.6 Creación de programas con varios módulos

1. Verdadero.
2. Falso.
3. Verdadero.
4. Falso.

D.9 Cadenas y arreglos

D.9.1 Introducción

No hay preguntas de repaso.

D.9.2 Instrucciones primitivas de cadenas

1. EAX.
2. CMPSD.
3. (E)DI.
4. LODSW.
5. Se repite mientras que ZF = 1.
6. 1 (activa).
7. 2.
8. Sin importar qué operandos se utilicen, CMPS de todas formas compara el contenido de la memoria a la que apunta ESI con la memoria a la que apunta EDI.
9. 1 byte más allá del carácter coincidente.
10. REPNE (REPNZ).

D.9.3 Procedimientos de cadenas seleccionados

1. Falso (se detiene cuando se llega al terminador nulo de la cadena más corta).
2. Verdadero.
3. Falso.
4. Falso.
5. 1 (activa).
6. Comprueba que la cadena sólo contenga el carácter a eliminar.
7. El dígito no se modifica.
8. REPNE (REPNZ).
9. La longitud sería $(EDI_{final} - EDI_{inicial}) - 1$.

D.9.4 Arreglos bidimensionales

1. Cualquier registro de 32 bits de propósito general.
2. [ebx + esi].
3. arreglo[ebx + esi].
4. 16.
5. Ejemplo de código:

```
mov esi,2          ; fila
mov edi,3          ; columna
mov eax,[esi*16 + edi*4]
```

6. BP apunta al segmento de pila en modo de direccionamiento real.
7. No (el modelo plano de memoria utiliza el mismo segmento para pila y datos).

D.9.5 Búsqueda y ordenamiento de arreglos de enteros

1. $n - 1$ veces.
2. $n - 1$ veces.
3. No: se decrementa en 1 cada vez.
4. $T(5000) = 0.5 * 10^2$.
5. $(\log_2 128) + 1 = 8$.
6. $(\log_2 n) + 1$.
7. EDX y EDI ya se compararon.
8. Se cambia cada instrucción JMP L4 por JMP L1.

D.10 Estructuras y macros

D.10.1 Estructuras

1. Las estructuras son esenciales cada vez que se requiere pasar una gran cantidad de datos entre dos procedimientos. Puede usarse una variable para guardar todos los datos.

2. Definición de estructura:

```
MiEstruct STRUCT  
campo1 WORD ?  
campo2 DWORD 20 DUP(?)  
MiEstruct ENDS  
  
3. temp1 MiEstruct <>  
4. temp2 MiEstruct <0>  
5. temp3 MiEstruct <,20 DUP(0)>  
6. arreglo MiEstruct 20 DUP(<>)  
7. mov ax,arreglo,campo1  
8. Ejemplo de código:  
  
    mov esi,OFFSET arreglo  
    add esi,3 * (TYPE miEstruct)  
    mov (MiEstruct PTR[esi]).campo1.ax
```

9. 82.

10. 82.

11. TYPE MiEstruct.campo2 (o: SIZEOF Miestruct.campo2)

12. Varias respuestas:

- a. Sí
- b. No
- c. Sí
- d. Sí
- e. No

13. Ejemplo de código:

```
.data  
hora SYSTEMTIME <>  
.code  
mov ax,hora.wHour
```

14. Ejemplo de código:

```
miFigura Triangulo < <0,0>, <5,0>, <7,6> >
```

15. Ejemplo de código (inicializa un arreglo de estructuras Triangulo):

```
.data
TAM_ARREGLO = 5
triangulos Triangulo TAM_ARREGLO DUP(<>)
.code
    mov    ecx,TAM_ARREGLO
    mov    esi,0
L1:   mov    eax,11
    call   RandomRange
    mov    triangulos[esi].Vertice1.X, ax
    mov    eax,11
    call   RandomRange
    mov    triangulos[esi].Vertice1.Y, ax
    add    esi,TYPE Triangulo
    loop   L1
```

D.10.2 Macros

1. Falso.
2. Verdadero.
3. Las macros pueden tener parámetros.
4. Falso.
5. Verdadero.
6. Falso.
7. Para permitir el uso de etiquetas en una macro que se invoca más de una vez desde el mismo programa.
8. ECHO (también el operador %OUT, que se muestra en la última parte del capítulo).
9. Ejemplo de código:

```
mImprimirCar MACRO car,cuenta
LOCAL temp
.data
temp BYTE cuenta DUP(&car),0
.code
    push   edx
    mov    edx,OFFSET temp
    call   WriteString
    pop    edx
ENDM
```

10. Ejemplo de código:

```
mGenAleatorio MACRO n
    mov   eax,n
    call  RandomRange
ENDM
```

11. mPedirEntero:

```
mPedirEntero MACRO indicador,valRetorno
    mWrite indicador
    call   ReadInt
    mov    valRetorno,eax
ENDM
```

12. Ejemplo de código:

```
mEscribirEn MACRO X,Y,literal
    mGotoxy X,Y
    mWrite literal
ENDM
```

13. Ejemplo de código:

```
mWriteStr indicadorNombre
1 push edx
1 mov edx,OFFSET indicadorNombre
1 call WriteString
1 pop edx
```

14. Ejemplo de código:

```
mReadStr nombreCliente
1 push ecx
1 push edx
1 mov edx,OFFSET nombreCliente
1 mov ecx,(SIZEOF nombreCliente) - 1
1 call ReadString
1 pop edx
1 pop ecx
```

15. Ejemplo de código:

```
;-----
mDumpMemx MACRO nombreVar
;
; Muestra una variable en hexadecimal, usando los
; atributos de la variable para determinar el número
; de unidades y el tamaño de la unidad.
;-----
push ebx
push ecx
push esi
mov esi,OFFSET nombreVar
mov ecx,LENGTHOF nombreVar
mov ebx,TYPE nombreVar
call DumpMem
pop esi
pop ecx
pop ebx

ENDM
; Ejemplos de llamadas:

.data
arreglo1 BYTE 10h,20h,30h,40h,50h
arreglo2 WORD 10h,20h,30h,40h,50h
arreglo3 DWORD 10h,20h,30h,40h,50h
.code
mDumpMemx arreglo1
mDumpMemx arreglo2
mDumpMemx arreglo3
```

D.10.3 Directivas de ensamblado condicional

1. La directiva IFB se utiliza para comprobar parámetros en blanco de la macro.
2. La directiva IFIDN compara dos valores de texto y devuelve verdadero si son idénticos. Realiza una comparación sensible a mayúsculas y minúsculas.
3. EXITM.
4. IFIDNI es la versión insensible a mayúsculas/minúsculas de IFIDN.

5. La directiva IFDEF devuelve verdadero si ya se ha definido un símbolo.

6. ENDIF.

7. Ejemplo de código:

```
mWriteLn MACRO texto:=<" ">
    mWrite texto
    call Crlf
ENDM
```

8. Lista de operadores relacionales:

LT Menor que

GT Mayor que

EQ Igual que

NE No es igual que

LE Menor o igual que

GE Mayor o igual que

9. Ejemplo de código:

```
mCopiarPalabra MACRO valEnt
    IF (TYPE valEnt) EQ 2
        mov ax,valEnt
    ELSE
        ECHO Tamano invalido del operando
    ENDIF
ENDM
```

10. Ejemplo de código:

```
mComprobar MACRO Z
    IF Z LT 0
        ECHO **** Operando Z invalido ****
    ENDIF
ENDM
```

11. El operador de sustitución (&) resuelve las referencias ambiguas a los nombres de los parámetros dentro de una macro.

12. El operador de carácter-literal (!) obliga al preprocesador a tratar un operador predefinido como un carácter ordinario.

13. El operador de expansión (%) expande las macros de texto o convierte expresiones constantes en sus representaciones de texto.

14. Ejemplo de código:

```
CrearCadena MACRO valCad
    .data
    temp BYTE "Var&valCad",0
    .code
ENDM
```

15. Ejemplo de código:

```
mLocate -2,20
; (no se genera código ya que xval < 0)
mLocate 10,20
1 mov bx,0
1 mov ah,2
1 mov dh,20
1 mov dl,10
1 int 10h
```

```
mLocate col,fila
1 mov bx,0
1 mov ah,2
1 mov dh,fila
1 mov dl,col
1 int 10h
```

D.10.4 Definición de bloques de repetición

1. La directiva WHILE repite un bloque de instrucciones con base en una expresión booleana.
2. La directiva REPEAT repite un bloque de instrucciones con base en el valor de un contador.
3. La directiva FOR repite un bloque de instrucciones iterando a través de una lista de símbolos.
4. La directiva FORC repite un bloque de instrucciones iterando a través de una cadena de caracteres.
5. FORC
6. Ejemplo de código:

```
BYTE 0,0,0,100
BYTE 0,0,0,20
BYTE 0,0,0,30
```

7. Ejemplo de código:

```
mRepetir MACRO 'X',50
    mov cx,50
    ??0000: mov ah,2
    mov dl,'X'
    int 21h
    loop ??0000

mRepetir MACRO AL,20
    mov cx,20
    ??0001: mov ah,2
    mov dl,AL
    int 21h
    loop ??0001

mRepetir MACRO valByte, valCuenta
    mov cx,valCuenta
    ??0002: mov ah,2
    mov dl,valByte
    int 21h
    loop ??0002
```

8. Si examinamos los datos de la lista enlazada (en el archivo de listado), es aparente que el campo **ApuntSig** de cada **NodoLista** es siempre igual a 00000008 (la dirección del segundo nodo):

Offset	NodoLista	
00000000	00000001	DatosNodo
	00000008	ApuntSig
00000008	00000002	DatosNodo
	00000008	ApuntSig
00000010	00000003	DatosNodo
	00000008	ApuntSig
00000018	00000004	DatosNodo
	00000008	ApuntSig

```

00000020 00000005 DatosNodo
00000008 ApuntSig

00000028 00000006 DatosNodo
00000008 ApuntSig

```

Hicimos esta sugerencia en el texto, cuando dijimos “el valor del contador de ubicación (\$) permanece fijo en el primer nodo de la lista”.

D.11 Programación en MS Windows

D.11.1 Programación de la consola Win32

1. /SUBSYSTEM:CONSOLE
2. Verdadero.
3. Falso.
4. Falso.
5. Verdadero.
6. BOOL = byte, COLORREF = DWORD; HANDLE = DWORD, LPSTR = PTR BYTE, WPARAM = DWORD.
7. GetStdHandle.
8. ReadConsole.
9. Ejemplo del programa *ReadConsole.asm* en la sección 11.1.3:

```
INVOKE ReadConsole, manejadorEntStd, ADDR bufer,
      TamBuf - 2, ADDR bytesLeidos, 0
```

10. La estructura COORD contiene las coordenadas X y Y de la pantalla en medidas de caracteres.
11. Ejemplo del programa *ConsolaI.asm* en la sección 11.14:

```
INVOKE WriteConsole,
      manejadorConsola,          ; manejador de salida de consola
      ADDR mensaje,              ; apuntador a la cadena
      tamanoMensaje,             ; longitud de la cadena
      ADDR bytesEscritos,        ; devuelve el núm de bytes escritos
      0                          ; no se utiliza
```

12. Llamada a **CreateFile** para leer una entrada:

```
INVOKE CreateFile,
      ADDR nombrearchivo,         ; apuntador al nombrearchivo
      GENERIC_READ,               ; modo de acceso
      DO_NOT_SHARE,               ; modo de compartición
      NULL,                      ; apunt a los atrib. de seguridad
      OPEN_EXISTING,              ; opciones de creación de archivo
      FILE_ATTRIBUTE_NORMAL,      ; atributos del archivo
      0                           ; manejador archivo de plantilla
```

13. Llamada a **CreateFile** para crear un nuevo archivo:

```
INVOKE CreateFile,
      ADDR nombrearchivo,
      GENERIC_WRITE,
      DO_NOT_SHARE,
      NULL,
      CREATE_ALWAYS,
      FILE_ATTRIBUTE_NORMAL,
      0
```

14. Llamada a **ReadFile**:

```
INVOKE ReadFile          ; lee archivo y lo coloca en el búfer  
    manejadorArchivo,  
    ADDR bufer,  
    tamBufer,  
    ADDR cuentaBytes,  
    0
```

15. Llamada a **WriteFile**:

```
INVOKE WriteFile         ; escribe texto en archivo  
    manejadorArchivo,   ; manejador del archivo  
    ADDR bufer,         ; apuntador al búfer  
    tamBufer,           ; número de bytes a escribir  
    ADDR bytesEscritos, ; número de bytes escritos  
    0                  ; bandera de ejecución traslapada
```

16. SetFilePointer.

17. SetConsoleTitle.

18. SetConsoleScreenBufferSize.

19. SetConsoleCursorInfo.

20. SetConsoleTextAttribute.

21. WriteConsoleOutputAttribute.

22. Sleep.

D.11.2 Escritura de una aplicación gráfica de Windows

Nota: podrá responder a la mayor parte de estas preguntas si analiza *GraphWin.inc*, el archivo de inclusión que se suministra con los programas de ejemplo de este libro.

1. Una estructura POINT contiene dos campos, ptX y ptY, que describen las coordenadas X y Y (en píxeles) de un punto en la pantalla.
2. La estructura WNDCLASS 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 que pueda mostrarse la ventana principal.
3. *lpfnWndProc* es un apuntador a una función en un programa de aplicación que recibe y procesa los mensajes de eventos activados por un usuario.
4. El campo *style* es una combinación de distintos tipos de opciones, como WS_CAPTION y WS_BORDER, que controlan la apariencia y comportamiento de una ventana.
5. *hInstance* contiene un manejador para la instancia del programa actual. El sistema operativo asigna de manera automática un manejador a cada programa que se ejecuta en MS Windows, cuando el programa se carga en la memoria.
6. (En la sección 11.2.6 se muestra un programa que llama a CreateWindowEx).

El prototipo para **CreateWindowEx** se encuentra en el archivo *GraphWin.inc*:

```
CreateWindowEx PROTO,  
    classexWinStyle:DWORD,  
    className:PTR BYTE,  
    winName:PTR BYTE,  
    winStyle:DWORD,  
    X:DWORD,  
    Y:DWORD,  
    rWidth:DWORD,
```

```
rHeight:DWORD,
hWndParent:DWORD,
hMenu:DWORD,
hInstance:DWORD,
lpParam:DWORD
```

El cuarto parámetro, *winStyle*, determina las características de estilo de la ventana. En el programa WinApp.asm de la sección 11.2.6, cuando llamamos a CreateWindowEx, le pasamos una combinación de constantes de estilo predefinidas:

```
MAIN_WINDOW_STYLE = WS_VISIBLE + WS_DLGFRADE + WS_CAPTION
+ WS_BORDER + WS_SYSMENU + WS_MAXIMIZEBOX + WS_MINIMIZEBOX
+ WS_THICKFRAME
```

La ventana que se describe aquí será visible y tendrá un marco de cuadro de diálogo, una barra de leyenda, un borde, un menú del sistema, un ícono para maximizar, un ícono para minimizar y un marco grueso a su alrededor.

7. Llamada a MessageBox:

```
INVOKE MessageBox, hMainWnd, ADDR TextoBienvenida,
ADDR TituloBienvenida, MB_OK
```

8. Seleccione cualquiera de las dos siguientes (de *GraphWin.inc*):

```
MB_OK, MB_OKCANCEL, MB_ABORTTRYIGNORE, MB_YESNOCANCEL, MB_YESNO,
MB_RETRYCANCEL, MB_CANCELTRYCONTINUE
```

9. Constantes de iconos (seleccione dos):

```
MB_ICONHAND, MB_ICONQUESTION, MB_ICONEXCLAMATION, MB_ICONASTERISK
```

10. Tareas realizadas por **WinMain** (seleccione tres):

- Obtener un manejador para el programa actual.
- Cargar el ícono y el cursor del ratón para el 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 para recibir y despachar mensajes.

11. El procedimiento **WinProc** recibe y procesa todos los mensajes de eventos relacionados con una ventana. Decodifica cada mensaje, y si éste se reconoce, realiza tareas orientadas a la aplicación (o específicas de la aplicación) en relación con el mensaje.

12. Se procesan los siguientes mensajes:

- WM_LBUTTONDOWN, que se genera cuando el usuario oprime el botón izquierdo del ratón.
- WM_CREATE, indica que se acaba de crear la ventana principal.
- WM_CLOSE, indica que está a punto de cerrarse la ventana principal de la aplicación.

13. 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.

14. El cuadro de mensaje se muestra antes de que aparezca la ventana principal de la aplicación.

15. El cuadro de mensaje aparece antes de que se cierre la ventana principal.

D.11.3 Asignación dinámica de memoria

1. Asignación dinámica de memoria.

2. Devuelve un manejador entero de 32 bits para el área del montón de datos existente del programa en EAX.

3. Asigna un bloque de memoria de un montón de datos.

4. Ejemplo de **HeapCreate**:

```
HEAP_INICIO = 2000000 ; 2 MB
```

```
HEAP_MAX = 400000000 ; 400 MB
.data
hHeap HANDLE ? ; manejador para el montón
.code
Invoke HeapCreate, 0, HEAP_INICIO, HEAP_MAX
```

5. Pasa un apuntador al bloque de memoria (junto con el manejador del montón).

D.11.4 Administración de memoria en la familia IA-32

1. (a) La multitarea permite que varios programas (o tareas) se ejecuten al mismo tiempo. El procesador divide su tiempo entre todos los programas en ejecución.
(b) La segmentación provee los medios para aislar segmentos de memoria, unos de otros. Esto permite que varios programas se ejecuten en forma simultánea, sin interferir uno con el otro.
2. (a) Un selector de segmento es un valor de 16 bits almacenado en un registro de segmento (CS, DS, SS, ES, FS, o GS).
(b) Una dirección lógica es una combinación de un selector de segmento y un desplazamiento de 32 bits.
3. Verdadero.
4. Verdadero.
5. Falso.
6. Falso.
7. Una dirección lineal es un entero de 32 bits que se encuentra entre 0 y FFFFFFFFh, que hace referencia a una ubicación de memoria. La dirección lineal también puede ser la dirección física de los datos de destino, si se deshabilita una característica conocida como paginación.
8. Cuando la paginación está habilitada, el procesador traduce cada dirección lineal de 32 bits en una dirección física de 32 bits. Una dirección lineal se divide en tres campos: un apuntador a una entrada en el directorio de páginas, un apuntador a una entrada en la tabla de páginas, y un desplazamiento hacia un marco de página.
9. La dirección lineal es de manera automática una dirección de memoria física de 32 bits.
10. La paginación hace que una computadora pueda ejecutar una combinación de programas que de otra forma no cabrían en la memoria. Para ello, el procesador carga al principio sólo parte de un programa en la memoria, mientras se mantienen las partes restantes en el disco.
11. El registro LDTR.
12. El registro GDTR.
13. Una.
14. Muchas (cada tarea o programa tiene su propia tabla de descriptores).
15. Seleccione cuatro de la siguiente lista: dirección base, nivel de privilegios, tipo de segmento, bandera de segmento presente, bandera de granularidad, límite del segmento.
16. Directorio de páginas, Tabla de páginas y Página (marco de página).
17. El campo Table de una dirección lineal (vea la figura 11-4).
18. El campo Offset de una dirección lineal (vea la figura 11-4).

D.12 Interfaz con lenguajes de alto nivel

D.12.1 Introducción

1. La convención de nomenclatura utilizada por un lenguaje se refiere a las reglas o características relacionadas con el nombramiento de variables y procedimientos.
2. Diminuto, pequeño, compacto, medio, grande y enorme.
3. No, porque el enlazador no encontrará el nombre del procedimiento.

4. El modelo de memoria determina si se realizan llamadas cercanas o lejanas. Una llamada cercana mete sólo el desplazamiento de 16 bits de la dirección de retorno en la pila. Una llamada lejana mete una dirección de segmento/desplazamiento de 32 bits en la pila.
5. C y C++ son sensibles a mayúsculas y minúsculas, por lo que sólo ejecutarán las llamadas a procedimientos cuyos nombres sigan las mismas reglas.
6. Sí, muchos lenguajes especifican que EBP (BP), ESI (SI) y EDI (DI) deben preservarse entre las llamadas a procedimientos.

D.12.2 Código ensamblador en línea

1. El código ensamblador en línea es código fuente en lenguaje ensamblador que se inserta directamente en los programas en lenguajes de alto nivel. Por otro lado, el calificador inline en C++ pide al compilador de C++ que inserte el cuerpo de una función directamente en el código compilado de un programa para evitar el tiempo de ejecución adicional que se requiere para llamar a la función y regresar de ella. Nota: para responder a esta pregunta se requiere cierto conocimiento del lenguaje C++, que no vimos en este libro.
2. La principal ventaja de escribir código en línea es la simpleza, ya que no hay que preocuparse sobre las cuestiones relacionadas con la vinculación externa, los problemas de nomenclatura y los protocolos de paso de parámetros. En segundo lugar, el código en línea se puede ejecutar con más rapidez, ya que evita el tiempo de ejecución adicional que, por lo general, se requiere para llamar a un procedimiento en lenguaje ensamblador y regresar de él.
3. Ejemplos de comentarios (seleccione dos):

```
mov esi,buf           ; inicializa el registro índice
mov esi,buf           // inicializa el registro índice
mov esi,buf           /* inicializa el registro índice */
```

4. Sí.
5. Sí.
6. No.
7. No.
8. Podría ocurrir un error en el programa, ya que la convención __fastcall permite al compilador utilizar registros de propósito general como variables temporales.
9. Usar la instrucción LEA.
10. El operador LENGTH devuelve el número de elementos en el arreglo, especificados por el operador DUP. Por ejemplo, el valor colocado en EAX por el operador LENGTH es 20:

```
miArreglo DWORD 20 DUP(?), 10, 20, 30
.code
mov eax,LENGTH miArreglo      ; 20
```

Observe que el operador LENGTHOF, que presentamos en el capítulo 4, devolvería 23 al aplicarse a miArreglo.

11. El operador SIZE devuelve el producto de TYPE (4) * LENGTH.

D.12.3 Enlace con C++ en modo protegido

1. Deben usarse las palabras clave extern y “C”.
2. La biblioteca Irvine32 utiliza STDCALL, que no es lo mismo que la convención de llamadas de C que utilizan C y C++. La diferencia importante es la forma en que se limpia la pila después de la llamada a una función.
3. Por lo general, los valores de punto flotante se meten en la pila de punto flotante del procesador antes de regresar de la función.
4. Se devuelve un entero corto en el registro AX.

5. printf PROTO C, pCadena:PTR BYTE, args:VARARG.
6. X se meterá al último.
7. Para evitar que el compilador de C++ decore (altere) los nombres de los procedimientos externos. La *decoración de nombres* (también conocida como *planchado de nombres*) se realiza por parte de los lenguajes de programación que permiten la sobrecarga de funciones, con la cual varias funciones pueden tener el mismo nombre.
8. Si la decoración de nombres está en efecto, el nombre de una función externa generada por el compilador de C++ no será igual que el nombre del procedimiento al que se llamó, escrito en lenguaje ensamblador. Es comprensible que el ensamblador no tenga ningún conocimiento sobre las reglas de decoración de nombres que utilizan los compiladores de C++.
9. Prácticamente no hubo cambios, lo cual demuestra que los subíndices de arreglos pueden ser igual de eficientes que los apuntadores, en relación con la manipulación de arreglos.

D.12.4 Enlace con C/C++ en modo de direccionamiento real

1. Los procedimientos en ensamblador que llama Borland C++ deben preservar los valores de BP, DS, SS, SI, DI y la bandera Dirección.
2. INT = 2, enum = 1, float = 4, double = 8.
3. mov eax,[bp + 6].
4. La instrucción ror eax,8 rota hacia fuera el último dígito de EAX, con lo cual evita un patrón recurrente al generar secuencias de números aleatorios pequeños.

D.13 Programación en MS-DOS de 16 bits

D.13.1 MS-DOS y la IBM-PC

1. 9FFFFh.
2. Tabla de vectores de interrupción.
3. 00400h.
4. El BIOS.
5. Suponga que un programa se llama miProg.exe. El siguiente comando redirige su salida a la impresora predeterminada:
`miProg > prn`
6. LPT1.
7. Una rutina de servicio de interrupciones (también conocida como *manejador de interrupciones*) es un procedimiento del sistema operativo que (1) proporciona servicios básicos a los programas de aplicaciones y (2) maneja los eventos de hardware. Para obtener más detalles, vea la sección 16.4.
8. Meter las banderas en la pila.
9. Vea los cuatro pasos de la sección 13.1.4.
10. El manejador de interrupciones ejecuta una instrucción IRET.
11. 10h.
12. 1Ah.
13. 21h * 4 = 0084h.

D.13.2 Llamadas a funciones de MS-DOS (INT 21h)

1. AH.
2. La función 4Ch.

3. Las funciones 2 y 6 escriben un solo carácter.
4. La función 9.
5. La función 40h.
6. Las funciones 1 y 6.
7. La función 3Fh.
8. Las funciones 2Ah y 2Bh. Para mostrar la hora, hay que llamar al procedimiento **WriteDec** de la biblioteca del libro. Ese procedimiento utiliza la función 2 para enviar dígitos a la consola. (Vea los detalles en el archivo *Irvine16.asm*, ubicado en el directorio \Ejemplos\Lib16).
9. Las funciones 2Bh (establecer fecha del sistema) y 2Dh (establecer hora del sistema).
10. La función 6.

D.13.3 Servicios estándar de E/S de archivos de MS-DOS

1. Manejadores de dispositivo: 0 = Teclado (entrada estándar), 1 = Consola (salida estándar), 2 = Salida de error, 3 = Dispositivo auxiliar (asíncrono), 4 = Impresora.
2. Bandera Acarreo.
3. Parámetros para la función 716Ch:

```

AX = 716Ch
BX = modo de acceso (0 = lectura, 1 = escritura, 2 = lectura/escritura)
CX = atributos (0 = normal, 1 = sólo lectura, 2 = oculto,
                 3 = sistema, 8 = ID de volumen, 20h = archivo)
DX = acción (1 = abrir, 2 = truncar, 10h = crear)
DS:SI = segmento/desplazamiento del nombre de archivo
DI = sugerencia de alias (opcional)

```

4. Abrir un archivo existente en modo de entrada:

```

.data
archent BYTE "miarchivo.txt",0
manejEnt WORD ?
.code
    mov ax,716Ch           ; abrir o crear extendido
    mov bx,0                ; modo = sólo lectura
    mov cx,0                ; atributo normal
    mov dx,1                ; acción: abrir
    mov si,OFFSET archent   ; llama a MS-DOS
    int 21h                 ; termina si hay error
    jc terminar
    mov manejEnt,ax

```

5. Es mejor leer un archivo binario de un archivo con la función 3Fh de INT 21h. Se requieren los siguientes parámetros:

```

AH = 3Fh
BX = manejador de archivo abierto
CX = número máximo de bytes a leer
DS:DX = dirección del búfer de entrada

```

6. Después de llamar a INT 21h, se compara el valor de retorno en AX con el valor que se colocó en CX antes de la llamada a la función. Si AX es menor, debemos haber llegado al final del archivo.
7. La única diferencia es el valor en BX. Al leer del teclado, a BX se le asigna el manejador del teclado (0). Al leer de un archivo, a BX se le asigna el manejador del archivo abierto.
8. La función 42h.

9. Ejemplo de código (BX ya contiene el manejador del archivo):

```
mov ah,42h          ; mueve el apuntador del archivo
mov al,0           ; ; método: desplazamiento desde el inicio
mov cx,0           ; ; desplazamientoSup
mov dx,50          ; ; desplazamientoInf
int 21h
```

D.14 Fundamentos de los discos

D.14.1 Sistemas de almacenamiento en disco

1. Verdadero.
2. Falso.
3. Cilindro.
4. Verdadero.
5. 512.
6. Para un acceso más rápido, ya que entre más cercanos estén los cilindros, menor será la distancia que deben recorrer las cabezas de lectura/escritura.
7. Las cabezas de lectura/escritura deben saltar sobre otros cilindros, desperdiциando tiempo e incrementando la probabilidad de que ocurran errores.
8. Volumen.
9. La cantidad promedio de tiempo requerido para mover las cabezas de lectura/escritura de una pista a otra.
10. El marcado de los sectores físicos en las superficies de los discos.
11. La tabla de particiones de disco y un programa que localiza el sector de inicio de una sola partición y ejecuta otro programa que carga el sistema operativo.
12. Una.
13. De sistema.

D.14.2 Sistemas de archivos

1. Verdadero.
2. No, está en el directorio del disco.
3. Falso (todos los sistemas, incluyendo NTFS, requieren por lo menos un clúster para almacenar un archivo.)
4. Falso.
5. Falso.
6. 4GB (se muestra en la tabla 14-1)
7. FAT32 y NTFS.
8. NTFS.
9. NTFS.
10. NTFS.
11. NTFS.
12. 8GB.
13. Registro de inicio, tabla de asignación de archivos, directorio raíz y el área de datos.
14. Esta información está en el desplazamiento 0Dh, en el registro de inicio.

15. Se requerirían dos clústeres de 8KB, para un total de 16,384 bytes. El número de bytes desperdiciados sería de $(16,384 - 8,200)$, o 8,184 bytes.
16. ¡Esta pregunta deberá responderla por su cuenta!

D.14.3 Directorio de disco

1. Verdadero.
2. Falso (se conoce como directorio raíz).
3. Falso (contiene el número de *clúster* inicial).
4. Verdadero.
5. 32.
6. Nombre de archivo, extensión, atributo, etiqueta de hora, etiqueta de fecha, número de clúster inicial, tamaño de archivo.
7. Los bytes de estado y sus descripciones se presentan en la tabla 14-5.
8. Bits 0 a 4 = segundos; bits 5 a 10 = minutos; bits 11 a 15 = horas.
9. El primer byte de la entrada es 4xh, en donde x indica el número de entradas de nombre de archivo extenso que se van a utilizar para el archivo.
10. Dos.
11. En realidad, hay tres campos nuevos: fecha del último acceso, fecha y hora de creación.
12. Vínculos de la tabla de asignación de archivos:

	3	7	8		4	6	eof								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

D.14.4 Lectura y escritura de sectores de disco (7305h)

1. Verdadero.
2. Falso (la función se ejecuta en modo de direccionamiento real).
3. Parámetros:

AX: 7305h
 DS:BX: Segmento/desplazamiento de una variable de la estructura DISKIO
 CX: 0FFFFh
 DL: Número de unidad (0 = predeterminada, 1 = A, 2 = B, 3 = C, etc.)
 SI: Bandera de lectura/escritura
4. INT 10h muestra caracteres gráficos ASCII especiales, sin tratar de interpretarlos como códigos de control (como Tab y retorno de carro).
5. La bandera Acarreo se activa si la función 7305h no puede leer el sector solicitado, y el programa muestra un mensaje de error. (Recuerde que no puede probar este programa en Windows NT, 2000 o XP).

D.14.5 Funciones de archivo a nivel del sistema

1. La función 7303h.
2. La función 7303h.
3. La función 39h (crear subdirectorio) y la función 3Bh (establecer directorio actual).
4. La función 7143h (obtener y establecer atributos de archivo).

D.15 Programación a nivel del BIOS

D.15.1 Introducción

No hay preguntas de repaso.

D.15.2 Entrada de teclado mediante INT 16h

1. Es mejor usar INT 16h.
2. En el búfer de escritura adelantada del teclado, en la ubicación 0040:001E.
3. INT 9h lee el puerto de entrada del teclado, obtiene el código de exploración y produce el código ASCII correspondiente. Inserta ambos en el búfer de escritura adelantada del teclado.
4. La función 05h.
5. La función 10h.
6. La función 11h examina el búfer y nos permite saber qué tecla (si la hay) está esperando.
7. No.
8. La función 12h.
9. El bit 4 (vea la tabla 15-2).
10. Ejemplo de código:

```
L1: mov ah,12h          ; obtiene las banderas del teclado
    int 16h
    test al1,100h        ; ¿se oprimió la tecla Ctrl?
    jz L1                ; no: repite el ciclo
; En este punto, se ha oprimido la tecla Ctrl
```

11. Para comprobar otras teclas, agregue más instrucciones CMP y JE después de las que ya existen en el ciclo. Suponga que queremos comprobar las teclas ESC, F1 e Inicio:

```
L1: .
.
.
cmp ah,1          ; ¿código de exploración de la tecla ESC?
je terminar      ; sí: termina
cmp ah,3Bh        ; ¿tecla de función F1?
je terminar      ; sí: termina
cmp ah,47h        ; ¿tecla Inicio?
je terminar      ; sí: termina
jmp L1            ; no: comprueba el búfer de nuevo
```

D.15.3 Programación de video con INT 10h

1. Nivel de MS-DOS, nivel del BIOS y nivel de video directo.
2. Video directo.
3. En MS Windows hay dos formas de cambiar al modo de pantalla completa:
 - Crear un acceso directo al archivo EXE del programa. Después abrir el cuadro de diálogo Propiedades para el acceso directo, seleccionar las propiedades de Pantalla y seleccionar el modo de Pantalla completa.
 - Abrir una ventana de Símbolo del sistema desde el menú Inicio, después oprimir Alt-Intro para cambiar al modo de pantalla completa. Con la ayuda del comando CD (cambiar directorio), navegue hasta el directorio de su archivo EXE y para que ejecute el programa, escriba su nombre. Si oprime Alt-Intro de nuevo, regresará el programa al modo de ventana.
4. Modo 3 (color, 80 × 25).

5. Código ASCII y atributo (2 bytes).
6. Rojo, verde, azul e intensidad.
7. Fondo: bits 4 a 7. Texto: bits 0 a 3.
8. La función 02h.
9. La función 06h.
10. La función 09h.
11. La función 01h.
12. La función 00h.
13. AH = 2, DH = fila, DL = columna, y BH = página de video.
14. Hay dos formas: (1) usar la función 01h de INT 10h para establecer la línea superior del cursor a un valor ilegal, o (2) usar la función 02h de INT 10h para posicionar el cursor fuera del rango visible de filas y columnas.
15. AH = 6, AL = número de líneas a desplazar, BH = atributo de las líneas desplazadas, CH & CL = esquina superior izquierda de la ventana, y DH & DL = esquina inferior derecha de la ventana.
16. AH = 9, AL = código ASCII del carácter, BH = página de video, BL = atributo y CX = cuenta de repetición.
17. La función 10h, subfunción 03h (establecer AH a 10h y AL a 03h).
18. AH = 06h y AL = 0.
19. Cada píxel en la pantalla está compuesto de tres colores: rojo, verde y azul. Los perros no pueden ver los colores, por lo que no pueden ver los píxeles formados a partir de colores. He tratado de mostrar una imagen de un gato en la pantalla, pero mi perro parece no darse cuenta.

D.15.4 Dibujo de gráficos mediante INT 10h

1. La función 0Ch
2. AH = 0, AL = valor de píxel, BH = página de video, CX = coordenada X, y DX = coordenada Y.
3. Es muy lenta.
4. Ejemplo de código:

```
mov ah,0           ; establece el modo de video
mov al,11h         ; al modo 11h
int 10h            ; llama al BIOS
```

5. El modo 6Ah.
6. Fórmula: sx = (sXOrig + X).
7. a. (350,150) b. (375,225) c. (150,400)

D.15.5 Gráficos de mapas de memoria

1. Falso (cada byte corresponde a 1 píxel).
2. Verdadero.
3. El modo 13h asigna el valor entero de cada píxel a una tabla de colores llamada paleta.
4. El índice de color de un píxel identifica qué color de la paleta se va a usar para dibujar el píxel en la pantalla.
5. Cada entrada en la paleta consiste en tres valores enteros separados (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.
6. (20,20,20).
7. (63,63,63).
8. (63,0,0).

9. Ejemplo de código:

```
; Establece el color de fondo a verde brillante.
mov dx,3c8h ; puerto de paleta de video
mov al,0 ; índice 0 (color de fondo)
out dx,al
mov dx,3c9h ; los colores van al puerto 3C9h
mov al,0 ; rojo
out dx,al
mov al,63 ; verde (intensidad = 63)
out dx,al
mov al,0 ; azul
out dx,al
```

10. Ejemplo de código:

```
; Establece el color de fondo a blanco
mov dx,3c8h ; puerto de paleta de video
mov al,0 ; índice 0 (color de fondo)
out dx,al
mov dx,3c9h ; los colores van al puerto 3C9h
mov al,63 ; rojo = 63
out dx,al
mov al,63 ; verde = 63
out dx,al
mov al,63 ; azul = 63
out dx,al
```

Las últimas dos instrucciones MOV pueden eliminarse si queremos reducir la cantidad de código en este ejemplo.

D.15.6 Programación del ratón

1. La función 0.

2. Ejemplo de código:

```
mov ax,0 ; restablece el ratón
int 33h ; llama al BIOS
cmp ax,0 ; ¿no está disponible el ratón?
je RatonNoDisponible ; sí: muestra mensaje de error
```

3. Funciones 1 y 2.

4. Ejemplo de código:

```
mov ax,2 ; oculta el puntero del ratón
int 33h
```

5. La función 3.

6. Ejemplo de código:

```
mov ax,3 ; obtiene posición y estado del ratón
int 33h
mov ratónX,cx
mov ratónY,dx
```

7. La función 4.

8. Ejemplo de código:

```
mov ax,4 ; establece la posición del ratón
mov cx,100 ; valor de X
mov dx,400 ; valor de Y
int 33h
```

9. La función 5.

10. Ejemplo de código:

```
mov ax,5           ; obtiene información de botones oprimidos del ratón
mov bx,0           ; ID para el botón izquierdo
int 33h
test ax,1          ; ¿está oprimido el botón izquierdo?
jne Boton1         ; si: salta a la etiqueta
```

Nota de implementación: esta función le indicará si cierto botón se encuentra oprimido. Pero si sólo desea las coordenadas del último botón que se oprimió del ratón, no hay necesidad de la instrucción TEST que usamos en el ejemplo.

11. Función 6.

12. Ejemplo de código:

```
mov ax,6           ; obtiene info de los botones que se soltaron del ratón
mov bx,1           ; ID del botón
int 33h
test ax,2          ; ¿se soltó el botón derecho?
jz saltar          ; no - salta
mov ratonX,cx      ; sí, almacena las coordenadas
mov ratonY,dx
saltar:
```

13. Ejemplo de código:

```
mov ax,8           ; establece los límites verticales
mov cx,200          ; límite inferior
mov dx,400          ; límite superior
int 33h
```

14. Ejemplo de código:

```
mov ax,7           ; establece los límites horizontales
mov cx,300          ; límite inferior
mov dx,600          ; límite superior
int 33h
```

15. Suponiendo que las celdas de caracteres son de 8 por 8 píxeles, los valores de las coordenadas X, Y serían (8 * 20), (8 * 10). La celda estará en la posición 160, 80.

16. La esquina superior izquierda de la celda estará en (8 * 22), (8 * 15). Si sumamos 4 a cada uno de esos valores para llevar el ratón hasta el centro de la celda, la respuesta es 180, 124.

17. Douglas Engelbart inventó el ratón de computadora en 1963, en el Instituto de Investigación de Stanford (Stanford Research Institute). (Fuente: http://en.wikipedia.org/wiki/Computer_mouse).

D.16 Programación experta en MS-DOS

D.16.1 Introducción

No hay preguntas de repaso.

D.16.2 Definición de segmentos

1. Declara el inicio de un segmento.
2. Devuelve la dirección de segmento de una etiqueta de datos o de código.
3. La directiva ASSUME hace posible que el ensamblador calcule los desplazamientos de las etiquetas y variables en tiempo de ensamblado. Una directiva como

```
assume DS:miDatos
```

dice al ensamblador, “asume que desde este punto en adelante, todas las referencias a las etiquetas de datos (a través de DS) se localizarán en el segmento llamado **misDatos**”.

4. BYTE, WORD, DWORD, PARA y PAGE.
5. PRIVATE, PUBLIC, MEMORY, STACK, COMMON y AT.
6. DWORD.
7. El tipo de combinación indica al enlazador cómo combinar segmentos que tengan el mismo nombre.
8. Use el tipo de combinación AT. La siguiente instrucción define un segmento con el valor 0040h:

```
bios SEGMENT AT 40h
```

9. El tipo de clase de un segmento proporciona otra forma de combinar los segmentos, en especial aquellos con distintos nombres. Los segmentos que tienen el mismo tipo de clase se cargan juntos, aunque pueden listarse en un orden distinto en el código fuente del programa.
10. Ejemplo de código:

```
mov al,es:[di]
```

11. El tercer segmento también empezará en la dirección 1A060h.

D.16.3 Estructura de un programa en tiempo de ejecución

1. El procesador de comandos comprueba si hay un nombre de archivo con la extensión COM en el directorio actual. Si se encuentra un archivo, se ejecuta. Si no encuentra un archivo que coincida, en la sección 16.3 encontrará una descripción de los pasos subsiguientes.
2. No.
3. Programas de aplicación que se cargan en los 640K inferiores de la memoria. Son transientes ya que, cuando terminan de ejecutarse, se descargan automáticamente de la memoria.
4. Prefijo de segmento del programa.
5. En el desplazamiento 2Ch, dentro del área del prefijo de segmento del programa.
6. Un programa COM es un programa de MS-DOS de un solo segmento. Cuando se almacena en disco como archivo COM, tan sólo es una imagen binaria del programa cuando se carga en memoria.
7. Diminuto (tiny).
8. /T.
9. 64KB.
10. No es eficiente, ya que hasta el programa COM más pequeño utiliza un segmento de memoria de 64K completo.
11. Uno.
12. Todos los registros de segmento se establecen con un desplazamiento de cero dentro del programa. A su vez, el programa se carga en memoria en la primera ubicación de segmento disponible después de otros programas que aún siguen en memoria.
13. La directiva ORG asigna un desplazamiento específico a la siguiente etiqueta o instrucción que va justo después de la directiva. Las direcciones de todas las subsiguientes etiquetas se calculan a partir de ese punto. Por ejemplo, los programas COM siempre tienen ORG 100h en el inicio del código del programa, por lo que la primera instrucción ejecutable se localizará en el desplazamiento 100h.
14. Módulo de carga.
15. DS y ES apuntan al área del prefijo de segmento del programa.
16. MS-DOS asigna de manera automática toda la memoria disponible a un programa cuando se carga por primera vez, a menos que el encabezado EXE del programa limite de manera específica su tamaño máximo de asignación de memoria.

17. El programa EXEMOD muestra estadísticas acerca del uso de la memoria de un programa, y también permite modificar muchas opciones en el encabezado EXE.
18. Ejecutar el programa EXEMOD y pasarle el nombre del archivo EXE. La última línea de la pantalla mostrará el número de entradas de reubicación.

D.16.4 Manejo de interrupciones

1. Muestra un mensaje en la pantalla: "Abortar, reintentar o ignorar?" y termina el programa actual.
2. Una dirección tipo segmento/desplazamiento de 32 bits que apunta a un manejador de interrupciones.
3. En la dirección 0000:0040h, ya que 0040h es igual a $10h * 4$.
4. El chip Controlador de interrupciones programable 8259.
5. La instrucción CLI (borrar bandera de interrupción).
6. La instrucción STI (establecer bandera de interrupción).
7. IRQ 0 tiene la prioridad más alta.
8. Antes de que se haya creado el archivo, ya que el teclado (IRQ 1) tiene una prioridad más alta que la unidad de disco (IRQ 14).
9. INT 9h.
10. Una instrucción IRET al final del manejador de interrupciones devuelve el control al código que se estaba ejecutando cuando ocurrió la interrupción.
11. Las funciones 25h y 35h.
12. Un manejador de interrupciones es cualquier procedimiento que se encarga del procesamiento de una interrupción. Podría cargarse cuando se inicia una aplicación y después descargarse cuando ésta termine. Por otro lado, un programa residente en memoria permanece en la memoria aún hasta después de que el programa que lo instaló haya terminado. Un programa residente en memoria no necesariamente tiene que ser un manejador de interrupciones.
13. Un programa TSR (terminar y permanecer residente) deja parte de sí mismo en la memoria al terminar. Para lograr esto, llama a la función 31h de INT 21h.
14. La computadora puede reiniciarse, o un programa utilitario especial puede eliminar el TSR.
15. En vez de ejecutar una instrucción IRET al terminar, puede ejecutar una instrucción JMP para saltar a la dirección que estaba almacenada antes en el vector de interrupción.
16. Un programa TSR (terminar y permanecer residente).
17. Ctrl + Alt + Mayús derecha + Supr.

D.17 Procesamiento de punto flotante y codificación de instrucciones

D.17.1 Representación binaria de punto flotante

1. Porque el recíproco de -127 es $+127$, lo que generaría un desbordamiento.
2. Porque sumar $+128$ a la desviación del exponente (127) generaría un valor negativo.
3. 52 bits.
4. 8 bits.
5. $1101.01101 = 13/1 + 1/4 + 1/8 + 1/32$.
6. 0.2 genera un patrón de bits que se repite de manera indefinida.
7. $11011.01011 = 1.101101011 \times 2^4$.
8. $0000100111101.1 = 1.001111011 \times 2^{-8}$.
9. $+1110.011 = 1.110011 \times 2^{-3}$, por lo que la codificación es 0 01111100 11001100000000000000000000000000.
10. NaN silencioso y NaN de señalización.
11. $5/8 = 0.101$ binario.

12. $17/32 = 0.10001$ binario.
13. $+10.75 = +1010.11 = +1.01011 \times 2^3$, codificado como 0 10000010 01011000000000000000000000.
14. $-76.0625 = -01001100.0001 = -1.00110000001 \times 2^{-6}$, codificado como:
1 10000101 00110000010000000000000000000000
15. Infinito positivo o negativo, dependiendo del signo del numerador.

D.17.2 Unidad de punto flotante

1. fld st(0).
2. R0.
3. Seleccione tres: código de operación, control, estado, palabra de etiqueta, apuntador de la última instrucción, apuntador de los últimos datos.
4. Decimal codificado en binario.
5. Ninguna.
6. REAL10, 80 bits.
7. Saca a ST(0) de la pila.
8. FCHS.
9. Ninguno, m32pf, m64pf, registro de pila.
10. FISUB convierte el operando de origen de entero a punto flotante.
11. FCOM, o FCOMP.
12. Ejemplo de código:

```
fnstsw ax
lahf
```

13. FILD.
14. Campo RC.
15. 1.010101101 redondeado al número par más cercano se convierte en 1.010101110.
16. -1.010101101 redondeado al número par más cercano se convierte en -1.010101110.
17. Instrucciones de ensamblador:

```
.data
B REAL8 7.8
M REAL8 3.6
N REAL8 7.1
P REAL8 ?
.code
fld M
fchs
fld N
fadd B
fmul
fst P
```

18. Código de lenguaje ensamblador:

```
.data
B DWORD 7
N REAL8 7.1
P REAL8 ?
.code
fld N
fsqrt
fiadd B
fst P
```

D.17.3 Codificación de instrucciones Intel

1. (a) 8E (b) 8B (c) 8A (d) 8A (e) A2 (f) A3
2. (a) 8E (b) 8A (c) 8A (d) 8B (e) A0 (f) 8B
3. (a) D8 (b) D3 (c) 1D (d) 44 (e) 84 (f) 85
4. (a) 06 (b) 56 (c) 1D (d) 55 (e) 84 (f) 81
5. Bytes de lenguaje máquina:
 - a. 8E D8
 - b. A0 00 00
 - c. 8B 0E 01 00
 - d. BA 00 00
 - e. B2 02
 - f. BB 00 10

Índice

A

AAA (Ajuste ASCII después de la suma), instrucción definición, 216 detalles, 216 suma de varios bytes con, 216-218
AAD (Ajuste ASCII antes de la división), instrucción, 218, 621
AAM (Ajuste ASCII después de la multiplicación), instrucción, 218, 621
AAS (Ajuste ASCII después de la resta), instrucción, 218, 621
Acarreo auxiliar, bandera, 90-91 bandera, 89, 90-91, 198, 204
Acceso aleatorio, memoria de (RAM), 26
ADC (suma con acarreo), instrucción, 194, 213, 621
ADD, instrucción, 59, 87-88, 622
ADDR, operador, 249-250
Administración de memoria, 393-399 direcciones lineales, 394-397 terminología, 393-394 traducción de páginas, 397-398
Administrador de tareas, herramienta, 394
ALIGN, directiva, 95
Alineación miembros de estructuras, 301 tipos de, 539 variables de estructuras, 302

AND

instrucción, 21, 152-153 banderas y, 152 combinaciones de operandos, 152 definición, 151 detalles, 622 lógico, operador, 173-174, 186

Anidado(a)s

ciclos, 106 llamadas a procedimientos definición, 136 ilustración, 138 macros, 317-318

ANSI, conjunto de caracteres, 17

Apuntador de instrucciones, 27, 35

Apuntadores

apuntador cercano, 102 apuntador lejano, 102 comparación con subíndices, 414

definición, 102

ejemplo, 103

establecer un apuntador

de archivo, 365

mover un apuntador

de archivo, 452

retorno de procedimientos, 307

tipos de apuntadores, 102

Archivos

abrir, 451-452

apuntadores, mover, 452

atributos, obtener/establecer, 486-487

binarios, crear, 458-461

cifrado de, 406-409

creación de, 361-364, 451-452

de texto, leer/copiar, 454-456

E/S, 365-370

escribir en, 365

leer, 364

manejadores de, cerrar, 452

Argumentos

comprobación en tiempo

de ensamblado, 254-255

conflicto de tamaño, 257

de 8 y 16 bits, 229-230

de referencia, 226

de varias palabras, 230-231

faltantes, comprobación,

326-328

inicializadores,

predeterminados, 328

meter en la pila, 225, 226

paso de, a procedimientos,

138-139

valor, 226

Aritméticas, expresiones, 89,

211-212

Aritméticos, desplazamientos, 195

Arreglos

bidimensionales, 282-285

búsqueda en, 287-294

búsqueda secuencial en, 164

cálculo del tamaño de, 73

de dobles palabras, 68, 73, 85

de dobles palabras, copiar,

271, 272

de enteros, suma de, 106,

139-140

de palabras, 68, 73, 85

iterar a través de, 303-304

multiplicación de, 275

número de elementos en, 97

- operando indirecto y, 100
ordenamiento de, 286-287
paso de, 227
producto punto, 585
suma de, 584
- Arriba-abajo, diseño, 143
- ASCII
aritmética con decimales
desempaquetados y, 215-219
cadenas, 17
de dígitos, 18
caracteres de control, 17-18, 439
definición, 16
tabla, uso de, 17
- Asignación dinámica de memoria, 387-393
definición, 387
funciones, 387-393
- ASSUME, directiva, 540
- Atributos, bytes de, 500-501
- AutoCAD, 569
-
- B**
- Banderas, 619-620
acarreo, 89, 90-91, 198, 204
auxiliar, 90, 91
activar, 157
afectadas por la suma y la resta, 89-92
borrar, 157
Cero, 89
de la CPU, 151
desbordamiento, 89, 92, 199
dirección, 271, 624, 641
estado, 36, 84
granularidad, 397
identificadores de letra, 619
instrucción
AND, 152
CMP, 156
NOT, 155
OR, 154
TEST, 155
XOR, 154-155
paridad, 89, 91, 154-155
segmento presente, 397
signo, 91
teclado, 495-496
- Base-índice, operandos, 283-284
- Base-índice-desplazamiento, operandos, 285
- Bibliotecas de vínculos dinámicos (DLLs), 112
- Bidimensionales, arreglos. *Vea también Arreglos*
- operandos
base-índice, 283-284
desplazamiento-base-índice, 285
orden por fila/columna, 282-283
- Binaria, búsqueda, 287-294
definición, 288
eficiencia, 288
implementación
en lenguaje
ensamblador, 289-290
programa de prueba, 290-293
- multiplicación, 202
- representación de punto flotante. *Vea también Punto flotante*
- Punto flotante, procesamiento
creación, 565-567
exponente, 364-365
IEEE, 563-564
mantisa, 563-564
normalizados, 565
signo, 563
suma, 11-12
- Binario largo, método de división 567-568
- Binarios
archivos
creación, 458-461
definición, 458
- enteros
con signo, 10, 14-16
definición, 10
sin signo, 10-11
valores de posición de bit, 10
- BIOS (sistema básico de entrada-salida), 43
área de datos, 491
definición, 47
entrada del teclado, 492
nivel, 48
- programación a nivel de, 490-535
ROM, 434
- Bit
más significativo (MSB), 10, 14
menos significativo (LSB), 10
orden little endian, 70
- Bits
binarios, mostrar, 202-203
cadenas de, 203
de estado, probar, 163-164
instrucciones de prueba, 167-168
intercambiar grupos de, 197
prueba de, 155
recuperar, de bandera
Acarreo, 198
- Booleana, álgebra, 20
- Booleanas
expresiones, 328
funciones, 22-23
- operaciones
AND, 21
expresiones, 20-22
NOT, 20, 21
OR, 21
precedencia
de operadores, 22
- Borland C++, 424
- BOUND, instrucción, 622
- BSF, instrucción, 622
- BSR, instrucción, 622
- BSWAP, instrucción, 623
- BT (prueba de bit), instrucción, 167
definición, 151, 623
detalles, 623
- BTC (prueba de bit y complemento), instrucción, 167
definición, 151, 623
detalles, 623
- BTR (prueba de bit y restablecer), instrucción, 167-168
definición, 151, 623
detalles, 623
- BTS (prueba de bit y establecer), instrucción, 168
definición, 151, 623
detalles, 623
- Burbuja, ordenamiento de, 286-287
definición, 286
lenguaje ensamblador, 287
programa de prueba, 290-293

- seudocódigo, 286-287
- usos, 286
- Bus serial universal (USB), 45
- BuscarArreglo, función, 410-414
 - ejemplo, 410
 - generación de código, 410-412
 - marco de pila, 411
 - rendimiento, comprobar, 413-414
- Buses
 - de datos, 26
 - de dirección, 26
 - PCI, 26, 44
- Búsquedas
 - binarias, 287-294
 - en arreglos de enteros, 287-294
 - secuenciales, 287-288
- BYTE
 - (definir byte), directiva, 66
 - tipo de alineación, 542
- Bytes
 - atributo, 500-501
 - comparación, 272
 - definición, 12
 - estado de nombre de archivo, 473
 - estado del teclado, 554
 - mover, 271

- C**
- C, lenguaje
 - acceso a los parámetros de pila y, 227-228
 - administrador del montón, 387
 - Biblioteca C estándar, 419-422
 - enlazar con
 - en modo
 - de direccionamiento real, 423-430
 - en modo protegido, 409-423
 - especificador, 248
 - funciones, llamar a, 415-416
- C++
 - acceso a los parámetros de pila, 227-228
 - administrador del montón, 387
 - código sin optimizar, comparación con ensamblador, 414
- compiladores, 211
- decoración de nombres, 410
- directiva _asm, 404-406
- enlazar con
 - en modo
 - de direccionamiento real, 423-430
 - en modo protegido, 409-423
- enlazar MASM con, 412
- funciones, llamar a, 415-416
- optimización del lenguaje
 - ensamblador, 410-414
- programa de inicio, 406, 418-419
- Caché, memoria, 30
- Cadena, constantes de, 54
- Cadenas
 - ASCII, 17, 18
 - cálculo del tamaño de, 73
 - cifrado, 165-167, 444
 - como indicadores de pantalla, 145
 - comparación, 273-274
 - copiar, 106-107, 270-271
 - de bits, 203
 - de colores, 509-511
 - definición, 67
 - escribir, 324
 - en modo de teletipo, 508
 - instrucciones de primitivas de, 270-276
 - lista de, 270
 - uso de la bandera Dirección, 271
 - invertir, 133-134
 - lectura, 321-322
 - numéricas, 54
 - primitivas de, 270
 - procedimientos de, 276-282
 - programa de demostración de la biblioteca de, 280-282
 - validación, 180
- Caja blanca, prueba, 172-173
- CalcSuma, procedimiento, 243
- CALL, instrucción
 - definición, 136, 623
 - detalles, 623
 - ejecutar, 137
 - ejemplo, 136
- Canalización de varias etapas, 27-29
- Canalizada, ejecución
 - con canalización de seis etapas, 29
 - con una sola canalización, 29
 - superescalar de seis etapas, 30
- Caracteres
 - almacenamiento, 16-18
 - constantes, 54
 - destello, 509
 - escritura, 506-507
 - lectura, 506
- Carácter-literal, operador (!), 335
- Cargar y ejecutar, proceso, 31
- Cartesianas, coordenadas
 - conversión a coordenadas de pantalla, 517-518
 - eje X, eje Y, 515
 - programa de ejemplo 515-518
- CBW (convertir byte en palabra), instrucción, 209, 623
- CDQ (convertir doble palabra en palabra cuádruple), instrucción, 209, 624
- Ciclos
 - anidados, 106
 - destino, 105
 - instrucción IF anidada en, 175-176
 - WHILE, 174-176, 188
- Cifrado
 - de archivos, 406-409
 - de cadenas, 165-167, 444
 - simétrico, 165
- Cilindros, 66
- Clase, tipo de, 540
- CLC (borrar bandera Acarreo), instrucción, 624
- CLD (borrar bandera Dirección), instrucción, 624
- CLI (borrar bandera Interrupción), instrucción, 550-551, 624
- CloseFile, procedimiento, 113, 115
- CloseHandle, función, 364
- Clrscr, procedimiento, 113, 115, 511
- Clústeres. Vea también Archivos, sistemas de
 - definición, 468
 - número inicial de, 473
 - tamaños de, 469
- CMC (complementar bandera Acarreo), instrucción, 624
- CMP (comparar), instrucción, 156, 159, 625

- CMPSB (comparar bytes), instrucción, 272-273, 625
- CMPSD (comparar dobles palabras), instrucción, 272-273, 625
- CMPSW (comparar palabras), instrucción, 272-273, 625
- CMPXCHG (comparar e intercambiar), instrucción, 625
- .CODE, directiva, 537
- Codificación
estilos de, 60
instrucciones de, 588-596
- Código
clave virtual, 357
etiquetas de, 56
estándar estadounidense para el intercambio de información.
Vea también ASCII
macros que contienen, 317
MASM, generación de, 238
mezclar datos y, 71
ver, 211
- Cola de instrucciones, 27
- Color, índices de, 519
- Colores
cadenas de, 509-511
control de, 499-511
ejemplo de mezcla de, 500
intensidad de, 508
primarios, mezcla de, 499-500
RGB, 520
- Columnas, orden por, 282
- COM, programas, 545-546
- Combinación, tipo de, 539-540
- Comentarios
campos, 61
información, 57
macro, 316
métodos de especificación, 57
- Comparaciones
con signo, 161-162, 185-186
de bytes, 272
de cadenas, 273-274
de dobles palabras, 272-273
de palabras, 272
igualdad, 159-160, 581-582
registros, 186
sin signo, 161, 185-186
valor de punto flotante, 579-582
- Complemento a dos, notación de, 15-16
- Compuestas, expresiones, 186-187
- Computadora con Conjunto complejo de instrucciones (CISC), diseño, 38, 588
- Condicional
directivas de ensamblado, 326-338
procesamiento, 150-192
directivas de decisión, 184-189
estructuras, 170-179
instrucciones booleanas y de comparación, 151-158
instrucciones de ciclo, 169-170
máquinas de estado finito, 179-183
saltos, 158-169
transferencia, 104
- Condicionales
estructuras. *Vea también*
Estructuras
ciclos WHILE, 174-177
definición, 170
expresiones compuestas, 173-174
instrucciones IF
estructuradas por bloques, 170-173
selección controlada por tablas, 177-179
- saltos
aplicaciones, 163-167
comparaciones con signo, 161-162
comparaciones de igualdad, 159-160
comparaciones sin signo, 161
definición, 158
nemónicos, 629
para búsqueda secuencial en arreglos, 164
para cifrado de cadenas, 165-167
para prueba de bit de estado, 163-164
rangos, 162-163
tipos, 159-163
- Condiciones
compuestas, 184
definición, 184
previas, 135
- Conjunto
de instrucciones, arquitectura de, 7, 8
reducido de instrucciones (RISC), procesadores, 38, 588
- Consola, entrada de, 354-360. *Vea también* Win32, Consola búfer, 354
carácter individual, 357-358
estado del teclado, 359
salida de, 360-361
- Consola1, program, 360-361
- Constantes
de cadena, 54
de caracteres, 54
de números reales, 53-54
enteras, 52
simbólicas, 72-75
- Control
banderas de, 36
caracteres ASCII de, 17-18, 439
registros de, 36
- Controlada por tabla, selección, 177-178
- Controlador
de disco, firmware, 464
de interrupciones programable (PIC), 44
- Convención de llamadas, 403
- CreateFile, función, 361-364
definición, 361
ejemplos, 363-364
parámetros, 362-363
- CreateOutputFile, procedimiento, 113, 115-116
- Crlf, procedimiento, 113, 115
- Cuadros de mensajes, mostrar, 352-354
- Cuádruples, palabras, 12
- Cursos
control de, 373
establecer líneas, 502-503
establecer posición de, 503
obtener posición de, 503-504
ocultar/mostrar, 504
posicionar, 321

- CWD (convertir palabra a doble palabra), instrucción, 209, 625
-
- D**
- DAA (ajuste decimal después de la suma), instrucción, 219-220, 626
- DarPasosBorracho, 309-310
- DAS (ajuste decimal después de la resta), instrucción, 220, 626
- .DATA?, directiva, 71
- Datos
- bus de, 26
 - creación de archivos, 453
 - definición de, 64-71
 - estructura de, 360
 - etiquetas, 56
 - instrucciones de transferencia, 79-87
 - macros que contienen, 317
 - mezcla de código y, 71
 - números reales, 69
 - representación de, 9-20
 - caracteres, 16-18
 - numéricos, 9-16, 18
 - sin inicializar, 71
 - tipos intrínsecos de, 64-65
- DEC (decremento), instrucción, 87, 626
- Decimal codificado en binario (BCD), operando, 573
- Decimales
- aritmética
 - desempaquetados, 215-219
 - empaquetados, 219-221
 - enteros
 - conversión de enteros
 - binarios con signo, 15-16
 - binarios sin signo, 10-11
 - conversión de enteros hexadecimales con signo, 16
 - conversión de enteros hexadecimales sin signo, 13-14
 - empaquetados, 219
 - reales, 53
- Decisión, directivas de, 184-189
- Decodificar, 27
- Decoración de nombres, 403-410
- Definición de datos, instrucción, 64-66
- Delay, procedimiento, 113, 116
- Depuración
- Borland C++, 424
 - programas con macros, 315
 - registros de, 36
 - tips, 256-257
- Desbordamiento
- bandera, 89, 92, 199
 - con signo, 199
 - detección de hardware, 92
 - por división, 210-211
- Descomposición funcional, 143
- Descriptoros
- globales, tablas de (GDT), 41, 395
 - locales, tablas de (LDT), 395
 - definición, 41
 - indexar a, 396
 - tablas de, 31, 395
- Desempaquetados, aritmética con decimales, 215-219
- Desplazamiento
- directo, operando, 84-85
 - operaciones de
 - aplicaciones, 201-203
 - aritméticas, 195
 - instrucciones, 195-197, 199-200
 - lógicas, 194
 - múltiples, 195, 196
 - múltiples dobles palabras, 201-202
- Diagramas de flujo
- definición, 140
 - ejemplo de procedimiento, 141
 - estructura IF, 171
 - FSM de entero con signo, 183
- DibujaLineaHorizont, procedimiento, 515
- DibujarLinea, programa, 514-515
- DibujarLineaVert, procedimiento, 515
- Dirección
- bandera, 271, 624, 641
 - bus de, 26
- Direccionamiento real, modo de administración de memoria en, 39-41
- cálculo de direcciones lineales de 20 bits, 40-41
- definición, 33
- enlazar con C/C++ en, 423-430
- MS-DOS en, 432-433
- operandos indirectos, 99
- programación en, 75-76
- Direcciones
- cálculo lineal de 20 bits, 40-41
 - definición, 40
 - lineales, 394-397
 - lógicas, 394
 - segmento-desplazamiento, 40
- Directivas
- de decisión, 184-189
 - de ensamblado condicional, 326-338
 - definición, 55
 - en instrucción de definición de datos, 65
 - iteración, 338
 - MASM, 604-614
 - relacionadas con datos, 94-98
 - signo de igual, 72-73
- Directorios de disco, 472-477
- campo
- atributo, 473-474
 - etiqueta de fecha, 474
 - etiqueta de hora, 474
 - nombre de archivo, 473
- entrada de archivo, 475
- estructura de MS-DOS, 473-475
- padre, 472
- raíz, 471, 472
- subdirectorios, 472, 485-486
- Dirigidos, gráficos, 179
- Discos, 464-468
- áreas principales, 470-471
 - cilindros, 466
 - elementos físicos, 465
 - espacio libre, 483-485
 - fragmentación, 466
 - geometría física, 466
 - particiones, 466-468
 - pistas, 466
 - sectores, 466, 477-482
 - sistemas de archivos, 468-472

- DispatchMessage, función, 382
 Dispositivos, controladores de, 47
 DIV (división sin signo),
 instrucción, 208, 626
 División
 con signo, 197
 de enteros con signo, 209-211
 desbordamiento de, 210-211
 instrucciones de, 208-213
 larga, binaria, 567-568
 operando, 210
 rápida, 196
 Dobles palabras
 arreglo de, 68, 73, 85
 arreglo, copiar, 271-272
 comparación, 272-273
 definición, 12
 mover, 271
 múltiples, desplazar, 201-202
 DumpMem, procedimiento, 113,
 116
 DumpRegs, procedimiento, 113, 116
 DUP, operador, 67, 68
 DWORD (definir doble palabra),
 directiva, 68
-
- E**
- E/S, puertos de, 558
 asignación por memoria, 558
 basados en puertos, 558
 control de hardware con,
 558-560
 instrucciones, 558
 ECHO, directiva, 316
 EFLAGS, registro, 35
 Ejecutables, archivos, 62
 Ejecutar, 27
 Ejemplo1, procedimiento, 238, 239
 Ejemplo2, procedimiento, 239
 ELSE, directiva, 328-329
 .ELSE, directiva, 184-185
 .ELSEIF, directiva, 184
 Empaquetados, aritmética con
 decimales, 219-221
 En línea, código ensamblador,
 404-409
 .asm, directiva, 404
 características, 404-405
 comentarios, 404
 limitaciones, 405
 valores de registros, 405
 END, directiva, 59
 ENDIF, directiva, 184
 ENDIF, directiva, 328-329
 ENDM, directiva, 313-314
 ENDP, directiva, 59
 ENDS, directiva, 539
 Enlazadas, listas, 340-342
 Enlazadores
 archivos creados/actualizados
 por, 64
 definición, 2, 62
 función, 62
 opciones de comandos, 112
 Ensambladores. *Vea también*
 MASM
 definición, 2
 historia, 9
 Microsoft, 602-603
 NASM, 1, 9
 TASM, 1, 9
 Ensamblado-enlazado y ejecución,
 ciclo, 62-64
 ENTER, instrucción, 236-237, 626
 Enteras, expresiones, 52-53
 Enteros
 aleatorios, 127
 aritmética de, 193-223
 aplicaciones de desplazar
 y rotar, 201-204
 ASCII y decimal
 desempaquetado,
 215-219
 decimal empaquetado,
 219-221
 instrucciones de
 multiplicación y
 división, 204-213
 instrucciones para
 desplazar y rotar,
 194-201
 suma y resta extendida,
 213-215
 binarios, 10-11
 con signo, 14-16
 comparación, 185-186
 conversión de enteros
 decimales, 15-16
 división, 209-211
 notación de
 complementos a dos,
 15-16
 positivos/negativos, 14
 tamaños de
 almacenamiento/
 rangos, 16
 validación, 180-183
 valores máximos/
 mínimos, 16
 constantes, 52
 E/S, 125-126
 extensión de cero/signo, 82
 hexadecimales, 13-14
 intercambiar, 256
 largos aleatorios, 428-430
 sumar, 58-62
 tamaños de almacenamiento,
 12-13
 Entorno de ejecución, 34-39
 Entrada-salida, sistema de, 46-49
 Envolturas, programa, 324-325
 EQU, directiva, 74
 Errores
 códigos de MS-DOS, 450
 comprobación de, 356-357
 manejo de, 383, 390
 ErrorHandler, procedimiento, 383
 Escala, factores de, 284
 EscribirColores, programas, 374-375
 Espacios, escribir, 323-324
 Específicos del modelo, registros,
 236
 EstablecerPosicionCursor,
 procedimiento, 186-187
 Estado, banderas de. *Vea también*
 Banderas
 almacenar AH en, 84
 carga, 84
 definición, 36
 Estándar, biblioteca de C, 419-422
 Estática, RAM (SRAM), 45
 Estructuras, 299-313
 alineación de miembros, 301
 de datos, 360
 definición, 299
 definir, 300-301
 distribución de memoria, 300
 estructuras que contienen, 307
 inicializadores de campo,
 300-301
 necesarias, 380-381
 pasos para usar, 300
 predefinidas, 305
 rendimiento de los miembros
 alineados, 304-305

- uniones, 310-312
usos, 299
variables de. *Vea también*
 Variables
 alineación, 302
 declaración, 301-302
 referencia, 302-305
- Etiquetas
 código, 56
 datos, 56
 definición, 55
 inserción, 97
- Exabytes, 13
- Excepciones
 de punto flotante, 573
 enmascarar/desenmascarar, 586-587
 sincronización de, 583-584
- EXE, programas
 definición, 546
 encabezado de, 547
 segmentos, 546
 uso de memoria, 547
- EXITM (terminar macro),
 directiva, 336
- Expansión, operador (%),
 333-335
- Exponentes, 546-565
- ExpressEXITM, directiva, 327
- EXTERN, directiva, 258-259
- EXTERNDEF, directiva, 259-260
- Externos
 identificadores, 403
 llamada a procedimientos,
 258-259
-
- F**
- F2XM1, instrucción, 644
- FABS, instrucción, 644
- Factorial, programa, 244-245
- Factoriales, calcular, 243-246
- FADD (suma), instrucción,
 576-577, 644
- FADDP, instrucción, 644
- FAT12, sistema de archivos, 469
- FAT16, sistema de archivos, 469
- FAT32, sistema de archivos,
 469-470
- FBLD (cargar decimal codificado
en binario), instrucción, 644
- FBSTP, instrucción, 644
- FCHS (cambiar signo), instrucción,
 576, 644
- FCLEX (borrar excepciones),
 instrucción, 644
- FCMOVcc (movimiento
condicional de punto flotante),
 instrucción, 644
- FCOM (comparar valores de punto
flotante), instrucción, 579, 645
- FCOMI, instrucción, 581, 645
- FCOMP, instrucción, 580
- FCOMPP, instrucción, 580
- FCOS (coseno), instrucción, 645
- FDECSTP (decrementar apuntador
de la parte superior de la pila),
 instrucción, 645
- FDIV, instrucción, 579, 645
- FDIVP, instrucción, 645
- FDIVR, instrucción, 645
- FDIVRP, instrucción, 645
- Fecha
 establecer, 446
 etiqueta de, 474
 mostrar, 446
- FFREE, instrucción, 645
- FIADD (sumar entero),
 instrucción, 577-578, 644
- FICOM (comparar entero),
 instrucción, 646
- FIDIV, instrucción, 579, 645
- FIDIVR, instrucción, 645
- Filas, orden por, 282
- FILD (cargar entero), instrucción,
 575, 646
- FIMUL, instrucción, 578, 647
- Fin de línea, caracteres de, 67
- FINCOS, instrucción, 647
- FINIT, instrucción, 574, 646
- FireWire, 46
- FIST (almacenar entero),
 instrucción, 576, 646
- FISTTP, instrucción, 646
- FISUB (restar entero), instrucción,
 578, 648
- FISUBR, instrucción, 648
- Flancos, 179
- FLD (cargar valor de punto
flotante), instrucción, 574-575,
 646
- FLDCW, instrucción, 646
- FLDENV, instrucción, 646
- FMUL, instrucción, 578, 647
- FMULP (multiplicar con pop),
 instrucción, 578, 647
- FNOP, instrucción, 647
- FOR, directiva, 339
- FORC, directiva, 340
- FormatMessage, función, 356
- FPATAN, instrucción, 647
- FPREM, instrucción, 647
- FPTAN, instrucción, 647
- Fracciones, convertir, 567-568
- FRNDINT, instrucción, 647
- FRSTOR, instrucción, 647
- FSAVE, instrucción, 647
- FSCALE, instrucción, 647
- FSIN, instrucción, 647
- FSQRT, instrucción, 647
- FST (almacenar valor de punto
flotante), instrucción, 575-576,
 647
- FSTCW, instrucción, 647
- FSTENV, instrucción, 648
- FSTP (almacenar valor de punto
flotante y pop), instrucción,
 576
- FSTSW, instrucción, 648
- FSUB, instrucción, 578, 648
- FSUBP (restar con pop),
 instrucción, 578, 648
- FSUBR, instrucción, 648
- FSUBRP, instrucción, 648
- FTST, instrucción, 648
- Funciones
 C/C++, 415, 416
 de archivos a nivel
 del sistema, 482-487
 de biblioteca de C, llamar a,
 419-422
 de entrada, 442-443
 de fecha/hora, 446-449
 de MS-DOS, 438-449
 de video, 501
 macro, 336-337
 prototipos de, 415
 relacionadas con píxeles, 513
 salida de, 439-442
 valores de retorno de, 416
- FWAIT, instrucción, 648
- FXAM, instrucción, 648
- FXCH, instrucción, 648
- FXRSTOR, instrucción, 649
- FXSAVE, instrucción, 649
- FXTRACT, instrucción, 649

G

GDTR (Registro de tabla de descriptores globales), 36
GetCommandTail, procedimiento, 113, 116-117, 456-458
GetConsoleCursorInfo, función, 373
GetConsoleScreenBufferInfo, función, 370-371
GetDateTime, procedimiento, 378
GetKeyState, función, 359
GetLastError, función, 356
GetLocalTime, función, 375-376
GetMaxXY, procedimiento, 113, 117
GetMseconds, procedimiento, 113, 117, 128
GetProcessHeap, función, 387-388
GetTextColor, procedimiento, 113, 117
GetTickCount, función, 377
Gigabytes (GB), 12
Gotoxy, procedimiento, 113, 117, 511
Gráficos
 de mapas de memoria, 519-523
 dibujar, 512-519
 modo de, 499
 pixeles, leer/escribir, 513
Granularidad, bandera, 397

H

Hardware
 control con puertos de E/S, 558-560
 detección de desbordamiento, 92
 interrupciones de, 549-550
 requerimientos de, este libro, 2
HeapAlloc, función, 389-390
HeapCreate, función, 388-389
HeapDestroy, función, 389
HeapFree, función, 389-390
Hexadecimales, enteros. *Vea también* Enteros
 complemento a dos, 15
 con signo, conversión decimal, 16

definición, 13
 sin signo, conversión decimal, 13-14
HLT (detener), instrucción, 627
Hola programador, programa, 441-442
Hora
 de creación de archivos, 453
 del sistema, 305-306, 447-449
 etiqueta de, 474

I

IA-32
 administración de memoria, 393-399
 direcciones lineales, 394-397
 terminología, 393-394
 traducción de páginas, 397-398
formato de instrucciones, 588-589
microcomputadora
 interfaces de dispositivos, 45-46
 memoria, 45
 puertos de entrada-salida, 45-46
 salida de video, 44-45
 tarjeta madre, 43-44
procesador
 administración de memoria, 39-43
 arquitectura, 33-39
 conceptos, 25-50
 entorno de ejecución, 34-36
 espacio de direcciones, 34
 familia, 1, 38
 modo 8086 virtual, 33, 39
 modo de administración del sistema (SMM), 33
 modo de
 direcccionamiento real, 33, 39
 modo protegido, 33, 39
 modos de operación, 33
 paginación, 41-43
 prefijo de tamaño de operando, 591-592

registros de ejecución, 34-36
registros del sistema, 36
soporte para máquina virtual, 9
unidad de punto flotante, 36-37

IDE (electrónica de unidad inteligente), 46
Identificadores
 definición, 54
 externos, 403
 lineamientos sobre la creación de, 54-55

IDIV (dividir con signo), instrucción, 209-210, 627
IDTR (Registro de tabla de descriptores de interrupciones), 36

IF
 directiva, 184, 185, 188, 328-329
instrucciones
 anidadas en ciclos, 175-176
 diagrama de flujo, 171
 ejemplo, 171-172
 prueba de la caja blanca, 172-173

IFIDNI, directiva, 329-330
Igualdad, comparaciones de, 159-160
ImprimirArreglo, procedimiento, 292-293

IMUL (multiplicar con signo), instrucción
 definición, 205, 627
 detalles, 627
 ejemplos, 206-207
 formatos
 con dos operandos, 205
 con tres operandos, 205-206
 con un operando, 205
 multiplicación sin signo, 206

IN, instrucción, 558, 628
INC (incrementar), instrucción, 87, 628
INCLUDE, directiva, 59, 75, 125
Incondicional, transferencia, 104
Indexados, operandos. *Vea también*
 Operandos
 definición, 101
 factores de escala, 101-102

- para acceder a arreglos de estructuras, 303 para procesar arreglos, 101
- Indirecto, direccionamiento apuntadores, 102-103 arreglos, 100 definición, 99 operandos indexados, 101-102 operandos indirectos, 99, 303
- Inicializadores de campo, 300-301 definición, 65-66 múltiples, 66-67
- Inicio estado, 179 procedimiento de, 59-60 registro de, 471
- INS, instrucción, 628
- INSB, instrucción, 628
- INSD, instrucción, 628
- Instrucciones ciclo de ejecución arquitectura superescalar, 29-30 canalización de varias etapas, 27-29 decodificar, 27 definición, 27 división, 208-213 ejecutar, 27 multiplicación, 204-208 obtener, 27 pasos, 27 seudocódigo, 27 de comentarios, 57 de control de interrupciones, 550-551 de etiqueta, 55-56 de extensión de signo, 209 de modo de memoria, 592-595 de modo de registro, 591 de primitivas de cadena, 270-276 de punto flotante, 573, 644-649 de símbolos de formato, 620 de transferencia de datos, 79-87 de un solo byte, 589-590
- definición, 55 nemónico de instrucción, 56-57 operandos, 56-57 para desplazar y rotar, 194-201
- INSW, instrucción, 628
- INT (llamada a procedimiento de interrupción), instrucción, 435-436, 628
- INT 10h, funciones de, 657 Función 00h, 502 01h, 502-503 02h, 503 03h, 503-504 06h, 504-505 07h, 506 08h, 506 09h, 506-507 0Ah, 507 0Ch, 513 0Dh, 513 0Fh, 508 10h, 508 13h, 508-509 lista de, 501 modos de gráficos de video, 512 servicios de video, 436
- INT 16h, funciones de, 492-498, 658 Función 03h, 493 05h, 493 10h, 494 11h, 494-495 12h, 495 servicios de teclado, 436, 492-498
- INT 17h, servicios de impresora, 436
- INT 1Ah, hora del día, 436
- INT 1Ch, interrupción de temporizador del usuario, 436
- INT 21h, funciones, 653,656 archivo y directorio, 449
- Función 0Ah, 443 0Bh, 443 1, 442
- 2, 439 25h, 551 2Ah, 446 2Bh, 446 2Ch, 447 2Dh, 447 35h, 551 39h, 485-486 3Ah, 486 3Bh, 486 3Eh, 452 3Fh, 444-445, 461 40h, 441, 460 42h, 452 47h, 486 4Ch, 438-439 5, 440 5706h, 453 6, 440, 442 7143h, 486-487 716Ch, 451 7303h, 483-485 9, 440
- Servicios de disco, 483 servicios de MS-DOS, 436
- INT 23h, manejador de interrupciones Procedimiento, 552
- INT 33h, funciones, 658 Función 0, 524 1, 524 2, 524 3, 525 4, 525 5, 526 6, 526-527 7, 527 8, 527
- Intel 80286, 37 8086, 37 8253, chip temporizador, 559 8255, chip de Interfaz periférica programable, 559
- IA-32, procesador. *Vea también IA-32, procesador*
- Intercambiar, procedimiento, 257
- Interfaz de Lenguaje de alto nivel código de ensamblador en línea, 404-409

convenciones generales, 402-403
enlace
 en modo
 de direccionamiento real, 423-430
 en modo protegido, 409-423
Interfaces de programación de aplicaciones (APIs), 347
Interrupción
 tablas de vectores de, 549
 vectores de, 433
Interrupciones, 650-658
 activadores, 549
 comunes, 436
 de hardware, 549-550
 de PC, 651-652
 de software, 435
 definición, 39, 432
 instrucciones de control, 550-551
 manejadores de, 548-558
 definición, 548-551
 ejecutar, 549
 ejemplo de Ctrl-Inter, 552-553
 personalizadas, escribir, 551-553
 residentes en memoria, 556-557
 usos, 548
 vectorización, 435-436
INTO (interrupción por desbordamiento), instrucción, 628
Intrínsecos, tipos de datos, 64, 65
INVOKE, directiva
 creación de módulos con, 264-266
 definición, 248
 sintaxis, 248-249
 tipos de argumentos, 249
IRET (retorno de interrupción), instrucción, 629
Irvine16.lib, 112
Irvine32.lib, 112, 125
 E/S de archivos, 365-367
 procedimientos de teclado, 357-358
IsDigit, procedimiento, 113, 117-118, 182

J

Java

 administrador del montón de datos, 387
 código byte, 7
 máquina virtual (JVM), 7
 relación de uno a varios, 4
JC (saltar si hay acarreo), instrucción, 457
Jcond, instrucción, 158-159, 629
JCXZ, instrucción, 630
JECXZ, instrucción, 630
JMP, instrucción, 104, 159, 630
JNZ (saltar si no es Cero), instrucción, 158
JZ (saltar si es Cero), instrucción, 158

K

Kilobytes, 12

L

LABEL, directiva, 97-98
LAHF (cargar banderas de estado en AH), instrucción, 84, 630
Largos, nombres de archivo, 475-476
LDS, instrucción, 630
LDTR (Registro de tabla de descriptores globales), 36
LEA (cargar dirección efectiva), instrucción, 235-236, 631
LEAVE, instrucción, 237, 631
LeerArchivo, programa, 368-370
LENGTH, operador, 405-406
LENGTHOF, operador, 97
Lenguaje
 de alto nivel
 comparación con el lenguaje ensamblador, 6
 en la máquina virtual, 8
 funciones de, 46
ensamblador
 aplicaciones, 5-6
 comparación con lenguajes de alto nivel, 6
 definición, 1

elementos básicos, 51-58
en máquina virtual, 8
lenguaje máquina y, 3-4
niveles de acceso, 46
portabilidad, 4
razones para aprender, 4-5
reglas, 5
especificadores de
 lenguaje C, 248
MODEL, directiva, 247-248
LES, instrucción, 630
LFS, instrucción, 630
LGS, instrucción, 630
Lineales, direcciones. *Vea también*
 Direcciones
 definición, 394
 traducción
 a dirección física, 398
 de dirección lógica a, 394, 395

Listado

archivos de, 63
de directorios, programa, 422-423
Little endian, orden, 69-70, 201, 230
LlenarArreglo, procedimiento, 235, 293
LOCAL, directiva, 237-238, 239, 316-317
Locales, variables, 233-236
 crear espacio para, 239, 240
 declaración, 233, 237-238
 definición, 233
 que no son dobles palabras, 239-240
 símbolos, 234
LOCK, instrucción, 631
LODS, instrucción, 631
LODSB, instrucción, 275, 631
LODSD, instrucción, 275, 631
LODSW, instrucción, 275, 631
Lógica, dirección, 394
Lógicos
 desplazamientos, 194
 operadores, 184
LOOP, instrucción
 definición, 105, 632
 detalles, 632
 ejecución de, 105
LOOPD (iterar IA-32), instrucción, 632

- LOOPE (iterar si es igual), instrucción, 169, 632
LOOPNE (iterar si no es igual), instrucción, 169, 632
LOOPNZ (iterar si no es cero), instrucción, 169, 632
LOOPW, instrucción, 632
LOOPZ (iterar si es cero), instrucción, 169, 632
LSS, instrucción, 630
-
- M**
- MACRO, directiva, 313-314
Macros, 313-326
anidadas, 317-318
características, 315-318
comentarios, 316
con código y datos, 317
declarar, 313
definir, 313-314
depurar programas con, 315
en la biblioteca Macros.inc, 318
funciones, 336-337
HolaNuevo, programa, 337
inicializadores de argumentos
predeterminados, 328
invocar, 314-315
llamar, 313
parámetros, 314
requeridos, 315
sintaxis de llamadas, 314
Macros.inc, biblioteca
lista de macros, 318
mDump, 320
mDumpMem, 319-320
mGotoxy, 321
mReadString, 321-322
mShow, 322
mShowRegister, 323
mWriteSpace, 323-324
mWriteString, 324
Maestros, programas
beneficios, 145
definición, 144
ejemplo, 144-145
Main, procedimiento, 538, 553
Manejador_Inter, procedimiento, 553
Manejadores
cerrar, 364
de archivos, cerrar, 452
de consola, 350
- Mantisa (punto flotante), 563-564
definición, 563
precisión, 564
Map, archivos, 64
Mapas de memoria, gráficos de, 519-523
colores RGB, 520
Modo 13h, 519-520
programa, 520-523
Máquina virtual, administrador de (VMM), 397
Máquinas
de estado finito (FSMs), 179-183
definición, 179
diagrama de flujo, 183
entero decimal con signo, 180, 183
estados, 179
flancos, 179
implementación, 181-183
nodos, 179
para una cadena, 180
traducidas a lenguaje ensamblador, 181
virtuales
arquitectura del
procesador IA-32,
soporte, 9
concepto, 7-9
definición, 7, 398
Java (JVM), 7
níveis, 8
MASM (Macro Assembler), 1
comprobación de rango, 85
directives, 604-614
enlazar con Visual C++, 412
generación de código, 238
modo de compatibilidad, 9
nombres de registros, 601
palabras reservadas, 601
referencia, 600-618
tipos de datos de punto
flotante, 574
traducción de tipos de MS Windows, 349
Mayúscula, procedimiento, 229-230
Memoria
administración de, 39-43
caché, 30
de video, 433
- dinámica
asignación de, 387-393
de acceso aleatorio
(DRAM), 45
directa, operandos de, 80-81
DRAM, 45
EPROM, 45
leer de, 30-31
microcomputadora IA-32, 45
modelos, 246, 247, 403
mostrar bloque, 319
operandos, 56
organización de MS-DOS, 433-434
RAM de CMOS, 45
ROM, 45
segmentada, 39
SRAM, 45
virtual, 41
VRAM, 45
MessageBox, función, 381
MessageBoxA, función, 352
Metal-óxido semiconductor complementario (CMOS), RAM, 45
Mezclar, procedimiento, 238
Microarquitectura, 8
Microcomputadoras
diagrama de bloques, 26
diseño básico, 26-27
IA-32, 43-46
Microsoft Assembler (ML), 602-603
Microsoft Visual C++. *Vea también C++*
código de BuscarArreglo
generado por, 410-412
enlazar MASM con, 412
Microsoft Visual Studio,
depurador, 187
MiSub, procedimiento, 213-232, 237-238
MiSub1, procedimiento, 232
MODEL, directiva, 246-248
especificadores de lenguaje, 247-248
modelos de memoria, 246, 247
para codificar programas de 16 bits, 436
sintaxis, 246

-
- Modo de memoria, instrucciones de, 592-595
- Módulos
- crear con directiva EXTERN, 261-264
 - crear con directivas INVOKE y PROTO, 264-266
 - límites, variables/símbolos a través de, 259-260
- Montón de datos, asignación de. *Vea también* Asignación dinámica de memoria
- programas de prueba, 390-393
- Mostrar, procedimiento, 538
- MostrarBufer, procedimiento, 427
- MostrarSuma, procedimiento, 262
- MostrarTabla, función, 416-417
- MOV, instrucción, 59, 81-82
- codificaciones, 591
 - códigos de operación, 593-594
 - de memoria a memoria, 81-82
 - definiciones, 81, 632
 - detalles, 632
 - ejemplo con código máquina, 595
 - ejemplos, 593-596
 - operando de memoria y, 107
 - reglas de uso de operandos, 81
- MOVS (mover cadena), instrucción, 633
- MOVSB (mover bytes), instrucción, 271-272, 633
- MOVSD (mover dobles palabras), instrucción, 271-272, 633
- MOVSW (mover palabras), instrucción, 271-272, 633
- MOVSX (mover con extensión de signo), instrucción
- definición, 83, 633
 - detalles, 633
 - diagrama, 84
 - ejemplos de tamaños de registros, 83
- MOVZX (mover con extensión de ceros), instrucción, 82-83, 331, 633
- MS Windows
- aplicaciones gráficas, escritura de, 379-387
 - nombres de archivo largos, 475-476
- programación, 346-401
- VMM, 397-398
- MS-DOS**
- campos de fecha de archivo, 203
 - códigos de error extendidos, 250
 - cola de comandos, 456-458
 - estructura de directorios, 473-475
 - programación, 432-463
 - en modo de
 - direcciónamiento real, 432-433
 - experta, 536-561
 - interrupciones de software, 435
 - llamadas a funciones, 438-449
 - nombres de dispositivos estándar, 435
 - organización de memoria, 433-434
 - redirección de la entrada-salida, 434-435
 - servicios de E/S de archivos, 449-461
- MsgBox, procedimiento, 113, 118
- MsgBoxAsk, procedimiento, 114, 118
- MUL (multiplicar sin signo), instrucción**
- definición, 204, 633
 - detalles, 633
 - ejemplos, 204-205
 - operandos, 204
- Multimódulos, programas, 258-267
- Multiples segmentos de código, programa, 538
- Multiplexores, 23
- Multiplicación**
- arreglos, 275
 - binaria, 202
 - instrucciones, 204-208
 - operaciones, medir rendimiento, 207-208
- propiedad distributiva de, 202
- rápida, 196
- sin signo, 206
- tabla, 416-419
- Multisegmentos, modelo, 41, 42
- Multitarea**
- comutación de tareas, 32
 - definición, 32, 393
 - preferente, 32
-
- N**
- NaN (no es un número), 566
 - NASM (Netwide Assembler), 1, 9
 - Negativos, enteros, 14, 15
 - NFS, sistema de archivos, 470
 - NGE (negar), instrucción, 88-89, 92, 634
 - Nodos, 179
 - NOLIST, directiva, 125
 - Nombres de archivo, 473, 475-476
 - NOP (ninguna operación), 57, 634
 - Normalizados
 - números
 - binarios de punto flotante, 565
 - finitos, 566
- NOT, operador, 20, 21, 151, 634
- Numéricas, cadenas, 18
-
- O**
- Objeto, archivos, 62
 - OFFSET, operador, 303
 - definición, 94
 - devolver apuntadores, 307
 - ejemplo, 95
 - uso, 102
- OpenInputFile, procedimiento, 114, 119
- Operadores
- carácter-literal, 335
 - de expansión, 333-335
 - de sustitución, 333
 - en tiempo de ejecución, 618
 - especiales, 333-335
 - lista, 615-618
 - lógicos, 184
 - relacionados con datos, 94-98
 - relacionales, 184
 - texto-literal, 335
- Operandos, 56-57
- 16 bits, 592
 - base-índice, 283-284
 - base-índice-desplazamiento, 285
 - codificados en binario (BCD), 537

- desplazamiento directo, 84-85
división, 210
indexados, 101-102, 303-304
indirectos, 99, 303-304
memoria directa, 80-81
notación de instrucciones, 80
tamaño declarado, redefinir, 95-96
tipos de, 80
- OR**
instrucción, 21, 153-154
banderas y, 154
combinaciones de operandos, 153
definición, 151, 634
detalles, 634
uso, 153
operadores lógicos, 174, 186
- Ordenamiento**
burbuja, 286-287
de enteros, 286-287
- OrdenBurbuja**, procedimiento, 238
- OUT**, instrucción, 519, 558, 634
- OUTS**, instrucción, 635
- OUTSB**, instrucción, 635
- OUTSW**, instrucción, 635
-
- P**
- P6**, familia de procesadores, 38
- Paginación**
definición, 41, 394
directorio de páginas, 394
fallos de página, 42
tablas de páginas, 394
traducción de páginas, 394, 397-398
- Palabras**
arreglo de, 68, 73, 85
comparar, 272
definir, 12
mover, 271
reservadas
 de MASM, 601
 definición, 54
- Pantalla completa**, modo de, 498
- Paralelos**, puertos, 46
- Parámetros**, clasificaciones de, 255-256
parámetro de entrada, 255
- parámetro de entrada-salida, 256
parámetro de salida, 255-256
- Paridad**
bandera, 89, 91, 154-155
registro de 16 bits, 155
- ParseDecimal32**, procedimiento, 114, 119
- ParseInteger32**, procedimiento, 114, 119
- Particiones**, 466-468
extendidas, 466
lógicas, 466
sistema multiinicio, 467
tamaños, 469
- Paso**
de argumentos
 de 8/16 bits, 229-230
 de arreglos, 227
 de varias palabras, 230-231
por referencia, 226
por valor, 226
tipo incorrecto de apuntador, 257
valores intermedios, 257
- del borracho**, programa, 307-310
- PC**
interrupciones, 651-652
programa de sonido, 558-560
- PC-DOS**, 432
- PCI (Interconexión)**
de componentes periféricos, bus, 26, 44
- PedirEntero**, función, 417
- PedirEnteros**, procedimiento, 264, 265
- Pentium 4**, procesador, 38
- Petabytes**, 13
- Peticiones de interrupción (IRQs)**
asignaciones, 550
niveles, 537, 549
- Pila**
marcos de, 225-242
 definición, 225
 después de crear
 variables locales, 234
estructura, 225
función BuscarArreglo, 411
- parámetros de, 225-233
acceso a, 227-228
definición, 225
dirección, 236
eliminar, 228-229
pasos para la creación, 225
programa Factorial, 244-246
registro apuntador de (ESP), 130
tipo de datos abstracto, 129
- Pilas**
afectadas por el operador USES, 232-233
definición, 129
en tiempo de ejecución, 129-131
espacio, reservar, 240
estructura de datos, 129
limpieza de, 228-229
meter enteros en, 130
operaciones, 129-134
registros en, 231
sacar valores de, 131
- Pistas**, 466
- Píxeles**, funciones relacionadas con, 513
- Plano**, modelo de segmentación, 41, 42
- POP**, instrucción, 132, 140, 635
- Pop**, operación, 130-131
- POPA**, instrucción, 132, 635
- POPAD**, instrucción, 133, 635
- POPF**, instrucción, 635
- POPFD**, instrucción, 132, 635
- Positivos**, enteros, 14
- PRE (repetir cadena)**, instrucción, 637
- Precedencia (operadores)**, 53
- Precisión simple**, convertir valores de, 568
- Preferente**, multitareas, 32
- Prefijo de segmento del programa (PSP)**, 544-546
- Preguntas de repaso**, respuestas, 659-704
 Administración de memoria del IA-32, 663-691
 Aplicación: máquinas de estado finito, 674
 Aplicaciones de desplazamiento y rotación, 676

- Aritmética
 ASCII y con decimales
 desempaquetados, 678
 decimal empaquetada,
 678-679
- Arquitectura del procesador
 IA-32, 662-663
- Arreglos bidimensionales, 682
- Asignación dinámica de
 memoria, 690-691
- Búsqueda y ordenamiento de
 arreglos de enteros, 682-683
- Capítulo 1, 649-661
- Capítulo 2, 662-664
- Capítulo 3, 664-666
- Capítulo 4, 666-668
- Capítulo 5, 668-671
- Capítulo 6, 671-675
- Capítulo 7, 675-679
- Capítulo 8, 679-681
- Capítulo 9, 682-683
- Capítulo 10, 683-688
- Capítulo 11, 688-691
- Capítulo 12, 691-693
- Capítulo 13, 693-695
- Capítulo 14, 695-696
- Capítulo 15, 697-700
- Capítulo 16, 700-702
- Capítulo 17, 702-704
- Codificación de instrucciones
 Intel, 704
- Código ensamblador en línea,
 692
- Componentes de una
 microcomputadora IA-32,
 663
- Concepto de Máquina virtual,
 660
- Conceptos básicos, 659-660
- Conceptos generales, 662
- Constantes simbólicas, 666
- Creación de programas
 con varios módulos, 681
- Definición
 de bloques de repetición,
 687-688
 de datos, 665
 de segmentos, 700-701
 y uso de procedimientos,
 670
- Dibujo de gráficos mediante
 INT 10h, 698
- Direccionamiento indirecto,
 668
- Directivas de ensamblado
 condicional, 685-687
- Directorio de disco, 696
- Diseño de programas
 mediante el uso
 de procedimientos, 670-671
- Ejemplo: suma de tres
 enteros, 664-665
- Elementos básicos del
 lenguaje ensamblador
 664
- Enlace a una biblioteca
 externa, 668-669
- Enlazar
 con C/C++ en modo de
 direccionamiento
 real, 693
 con C++ en modo
 protegido, 692-693
- Ensamblar, enlazar y ejecutar
 programas, 665
- Entrada de teclado mediante
 INT 16h, 697
- Escriftura
 de una aplicación gráfica
 de Windows, 689-690
 de un programa en
 tiempo de ejecución,
 701-702
- Estructuras, 683-684
 condicionales, 673-674
- Gráficos de mapas
 de memoria, 698-699
- Instrucciones
 booleanas y
 de comparación,
 671-672
 de ciclo condicional,
 672-673
 de desplazamiento y
 rotación, 675-676
 de multiplicación y
 división, 676-677
 de primitivas de cadena,
 682
 de transferencia de datos,
 666
 JMP y LOOP, 668
- INVOKE, ADDR, PROC y
 PROTO, 681
- La biblioteca de vínculos del
 libro, 669
- Lectura y escritura de sectores
 de disco (7305h), 696
- Llamadas a funciones
 de MS-DOS (INT 21h),
 693-694
- Macros, 684-685
- Manejo de interrupciones, 702
- Marcos de pila, 679-680
- .MODEL, directiva, 681
- MS-DOS y la IBM-PC, 693
- Operaciones
 booleanas, 661
 de la pila, 669-670
- Operadores y directivas
 relacionados con datos,
 667-668
- Procedimientos de cadena
 seleccionados, 682
- Programación
 de video con INT 10h,
 697-698
 del ratón, 699-700
 en la consola Win32,
 688-689
- Recursividad, 680-681
- Representación
 binaria de punto flotante,
 702-703
 de datos, 660-661
- Saltos condicionales, 672
- Servicios estándar de E/S
 de archivos de MS-DOS,
 694-695
- Sistema de entrada-salida,
 663-664
- Sistemas
 de almacenamiento
 en disco, 695
 de archivos, 695-696
 de archivos a nivel
 del sistema, 696
- Suma y resta, 666-667
 extendidas, 677-678
- Unidad de punto flotante,
 703-704
- Preguntas del lector, este libro, 2-5
- Printf, función (C/C++), 420-421
- PROC, directiva, 59, 134-136
 campos de atributos, 205,
 252-253

- especificación de protocolo para paso de parámetros, 252-253
listas de parámetros, 250-252
RET, instrucción modificada por, 252
Sintaxis, 250
- Procedimientos avanzados, 224-268
clasificaciones de parámetros, 255-256
de cadenas, 276-282
de la biblioteca de vínculos, 113-114
de macros, 313-326
de teclado de Irvine32, 357-358
declarar con lista de parámetros separada por comas, 250-252
definición, 134-135
descripciones, 115-125
diagramas de flujo, 140, 141
diseño de programas con, 143-147
documentación, 135-136
externos, llamar a, 258-259
llamadas a, 139-140 anidamiento, 136-138 omisión, 408-409 sobrecarga, 407-409
nombres, ocultar/exportar, 258
paso de argumentos de registros a, 138-139
procedimientos de E/S de archivos, 367-368 prototipos par, 253-255 recursivos, 243
- Proceso, código de retorno de, 438
- Programa, archivos de base de datos de, 64
- Programación a nivel del BIOS, 490-535 consola Win32, 346-379 de video, 498-512 en modo de direccionamiento real, 75-76 en MS-DOS de 16 bits, 432-463 en varios niveles, 47-48 experta en MS-DOS, 536-561
- lenguajes de, 224, 225 MS Windows, 346-401 MS-DOS, 432-463 por turnos (Round-Robin) definición, 32 ilustración, 32 meter enteros en, 130 ratón, 523-533
- Programas COM, 545-546 con varios módulos, 258-267 diseño con procedimientos, 143-147 ejecución de, 386 en lenguaje ensamblador, 465 en modo de pantalla completa, 498 estructura en tiempo de ejecución, 544-548 EXE, 546-547 maestros, 144-145 plantillas, 61 proceso de carga y ejecución de, 31 terminación de, 438-439 TSR, 537, 553-554
- Propósito general, registros de. *Vea también* Registros definición, 34 usos especializados, 35
- Protegido, modo administración de memoria en, 39, 41-43 definición, 33 enlazar con C/C++ en, 409-423 modelo con varios segmentos, 41, 42 modelo plano de segmentación, 41, 42
- PROTO, directiva, 253-255 comprobación de argumentos en tiempo de ensamblado, 254-255 creación de módulos con, 264-266 definición, 253 SumaArreglo, ejemplo, 255
- Prototipos creación, 253-255 de funciones, 415 definición, 253
- PTR, operador con operandos indirectos, 99 definición, 95 DWORD, 96 WORD, 96
- PUBLIC, directiva, 258
- Punto, producto arreglo, 585 cálculo, 582
- Punto flotante instrucciones, 573-576, 644-649 procesamiento, 562-588 FPU, 569-588 representación binaria, 563-569 tipos de datos, 574 valores de comparar, 579-582 escribir, 582-583 leer, 582-583
- PUSH, instrucción, 131-132, 140, 636
- Push, operación, 130
- PUSHA, instrucción, 133, 636
- PUSHAD, instrucción, 132-133, 636
- PUSHD, instrucción, 636
- PUSHFD, instrucción, 132, 636
- PUSHW, instrucción, 636
-
- Q**
- QWORD (definir palabra cuádruple), directiva, 69
-
- R**
- Raíces cuadradas, suma de, 584-585
- Raíz, directorio, 471
- Ralf Brown's Interrupt List, 438
- Random32, procedimiento, 114, 119-120, 429
- Randomize, procedimiento, 114, 120
- RandomRange, procedimiento, 114, 120
- Rango, comprobación de, 85
- Rápida división, 196 multiplicación, 196

- Ratón
apuntador, mostrar/ocultar, 524
botones oprimidos/liberados, 526-527
estado, 524, 525
funciones, 523-528
INT 33h, funciones, 523-528
límites vertical-horizontal, 527
posición, 525-526
programación, 523-533
rastrear programa, 528-532
restablecer, 524
- RCL (rotar acarreo a la izquierda), instrucción, 198, 637
- RCR (rotar acarreo a la derecha), instrucción, 198-199, 637
- ReadChar, procedimiento, 114, 120, 357
- ReadConsole, función, 354-355
- ReadDec, procedimiento, 114, 120-121
- ReadFile, función, 364
- ReadFloat, procedimiento, 582
- ReadFromFile, procedimiento, 114, 121
- ReadHex, procedimiento, 114, 121
- ReadInt, procedimiento, 114, 121
- ReadKey
procedimiento, 114, 121-122, 358
programa de prueba, 358
- ReadSector, procedimiento, 425-427
- ReadString, procedimiento, 114, 122, 453-454
- Reales
binarios, convertir fracciones decimales a, 567-568
codificados, 54, 565, 567
datos, definición, 69
decimales, 53
- Recursividad, 242-246
cálculo de sumas, 243
condición de terminación, 243
definición, 242
factoriales, 243-246
infinita, 242-243
- Redefiniciones de segmentos, 457, 542
- Redirección de la entrada-salida estándar, 114-115
- Redondeo, 571-573
- Referencia, acceso a parámetros de, 234-235
- Registro
de inicio maestro (MBR), 467
índice
de destino extendido, 35
de origen extendido, 35
instrucciones en modo de, 591
parámetros, 225
- Registros
acumulador extendido, 35
apuntador de instrucciones, 35
apuntador de marco extendido, 35
apuntador de pila, 130
extendido, 35
argumentos, pasar a procedimiento, 138-139
comparación, 186
control, 36
de 16 bits, 42, 101
de 32 bits, 42
de propósito general, 34-35
de segmentos, 35
de sistema, 36
depuración, 36
EFLAGS, 35
ejecución, 34-36
específico del modelo, 36
GDTR, 36
guardar, 140-142, 231-232
guardar, Borland C++, 424
IDTR, 36
LDTR, 36
mostrar nombre/contenido, 322, 323
- movimiento inmediato a, 590-591
nombres, 601
restaurar, 140-142, 231-232
tarea, 36
- Relacionales, operadores, 184
- Reloj, 26
ciclos, 26-27, 30
generador de, 44
- Rendimiento
BuscarArreglo, función, 413-414
de miembros de estructura
alineados, 304-305
sincronización, 128
- REPEAT
directiva, 188, 338-339, 340
- Repetición
bloques de, 338-342
prefijos de, 270
- REQ, calificador, 315
- Resta
bandera Acarreo y, 90-91
afectadas por, 89-92
con instrucción
DEC, 87
SUB, 88
extendida, 214
- RET (retornar de procedimiento), instrucción
definición, 136, 638
detalles, 638
ejecución, 137
ejemplo, 136
eliminar, 330
modificada por la directiva PROC, 252
- RETF, instrucción, 638
- RETN, instrucción, 638
- RGB, colores, 520
- ROL (rotar a la izquierda), instrucción, 197, 638
- ROM programable y borrible (EPROM), 45
- ROR (rotar a la derecha), instrucción, 198, 430, 638
- Rotar. *Vea también*
Desplazamiento, operaciones de instrucciones, 197-199
múltiples rotaciones, 197-198
-
- S
- SAHF (almacenar AH en banderas de estado), instrucción, 84, 638
- SAL (desplazamiento aritmético a la izquierda), instrucción, 196-197, 639
- Salida, operando de, 27
- Saltos
con base en los valores de las banderas, 160
condicionales, 158-169
de destino, 159
en modo de 16 bits, 162-163
en modo de 32 bits, 163

- SAR (desplazamiento aritmético a la derecha), instrucción, 639
- SBB (restar con préstamo), instrucción, 194, 214, 639
- SBYTE (definir byte con signo), directiva, 66
- Scanf, función, 421-422
- SCAS (explorar cadena), instrucción, 639
- SCASB (explorar cadena), instrucción, 274, 457, 639
- SCASD (explorar cadena), instrucción, 274, 639
- SCASW (explorar cadena), instrucción, 274, 639
- SDWORD (definir doble palabra con signo), directiva, 68
- Sectores (de disco)
asignación, 470
definición, 466
ilustración, 466
lectura/escritura, 477-482
mostrar, 478-482
traducción de números lógicos, 466
- Secuenciales, búsquedas, 287-288
- SEGMENT, directiva, 539
- Segmentación, 393
- Segmentada, memoria. *Vea también Memoria*
definición, 39
mapa, 40
- Segmento
descriptores de
definición, 393
detalles, 395-397
presente, bandera, 397
tablas de descriptores, 41
- Segments
combinación de, 542-543
de datos, 60, 541
definición, 35, 59, 393
definiciones explícitas, 538-541
definir 55, 537-544
directivas simplificadas, 537-538
límite de segmento, 397
nombres, 403
pila, 60
procedimiento de inicio, 59-60
redefinir, 457, 542
- tipo, 397
variables, identificar, 537
- Selectores de segmentos, 394
- Serials, puertos, 46
- Servicio de interrupciones, rutina de (ISR), 536, 551
- SETcondición, instrucción, 640
- SetConsoleCursorInfo, función, 373
- SetConsoleCursorPosition, función, 373
- SetConsoleTextAttribute, función, 374
- SetConsoleTitle, función, 370
- SetConsoleWindowInfo, función, 371-373
- SetFilePointer, función, 365
- SetLocalTime, función, 376-377
- SetTextColor, procedimiento, 114, 122-123, 415
- SHL (desplazar a la izquierda), instrucción, 195-196, 640
- SHLD (desplazar doble a la izquierda), instrucción, 199-200, 640
- ShowFPStack, procedimiento, 582
- SHR (desplazar a la derecha), instrucción, 196, 640
- SHRD (desplazar doble a la derecha), instrucción, 199-200, 641
- Signo
bandera, 89
de igual, directiva, 72-73
- Simbólicas, constantes, 72-73
- Símbolos
de variables locales, 234
en formatos de instrucciones, 620
exportar, 259
externos, acceder a, 259
lista de, 614-615
- Sin inicializar, declarar datos, 71
- Sin signo
comparaciones, 161, 185-186
- enteros binarios. *Vea también Enteros binarios con signo*
enteros, 10
traducir a decimales, 11
traducir enteros decimales a, 11
- Sistema
hora del establecer, 447
mostrar, 305-306, 447-449
obtener, 306, 447
- registros del, 36
- Sistemas de archivos
clústeres, 468-469
- FAT12, 469
- FAT16, 169
- FAT32, 469-470
- NTFS, 470
- operativos, 8
administradores de memoria virtual, 41
API, 464
funciones, 47
multitareas, 32
- Sitio Web de este libro, 3
- SIZE, operador, 405-406
- SIZEOF, operador, 97
- Sleep, función, 377-378
- SmallWin.inc, 348-350
- Software
interrupciones de, 435
requerimientos, este libro, 2
- Sólo lectura, memoria de (ROM), 45
- STACK, directiva, 436
- STC (activar bandera Acarreo), instrucción, 641
- STD (activar bandera Dirección), instrucción, 641
- STDCALL, convención de llamadas, 229, 247-248, 263
- STI (activar bandera Interrupción), instrucción, 550, 641
- STOS (almacenar datos de cadena), instrucción, 642
- STOSB (almacenar datos de cadena), instrucción, 274-275, 642
- STOSD (almacenar datos de cadena), instrucción, 274-275, 642
- STOSW (almacenar datos de cadena), instrucción, 274-275, 642
- Str_compare, procedimiento, 276-277

Str_copy, procedimiento, 278
 Str_length, procedimiento, 277
 Str_trim, procedimiento, 278-279
 Str_uppercase, procedimiento, 279-280
 StrLength, procedimiento, 114, 123
 SUB, instrucción, 59, 88, 642
 Subdirectorios. *Vea también*
 Directorios de disco
 crear, 458-486
 definición, 472
 eliminar, 486
 Subíndices, 414
 Subrutinas, 134
 Suma
 bandera Acarreo y, 90
 banderas afectadas por, 89-92
 con la instrucción ADD, 87-88
 con la instrucción INC, 87
 de enteros, programa, 143-147
 enteros de 32 bits, 100
 extendida, 213-214
 Suma_Extendida, procedimiento, 213-214
 SumaArreglo
 procedimiento, 253, 255, 261
 programa, 260
 SumarDos, procedimiento, 248
 SumaResta, programa, 60, 70-71
 SumaResta2, programa, 76
 SumaResta3, programa, 92-93
 Superescalar, arquitectura
 canalizaciones de ejecución, 29, 30
 definición, 29
 varias instrucciones, 30
 Sustitución, operador (&), 333
 SWORD (definir palabra con signo), directiva, 67
 SystemTimeToFileTime, función, 378

T

Tabla de asignación de archivos (FAT), 471, 476-477
 Tareas
 comutación de, 32
 registro, 36
 Tarjeta madre
 componentes, 43-44
 conjunto de chips, 44
 definición, 43

TASM (Turbo Assembler), 1, 9
 TBYTE (definir diez bytes)
 directiva, 69
 Teclado
 banderas, 495, 496
 búfer de, 493, 494-497
 borrar, 495-497
 comprobar, 494-495
 meter teclas en, 493
 byte de estado, 554
 caracteres, constantes
 simbólicas para, 72-73
 entrada, 492-498
 estado, obtener, 359
 procedimientos en la biblioteca Irvine32, 357-358
 secuencia de procesamiento de tecleos, 492
 velocidad de repetición de teclas, 493
 Terabytes (TB), 13
 Terminales, estados, 179
 Terminar y permanecer residente (TSR), programas, 537, 553-554
 activación, 553
 ejemplo de teclado, 553-554
 TEST, instrucción, 155-156
 banderas, 1563
 definición, 151, 642
 detalles, 642
 ejemplo de uso, 155
 TEXTEQU, directiva, 74-75
 Texto
 archivos, leer/copiar, 454-456
 escribir en ventanas, 505-506
 operador literal(<>), 335
 video, 498-499
 Tiempo de ejecución
 operadores en, 618
 pilas en. *Vea también* Pilas
 aplicaciones, 131
 definición, 129
 sacar valores de, 131
 TraducirBufer, función, 406, 407, 408
 Transferencia de control, 104
 TYPE, operador, 96-97, 102, 405-406
 TYPEDEF, operador, 102-103

U

UEPS (último en entrar, primero en salir), 129, 133
 Un solo byte, instrucciones de, 589-590
 Un solo carácter, entrada de, 357-358
 Unicode, estándar, 17
 Unidad
 aritmética-lógica (ALU), 26
 central de procesamiento (CPU)
 acceso a memoria, 96
 banderas, 151-157
 definición, 26, 43
 ilustración, 28
 zócalo, 43
 de almacenamiento de memoria, 26
 de interfaz de bus (BIU), 28
 de punto flotante (FPU), 36-37, 44, 569-588
 definición, 569
 ejemplos de código, 584-585
 excepciones, 573, 586-587
 instrucciones aritméticas, 576-579
 palabra de control, 572, 587
 pila
 de expresiones, 569-570
 de registros, 569-571
 redondeo, 571-573
 registros de datos, 570-571
 registros de propósito especial, 571
 sincronización de excepciones, 583-584
 Uniones
 declaración, 310
 estructuras que contienen, 311
 uso de, 310-312
 variables de, 311-312
 USES, operador, 140, 232

V

Variables
 alineación, 95
 de estructuras, 301-305

- de segmento, identificar, 537
de uniones, 311-312
exportar, 259
externas, acceso a, 259
locales, 233-236
mostrar
 contenido, 320
 nombre/contenido, 322
sumar, 70-71
- Ventanas
 definición, 504
 definir, 504
 desplazarse hacia abajo, 506
 desplazarse hacia arriba, 504-505
 escribir texto en, 505-506
- Video
 control de colores, 499-511
 de texto, 498-499
 establecer modo de, 502
 funciones, 501
 información sobre el modo de, 508
 modos gráficos, 512
 niveles de acceso, 498
 programación, 498-512
 salida, 44-45
- Video RAM (VRAM), 45, 433
- Vínculos, bibliotecas de
 definición, 112
 dinámicos, 112
 procedimientos, 113-114
- Virtual-386, modo, 33, 39
- Volúmenes. *Vea también Particiones*
-
- W**
- WAIT, instrucción, 642
WaitMsg, procedimiento, 114, 123
WHILE
 ciclos, 174-176, 188
 directiva, 188, 338
- Win32
 consola
 acceso de alto nivel, 348
 acceso de bajo nivel, 348
 antecedentes, 347-350
 control de los colores del texto, 373-375
 control del cursor, 373
 entrada, 354-359
 información de referencia, 347-348
 manejadores, 350
 programación, 346-379
 salida, 360-361
 funciones, 350-352
 entrada de consola, 354-359
 fecha y hora, 375-379
 leer/escribir archivos, 361-370
 manipulación de la ventana de consola, 370-373
 mostrar cuadro de mensajes, 352-354
 referencia rápida, 351-352
 salida de consola, 360-361
 tipos de datos, 348
 ventana de consola
 definición, 114
 manipulación, 370-373
 salida escrita a, 114
- WinMain, procedimiento, 382
WinProc, procedimiento, 382
WORD (definir palabra), directiva, 67
WriteBin, procedimiento, 114, 123
WriteBinB, procedimiento, 114, 123
WriteChar, procedimiento, 114, 123, 313
WriteConsole, función, 360
WriteConsoleOutputCharacter, función, 361
- WriteDec, procedimiento, 114, 123
WriteFile, función, 365
WriteFloat, procedimiento, 582
WriteHex, procedimiento, 114, 123-124
WriteHex64, procedimiento, 230-231
WriteHexB, procedimiento, 114, 124
WriteInt, procedimiento, 114, 124
WriteStackFrame, procedimiento, 240-241
WriteString, procedimiento, 114, 124, 360, 454
WriteToFile, procedimiento, 114, 124
WriteWindowsMsg, procedimiento, 114, 124, 357
-
- X**
- XADD, instrucción, 643
XCHG (intercambiar datos), instrucción, 84, 643
Xeon, procesador, 38
XLAT (traducir byte), instrucción, 643
XLATB (traducir byte), instrucción, 643
XOR, instrucción, 154-155
banderas, 154
 de Paridad y, 154-155
combinaciones de operandos, 154
definición, 151, 643
detalles, 643
-
- Y**
- Yottabytes, 13
-
- Z**
- Zero, bandera, 89
Zettabytes, 13

decimal	➡	0	16	32	48	64	80	96	112
↓	hexa-decimal	0	1	2	3	4	5	6	7
0	0	nulo	►	espacio	0	@	P	~	p
1	1	☺	◀	!	1	A	Q	a	q
2	2	☻	❖	"	2	B	R	b	r
3	3	♥	!!	#	3	C	S	c	s
4	4	♦	Π	\$	4	D	T	d	t
5	5	♣	§	%	5	E	U	e	u
6	6	♠	■	&	6	F	V	f	v
7	7	•	↕	'	7	G	W	g	w
8	8	●	^	(8	H	X	h	x
9	9	○	↓)	9	I	Y	i	y
10	A	◎	→	*	:	J	Z	j	z
11	B	♂	←	+	;	K	[k	{
12	C	♀	⟲	,	<	L	\	l	
13	D	♪	↔	-	=	M]	m	}
14	E	♪	▲	.	>	N	^	n	~
15	F	☀	▼	/	?	O	_	o	Δ